Practical Aspects of Interacting Garbage Collectors

by

Yannis Chicha

Graduate Program in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
July, 2002

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Chief Advisor	Examining Board
_____	_____
Advisory Committee	
	_____
_____	_____
_____	_____

The thesis by
Yannis Chicha
entitled

Practical Aspects of Interacting Garbage Collectors

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

_____	_____
Date	Chair of Examining Board

Abstract

In this thesis we investigate a novel approach to creating tools for the study and implementation of garbage collectors (GC) in contexts ranging from hand-held computers to the Web. We define interactions between GC entities and the global strategy that uses them, e.g. distributed garbage collection (DGC) algorithms. In this setting, we define the notion of a *generic collector* to be the representation of the ideal collection entity from the global strategy point of view. This definition helps create several tools to improve the way people work with uniprocessor and distributed garbage collectors.

This work has led us to consider a new heap organization for uniprocessor GCs. We have designed a *Localized Tracing Scheme* to optimize GC tracing process at caching and paging levels with possible applications to platforms with limited resources such as hand-held computers. We provide experimental results and show how this organization naturally applies to a parallel setting.

To help implement DGCs, we propose a *design method* based on generic collectors. This method uses models to list important characteristics of each entity in a system. Our methodology explains how to use these models to create local collectors adapted to a particular DGC. This process also allows the design of heterogeneous systems, where each node can choose its own GC.

Finally, we use this method to design and implement a garbage collection mechanism for the *World Wide Web*. After exploring the mapping of memory management paradigms onto concepts of the Web environment, we show that DGC algorithms can be used to detect garbage in websites and avoid dangling links on webpages. We observe that web authors are currently managing web

objects explicitly, and an automatic solution should be beneficial. We report on practical implementations and experiments in this web setting. This study leads to the creation of a Web-based platform for experiments in garbage collection research.

Keywords: Garbage Collection, Distributed Garbage Collection, World Wide Web.

Dedication

I would like to dedicate this thesis to Chantal Chicha, my mother, who gave me life, love, support and who taught me to always try and do my best. Thanks Mom.

Acknowledgments

First of all, I would like to thank my supervisor, Dr Stephen Watt, for his support, advice and precious guidance all along this thesis. During these five years, Dr Watt allowed me to work on many interesting topics such as Aldor and the FRISCO project. This helped me discover and appreciate many aspects of research and administrative work. Thank you for this invaluable experience.

Thank you to my advisors, Hanan Lutfiyya and Mark Giesbrecht, who helped me on a number of occasions with documents, discussions, and useful advice. Thanks to people in the SCL for many great discussions and advices. Thanks to Cosmin, Laurentiu, Jason and Florence for proof-reading this thesis. I would also like to thank Richard Jones for his very useful Garbage Collection bibliography. Thanks to Angie, Bethany, Cheryl, Dianne, Janice, and Sandra for your help and kindness.

Finally, many thanks to Florence Defaix for helping and supporting me during this thesis. Her love and moral support mean everything to me, and gave me a lot of strength. Thanks Flo, I could not have done it without you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction, Motivation and Direction

Automatic memory management gains in popularity and many developers now consider this feature as essential.

Modern languages, such as Java [58] or Aldor [48], feature garbage collectors to handle this activity. In a parallel or distributed context, explicit allocation and deallocation are a true challenge that shifts the attention of programmers from algorithms to memory maintenance. Automatic memory management thus proves necessary in these environments. Java RMIs [59] use a Distributed Reference Counting algorithm to detect distributed garbage. Currently, such a garbage collection scheme is sufficient because distributed applications rely mostly on a client-server model. In this architecture, reclaiming only *most* distributed garbage (as opposed to *all* of it) does not usually hamper the execution of the programs. We expect, however, applications to become truly distributed over time, and for network sites to communicate with others directly without relying on any hierarchical organization. Peer-to-peer architectures gain in popularity and loosely coupled networks, such as the World Wide Web, become ubiquitous. Although the Internet has a hierarchical organization to route messages, the application level

does not rely on this structure to function properly. Two-way communication and data exchange between two network sites have the unfortunate consequence that many distributed cycles of objects can be created. This implies that developers will soon need to rely on distributed garbage collectors (DGCs) capable of detecting distributed garbage cycles. Several sophisticated algorithms already exist, but very few implementations have been realized so far. A possible reason is the difficulties and challenges posed by the design and implementation of DGCs. Very few support tools are available to help with this activity, and many issues have to be faced.

In this thesis, we propose an approach that helps create support tools for various garbage collection environments including DGCs. To achieve this result, we observe and define interactions between GC entities and the global strategies that link them together. As we will see, such entities may be local collectors for a DGC, but also specific parts of a uniprocessor algorithm. This results in the creation of a novel entity: the *Generic GC*, which builds a clear and explicit bridge between GC entities and their global strategy. We use this element to implement an optimized tracing scheme – called *Localized Tracing Scheme* (LTS) – for uniprocessor GCs. Around the notion of Generic collector, we also construct a design method to help implement DGCs and, in particular, we introduce the idea of *GC interoperability* which allows different GC techniques to work together with a single global strategy. Finally, we propose the first work on garbage collecting objects in the *World Wide Web* environment. This web management tool, that we call *W3GC*, can be used both to maintain link integrity on websites and to experiment with DGCs in simple and efficient manner. We show the design and implementation of two uniprocessor GCs (Mark-and-Sweep and Generational) and two DGCs (distributed reference counting and "Garbage Collecting the World" [51]). Finally, we analyze the results of our empirical experiments about the structure of websites in four different contexts.

1.1 Thesis outline

The rest of this chapter presents our motivations in more detail and introduces the various research directions of our work. The thesis itself is organized as follows. Chapter 2 provides a detailed background on garbage collection and present our study of interactions between garbage collection entities and their overall strategy. Chapter 3 introduces the Generic Garbage Collector that we created to help us understand and work with distributed collectors. Chapter 4 reports our research about an optimized tracing scheme for uniprocessor collectors. Its design is a direct consequence of our study on interactions. In Chapter 5, we use the generic garbage collector to create models and a method to facilitate the design of homogeneous, as well as heterogeneous, distributed garbage collection environments. We also introduce the notion of interoperability of garbage collectors in a DGC context. Chapter 6 presents W3GC, a distributed garbage collection mechanism for the World Wide Web. Chapter 7 describes how W3GC was implemented using our design method and models, and allows one to detect dangling links as well as unlinked web pages in websites or a group of websites. Chapter 8 is preliminary study of an extension to this work. We claim that the Web is a convenient platform for the development of an experimentation platform destined to support research in garbage collection. Finally, Chapter 9 concludes the thesis and presents future directions.

1.2 Motivation

We explain our motivation for this thesis by discussing important aspects of distributed garbage collection. We have identified these from the literature study presented in Chapter 2. In particular, we emphasize the need for various software and theoretical tools.

1.2.1 Difficult implementation

One of the main challenges in the field of distributed garbage collection is implementation. Very few – about half a dozen – distributed cyclic collectors have actually been implemented so far in spite of the existence of several algorithms. There are several reasons for this state of affairs:

- **Nature of algorithms**. Currently, distributed GCs exist mostly in research environments. A reason for the lack of usable implementations lies in the nature of the algorithms. Most of them were created either as a purely algorithmic solution or as a module for a very specific application. It appears however that certain algorithms could be adapted to many situations, if proper support were available.

- **Difficult design and adaptation for a given system**. Most implementers need to integrate a DGC within an already working framework. It would not be efficient to redesign the entire environment from scratch simply to integrate a DGC. However, the lack of support tools makes this a complicated task. We provide more details in Section 1.2.2.

- **Practical implementation barriers.** Activities such as *testing* and *debugging* DGCs usually prove difficult due to the complexity of most algorithms. Furthermore, it is difficult to *evaluate the resources* to allocate to writing a DGC. Finally, distributed garbage collection is a complex problem leading to complex algorithms, understanding their *behavior* and *comparing* them can be quite challenging.

1.2.2 Challenging design

The design of distributed garbage collectors reveals many challenges. Obviously, a difficult design is one the main reasons for a small number of implementations.

We list here choices and challenges one faces when designing a DGC:

- There are a large number of parameters: choice of local collectors, constraints of the underlying system, need for future evolution, and so on.
- There is no means to evaluate the suitability of a DGC for specific needs.
    1. No complexity analysis of DGCs.
    2. No estimate of the feasibility of a DGC implementation under the constraints of an environment.
    3. Difficult identification of important characteristics in a DGC (features, assumptions, needs, etc.)
- There is no support to manage heterogeneous systems (i.e. using different local GCs) in a simple and flexible manner. In addition, to our knowledge, there exists no support tool to easily analyze the existing memory management situation on a distributed system. This has two consequences:
    1. If we start with an existing pool of local GCs, it is difficult to know if the collectors are going to work with the DGC. It is equally challenging to understand what is required to adapt them to the DGC.
    2. If a DGC has to be replaced by another on a given system, it is difficult to identify potential problems, such as supporting algorithms (distributed termination detection for example), adapting local GCs, and so on.

1.2.3 Lack of tools

In the study of the situation presented above, we observe that there is a need for support tools both at a theoretical and at a practical level. Most DGCs are still in the form of algorithms or prototypes for experimentation purpose, and

the transition has to be made to generally available implementations. This can happen only if the GC community provides the means to study the behavior of distributed collectors, debug their implementations, teach their algorithms, and classify each aspect of a distributed collector. In the following paragraphs, we discuss a number of tools that could prove useful to attain this goal.

To support practical implementation, we need *debugging*, *testing*, and *profiling* tools. Particularly, we observe that **traces** are required to understand and to study the behaviors of allocators and collectors. Such traces already exist in certain environments. For example, in a uniprocessor context, Chilimbi, Jones and Zorn [20] propose the use of a language called MetaTF to efficiently format traces for allocation events. Tools to exploit such traces are currently being designed (see [61] for more details). On the distributed side, we can cite the PerDis [77] project, that uses a distributed collector, and traces within a memory simulator to experiment with diverse allocation strategies.

In terms of testing, fair comparisons between garbage collection techniques must use *standard benchmarks*. There exists a number of these (for example, GC benchmarks for Scheme implementations can be found at [21]) for uniprocessor GCs. Unfortunately, this does not seem to be the case for distributed garbage collection. A reason is that few applications currently require complex DGC mechanisms. On a practical note, we believe that *repositories* of DGCs implementations and/or algorithms would prove beneficial. These could be accompanied by users' evaluations, descriptions of integration work, listings of characteristics, known benchmarks, and so on.

Although practical tools are required to support the implementation of distributed collectors in application contexts, theoretical tools can also prove beneficial in a number of situations. For example, little work has been done to create a general mechanism to *prove properties* of distributed garbage collection algorithms, and provide formal representations to study them. Evaluating collectors

according to chosen criteria such as performance or security proves difficult as well, because no support is provided.

Choosing a DGC for implementation should be done with respect to diverse analyses conducted beforehand (for example, a method to analyze the complexity of a technique). In order to implement a DGC, we also need methods to *analyze* its characteristics and organize its different components. Such an analysis would also be useful in studying the behavior of collectors. Subsequent tools would include integration methods in preexisting environments with a set of local GCs and/or other DGCs, for example.

1.3 Directions for this thesis

This thesis provides solutions to a number of needs described in the previous section. We present an approach based on the study of interactions between local GCs and DGCs, using an abstract view of their relations. This leads to different tools supporting design and implementation of DGCs as well as experiments into their behaviors.

1.3.1 Interactions in garbage collection

First, we propose a study of interactions between garbage collection entities and the overall strategy that ties them together. We observe different configurations in various environments: uniprocessor, multiprocessor, and distributed. Although traditional uniprocessor collectors are monolithic, certain techniques are organized differently: generational garbage collection schemes use two or more logical GCs, CMM [3] divides the heap according to memory management needs, and so on. Multiprocessor collectors are divided into two categories: parallel and concurrent. Concurrent GCs, where a processor is dedicated to collect garbage *while* mutation is running, do not propose an architecture to study this particular

question. Parallel GCs, however, relate several entities and an overall strategy. Finally, distributed GCs contain several entities and organize their cooperation. We focus on the relationships between the DGC algorithm and each of the local collectors.

In a DGC environment, we identify these interactions, and deduct a model to classify them. We introduce the notion of *Generic Garbage Collector*, which is defined as a template to list the characteristics necessary to a local GC to support the algorithm of a given DGC.

1.3.2 Localized Tracing Scheme

We use our study of interactions between GC entities and a general strategy to investigate the tracing process of uniprocessor GCs. We present a technique to visit all nodes in a forest of data structures that takes into account locality of reference to improve traffic in the memory hierarchy. This method is applicable to a wide range of garbage collection algorithms for a uniprocessor, and generalizes naturally to a multiprocessor setting. We call this technique the "*Localized Tracing Scheme*" as it improves locality of reference during the object tracing activity.

1.3.3 DGC design and implementation

One obstacle to promoting the use of distributed garbage collection is the small number of usable implementations. We suspect the reason for this is the complexity of design. Implementing a DGC is difficult because there are many issues to take into account, and no support method exists to help identify and classify them. We use our study of interactions between local GCs and DGCs to propose a solution for such design questions.

The basis of our work is the semantic separation between local GCs and stand-alone GCs. A stand-alone collector is a GC that has no concept of a distributed

system, and does not consider remote pointers. A local collector is both a collector for a specific node and a representative of the DGC algorithm at this specific node. Using the notion of generic garbage collection we developed earlier, we explore a design method to transform a stand-alone GC into a local GC. This has the interesting property of allowing a natural design of heterogeneous systems, where several types of local collectors participate in a single distributed garbage collection.

1.3.4 Garbage Collecting the Web

In order to test our design method, we choose to study distributed garbage collection in a non-trivial context, namely the World Wide Web. This environment provides a challenging and interesting platform to work with. Although it may not be obvious at first, we can view the Web as a very large distributed memory. Each site serves pages with information and references to other sites or pages roughly in the same way objects and pointers are manipulated in a primary memory environment. We explore the use of garbage collection in this context by designing and implementing a distributed garbage collector called "Garbage Collecting the World" (see [51] for details). Beyond simply testing our design method, we believe that this application can be a powerful tool for web management.

1.3.5 W3GC: a platform for experiments

Understanding the behavior of distributed collectors is difficult and little support exists for experimentations. Furthermore, finding applications to test and study DGCs proves to be a difficult task. Our experience with implementing garbage collectors for the Web resulted in the observation that this is a convenient platform for distributed garbage collection research. We propose a preliminary design of a Web-based experimentation platform to support behavior study, fair comparison and implementation experimentation for distributed garbage collectors.

Chapter 2

Garbage Collection and Interactions

In this chapter, we present an overview of garbage collectors as well as a novel point of view to describe them. We explain widely used techniques and show specific algorithms for uniprocessor (Section 2.3), multiprocessor (Section 2.4) and distributed (Section 2.5) garbage collection. Further details about these algorithms and presentations of other ones may be found in other surveys, including Jones' book [41] on Garbage Collection. Up to date references to papers can be found at Baker's GC webpage [4], Jones' GC webpage [42] and Harlequin's GC reference page (now hosted by Ravenbrook) [78]. Please note that a glossary is provided in Appendix A.

In order to prepare the reader for the discussion of this thesis, we also examine the architecture of selected algorithms from a novel point of view. These algorithms use several entities cooperating according to the rules of a global strategy. Our presentation lists those entities and explores their interactions with the global strategy.

2.1 Generalities

Before describing GC techniques, we start with a brief reminder of what garbage collection represents. Garbage collection is about memory management. It is essentially a technique to automatically find out and reclaim unused objects. Any program needs a way to store and retrieve data that is created, manipulated, and modified for the purpose of an application. Most environments use the concept of dynamically allocated data. The **heap** is a logical space where all data necessary to the execution of a program is stored. A heap mainly uses RAM, but can also use virtual memory on secondary storage when primary storage fills up.

Managing the data or objects stored in the heap can become quite complicated when done manually. A garbage collector is run regularly to find out what objects are no longer needed (because they are no longer referenced by the program). Relying on an automatic tool also results in the disappearance of unreliable pointers to objects. If programmers are allowed to manage data in the heap, it is likely that certain objects will be forgotten leading to memory leaks and certain others will be erased even when the program is still able to access them. This can get worse if a new object is allocated in the heap at the exact location of a previous object. If the program accesses this new object believing it is the old one, this may have disastrous consequences. Another advantage of garbage collection is to help gain abstraction on the programming process. When details of memory management are hidden, programmers have more time to concentrate on the logic of the program, rather than on details for maintaining it. This results in cleaner code, stripped from all undesirable primitives used for memory management.

Although garbage collection has existed since 1960 (see [55]), and was used for example in Lisp and Smalltalk, it never made its way into the most popular languages before Java. Well known languages such as C or C++ have typically used **explicit** memory management. The programmer has access to some primitives to allocate and deallocate memory (e.g malloc and free in C). It is the

programmer's responsibility to free the memory when there is no longer a link to an object, leading to possible problems as explained earlier. Although garbage collection removes the burden of explicit memory management, many programmers did (and still do) not like this tool for two main reasons. The first one is that, in the early days, garbage collection algorithms were slow (20% to 40% overhead), with uncontrolled starting times. C programmers are usually focused on performance (e.g. in system programming), and unpredictable, time-consuming memory management techniques were clearly not acceptable. We observe that this is no longer the case, and except for some very specific fields of application, garbage collection is efficient enough and can be controlled to avoid unpredictable pauses. The second reason is psychological: a number of programmers like allocating the memory they need "themselves". Some process dealing with "their" memory behind their back raises suspicion.

In this chapter, we first describe garbage collection algorithms in a uniprocessor setting. This is currently the most widely used environment for GC. We then move to concurrent and parallel GC, a topic born in 1975 with Steele's algorithm [84]. Nowadays distributed systems are more common and distributed collectors are becoming of more interest. As we are going to see, many issues arise with distributed GC making it a complex field to study and to implement.

2.2 Acronym definition

- **ACK**: Acknowledgment. Message found in most network protocols.
- **AMD**: Advanced Micro Devices.
- **API**: Application Programming Interface.
- **BDW garbage collector**: Boehm, Demers, Weiser garbage collector. Widely used implementation of a uniprocessor GC for C and C++. See [13].

- **BP**: Back Pointer.
- **CCS**: Calculus of Communicating Systems. See [60].
- **CMM**: Customizable Memory Management.
- **CORBA**: Common Request Object Broker Architecture. See [70].
- **CGI**: Common Gateway Interface. Standard for the environment passed by a server to scripts used in the WWW.
- **CSD**: Computer Science Department.
- **CSSPC**: Cyclic version of SSPC. See [52].
- **DGC**: Distributed Garbage Collection.
- **DMOS**: Distributed Mature Object Space. See [37].
- **DRC**: Distributed Reference Counting. See Section 2.5.2.
- **DRL**: Distributed Reference Listing. Variant of DRC. See Section 2.5.3.
- **DTD**: Distributed Termination Detection. Process used to decide whether a global state corresponds to one in which a distributed computation has terminated.
- **FRISCO**: FRamework for Integrated Symbolic/numeric COmputation. European ESPRIT project. See [69].
- **GC**: Garbage Collection.
- **GGC**: Generic Garbage Collector.
- **GIF**: Graphics Interchange Format.
- **GCW**: Garbage Collecting the World. See [51].

- **HDRC**: Hierarchical Distributed Reference Counting. See [63].
- **HP**: Hewlett-Packard.
- **HREF**: Hypertext Reference.
- **HTML**: HyperText Markup Language. Standard web page description language.
- **HTTP**: HyperText Transfer Protocol. Standard network protocol on the World Wide Web.
- **HTTPS**: HTTP over SSL.
- **IBM**: International Business Machines.
- **ID**: Identifier.
- **IP**: Internet Protocol.
- **IMG**: Image.
- **INRIA**: Institut National de Recherche en Informatique et en Automatique.
- **JDK**: Java Development Kit. See [58].
- **JPEG**: Joint Photographic Experts Group.
- **KB**: KiloByte.
- **LGC**: Local Garbage Collector.
- **LiLo**: Linux Loader. Small program used by Linux systems to boot the system. In particular, it offers the possibility to choose which partition of a hard drive to boot.

- **LISP**: LISt Processor. Language featuring the first garbage collector. See [55].
- **LMS**: Localised Mark-and-Sweep. Mark-and-Sweep using the LTS. See SectionB.4.
- **LTS**: Localised Tracing Scheme. See Chapter 4.
- **MB**: MegaByte.
- **M&C** or **MC**: Mark-and-Copy. Also called Stop-and-Copy. See Section 2.3.3.
- **MHz**: MegaHertz.
- **MIME**: Multipurpose Internet Mail Extension.
- **MOS**: Mature Object Space. See [38].
- **M&S** or **MS**: Mark-and-Sweep. See Section 2.3.2.
- **NFS**: Networked File System.
- **OMT**: Object Modeling Technique. See [80].
- **OO**: Object-Oriented. Design and programming style using objects to describe an environment.
- **OS**: Operating System.
- **PDF**: Portable Document Format.
- **PHP**: PHP: Hypertext Preprocessor (this is a recursive acronym). This is a scripting language that is usually embedded in HTML files to allow web authors to provide dynamically generated pages. Connections to databases are supported.

- **POPL**: Principles Of Programming Languages. Annual ACM conference on programming languages.
- **RAM**: Random Access Memory.
- **RC**: Reference Counting. See Section 2.3.1.
- **RDFPCC**: Rooted Depth First Partial Cycle Count. See 6.5.
- **REF**: Reference.
- **RISC**: Reduced Instruction Set Computer.
- **RMI**: Remote Method Invocation. This is a high-level Java library for distributed programming. See [59].
- **SCL**: Symbolic Computation Laboratory.
- **SDK**: Software Development Kit.
- **SSL**: Secure Socket Layer. This is a security protocol for TCP/IP. It is widely used on the Web.
- **SSPC**: Stub-Scion Pair Chain. See [81].
- **TCP**: Transmission Control Protocol.
- **UDMA**: Ultra Direct Memory Access. This is a norm for hard disk access speeds and can be found in various models: UDMA33, UDMA66, UDMA100, and so on.
- **UML**: Unified Modeling Language. Object-oriented design method. See [14].
- **URI**: Universal Resource Identifier.
- **URL**: Uniform Resource Locator.

- **UWO**: University of Western Ontario.
- **W3C**: World Wide Web Consortium.
- **W3GC**: World Wide Web Garbage Collector. See Chapter 6.
- **WWW**: World Wide Web.
- **XHTML**: XML version of HTML.
- **XML**: eXtensible Markup Language.

2.3 Uniprocessor GC

This section details techniques used in uniprocessor garbage collection. The term "uniprocessor" is used to differentiate these algorithms from the concurrent, parallel and distributed ones. Unlike these other techniques, a uniprocessor GC only deals with *one* process managing *one* heap.

Historically, the first garbage collector was designed for a programming language called LISP. In 1960, J. McCarthy published a paper [55] where he explained the details of this language including a good half page on a technique to automatically reclaim dead memory cells. This technique would be later known as Mark-and-Sweep.

We present here five important techniques in garbage collection: reference counting, mark-and-sweep, mark-and-copy, generational, and, finally, incremental. For a complete overview, please refer to Paul Wilson's survey [91].

2.3.1 Reference Counting

The idea behind reference counting is fairly simple. With each object, we associate a counter of pointers. At the creation of the object, the counter is 1 (one pointer is returned from the allocation function). When a pointer to the object is created,

the counter is increased. If a pointer to the object is deleted the counter is decreased. At this time, if the counter reaches the value of 0, the object can safely be reclaimed as it is no longer referenced.

The advantage of this technique is that memory is reclaimed as soon as it is no longer needed. However, one can also identify three main problems. First, the extra counter incurs a non-negligible space overhead. Second, circular structures are not dealt with. Imagine a cell which contains data and a pointer to itself. Its counter never reaches zero and, thus, the object is never recycled. Third, pointer assignments can no longer be simple register operations. Each assignment must read and write memory for the reference counts.

2.3.2 Mark & Sweep

Mark & Sweep is the first garbage collection algorithm in history. In 1960, McCarthy described this technique in a paper about the LISP language [55].

We consider the directed graph of objects $G = (V, E)$. Each vertex $v \in V$ represents an object, and each edge $e \in E$ represents a pointer ($e.origin \in V$ and $e.dest \in V$). $R \subset V$ is a special subset of vertices that we call **roots**. Usually, the roots contain all references in the stack (for example, local variables), the processor registers and the static area (global variables). For convenience, a color is associated with each vertex and can be accessed via $v.color$. We use two colors: black and white. Black means "marked" and white means "not marked yet". The algorithm for the *mark phase* proceeds as follows:

- $\forall v \in V, v.color :=$ **white**.
- $\forall v \in R, v.color :=$ **black**.

- Repeat

  $\forall e \in E \ s.t \ e.origin.color =$ **black**, and $e.dest.color =$ **white**,

  $e.dest.color :=$ **black**.

  until $\nexists e \in E \ s.t \ e.origin.color =$ **black** and $e.dest.color \neq$ **black**.

At the end of this phase, all vertices that are still colored in white are known as *garbage* because they are not *reachable* from the roots. This means that there is no path between any vertex in R and these vertices. In computer programming terms, garbage objects are no longer in use, because there is no way for the program to access those objects. The only way to access an object is through the roots (stack, registers, static area).

Garbage objects can be recycled or *swept*. This is the *sweep* phase:

- $\forall v \in V \ s.t \ v.color =$ **white**, add v to the freelist and remove v from V

The freelist links together all free space that is allocated to a program by the OS. It is used by the allocator to try and get memory space for an object before asking the operating system for extra memory. The freelist is managed by the program, not by the OS.

2.3.3 Mark & Copy

We describe here the simple but space-inefficient technique of "semi-spaces". The heap H is divided into two equal subheaps H_A and H_B. We are now going to use aliases to name these subheaps: H_{from} will refer to H_A and H_{to} refers to H_B. Although, at this moment, it may seem unclear that these aliases are needed, the following explanation will (hopefully) clarify this.

Object allocations and mutations are only done in H_{from}, while H_{to} is considered empty. When the collector runs, it considers the directed graph of objects $G = (V, E) \in H_{from}$. As for the Mark&Sweep (M&S) algorithm, we follow all

possible paths from the roots. Unlike M&S, we do not mark (or blacken) encountered objects (or vertices). Instead, they are copied to H_{to}. This part is also called *Compact* because objects are likely to be scattered in H_{from}. When objects are moved to H_{to}, space is allocated *sequentially* thus allocating contiguous spaces for related objects (linked via pointers). Once live objects have been moved, the roles of H_A and H_B are reversed. H_{from} refers now to H_B and H_{to} refers to H_A. After the next GC, H_{from} will refer again to H_A and H_{to} to H_B.

One of the main issues with Mark&Copy occurs when two or more objects X_i point to one object O. Assuming O is reachable, the first time the collector encounters it, O will be moved to H_{to}. Because O is referred to by several objects, the collector will reach the old location of O through another path. If no special action is taken, incorrect behavior occurs. For example, the contents of the object is copied again to H_{to}. The solution is to set a "forwarded" flag in the header of the object, and put a forwarding pointer at O's old location. This allows the collector to know that O has been moved and where. To know where is important, because the pointer to O in each X_i object has to be updated. For information on more efficient Mark&Copy algorithms, please refer to [91].

Mark&Copy has two advantages compared to Mark&Sweep. First, there is no sweep phase, this avoids looking at ALL the objects in the heap to figure out which ones should be recycled. Indeed, time complexity with a Mark&Copy collector depends on the number of live objects, not on the *total* number of objects. The second advantage is compaction. M&C offers better locality of objects, which means that the mutator is likely to run faster, avoiding cache misses and page faults. Locality is also good for the collector, because cache misses and page faults will be avoided in the trace phase of subsequent GCs. However, Mark&Copy has the drawback of moving objects. This is a costly procedure, which can prove very inefficient when many objects are live. Also, it requires certain data structures (such as hash tables) which rely on addresses to be reconstructed at each copy.

Java (in version 1.1 and 1.2) uses both algorithms (see [29]), and implements a policy to switch from one kind of GC to another so that there is a minimum loss of efficiency.

2.3.4 Generational GCs

Generational collectors [53] rely on the observation (made from practical experiments) that objects tend to die young. This means that, a short period of time after their allocation, these objects are no longer needed by the program and can thus be reclaimed. Consequently, one can organize the heap to provide a specific location for young (i.e. newly allocated) objects. The knowledge of this location allows one to run a garbage collector focusing only on young objects and thus to reduce the time spent doing garbage collection.

Concretely, the heap is divided into memory areas called "generations", the area reserved for young objects being called *nursery*. Objects that survived a garbage collection in a given generation are *promoted* to the next generation. In general, only two generations are used: young (or nursery) and old. The young generation is collected very often, while the old generation is not. The technique to collect garbage in each generation can be of type Mark-and-Sweep or Mark-and-Copy.

From the implementation point of view, there exists one major issue: inter-generational pointers. When a given generation is collected, we need to take into account those pointers from other generations to objects in the current generation. A first observation is that few pointers normally exist from old objects to young objects, while pointers from young to old objects are frequent. Also, young generations are usually much smaller than old generations. Consequently, when a generation is collected, all younger generations are collected as well. This measure avoids extra work to identify young-to-old pointers. Various techniques exist to identify old-to-young pointers when collecting young generations: remembered

sets [88], card marking [83], page marking [62], and so on. Pointers originating from an older generation are considered as roots.

Finally, we note an interesting work on a high level view of generational algorithms: age-based algorithms [85]. Different strategies can be applied based on the age of objects. Generational algorithms, as described above, become a particular case of age-based algorithms.

From the interactions point of view...

Generational collectors [53] use several GC entities. We can view the collection phase of the young generation as a different entity than the collection phase of the old generation. The global strategy defines the generational architecture and the rules that each entity should follow. These rules are what we call "interactions" between local entity (a generation) and global strategy (the generation algorithm itself).

Firstly, each entity has a well-defined scope of action. The collection phase of the nursery limits its visit of the heap to the nursery, while collecting the old generation usually performs a visit of the full heap. Secondly, the main interactions we identify relate to cross-generation pointers. To handle young-to-old pointers, the strategy specifies that younger generations should be collected along with older generations.

For old-to-young pointers, we identify three interactions. One is actually an interaction between allocator and collector: the allocator records – for GC purpose – these pointers when they are created. The other interaction requires a generation to consider as roots those pointers coming from older generations. Finally, the last interaction is to record that some of these pointers are not special anymore once a collection occurred. This is necessary because surviving objects will be promoted and, thus, certain pointers will not remain inter-generational.

2.3.5 Incremental GCs

Applications like real-time software require a bound on the time spent doing each GC. By limiting the amount of memory collected at each GC, it is possible to offer such a bound.

We describe the train algorithm [38] (also known as MOS), which has been integrated in Java since version 1.3 [58]. This algorithm is the result of observations and experiments about generational algorithms. In most generational GCs, the nursery is small (to allow fast collection) and the oldest generation is very large. Although the nursery collection runs quickly, the not-often run GC for the oldest generation (which actually collects the whole heap) can be disruptive and is not suitable for real-time applications.

The idea is to divide the old generation into blocks called **cars**. All cars have the same size, as determined by the implementation. To handle cycles spanning several blocks, cars are grouped into larger structures called trains. A car belongs to only one train and trains are ordered by age. At each collection of the old generation, only one car is collected along with the nursery (it is assumed that the generational algorithm used as a basis has only two generations). Which car is being collected is a matter of implementation policy, the article [38] describing the algorithm proposes a round-robin mechanism. It is important to start from the oldest train to evacuate all its objects and reclaim quickly the garbage it holds. As for the generational technique, the garbage collector for each car can be of type Mark-and-Sweep or Mark-and-Copy.

As can be imagined, keeping track of inter-car pointers is essential so a remembered set (see Section 2.3.4) is associated with each car. A remembered set is also associated to each train and is the union of its cars' remembered sets (remsets) minus all pointers between its cars. Before each collection, the remset of the train owning the chosen car is checked. If the remset is empty, all objects inside the train are garbage and the train is reclaimed as a whole.

Now we can see easily why this algorithm is incremental. Each collection is bounded by the size of a car. It is possible to compute precisely the upper bound of the time needed to perform the GC.

From the interactions point of view...

As for generational collectors, the main interactions are induced by cross-boundary pointers. In this case, pointers across cars and across trains. Remembered sets (remsets) are used to keep track of them. We identify collection entities as those GC phases collecting a car and focus on the interaction between collection entities for maintenance of the remembered sets. Each collection entity agrees to use its remset as an additional root set.

We also observe that a normal collector, such as a Mark-and-Sweep or Mark-and-Copy, has to be modified to limit its work to only one car at once and to use the car's remset as a root set. In this sense, it is an interaction between the low-level collection algorithm and MOS. MOS determines the strategy that the collector has to follow.

2.3.6 CMM: Customizable Memory Management

CMM [3] is a garbage collection framework for uniprocessor environments that organizes the heap into subheaps (composed of non-contiguous blocks of memory). Each subheap can use its own memory manager. The framework is composed of a primary collector, a variant of Bartlett's "Mostly-Copying" algorithm [5], and C++ classes to handle multiple heaps. The main challenge in CMM is to handle cross-boundary pointers. This is done using specific methods and by controlling the actions of a collector when it visits a subheap it does not manage.

In this GC, we identify collection entities as the collectors handling each subheap. The primary collector is one entity for example. CMM specifies that when a collector traverses a subheap it does not manage, no modification should be

made by this collector. This rule constitutes the only interaction between collection entities in CMM. Concretely, the use of the methods `traverse` (to visit objects in memory) and `scavenge` (to move or mark live objects) help implement this interaction. When a GC entity traverses a subheap it does not manage, the method `scavenge` does nothing, (it does not move live objects, for example). The method `traverse` depends on the type of object rather than the collection entity and is thus helpful for the collection activity but not central to our discussion.

In summary, any collection entity in CMM allows other GCs to traverse its subheap, and expects them to do nothing with live objects found there. In practice, this is enforced by the method `scavenge`.

2.4 Multiprocessor garbage collection

As *single-memory, multiprocessor* machines began to spread, people interested in garbage collection were tempted to dedicate a number of processors to do the collection concurrently with working processors. The overhead of garbage collection becomes minimal, thus allowing this memory management technique to be used in a wide range of applications.

Before reviewing some techniques, we describe the general idea behind the term "concurrent GC", and differentiate the terms **concurrent** and **parallel** as they are often interchanged whereas they probably should not be. In a traditional uniprocessor system, the garbage collector operates in *stop-the-world* mode. This means that the GC may stop the computation at almost any time to fulfill its requirements, and mutation resumes after the collection. Multiprocessor collectors propose two possibilities. Either collection is performed concurrently with mutation, or mutation is stopped, as in a uniprocessor environment, and garbage collection is done in parallel by all the processors. We can see the difference between concurrent collectors and parallel collectors. The former runs at the same time as mutation and requires minimal pauses from the mutator. The latter

pauses the mutator and performs garbage collection very fast.

In this section, we summarize the ideas of four relevant papers: Steele's 1975 article [84] which is the first paper on the topic, Dijkstra *et al*'s 1978 article [28] which introduced the terms of **collector** and **mutator**, Boehm's 1991 paper [12] which shows a more recent example of research work on the topic, and Endo's work [30] on a parallel garbage collector. With the latter, we provide a detailed study of interactions.

2.4.1 Multiprocessing compactifying GC – Steele

This work was published in 1975 by Steele [84] and has the advantage to present multiprocessor garbage collection as a "fresh idea". Interesting issues are explained including concurrency aspects and problems occurring when more than two processors are used. Experiments were conducted using a Lisp dialect, the working processor was called a "list processor".

The main challenge of this algorithm resides in the fact that the collector compacts data (i.e. move objects close together in memory). In a uniprocessor environment, this does not present any particular problem: data is copied from one place to another in memory, pointers are updated and the computation is restarted. However, in a concurrent environment, processors used for computation may try to dereference an object that has already been moved. The relocation algorithm thus needs a way to handle this problem. The solution is to implement forwarding pointers coupled with semaphores. Like in the uniprocessor Mark-and-Copy algorithm, forwarding pointers are left in the old location to refer to the new one until all references have been updated (this is guaranteed to happen because the collection process visits *all* objects). Semaphores are needed to synchronize with the computation processors: the GC locks access to an object while it is moving it. Once the semaphore is released, the computation simply needs to follow the forwarding pointers updating its reference at the same time.

Other problems include the management of the freelist and newly created cells. New cells have to be marked in some way to warn the collector of their existence. To avoid concurrency problems, free cells are added to the end of the freelist. The allocator takes free cells at the head of the list.

[84] also discusses various multiprocessor configurations. In particular, the case of more than two processors is explored. As long as only one collection processor is used, there is no noticeable difference. However, if we use several collectors, two possible schemes are identified. In the first one, all collectors are synchronized and in the same state. The work is divided and everything happens as if we have only one collector. Another interesting case is to have synchronous collectors. For example, we can divide the memory in several blocks, each block being managed by a collector. Of course, inter-block pointers have to be taken into account. Further details can be found in [84].

2.4.2 On-the-fly Garbage Collection – Dijkstra

In [28], Dijkstra *et al* explore the minimum constraints required to obtain a fully concurrent garbage collector (computation processors are required to assist garbage collection processors). This paper also introduces the now widely used terms of *mutator* and *collector*.

To solve the problem of collecting garbage using another processor, the authors propose to use a graph algorithm. As we have seen in Section 2.3.2, the heap can be considered as a graph for GC purpose. Objects correspond to vertices and pointers are similar to edges. The scheme is based on a graph traversal algorithm. Concretely, the GC algorithm used in this paper is of type Mark-and-Sweep.

Tricolor marking is used to keep track of garbage and non-garbage objects. During a collection, *black* denotes a reachable object, *gray* means that an object is potentially reachable, and *white* corresponds to garbage. Gray is used to handle the fact that a pointer can be modified by the mutator at any time, even during

collection. If the collector has already examined the object, then something has to be done to indicate to the collector that it should examine it again.

The paper is focused on finding the minimum level of interaction between the two processors. It is proved by an example that shows that the mutator is *required* to take actions to help the collector (i.e. mutator and collector can not be completely independent). We consider three cells A, B and C. A and B are reachable nodes. Originally B and C are linked. The mutator and collector are working in parallel. The following steps may occur in this order:

1. The collector examines A and marks it as reachable. A has no descendant.
2. The mutator creates a new link A-C. A now has a descendant.
3. The mutator breaks the link B-C.
4. The collector examines B and marks it as reachable. B has no descendant.

Eventually, C will be reclaimed because it has not been marked as reachable, which is wrong. We need the mutator to take specific actions when it manipulates already marked pointers. From this result, the authors designed an algorithm that solves this problem using the smallest possible constraints. The main idea is to reduce large synchronized operations to the smallest possible actions in order to avoid loss of efficiency on the mutator side.

2.4.3 Mostly Parallel Garbage Collection – Boehm

The paper [12] looks at a practical solution to implement a concurrent GC and explores a compromise between practical and fully concurrent GC. We start this description with a warning about terminology: in that paper, the term "parallel" is used in place of "concurrent". This is unfortunate as it may lead to confusion. Our overview "translates" and explains the algorithm as if the paper were called "Mostly Concurrent Garbage Collection".

Dijkstra's solution [28] to a concurrent M&S is rather complicated and an implementation could become complex as well. Instead of trying to achieve optimal concurrency, the technique of [12] strives to achieve practicality. Indeed, the resulting algorithm is very simple. It relies on dirtying (i.e. marking as modified) pages during the concurrent phase and on an hopefully small stop-the-world pause to adjust certain marks. The length of the pause varies according to the allocation behavior of the mutator. If the mutator allocates memory frequently after the beginning of the GC, the pause is likely to be important. However, it is noted in the article that this behavior has rarely been observed in practice.

2.4.4 Parallel GC – Endo

Endo *et al* [30] designed a parallel garbage collector based on the BDW collector [13]. Mutation is stopped to let the processors work on the heap. Objects are marked from the local roots of each processor. An important characteristic of the collector is that **dynamic load balancing** is employed to handle load imbalance between processors (e.g. when a large tree is shared between processors, only one processor will mark it, leaving the other ones idle). From the description of the algorithm, we gathered many interactions with the overall strategy. Processors can not simply run a uniprocessor GC independently of the others. Specific data structures and actions are needed. We found the following elements:

- Each processor has its own local **mark stack** (a mark stack is a buffer used to tell the collector what pointer to follow next). The global strategy defines this new data structure.
- Each processor **marks objects** from its **local roots**. This is the basic interaction. No mention has been made of a common static area as root, hence we do not discuss interactions resulting from possible synchronization choices at this level.

- Synchronization is needed to mark objects. Before a processor accesses an object, it attempts to **acquire a lock** on the mark bits.
- A minimal solution to the problem of load balancing is brought with **stealable mark queues**. This is a new data structure to handle. This allows processors to "steal" tasks (i.e. pointers to follow) when they are idle. Every processor should perform several steps regularly. No specifics are given on the matter of exactly *when* these have to be performed. This interaction seems to be left open to implementers. The steps are as follows:
  - Check if the associated stealable mark queue is empty.
  - If it is, pointers from the mark stack are moved to the stealable mark queue.

  This effectively defines an algorithm that each GC entity has to handle.
- Processors also have to **use** the stealable mark queue. The overall strategy defines rules for this to happen. This is another algorithm that each processor has to execute. When a processor becomes idle, it verifies stealable mark queues (starting with its own), tries to lock one, and **steals** half of the entries by integrating them into its own mark stack.
- Termination is detected with a **global counter**. Interactions lie here in the need to increment and decrement the counter, whenever an event occurs (idle, awake).
- To handle the sweep phase in parallel, processors are required to **acquire a number of heap blocks**, and **merge** contiguous free blocks when possible. This happens **repeatedly** until all blocks have been examined. This avoids the need for synchronization.

These interactions are sufficient to define how a processor operates within the global strategy. However, it was observed during Endo's experiments that the

speed-up could be made better when certain operations are handled differently. This means that new interactions happen.

- To avoid load imbalance due to large objects, a processor is required to **split** such objects into chunks to allow other processors to steal the task of marking and examining one chunk, rather than waiting for *one* processor to complete the marking of a large object.
- Synchronization can be improved with a different procedure. Instead of waiting to lock a queue to steal its contents, a processor is required to only **try** and **proceed to next queue** if the lock could not be acquired.
- A problem is the cost linked to detecting termination. Instead of a serialized access to the global counter, additional data is maintained. **Two flags** are associated with the mark stack and stealable mark queue for each processor. No lock is needed to check them. However, each processor needs to check all flags in order to detect termination. A third flag is used to allow synchronized checks without locking the flags.
- During the mark phase, mark bits of objects are locked for checking and possibly marking. However, this locking operation occurs even to simply check if an object is already marked, which is not efficient. The new operation required by the overall strategy first reads the mark bit without locking and continues if the object is already marked, otherwise an attempt is made at acquiring the lock.

An interesting observation here is that this collector does not act on any subheap. It operates on all the objects that it can access. This is different from what we have seen with uniprocessor collectors and what we will see with distributed GCs. This peculiarity comes from the fact that the architecture is "reversed": several computing devices act on a single store, instead of a single device acting on several logical stores or multiple devices acting on multiple stores.

2.5 Distributed garbage collection

Distributed computing environments require powerful and flexible memory management. Although theoretical papers have been published in the area of distributed garbage collection since the early 1980s, design and implementation of DGCs have always been considered too complex for practical use. Indeed, a number of new and interesting problems arise when considering a distributed scale. This section discusses distributed garbage collection by presenting a general model, a classification of DGCs, various DGC-related issues, and an overview of several current algorithms. A more complete presentation of distributed garbage collection can be found in [1], [75], and [41].

2.5.1 Basics

Terminology

Let us consider a distributed system with the architecture illustrated in Figure 2.1. From the memory management point of view, we define several layers of the system. First, we consider the graph of **objects** including heap objects and their links (pointers). The second layer is the graph of network **nodes** (also called *sites* or *spaces*) with logical links between them. Such links are defined by the application rather than physical configuration. Finally, the network graph corresponds to the physical layout of the machines involved in the distributed system. In all schemes we present in this section, **message passing** is the only means of communication. No read/write in a shared memory has been considered.

Distributed garbage collectors primarily focus on **public objects**, which correspond to those objects referenced by objects on a remote node. A node usually denotes a process managing its own heap (i.e. using its own garbage collector to handle the local graph of objects). The graph of objects on a node is organized as follows. Local roots (stack, static area, and registers) keep objects alive. As long

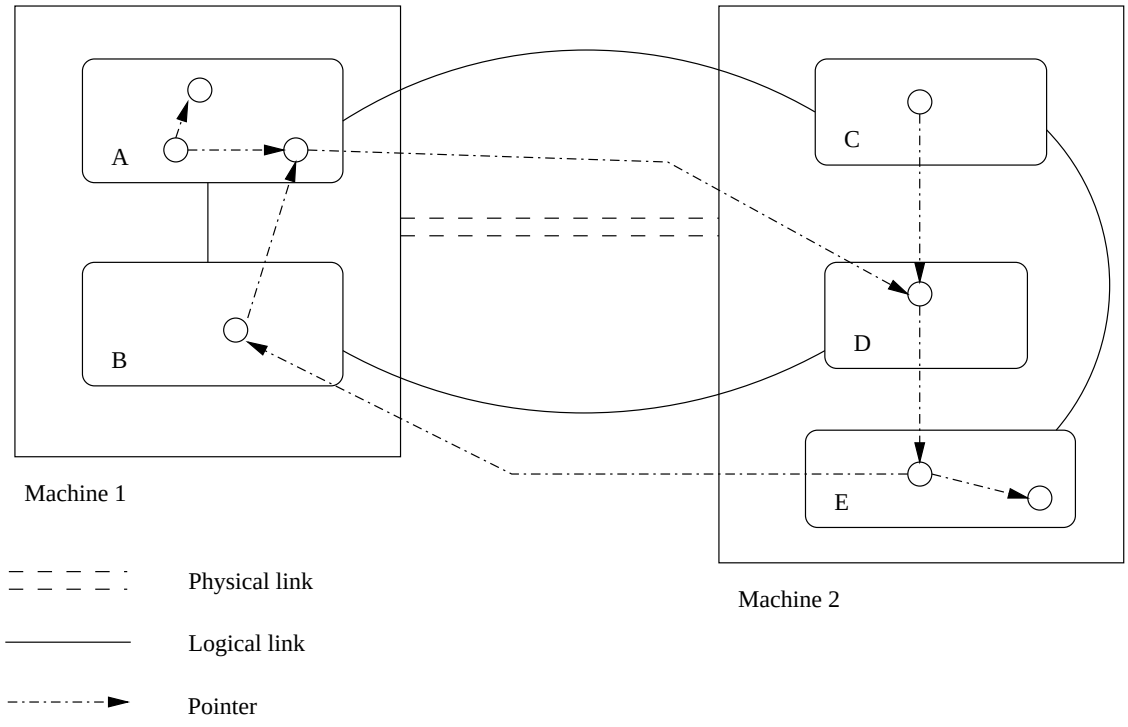


Figure 2.1: *General DGC model*

as an object is accessed locally or accesses other local objects, no difference can be made with a uniprocessor environment. However, in a distributed setting, it is likely that certain objects become public and are accessed by objects on another node using what is known as **remote pointer**. Such a pointer is a cross-node link between two memory objects. We also introduce two new sets of elements that are part of the heap: **entry items** and **exit items**. They are gathered under the term **opaque addressing**, and can be thought of as representatives of local objects for other nodes. Entry items contain pointers to their corresponding objects and are referenced by other nodes that need to manipulate this object. Exit items act like proxies, when a local object references a remote one. This technique provides a simple way to access remote memory objects as if they were local. Only these special items will hold network addresses and data needed to access the actual remote object.

It should be clear now that a DGC is not a single entity that migrates from node to node collecting garbage. Rather, a DGC is a composition of:

- **Local GCs** which handle memory management at each node of the distributed system. They are usually responsible for reclaiming garbage that has been discovered either locally or by actions of the DGC algorithm.
- **Network protocols** which define messages that should be transmitted between nodes to coordinate the distributed garbage collection effort. Once a node knows what objects are used by another node, it can figure out what objects to keep and what objects to collect. For example, a Distributed Reference Counting algorithm relies on "increment" and "decrement" messages.
- A **general algorithm** which explains the global strategy as well as some optimization ideas (e.g. batching of messages).

Classes of distributed collectors

We distinguish the following families of distributed garbage collectors: Distributed Reference Counting, Hybrid, Global structure based, and DTD-based.

A **DRC algorithm** relies on counters placed on entry items to keep track of the number of remote references to a local object. No local action other than maintaining the local proxies and associated counters is required. Basically, this type of DGC needs the local GC to handle counter `increment` and `decrement` operations, resulting from the export of a reference or the reception of a DECREMENT message. The local GC handles the updating of counters and reclaims entry items when their counters reach zero. The subsequent actions depend on the nature of the local GC. If it is a reference counting scheme, the corresponding counter will immediately be updated. Otherwise, the object previously referenced by the entry item will be considered for reclamation if no other object references it. Local GCs can use any technique (although explicit memory management may require some adaptation).

In this class of DGCs, we find techniques such as Weighted Reference Counting (see [90]) or Distributed Reference Listing algorithms (e.g. Shapiro's SSPC [81], Moreau's HDRC [63] and so on). DRL algorithms are slightly different in that they list the nodes referencing a particular object instead of simply counting them. This allows for a finer management and aids in supporting failure handling.

Of course the problem with DRC algorithms is that garbage cycles can not be collected. **Hybrid algorithms** usually provide a way to deal with such data structures in the form of a specific algorithm on top of a DRC collector. Such techniques are also called "Augmented DRC" or "Augmented DRL", because they are actually a layer on top of those algorithms. This layer is usually derived from a Mark-and-Sweep style algorithm and adapted to a distributed environment. Hybrid techniques are widespread as they provide simple solutions involving low development time. In terms of local requirements, hybrid collectors are very de-

manding. They often require the propagation of information within a node from entry items to exit items. This is the usual way to maintain global information about the distributed graph of objects. Of course, there is no synchronized "stop-the-world" process and global consistency does not exist in this environment (as mentioned in [82]). Using a local collector allows one to use incomplete knowledge while providing a reliable solution to reclaim all garbage (i.e. to be complete). Furthermore, hybrid collectors usually rely on a Distributed Termination Detection algorithm to assess the end of a detection phase. Upon termination, it is possible to identify objects that are part of a garbage cycle using the marks (whatever their nature is: constant values, timestamps, ...) propagated by the local GCs.

Global structure collectors describe a class of distributed collection algorithms that rely on maintaining distributed data structures other than the graph of objects. For example, [37] describes the port of the local GC called MOS [38] (a.k.a. the train algorithm) to a distributed setting. The idea is to maintain a distributed version of the trains by using specifically developed protocols. This class is different from the hybrid class in that it manages extra distributed data structures just for the purpose of garbage collection. This is likely to result in different requests for local GCs. Indeed, instead of simply requiring specific actions, the local GC has to maintain this distributed structure. This means that its nature might change according to what part of the structure is locally managed. For example, DMOS – as described in [37] – requests the local GC to behave like MOS, managing cars and trains. In comparison, hybrid collectors do not require any change in behavior except for managing *independently created and managed* local data structures.

The ideas developed in [8] define a separate class of DGC algorithms: **DTD-based** distributed collectors. The idea is to extend a stand-alone GC to a distributed setting by adding new network protocols to map the garbage collection

paradigm to that of a DTD algorithm. In a sense, it is the only class which *actually* distributes a garbage collector over a network of nodes. The technique described in this paper requires the exact same actions from all local GCs: creating and managing jobs and tasks for the DTD algorithm. Although we found only one paper on this topic, we believe that such high-level garbage collection schemes similar to this technique are likely to become more widespread because they propose a simple solution to a quite complex problem. The next step would be to try implementing such algorithms. In Chapter 5, we describe a design method that could be useful for this task.

2.5.2 Distributed Reference Counting

Distributed reference counting is the most common form of distributed garbage collection. For example, it is used in Java RMIs [59] in association with *leases* (which are expiry dates: an object is made available for a certain period – a lease – and can be reclaimed after this period has passed). Important characteristics of this technique are its simplicity and, like its uniprocessor version, the inability to detect garbage cycles. We first detail a simple algorithm, and then explain an optimization allowing reduction of the number of messages.

Simple version

We start by explaining how *basic* DRC works. We rely on the general DGC model and terminology described in Section 2.5.1.

A counter is associated with each entry item. Remote nodes send *increment* and *decrement* messages to allow these items to maintain accurate information about public object reachability (the degree of accuracy depends on the chosen mechanisms for synchronization and ordering of messages). Counters are initialized when entry items are created. Their initial value is usually 1 because an entry item is created when the reference to an object is exported. A counter on

an entry item reaching zero means that no remote pointer exists on the object represented by the entry item. This item can then be reclaimed. Exit items do not have to hold a counter, because they are handled by the local GC on each node. Entry items are used by the local garbage collector as roots. For example, if the local GC is a Mark-And-Sweep scheme, marking will be done from the stack, the registers, the static area (i.e. local roots) **and** from any entry item whose counter is not zero. The sweep phase would reclaim all non-marked objects and all entry items whose counter is zero.

Adapter code may be required depending on the local GC scheme. For example, if the local GC is reference counting, entry items whose counter reaches zero will be reclaimed right away, decrementing counters on the local objects they reference. Furthermore, it is not advisable – although possible – to move entry items when using a copying-based scheme. Indeed, remote references to a local object is done through these entry items, and remote nodes store addresses of these entry items in their exit items along with the remote node identification. Moving an entry item would require either a listing of nodes referring to the item and many update messages or forwarding pointers left for an undetermined amount of time. As mentioned earlier, this is possible, but it does not seem to be a particularly efficient mechanism.

Indirect Reference Counting

Different methods have been proposed to improve performance over the naive version of distributed reference counting. The most widely known is probably "Weighted Reference Counting" (see [90]) where weights are used instead of counters. Each time a remote pointer is duplicated, its weight is divided. This allows to reduce the number of "increment" messages because when a remote reference is created, a weight is automatically assigned. Also, duplications are handled without informing the "home node" of the referenced object because weights are

simply divided. This method does not reduce the number of decrement messages.

We now explain the concept of pointer duplication in a distributed environment. We consider that a node p is the owner of object o, and it sends a reference to o to node q. Then, node r sends a request to node q for a copy of this pointer. If node q agrees then the pointer is **duplicated**. The problem is that duplication happens without node p knowing about it. There are several ways to deal with this. The naive solution is for node r to send an increment message to p. As we are going to see, indirect reference counting avoids this cost.

Indirect Reference Counting is a technique to eliminate increment messages. It is sufficient to send pointers to other nodes along with some information.

To achieve this, an inverted tree structure called a *diffusion tree* is maintained globally. Please note that we are going to use the term *site* instead of *network node* for this explanation to avoid confusion with tree nodes. There exists one tree per public object. Each site manages its own node of this distributed tree. Each tree is single-rooted and the root is the home site of the object being remotely accessed.

Four scenarios are handled:

- **Creation** of a remote pointer.
- **Duplication** of a remote pointer.
- **Deletion** of a remote pointer.
- **Migration** of the object.

Each node counts its direct children. This means that when a site *creates* a remote pointer to one of its objects, the pointer is sent and the counter of direct children (remember this is an inverted tree) is incremented **locally** (however, a decrement message is needed if the remote pointer already existed at the destination node).

When a pointer is *duplicated*, the same process occurs but the home site of the object does not need to know about it. The node of the tree at the destination site sets its "parent" to be the node who agreed to send a duplicate. The counter of direct children does not change at all, thus avoiding the cost of an increment message.

Deletion of a remote pointer can be a problem. Remember that the tree represents the "history" of duplication of remote pointers to an object. It does not represent an actual distributed structure used by the mutator. The situation is the following: node q deletes its remote pointer to an object o. If the site was represented by a *leaf* in the tree, then this leaf can be removed and a *decrement* message is sent to the parent node in the tree.

A problem occurs when the site is represented by an interior node. It can not be removed because other sites rely on this node to link them to the tree. Consequently, a "ghost" node is used and automatically removed as soon as it becomes a leaf (i.e. there are no children anymore). A decrement message is sent only when this node is removed.

Migration is handled by changing the root of the tree. The new node representing the home site is detached from the tree (if it previously existed in the tree) with its whole subtree and becomes the new root of the tree. The node of the old home site becomes a child of that node. We can see that very few updates are necessary.

Please refer to [73] for more details about issues and implementation. From this algorithm, Piquer derived a more general model called Indirect Garbage Collection and published an algorithm called Indirect Mark-And-Sweep [74].

2.5.3 Distributed Reference Listing

Distributed Reference Listing is an extension of Distributed Reference Counting. Instead of recording the number of remote pointers to an object, the exact list of

hosts referring to this object is recorded and maintained. This allows for better fault management. For example, if a faulty node is known, measures can be taken to decide the future behavior of the memory management scheme (exclusion, for example). As DRC, this technique does not reclaim distributed garbage cycles.

Hierarchical Distributed Reference Counting [63]

This DRL scheme presents a solution to the scalability problem (see Section 2.5.5). Certain objects are popular and many hosts have references to them. In this context, the number of messages may become enormous. This DGC solves the problem with a hierarchical organization which helps reduce the burden on most nodes of the system.

Although its title refers to reference counting, this scheme is really a reference listing DGC. It uses the concept of hierarchical organization of nodes to achieve better scalability. The model follows a regular model for reference listing, each node maintaining a list of hosts associated with each reference. In [63], two reasons are given to explain why distributed garbage collectors based on reference listing do not scale: (i) popular objects impose large space requirements to record their referents, (ii) locality is not taken into account (i.e. DRL algorithms do not rely on any notion of node proximity).

The claim here is that "massively distributed computations may be conceptually organized in a hierarchy". We can imagine machines in a lab, several labs in a department, several departments in a university and so on. This is a hierarchy of machines. For every level in the hierarchy, a *gateway* is set up which acts as the representative of its subnodes for the outside and conversely. This is actually a conceptual hierarchy, because the gateway may be part of the computation at the same level as its subnodes.

Total space needed in the system is greatly reduced although the higher a node is in the hierarchy, the larger its reference listing set must be. The root of

the hierarchy will contain all references, which might be a problem as it becomes a special node in the system. It is important to note that this hierarchical organization is only used by the garbage collector. Mutation messages can be sent directly to any node (DGC messages are sent to nodes in hierarchical order to maintain consistency). The paper presents two schemes: flat distributed reference counting and hierarchical distributed reference counting. To our knowledge, only the first one has been implemented to date. This work is, to our knowledge, the first to address directly the scalability problem. It also considers the idea of physical locality of computation nodes.

Stub-Scion Pair Chain [81]

In this section, we describe a scheme which provides fault tolerance and robustness to lost and non-ordered messages. Shapiro and Plainfossé suggest that a garbage collector has to be able to cope with failures. Based on reference listing, this scheme uses timestamps to achieve this result.

Stub-Scion Pair Chain (SSPC) is an acyclic distributed garbage collector relying on reference listings and timestamps on messages. As we saw in the previous section, reference listings imply a limitation on the scalability aspect of the scheme. The reason is that it is costly for each object of each node to maintain a list of all nodes that have a reference to it. This scheme does not propose any solution for this scalability problem. However, it provides a complete solution to handle failures.

The model of the SSPC scheme is as follows. The distributed system is a set of nodes called *spaces*. Asynchronous messages constitute the unique way to communicate. These messages may be lost, duplicated or delivered out of order. Remote referencing uses opaque addressing: *stubs* are very much like exit items and *scions* correspond to entry items. Timestamps are used to control the order of messages. Each space checks its communications with all other spaces

by maintaining a threshold. If a message arrives too late, it is refused and has to be retransmitted. Each space regularly sends a message to announce what are its live objects. Remote spaces can then update their view about the liveness of remote references.

2.5.4 Cyclic DGCs

Although DRC and DRL collectors are simple to implement, they lack an important feature: they can not reclaim distributed garbage cycles. Several solutions have been designed to solve the problem. Most algorithms are of the *Hybrid* class and are thus augmented DRCs or DRLs. In this section, we will see an example of an augmented distributed reference counting scheme as well as an example of augmented distributed reference listing. We will also present a DGC of the *Global structure-based* class. This algorithm called DMOS is the only existing distributed collector based on an incremental GC algorithm (MOS). We conclude by presenting a migration-based algorithm which migrates garbage to a single node to let it be reclaimed by the local collector.

Augmented DRC: Garbage Collecting the World [51]

This scheme is one of the most interesting because it provides answers to a number of issues. It claims to be fault-tolerant, to avoid the need for a centralized control and to be comprehensive. We explain how these results are achieved and emphasize on three characteristics: groups, reclamation of distributed garbage cycles, and fault-tolerance.

Based on distributed reference counting, this algorithm integrates a solution to detect and reclaim distributed garbage cycles using a mark-and-sweep technique. A specific architecture is created around the notion of *group* that is simply defined as a set of nodes.

A group limits the scope of action of the DGC, which thus detects distributed

garbage cycles in an acceptable timeframe even in very large networks because it only acts on part of the network rather than the entire system. Several groups can be defined to cover the entire network. Groups also help to manage fault-tolerance by allowing dynamic removal upon detection of failed nodes.

The technique used to detect dead cycles is a distributed Mark-and-Sweep. The main idea is to have local collectors propagate marks from roots and entry items to exit items. A special network protocol is used to globally propagate marks from exit items to entry items. After global stabilization of the system (i.e. marks on items are definitive), the system can look at non-marked items and reclaim them, thus breaking the cycles and leaving the rest of the work to local collectors.

Augmented DRL: Cyclic SSPC [52]

This extension of the SSPC algorithm uses timestamps to reclaim cycles. It relies on a central server to compute a minimum global time from local minima sent from all the nodes. The idea is that, at each local collection, a constantly increasing timestamp will be propagated among stubs and scions. If a garbage cycle exists, stubs and scions will hold a constant timestamp. When the minimum global time is increased to a value greater than the one held in the garbage cycle, the cycle can be reclaimed. Due to the need of timestamp propagation, this DGC requires a tracing algorithm for its local GC. For more details about this extension to SSPC, please refer to [52].

Garbage Collecting the World: one car at a time [37]

This DGC explores the possibility of using an incremental scheme for a distributed system. The paper [37] proposes to use the train algorithm (also known as the MOS system [38]) in a distributed environment. This scheme is called Distributed Mature Object Space.

The idea behind the train algorithm is that, in generational schemes, the collection of the old generation can be disruptive because the memory area is usually large. The train technique, as we have described for MOS, divides the old generation into a number of blocks called cars, so that, at each collection, only one car is collected. The time of one collection is thus bounded. Trains group cars together to collect cycles spanning several cars (concretely trains group together data structures larger than a car).

In DMOS, we distinguish three elements: computation, objects and pointers. The computation represents the node (a process managing its own heap). Objects are the basic structures used in the computation and pointers link objects together within a single node or over a network. This is a classical model. Nodes communicate via messages. A computation is allowed to mutate or move any object. A particular object can only reside on one node, called the *home* of the object, at a time.

The garbage collector is a locally copying collector. This means it is allowed to move objects within a node, but not from one node to another. Only the mutator is allowed to perform this operation. The scheme describes algorithms and protocols to handle object migration (using a "pointer tracking algorithm"). The purpose of DMOS is to provide a comprehensive scalable scheme also offering safety and incrementality. Fault-tolerance is not handled in this scheme, because the authors suggest that this aspect is the responsibility of the underlying system.

Each local collector is a modified MOS collector. A car belongs to one node only whereas trains can span several nodes. Reclaiming distributed garbage cycles is allowed through this facility. By isolating a distributed cycle in one train only (even if it spans several nodes) and evacuating live objects from this train, it is possible, upon detection, to reclaim the entire train, releasing its memory space. This is done *asynchronously* on each node.

Recently, the authors published an update to this algorithm [8], using a new technique – based on distributed termination detection algorithms – to simplify most of its aspects.

Migration-based

Migration-based distributed garbage collection algorithms rely on **moving** objects from one node to another. Many people consider migration for GC purpose unacceptable, arguing that garbage collection should be "transparent". In particular, DGCs should not migrate objects between nodes because it could hurt the computation process in terms of performance (the location of an object might be a strategic choice made for optimal performance).

Migration-based DGCs took the step to propose techniques to reclaim distributed garbage cycles by trying to move them to a single node where they can be reclaimed by the local collector. We are going to take a look at a particular technique called "Controlled Migration" [54]. This technique, created for databases and persistent systems, avoids unnecessary migration by scheduling for migration only those objects part of a distributed garbage cycle. Furthermore, it strives to reduce the number of "leaps" from one node to another by estimating the final destination node. In this case, migration is acceptable, because garbage objects are not accessible to the mutator.

The algorithm proposed in [54] is based on a reference listing mechanism. It migrates garbage objects deemed part of a distributed garbage cycle to a single node (which may be different for each cycle) to let the local GC take care of reclaiming the cycle. The probability of being part of a distributed garbage cycle is assessed by a threshold on estimated distances to a root object. The use of this threshold avoids as much as possible migration of live objects (although there is no guarantee that live objects will not be moved). Optimizations include objects batching and estimation of final destination to avoid multiple migrations.

This DGC has been created for an object-oriented database system called Thor [34] where many objects are persistent and their respective locations are vital for good overall performance. In such a system, failure handling and efficient garbage cycle reclamation is very important. Failure handling is supported by the distributed garbage collector through the use of a distributed reference listing DGC. The terminology used in this article is **inlists** for references to a local object (entry items) and **outlists** for references to remote objects (exit items).

Cycles are handled by estimating, for each object, the minimum distance across nodes to a root (it is actually to a *persistent* root, but it is not important for this overview). To compute the distance of an object, the shortest path from a root to this object is chosen. Locally reachable objects have a distance of zero. If an object o is on node B and is accessible only from a root r on node A, then the distance of o is 1. The idea is that garbage objects that are part of a cycle will have a constantly increasing distance as they are not linked to any root. This happens because distances are estimated with a distance field in inlists and outlists. When a local collection occurs, distances are propagated from roots and inlists to outlists. The distance field of an outlist will contain 1 + the minimum distance propagated to it. It is important to note that distance fields are associated only with inlists and outlists not with all objects, and, to simplify distance propagation, inlists are sorted by distance. When local propagation of distances occurs, garbage objects part of a cycle will not have roots to keep their distances low. At each node, the local GC propagates distances and adds 1, thus increasing distances without bound. The algorithm computes a **threshold value** beyond which objects are considered as garbage. This value is based on expected distances of live objects. This is why there is no guarantee that live objects will not be migrated. However, the scheme is still safe and complete, because live objects, even if migrated, can not be incorrectly reclaimed, and all garbage is eventually deleted.

To optimize migration, all related garbage objects on a node are batched for transmission to avoid creating too many remote references while having to migrate these objects anyway but at a later date. The question of *where to* migrate garbage objects is also an important one. Multiple migration should be avoided for performance reasons. A possible solution is to use a fixed "dump node", but this has the drawback of placing unnecessary stress on a single node. Instead, the algorithm of [54] orders nodes by IDs (identifiers) and tends to migrate objects to the node with the highest ID, which holds a part of the distributed garbage cycle. To estimate this node, a "destination" field is added to inlists and outlists and destinations are propagated in the same way than distances. Instead of a termination detection algorithm, a second threshold is used to decide when the destination has been evaluated. Once again, there is no guarantee that the destination node will be correctly evaluated. However, this affects only performance, not safety and completeness of this DGC.

2.5.5 Issues

Garbage collection is a useful technique to automatically deal with memory management problems such as garbage objects and dangling pointers. However, a certain number of issues have to be solved. In a uniprocessor environment, these issues are: completeness (to reclaim all garbage) and safety (to reclaim only garbage). In a concurrent environment, the most important problem is concurrent accesses to the heap. In a distributed system, other problems arise:

One important issue is related to **node failures**. Distributed garbage collection has to continue working even in the presence of failure. This question receives a different attention in published algorithms according to the authors point of view. Opinions range from "failure is the responsibility of the underlying system" to "failure handling is critical and DGCs have to handle or even support it". This problem is hard to solve and the solution is usually not cost effective.

People usually assume that the frequency of failures is not worth trying to solve the problem. This is true for small local networks. However, on a system like the Internet, nodes may fail very often.

Of all problems, **scalability** is probably the least treated issue. Most schemes claim to be scalable, but this is rarely proven or even discussed convincingly. None of the schemes encountered presented an analysis of their scalability. However, Moreau [63] took a first step by providing a solution to improve scalability in DRL algorithms. Furthermore, most DGCs deal with a maximum of ten or twenty machines in their experiments. Handling thousands or even millions of machines remains an open problem.

Efficiency is, of course, an important issue but not specific to distributed environments, since uniprocessor techniques are also concerned with efficiency. Yet, certain details may make this question more problematic in a distributed environment. Communication between processes is achieved by message passing. On a slow (compared to internal communications) network, a designer has to be careful about the number of messages needed.

In the literature, published DGC algorithms often provide informal statements destined to sketch a **correctness** proof. Unfortunately, these statements are rarely accurate and may not be reliable. The lack of simple formal constructs makes proving the correctness of DGC algorithms awkward. Distributed collectors have to cope with many parameters that can affect their correctness. The paper [65] defines a formal method to study and prove properties about garbage collection. Garbage collection is defined as "a relation that removes portions of the heap without affecting the outcome of the evaluation." Other interesting results of this paper include a proof that there exists no optimal collector and a technique that uses type inference to collect accessible but unused objects. A formal language called λ_{GC} is described which makes operations of allocation explicit, and exposes the heap as a property of a program. Specific definitions are

provided for different techniques of garbage collection. Details are available in [65] and a proof of correctness can be studied in [66]. [89] proposes an extension to handle distributed memory management. To our knowledge, this is the first and only work on this topic. This paper defines a new calculus called $\lambda_{//}$ which combines λ_{GC} and CCS, used for parallel and distributed systems and defined in [60]. In $\lambda_{//}$, allocation, processes, and communications are made explicit. A program is a set of nodes which are threads associated with heaps. This high-level calculus allows us to express allocation and mutation in the heaps, and manipulates integers, and GC and DGC algorithms can be expressed in a concise manner. Proofs rely on the definition of the "Program equivalence" property. As in [65], garbage collection is a transition preserving evaluation of programs. The program resulting from a garbage collection at a node is equivalent to the program before GC, in the sense that the thread part is not modified, the same evaluation is maintained and the resulting heap is smaller than the original heap.

The last DGC issue we describe here is the question of **Local/Global GC cooperation**. This aspect has largely been forgotten because algorithm designers focused on a specific architecture: most cyclic DGCs are a combination of DRC/DRL algorithms with a technique based on information propagation. Local collectors should be of the tracing family to accommodate these DGCs. Unfortunately, this does not take into account local needs of nodes where memory management performance is usually the most important. The purpose of local/global GC cooperation is to provide the DGC with global knowledge about the state of the system without requiring any synchronization (in [82], Shapiro compared the GC problem to the consistency problem). We study this question in Section 2.6.

2.6 Interaction semantics in a DGC environment

As we did for uniprocessor and multiprocessor environments, we now look at distributed garbage collectors from the point of view of interactions between global

strategy and local elements. We also use the DGC presented in Lang's "Garbage Collecting the World" [51] as an example. Our goal is to understand the structure of a distributed garbage collector and try to break it down in several well-defined pieces. As a result, we create a new element, called "Generic Garbage Collector", that separates the concerns of distributed protocols and local GC strategies in a DGC environment. The generic GC is described in Chapter 3.

2.6.1 Terminology

We start this study by establishing important terminology distinctions.

First, we call **stand-alone GC**, a GC that is not part of a distributed system or does not concern itself with the distributed environment it is possibly in. This includes uniprocessor and multiprocessor GCs. We prefer the term of stand-alone because it can relate to both. Also the terminology of uniprocessor and multiprocessor is historic. It does not take into account recent development such as threads with which we can have parallel or concurrent GCs using only one processor. The use of multiple processors can be left to the operating system to deal with.

A **local GC** is a GC that takes its distributed environment into account. Usually a local GC is a stand-alone GC that has been modified to work with a particular DGC. For example, to work with the "Garbage Collecting the World" [51] scheme, a Mark-and-Sweep algorithm can be adapted to propagate DGC marks while marking. To illustrate the difference between both terms, in the case of GCW, a stand-alone GC would be a normal M&S algorithm and a local GC for GCW would be the same algorithm modified to propagate GCW's marks.

2.6.2 Interactions

The interaction between a local GC and a distributed GC can be characterized as a client/server relationship. The DGC is the client which requests actions to

be taken by the local GC. We call the set of services that should be provided by the local GC from the DGC point of view "requirements or need of the DGC." If those requirements are not provided, the DGC will not be able to operate properly. Depending on the class of the DGC, requirements will be more or less demanding. We focus on hybrid collectors, which have the largest number of algorithms.

Although we do not concentrate our efforts on the interactions *between* local collectors, it is still useful to mention this here for comparison. A DGC is a *strategy* for local GCs to cooperate across networks and manage cross-node references safely. This means that local GCs are *required* by the DGC to interact and they do so through network protocols. Messages sent between nodes prove vital to this global strategy, and it is important for a local GC to know *what* and *when* information should be sent. The answers to those questions determine the behavior and efficiency of the distributed collector. Thus, these particular elements are part of the interactions between GC and DGC, because they are consequences of the interactions between local GCs in a DGC environment.

Most of hybrid distributed collectors have two important requirements: local propagation of data and termination of dead cycle identification. *Local propagation* is an activity usually performed by the local collector on a node (but it does not have to be). The local collector examines entry items on this node and copies their associated values (e.g. a timestamp, the nature of these values is determined by the DGC algorithm) to the corresponding exit items located on the same node, by following the paths of pointers within local memory. If an exit item is reachable from several entry items, the "strongest" value is usually selected. *strongest* means highest value, more recent timestamps, or something else, as determined by the distributed collector. When an exit item is reachable from an element of the local root set, the propagated value is the strongest possible, as determined by the distributed collector. The local collector also helps with *dead cycle iden-*

tification by supporting the DTD algorithm chosen by the distributed collector. The local GC is usually responsible for computing one or several values that will help assessing the progress made towards termination. It is often the case that these values are computed right after local propagation.

We illustrate the nature of these interactions with an example using the DGC presented in Lang's paper "Garbage Collecting the World" [51] and described in Section 2.5.4.

Figure 2.6.2 illustrates how GCW operates. The black color indicates a HARD mark, gray indicates a SOFT mark, and white indicates no mark. We explain each step:

1. The algorithm uses two marks: HARD and SOFT. It starts by setting initial marks on entry items on the nodes belonging to the group (HARD for objects reachable from outside the group and SOFT for objects only reachable from inside the group). The requirement for maintaining marks and opaque addressing items is a **first interaction**.

2. Each node takes care of propagating these marks from entry items to exit items. At the same time, if an exit item happens to be reachable from a local root, its mark is set to HARD. This means that the local GC is asked to propagate marks. This is a **second interaction**.

3. Once local propagation is completed, messages are sent to connected nodes to propagate marks globally. This operation constitute a **third interaction** because sending a message should be done by the local GC as required by the DGC. As soon as all marks have been propagated throughout the group, global stability is reached. To discover this fact, the DGC requires each node to provide local stability information (not shown in the figure). This is a **fourth interaction**.

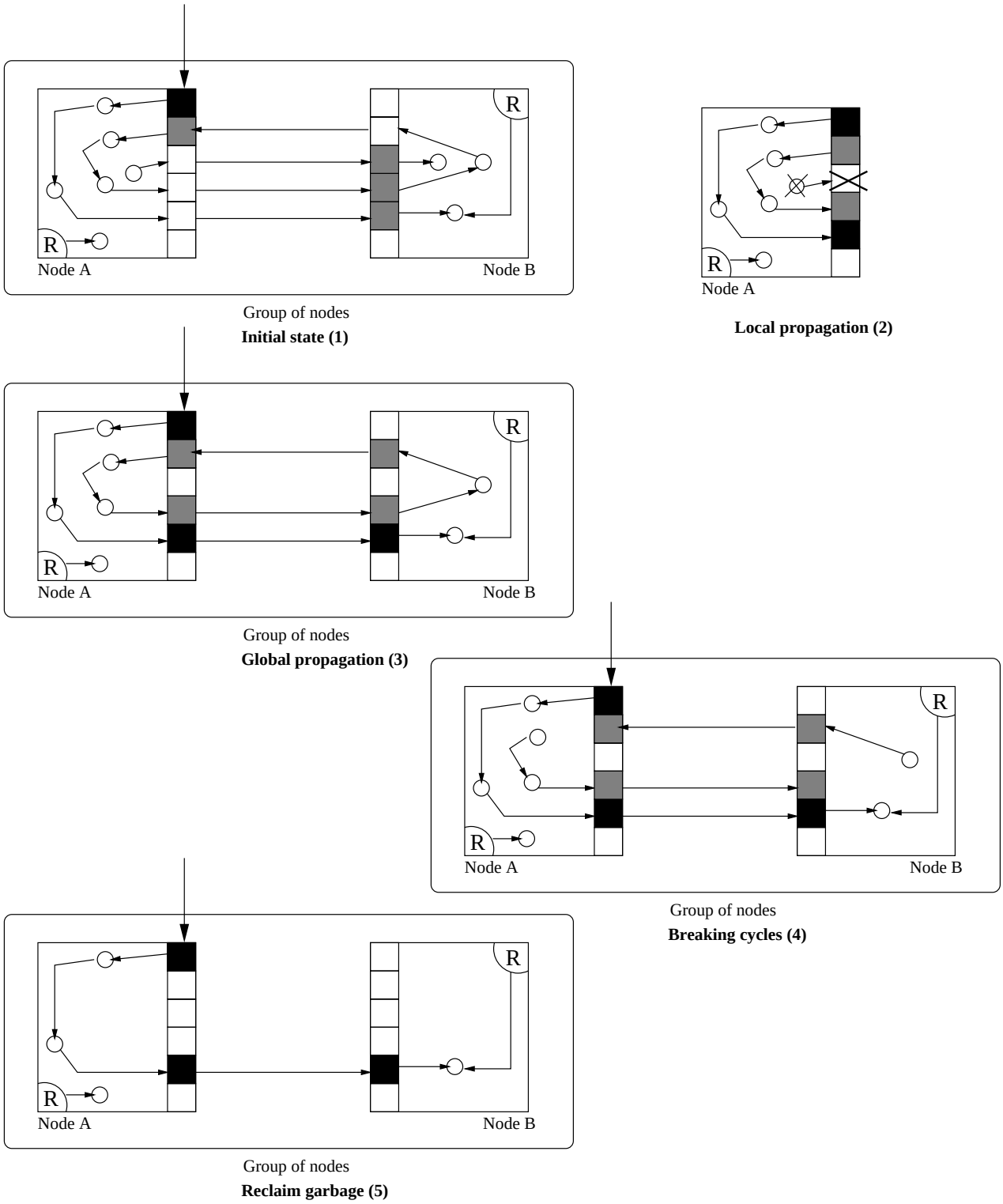


Figure 2.2: *Garbage collecting the World: an example.*

4. When global stability is decided, items marked SOFT are deemed part of a distributed garbage cycle. Such cycles are broken artificially by setting the pointers on corresponding entry items to NULL (**fifth interaction**).

5. Local GCs will then take care of reclaiming the garbage. Although this is an implicit requirement, we can consider this as a **sixth interaction**.

In this example, we can observe the different interactions between the local and distributed GC. We can observe that new **data structures** appear: entry and exit items with counters for the DRC algorithm and marks for the extension handling distributed garbage cycles. **Network protocols** also have to be handled by each node:

- increment and decrement messages to deal with distributed reference counting,
- negotiation protocol to create groups,
- global propagation of marks (i.e. between nodes, from exit items to entry items), and,
- a distributed termination detection protocol will be used to detect global stability of the system.

Furthermore, **local actions** are required from the node. Two important operations have to be dealt with locally as mentioned earlier in this chapter: propagation of marks (from entry items to exit items) and detection of local stability for the DTD algorithm. Although these actions would be performed more easily if they were integrated within the local GC, they are not required to be part of it. Indeed, we can even imagine that there is no local GC and local memory management is explicit. Some specific protocol would then be needed within the mutator to take care of the DGC's requirements.

This leads us to an interesting observation: identifying garbage objects can normally be done independently of any actual garbage collection. The work of a DGC is to inform the local collector that objects previously used by a remote entity do not need to be exported anymore. However, it is up to the local collector to actually reclaim those objects and free the memory. Consequently, another interaction between GC and DGC is the implicit *trust* that the local collector is *actually* going to take some action about the objects identified as garbage by the DGC.

Chapter 3

Generic Garbage Collector

In this chapter, we describe a novel entity that we call "Generic Garbage Collector". The generic GC can be applied to uniprocessor and multiprocessor collectors, even though we focus on distributed garbage collection. The purpose of the generic GC is the definition of a local garbage collector perfectly suited to answer the needs of a distributed collector. The DGC uses this entity to list its needs in terms of local treatment. The generic collector never exists as a piece of software, but rather acts as a template to specify interactions between a local collector and a distributed collector. It allows one to concretely model the different interactions recorded between GCs and DGCs. In essence, it is the model that has to be adapted when transforming a stand-alone collector into a local one for a specific DGC. It is a contract between both entities.

In this chapter, we present the model and an annotated example based on Liskov's Migration algorithm [54]. Finally, we explain how the generic GC has been used in practice in the context of our work on a DGC for the World Wide Web.

3.1 Model

Focus class

Object-focused or *Region-focused* or *Heap-focused*.

This class defines the general type of a local collector. It helps understand the view that the local GC should have of the local heap. This characteristic can be thought of as a high-level "hint" of what the DGC requires from its local GC. Most DGCs we encountered are said to work with "tracing local collectors." The corresponding class is *Heap-focused*, because each GC phase should visit the heap in its entirety.

Object-focused memory management techniques act on one object at a time. The graph of objects in memory is never seen in its entirety. Examples of such techniques are Explicit memory management and Reference Counting.

In contrast, *heap-focused* GCs have a complete view of the graph of objects at each collection. They visit each and every live object, but, depending upon the technique, garbage objects may not be visited. Classical examples of such GCs are Mark-and-Sweep and Mark-and-Copy.

Finally, *region-focused* techniques consider a subgraph of the graph of objects at each collection. Each collection visits only a specific part of the heap. The order of visiting is determined by the algorithm. Generational collectors are members of this class.

Concurrency class

Uniprocessor or *Multiprocessor Parallel* or *Multiprocessor Concurrent*.

Current distributed collectors assume uniprocessor GCs only. However, applications now use threads and garbage collectors are likely

going to use multiprocessor techniques more often. A distributed garbage collector may require a multiprocessor GC for performance reasons, for example.

Data structures

A local garbage collector is often required to maintain DGC-related data structures. We describe them using the following high-level information:

- *Identification.* To identify and refer to the data structure.
- *Description.* Description of purpose, usage, peculiarities, and so on, of this data structure using natural language. Although, this is not necessary, it may be useful for better understanding.
- *Operations.* Operations allowed on this data structure. Each operation should indicate its **input**, **output** and **effect on the structure**.

Data

In a local collector, there is usually a number of important constant values or global variables. It is useful to clarify the roles of each of them to support the description of the "Actions." We use the following elements to describe a datum:

- *Identification.* To identify and refer to the datum.
- *Description.* To explain the purpose and use of the datum.
- *Contents.* Show the datum itself possibly with its type and value.

Actions

DGCs usually require their local collectors to take specific actions to support distributed garbage collection. While distributed reference counting simply asks for a decrement operation of entry items, cyclic collectors usually need sophisticated operations to be performed.

- *Identification.* To identify and refer to the algorithm.
- *Description.* Basic, high-level information about the action. It can be used to specify assumptions for example.
- *Input.* Data or data structures provided to the action.
- *Output.* Results from this action.
- *Side-effects.* Non direct results from the action.
- *IDs of needed data and data structures.* List of variables, constants and structures used and relied upon by the action.
- *Algorithm.* This is a formal description of the steps required to perform the action.

3.2 Example: Liskov's Migration algorithm [54]

The distributed collection algorithm displayed in this section migrates objects deemed part of a distributed garbage cycle to a single node for local reclamation. The interested reader is referred to Section 2.5.4 for an overview of the algorithm. The local GC for this DGC essentially computes and propagates distance values. It triggers migration of objects when necessary.

- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Data structures:**

1. – **ID:** inlist
   – **Description:** List of public objects. This corresponds to entry items.
   – **Operations:**
     * `create`. Input: received outlist, Output: inlist, Effect: inlist contains pointers to remotely accessed objects, as well as estimated distances and destinations.
     * `getDistance`. Input: index of a pointer in the list, Output: distance to the referenced object as computed so far by remote nodes, Effect: none.
     * `getDestination`. Input: index of a pointer in the list, Output: estimated destination for deemed garbage as computed so far by remote nodes, Effect: none.
     * `getOrigin`. Input: index of a pointer in the list, Output: ID of the node referring to the corresponding local object, Effect: none.
2. – **ID:** outlist
   – **Description:** Records pointers to remote objects. This corresponds to exit items.
   – **Operations:**
     * `addRef`. Input: (node ID, reference on remote node), Output: none, Effect: new remote reference added.
     * `setDistance`. Input: (index, new distance), Output: none, Effect: distance of pointer in the list is updated.

* `setDistance`. Input: (index, new destination), Output: none, Effect: destination of pointer in the list is updated.

3. – **ID:** Distance
   – **Description:** This type is used to find out what objects are part of garbage cycles. The distance is based on the knowledge of the location of root objects. When an object is pointed to by a rooted object in the same node, the distance between them is 0, when this object is on a node A, the distance is 1. If a rooted object o_1 on node A points to o_2 on node B and o_2 points to o_3 on node C, the distance for o_3 is 2.
   Distances are propagated from one node to another, the smallest distance for an object is the distance of this object. Garbage cycles will hold objects with potentially infinite distances as there will be no root to keep the value from increasing as this goes from one node to another.
   `typedef int Distance`.
   – **Operations:**
   * `compare`. Input: another Distance, Output: *less than*, *equal*, *greater than*. Effect: none.

The focus class of the algorithm is *Heap-focused* due to the requirement of propagation of distances. We also observe that, like most DGCs, this collector relies on opaque addressing, using `inlist`s and `outlist`s. The notion of "Distance" is central to the DGC algorithm. This is why the description is detailed, although its representation is very simple.

- **Data**

  1. – **ID:** Threshold
     – **Description:** Chosen value that will serve as a threshold to decide whether or not objects are to be migrated because they might belong to a garbage cycle. Refer to [54] for more details about how to choose this value.
     – **Contents:**

     ```
     Threshold = A_CHOSEN_THRESHOLD
     ```

  2. – **ID:** Threshold2
     – **Description:** Chosen value that will serve as a threshold to decide when the final estimated destination for the objects of this garbage cycle has been determined and propagated. Once this is done, migration can begin. This value is thus very important. Refer to [54] for more details about how to choose this value.
     – **Contents:**

     ```
     Threshold2 = A_CHOSEN_THRESHOLD
     ```

- **Actions**

  1. – *ID:* local_propagation
     – *Description:* This function implements the core functionality of the collector. It describes what actions should be taken by each node to support the DGC activity. This local propagation can be easily integrated within a tracing collector such as Mark-and-Sweep. The main tasks are: propagate and update distances, propagate and update destination nodes. The Controlled Migration DGC is based on a DRL, the **outlist** is reconstructed at each

call to this function.

- *Input:* inlist.
- *Output:* outlist.
- *Side effects:* update outlists with distances and destinations from inlists and roots.
- *Data Structures IDs:* outlist, inlist.

2. - *ID:* propagate_local_roots
   - *Description:* helper function for *local_propagation*. Its purpose is to trace objects and propagate distance values to the outlist. This starts with a distance value of 0 as we start from roots.
   - *Input:* none.
   - *Output:* outlist.
   - *Side effects:* update outlists with distances and destinations from roots.
   - *Data Structures IDs:* outlist

Although not all of the required operations are listed in this example, this should be sufficient to illustrate the `Actions` category of the Generic GC model. A more complete model (including the description of the "Algorithm"s) is given in Section C.3. Interactions between local collectors and this migration-based DGC are visible within the described model. The DGC requires data structures such as `inlist` and `outlist`. The local GC must estimate new *distances* and *destinations* for remotely accessed objects. To achieve this, the local GC expects its `inlist` to be updated regularly. If these interactions are preserved and the needs of the DGC are fulfilled, the overall strategy (i.e. distributed collector) can operate properly.

3.3 In practice

As we will see in Chapter 7, we used the generic garbage collector to help us implement a DGC mechanism for the Web. The generic GC allowed us to identify the various parts of the system we wanted to build and we could then deduce the different steps in our design. We also gained in modularity because the layers defined by this architecture are part of the design from the beginning. This allowed us to create a complementary local collector (in the context of experimenting with interoperability in Chapter 8) for our distributed GC implementation without modifying the rest of our code.

Chapter 4

A Localized Tracing Scheme Applied to Garbage Collection

We present a method to visit all nodes in a forest of data structures that takes into account locality of reference to improve traffic within the memory hierarchy. This method is applicable to a wide range of uniprocessor garbage collection algorithms, and to a multiprocessor setting.

We call this technique a *Local Tracing Scheme* (LTS) as it improves locality of reference during the object tracing activity. An LTS can be used as an optimization technique at several levels of the memory hierarchy (cache, virtual memory, network).

We organize the heap into regions and use trace queues in the same way entry items are used in a Distributed Garbage Collection environment. Experiments with a Mark-and-Sweep collector for the language Aldor show performance improvements up to 75% at the virtual memory level. Although preliminary tests at cache level did not show significant speed-ups, increasing cache sizes and cache line sizes should allow the LTS to optimize this level of memory hierarchy eventually.

4.1 Presentation

Many algorithms require visiting all objects in a forest of data structures in a systematic manner. When there are no algorithmic constraints on the visiting order, we are free to choose any strategy to optimize system performance. An instance of this situation occurs in memory management where reachable objects must be visited as part of a garbage collection method.

Uniprocessor garbage collection is mature and offers satisfactory performance for many applications. It is now possible to use garbage collected memory in situations where it would not have been suitable just a few years ago. Incremental improvements in garbage collection technology thus have impact on a much broader audience than before.

These techniques share a common characteristic: they trace heap objects. Starting from specific objects known to be live (these are usually called *roots*), the process follows all paths in the graph of objects. It terminates when all vertices in the graph or predefined sub-graph (i.e. objects of the heap or predefined sub-heap) have been visited.

Mark-and-Sweep (M&S) collectors first visit all live objects, marking them, and then sweep the memory area to recover unused space. Optimization of memory traffic during the sweep phase has been considered by Boehm [13]. We observe that memory hierarchy traffic can be improved during the mark phase using an LTS. Since objects do not move in memory, the benefits of an LTS are similar at each GC occurrence, if data structures are preserved. Improvements of the overall GC time decrease when few objects are live. In this case, the mark phase is short and optimizations have a small impact.

Stop-and-Copy (S&C) garbage collectors move objects to a new location at each GC occurrence. To do this, they must visit all live objects. Although objects are close together in memory once copied, performance of the collector and the allocator may not improve with respect to this locality (see chapter 11

on cache-conscious algorithms in Jones' book [41]). Furthermore, Boehm [10] and Zorn [94] argue that S&C collectors do not necessarily perform better than M&S. Particularly, Zorn compares both techniques in a generational setting and concludes that M&S typically uses 20% less memory than S&C, but was 3%-6% slower on the problems he tested. While a copying collector apparently improves locality over time, these analyses prove that this factor is not sufficient to clearly improve performance. It is not clear whether the LTS could be beneficial for a S&C algorithm, and further study is needed. An obstacle is the fact that objects are copied in a far location and then examined for pointers that would typically be in the vicinity of the original location of the object.

Generational collectors [53] also use tracing because each generation is handled by a copying or mark-and-sweep collection algorithm (although copying is used more often).

The contribution of this work is to propose an optimization for the tracing algorithm used by these garbage collectors. Preliminary tests showed that most pointer distances are small. The difference between heap addresses at both ends of a pointer is very often less than the size of one or two pages. From this observation, we investigated how limiting the scope of tracing to deal with small distances first and larger distances later could improve a collector's performance. The LTS organizes its visit of the heap based partly on the graph of objects, and partly on the location of objects. A consequence is that the Localized Tracing Scheme is memory hierarchy friendly, which means that it is able to optimize visits of objects at different levels of the memory hierarchy: cache, virtual memory, network.

This technique divides the heap into *regions*. With each region, we associate a *trace queue* which holds a list of objects to visit. To a certain degree, this data structure is similar to the notion of *mark stack* used by Boehm [13]. Trace queues are actually the origin of the performance improvements displayed by the

LTS. They help to delay the tracing of "remote objects" (i.e. located in another region), concentrating on "local objects". This is a way to simulate locality of objects, relying on object location rather than object connectivity. The sizes of regions and trace queues are determined by the level of the memory hierarchy we wish to optimize for. For example, to obtain a cache-conscious algorithm, a region and the trace queues should be small enough to fit entirely in the cache.

The rest of this chapter is organized as follows. Section 4.2 describes a family of tracing algorithms dividing the heap to control visiting order. We also present our algorithm (the LTS) and make a comparison with a traditional technique. Section 4.3 details an example to illustrate the LTS. Section 4.4 presents our scheme from the point of view of the study on interactions presented in Chapter 2. Section 4.5 discusses an informal proof of correctness for this algorithm. Section 4.6 details our experiments and results with the GC for the Aldor compiler (see [48], [47] and [49]). We also discuss the different parameters of the algorithm. Section 4.7 explores advantages and drawbacks of the LTS in a multiprocessor environment. Section 4.8 and Section 4.9 present related work and conclude this chapter.

4.2 The LTS algorithm

The LTS can be customized in a number of ways, while retaining its main characteristic: to control the visiting strategy of the heap to improve performance over a regular, depth-first traversal of data structures.

4.2.1 Regular trace phase

We start by considering a regular trace algorithm (that we can find, for example, in a mark-and-sweep collector). This is useful for comparison purposes with our tracing algorithm and to explain a couple of issues we observed.

```
Main: For each root r
          Trace(r)

Trace(p): Mark object o pointed by p
          For each valid pointer p' in o
             Trace(p')
```

The operation called "Mark" has different meanings according to the collection algorithm that is used. For example, a mark-and-sweep collector simply sets a bit corresponding to the object, while a copying collector moves this same object to a "live area". In any case, the technique chosen to mark an object is only essential to the LTS to ensure termination of the process. It does not add to the principal idea of the optimization we propose in this work. Instead, we focus on how objects are visited.

We observe that the algorithm presented above uses a depth-first traversal. We identify two problems with this technique:

- The topology of the graph of objects has a direct influence on the behavior of the process within the memory hierarchy. A traditional tracing algorithm does not take advantage of the relative locations of objects in the heap. The focus is usually put on locality of reference rather than actual closeness of addresses. A possible consequence is a poor behavior up and down the memory hierarchy. For example, when virtual memory is required (i.e. the heap is larger than physical memory), a page may be brought from disk to main memory to visit only one object even if other live objects become available in main memory. The page can then be discarded to visit related objects that may be on another page. This could lead to thrashing. The same observations can be made for caching behavior.

- Recursion can be very deep (e.g. with a linked list). This causes a lot of activity on the stack side (allocation/deallocation of stack frames, ...). Traversing large data structures such as trees, matrices, linked lists, and so

on, is likely to incur significant stack traffic during the tracing phase of a GC.

In the rest of this chapter, we study the possibility of improving on those two aspects by transforming this depth-first process into a "semi-breadth-first" one. The following paragraphs will describe a family of these algorithms and a possible instance we call the LTS.

4.2.2 Family of tracing algorithms

The principal idea behind our tracing technique is to defer visiting objects which lie outside of a working set by maintaining queues in close memory (cache for example). When a queue becomes full, the deferred visits are made, altering the working set in a controlled fashion. This idea to localize the tracing process can be applied with minimal, localized modification to existing trace based garbage collectors.

Several strategies are possible for managing the deferred trace queues:

- One may keep all deferred object pointers in a common list, allowing or disallowing duplicates. When the list becomes full, it is analyzed to determine how to alter the working set. This has the advantage that the memory of the global queue is fully used, but the cost of the analysis may outweigh the benefit of making the optimal choice of working set alteration.

- One may associate a sub-queue to each range of addresses (heap region), with the number of ranges and size of sub-queues being parameters. Deferred object pointers are added to the appropriate sub-queue, allowing or disallowing duplicates. When a queue is full, the associated region is added to the working set and visits are made. This has the advantage that deferring visits is fast, but the disadvantage is the deferred trace queue as a whole

may be largely empty. This may be addressed by dynamically adjusting the size of the sub-queues based on use.

We have enumerated six strategies (common list, static sub-queues or dynamic sub-queues each of which allows or disallows duplicate deferred object pointers). We would expect the sub-queue strategies to be best when the far memory (RAM or secondary storage) speed is within a few orders of magnitude of the close memory (cache or RAM) speed. Beyond this, we would expect the common list strategy to yield better results.

Note that performing the deferred visits in a region (let us call it region A) may cause the trace queue of another region (region B) to fill before region A has been handled completely. It is then necessary to remove region A from the working set to allow the newly filled trace queue (for region B) to be handled. Unfortunately, this may cause thrashing if A's and B's deferred trace queues are nearly full for too many mutually referencing pages. Tracing then degenerates to the usual handling of tracing, but with substantial additional overhead. This may be avoided by taking one additional action: before performing the deferred marks on a region, the trace queue can be flushed into local store in the region itself. This local queue can be substantially larger than the per-region queue maintained in near memory.

4.2.3 Algorithm

We present a tracing algorithm where trace queues are associated with each heap region (the static sub-queues allowing the duplicates strategy described above). To allow fast access to these queues, they are contained in one contiguous area that we choose to be small enough to be maintained in cache.

Each region contains objects that will be marked and scanned. The difference with a regular tracing process is that scanning an object can reveal pointers *inside* the region currently collected or *outside*. If the pointer is to an object in

the region, the object is visited recursively. When it points to another region of the heap, it means that following this pointer would not be optimal for the working set or cache behavior. In this case, we simply place this pointer in a trace queue for later examination. We thus maintain the working set for as long as possible, and reduce the number of cache misses or page faults.

When the process for a region is completed, we proceed to another region. The policy to determine the order in which regions are visited is implementation- or even application-dependent. However, it is likely that choosing a region with a full or close to full trace queue will improve performance. A simpler solution, which avoids the cost of choosing the most populated queue, is to use a round-robin mechanism, and visit regions one by one. This is what we describe here.

In the initial step of the algorithm, roots are entirely dispatched into the different trace queues as if those pointers originated from an "external" region. Once the roots have all been visited, actual marking begins.
The complete algorithm is as follows:

```
mainTrace()
  InitialRootsScan() -- to fill in trace queues
  While not all queues are empty
     Q := choose a trace queue
     emptyQueue Q

emptyQueue(Q)
  While Q is not empty
     p := dequeue Q
     followRef p

followRef(p)
  o := object pointed to by p
  if (not marked(o))
     mark o
     Trace o

InitialRootsScan()
  For each root r
     Q := get trace queue for region where r points to
     enqueue(Q,r)
```

```
Trace(obj)
  For each valid pointer p in obj
     if p points in the current region
        o := object pointed to by p
        if (not marked(o))
          mark o
          Trace o
        else
          Q := get trace queue for region where p points
          enqueue(Q,p)
```

4.2.4 Algorithm with finite-size queues

We choose to explore the static sub-queues strategy. In this scenario, it is required that a limit is placed on the size of the queues. We thus need to handle the problem of untimely full queues. In particular, when we visit a region and need to enqueue a pointer into a full queue, the current working set is progressively discarded to switch to a new one dealing with the region corresponding to the full queue. Several strategies can be adopted:

- empty the queue and deal with the pointer
- deal with the pointer first and then empty the queue
- empty a percentage of the queue and insert the pointer in the queue.

The *first* strategy is likely to be the safest, because the first action is to remove a pointer from the queue which is not full anymore, thus allowing a new pointer to be enqueued. A situation, where we need to add a new pointer to this queue, can occur if, for example, the first visited object holds a pointer to a region which has also a full queue. In this case, this region is chosen to be visited and the first pointer may be to an object holding a pointer to the region we just visited. The *second* strategy allows to follow the new pointer first, thus removing the need to keep its information on the stack, but it is likely to become too costly in the

case described above. The *last* strategy we described may be chosen when the working set is not entirely filled by pages of the current region. In this case, a certain number of pages can be brought into memory without dismantling the current working set.

We choose here to empty the queue first and then deal with the pointer, although experiments would be required to decide what are the best strategies.

```
mainTrace()
  InitialRootsScan() -- to fill in trace queues
  While not all queues are empty
     Q := choose a trace queue
     emptyQueue Q

emptyQueue(Q)
  While Q is not empty
     p := dequeue Q
     followRef p

followRef(p)
  o := object pointed to by p
  if (not marked(o))
    mark o
    Trace o

InitialRootsScan()
  For each root r
     Q := get trace queue for region where r points
     if (not full(Q)) enqueue(Q,r)
     else
       emptyQueue Q
       followRef r

Trace(obj)
   For each valid pointer p in obj
     if (p points in the current region) followRef p
     else
       Q := get trace queue for region where p points
       if (not full(Q)) enqueue(Q,p)
       else
         emptyQueue Q
         followRef p
```

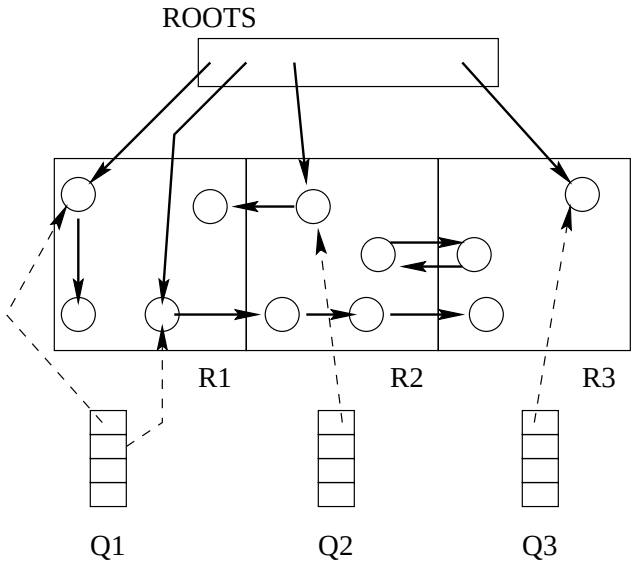


Figure 4.1: *Step 1: copy roots.*

4.3 Example

This section presents an example of the behavior of our algorithm. We follow the process of the LTS step by step.

First, the roots (taken from registers, stack and static area) are copied to the trace queues:

Once the initial phase is completed, we can see that two pointers have been recorded in the trace queue Q1. We dequeue the first pointer and mark (using black coloring) the corresponding object. This object holds a pointer to another object in the same region R1; we thus continue tracing along this path to mark the other object (see Figure 4.3).

We now use the second pointer recorded in Q1. The object it points to is marked and scanned, and is found to hold a pointer to an object in R2. This pointer is thus recorded in Q2 as shown in Figure 4.3. Once this is done, we see that Q1 is empty for now, so we continue the process with Q2.

We retrieve each pointer of Q2 and mark the objects, as we did for Q1. We

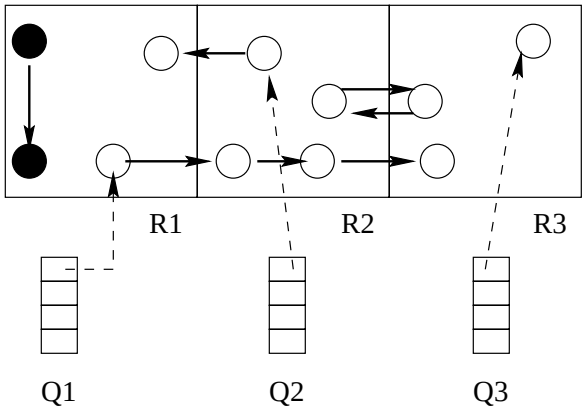


Figure 4.2: *Step 2: Marking in region 1.*

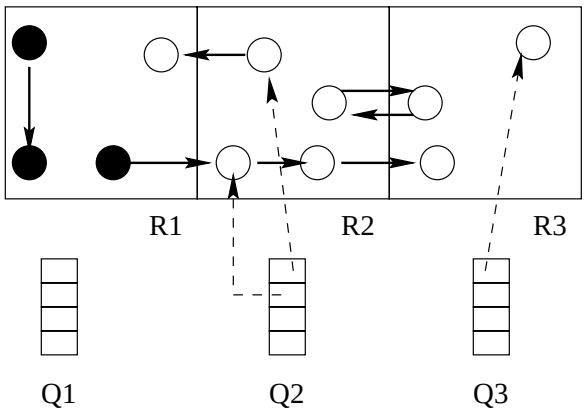


Figure 4.3: *Step 3: Marking in region 1. Pointer outside region 1.*

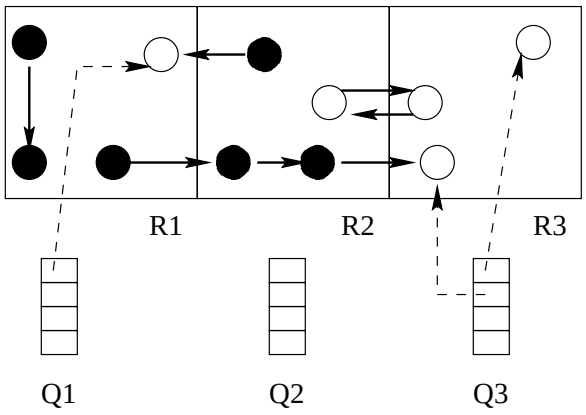


Figure 4.4: *Step 4: Marking in region 2.*

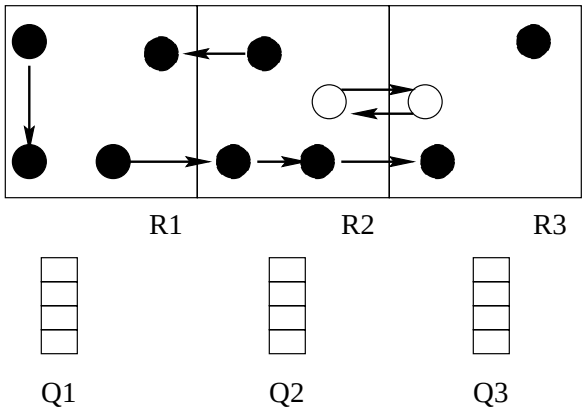


Figure 4.5: *Step 5: Marking in region 3. Complete marking.*

can see in Figure 4.3 that Q1 has been updated because an object of R2 was pointing to R1. A pointer is also added to Q3. Once Q2 is empty, we visit Q3.

Q3 is visited and all reachable objects are marked. It is now empty, and we continue with Q1. Once Q1 has been visited, all queues are empty, and the tracing process is thus over (see Figure 4.3).

We can see that these trace queues act in a manner similar to entry items in a distributed garbage collection environment. Each pointer included in the queue indicates that an object of the region is reachable. All objects identified as live in

a given phase of the LTS (depending on the graph of objects, there may be several phases) will be visited before starting the visit of another region, thus improving locality of treatment. Another analogy can be done: our trace queues are simply remembered sets that will be used to keep track of cross-boundary pointers.

4.4 Interactions

The idea of the LTS comes from our study of GC interactions presented in Chapter 2. In the present case, we considered collectors such as Mark-and-Sweep. We imagine how such a GC could be considered a global strategy and what its components would be.

Coordinating actions on regions of heap is similar to the work of a distributed collector which organizes cooperation between different components of a large distributed memory. We thus consider that each visit of a region is done by a different entity, which relates in some way to the overall tracing activity.

Interactions we find in the LTS are very close to the ones we find in CMM and generational collectors. The object is to define a method to handle cross-boundary pointers. The only difference is that, with these methods, such pointers are a burden. In our case, they are essential to optimize the process.

We rely on the following interactions:

- Do not follow pointers to an outside region.
- Record such "remote" pointers in the queue corresponding to its region.
- Use trace queues as local roots.

It is interesting to observe that these interactions are exactly those relations we can find in any DGC context: remote pointers, proxies, use of proxies as roots. The only difference is that there is only one entity working on all the regions

in sequence. However, we present an extension to a multiprocessor setting in Section 4.7.

4.5 A proof of correctness for the LTS

This informal proof shows that the LTS algorithm rephrases a regular mark phase algorithm which is assumed to be correct. The LTS algorithm has two phases: initial root scan and trace phase. The code for the **initial scan** shows a simple reorganization of the roots. It is correct as it does not lose any root and uses only roots (as guaranteed by the statement `for each root`). Roots are simply dispatched into different data structures called "trace queues".

We now focus on the rest of the algorithm. We first prove its correctness with the assumption that trace queues have infinite sizes (i.e we can never reach a state where a queue is full). Later, we explain the case of finite size queues.

4.5.1 Safety and Completeness properties

Definition

In this section, we start this informal proof of correctness for the LTS by looking at the GC properties of "Safety" and "Completeness". The LTS verifies the *Safety property* in the sense that it will not allow the collector to reclaim live objects. It also verifies the *Completeness property* of garbage collectors by allowing the GC to reclaim all the garbage.

Proof

`while not all queues are empty` guarantees that if a pointer is in a queue, it will be seen eventually. The only place in the code where we dequeue is just before dealing with the reference (possibly marking and scanning the corresponding object). We do not lose references because the only purpose of removing a pointer

from a queue is to visit the referenced object. If we find a valid pointer, there are two cases:

1. *pointer to current region:* recursive call similar to the original algorithm (marking ensures termination).

2. *outside region:* pointer added to a trace queue. We already argued that this is guaranteed to be seen. Thus, safety is ensured.

The LTS is also complete: no garbage object is marked by the process. Indeed, only objects that would be visited by the original algorithm will also be visited by the LTS. This is guaranteed by the fact that we use the same root set as a starting point. Consequently, only pointers found inside an object pointed to by a root or by a reachable object can be recorded in the queues.

4.5.2 Termination

The number of objects is bounded (even if the size of queues is not). Termination occurs when all queues are empty. This is ensured by marking objects as live and not visiting marked objects. Termination occurs for these two reasons: there exists a limited number of objects and these are marked when reachable. The argument is same for the original tracing algorithm and using trace queues does not affect termination. However, we observe that one element is not guaranteed in our description of the algorithm: fairness of visits of queues. The action `choose a trace queue` is not described, because it can be chosen independently of the algorithm (see Section 4.6.4). Trace queues can be organized into a queue, a heap, or any chosen data structure. To guarantee termination, all trace queues must eventually be visited. For example, if trace queues are organized into a queue, then we can think of the trace queues as one big queue, thus ensuring termination.

4.5.3 Argument of correctness for fixed-size queues

So far, we discussed correctness and termination proofs in the case of queues with infinite sizes. We pursue our proof in a more practical scenario and show that correctness and termination are still guaranteed.

Safety: There is no loss of reference. The only case where this might happen with fixed size queues is when a queue is full and the reference we want to add to the queue is lost. However, the algorithm specifies that once the full queue has been emptied, we actually deal with this reference.

Completeness: The size of queues has no impact on the visited objects. We still start from the roots, in the same way the non-LTS algorithm does. Garbage is still guaranteed to be found.

Termination: In the new algorithm, the only obstacle to termination would be to indefinitely cycle through queues (emptying Qa fills Qb up, so we need to empty Qb but it fills Qa up). We guarantee progression by dequeuing **first** (thus changing the state of the queue to "non-full"). Emptying a queue will treat at least one object, and since there are a finite number of objects it is impossible to indefinitely cycle through queues.

4.6 Experiments and results

In this section, we present the results of our experiments with the LTS. This algorithm was implemented within the garbage collector of the Aldor compiler (available at [48]). The GC featured by this compiler is of conservative mark-and-sweep type, because Aldor code compiles down to C, and encourages multi-language programming. We compared the performance of the LTS-based mark phase with that of the traditional mark phase. In our tests, we found that the mark phase was very sensitive to the topology of the graph of objects. If the graph is composed of a single linked list, for example, very few cross-region pointers

will exist, leading to no improvement due to our technique, because paging or caching behavior will be close to optimal. Our experiments focus on improving paging performance, but we also made preliminary tests with cache-conscious configurations. The following paragraphs discuss benchmarks, test results, and consequences of the experiments.

4.6.1 Benchmarks

Finding appropriate benchmarks for garbage collection algorithms is quite difficult. We discovered that GC benchmarks are quite rare and usually focus on small sizes of applications (see [9] and [36]). In particular, we did not find any standard benchmark using heaps larger than the size of physical memory.

A reason for this is that most test applications are typically contained in today's main memory sizes. Programs using swap are long-lived and are often considered inappropriate test cases.

Furthermore, new computing devices such as hand-held computers feature RAM sizes that were common on desktop computers a few years ago. Running today's applications on such platforms proves challenging and requires important resources to be allocated to the development of adapted versions of software. It is conceivable to use virtual memory across networks for these computers, rather than secondary storage in the same machine. This would help to use applications that are difficult to distribute, directly without any update. We propose that the LTS can be used to improve GC timings on very large applications both for workstations – in this case, large means gigabytes – and for hand-held computers – in this case, large means a few hundred megabytes.

In this context, we built a test suite that uses small programs by today's standards of desktop machines but that helps us confirm that the LTS is indeed an appropriate solution for large applications. Note that our tests are obviously not designed to represent real-life programs; rather, we have tuned them to exercise

specific situations that we suspected would help us understand better the limits of the LTS. In particular, we found out that the LTS allows interesting improvements when there is little or no garbage in the heap. Conversely, if few objects are live, the tracing phase is not likely to be improved by the LTS. Section 9.2 describes future experiments that would test the LTS in more realistic contexts.

Test environment

Our tests have been conducted with a Pentium III - 500MHz under Redhat Linux 7.1 (kernel 2.4.2). It is important to note that the disk is using the UDMA66 technology. Machines using UDMA33 or UDMA100 will obviously result in different improvements (tests conducted on a machine using UDMA33 technology showed up to 78% improvement instead of 75% here).

We remark that we are interested in testing our algorithm in an environment that features a heap several times larger than physical memory. Testing very large programs is time-consuming, so we simulated the situation by working with programs using heaps of up to 178MB while using LiLo's ability to set the amount of RAM at 32MB. At boot time, we used the following command: `linux mem=32M`.

Test suite

Most benchmarks published in the literature use small amounts of memory (20MB or 30MB). Consequently, finding appropriate tests for our algorithm was quite complicated. A good test for the LTS is a test that has a memory consumption of several times the amount of RAM available. This is not a sufficient but a necessary condition, because programs using only main memory cannot be improved by the LTS, if optimized for virtual memory behavior.

Programs using a linear graph of objects or featuring very few live objects should not be used with the LTS. The reason is that the LTS improves the way live objects are traced; if there are only a few live objects, there will be no

improvement. However, we will see that the LTS can be used to optimize the garbage collection process when the program generates few or no garbage. If a linear graph of objects is used, the overhead of the algorithm (to maintain trace queues) will not be compensated by the slight improvement we may observe. Consequently, such programs cannot demonstrate the advantage of the LTS.

Favorable tests should feature many data structures that span several regions. The size of the heap should be larger than the amount of available RAM. Finally, there should be little garbage. Our tests have been tailored to reflect these requirements. We made "favorable tests" as well as tests that show an overhead (based on the observations we made above).

Test suite common algorithm

The test program we used as a basis for most of our experiments creates arrays holding lists of integers. Additional code can be activated to run a loop in which new lists are created to replace old ones and/or existing lists are assigned to a different array.

The program's parameters are:

- Location of the arrays
- Behavior of the program (list creation only *or* list creation and list reassignment)
- Size of the problem (i.e amount of memory needed by the program)

The following is the general algorithm of this test program. The different tests will modify it to achieve their goal.

```
NBARRAYS  := <value>
SIZEARRAY := <value>
SIZELISTS := <value>

Allocate NBARRAYS arrays
-- These arrays are independent, there is no structure
-- holding them together.
-- NBARRAYS is simply used to avoid using an actual number.
-- In the actual code, arrays are names A, B, C, D and so on.
-- This is important because we want several root pointers to
-- the graph of computation.

Initialize arrays with empty lists

for idx in 1..SIZEARRAY repeat
   for each array ARR -- won't appear in actual code
      ARR[idx] := nil
   for i in 1..SIZELISTS repeat
      for each array ARR -- won't appear in actual code
         ARR[idx] := cons(RANDOM_VALUE, ARR[idx]);
-- The order of the loops is important.
-- It allows lists to be allocated ``interlaced'' rather
-- than in sequence. This gives a better mix in memory
-- and thus creates many ``remote'' references.

-- Depending on the tests, the code will continue or not.
-- In the following "some" either means "all" or "every other"
for i in 1..SOME_VALUE
   Swap some lists in the array
   Remove some lists from the array
   Create new lists to replace removed lists
```

This test program has the advantage of simplicity. Simplicity is important here, because test suites have to be flexible in order to experiment with several configurations. The test program also uses a large amount of memory very fast, which allows many tests. From this algorithm, we tested 7 scenarios and 1 other test which does not use this code, but tests the behavior of the GC with a linear structure.

4.6.2 Detailed description of the test suite

Test 1: Fit in RAM

RAM used by the test: 6MB - *Total RAM available:* 32MB.

Description.

Parameters SIZELISTS and SIZEARRAYS are chosen so that the graph of objects fits entirely in RAM. Because the tested version of the LTS is configured to improve timings in presence of page swapping, we should observe an overhead due to the management of extra data. This test allows us to quantify this overhead.

Test 2: Linear structure

RAM used by the test: 90MB - *Total RAM available:* 32MB.

Description.

Our algorithm improves paging behavior during tracing. If the graph of objects is structured linearly (i.e. a single linked list), paging behavior is acceptable in a non-LTS environment because the tracing process looks at each object in sequence. In this situation, the only advantage that the LTS could bring is the decrease of the maximum stack size by cutting the recursion and keeping already marked objects in memory to check whether they contain more pointers or not. This can be useful because the non-LTS trace process is depth-first. This means that, once a pointer is found in an object, all children accessible via this pointer should be visited before checking the object again for another pointer – even though there are none. The purpose of this test is to find out if the overhead is compensated by this small speed-up.

The list structure for each element is as follows:

```
[ Pointer to Big Object, Pointer to Next ]
```

This allows to fill the memory pretty quickly as well as maintaining a linear structure. Because the mark phase scans each object starting with the first word, objects will be marked before the next element in the list, thus preserving a linear visit of memory.

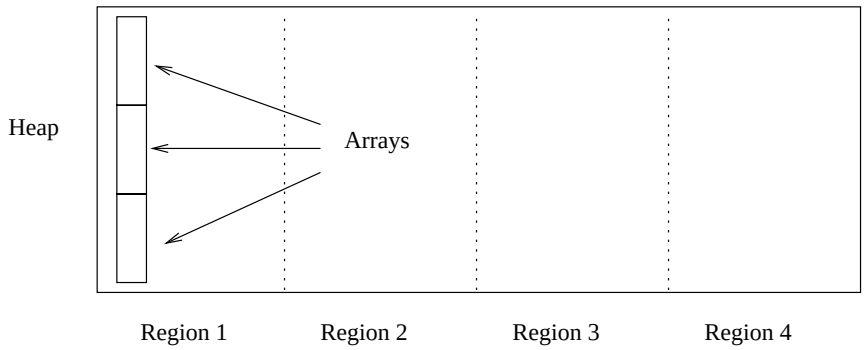
Test 3: Parallel list creation

RAM used by the test: 90MB - *Total RAM available:* 32MB.

Description.

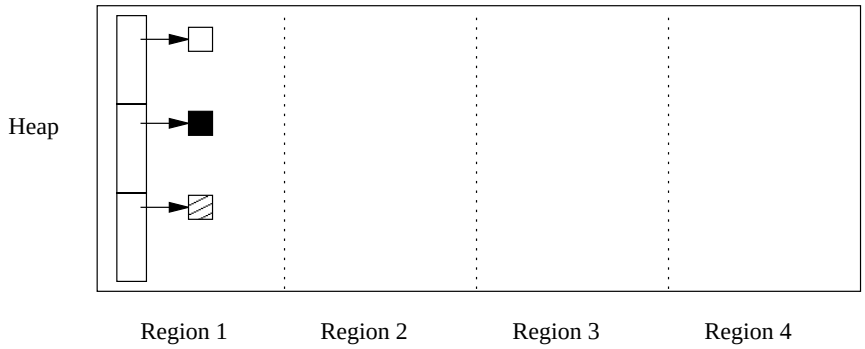
In this test, the last part of the general algorithm is omitted. The program consists in a simple loop of parallel creation of the lists. A picture of memory looks like this:

Step 1: Arrays creation

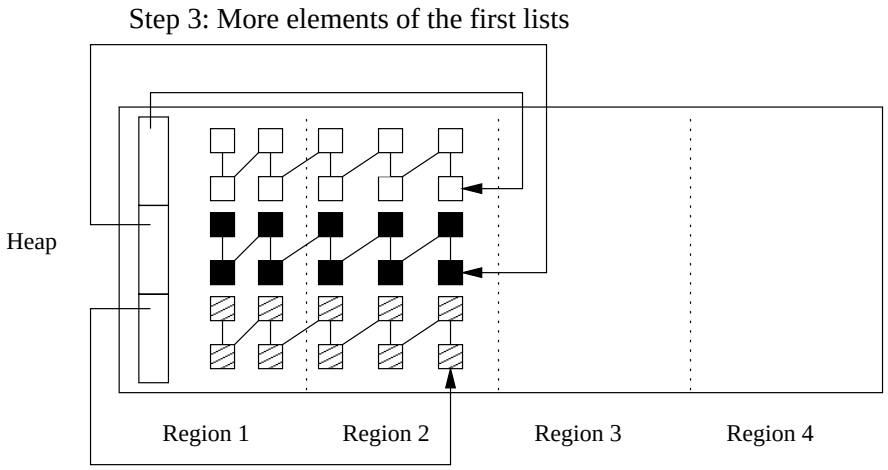


Arrays are created in the first region of the heap because they are created first.

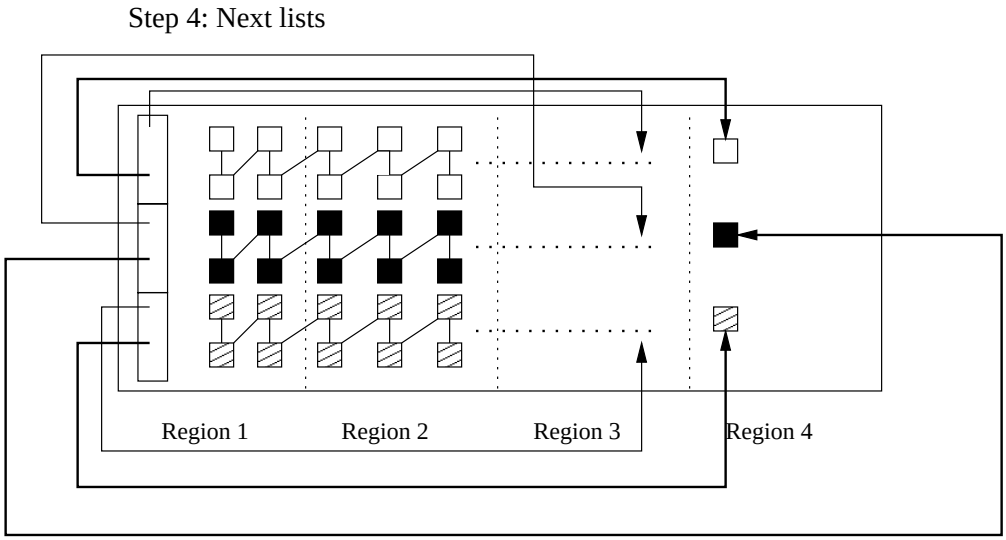
Step 2: First elements of the first lists



By creating the lists interlaced, we see that they will actually be parallel in memory.



When the first region has been filled, the next regions will be used. We can see that the lists are still being created in parallel. Note that the head of the lists are not in the first region (due to the use of 'cons').



Once the first lists have been created, we get to the next index of the arrays and create more lists. The first elements of these lists will be allocated in remote regions.

As can be seen, there is a particularity to this test due to list programming. By using the `cons` primitive, we construct a list that is reversed with respect to allocation order: the first object allocated is the last object of the list.

Test 4: Parallel list creation and use.

RAM used by the test: 178MB - *Total RAM available:* 32MB.

Description.

This is the general algorithm. It shows what happens when the arrays are modified. Once all lists are created, a loop of modifications is started. First, lists are swapped to allow the order of marking to be different from the original created structure. Then, we delete some lists, thus creating a bit of garbage. Finally, we create new lists, still in parallel, to re-populate the arrays. This test shows the behavior of the LTS in a case when there is update of the memory and garbage is created. This is closer to a "real" application.

Test 5: Pointers everywhere!

RAM used by the test: 178MB - *Total RAM available:* 32MB.

Description.

Here, the general algorithm is modified to allocate and populate the arrays differently. Indeed, by allocating all arrays at the beginning, we create a specificity: the first region of memory contains all starting points. This test brings us closer to a real-world application by behaving the following way: create an array, populate it with lists created in parallel, create another array and so on. Arrays which contain root pointers of the lists are now spread all over the heap.

Test 6: Cons-reversed lists.

RAM used by the test: 178MB - *Total RAM available:* 32MB.

Description.

Here, the algorithm of the test 4 is modified to use reversed lists. This is achieved by reversing the list after it has been created. This effectively transforms list connectivity to match the order of list elements to heap addresses. In the first part, regular lists are created and, then, all of them are reversed. The second

part creates and moves lists as in test 4, but all the new lists are also reversed. With this test, we would like to observe the behavior of the algorithm when data structures are organized from the beginning of the heap towards the end.

Test 7: Mixed-order lists.
RAM used by the test: 178MB - *Total RAM available:* 32MB.
Description.
We use the test 6 but modify it to only reverse every other list. Arrays are still located at the beginning of the heap. This mixed order should show the behavior of the LTS in presence of data structures that are accessible from different regions.

Test 8: Mixed-order lists with pointers everywhere.
RAM used by the test: 178MB - *Total RAM available:* 32MB.
Description.
This test is a blend of Test 5 and Test 7. Arrays are spread all over the heap while some lists are in reversed order and other are not. We hope to observe the behavior of the algorithm in presence of a graph of objects evolving in a less obvious manner than previous tests.

4.6.3 Test results: page-level tests

We discuss the results of our tests displayed in Table 4.1. Tests were run three times in a row and numbers shown here are the average of the results we obtained. We observed very little variation in the results (as can be expected because these tests are not random). The table displays test numbers as the header of each column, and the explanation for the header of the rows is as follows:

- *Total app time* is the total application time (including the time taken by the Non-LTS GC).

- *Non-LTS* corresponds to the results obtained with a regular tracing algorithm.
- *LTS-XX-YY* shows the results using the LTS with a region size of XXMB and a total size for the trace queues of YYKB. For example, LTS-4-512 corresponds to a test with a region size of 4MB and trace queues of 512KB.

Results show the time spent by each element (application, non-LTS GC, LTS GC) on each test. They also show the ratio between results with the non-LTS GC and the LTS GC.

Test 1: Fit in RAM

This test illustrates a situation which is not favorable to our configuration. This is explained by the fact that extra maintenance is needed to deal with the queues. Here, we use regions up to 16MB, and we obtain an overhead of up to 25%. However, we note that this is for a total application time of 1 second, and this actual overhead is about 70ms, which is extremely small in most cases. Also, note that this overhead disappears if the GC is configured to use the LTS only when it can be beneficial (a simple criterion is "do not run the LTS when the heap is smaller than available RAM"), see Section 4.6.4 for more details.

Test 2: Linear structure

With a single linked list, there exist very few cross-region pointers. Consequently, it is unjustified to delay recursive marking in this case. This incurs an overhead due to unnecessary operations. We can see in these results that the overhead can be very serious: up to 57%. However, it is equally important to acknowledge the differences in *absolute time* spent in the mark phase, and not only to look at the proportional comparison. Also, we can see that by choosing carefully the size of the region and the queues, this overhead can be brought down to a minimal level (15%).

	(1)	(2)	(3)
Total app time	1 second	1mn48	6mn44
Non-LTS	297 ms	0mn53	4mn34
LTS-4-256	366 ms (*1.23*)	1mn02 (*1.17*)	2mn07 (*0.46*)
LTS-4-512	366 ms (*1.23*)	1mn05 (*1.23*)	2mn08 (*0.47*)
LTS-4-1024	366 ms (*1.23*)	1mn01 (*1.15*)	2mn11 (*0.48*)
LTS-8-256	371 ms (*1.25*)	1mn08 (*1.28*)	2mn11 (*0.48*)
LTS-8-512	371 ms (*1.25*)	1mn12 (*1.36*)	2mn25 (*0.53*)
LTS-8-1024	372 ms (*1.25*)	1mn09 (*1.30*)	2mn21 (*0.51*)
LTS-16-256	371 ms (*1.25*)	1mn23 (*1.57*)	2mn19 (*0.51*)
LTS-16-512	371 ms (*1.25*)	1mn15 (*1.41*)	2mn21 (*0.51*)
LTS-16-1024	371 ms (*1.25*)	1mn20 (*1.51*)	2mn21 (*0.51*)

	(4)	(5)	(6)
Total app time	32mn03	50mn38	28mn17
Non-LTS	20mn22	30mn31	20mn17
LTS-4-256	5mn22 (*0.26*)	12mn35 (*0.41*)	5mn41 (*0.28*)
LTS-4-512	5mn17 (*0.26*)	15mn58 (*0.52*)	5mn42 (*0.28*)
LTS-4-1024	5mn12 (*0.25*)	17mn41 (*0.58*)	5mn47 (*0.28*)
LTS-8-256	5mn26 (*0.27*)	19mn05 (*0.62*)	6mn03 (*0.30*)
LTS-8-512	5mn52 (*0.29*)	17mn05 (*0.56*)	5mn58 (*0.29*)
LTS-8-1024	5mn44 (*0.28*)	17mn19 (*0.57*)	6mn01 (*0.30*)
LTS-16-256	5mn38 (*0.28*)	14mn32 (*0.48*)	6mn08 (*0.30*)
LTS-16-512	5mn50 (*0.29*)	15mn21 (*0.50*)	6mn21 (*0.31*)
LTS-16-1024	5mn55 (*0.29*)	16mn34 (*0.54*)	6mn16 (*0.31*)

	(7)	(8)
Total app time	32mn26	50mn51
Non-LTS	21mn03	31mn09
LTS-4-256	5mn52 (*0.28*)	21mn36 (*0.69*)
LTS-4-512	5mn36 (*0.27*)	21mn51 (*0.70*)
LTS-4-1024	5mn50 (*0.28*)	23mn46 (*0.76*)
LTS-8-256	5mn29 (*0.26*)	22mn01 (*0.71*)
LTS-8-512	5mn32 (*0.26*)	23mn05 (*0.74*)
LTS-8-1024	5mn22 (*0.25*)	23mn54 (*0.77*)
LTS-16-256	5mn38 (*0.27*)	19mn23 (*0.62*)
LTS-16-512	5mn53 (*0.28*)	19mn04 (*0.61*)
LTS-16-1024	6mn00 (*0.28*)	19mn16 (*0.62*)

Table 4.1: *Results of the tests (with ratio LTS/Non-LTS)*

We note that no improvement due to re-examining the object for pointers has been possible. This is because the original GC already has specific code to use tail recursion when the pointer we examine is located at the end of the object.

Test 3: Parallel list creation
This is the first test where we can observe the advantage of using the LTS. As explained before, lists in this example are created in parallel, resulting in many cross-region pointers. While the regular GC recursively marks the objects, abusively swapping pages in, the LTS considers that marking some of these objects can be delayed to improve the paging behavior. This results in an interesting improvement (at least 2mn for a 6mn application).

Test 4, 6, and 7:
These tests give significant results: the structure of the lists in memory (from beginning to end, or end to beginning, or mixed) does not seem to influence the behavior of the tracing process. All three cases are different from *test 3* because they manipulate the lists and thus create a graph that is not so linear. Consequently, the non-LTS tracing process will require paging when visiting the objects, while the LTS controls this activity. We note that – with *test 4* for example – we gain about 14mn on a 32mn application!

Test 5 and 8:
Although speedups are less spectacular, they are still quite interesting: between 38% and 59% for *test 5* and between 23% and 39% for *test 8*. These results can be explained by the fact that "roots" (i.e the arrays that hold the lists) are scattered in memory. Instead of gathering all of them in the same set of pages, the GC has to swap extra pages in to reach these special objects. Even though we improve paging, a slight overhead is incurred due to the configuration of the graph of objects in this application. For *test 8*, it is interesting to observe that the size

of a region appears to play an important role: we get 38% or 39% improvement when we use regions of 16MB.

Conclusion

We observe a loss of performance for applications smaller than main memory. We first note that although the overhead can be up to 57%, it mostly concerns small applications which tend to be fast anyway (on the order of 5 or 6 seconds). Improvements we show can reach 75% for 30 to 50 *minutes* of runtime. Furthermore, in Section 4.6.4, we discuss a technique that can remove the overhead. We simply have to add a dynamic choice between a standard mark phase and an LTS-based mark phase according to the size of memory that is used. As we will see, we can also add criteria to help make better decisions.

These results are quite encouraging and these benchmarks could be reused to test the LTS on hand-held computers (see future directions in Section 9.2). We also observe that the different sizes we chose in our experiments do not change the results dramatically. We find little difference between these values.

4.6.4 Consequences of the experiments

In this section, we discuss the implications of the experimental results on different aspects of the LTS.

Choosing proper values for the parameters

We distinguish two parameters of the LTS: size of regions and size of trace queues. The main issue is probably the choice of the optimal size of "window of collection" (or **region**). A region should be large enough to avoid the need for large trace queues and small enough to avoid thrashing and to keep a reasonable working set. Obviously, there is no best choice, as the size of a region largely depends on the nature of the applications. In our experiments, we found that region sizes of

4MB gave the best results *most of the time*, but this is not always the case (see for example *test 7* and *8*). An interesting future work would be to categorize the applications and see if a pattern emerges. We could then tune the GC with the proper sizes accordingly.

The second parameter is the *size of the trace queues* and is dependent on the size of a region. Both should be chosen at the same time, although the main criterion seems to be the type of application that is run. Our experiments did not show any consistent difference in terms of performance. Although more tests would be required, we can conclude that the size of a region has more importance than the size of trace queues.

Optimal conditions

The LTS appears to perform best with specific patterns of applications. Particularly, this technique provides improvements as soon as virtual memory is used (see *test 1*) *and* the graph of objects is not trivial (see *test 2*). The graph of objects should be complex enough to produce important IO activity within a normal tracing (non-LTS) behavior. Although further experiments should be performed on this topic, it would also appear that the "roots" (i.e. main entry points) of the graph of objects should be located in the same area. Finally, we observe that the LTS algorithm performs better in hard cases of garbage collection where little or no garbage exists.

Loss of performance and workaround

Our algorithm performs better when swap space is involved. Most small programs fit in RAM and, thus, do not need the LTS. In fact, as emphasized by our experiments, our modifications may generate some overhead due to unnecessary actions such as tests to figure out if two objects are in the same region. This is a case where we cannot improve paging behavior because there is none. Unfortunately,

if the LTS is active, it still requires extra data structures to be maintained. These are the reason for the overhead. We propose a solution to avoid this cost.

We simply add a test before starting the tracing process. If the size of the heap is smaller than main memory, then no thrashing can occur and the LTS becomes undesirable. If the heap is larger than main memory, we activate the LTS. Another solution would be to activate the LTS in a cache-oriented configuration when the heap is smaller than main memory. Further experiments are required to understand the pros and cons of such a solution.

We can imagine a more sophisticated version of this solution, which would require dynamic feedback on the application pattern. This topic is outside the scope of this thesis. As future work, we believe that studying tracing behaviors with respect to application patterns could lead to a very efficient tracing process. It would dynamically choose between different techniques according to past GC behaviors and predicted application patterns.

As a first approximation, we implemented a policy to decide, at the beginning of each GC, whether to use the LTS or not. This policy is a simple test comparing the total amount of memory used by the program and the RAM available in the system. If the heap is larger than available RAM, our tracing algorithm is selected, if not the regular tracing method is chosen. This results in removing the overhead due to extra data structures maintenance when the problem fits in RAM.

4.6.5 Cache-level preliminary tests

We started our study of cache behavior in the context of the LTS by using the same tests as for our page-level study. Smaller sizes were chosen, with applications using heaps up to 10MB instead of 128MB, and regions of up to 256KB instead of 16MB or 32MB. We found that the LTS offered very little improvement (a maximum of 2% or 3% on a total application time of 8-10 seconds).

We believe the reason is that our benchmark model is oriented towards improv-

ing paging behavior. The unit it uses is the page while cache behavior depends on cache lines. This leads us, for future tests, to consider an additional model of application to test and use the cache version of the LTS. A cache line is typically small (32 bytes on a Pentium III-500). When it is imported into cache, it is not likely to contain a large number of objects unlike pages which hold 4KB of data on a Pentium III-500. Importing objects as a side-effect of swapping in a page is the main source of improvement with the LTS. In the context of the cache, this advantage disappears.

A cache line is usually reused by the tracing process if it does not contain a pointer that leads to a large data structure. If pointed-to structures are small, a cache line for a given sequence of bytes will not be removed from cache. In this case, finding out if an object (or part of an object) has more pointers becomes very fast. This is the aspect we should capitalize on in our future tests.

These preliminary experiments are interesting because they emphasize the need for several benchmark families. Current uniprocessor benchmarks appear quite limited as they use very little memory and understanding their allocation and mutation is a complicated task. We see here the need for page-level, cache-level, and network-level (for hand-held computers) tests. Also, as we said in the introduction to this chapter that preliminary tests showed that pointer distances in certain applications are rarely greater than 1 or 2 pages; this is not good news for improving cache performance with the LTS. However, we believe that certain applications will not show the same results. That is why a classification of mutators and the development of appropriate benchmarks for each of these families would be useful. Most current benchmarks are derived from actual applications and have thus more value than artificial programs, but new memory management techniques such as the LTS could lead to new mutation patterns. This is why we believe that artificially created tests are of value and help us shape new classes of applications.

4.7 Multiprocessor LTS

The LTS organizes the heap in such a manner that parallelization becomes natural. The heap is divided into regions that can each be mapped to a thread or a processor. In this section, we discuss various aspects of porting the LTS to a multiprocessor environment. Experiments are required to assess the value of the LTS in such a context.

4.7.1 Paging behavior

The optimization discussed in this chapter describes the control of the working set to reduce inefficiencies within the memory hierarchy during tracing. A multiprocessor version should consequently strive to preserve this essential characteristic of the algorithm. We can easily imagine regions to be handled independently and in parallel by several threads, which could be mapped onto several processors. Each thread would scan a group of regions repeatedly and update the different trace queues.

Although performance is likely to improve due to the parallel nature of processing, this technique does not preserve the working set and might lead to bad paging behaviors. When each thread concentrates its work on a particular region of memory, pages of this region are imported. If the region size is chosen to be almost the same as the size of physical memory, each page brought by a thread is likely to result in swapping out a page previously imported by another thread.

This problem could be solved by using region sizes that depend both on the size of physical memory *and* the number of threads. Each thread would then limit its activity to a small region, allowing other threads to import pages in main memory without thrashing. Practical experiments can be made to estimate optimal, or at least acceptable, region sizes and number of threads.

4.7.2 Caching behavior

When the LTS is configured to improve cache behavior, its multiprocessor version can also be optimized. This results from an interesting property of multiprocessor environments. While the heap is common to all processing units, there is actually one cache per processor. This leads us to consider the multiprocessor LTS in the context of several processors rather than several threads. We note, however, that advances in semiconductor technology may result in integrating multiple processors on a single chip, with shared cache mechanisms (see [68] for more details). It is outside the scope of this thesis to study this scenario further.

Our conjecture is that, when several processors are used (in the context of a parallel collector), each of them will use its cache while accessing objects. Cache improvements due to the preservation of the working set will be visible at each processor. An advantage of the LTS is that cache consistency is maintained very simply by assigning a range of regions to each processor. A given processor will never visit an object in a region assigned to another one (except in the case of work stealing as described below, but, in this case, the region can be reassigned to another processor). The only synchronization required is to manage accesses to trace queues.

As for the uniprocessor version, it would be possible to dynamically choose the type of optimization that is required according to various parameters. This illustrates an interesting property of this technique: that is, non-intrusiveness. Using the LTS has no impact whatsoever on allocation or garbage collection. It can be activated or deactivated at any GC phase, because a given tracing process only lives during each GC and is completely independent of previous phases.

4.7.3 Issues

An advantage of the heap organization in the LTS is that it leads to a very simple implementation with minimal synchronization. A parallel LTS requires solutions to two issues: termination and trace queue synchronization.

Termination

A simple idea to discover termination is to maintain a counter of threads going to sleep when no more work is available. If a thread adds a pointer to a queue, it wakes the associated thread up. Termination occurs when the last thread goes to sleep. If a thread appears to be the last one going to sleep, it synchronously checks the counter and the queues to make sure no reference has been left behind (this can happen only if the associated thread is in the process of waking up, but did not update the counter yet).

Work stealing

It is possible that regions are unequally populated. One region may hold a large number of objects, while others contain no or few objects. In this case, most processors "starve" due to the lack of work. Endo [30] proposed a solution in the form of work stealing [16].

The idea of work stealing is quite simple. In our case, each thread maintains a "work queue" associated with each region. It contains pointers that the thread should examine next. Once it is empty, there are two possibilities: 1. the thread goes to sleep until something has been put in its queue, 2. the thread helps other threads by "stealing" pointers from their queues and inserting them into its own queue.

As can be observed, if work-stealing is used as is, the parallel version of the LTS reverts to Endo's technique where several processors scan a single region, involving a synchronization mechanism to access objects. This can be avoided by

making regions small enough to assign several regions to one processor. It would visit each region in a certain order that can be implementation-dependent. When all regions assigned to a processor have been visited, regions – instead of pointers – can be stolen. This requires a simple locking mechanism at the level of regions and not objects anymore. We believe this coarser-grained approach could lead to a significant improvement over Endo's results.

4.8 Related work

This section presents garbage collection techniques – both in uniprocessor and multiprocessor contexts – that we can relate to the LTS.

Generational algorithms divide the heap into "regions" (called generations) to reduce to a minimum the work done by the collector at each call. Because each collection of the nursery is focused in a small area of memory, a side-effect of this organization is to localize data treatment thus reducing page faults and possibly cache misses. Collecting the old generation often involves collecting the entire heap. In this case, the LTS can be used in the same way as with non-generational algorithms. We would then benefit from the use of generations *and* of an improved trace process for the collection of old generations when large heaps are collected.

The observation that collecting the old generation is disruptive has been previously made in MOS [38]. This incremental GC precisely defines the memory block to examine at each call of the collector for the old generation (see Section 2.3.5). It is claimed that this allows a more suitable solution for real-time applications, for example. While the LTS does not solve the problem of real-time applications, we believe it proposes a simple, useful technique to reduce the time spent in collecting the old generation.

Attardi's CMM [3] proposes a heap organization similar to the LTS but for a different purpose. In CMM, each region of the heap is associated with a specific memory management scheme. This allows to potentially use a different GC for

each subheap. Consequences for paging and caching behaviors were not considered. The point of view proposed by the LTS could be used to CMM's advantage. The natural technique used by CMM is to allow collectors to follow pointers even in other subheaps to possibly discover live objects in the current subheap. Such out-of-subheap pointers could be buffered in trace queues to preserve the working set of the collector, which is the job of the LTS.

In [11], Boehm studies a technique to improve caching behavior during tracing of a Mark-and-Sweep garbage collection. It relies on a standard hardware feature (which can be found on Intel and AMD platforms, as well as HP RISC machines) to prefetch "children" objects into the cache when an object is examined. When the object is required by the tracing process, it is already in cache. In comparison, the LTS acts at another level of tracing. Instead of importing objects before they are needed, it keeps objects in cache as much as possible to increase the probability they will be available in case they are needed. It is likely that both techniques could be combined.

[11] also mentions an improvement of the sweep phase, which uses a bitmap to mark dirty pages. A dirty page is a page which contains live objects. When sweeping memory, the GC checks the bitmap before examining a page in detail to rebuild its free list of fixed-sized objects. If the bit is not set, the page can be reclaimed as a whole. The LTS provides a simple solution to store the bitmap: it may be placed in the trace queues. In addition to storing pointers, we maintain a bitmap of pages in the same memory area. This is useful because trace queues are designed to fit in main memory of cache, which also allows fast access to the bitmap. The overhead is of 1 bit per page, that is 512 bytes for a region of 16MB on a Pentium III machine. Preliminary experiments showed up to 55% improvement with the Aldor compiler.

Multiprocessor parallel collectors do not benefit from the same attention as concurrent GCs. However, several techniques were studied: [30] and [87], for

example. An advantage offered by the LTS compared to the parallel collector described by Endo *et al* in [30] is that there is no need for synchronization at the object level. Even though Endo proposed an optimization to access these objects, a synchronization mechanism is still required. This can lead to a costly marking process (although this aspect is not the only issue, as observed in the paper). Instead of asking each processor to trace a given data structure from beginning to the end, the LTS limits the activity of each processor to regions of memory. If a structure steps over a "frontier", the rest of its tracing is handled by another processor. This removes the need for complex synchronization at this level.

We also note that, as mentioned in Section 4.7, the LTS offers a simple organization of the heap to be ported to a parallel configuration of the heap. The advantage is that uniprocessor and multiprocessor environments can use the same memory management technique with very little modification, and featuring interesting performance optimization in the uniprocessor case. If future cache-level experiments show interesting performance improvement, the LTS will also improve cache behavior of its parallel version when several processors are used. This aspect of parallel garbage collection has not been studied in the literature we found, and could trigger interesting future developments.

4.9 Conclusion

In this chapter, we described the Localized Tracing Scheme, a technique to improve performance of tracing activities used in garbage collectors such as Mark-and-Sweep. The LTS localizes the tracing process by dividing the heap into regions. This limits the working set to only one region of the heap rather than the entire heap. If the region can entirely fit in cache, cache misses are reduced. In the same way, if the region is smaller than available RAM, thrashing due to numerous page faults diminishes. Consequently, optimizations can be made at different levels of the memory hierarchy: cache, virtual memory, network.

We implemented this algorithm in Aldor and created a test suite to observe the behavior of the LTS in practice. We obtained up to 75% improvement with a configuration oriented towards virtual memory optimization. Cache-level preliminary experiments did not reveal significant improvements. Cache and cache lines will soon increase in size, Itanium2 [26] already features up to 4MB, and [44] mentions that the next generation ("Madison") could increase this to 6MB. We expect the LTS may display interesting improvements on these platforms.

Finally, we presented an extension work to this algorithm: a multiprocessor version. We observed two axes: (i) independently of any optimization, the organization of the heap in regions results in a natural setting for parallel garbage collections, and (ii) parallel GCs in multiprocessor environments may be improved at cache and virtual memory levels. Future work on the topic will implement this parallel version and compare results with other techniques such as Endo's scalable parallel Mark-and-Sweep [30] or Boehm's "Mostly Parallel GC" [12].

Chapter 5

DGC Design

We observe that practical implementations of distributed garbage collectors are rare, and we believe this is due to the numerous difficulties involved in the creation of a memory management solution. Although garbage collection was introduced to alleviate the burden of memory management, it is now common for application implementers to design certain low-level parts of their system according to the available memory management strategy. This is unfortunate because it shifts the focus back to memory management and away from the logic of the program.

In this chapter, we present a method to choose, organize, and design a distributed garbage collection environment. The resulting DGC is flexible at the levels of design and implementation, and it becomes quite simple to modify the architecture of the system. This design method and its associated models are a direct result of our work on interactions presented in Chapter 2 and Chapter 3. A major aspect described in this thesis is the possibility to *choose* any garbage collection algorithm to act as a local GC for any distributed collector. A product of this work is the natural application of this method to solve the problem of *GC interoperability*. We observe that each node of a distributed system should be allowed to use its own memory management strategy, leading to a heterogeneous system. Our method, based on the organization of interactions between local

collectors and the distributed strategy, naturally allows such a scenario to occur.

We note that we successfully used this method to design and implement (in Java [58]) garbage collectors for the Web environment in Chapter 7 and Chapter 8.

Previous work on the topic proves quite limited. Shapiro *et al* [82] list important elements to consider when designing a DGC, but do not detail any strategy with respect to local collectors. In this chapter, we first review Shapiro's paper in order to give context to our work, and use the results to sketch an overall design process method (Section 5.1). We then detail our strategy to create local GCs, describing a more focused design method and its components (Section 5.2 and Section 5.3). We show that the problem of interoperability of local collectors in a distributed system can be naturally solved with our method (Section 5.4). We also illustrate the work of this chapter by an imaginary but non-trivial example of application for this design method (Section 5.5). Finally, extensions of the model are proposed with a method to handle interoperability of DGCs in a multi-DGC context (Section 5.6).

5.1 Designing DGCs

This section includes a review of a paper from Shapiro *et al.* [82] on designing distributed garbage collectors. From this review, we extrapolate a design process integrating all discussed components, and propose to investigate further the *local GC handling* design space.

Note that this paper uses the same terminology for two concepts: a "space" can be a design component or a node in a distributed system. In this section, we will use "design space" to talk about a design component and "node" or "space" for a node of a distributed system.

5.1.1 Existing work

In Chapter 1, we explained that design methods for DGCs would prove very useful by helping designers to integrate a suitable DGC in their environment rather than using a less-suitable-but-easy-to-integrate DGC.

Shapiro *et al.* [82] lists design elements (denoted as design spaces) and explains what important aspects of distributed collectors should be taken into account at design time. The authors of that paper hope to offer a starting point to help designers build distributed collectors better suited to the context they are used in.

In this section, we present an overview of this paper to lay out the basis for a complete design process. [82] shows the GC problem as a consistency problem. Local collectors are not synchronized which may cause inconsistent local understanding of the distributed graph of objects. The job of the DGC is to cope with this issue. The authors created the Reference Consistency Protocol to solve the problem in their system. Basically, this protocol records references that have been sent and detects unreachable objects according to the local understanding of the graph. From their experience with building distributed collectors (see [81] for example), the authors propose a model of the overall problem. The distributed system is seen as a set of nodes and the distributed collector has to handle three tasks: *tracking references*, *detecting garbage objects*, and *reclaiming garbage objects*. This model is based on opaque addressing, as are most models in distributed garbage collection.

While tracking references and detecting garbage is the responsibility of the distributed collector algorithm, reclaiming garbage is handled at each node by the local GC. In the paper, interactions between GC and DGC are reduced to a simple matter: entry items (or scions) are part of the root set. Indeed, it is stated that no specific work has been done on local GCs. This is the topic we study and we intend to show how the Generic GC could help with the design of DGCs.

We now overview the design components presented in Shapiro et al's paper. Using these design spaces, we propose, in Section 5.1.2, a complete design process to create distributed collectors for a specific environment.

- **Communication semantics**. In this model, communication is mainly related to fault-tolerance. Systems range from completely reliable to completely unreliable. The speed of communication is also mentioned and ranges from instantaneous messages to classical systems. It is claimed that scale is an important aspect to design a distributed system. Indeed, large systems should not assume reliable communications. However, as we noted earlier, the DGC community is divided on this topic. Indeed, some suggest that failures should be handled by the underlying system (see [37] for example), leaving the collector to deal only with memory management considerations.

- **Space failure semantics**. Nodes may feature a complex failure model (different types of failures, detection or not). Once again, this design component is not essential for certain algorithms which assume that once a node fails, the whole application fails. The authors claim that objects that are reachable only from a failed node should be considered garbage. An essential precision is made: nodes that can fail, but can be recovered perfectly, are considered failure-free nodes. This means that objects considered garbage are either really garbage or pointed to by objects on a node that failed and can not be recovered.

- **Reference Consistency Protocol**. This component relates to the local GC and the maintenance of opaque addressing. It is mentioned that this design space includes techniques for propagating reachability by the local GC. However, it seems that the actual topic is the *distributed* algorithm rather than the local one. Basically, the algorithms are of the counting

(DRC, DRL) or tracing (GCW, Cyclic SSPC, Liskov's DGC by Migration, and so on) type. Local solutions to maintain and propagate information are not addressed.

- **Nature of spaces and addressing within a space**. This concerns the underlying environment: regular nodes (processes managing their own memory), subprocesses cooperating within a process, and shared memory among processes or hierarchical organization ("nesting") of nodes. The authors claim that a hierarchical system is the best design for a large-scale system. We find this statement arguable, because such a layout places considerable stress on the top of the hierarchy, and might not be desirable for security or fault-tolerance concerns.

- **Cross-space reference scheme**. This design component focuses on the referencing mechanism used for remote objects. In particular, the authors review one of the algorithms they used in their system [81]. It features an interesting technique: short-cutting of long forwarding chains of stubs and scions (i.e. exit and entry items). When objects migrate and references are copied from one node to another, it is often more efficient to add forwarding pointers (using opaque addressing in the form of scions and stubs) rather than communicating with the node that exports the object. However, this advantage can be lost if the chain of forwarding elements is too long. A mechanism has been created to gradually cut the chains, therefore obtaining both benefits: speed at reference creation *and* at reference usage.

In his thesis [7], Bhudia designs and implements the SSPC distributed collector [81] for the Java programming language [58]. Using the study from Shapiro *et al*, design spaces are detailed with a focus on the Java environment. A new design space is added: **amount of concurrency within a system**. This was necessary to handle garbage collection in presence of threads. This shows that practical

issues bring new concerns even at a design level. When multiple threads (Bhudia mentions *hundreds*) are used for the mutator and collector, the interactions with the DGC component might be differently implemented.

5.1.2 A design process

Although Shapiro's work proves a useful starting point, it does not provide an explicit step-by-step design strategy. In this section, we present a general method – using Shapiro's design spaces – to integrate DGC algorithms into distributed systems.

What to design?

It is important to decide what is the scope of design. Issues can be different when designing a completely general system or a very specific one. For example, designing a DGC system for CORBA [70] (with obviously different stand-alone garbage collectors) does not require the same work as designing a system for the Paraldor project [43] which is a distributed application focused on dealing with large sets of data.

System analysis

The first step is to study the underlying system, and understand its nature and limitations. It is important to know the characteristics of the environment, which will host the DGC and local collectors. To that effect, we use the following design spaces from Shapiro's paper: *communication semantics*, *space failure semantics* and *nature of spaces and addressing within a space*.

The layout of the system should be detailed in this analysis to help identify particular characteristics or behaviors. This may have an influence on the design in that it may reveal possible constraints on parts of the memory management strategy. For example, it may be that some nodes of the distributed system

already have their own memory management strategy. If this strategy is not allowed to be changed, the design should be done accordingly.

At the end of this analysis, we should obtain a general model describing communication protocols and their characteristics (when relevant for memory management such as "possibility to batch messages"), types of failures that memory management should handle (none, loss of messages, space failure, and so on), and the complete layout of the system including existing local memory management mechanisms.

Memory management analysis

The next step is an analysis of the **memory management needs** with respect to the system. We use the overall map of the distributed memory established in the previous phase. We distinguish several scenarios:

- Existing memory management strategy at several nodes. It is strict and can not be changed. This can happen in a distributed environment when certain nodes are required to perform a particular task and the chosen GC allows for the best performance.
- Existing but flexible memory management strategy at several nodes.
- No existing memory management strategy.

Depending on targeted applications *and* constraints of the system, a choice has to be made, which will likely involve compromises (especially if certain nodes use strict memory management strategies). Different application behaviors may require different memory management strategies at each node. If certain requirements or behavioral patterns are known in advance, it is important to specify them in order to choose the strategy accordingly for the rest of the nodes and for the distributed collector. Of course, a "generic" solution can be imposed on the system, but it might not be optimal. We consider it outside the scope of this

thesis to provide a method of memory management needs analysis with respect to application behavior.

We note that, in an environment such as CORBA, supported languages have a non-negligible influence on the distributed memory management design. Indeed, each language (e.g. C++ or Java) features its own ideas about managing its heap. These constitute constraints on the system and should be listed to allow proper choice of a distributed cyclic collector. Furthermore, although not essential, the design space called "Cross-space reference scheme" in Shapiro's study could be useful at this point. This aspect may have consequences on the process of adaptation of various collectors, and on the integration procedure in the global environment.

This analysis should provide a completed map of the system with memory management constraints and requirements at each node of the distributed environment. It should also list desired characteristics of the overall memory management mechanism.

Choosing garbage collectors

Once components, needs and constraints of the system have been listed, we can start **choosing garbage collectors** for each node as well as a distributed collector for the overall system. Analyses described in the previous paragraphs lead this choice. For example, the "failure semantics" design space can be very important in this system, leading to choose SSPC [81]. If distributed garbage cycles are likely to be frequent, one might prefer the cyclic version of SSPC [52] or GCW [51]. The "Reference Consistency Protocol" design space also helps define what classes of DGC (see Section 2.5.1) can be used with respect to the needs we previously defined. This design space should be studied to understand the algorithm of the DGC and its high-level requirements.

Although we provide a tool to help with the decision process, choosing collectors is difficult. It should be interleaved with a *study of feasibility* to adapt stand-alone GCs to work with the DGC, as this activity is likely to be a highly interactive one. Even within the the constraints discovered by the system analysis, several scenarios might be available. Later in this chapter, we will show how it is possible to organize *negotiations* between different collectors in order to help with this aspect of the design (see Section 5.3.5).

Once stand-alone collectors are chosen and the DGC is selected, designers can use the design method, described in the following sections, to verify the feasibility of the chosen architecture. The purpose is to **create a local GC** for each (stand-alone GC, DGC) couple. When there are several such couples, the job can be done by several teams of designers. Once solutions have been found, possible conflicts between the solutions should be evaluated to decide whether or not the system is coherent. If acceptable solutions are found for each scenario and the system is globally usable, then the finalization of the design can begin. If some conflicts exist, *negotiations* can take place. The result will be either acceptable solutions or non-acceptable solutions. In the latter case, it is necessary to identify the problem elements and start again with the process decision for these elements. We note that a repository of scenarios with their solutions (we denote a *scenario* to be a (stand-alone GC, DGC) couple) could be created to allow *reuse* of solutions. This should help reduce resources allocated to the design process.

Eventually, solutions will be selected and we can update the map of the system by attaching created local collectors to nodes and by referring to the chosen DGC.

Integration

Once collectors have been chosen and local GCs have been created, the last step is to finalize global **integration**. Simulations can be conducted using a software tool such as the one we describe in Chapter 8. The purpose is to detect practi-

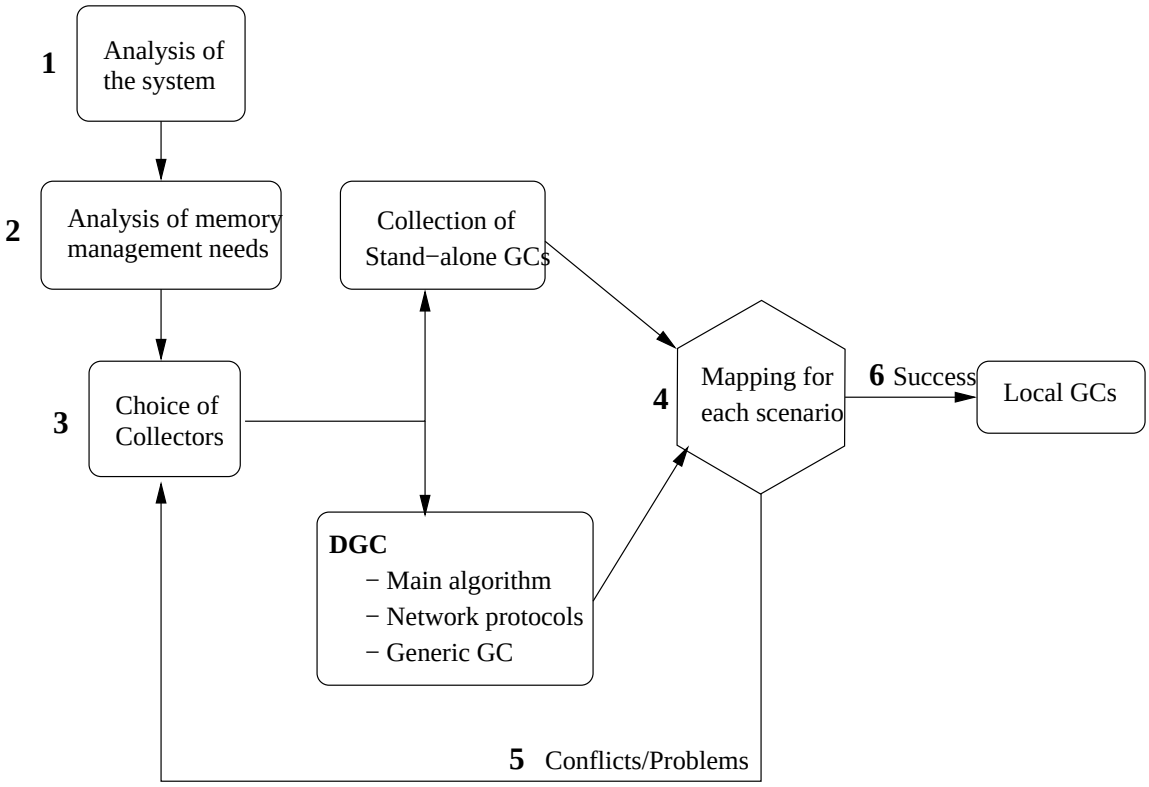


Figure 5.1: Design process for distributed garbage collection.

cal anomalies that could not be found (or have been missed) during the design process. If simulations are acceptable, integration into the target environment can proceed. It is outside the scope of this work to describe such an integration procedure.

5.1.3 Summary

Figure 5.1 summarizes the design process. Here is a description of each step:

1. Analysis of the underlying system.
2. Analysis of the memory management needs at each node and for the overall system.
3. Choice of the collectors (or an other memory management solution).
4. For each (GC, DGC) couple, build a local adaptation.
5. If there are some conflicts, conduct negotiations and/or choose other GCs.
6. Once conflicts are resolved, local collectors are created and ready to be used.

The rest of the chapter describes in details how local collectors are built from each (stand-alone GC, DGC) couple.

5.2 Design models

In this section, we propose templates to model stand-alone, distributed and generic collectors. These templates are designed to support the first part of step 4 of the design process (see Section 5.1.3). For each (stand-alone GC, DGC) couple, we create models for each actor and define interactions between the DGC and its local GCs using the Generic GC template.

This template is a simple extension of the model describing concretely our generic garbage collector (see Chapter 3). This model helps identify relationships between a garbage collection entity and a global strategy. In the present case, we focus on GC/DGC interactions in a distributed environment, and use this point of view to help with the design of distributed memory management solutions.

The templates defined in this section allow us to characterize and classify GC/DGC relationships in order to organize the creation of local GCs more efficiently. Templates do not constitute a solution by themselves, they are simply place-holders for important information. Section 5.3 will show how to use these templates in order to actually create local GCs (this is the second part of step 4 in the design process). Note that these templates contain information that are not essential for describing interactions and characteristics, but these may be useful for designers to help them organize their work. Consequently, we also consider the final users of these templates.

5.2.1 Stand-alone GC

Stand-alone GCs are collectors which are not set up to handle an external world. It is assumed that the only existing roots are *local*. This type of garbage collector includes uniprocessor and multiprocessor GCs. In a distributed environment, local GCs are used on each site and roots are not only local, but also include remote pointers. All distributed collectors encountered so far rely on a stand-alone collector modified to handle interactions related to the distributed context (remote pointers, local actions, and so on). We propose a template to model stand-alone collectors because they are the basis of our work (i.e. to adapt stand-alone GCs into local GCs).

Model

Identification

The name of the GC.

Algorithm class

This is the family this GC belongs to. We currently distinguish the following families: *Reference Counting, Tracing non-copying, Tracing copying, Age-based (including generational), Incremental, Persistent.*

Focus class

Object-focused or *Region-focused* or *Heap-focused*.

See Section 3.1 for a detailed explanation of this class.

Concurrency class

Uniprocessor or *Multiprocessor Parallel* or *Multiprocessor Concurrent*.

From a DGC point of view, there is little difference between multiprocessor and uniprocessor GCs. One could think that different problems would emerge. However, through the Generic GC, the DGC publishes its own "API", its interface. The underlying technique used to accommodate its needs is of little concern at this point. The nature of the multiprocessor GC which can be heap-based (e.g parallel GCs) or region-based (e. g. concurrent GCs where the GC never really stops and there is no need/guarantee for the GC to look at all the memory depending on the algorithm used) will determine how easy it is for this stand-alone GC to become a local GC for this distributed collector. In summary, this class can be useful as an indication of the difficulty level the designers will face.

Description

This describes the main ideas and possibly a general algorithm. It is also possible to include preferred languages of implementation.

Data structures

All data structures needed to implement this GC. In the case of a generic GC, this lists the structures needed by the DGC.

- *Identification.* To identify and refer to the data structure.
- *Description.* To explain the purpose of the data structure and also possible peculiarities, usage, and so on.
- *Contents.* All elements that constitute this data structure.

- *Operations.* Operations allowed on this Data Structure. It should contain: **input** and **output** and **effect on the structure**. This field is optional when operations are accessors only.

Data

A datum is a constant or a global variable used in some algorithms of the GC. Listing this datum is important to get a better understanding of how the GC works. A piece of "Data" is composed of:

- *Identification.* To identify and refer to the datum.
- *Description.* To explain the purpose and use of the datum.
- *Contents.* Show the datum itself possibly with its type and value.

Actions

This describes all actions the GC has to take in order to perform its duties. In the case of a generic GC, this lists the local actions needed by the DGC and, if applicable, an example of a way to do it (the "assumptions" part will then describe what is needed for this solution to work, e.g a tracing GC allows easy information propagation).

- *Identification.* To identify and refer to the algorithm.
- *Description.* Basic, high-level information about the algorithm.
- *Input.* Data or data structures provided to the action.
- *Output.* Results from this action.
- *Side-effects.* Non-direct results from the action.
- *IDs of needed data and data structures.* List of variables, constants and structures used and relied upon by the algorithm.
- *Algorithm.* This is a formal description of the steps required to perform the action. This field is optional.

We observe that this template is similar to (but not exactly the same as) the model described for the Generic GC in Section 3.1. The reason is very simple: a generic collector is a specific vision of a local collector, which is a modified version of a stand-alone collector. The Generic GC model is used to describe interactions between GCs and DGCs, while the model of this section is used to list the characteristics of a stand-alone GC. On the one hand, we study interactions, while, on the other hand, we design a system.

A commented example illustrating the model described in this section is available in Section B.4.

5.2.2 DGC: main algorithm and network protocols

It is not necessary to identify network protocols and the overall DGC algorithm to understand interactions between GC and DGC. However, it may help understand the context in which local GCs have to be designed. In this section, we discuss the basic elements of a DGC and describe network protocols it uses. Once this task is achieved, sufficient information should be available to create the generic GC and start designing local collectors.

Note that this template is not intended as support for a learning tool and it might be difficult to fully understand the DGC only from this model. What we provide here are particular points where the designer's attention should be drawn in order to ease the design process.

Core of the model

Identification

This allows to refer to the DGC.

Algorithm class

We currently distinguish the following families: *DRC, DRL, Augmented DRC/DRL, Age-based, DTD-based.*

Inherits from

A DGC technique can be based on another one. It is especially true with "Augmented" DGCs, which are a combination of DRC/DRL and a specific technique to detect distributed garbage cycles.

Description

Natural language is used in this category to explain the main characteristics of a DGC. It is also possible to list preferred implementation environments.

Reference papers

DGC is still not a widely used technique (this is the reason for this thesis) and close relationship with the research world is needed. That is why listing reference papers may be of benefit.

Assumptions

An assumption contains a **Type** (e.g *message ordering, secure, opaque addressing* or *no loss of messages*) and **Comments** about this assumption (usually precision about the Type chosen).

Conditions of use

This is related to the "preferred implementation environments" of the description. Designers should describe here *situations* in which to use the DGC and the *consequences*. Consequences are usually in terms of number of messages, special extra data structures, more/less bookkeeping, and so on.

Addressed issues

This is quite important because distributed garbage collection is faced with a large number of issues and most DGCs select those questions they will provide an answer to. We distinguish at this time: *Distributed acyclic garbage, safety, completeness, scalability, fault tolerance, speed, ease of integration, network dynamicity.* We note that all

DGCs guarantee safety, but not completeness (DRC and DRL do not reclaim garbage objects that are part of cycles). The first "issue" is addressed by all DGCs.

Supplies

Information supplied to the local collector. Listing those here allows designers to think about both ends for each mapping: what is the input (described by the field *Supplies*) and what is the output (described by the *Generic GC*).

This category is detailed with an **Entity** (e.g entry) and a **Description** (e.g message to decrement counter).

External algorithms

This important category lists algorithms needed by the DGC, but that are not specific to the technique. Such algorithms might be more general and described in other documents. For instance, most DGCs require a distributed termination detection algorithm.

We recommend two possible formats: either the format used to describe an action in a stand-alone GC or the following, which is less formal: **ID**, **Description** and **Purpose**. The field "Description" presents the algorithm needed and any peculiarity. "ID" can be a general term or the name of a specific algorithm. The "purpose" field describes the use of this algorithm in the DGC.

Main algorithm

This is a description, usually in general terms, of the actions of the DGC. This is an important part, because it helps designers understand the "big picture". This description should provide designers with a grasp of the essence of the DGC, which is essential for the creation of local GCs. We do not define any syntax, which should be chosen with respect to usual practice in design teams.

Protocols

We provide a template to list garbage collection-related network messages. "Protocols" describe the actions that trigger a message and, inversely, what actions are invoked upon reception of given messages. As was previously specified, protocols are a vital part of the DGC. They actually drive most of the interactions between DGC and GC, but are not part of them. An accurate identification of those protocols allows for an easier creation of local GCs.

Identification

It will be used to refer to the protocol.

Description

This is used to explain the main characteristics of the protocol using natural language. It should also explain context and purpose of this protocol.

Local actions needed

List of identifiers referring to the DGC's needs described in the Generic GC. These are all the elements needed by the protocol to work.

Local actions triggered

List of identifiers referring to actions described in the Generic GC but having a local effect rather than a distributed one. For example, if a counter on an entry item is decremented, a local action is executed to check if the counter is 0 and take appropriate actions.

Contents

The description itself. Preferably described this way:

- Number of sites involved
- Ordered list of messages
- Messages (see below for format)

A message uses the following format:(From, To, Tag, Params)

'From' is the origin.

'To' is the destination.

'Tag' is the type of message.

'Params' is a list of additional parameters.

Set of sites

S == $\{s_1, s_2, \ldots\}$ or $\{A, B, C, \ldots\}$.

ALL means broadcast.

GROUP means broadcast within the group – if applicable (see [51] for example) – that the "From" site belongs to.

In Section 3.2, we have shown a Generic GC for Liskov's Migration-based DGC. Please refer to Section C.3 where we annotate the complementary part of this DGC, providing a model for the main algorithm and network protocols used by the DGC.

5.2.3 DGC's Generic GC

In a design context, the template chosen in Chapter 3 is minimal. A more comfortable template should also include human-oriented information such as an identification and a description. The resulting template is the same as the template describing a stand-alone collector (detailed in Section 5.2.1). In a design environment, such a model provides a concrete contract between the DGC and stand-alone collectors. It is also used as a guide to build local collectors.

The next section will describe the heart of the chapter: a design method to create local GCs.

5.3 Design method

In this section, we propose a design method to create local collectors based on previously identified scenarios involving stand-alone GCs and a distributed col-

lector. Using the generic garbage collector defined for the DGC, we can adapt a stand-alone GC to comply with the requirements of the DGC, thus generating a local collector for this distributed garbage collector.

Although this method is intended to be used after the collectors have been chosen, we expect these two activities to be interleaved in practice. Indeed, the choice of the collectors will depend on the needs and complexity of the interactions. Although our method makes it easier to design heterogeneous systems, it is obvious that certain scenarios will be harder to solve than others. Depending on the allocated resources, another scenario might be recommended.

5.3.1 Creating local collectors

We distinguish different components in the system: stand-alone GCs, DGC general algorithm, DGC's generic collector and network protocols. The method we are going to explain leads designers step-by-step to modify stand-alone collectors to simulate the behavior described by the generic GC. This will result in a local collector ready to handle the algorithm of the DGC. The feasibility of such a task varies according to the **degree of freedom** allowed to the system designers. If stand-alone GCs cannot be modified, the solution will be difficult to design because, by default, a stand-alone GC is not even aware of the special nature of entities such as entry items and exit items for opaque addressing.

The first step in creating appropriate local collectors is to **identify** the different components of the distributed GC and the stand-alone GCs. To achieve this task, we provide models (see Section 5.2) that list categories of components to take into account. Designers can use these models as templates to create specific descriptions of the collectors they are working with.

We note that the following steps describe explicitly the process to analyze the models and find a solution. Depending on those models and the system to build, these steps might not all be necessary, because some issues are solved

immediately. For example, if the scenario is to map a Mark-and-Sweep algorithm onto the generic GC for the cyclic version of SSPC [52], we know that this stand-alone collector is the better suited GC, and there is no need to prepare for a complicated adaptation process. We would just have to follow the model of the generic GC, and create those elements. However, when a different collector such as MOS [38] is chosen, further work is needed and the following design steps provide some support for this task.

5.3.2 Listing needs into categories

After creating the models, it would be useful to classify the requirements found in the **generic GC model** according to the categories defined here. This would help separate the tasks into two groups: easy-to-do and complex. It is important to keep each need minimal and precise. If the description of a *need* includes too many elements, analysis will be harder and classification into categories may not be accurate. The list of categories we identified includes:

- *Design decisions.* This concerns all the policies that are used by the DGC. A very good example is shown by Liskov's migration-based collector (see [54]). Design decisions have to be made about when to migrate objects, what objects should be migrated, what happens to newly created objects, and so on.

- *Process decisions.* They are similar to design decisions but have to be made at runtime because they depend on dynamic conditions. This is the case with *batching messages* for example. These decisions are usually driven by design decisions. The latter define theoretical limits, while the former make judgment calls according to the current situation.

- *Network Protocol category.* All actions needed to handle DGC-related messages that are sent from one node to another.

- *Propagation category.* Many DGCs require local propagation of values. This can be a date propagation [52] or a mark propagation [51] for example.
- *Property category.* It often happens that some synchronization operations require the computation of a local property. For example, DTD algorithms usually need to know the value of a property to detect termination (for example, the stability property in GCW [51]).
- *Information received from local GC.* This is DGC-related data sent by a node to the others.
- *Information sent to local GC.* This is DGC-related data received by a node from the others.
- *Utilities category.* This has no real impact, we only list here extra useful operations. For example, this can be an operation used to optimize an implementation.

Once we established this classification, we identify complex tasks in order to understand what resources should be allocated to producing a solution. We observe that most DGCs we encountered require propagation of information and property computation. These tasks will usually be the most difficult to handle.

5.3.3 Relations between requirements

Certain relations between requirements listed in the model of the generic GC should be detailed to help designers build their strategy. For example, in the cyclic version of SSPC (see [52]), `localmin` has to be computed *after* local propagation of the dates. In this case, an operation of the *Property* category is performed after one of the *Propagation* category. We expect this particular relation (computation of a property after local propagation of values) to be used in many DGC algorithms. The point is that some operations are dependent on others

and it is important to know those dependencies to facilitate the creation of local collectors.

We list here four possible relations. Of course, this list is not exhaustive. The model is extensible and new relations can be added at a later date.

- *before* and *after*. These are timing relations. An operation must be called before or after some other operation.
- *after an event*. This is different from the relation *after* which actually cares only about timing relations between operations. Here, an event occurred, which triggers the operation. An event can be a function call or the reception of a network message.
- *condition*. A set of conditions must be true to execute an operation. This corresponds to a precondition statement.

Note that we don't focus on dependencies to external or global algorithms, we only care about local aspects. However, we have seen that we pay attention to events such as the reception of a network message. We can argue that this is part of the global algorithm, but it is actually a local event.

5.3.4 Listing potential problems

Listing potential problems is the heart of this process. Indeed, once we know what the problems are, we can actually start finding solutions and creating local collectors. Such problems are derived from the classification described in the previous paragraphs.

We note that it is particularly important to list those DGC needs that potentially "break" the essence of the stand-alone collector. Clearly identifying those problems is the core of this method. Indeed, there is no point in using an incremental collector when the DGC requires a tracing algorithm such as a mark-and-sweep GC, and no effort is made to keep each local collection respect

the property of incrementality. It may be that no solution can be found, in which case either a tracing algorithm replaces the incremental one, or the incremental GC is kept but can not be expected to behave as such when work for the DGC is performed. Another possibility is of course to use another distributed collector.

Once this list is established, designers can begin to work on solutions, look for existing solutions (see Section 9.2 for more details), or try to find other DGC or GC algorithms that fit better while still providing the appropriate features. As we will see in Section 9.2, we hope that repositories of solutions will be created by designers to help the community with this difficult task.

Propagation category

With most distributed collectors, the main problem is the *local propagation of information*. Indeed many cyclic DGCs use this technique to spread and discover reachability status of objects. From roots and entry items to exit items, some data usually has to be propagated by the local GC. This is the case for such collectors as GCW [51], Cyclic SSPC [52], Liskov's DGC [54], and so on.

Stand-alone collectors that belong to the focus category called *heap-focused* should have no problem with such propagation actions. Mark-and-Sweep and Mark-and-Copy algorithms see all reachable objects of the heap at each collection. It thus becomes simple to ensure completeness of information propagation.

Reference counting techniques do not benefit from this *heap-focused* view. They are only concerned with one object at a time and never have a higher-level picture of the local graph of objects. In this case, propagation becomes a real problem. So far, the only solution we can propose is a regularly called external function which performs the propagation. This function is a simulation of a tracing process found in collectors such as mark-and-sweep. When to call this function is a matter of DGC policy, depending on how often the DGC requires such an operation to be performed. Obviously, this solution breaks the essence

of the existing memory management solution (reference counting). Except for distributed reference counting algorithms which do not require any specific local action, stand-alone reference counting collectors are not adapted to work with DGCs.

We note that the solution of an external function is feasible with any stand-alone collector. Although it does not respect the characteristics of those collectors, it is usually the easiest solution to set up.

Region-focused stand-alone GCs also have problems ensuring completeness of propagation. Indeed, collectors such as generational algorithms or incremental ones will either rarely or never have a complete picture of the graph of objects. Generational collectors usually collect the entire heap when looking at the oldest generation, which can provide an opportunity for complete propagation. However, the family of age-based [85] algorithms do not necessarily have a complete view of the local graph of objects.

In the case of MOS [38], which collects only parts of the memory at each call of the GC, we offer several possible solutions. Of course, it is always possible to use a special function called regularly, but this does not respect the nature of this incremental algorithm. Consequently, we need to find a more appropriate way to handle the problem. For more details, please refer to Section 5.5 where we describe how this algorithm can work with the Cyclic SSPC distributed collector. A complete model of MOS can be found in Section B.7.

Property category

Many distributed collectors require certain local properties to be computed. Our survey of DGC techniques revealed that most of them are related to termination detection. Indeed DGC algorithms usually require to be informed of the completion of a phase of information propagation. This is done using a DTD algorithm.

For example, GCW [51] specifies that mark propagation is completed ("the system is globally stable") when each node is locally stable and no message is in-transit. This local *stability* property of each node – necessary to compute the global stability – has to be computed. Another example is the cyclic version of SSPC. It needs the computation of a value called *localmin* that will be used by a time server to compute a global property called *globalmin*.

These properties are usually intimately linked to information propagation. A common relation is *after*: properties are usually computed after local propagation. This allows the algorithm to discover the progress of the garbage cycle detection phase. The difficulty to compute properties is consequently dependent on the complexity of information propagation at each node.

Miscellaneous

- Based on our review of DGC techniques, we remark that most actions of the protocol category are simple operations, usually focused on one object only. It is the case for *increment* and *decrement* messages, for example.

- DMOS [37] is a very special DGC in that it uses cars and trains to organize its graph of objects. These entities are only found in the stand-alone GC called MOS which is also known as the "train algorithm". However, it is possible – although difficult – to adapt other stand-alone GCs. The only requirement is to be able to maintain a proper notion of cars and trains. We expect this to be a non-trivial task with non-copying collectors such as Mark-and-Sweep.

5.3.5 Possible negotiations?

After creating the models, and identifying the different components and potential problems, a solution to the scenario (stand-alone GC, DGC) can be worked out.

At this point, no support tool can replace designers' creativity and this process must be carried out.

We can imagine that certain situations would benefit from modifying the DGC algorithm rather than adapting stand-alone collectors. This strategy can prove problematic: conflicts may occur with the mapping of other stand-alone GCs into local collectors. Indeed, when several stand-alone collectors are used and the mapping of one of them would benefit from a modification of the DGC, one must be aware of the consequences for the other collectors.

This is why we propose a **negotiation** phase between all stand-alone GCs and the DGC. The process is highly iterative and creative, because negotiating for one mapping may change conditions for all others. The idea is to eventually reach an acceptable situation where all important features have been kept and the integration is easier to perform. The resulting system becomes specific to this configuration and is likely to be more efficient.

We propose a simple method to solve this problem. We should point out that this is preliminary work which should be developed further if such a negotiation phase proves useful in practice. For each need or requirement,

1. List stand-alone GCs for which there is a potential adaptation problem.
2. For each of those collectors, list DGC elements that could be modified to ease the adaptation.
3. For each DGC element, explain how modifying this element would affect both the DGC general algorithm and network protocols.
4. Explain how it would affect other collectors and their mapping.
5. Select a solution.

It is interesting to note that these negotiations could occur between GCs and the DGC, but they could also occur with the mutator. Indeed, we can imagine

some negotiation protocol to allow classes of applications to describe their needs and behavior in terms of memory management. Using this information, better suited collectors can be designed, and applications could profit from a clear description of the collectors they are going to work with. Although such applications should not worry about memory management, this happens in practice. Providing clear models of the collectors functions is the minimum support we should be able to provide as garbage collectors implementers.

5.3.6 Mapping template

To conclude, we propose a final template to complement the set of templates we already described. It is not essential to the design process, but could prove helpful for reusing past solutions. This template allows one to organize the different components of the adaptation solutions into one entity. This could benefit designers to quickly find out existing solutions to a given problem. This could also be helpful during negotiations to understand what adaptation is made in what case, and quickly discover conflicts. Here are the components of the model:

Stand-Alone GC Identification

Collector that we want to integrate into the system.

Generic GC's DGC ID

Generic GC to map to.

Description

General information about the context and techniques of adaptation.

Mapping/Creation of extra data structures

It is sometimes necessary to modify existing data structures to achieve the mapping. Object-oriented systems could use inheritance, but it is not always possible. In this model, we describe the resulting data structure:

- *Identification*

  This is useful to refer to the original structure in the model of the Stand-Alone GC, unless of course, it is a new data structure, in which case it will simply be used to refer to it in the algorithms.

- *Description*

  Information about the modification or creation of the structure.

- *Affected algorithms*

  List of algorithms that might be affected by a change of the layout of the data structure.

- *Contents*

  To avoid any misunderstanding, it is recommended to specify the entire structure, even if the modification is minimal.

Specialization of Algorithms

The format to describe these algorithms is left open. However, we propose the following possibilities: (i) the format used in the stand-alone GC template, (ii) a limited version of this template, (iii) a less formal format, describing the mapping rather than showing it. Choosing between these two formats (or even another one) depends on a number of parameters such as context, algorithm to modify, modification to make, designer preference, and so on.

We present here the limited version of the template:

- *Algorithm Identification.* Reference to the original algorithm.
- *Description.* Details about the specialization/modification.
- *New Algorithm*
  - *IDs of needed data and data structures.* List of variables, constants and structures used by the algorithm. We recommend listing all of them, not only the new ones involved in

the modifications of the algorithm. However, data structures involved in the modifications should be marked to preserve a perfect understanding of the process, and to help in case this solution is reused later.

– *Contents.* List of actions to perform. The algorithm itself. As for the data structures, to avoid confusion, the complete algorithm should be restated.

In our experience, it happens that the formats we described so far are not appropriate to properly describe the new technique used to solve the problem. In this case, another template can be used. We propose one here, but other templates could chosen. The following template is used in Section 5.5.

- *Notes:* Introduction to the solution, placing the problem in the context of the local GC.
- *Technique:* Detailed description of how this mapping is done and how it works.
- *Main algorithm:* Overall algorithm which provides a high-level understanding of the solution. This can also be an existing algorithm, modified to fit the requirements.
- *Additional algorithms:* Helper functions for the main algorithm.
- *Comments:* Concluding remarks and precisions about the algorithm.

5.4 GC Interoperability in a distributed environment

It should be obvious by now that Distributed Garbage Collection is a complex problem leading to complex solutions. Numerous aspects have to be taken into

account: performance, scalability, reliability,... We have seen, with the definition of our design method, how to create local GCs fitting DGC requirements. In the literature, distributed collectors often specify that any local collector of a certain type (usually tracing collectors) works with their algorithms and protocols. However, distributed systems will soon expand and distributed systems are likely to include nodes (or sites) with different needs in terms of memory management (performance, reliability, flexibility,...).

As an example, we can imagine that a complex distributed computation has to be performed. According to previous measurements, memory activity has been foreseen, and a group of nodes would work more efficiently with a Mark-and-Sweep algorithm while others may require a generational one. The problem to solve is the following: what distributed collector can be chosen and how can we create local collectors, complying with the same DGC, for two such different stand-alone GCs? This is the issue of **interoperability** of local GCs within a DGC environment.

To fully understand this problem, we carefully consider the nature of a DGC. A distributed garbage collector is a composition of local garbage collectors (one per node) and network protocols to help them communicate. At each node, a local collector is created to handle outside pointers and new data structures such as entry items will also be introduced and should be maintained by the local collector. Network protocols are needed to handle garbage collection messages used to synchronize and maintain knowledge about the distributed graph of objects managed by the system.

Most distributed garbage collectors seem to rely on a particular type of local collector. This is acceptable with our current systems, but may soon be inadequate at the current growth rate. Distributed applications will soon have different memory management needs at different sites of computation. It is very likely that a situation includes DMOS [37] as a distributed collector with 30 sites using a

Mark-and-Sweep algorithm, 40 sites using MOS [38] and 35 sites using explicit management (with malloc and free).

Real-life examples illustrating a need for interoperability are found in many fields. For example, the world of computer algebra usually uses different memory management techniques depending on the situations. During the FRISCO project [69], G.Attardi created CMM [3], which is a technique to organize a heap into subheaps to allow different memory management algorithms to live and work together within the same heap. This was not distributed on a network but the ideas are similar. Also the CORBA environment is an obvious example. It allows language interoperability and, unfortunately, provides only a distributed reference counting mechanism. It would make sense, however, to upgrade this system to allow collectors to interoperate within a cyclic DGC environment as well as allowing programming languages to interoperate in this distributed context.

We propose a solution to the interoperability problem by using our model which semantically separates the concepts of stand-alone collectors and local collectors. A *stand-alone* collector works without any concern or knowledge about a possible distributed environment. A local collector is closely related to the DGC chosen for the distributed environment and is aware of remote pointers. Differentiating these concepts allows us to consider that a DGC does not require a specific local GC, although certain collection algorithm might be better suited to a particular distributed setup.

In this chapter, we created a method to precisely identify the needs and expectations of a distributed garbage collector with respect to its "ideal" local collector. We transform a chosen stand-alone GC into a local collector complying with the needs of this DGC. It is an interesting side benefit that this design method offers a simple solution to make different collectors interoperate within a DGC environment. The idea is simply to study distributed collectors further to identify each one of their characteristics rather than accepting the fact that they work only

with a given local collection algorithm.

In the next section, we illustrate our design method with a complex example. This shows that an heterogeneous context which poses the question of interoperability can be solved without any extra effort, by simply applying our method.

5.5 Example: (MS,MOS,RC) with Cyclic SSPC

This section details an example illustrating the work described in this chapter. We will use a little scenario to explain the design of a DGC using three different types of local collectors.

5.5.1 Preliminary work

We imagine that the first steps in the design of a system have been completed, i.e. that we have analyzed a selection of applications the system will be running, and decided to use three types of stand-alone GCs:

- **Reference Counting** because some nodes will work on non-cyclic local data structures (but it is possible for them to be part of a distributed cyclic structure),
- **Mark-and-Sweep** because some nodes have many fixed size objects and don't have a precise knowledge of the layout of the heap, *and*
- **Mature Object Space** (the train algorithm) because some nodes need an incremental GC, and can provide a precise knowledge of their object types to allow copying.

We now choose a DGC. The environment we work with is a local network with homogeneous machines. There is a small number of nodes and the speed of the network is reasonable. An algorithm migrating potential garbage between nodes (such as Liskov's DGC [54]) was a good candidate, but has finally been

discarded because it was feared that too much work would be needed to adapt the Mark-and-Sweep algorithm to allow moving objects.

The final decision was the **cyclic version of SSPC** [52]. It is simple enough and has already been implemented several times (for example in [7]). We assume that this choice is definitive, which means that there will be no negotiation or other choice. We focus our example on the creation of local GCs, rather than a complete design. Our work on Garbage Collecting the Web (see Chapter 6 and Chapter 7) presents a complete design and implementation work. Here, this example aims at illustrating how chosen stand-alone GCs can be modified to work with a chosen DGC.

The first step is to *adapt our terminology* to the one used in the original description of the algorithm. The SSPC scheme uses the terms of "space" instead of "node", "stub" instead of "exit item" and "scion" instead of "entry item". This is an important task, because we often need to refer to the original description.

We have three couples (stand-alone GC, DGC) to work on:

- (Reference Counting, Cyclic SSPC)
- (Mark-and-Sweep, Cyclic SSPC)
- (MOS, Cyclic SSPC)

Note that we plan to focus on essential points rather than give all the details. Our goal is merely to illustrate the method, not provide a complete solution for this fictional scenario. In a real setting, however, all details should be studied. We show an example of such a work in Chapter 7. Using the method described in Section 5.3.1, we create models for each component. Please note that these models could also be retrieved from a repository or a previous design.

- Reference Counting see Section B.2.
- Mark-and-Sweep see Section B.3.

- Mature Object Space see Section B.7.
- Cyclic SSPC (including protocols, main algorithm and generic GC) see Section C.4.

The next step is to find out our degree of freedom to modify the stand-alone collectors. To concentrate on the method, we allow the designers to modify any part of the collectors as long as their essence is preserved.

In the following steps of this method, we create the mapping models, by sorting into categories the needs expressed in the different generic GCs. Important relations are identified as well.

5.5.2 Listing needs into categories

We list requirements of the Cyclic SSPC into categories. By studying its model (see Section C.4), we observe that the class of algorithms is "Augmented DRL". For us, the designers, this means that there are basic requirements similar to Distributed Reference Listing algorithms. We also note the use of opaque addressing in the form of **stubs** and **scions**. These data structures should thus be added to create a local collector.

Furthermore, we note that a "time server" is required. This is a "special need" that will be dealt with separately. The description also helps us understand the main lines of the algorithm: constantly increasing timestamps are propagated to reachable objects, while garbage objects keep a non-increasing timestamp. We obtain more precision by looking at the main algorithm (see Section C.4). There is a **globalmin** value computed by the time server using **localmin** values received from all the spaces. This globalmin value will be used as a threshold for stubs and scions holding non-increasing timestamps. Once globalmin is larger than these timestamps, corresponding stubs and scions are considered as part of a distributed garbage cycle and can be reclaimed.

We now have a better understanding of the algorithm. There will be values to compute and timestamps to propagate. We also know that there is a special space called **time server** to compute a global time. By looking at the Assumptions and External algorithms, we also see that timestamps are computed using a Lamport Clock [50]. Finally, we know that everything is based on a DRL algorithm.

We summarize our tasks to create local GCs:

- Creation of stubs and scions.
- Setup of network protocols to add and remove remote reference (DRL algorithm).
- Integration of the **localmin** computation.
- Integration of the timestamps propagation.

The section entitled *Protocols* (see Section C.4) lists several types of messages to handle (message identifiers are in capital letters):

- `LOCALMIN` to send the value to the server.
- `STUBDATES` to propagate timestamps to other spaces. We see however in the description that it is also used for the DRL algorithm. This message transmits the list of live stubs so the remote space updates its local view of the connections between remote objects and its objects. This allows to remove those scions that are not referenced anymore.
- `THRESHOLD` is used for some lower level control on the localmin value (to control dates of arrival of messages in transit). This is not detailed in the main algorithm to keep it simple. However, it exists and should be integrated. The purpose of this message is to control timestamps on messages and discard messages that took too much time to arrive and were deemed lost. This is part of the fault-tolerant feature of the DGC.

- `ACK` is an acknowledgment message used by all protocols.

We use the mapping model to categorize the different needs and evaluate the feasibility of their integration.

- Data and Data structures simply have to be added to the GC. We have seven data structures to integrate, no specific problem is foreseen.
- `receive_STUBDATES`, `receive_THRESHOLD` and `receive_ACK` belong to the *Network Protocol category*. These operations can be easily added to the collector.
- The function `local_propag` described in the paper belongs to the *Propagation* category. However, a close study of this algorithm reveals that the function both propagates timestamps and compute the localmin value. This means that `local_propag` belongs both to the *Propagation* and the *Property* categories. We should separate these activities to provide a flexible model to build integration solutions.
- `decrease to` and `increase to` belong to the *Utilities* category and can be easily integrated.

5.5.3 Relations between operations

We now identify possible relations between actions required by the algorithm. We note that `local_propag` should be divided into two blocks of code: local propagation and localmin computation. This results in a first relation:

- `localmin` computed *after* `local propagation`.

We also identify the following, more obvious, relations:

- `receive_STUBDATES` *after an event*: reception of STUBDATES message.
- `receive_THRESHOLD` *after an event*: reception of THRESHOLD message.

- `receive_ACK` *after an event*: reception of ACK message.

The rest of this section details the techniques we used to adapt each stand-alone GC to simulate certain actions required by the generic GC defined by the distributed collector.

5.5.4 Adapting Mark-and-Sweep for Cyclic SSPC

We present a solution to create a local Mark-and-Sweep for the cyclic version of the SSPC distributed collector. The paper presenting the distributed collector specifies that it works best with tracing collectors. Consequently, the process of adaptation is likely to be quite simple because this stand-alone GC is of the tracing family.

For us designers, this means that data structures and algorithms described in the generic GC model for cyclic SSPC (see Section C.4) can be integrated "as is". The mapping model can be seen in Section D.8. As it is mentioned in the description of the mapping, most operations simply have to be integrated, no modification is necessary.

We started with this stand-alone collector to show that little work is required to design a system where the stand-alone GC is the one used by the DGC in its description. The algorithms were created to work with this GC, so in this case the local collector is quite similar to the generic collector. We will see with MOS that this is not always the case. Note, however, that models are still useful to design such a system, because it allows precise identification of all important components of the collectors.

At this point, if our distributed application did not need other stand-alone collectors, we would be done. However, the system includes spaces with special behaviors requiring other memory management techniques. This means that we should carry on with the adaptation work. Handling extra collectors allows us to illustrate our theory about interoperability. We will see, with the other models,

that our semantic separation between stand-alone collectors and local collectors helps us manage such a situation quite naturally. No specific task is required because certain local GCs are different, we simply apply our design method.

5.5.5 Adapting Reference Counting for Cyclic SSPC

Reference Counting algorithms are "object-focused." DGCs, such as the non-cyclic version of SSPC, work well with such collectors because no heap-focused work (e.g. timestamps propagation) is needed. However, the cyclic version of SSPC requires such operations.

Timestamps propagation is essential for this DGC. We can also identify the computation of localmin as being an important component. However, it depends on the local propagation activity (as can be seen from Section 5.5.3 on the relations between requirements), and the problems of their integration will consequently be solved together.

Unfortunately, reference counting algorithms do not provide much support to simulate other collection algorithms in their environment. The only solution we found was to create a separate function called at the discretion of the application. It performs an independent tracing of the heap. See Section D.9 for details.

5.5.6 Adapting Mature Object Space for Cyclic SSPC

MOS is a region-focused GC, and, as such, does not have a complete picture of the heap at each collection. Actually, it never has such a picture, because, unlike generational algorithms, it never collects the whole heap. Once again, we try to solve the problem of propagating timestamps from local roots and scions to stubs. Note that the issue of value propagation is common to many DGCs and the solution we propose here might be reused with little additional work in other distributed collectors (for example, it can be observed that the mapping models are very similar to those established for the GCW distributed collector).

We designed four possible mappings for this scenario. Practical experiments are required to assess the value and feasibility of these mappings, but they show how models can be used to create local collectors. These mappings can be found in Section D.10, Section D.11, Section D.12, and Section D.13. Section D.3, Section D.4, Section D.5, and Section D.6 present the same solutions adapted to the GCW collector.

5.5.7 Conclusion

This example illustrates our technique to design local collectors for distributed collectors. It shows three levels of difficulty: easy (M&S), extremely difficult (RC) and difficult (MOS). These choices were arbitrary, other selections could have been made. For example, we have also created a mapping to adapt a Mark-and-Copy to a Migration-based DGC. Please refer to the appendix for mappings with other DGCs as well.

It should be clear now that our method makes it easier to list all components and understand where design efforts should be focused (in our example, we need to focus on local propagation of data). Interoperability of garbage collectors in a DGC environment is a natural result of the method as illustrated in this section. Except for the extra work resulting from the additions of stand-alone GCs, no specific task should be accomplished in order to handle different types of local GCs in the same system.

We also observe that, although we showed a complex scenario, our method can be used for simpler situations. If our example had only featured one stand-alone GC (Mark-and-Sweep for example), this design technique would still be interesting to analyze the different components of the system and to prepare for an easier implementation. We see such an example in Chapter 7.

5.6 Extensions and DGCs interoperability

We built our method from the viewpoint of flexibility. Obviously, distributed garbage collection is not a solved problem and new techniques are likely to appear that might have proved problematic if we used rigid models and methods. Instead, we are aware that extensions will be needed, and our models will have to integrate new classes of collectors, new relations between components, and so on. Furthermore, new points of view and considerations might also find their way in such a future design environment.

An example of a future extension would be to handle dynamically adaptive collectors. For example CORBA could use an adaptive DGC scheme to dynamically choose the DGC according to stand-alone collectors used in the system. A specific agent could be automatically provide the proper local collector corresponding to the stand-alone GC and the generic GC.

Interoperability mapping techniques can also be modified. In fact, we do hope that new mapping solutions will be designed. We showed possible techniques to handle certain scenarios, but many more situations have to be handled and what we proposed is probably not optimal. Furthermore, solutions may depend on some characteristics of the host environment, which may lead to several "best" solutions for one scenario.

In this section, we illustrate the extension capacity of our method, by extending our environment to include the concept of interoperability between DGCs.

5.6.1 Problem definition

The problem is to allow systems to use several cooperating DGCs within the same distributed application. The motivation to solve such a problem is based on situations similar to the ones which led to investigating several types of stand-alone GCs within a single DGC context. The current state of the art is to use a cyclic DGC in a very specific implementation. Common systems such as Java

RMIs [59] still use a distributed reference counting technique. We believe that this situation will eventually evolve and more sophisticated configurations will be used where the behaviors of distributed applications can be measured and an appropriate set of distributed collectors can be chosen accordingly.

Different situations might require different solutions. When distributed applications reach sizes such as the size of the World Wide Web, it will be time to use appropriate tools such as several distributed collectors cooperating to manage one very large distributed memory.

5.6.2 Existing work

We based this preliminary study on interoperability of DGCs on two previous works. Philippsen [72] presents a specific DGC optimized for reliable networks (especially clusters of workstations). One concern of the paper was also to be able to handle different groups of nodes communicating with different protocols. This discussion led to investigate cooperating DGCs on a network with such different communication protocols. The idea is to consider different areas of communication each with its own technology and to assume that certain nodes can handle several technologies but others ("inside" the area) can not. It is proposed to use bridge objects to handle necessary cooperations of DGCs. In distributed computing, bridge objects already play the role of "glue" between environments, so it is only natural to also use them for DGC purposes.

In comparison to what we propose in this section, Philippsen's paper does not mention different DGC techniques or propose a solution to help them cooperate. With respect to this work, what we propose here would be an orthogonal technique. We are going to present an organizational method to deal with different types of DGCs on the same network. If this network uses several communication technologies, we could integrate Philippsen's technique within our method.

Lang's "Garbage Collecting the World" [51] collector defines groups of nodes

each using the same DGC. This work is focused on studying how local collectors can handle different GCW phases at once. As mentioned earlier (see Section 2.5.4), GCW uses hard and soft marks to reclaim distributed garbage cycles. Marks are located on entry items and exit items and are propagated locally. When dealing with several groups, it is possible that some of them overlap, one node can belong to several groups at the same time. Consequently, it has to handle local propagation of marks for all the DGC phases. This is done by having different flavors of the same "marks"; i.e, the values HARD or SOFT are still used but the marks are augmented with the id of the DGC phase.

In a sense, this technique corresponds to our proposal, but we try to handle different DGCs and not only different instances of the same collector. However, the proposed extension to our models and methods could use Philippsen's work for the low-level layer. Furthermore, Lang, Piquer and Queinnec's idea of groups and their preliminary work on cooperating DGCs will help us define a more general method based on our models.

5.6.3 DGCs can communicate

To help DGCs cooperate, we need to precisely define our environment. We assume a context composed of nodes and groups of nodes created according to a decision process. For example, a group can be set up because its nodes are part of a local network and it is more efficient to design the distributed application to have intense activity where network communications are fast. Another group could be created as a result of a security policy, say that we trust nodes from institution A, B and C, but not D. A group could include A, B and C, while another group will include D. Clearly, defining groups is both a matter of design and policy (even sometimes politics). We assume homogeneous communication technology. However, it would certainly be possible to adapt Philippsen's work to handle heterogeneous communication protocols if need be.

Scenarios

We consider three possible scenarios: disjoint groups, hierarchical groups and overlapping groups.

- **Disjoint groups** are groups with no common node. Two situations apply here. In the first one, very few cooperation exist between those groups, distributed garbage cycles are not likely to occur, and each group has its own DGC; no encompassing collector is needed. Each group should be considered a simple distributed system and DGCs are set up one by one in the same way as described in this chapter. In the second case, many pointers exist between those disjoint groups. This means that we require a DGC federating the distributed collectors assigned to each group. This situation is similar to "hierarchical groups" described in the next paragraph.

- **Hierarchical groups** provide us with a simple model; we have groups and subgroups. If a node belongs to a group A and a group B, it means that $A \in B$, $B \in A$, or $A = B$. In this case, the node will participate both to A's distributed collection and B's distributed collection. We distinguish two possibilities. The first is a strictly hierarchical organization which looks like a tree where internal nodes (there is a terminology conflict here: we mean nodes of a tree) are groups and leafs are nodes (in the sense of network node). The second possibility is an environment where a network node can be the sibling of a group. Those possibilities might not make a real difference, but further studies would be needed to prove it.

- **Overlapping groups** define a model where one or several nodes can be part of several groups. Clearly, hierarchical groups are a subset of this category. There is no real difference between overlapping groups and hierarchical ones in terms of work required from the local GC. Indeed, in both cases, the GC will have to cope with several different DGC algorithms. It might,

however, be useful to understand what is the situation when distributing the computation. Nodes used by several groups might considered important. For example, it could provide more computational power that other nodes. Identifying those nodes could help for certain collection algorithms. For example, they could be used as servers for several DGCs to cooperate.

A solution

Now the central question becomes: how can one local GC serve several DGCs?

We note that, at a design level, a DGC is composed of a main algorithm, network protocols and, last but not least, a generic GC. We also remark that a DGC is a set of local GCs communicating via network protocols. This means that if we want a local GC to work for several DGCs, we need to make local GCs comply with all generic GCs involved. Instead of **one** generic GC, we use several of them. The local GC has to comply with all of them which means that its construction will be more complicated.

A simple solution would be to organize sequential actions to accommodate the DGCs:

```
1. Run the GC.
2. Run actions required by DGC 1.
3. Run actions required by DGC 2.
...
n. Run actions required by DGC n-1.
```

This is easy to implement but not very efficient. We can obviously find a better solution by remembering that the *Generic GC* describes all actions required by the DGC. A generic collector is a contract provided by the distributed collector. The local GC has to fulfill this contract in order to make everything work smoothly. In the present case, we would obtain a more efficient result if, whenever possible, Generic GCs could be **combined**.

Most cyclic distributed collectors use operations belonging to the Propagation class (they usually are hybrid DGCs). In our context, we imagine that several values can be propagated at once, e.g. timestamps for the cyclic version of SSPC and marks for GCW.

We make a similar remark for property class actions. However, the order in which properties are computed is often important (this is the category of "Relations" between local actions). When generic GCs are combined, property class actions will define relational constraints. Furthermore, conflicts may occur, in which case negotiations are necessary. Such negotiations are not studied in this thesis.

We note that there already are DGCs that do not propagate values (usually non-hybrid DGCs). For example, DMOS [37] requires specific data structures rather than specific operations. In this case, for which the modeling process stays the same, the local GC may have to emulate the features of the generic GC, i.e. cars and trains organization, *as well as* propagation of value for some other DGC. This obviously requires creativity from the designers. Switching from a propagation-based local GC to a DMOS-like-based one and back again may prove difficult and has not be been studied here.

This section illustrated the flexibility of our models and methods with a non-trivial example of extension. We hope that this preliminary study of interoperability of distributed collectors offers an interesting starting point for research on this topic.

Chapter 6

W3GC: Garbage Collecting the Web

The World Wide Web can be considered as a very large distributed memory. In this chapter, we explore advantages and challenges of using distributed garbage collection to maintain web link integrity. We show that garbage collection uses a different point of view than other tools, focusing on the link rather than the object. Instead of trying to handle inconsistent situations (where a web document disappeared, for example), we advocate the use of garbage collection to *avoid* such a situation. This requires a slightly different model of web authoring, where authors have to rely on a software tool to create and modify web pages. However, this should be acceptable in many professional contexts where such software tools are already used.

In this chapter, we establish a precise correspondence between models of the Web and primary memory. The purpose of this work is to achieve the design and implementation (described in Chapter 7) of a distributed garbage collection mechanism for the Web. We believe that web management tools could integrate this as a feature to preserve link integrity on well-defined sets of websites. Authors regularly create, modify and manage a very large number of documents.

Users armed with browsers visit these pages and millions of connections are made to web servers. In such an environment, problems are bound to occur. In particular, everyone is familiar with the HTTP "Error 404" which characterizes a connection attempt to a page which no longer exists, or which is unavailable due to a temporary failure on the server side. If this situation happens too often, visitors might not return to the website, which is obviously undesirable. It is also likely that active websites, where many pages are frequently created and modified, contain many documents (images, pdf files, and so on) that are no longer referenced. After a while, many garbage files exist on the filesystem hosting the website, which may prove costly to manage.

If we look at the problem from an abstract perspective, we observe that this question can be reduced to the management of objects connected using a referencing mechanism. We would like to avoid leaks and dangling references. This model is similar to a well-known problem in primary memory, where automatic mechanisms are used to handle references to objects. In this context, we are particularly interested in garbage collection. As we can establish a semantic mapping between notions of memory management and the Web, we can also benefit from using garbage collectors, an area of memory management which profits from more that forty years of experience. We call this web management tool **W3GC**.

Furthermore, garbage collection for the WWW appears to be a natural step in the evolution of memory management. Since 1960, we saw garbage collectors for uniprocessor, multiprocessor, persistent and, finally, distributed systems. Current distributed collectors aim at handling memory management in well-connected, cluster-like environments. However, we believe that applications are likely to evolve towards widely distributed, loosely coupled platforms such as the Web. This projection into the future led us to study the application of garbage collection paradigms for object management in the WWW.

The current situation in this environment is similar to the situation garbage collection designers faced until quite recently. People handle their "own" memory management (although we should say "website management" in this instance, we will see that, conceptually, there is little difference between both worlds). As sites become more complex, so will their management, leading to a growing need for automatic solutions. We already see emerging tools such as automatic link checkers which help make sure websites are safely interconnected. We believe garbage collection is the next logical step for web management/authoring tools.

Currently based on the HyperText Markup Language [23], the World Wide Web starts evolving towards the use of the eXtensible Markup Language [22] with, for example, XHTML [24]. The work we describe in this chapter can be easily extended to take XML-type languages into account. As we will see, there are already studies (e.g. see [56]), which propose techniques to manage XLinks [25] in a more automatic manner. We observe that garbage collection is not mentioned in such studies and we believe this research topic has interesting potential.

The rest of this chapter is organized as follows. Section 6.1 presents possible application areas to explain our motivations for this work. Section 6.2 reviews previous work on maintaining link integrity on the web. Section 6.3 establishes a semantic correspondence between basic concepts found in Web and Memory environments. Section 6.4 discusses general problems brought by the particularities of the Web environment. Section 6.6 details three experiments we performed with W3GC. Finally, Section 6.7 concludes this chapter.

6.1 Motivation

Although the Web environment is usually not considered as a distributed memory environment, there exist similarities as explained in Section 6.3. Web pages are similar to objects in primary memory and URIs (a new version of URLs) can be thought of as pointers. This leads to explore potential benefits memory manage-

ment techniques could bring to the Web. In this section, we propose scenarios where a GC for the Web could be used to help with website maintenance.

6.1.1 A tool for web maintenance

When a web site is minimal, sophisticated management tools are not necessary. However, if the structure of a site is complex, maintenance and development become hard tasks without appropriate support. Web authoring products provide help in the creation of documents, but are usually limited to this activity.

We propose to use a garbage collector – a technique created to maintain links within primary memory environments – to handle problems such as bad links, non-referenced web documents and lack of statistics about web references. Of course, we can find other solutions than garbage collection to handle some of those problems (automatic redirection, Unix's grep, link checkers). Unfortunately, they suffer from lack of flexibility and are not always powerful or even reliable.

Automatic redirection allows to install the equivalent of a forwarding pointer to a new location for a Web document. Unfortunately, this is simply a convenient way to avoid updating referents. It is not an automatic solution to avoid dangling references. Tools such as Unix's *grep* can be also used to check the consistency of a personal website. A problem with this solution is that it does not handle garbage cycles nor scale very well to distributed environments. We found, on websites such as `http://www.softwareqatest.com/qatweb1.html` (valid on 2002/05/31), a list of web management tools sorted by categories. Among those categories, we find some "Load and Performance Test Tools", "Java Test Tools", security test tools and so on. An interesting one is "Link Checkers". A *link checker* follows all the links on a website and reports the bad ones (i.e what are the "dangling pointers" of the site). This tool is very useful, but there is no guarantee about its reliability. Furthermore, it can not be used to discover non-referenced web documents.

Those tools, which may use non-reliable algorithms, could be replaced by well-known GC/DGC algorithms, which have been proved correct and used for many years. Garbage collectors also offer more flexibility. For example, a DGC could be used to find out broken complex structures such as WebRings (see Section 6.4.7). Instead of collecting dead garbage cycles, it would verify the integrity of cyclic structures. Furthermore, we observe that, except for automatic redirection, these tools are not sufficient to **avoid** dangling URIs, only to detect them. Integrating a GC to a web authoring tool would take care of the situation where dangling links would be erroneously inserted in a web page.

6.1.2 Managing stand-alone websites

A garbage collector for the Web would prove useful to check and maintain the consistency and integrity of a stand-alone website. As mentioned earlier, if the site does not contain a lot of material, using a GC might not be beneficial, a simple solution like *grep* would be sufficient. However, it often happens that pages are created and then forgotten, documents are linked and then unlinked, files not published officially were made available to a specific person and left lying around, and so on. In these cases, regularly running a stand-alone garbage collector on the website could be useful to keep it clean. Unlike in a memory environment, a GC for the Web does not have to run often, every month for example. Bad links and other statistics can be reported by this tool. Also, non-referenced pages may be listed, moved or erased (see Section 6.4.6 on design issues).

6.1.3 A company or cooperating organizations

Large companies, such as IBM, or consortiums, such the W3C, have several offices of affiliates around the world. Each site develops its own projects but they usually are related to activities pursued in those other places. Consequently, web sites describing those projects must have many links between them. If each site is

managed by its own GC, it is possible to choose a distributed collector to handle this distributed "memory" and provide a valuable service by ensuring the link integrity of these related websites.

An example of cooperating institutions can be found at L.O.S.T. (see [67]). This is an international group of librarians cooperating via discussion and information sharing. They also seem to work on different projects to maintain and access information. These different projects such as distance education are likely to rely on the web and use a large number of interconnected documents. A web management tool to help maintain link integrity and detect unused documents is essential in such a context.

6.1.4 News sites

News sites such as `www.slashdot.org` or `www.freshmeat.com` are interesting because news are usually displayed in the main page of the site and later moved to an "old" section sorted by date, categories or some other criterion. This specific behavior could be taken into account when designing a collector.

Cycles occur quite often with news websites although they are quite simple. Indeed, those sites usually reference each other by providing links to the latest news appearing on the other sites. Also, we observe that websites referenced by news sites usually provide a link to the article mentioning their URI thus creating a cycle.

6.1.5 Documentation tools

We can also use web-based garbage collector for very specific applications. Garbage collection algorithms do not have to limit their activity to simple memory management – as proved by this work. Many applications require a tool to figure out the relative structure of components and take appropriate actions.

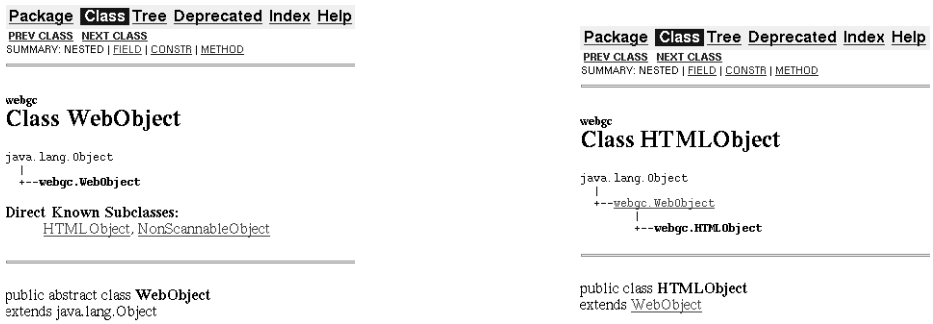


Figure 6.1: *Snapshots of javadoc-generated webpages for WebObject and HTMLObject classes.*

For example, an application for a Web-focused garbage collector would be to support a documentation tool, such as javadoc [57] or aldordoc [19]. These can generate documentation referencing several packages distributed on several machines. This is not exactly the World Wide Web but it is a set of web pages referencing each other. In Figure 6.1, two webpages describing two classes are displayed. We observe that there are many pointers and also cycles (`HTMLObject` inherits from `WebObject`, and the webpage for `WebObject` lists its known subclasses therefore creating a cycle). If these objects are not used anymore, a GC such as a Mark-and-Sweep can detect and report them. This might trigger a decision process about these classes: update or removal. As a development tool, a web-based GC, can thus give pretty good indications about the usage of diverse packages. For example in large size companies, classes are usually developed and shared for internal programming and can be used or not by many programmers. If not, identifying such classes would help to keep high quality components at all time by showing those classes that might need update. For this application, we can use either a stand-alone GC because the sources are all in one place or a DGC because we are in presence of a large project with multiple sites of development.

6.2 Related work

Maintaining links on the Web is a problem that has been the topic of many studies. As we will see, our research is the first to use distributed garbage collection instead of new, non-tested algorithms. This allows applications to maintain links by avoiding inconsistencies rather than repairing them. Furthermore, garbage collectors also detect unused web documents, a feature which is rarely proposed by web management tools. Our novel approach to the problem allows for mutual benefits: evolutions in DGC research help web links maintenance and the Web presents a challenging experimentation platform for DGCs where many issues have to be overcome.

In [45], Kappe presents a scalable solution to maintain link integrity in hypertext systems. A link database is used to record all links (local and remote) coming into and going out of a given server. The approach taken here is to try and handle remote inconsistencies. The technique described in this article allows notification to authors when a link to a remote server has been found broken. It is assumed that maintaining link integrity locally is a trivial (or at least simple) matter. When an object disappears, breaking a link, this information should be propagated to all referents. We observe that this environment uses bidirectional links and object meta-information replicas. Such an organization makes it easier to understand locally the consequences of modifications to a given object. Consequently, a first approach is to engage in "multi-server transaction" where all link databases are updated, but this does not scale well. Instead, this solution allows temporary inconsistencies and uses a flooding algorithm called `p-flood` (based on the flood-d algorithm described in [27]) to propagate information. The environment described here is certainly different from the Web (bidirectional links, links database, and so on). Our solution uses DGCs and thus creates a sort of links database but only for remote links. This database is the set of entry and exit items used for opaque addressing. We note that our point of view is differ-

ent as well. While Kappe's technique handles inconsistent states of the graph of objects, the use of garbage collection prevents this from happening (as long as certain rules about authoring are respected).

In [2], Anderson and Lennon present a document management system used in many contexts. A focus of this system is to maintain link integrity to external servers. The motivation for this work is a set of cooperating websites aimed at providing a distributed learning environment (course and lecture notes, slides, and so on). This system uses proxy objects for all remote documents and transparently intercepts requests. This effectively creates a system similar to distributed systems using opaque addressing. Direct access as currently used on the Web no longer exists. When a document is moved, the link is updated in the proxies, but documents themselves do not need to be modified. As we explain in this chapter, our model using a garbage collector does not rely on any such modification of the Web space. However, should this become widespread, our DGC mechanism would certainly benefit from it, as certain operations would be simplified. A problem addressed by this work is the actions to take when a document disappears and the link is effectively broken. A periodic check has been chosen with a frequency parameter. Both issues addressed here (moved document and deleted document) show a focus on objects rather than references. W3GC takes the opposite approach and strives to avoid broken links by not allowing linked documents to disappear. In this chapter, we describe one exception with expiry dates, where such a situation might occur but in a well-controlled and foreseeable fashion.

The paper [56] describes a technique to detect and repair broken links in an XML environment. XLinks are an extended version of URLs, designed to be used in XML documents for sophisticated linking (e.g. use of attributes, possibility to reference any part of a document, and so on). This work addresses the issue of broken links, and the authors note that XLinks, although sophisticated, are more fragile than simple URLs. XLinks can be easily broken when a document

is modified (by structural reorganization). An XLink can be composed of several links (as explained in [25]). To reduce the number of visited links, a dependency between links is established. A "containing link" is not broken if none of the contained links is broken. This solution allows to skip a large number of traces, and improves tracing speed. Although this solution is attractive, we can not rely on such a structure for websites, and distributed garbage collection proposes a solution for the general problem. However, if XLinks replace traditional `hrefs`, we can imagine integrating this optimization to a web-specific garbage collector in W3GC and activate it when the context allows it. Finally, we point out that this work focuses on the nature of the links and documents and does not address the problem of maintaining links in distributed environments. We believe this paper to be of value in our discussion because it explores an environment which is likely to be a future application context for W3GC.

Garbage collecting the Web has been studied in a preliminary report by Moreau and Gray [64]. However, the focus is on agents rather than garbage collection, which provides low-level support (in the form of a distributed reference listing). The overall architecture uses agents and intercept messages between web browsers and web servers. Three types of agents are used: user, author and administrator. User agents check link integrity and help manage bookmarks. Author agents handle document publishing and inform authors of the usage of the pages. Finally, administrator agents are used for server statistics as well as helping administrators to reorganize web sites while maintaining link integrity. Furthermore, a *publication contract* is set up to specify the lifetime of a page: forever, no guarantee, alive for a period of time and no known period of time but notification of removal. This concept of life expectancy has also been used by the Java RMI's DGC in the form of Leases. Finally, the authors make the interesting claim that weak pointers can be used for specific references such as references taken from search engines.

In comparison, our work concentrates on an abstract view of the Web based purely on garbage collection concepts. We precisely map memory and Web to offer a solution which does not intercept any message and is thus less intrusive. As a result, this approach allowed us to create a tool to study, debug and experiment with garbage collectors (described in Chapter 8). The tool described by Moreau and Gray is certainly more ambitious than ours and handles many elements (such as users' bookmarks). However, we believe that in time our solution could also propose the same functionalities (or at least integrate with their product).

The PerDis system [77] proposes a complete solution to share information over TCP/IP. It uses a distributed persistent object store and provides several tools for application programmers. In [79], Richer and Shapiro use the Web as an experimentation tool to test several allocation strategies in the context of the PerDis system. Using a simulator developed within PerDis, they study the behavior of the Web in different allocation scenarios for two websites.

The PerDis technology embeds objects in the Web, and provides memory management for them. Our work is much more general in that it proposes a solution to handle all web documents as though they were objects in primary memory.

6.3 Web vs Memory: semantic correspondence

In this section, we present a mapping of memory management concepts to notions in the World Wide Web. Understanding the basic concepts is necessary to help us properly design a GC for the Web. Figure 6.1 summarizes this correspondence.

6.3.1 Object

The basic notion we explain is the **object**. We consider a web page to be equivalent to a memory object. In primary memory, objects contain data and pointers,

Memory	**WWW**
Object	Web object (web page usually)
Pointer	URI
Internal pointer	URI with anchor
Mutator	Author or daemon
Garbage	Non-referenced Web object
Allocation	Web object creation (on disk)
Node	Web site
Reference duplication	Insertion of a URI in a Web object
Forwarding pointers	URI redirection (explicit or implicit)
Roots	site entry point and any other "main public file"
Dangling pointers	URIs generating an "Error 404"

Table 6.1: *Summary of semantic correspondence between Memory and Web.*

and can also be associated to methods – code that can be executed. A web **page**, or **document**, is no different (see Figure 6.2). There is data (usually text) and pointers to reference images or other pages. We can also find code in the form of Javascript, for example.

6.3.2 Pointer

In primary memory, the "pointer" is a central notion. A URI corresponds to this concept in the Web. We distinguish three types of pointers:

- An *internal pointer*, in a C environment for example, typically references an element **inside** an object. In the Web, such references to elements inside web documents are denoted by a URI using a "label" with the following syntax: `path#label`. A label is declared within an HTML file by `<a name=label>`.

- A *local pointer* is a pointer within the same node and to another object. URIs express this concept with `"path"` or `"path[#label]"`. This is an address to another document with possibly a reference within the page (i.e

Figure 6.2: *Snapshot of part of the main CSD webpage on 2002/04/26. This page contains text, links, and images.*

an internal pointer). Note that the same local reference syntax may be used for URIs relative to a base URI in remote documents. These should be considered as remote pointers for the purposes of the DGC algorithms. The protocol `file` can be used to guarantee that objects are referenced outside any networked context.

- A *remote pointer* is denoted using a more complicated syntax. In primary memory, we use a node ID and an address which is valid on the remote node. There is an equivalence in the WWW, but it adds the notion of communication protocol. Indeed, within the same page, we can offer access to other pages (or objects) via several different communication protocols. The syntax of a URI is exemplified by `"protocol://server/file[#label]"`. The protocol is usually `http` but could be `ftp`, `https`, and so on. The server corresponds to the remote node and `file[#label]` has already been described.

We note that references can be encoded in different ways in an HTML file or XML document. For example, references to an image can be made using the keyword `<img src=...>`, not only the standard reference syntax: `<a href=...>` normally used to link documents. Although the semantics is not the same (`<img src=...>` is used to specify a link to an image to display in the browser, not to give a link to this file), we need to be aware of this syntax to identify live objects.

The mapping we show here assumes a simplified version of the world of HTML referencing. No complex addressing, such as dynamic creation of addresses, is handled. This is analogous to primary memory GCs which assume no computed pointer values (e.g. offset or XOR-ed). However, the techniques and models that we propose can be adapted to many scenarios. Also note that no reference indirection such as opaque addressing can be used. URIs allow direct access to pages. Proxies can be set up but only to avoid remote network messages. Although it would initially seem to be a good thing, we will see that the lack of opaque addressing forces us to find alternative solutions to implement certain DGCs.

Finally, it is important to observe that HTML provides a precise knowledge of the types of objects. Furthermore, URIs are easily identifiable in HTML files and the access protocol is clearly specified. However, as we will see in Section 6.4, components such as Javascript can make it hard to obtain certain references.

6.3.3 Mutation

A natural idea would be to map the notion of **mutator** to the web server. However, unlike a regular mutator, a web server does not normally modify pages, but only accesses them for reading. A better correspondence would be to associate the notion of mutator with the website author and administrator. This mutator is either a human using an editor or authoring tool, or a script automatically generating and possibly modifying pages.

6.3.4 Allocation

In primary memory, when an object is allocated, different actions are taken: find a suitable free memory cell, invoke a GC if nothing is found, remove the cell from the free store, integrate it with the live objects, update bookkeeping information for the collector, and finally return a pointer to the object. The process is simpler on the Web: a file is created in the appropriate directory.

In practice, disk space is cheap and wasted space does not have to be recovered immediately. Often, new disks are physically added when available space in a file system is low. Furthermore, a person or monitoring process could choose to remove *live* data to recycle disk areas. One might then ask why garbage collecting web sites can be of benefit. The answer lies in the fact that garbage collectors not only free up memory space, they are also tools on which we can rely to avoid and detect dangling pointers thus maintaining a consistent state of the graph of objects.

6.3.5 Roots

In primary memory, roots are pointers stored in reliable locations such as stack, static area and registers. It is assumed that objects referenced by those pointers are used and thus alive. The Web equivalent to the root set would be the web site entry point (e.g. `index.html`). However, any other file can play this role. Indeed, a company could advertise a specific URI for people to visit. In this case, this URI should be considered a root.

6.3.6 Dangling pointers

In a memory environment, a dangling pointer is a reference which points to a memory location which does not contain any object anymore or, worse, which contains an object different from the one the pointer was supposed to reference.

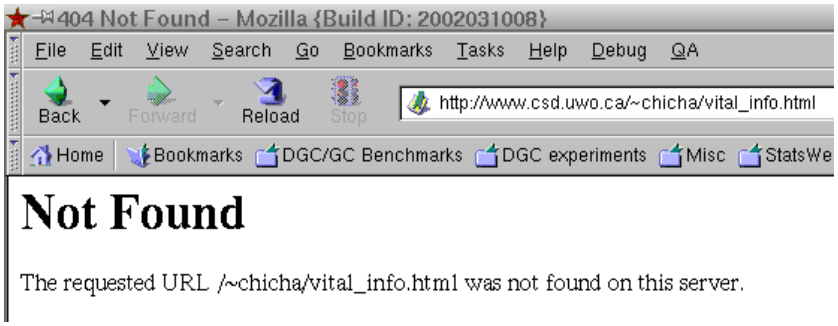


Figure 6.3: *Error 404 example.*

The main purpose of garbage collection is to avoid these dangling pointers. In a Web environment, dangling pointers are very common. Manifestations of those are known as the HTTP "Error 404" which we encounter quite often in practice. Figure 6.3 illustrates what happens when we wish to access important data using a wrong address. This can be the result from a mistake in typing the address or the manifestation of an object which no longer exists, even though its reference was known by another party.

6.3.7 Garbage

An object usually becomes garbage when the mutator no longer has access to it. As mentioned in [64], as soon as a web document is in the appropriate directory, it becomes accessible. However, in practice, this document becomes accessible only when its address is published. Otherwise, we can also consider that an object is reachable as soon as it is allocated in the heap, we just have to "guess" its address in memory.

We deem a web document to be garbage when it is not referenced anymore. However, as we will see in Section 6.4, the actions, taken once a document is found to be garbage, can be quite different from those in a memory environment.

6.3.8 Forwarding pointers

Forwarding pointers are used in memory management to handle relocations of objects. Certain GC algorithms *move* objects in the heap to isolate the garbage. During the relocation, it is necessary to leave forwarding pointers to the new location to help updating objects referring to the moved objects.

The Web counterpart (URI redirection) is used to inform users that a web page has moved. This can be done *manually*, by replacing the old page by a page containing a hyperlink to the new location, with the *assistance of the server*, which uses a special instruction in its configuration file to act as a read-barrier automatically redirecting the HTTP request to new location of the web page, or this can be done with *meta-tags* in the web document itself. For example:

```
<meta http-equiv="refresh" content="2; URL=newpage.html">
```

Figure 6.4 illustrates this setting up an old webpage called `myinteroppage.html` to refer to the new version called `interop.html`. The name of the file was deemed too long and the author decided to change it. In case visitors had a bookmark on this page, a message and a redirection are left to indicate the new location of the page. This is more elegant than simply removing the page and letting the web server return an "error 404" message. We note the use of "Update your bookmarks" which we find quite often on the Web. This is the explicit manner to inform "clients" that an object has moved.

6.3.9 Garbage collectors

Collectors show certain differences when used in a Web environment. However, we will see that DGCs can be reused almost "as is" in both environments, due to the fact that a DGC relies on its local GCs to make the necessary adaptations.

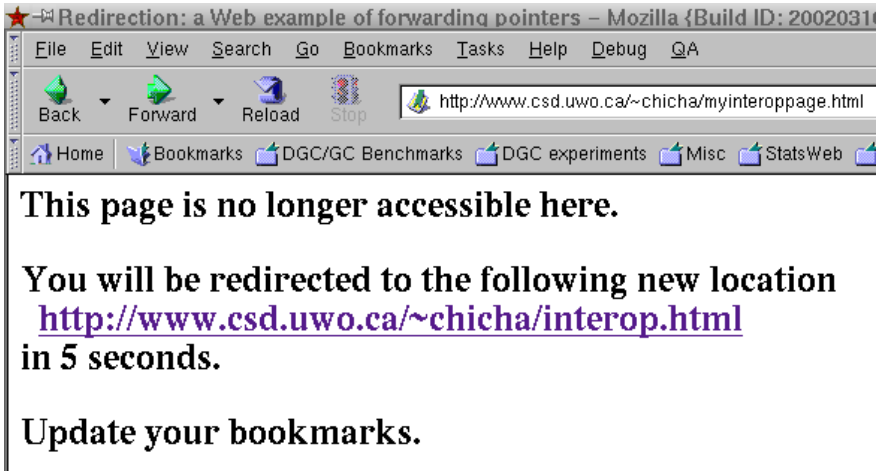


Figure 6.4: *Automatic redirection to the new location of a webpage.*

Uniprocessor GCs

One of the characteristics of uniprocessor GCs is that they stop the mutator when collecting garbage. No further mutation is done until the end of a collection. Obviously, it is not desirable to stop web servers just to figure out what pages are not referenced anymore, although this does occur in practice (we often see that a website has been taken offline for maintenance).

A garbage collector for the web does not have to stop the web server to identify garbage web documents. Indeed, conflicts are not likely to occur because a web server does not mutate web files, it simply reads them. There are several exceptions: CGI scripts, servlets, cookies, and so on. However, we observe that these programs or tools are usually designed to modify very specific parts of a website; local actions created by such entities can be controlled using equivalents of write- or read-barriers.

Note that synchronization between the collector and authoring tools (i.e. the actual mutator) is necessary because both entities might modify the same web documents simultaneously. This allows the semantics of stopping the mutation to be preserved while respecting the constraint of availability for the web server.

Multiprocessor GCs

We distinguish two cases: **parallel** and **concurrent**. Parallel collectors are similar to uniprocessor ones in that they also stop the mutation. On the other hand, a concurrent GC works while mutation is performed. In a Web environment, this means that the GC and authoring tools are working concurrently and access to a file must be synchronized.

We observe that, when websites are maintained manually, any type of collector should be considered concurrent, because mutation can occur at the same time as collection. This shows that web authors should rely on software tools rather than manual modification to avoid complex synchronization mechanisms as well as incorrect maintenance of websites.

DGCs

In both memory and Web environments, DGCs rely on local collectors and network messages. Whereas local garbage collectors have to be adapted to work with the Web, DGCs do not require any modification. This is because a distributed collector does not exist as a separate process but as a collection of local collectors communicating using network protocols defined by the DGC algorithm.

This means that the semantic mapping already exists and is direct because the design (and possibly implementation) can be reused directly.

Properties of DGCs

We describe our understanding of main DGC characteristics when ported to the Web.

- **Safety and completeness**. This corresponds to avoiding "error 404" *as much as possible*. Ensuring complete safety proves extremely difficult in this environment (see Section 6.4). Completeness is ensured when all non-referenced web documents can eventually be found.

- **Fault tolerance**. This notion is similar in both environments. Specific actions have to be taken to help the GC cope with failures (one can use the SSPC distributed collector [81] for example). A minimal alternative is to assume that failure handling is the responsibility of the underlying system.

- **Scalability**. This is a major concern here because the WWW is a very large distributed system, growing every day. Furthermore, algorithms of the distributed reference counting family are no longer suitable because cycles are frequent on the Web, as observed in [64]. Note that hybrid collectors – which use a special layer, such as a distributed mark-and-sweep (see [51]), on top of a DGC of the DRC algorithm – are acceptable because they can reclaim cycles.

6.4 Issues

In this section, we explore questions about using a garbage collector to maintain link integrity on the Web.

6.4.1 Cycles on the Web

In Figure 6.5, we show how cycles might be created in the Web environment. The existence of cycles confirms the need for sophisticated techniques possibly based on distributed garbage collection. Section 6.6 provides experimental results about the number of cycles found in various contexts.

As can be seen in our example here, the situation is similar to a situation we would have in a regular memory environment. To create distributed garbage cycles, we can imagine that John, Patrick and Malcolm do not want to have pages on bikes anymore. Each will remove the link from their "index.html" file. However, John's page will stay alive because Malcolm's site has a reference to

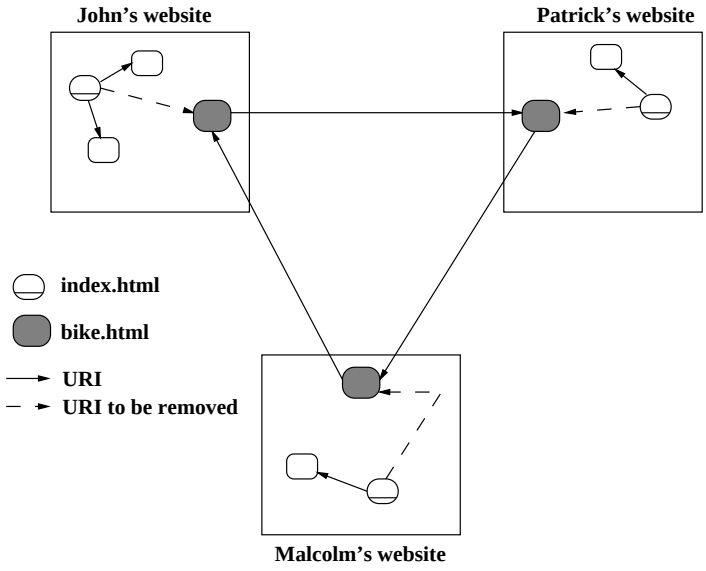


Figure 6.5: *Example of cycle on the Web.*

it and Patrick's web page on bikes was referencing Malcolm's. Finally John had references on Patrick's page.

6.4.2 Scale

The Web contains a very large number of sites, documents and links. Although this constitutes an interesting conceptualization for a large distributed garbage collection problem, distributed collectors are known to work best with tightly coupled distributed systems. We claim that current algorithms for distributed garbage collection can still be used to manage the Web, but within selected contexts.

Currently, garbage collectors can handle local websites and distributed GCs are suitable for intranet configurations such as websites in a university (we found 45 different web servers at the University of Western Ontario). In [2], it is mentioned that their virtual learning center – a set of websites gathering course material and lecture notes – would benefit from a tool to maintain link integrity on

the web. They estimate the number of files to handle at 600,000 files. Compared to the billions of documents we can find on the Web, this may seem small, but this is a practical and perfectly valid application. Although this virtual learning center is handled by a custom tool to maintain link integrity, a DGC-based tool could be used.

Of course, as DGC algorithms evolve, it will be possible to handle larger Web contexts. We believe that the essential ideas of this work will still be valid then. We also note that the Web, formulated as a garbage collection problem, would be able to drive and motivate sophisticated research about various DGC problems.

6.4.3 Failures

As well as being large, the Web is failure-prone. Many servers can go on- and off-line. This is obviously a problem for web maintenance. Our modeling is not invalidated by this issue. We simply rely on DGCs' capacity to handle such problems. Currently, a collector such as SSPC [81] could be chosen to handle failures. However, it is not clear whether this DGC is tolerant enough to support numerous sites that can go temporarily offline.

In our own experiments (see Section 6.6), we chose a simple DGC, without failure handling, in order to illustrate this work. We must point out that this choice was made based solely on the criterion of simplicity. An empirical study would be useful to understand what collector is actually the most suitable in the Web environment.

A specific problem related to scale and failure is how to handle **termination** on such a large and unreliable network. Indeed, many DGCs rely on distributed termination algorithms to complete a phase of distributed garbage cycle detection. This problem shows that the Web can be an interesting platform for distributed garbage collection research, which usually lacks real-world applications. It is outside the scope of this work to provide an answer for the termination problem

in such a large and unreliable environment. However, the DGC called GCW [51], which we chose as an implementation example (see Chapter 7), provide an entity called "groups" (although we did not implement it). These groups are used to limit the scope of action of a DGC by defining sets of nodes to work with. Groups of websites could be defined to reclaim garbage cycles local to a group. Also, when a failure is detected within a group, the corresponding node is excluded from the group. Experiments would be required to understand the exact topology of one particular set of sites and how to divide it into groups according to criteria of performance, reliability, security and so on, but this technique could certainly help in terms of scalability.

6.4.4 Security, trust and permissions

An interesting question about long-run environments such as the Web is that of security and trust among different sites. One might argue that web administrators are not likely to let anyone access their web servers to do garbage collection and discover links to remote objects. We observe that DGCs are composed of local GCs and network protocols for them to communicate. In our context, a local GC is completely controlled by web server administrators. The actual process of document deletion – if any (see Section 6.4.6 below) – is done by the local GC, not by an external actor over which we have no control.

The only external influence applied by the DGC on any of its local GCs is series of coordinating messages for garbage detection, and messages to specify that certain web documents are not accessed anymore from remote documents. We define a DGC server at each node which acts as a "firewall" and protects local documents from being accessed by remote entities (whether they are evil or not). The final decision to delete a document remains with the local administrator.

Questions about external security also trigger the issue of permissions. In an environment with many users (a university department for example), it is likely

that the web administrator does not have the same rights as the overall system administrator. In this case, garbage collecting web files might be cumbersome depending on the permissions allowed to the web administrator. Furthermore, users would not be happy to see "secret" files disappear because they were not linked. This is why, in this case, we recommend that each user runs his/her own local GC, participating or not to the DGC for the website. In Section 7.1.1, we describe other scenarios where a choice should be made between stand-alone GC and DGC according to permissions, topology, and so on.

6.4.5 Ensuring safety and finding references

In the Web environment, safety is very difficult to guarantee when we observe the usual behavior of users. Authors are allowed to directly delete files from their websites' directories, and thus explicitly create a dangling pointer. This is much easier to do than deleting an object in memory. This is why user education might be the main obstacle to this tool. In this environment, people are used to handle things manually and explicitly. It will be hard to convince them to rely on an external tool. In a sense, history is repeating itself: programmers have long hesitated to rely on garbage collectors to manage the memory of their programs.

Safety is also difficult to ensure due to the large number of object types: javascript, java, image files, sound files and so on. Many of those objects can actually hide references in some way and it is hard to find them. In the Web context, many different objects exist, which may or may not contain references. For example, an image file does not usually hold references to other documents, but a text file can contain an address that people can copy&paste into a browser. Unfortunately, as observed in [79], it is hard to find all references to a Web object. The design choice here is to decide what types of references will be supported.

We note that the widely used *assumption* made in the community of garbage collection that a pointer has to be explicit and never "built" is not true in this

```
suffix = new Array('100', '200', '300', '400');
img    = new Image();
for (var i = 0; i < suffix.length; i++)
   img.src = 'images/picture_' + suffix[i] + '.jpg';
```

Figure 6.6: *Javascript example for pointer construction.*

environment. Indeed, it is common – in Javascript code for example – to compose references from different elements. For example, the following piece of code allows to easily pre-load many different pictures:

This shows how we can easily make a reference from several components. For example, the reference
`images/picture_300.jpg` can be built from this code. In a regular context, this reference will not be found.

Many of the constructed references follow simple patterns, often building paths by string concatenation. It may be desirable to handle a well defined class of these but this does not add to the essential ideas of this work. Ensuring complete safety is not feasible at any given moment in time because new types of objects appear every day. A reasonable solution would be for the development of W3GC to closely follow the development of web browsers and creation of new types of web objects. If a new type of object is created to be accessible on the Web, browsers have to be able to handle it. This is done through continuous development of these software tools. Also, we note that many new types are accessed with current browsers via the use of plug-ins. This would be a solution for W3GC, and is certainly a direction to investigate in the future of this thesis.

A (partial) solution

In order to find references created by program (such as Javascript) and to avoid intercepting HTTP messages, we plan to use web server log analysis. These logs record the names of the files that are visited and downloaded. If an HTML file

contains javascript code to display images (for example, the code shown earlier), a web server simply records the list of files that have been downloaded. Analyzing web logs could therefore help us identify such files and deduce their liveness. This solution would help us increase the number of files we recognize as live. However, it is not enough to guarantee safety. A complementary solution would be to use browser (such as Mozilla [71]) and associated plug-in (such as Java [58]) code – which usually handle most MIME types – to examine web documents and "run" code included in those documents. A first approximation for Java applets could be that any file within the same directory as a class file known as live (i.e. called from an HTML page) is considered as live. Both solutions – web logs and browser code – would be added to a production version of the garbage collector to carry out a systematic visit of all files and to obtain completeness of type handling.

6.4.6 Dealing with garbage objects

Primary memory garbage collectors aim at finding and recycling garbage objects in order to free space for new objects. In the Web environment, we can not be that strict. Although removing files that are no longer referenced can be acceptable in certain situations, this is not always the case. Authors of websites may want to keep those files to copy contents, re-link them to certain pages, and so on. Depending on the purpose of the garbage collector, an acceptable action can be to *remove pages* that are not referenced, but also *list them*, *move them* to a special directory or simply *output statistics*.

In the latter configuration, we note that W3GC would then become a *link checker and non-referenced object detector*. These are useful features, but we would like to point out that W3GC can accomplish more than that. When coupled with a web authoring tool, W3GC can be used to **avoid** situations where dangling pointers occur. This results in a more reliable structure than one where the site is checked for bad links regularly.

6.4.7 Practical questions

We discuss specific questions related to the use of a garbage collector to maintain link integrity and discover non-referenced web documents.

Secret files and directories

Many authors have "secret files and directories", i.e. directories that are not linked from their homepages but which are used to communicate files to specific friends and colleagues. These files are live because their addresses have been published or rather sent to a specific person. The purpose is to allow someone to download or view information that is not supposed to be downloaded or seen by anybody else. Although this seems a rather primitive procedure to enforce security of data, this practice is widespread and illustrates two facts:

- A web document can be considered live only if its address has been published, or if it is linked to a document that is known as live.
- The level of security offered by choosing an obscure filename in an opaque directory is sufficient for many purposes.

In order for the garbage collector to recognize them as live, these files should be listed in the GC configuration file. Although this might be thought of as a security hole (exact paths saved in one specific location), this issue is easily solved by protecting the file against read accesses. We distinguish several possibilities in this context, that the GC should be able to handle:

- List all the files to protect.
- List a root for these files if they are supposed to be linked together.
- List directories to protect indicating that this protection is recursive. This would mean that the GC should not bother looking in this directory or any element it contains.

Publication contracts and versions

Currently, it is not realistic to try and maintain link integrity on the whole Web. Certain tools (as seen in Section 6.2) offer solutions to handle certain inconsistent states, but they might require complex solutions, involving databases of references for example. Moreau [64] proposes to use publication contracts. The idea is to maintain an expiry date for each published web document. This date would be integrated to the link itself in order for the client to manage its own updates. When a link has expired or is close to expiry, the client side is aware of it immediately and can take appropriate action. This is an efficient way to deal with loosely coupled websites. This technique is also known in Java RMI in the form of leases.

As mentioned in several works about maintaining link integrity on the web, versions of web documents could be important. They allow a client of a web document to know if a link that was set up on a specific document is still valid according to the referencing context. It is possible that the contents of a web document become unrelated to what it was previously. A simple URL has no way to know about this fact, and a web management tool should at least warn the author of a page that certain links now refer to changed pages. The author can then agree to keep the link or remove it. In a GC context, this problem is orthogonal, because it relates to web contents and semantics of web documents and not their relative structures. However, this feature can easily be integrated into a GC for the Web.

Preserving too many files

A danger of garbage collection on the web and in primary memory is that the definition of garbage object is intimately linked to the notion of reachability (as observed, for example, in [65]). On the Web, this might be a problem because many pages contain links to web documents that are likely to be outdated. Un-

fortunately, because there is a link, the corresponding web document is supposed to live. This is why publication contracts could help balance this situation. Experiments would be required to determine exactly how serious this problem is.

Search engines

Search engines and Web portals such as Google [33] or Yahoo! [93] have a special status among websites. The purpose of their existence is to store the largest possible number of links to web documents (for example, on March 27, 2002, Google had stored 2,073,418,204 references). In this context, many links are broken, and it would not be realistic for websites to take references from such websites into account for the liveness of web documents. Preserving too many documents as explained above is one of the dangers of garbage collection. Search engines are certainly one source for this danger.

Moreau [64] proposed that weak pointers would correspond to references originating from such websites. As we have seen, search engines list a very large number of references to webpages. It would not be reasonable for a garbage collector to use these references as real links. The reason is that once a link is listed, it will live almost forever. After search engines reference the entire Web, no web page would die because there would always be a link to it. It is more practical to consider that these nodes of the distributed system are "special" and should not be considered as holding real references to other sites.

We note that publication contracts would certainly be useful in this context. If all web documents are capable of announcing their own expiry date, search engines would be able to use a much more precise database of references, and caching documents would only be necessary for documents that are susceptible to disappear. It is likely that resources required to run a site such as Google would be much smaller.

WebRings

WebRings are not specific websites but specific structures for a set of websites. A webring organizes related websites in a ring. This organization originates at a particular point in the web space, but this origin has no actual influence on websites belonging to the ring. When the author of a site wants to be part of a webring, he/she has to contact the manager of the ring who will check out the site and return two URIs: one for the predecessor in the ring and one for the successor. The author is asked to add these links to the main page of his/her site. This results in maintaining a cycle of websites. However, this structure is straightforward and, usually, only URLs to the root of websites are inserted on the main page. Consequently, the nature of WebRings should not influence the use of a collector to maintain link integrity, because this ring is likely to never be garbage.

Symbolic links

Symbolic links usually constitute a problem for online link checkers. They can "trick" a link checker into considering documents as new whereas they have already been visited. This happens because link checkers use HTTP requests to visit websites and this protocol does not recognize symbolic links. Certain web servers, such as Apache [31], can disable them, making it easier for a management tool to visit pages. However, there is no guarantee that this is the case and termination is hard to ensure in these conditions. Link checkers usually handle this problem by limiting the depth of the path to a given web document.

An advantage of using a distributed collector which is composed of local collectors is that these local GCs have direct access to the underlying file system (in order to list non-referenced documents). Tracing objects can be done using files instead of HTTP requests and symbolic links can thus be recognized. This allows for computation and analysis of paths on the file system to decide whether to

follow the link of not. Links should be dereferenced, and only real paths should be used.

6.5 Counting cycles

This section proposes a measurement of cycles in a graph. It will be used in Section 6.6 which reports on various experiments we made and statistics we obtained in a Web environment using garbage collectors. Counting live and garbage objects, live and dangling links, and evaluating the time spent doing a GC, proves quite simple as long as we have access to appropriate logs (see Section 7.5.4). Unfortunately, finding the number of cycles in a given context is quite difficult. Such statistics are important because most of the work achieved in the field of automatic memory management relates to the ability to reclaim garbage cycles, distributed or not. Thus, this algorithm allows us to provide a study about the cyclic structure of websites, and a way to measure the effectiveness of W3GC.

The usual measure of elementary cycles, such as the one we can find in [86], is not particularly useful as a property of heap storage as the number may be exponential in the number of objects. What would be more useful is a measure of what proportion of objects are involved in any cycle. We created an algorithm called **Rooted Depth First Partial Cycle Count** (RDFPCC). It relies on the notion of "back pointer" that we define in this section. Evaluating the number of back pointers in the graph of objects gives a lower bound on its number of cycles.

6.5.1 Definition

We consider a single rooted directed graph $G = (V, E)$. We call R the root node of G. We consider an ordering of the nodes defined by the edges of G. This ordering corresponds to the layout of G resulting from a depth-first visit (with no revisit). We define a *back pointer* as an edge between two nodes N_1 – the origin

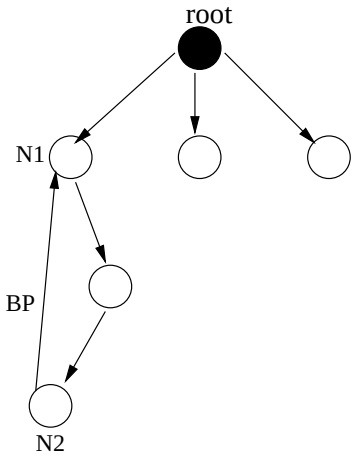


Figure 6.7: *Example of a back pointer. Here, the edge BP is a back pointer from $N2$ to $N1$.*

– and N_2 – the destination – such that N_2 is an ancestor of N_1 in a path leading from R to N_1. Figure 6.7 gives an example of back pointers. In our definition, we also trivially accept as back pointers those edges whose origin and destination are the same node.

6.5.2 Algorithm

The following function counts back pointers from a given node. It relies on the root node of the graph (declared a global variable).

```
global root: Node

countBackPointers(n: Node): integer {
  nbptrs := 0
  n.marked := true
  k := number of n's outgoing edges
  for i in 1..k
    a := edge i in n
    if (a.dest = root) nbptrs++
    else
      if (a.dest.marked = false)
        nbptrs += countBackPointers(a.dest)
```

```
      else
        if (a.dest is ancestor of n)
          nbptrs++
   return nbptrs
}

RDFPCC(): integer {
  return countBackPointers(root)
}
```

We note that `countBackPointers` relies on a function to find out if a given node is an ancestor of the node currently examined. The algorithm is not shown here, but it is quite simple. With each node, we associate a marker which indicates what branch is being visited and we retrace this path from the root. Another solution could use a stack of visited objects.

6.5.3 Properties

Property 1: maximum number of back pointers

We assume that $G = (V, E)$ (with $|V| = n$) does not contain multiple edges with the same origin and destination:

$\neg\exists e_1 \in E, e_2 \in E, e_1 \neq e_2$ and e_1.origin $= e_2$.origin and e_1.dest $= e_2$.dest

This assumption implies that only one loop edge is allowed per node and we therefore have a maximum number of n such edges. Back pointers are defined as edges towards ancestors. We obtain the maximum number of back pointers in G if it is linear and each node is the origin for the maximum number of edges towards ancestors. At each node, such a number is the number of ancestors + 1 (for the self-referencing edge). Consequently, the maximum number of back pointers in a graph depends on the number n of nodes and is evaluated as follows: $1+2+\ldots+n$. This gives us a maximum number of $n(n+1)/2$.

Property 2: maximum number of back pointers to the root

In the Web environment, it is interesting to count the number of back pointers to the root file. The root file is never going to disappear unless the whole site disappears. Consequently, such cycles can be safely assumed to be live. We suspect that the number of such pointers is often close to the maximum, which is the number of nodes: n.

Property 3: lower bound on the number of cycles

Counting the number of back pointers allows us to provide a lower bound on the number of cycles in G. Indeed, each such pointer is part of at least one cycle, because it links a node (N_1) to one of its known ancestors (N_2): there is a path from N_2 to N_1 and an edge from N_1 to N_2. We note that back pointers can be part of several cycles. Because each back pointer is part of at least one cycle, the number of back pointers in G indicates the minimum number of cycles we can find in G.

Property 4: the order of visit of the graph is important

One drawback of counting cycles using back pointers is that the result might be different depending on the order of visit of the graph. However, *Property 3* guarantees that, whatever the visiting order, the result we obtain is a lower bound on the total number of cycles.

Figure 6.8 illustrates this situation. From the root, we have two possible ways to visit the graph, using edge A_1 or edge A_2 first. If we use A_1, we find *one* back pointer and report at least one cycle. If we start from A_2, we find *two* back pointers and report at least two cycles. Note that this happens because from node A, we have two ways to reach B and B points to A. So if B's pointers are going up instead of down, we have two back pointers instead of one.

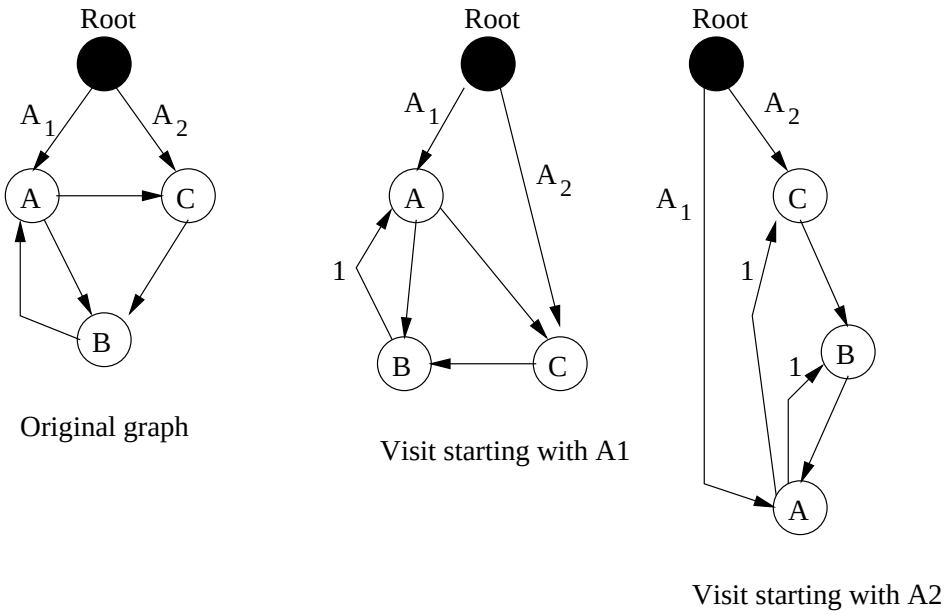


Figure 6.8: The order of visit is important.

Property 5: all cycles defined by back pointers are elementary cycles

In [86], Tarjan proposes an algorithm to find and output all elementary cycles (i.e. any vertex in such a cycle is present only once) in a directed graph (see also [40]). We decided not to use this method, because the complexity depends on the number of cycles and, in certain situations, this number might be *exponential* in the number of vertices. We are concerned with identifying how many objects are part of a garbage cycle, not how many garbage cycles they reside in.

We observe that all cycles found by our algorithm are elementary. This is useful to know because information we gather on the connectivity of a website should not be redundant. In particular, in our experimental results, we discuss the number of objects involved in cycles.

We can prove that all cycles we find using the RDFPCC algorithm are *elementary*. Objects that we visit are marked, when we find an edge between two vertices (A and B) and the destination vertex (B) has already been marked, there

are two possible cases:

1. This edge is not a back pointer. In this case, there is no cycle and we keep searching.

2. B is an ancestor. In this case, the cycle we count includes a path from B to A and an edge from A to B. Note that even if there are several paths from B to A and some are not elementary, we still count only one cycle. Consequently, only the elementary path is effectively considered and counted.

6.6 Experiments

We made four experimentations with our GC implementations (see Chapter 7):

- a statistical analysis of 40 websites hosted at the University of Western Ontario (Section 6.6.1).
- stand-alone garbage collection of several user websites (Section 6.6.2).
- garbage collection of a local copy of the Java API documentation directory (Section 6.6.3).
- DGC for a network of user websites in the Computer Science department at the University of Western Ontario (Section 6.6.4).

Obviously, while interesting, these tests are only a first analysis of the value of our distributed garbage collector implementation. Future work on this topic would be to integrate our garbage collection mechanism to a web-authoring tool and let it be used for a year. Two scenarios can be prepared: authors at different web domains (for example, www.csd.uwo.ca and www.apmaths.uwo.ca) or authors within a same web domain. The latter is also useful to test a distributed collector, because we can disregard the fact that all files are likely to be accessible on a single filesystem.

6.6.1 UWO websites

We studied 44 websites hosted at the University of Western Ontario. The statistics we obtained show the context in which a garbage collector would work. It is not an actual experiment of the garbage collectors we have written. We believe this data to be relevant in the sense that it provides information about the scale we can expect in an actual environment. Other studies can be found in the literature such as the ones reviewed in [18]. Besides [18], we found that most statistical studies about the Web focus on the dynamic nature of a website (frequency of changes, for example) as well as usage patterns by visitors (what link do they activate first, what pages are most visited, and so on). Few studies concentrate on the structure of a website, and we believe that the experiments described in this chapter are a useful contribution to the area of empirical studies of websites. Also, to our knowledge, the experiments described in the next sections (stand-alone GC, Java, and DGC) are the first studies to report information about "garbage pages".

This statistical experiment was done from a user's point of view. We did not have access to the underlying filesystems hosting the websites. Consequently, we do not provide any statistics about garbage objects. However, we provide information about the number of live objects we encountered as well as dangling links. In [79], the authors provided information about allocation patterns using the structure of two websites holding about 3000 and 9000 objects. We believe that our tests show a broader, and thus more interesting, context with statistical data on websites using between 2 and 40456 web objects.

We first describe the extent of our implementation. We could handle:

- `.html`, `.htm` and `.cfm` files. `.cfm` files are similar to HTML.
- `https` using Java 1.4's URL library.
- Redirections using META tags.

- Diverse details such as the syntax of a URL like `http:www.uwo.ca` (the usual `//` after `http:` are omitted), case-insensitive protocol part of a URL (`Http` is the same as `hTtP`).
- `href` in any tag, in `<a ...>` of course but also in `<area ...>` and others.
- errors such as non-closed strings.

We also had to skip complicated elements. This statistics-gathering program is still a prototype and we could not spend too many resources implementing a complete piece of software. We skip:

- URLs using network protocols other than `http` or `https` (for example, `ftp`, `telnet`, `gopher`, and so on).
- stand-alone anchors denoted by `#` should be skipped because they indicate internal pointers. There is no use for these in our program.
- URLs containing `cgi-bin` or a question mark. This latter is problematic because this does not allow us to call PhP code, but this was necessary to avoid infinite recursions with specific URLs (in particular, when we encounter directory listings).
- Simple infinite recursion (e.g. `/foo/foo/foo/foo/...`). This happens because we manipulate URLs as character strings and such recursive paths are not recognized, we thus set up a simple pattern recognition routine.
- References found in comments. This is a good thing because a comment should not be considered at all.

Finally, there are elements that we could not handle because their complexity required more time to treat than the resulting data would merit for a first investigation.

- Complex recursive patterns in URLs (e.g. /foo/bar/foo1/foo/bar/foo1/...). Our solution is similar to those of most link checkers. We limit the depth of paths. This can be dealt with differently (as we did in the previous experiments) if we have access to the underlying filesystem and can recognize symbolic links for example.

- Languages such as PhP, CGI, Perl, Javascript. See Section 6.4.5 for more explanation.

Observations and results

We first remark that our results are indicative but cannot be 100% accurate. Indeed, the HyperText Transfer Protocol [35] does not recognize symbolic links in the underlying filesystem. This results in possible problems such as infinite loops or duplicated paths. Using a garbage collector will not result in the same problem because it works on files and not webpages. Consequently, symbolic links can be recognized.

Depending on the location from which we ran our statistics program, we could obtain different results. This is because certain pages are accessible only under specific conditions (e. g. only accessible from clients located in UWO, accessible with a password, and so on).

These two observations are important because they show that online link checkers are not sufficient to handle websites. Local programs such as garbage collectors are more appropriate to the task of checking the consistency of a website.

We encountered three websites that were exclusively using PhP. We had to skip those sites, because we skip URLs containing a question mark (as explained above). We also skipped one website using Perl and one website using CGI scripts. One other website was using many CGI scripts, but we could get information from the rest of its objects. Two sites could not be studied because we encountered

complex recursive patterns in their URLs (probably due to symbolic links in their filesystems). Finally, one website was completely empty. Consequently, we have results for 36 websites (this includes the one website which used many CGI scripts but has other files as well).

Statistics

The following statistics have been established on March 19th, 2002.

Number of objects

We found the following distribution of sites compared to the number of objects:

Number of objects (x)	Number of sites	Average trace time
$x \leq 100$	11	1mn20s
$100 < x \leq 500$	12	5mn20s
$500 < x \leq 1000$	4	18mn49s
$1000 < x \leq 5000$	6	51mn3s
$5000 < x \leq 10000$	1	2h7mn26s
$10000 < x < 50000$	2	3h54mn24s

The two largest websites hold 37528 and 40456 objects. However, we observe that most sites use a small number of webpages. Reasons for these results may be that this domain has very few users or that users' websites are not referenced from the main website. This shows that a garbage collection of such a website would take a small amount of time, making it possible to disable mutation completely (note that the web server never has to stop) during garbage collection. This is useful to allow the use of simple, uniprocessor collectors. We also observe that this is not practical for large websites (an average of almost four hours to examine a website). Please note that these statistics were gathered from a machine of our laboratory and HTTP requests (supported by Java's `URL` class) were made remotely to the servers for each link found. A local garbage collection phase typically spends less time tracing objects by accessing objects locally (we

obtained a timing of about 1 hour versus 3 hours for the site containing 37528 objects). This is interesting because it shows that online link checkers are not sufficient to handle complete websites. Local treatment is required and this thesis shows what additional benefits can be gained (in particular, discover unreferenced objects and observe the Web as a possible computational platform).

Number of links

We found the following distribution of sites compared to the number of links:

Number of links (x)	Number of sites	Average trace time
$x \leq 500$	9	1mn31s
$500 < x \leq 1000$	5	46s
$1000 < x \leq 5000$	10	6mn57s
$5000 < x \leq 10000$	3	22mn19s
$10000 < x \leq 50000$	6	1h08mn
$50000 < x < 300000$	2	1h52mn
1115775	1	4h25mn

We note that one site holds a very large number of links (more than one million); this is the same site that holds the maximum of 40456 objects. In such a context, local collection is necessary, because more than one million HTTP connections proves very costly. Even in an intranet environment and with a responsive site, such a study is bound to require a lot of time. This illustrates the need for a DGC mechanism which has the advantage to localize activities, but still obtains complete and guaranteed results via a limited number of network messages. We also note that a close observation of this site shows that many objects have been visited twice due to the use of symbolic links. This confirms that local manipulation of objects is likely to lead to more precise results.

We also observe that the general tendency is that tracing time is actually dependent on the number of links. However, when we look more closely we remark that it took less time – on average – to trace sites holding 500 to 1000

links than those holding less than 500 links.

Number of external links (x)	Number of sites	Average trace time
$x \leq 100$	16	1mn19s
$100 < x \leq 500$	9	7mn33s
$500 < x \leq 1000$	2	10mn23s
$1000 < x \leq 5000$	6	1h32mn6s
$5000 < x < 20000$	3	2h11mn18s

In the results shown above, we found a correlation between the number of external dangling pointers (i.e. not including links to sites in the group of websites we examine) and the average trace time. Indeed, certain external sites take a long time to answer HTTP requests and that may explain the unexpected result we just observed. If the GC also has to verify the integrity of external links, only a large upper bound can be hypothesized. This bound is created by a limit we place on link integrity checking (in our tests 30 seconds per link), and can be customized. Obviously, in an intranet setting, non-responsive sites are less likely to happen, leading to much better timings.

Dangling links

Number of dangling links (x)	Number of sites
$x \leq 10$	11
$10 < x \leq 20$	6
$20 < x \leq 50$	5
$50 < x \leq 100$	6
$100 < x \leq 500$	6
$500 < x \leq 1000$	0
$1000 < x < 2000$	2

A large number of websites have a limited number of dangling links (11 out of 36 sites have less than 10 dangling links). This is an interesting and somewhat

unexpected result. We also found that these dangling links always represent less that 10% of the total number of links with a vast majority (with exactly half of the sites featuring less than 1% of dangling links). However, this does not change the fact that automatic management of links and objects is required (the maximum number of dangling links we recorded was 1722, which is quite large).

The percentages of internal dangling links with respect to the number of dangling links reveal a full range of possibilities. Five sites have no dangling links and do not participate in the results described in the rest of this section. The 31 remaining sites display internal dangling links from 0% to 100% of the total number of dangling links. We remark however that only 8 sites out of 31 have more than 50% of internal dangling links. We believe this is explained by the fact that internal links are usually easier to control and check than external links. A possible psychological explanation is that authors may not consider that link integrity to remote sites is really their responsibility. If the site has a dangling remote link, it is the fault of the remote site. These results show however that an automatic mechanism is required in most cases (if there are internal dangling links, they represent at least 10% of the total number of dangling links).

Percentage of internal dangling links	Number of sites
0%	7
0% to 9.99%	0
10% to 19.99%	5
20% to 49.99%	11
50% to 99.99%	7
100%	1

Group links

We define group links as external links towards a site in the list of sites we examine (i.e. on campus). This is a useful indication of the need for simple garbage collectors versus a distributed GC. We found 35 websites with less than

6% of group links (with respect to the total number of links). Five sites have no group links at all. One site has around 30%, but absolute numbers for this website are 8 group links for 27 links, which is quite limited. The highest number of group links we recorded in one site was 1316. Here is the distribution:

Number of group links (x)	Number of sites
$x \leq 50$	24
$50 < x \leq 100$	2
$100 < x \leq 200$	7
$200 < x \leq 500$	1
$500 < x \leq 1000$	1
$1000 < x$	1

We also report the percentage of group links that have been found dangling. We note that the websites we examined were always available. This means that a dangling link is actually to an object which no longer exists. As can be observed, we only report information on 31 sites, because five sites have no group link.

Percentage dangling group links (x)	Number of sites
0%	10
less than 5%	1
less than 10%	12
less than 20%	6
less than 30%	2
50%	1

We have a majority of dangling group links between 5% and 10%. This is sufficient to confirm that a DGC is needed. Furthermore, we see that 9 websites out of 36 have more than 10% of dangling group links. This is unfortunately a large number, which may drive visitors to other groups of websites. If this occurs in a company or consortium, many visitors might lose interest, which is obviously undesirable.

Cycles

In the context of the 36 websites we study, we found internal and distributed cycles.

Number of internal cycles (x)	Number of sites
$x = 0$	2
$0 < x \leq 10$	2
$10 < x \leq 100$	8
$100 < x \leq 1000$	15
$1000 < x \leq 5000$	6
$5000 < x \leq 10000$	1
$x > 10000$	2

It is useful to know the average number of objects involved in a cycle in order to evaluate the need for a cyclic garbage collector. The following table summarizes our findings:

Average number of objects per internal cycle (x)	Number of sites
$x = 0$	2
$0 < x \leq 5$	17
$5 < x \leq 10$	9
$10 < x \leq 20$	7
$x = 125$	1

We also observe, from the data we gathered, that the maximum number of objects in a cycle is 2,764, which is quite large. If such a cycle were to become garbage, it is vital to detect it. Note that 33 out of 36 sites feature cycles with a maximum of 100 objects.

In terms of **distributed live cycles**, we found 6501 such cycles over the 36 websites we studied. The minimum number of objects involved in a cycle is 2

(typically two sites referencing each others' root files), whereas the maximum reaches 508 objects in one single elementary cycle. On average, we find the significant number of 320 objects. We can easily conclude that manual management will never allow web authors to find out those cycles once they become garbage. A cyclic DGC is clearly necessary.

We also recorded the span distribution over network nodes of these distributed cycles:

Number of nodes (x)	Number of distributed cycles
$x = 2$	973
$2 < x \leq 5$	91
$5 < x \leq 10$	5415
$10 < x$	22

The maximum number of websites involved in a single cycle was 14. We remark that an overwhelming majority of cycles involve between 5 and 10 websites. We find this to be an interesting result because this can not be expected from simply looking at the websites and visiting them. What we can conclude here is that a DGC working on a large number of sites can work with groups of about 10 websites in order to have a chance to reclaim many garbage cycles. Of course, we base this number on the assumption that the number of websites involved in distributed garbage cycles is similar to that of live cycles. This might not be the case.

6.6.2 Simple user's website

We tested our stand-alone Mark-and-Sweep implementation on several user websites in the domain `http://www.scl.csd.uwo.ca` on April, 29th 2002. The collector was configured to only report garbage objects and not delete or move them. It also checked the links (both internal and external). Again, this is only one way to use a garbage collector for a website: running it regularly to discover garbage

pages and dangling links, and report them to the owner of the site. In primary memory, garbage collectors are used in a more automatic manner because programmers accept certain rules. For example, they do not manually "place" new objects in memory at an address chosen by them: allocation is also controlled by the underlying system. Here, this is not the case, new web pages can be created and added manually by an author. In the context of web authoring, a garbage collector should be integrated to a software which would control the structure of a website, allowing a GC to avoid dangling links rather than just to detect them.

Our test examined 13 user websites. We remark that the number of users in the system is 121. In this case, we see that only 10% of users actually have a website. This experiment starts from either `index.html` or `index.htm` and traces all objects linked from this "root". We recognize the following extensions: .html, .htm, .cfm, .jpg, .gif, .mpg, .pdf, .ps, .gz, .tgz, .zip, .Z, .png, .tex. Only the first three extensions correspond to HTML code and such objects will be parsed for references. The following tags are recognized: `HREF` (in any tag such as `A`), `META URL`), `IMG SRC`, and `FRAME SRC`. We also checked for external dangling links using Java's facility to access URLs. Certain URLs are difficult to check because certain addresses do not answer before a very long time. We set a timeout at 30 seconds in order to obtain results in a reasonable timeframe. Finally, we note that objects that are not readable are not considered garbage because they are intentionally private. However, links to these objects are considered dangling.

Objects

Number of objects (x)	Number of sites
$x \leq 3$	4
$3 < x \leq 100$	1
$100 < x \leq 1000$	5
$1000 < x$	2

This result shows that a majority of users' websites typically have less than

a thousand files. We also note that 4 websites (about one third of the sites we considered) have very few objects (maximum three). In fact, these are mainly used to publish a document or two, and no other information. We note, however, that 5 sites have between 100 and 1000 files, which shows a typical size for a user's website. We will see that these sites contain a significant amount of garbage/unreferenced objects, which is also typical of most websites:

Number of traced objects (x)	Number of sites
$x = 1$	7
$1 < x \leq 10$	3
$10 < x \leq 200$	3

Percentage of garbage objects	Number of sites
0%	3
less than 30%	0
less than 50%	1
less than 80%	3
more than 80%	6

The number of traced objects is far smaller than the total number of objects. The consequence is that the amount of garbage is quite large. The three sites with 0% of garbage are special in that they hold only two or three objects. We also found 9 sites which have more than 50% of garbage objects with a maximum of 99% (and a large number above 95%). This state of affairs might be explained with three reasons:

1. This is the first time a garbage collector is run on those sites. Once a user decides to rely on a GC, subsequent runs will probably find less garbage (depending on the frequencies of site modification and call to the GC).

2. Safety was not guaranteed by our implementation of a garbage collector in this environment. Objects might have been deemed garbage whereas they are reachable through Javascript code for example.

3. Some of these files might be "secretly published" as described in Section 6.4.7.

GC Time

We recorded the time necessary to identify garbage objects and dangling links in each site. The longest time required is 3mn40s with 9 sites requiring less than 1 minute to complete a collection. This means that, in such configurations, it is acceptable to lock users' web directory for a garbage collection in order to identify potential problems. Typically, an author would set a "cron job" or equivalent at a specific time to run the GC. This time should be chosen with respect to the author's usual schedule to reduce possible conflicts. For example, most cron jobs are set to be run at 4am, which is usually a time where users are not working.

Links

The number of links in live objects is generally fairly low, but can be quite high – with one instance observed with 3575 links. We count an average of up to 60 links per traced object (i.e live HTML files) and a general average of 14. Note that we did not record internal links inside traceable garbage objects.

Number of links on one site (x)	Number of sites
$x \leq 3$	5
$3 < x \leq 100$	4
$100 < x \leq 1000$	3
$x = 3575$	1

We also recorded the percentage of dangling links (both internal and external) with respect to traceable objects:

Percentage of dangling links (x)	Number of sites
$x = 0\%$	5
$0\% < x \leq 20\%$	1
$20\% < x \leq 50\%$	2
$50\% < x < 100\%$	1
$100\% \leq x$	4

We remark that 4 sites have 100% or more dangling links w.r.t. number of objects. These sites hold only one traceable object each, which explains this result. However, we remark that it is not impossible for a site to have a number of dangling links larger than its number of objects. In the current test, this does not appear to be the usual case, however. We note that 4 sites have dangling links, which is, of course, undesirable. As soon as only one link is likely to refer to a non-existing page, a checking mechanism should be used.

Cycles

We gathered data about cycles and garbage cycles on each site. We report the following information: number of cycles (live and garbage), average number of objects per cycle, number of cycles per link. We note that these are *internal* cycles (i.e. on one site only); statistical experiments about distributed cycles can be found in Section 6.6.4.

Number of live cycles (x)	Number of sites
$x = 0$	8
$0 < x \leq 20$	3
$x = 68$	1
$x = 233$	1

Number of garbage cycles (x)	Number of sites
$x = 0$	7
$0 < x \leq 20$	3
$100 < x \leq 200$	2
$x = 458$	1

We observe that the number of garbage cycles can be very significant: 3 out of 13 sites have more than 100 garbage cycles. However, we also note that a large majority of sites have no garbage cycle at all. Among these 7 sites, only one site had 68 live cycles and no garbage cycles. Other sites without garbage cycles had originally no cycle and featured a small number of objects.

We also recorded the average number of objects per cycle among the 5 sites featuring live cycles and the 6 sites with garbage cycles:

Average number of objects per live cycle	Maximum number of objects per live cycle
4	22
2	3
2	4
2	5
1	1
Average number of objects per garbage cycle	**Maximum number of objects per garbage cycle**
2	15
1	2
1	2
2	2
2	9
10	29

As we can see, certain cycles can be quite large (up to 29 objects), but in general, cycles are quite small. In comparison with official websites, cycles in user sites have a limited size because the total number of objects is generally

different by one or more orders of magnitude. Obviously, the more objects, the more likely it is to observe large cycles.

We note that among live cycles, very few cycles actually contain the main root file (typically `index.html`). This emphasizes the need for a garbage collector because there exists many cycles that could potentially become garbage.

Conclusion

Although the low number of sites studied here does not allow us to make strong claims (see the DGC experiment in Section 6.6.4), it appears that reference counting mechanisms are not sufficient to handle most single user websites. Indeed, we can observe from our results that many of these websites contain cycles, which are typically not detected by RC algorithms. We also found a large number – with respect to the total number of sites we examined – of very small websites. Clearly, when a site has only one or two files, it might not be useful to use a garbage collector. However, we can argue that, if the files contain links to outside resources, a checking mechanism would help ensure the correctness of the site. Also, in a distributed context, a local collector for such small sites would help strengthen the interconnection of documents through the use of a DGC. Finally, a simple argument is that a Mark-and-Sweep garbage collection for a small site (with one to five files) typically takes one or two seconds, which is negligible. If good habits are taken early, the site can grow and still stay well-connected.

6.6.3 Java documentation

We used our garbage collector program to gather data about the Java API documentation. This shows that we can use such a piece of software to manage any set of hyperlinked documents. In the Java API documentation for the JDK1.4.0, we found 6,915 files and directories, including 6572 HTML files. We counted 547,618 links, including 540,506 internal links and 7112 external links. This amounts to 83 links per traceable file.

Time

It took about 31mn to complete the collection of the directory. This can be explained by the large number of external links to check. Without checking external links, the GC takes 24mn. We also tested the same collection process without logging events. This seems strange, but the log file amounts to about 90MB on the local disk. Updating this file is thus costly and removing this operation results in a collection time of 13mn instead of 24mn. This shows that a better log mechanism is required. However, in a normal setting only the list of garbage should be reported, and this activity is not time-consuming.

Dangling links

We found 1 internal dangling link. It appears to be a simple mistake (the name of a sub-directory is missing in the path to a webpage), but it exists, showing that a tool such as W3GC is needed. We also found 13 dangling external links (leading to an error 404) to three different websites. Those websites exist, but objects were moved or removed. Also, 1 link was declared dangling after a timeout period. Finally, 1 link was wrongfully identified as dangling due to a syntax problem. After fixing the problem, the link was correctly identified as valid.

Garbage objects

Apart from reporting dangling links, our garbage collector also reports unreferenced objects (!). Quite interestingly, we found five GIF files that were not linked. This shows that garbage objects exist even in environments created for official distribution.

Cycles

We recorded 86359 cycles. It should be obvious that there is no garbage cycle (total number of garbage objects is 5 GIF files). The number of cycles include 6420 back-pointers to the root of the site, which is close to the maximum of 6572 (see Section 6.5.3 for further details).

Conclusion

Although the results show a very small number of garbage objects and dangling pointers, we can still conclude that, even in commercial products, such a tool as a GC for the Web is necessary. Dangling links to remote sites are hard to control, but expiry dates (as described in Section 6.2 and Section 6.4) would certainly help ensure the safety of the links. An evolution of HTML reference mechanism towards more flexible and powerful technologies (Xlinks [25] for example) would allow a fine-grained management of object references.

6.6.4 DGC for user websites

We consider the site `http://www.csd.uwo.ca`, where we found 75 user websites (faculty, staff, graduate students, alumni). Even though, these are located on the same filesystem, we can easily organize them into a "logical" network of websites. We use a local collector (Mark-and-Sweep) for each node and a distributed collector (GCW) to study the structure of the websites and possibly find distributed garbage pages. Apart from testing our DGC implementation, this scenario can be used as a real configuration for web management, because it has the interesting advantage to allow asynchronous collections of logical websites. This results in a responsive and non-intrusive solution.

A *first interesting result* is that we had to "prepare" a distributed structure of the sites to comply with the structure required by our DGC. Normally, the Web environment does not use opaque addressing. Consequently, we made a small program which scanned all accessible files in each directory of users' websites and created entry and exit items (note that we also scanned garbage files). In a practical exploitation of the tool, items should only be created once and the mutator (ideally a compliant web authoring tool) would correctly create and maintain entry and exit items.

An application of such a garbage collection mechanism could be several users working together and cooperating via their websites. To make sure each site is coherent, a local collector is used and a DGC can be set up to ensure global consistency. Each local collection can be run once or twice a week for example.

In the following, we present the results of our experiments and corresponding conclusions. We used the GCW collector (see Section 2.5.4 for a presentation and Section 7.4 for details about its design and implementation in the Web environment) and a local Mark-and-Sweep collector (see Section 7.2).

Number of objects

The total number of files and directories we attempted to examine using our distributed garbage collector is 24792. Note that we did not experiment with all the sites available in the www.csd.uwo.ca domain, only users' websites. We also exclude objects that we could not handle (i. e. not all types were recognized). Although this gives us an average of 330 files per site, we note that the population of web documents is very diverse and goes from almost no object (1 or 2) to a very unique maximum of 13,159. Here is the distribution:

Number of objects (x)	Number of sites
$x \leq 10$	14
$10 < x \leq 100$	34
$100 < x \leq 500$	21
$500 < x \leq 1000$	4
$x = 1839$	1
$x = 13159$	1

We can observe that most of the sites (34) have between 10 and 100 files. In the study of the sites at SCL, the majority was between 100 and 1000. This shows that different environments lead to different distributions, although we can conclude that most users maintain websites with less than 1000 files. This is important

when a collector is used to check local integrity, because the time required to do so will be very low and it would be acceptable to lock the directory during this time effectively removing the need for complicated synchronization mechanisms. Note that live pages would still be available for reading and webpage servicing is never impaired.

Our experiments showed a large number of live objects (17,612). However, we remark that among them, we found one site with 13131 live objects. If we exclude this "special" site, we obtain 4481 live files and directories for 11627 objects in total. We note that we found a total of 745 live directories (including 6 directories for the largest site).

Number of live objects (x)	Number of sites
$x \leq 10$	38
$10 < x \leq 100$	26
$100 < x \leq 500$	9
$500 < x \leq 1000$	0
$1000 < x$	2

Compared to our results from the experiments with the SCL users' websites (see Section 6.6.2), we observe a shift from a majority of sites having between 10 and 500 objects to a majority having less than 100 live objects. This is quite interesting, because it illustrates the evolving aspect of websites: many web documents are created and possibly linked at some point, but they are later discarded and forgotten. This seems to be a rather frequent event.

We also recorded 12008 traceable live objects (a traceable object is an object we can parse for URLs like HTML files and unlike JPEG files) including 10088 such objects for the large site.

Garbage objects are also important to study; we recorded 221 garbage directories and found 6959 garbage files with the following distribution:

Number of garbage files (x)	Number of sites
$x \leq 10$	22
$10 < x \leq 100$	35
$100 < x \leq 500$	15
$500 < x \leq 1000$	3

The number of objects remotely referenced by another site of the group is quite small. We found 43 public objects (live or garbage) on 20 different sites with a maximum of 10 public objects for one single site. This result can be explained by the usual organization of web sites in research environments. Few users provide references to other users' pages because a single website, typically associated to a laboratory, is the central point to gather information. Most users will reference the laboratory's website rather than members' websites.

Number of links

The total number of links we found is 129,055 links in live objects including a maximum of 68,118 links for a single site (the one featuring 13,159 objects). This gives an average of 7 links per live object if we include this site and an average of 13 links if we exclude it.

An interesting result is the number of links per traceable object. We found an average of 10 such links when the largest site is taken into account and **31** otherwise. This shows that websites have strong interconnections.

In terms of destination of the links, we found a very large majority of internal links (125,044), which is similar to primary memory environment: internal pointers are usually a lot more numerous than external pointers. We also recorded 3972 links to remote websites and only 97 links between websites of the group we studied.

Finally, we found 518 internal dangling links and 2 dangling links to a remote site within the group. No data was collected about dangling links to external remote sites, but interesting information on this topic can be found in the previous experiments of this chapter.

Cycles and Distributed garbage

We present here our results concerning cycles: internal live and garbage cycles as well as distributed live and garbage cycles. We also discuss the distributed garbage we found on these sites. The results we exposed above were taken from the logs of the first local GC on each site. This implies that most distributed garbage had not been discovered yet because of the need to decrement entry items.

A large majority of sites has a no cycles at all; this can be explained either by the use of frames (no need to refer to parent files) and by the fact that a majority of sites have a small number of objects. Yet, over 75 sites, there is an average of 18 cycles per site with the following distribution:

Number of live cycles (x)	Number of sites
$x = 0$	46
$0 < x \leq 5$	14
$5 < x \leq 10$	5
$10 < x \leq 100$	9
$x = 1080$	1

We note that the site with the largest number of live cycles (1080) is not the site with the largest number of objects. We also recorded that, on average, live cycles have less than 5 objects involved in any given cycle. Only one site has an average of 12 objects per live cycle. The maximum number of objects for a live cycle is 51, which is quite high for a user website.

As for our experiment studying stand-alone websites, garbage cycles are numerous than live cycles. The average is 24 such cycles per site, a maximum of 647, and the following distribution:

Number of live cycles (x)	Number of sites
$x = 0$	55
$0 < x \leq 5$	4
$5 < x \leq 10$	2
$10 < x \leq 100$	10
$100 < x$	4

We also recorded that, on average, 16 sites have less than 5 objects involved in any given garbage cycle and 4 sites have between 5 and 10. The maximum number of objects for a garbage cycle is 43.

Distributed garbage

In the initial configuration, we found 55 exit items which gives an average of less than 2 links per exit item. This means that few objects are referenced more than once.

During the execution of the DGC, 31 exit items on 10 websites were removed (leaving 24 live exit items on 13 websites) because they were not reachable or rather that local objects referencing certain remote objects were actually garbage.

Finally, we found 8 distributed garbage objects (i.e. objects referenced from garbage objects on other sites and not referenced locally). 1 of these objects was referenced twice by garbage objects on two different sites.

Distributed cycles

We found only 20 distributed live cycles and no distributed garbage cycle. This small number can be explained by the small number of public objects. As observed in our experiments on the websites of our university, the number of distributed cycles can be quite high. A conclusion we can draw from this result is that, in our department, work cooperation is not organized via user websites.

Manual examination of laboratories' websites shows that work is organized at this level and users usually refer to their laboratory's website (most of the time, only the main URL).

We note that the cycles we found hold between 4 and 19 objects for an average of 9 objects per cycle. It is also important to note that all cycles hold at least one root file, which means that these cycles will remain alive until they are broken. In this environment, a cyclic distributed collector does not have a lot of work to do. In our experiment, the process stabilized and completed without finding any distributed garbage cycle. This result (no garbage cycle) is explained by the small number of cycles (only 20). We note, however, that it is possible that few distributed garbage cycles exist in certain environments. Indeed, in our environment, most pages contain the names of laboratory members and participants to a given project or joint work. Usually, names are associated with the URL of homepages, which makes the cycle live.

Timings

We recorded the time spent doing a GC for each site and results are quite different from what we showed in our experiments with single user websites. The reason is that, in the present test, we did not activate the routine normally used to check remote link integrity. This allowed faster tests, because HTTP requests to certain remote websites take a long time, even though we had set a timeout at 30 seconds in the previous experiment. We found an average of 6 seconds to perform a GC on a user website. This is clearly very acceptable and this average goes down to 4 seconds if we exclude the site holding the largest number of files. The maximum time taken by one single local GC is 9mn30s (for the largest site). We also note the interesting result that the only first local GC handled about 13000 files on this site, but most of the files were only referenced by a garbage object on another site. This resulted in subsequent GCs of this site to take into account very few

files and timings were around 1 or 2 seconds.

Furthermore, each site used four local collections and we note that the last GC was actually used to detect termination because all the garbage had been detected before. Finally, the entire DGC process took 31mn to complete, which appears quite reasonable to handle 75 sites with almost 25000 files heavily connected to each other (we have seen that there is an average of 10 links *per* object).

Conclusion

This experiment was interesting on several accounts. First, it reports on an actual environment with a non-trivial number of websites (75) and it allowed us to test our implementation of a distributed collector. We observed interesting results. We had to create opaque addressing items to support the DGC, effectively formatting websites into appropriate distributed application nodes.

Most user websites have a low number of objects (less than 1000 with less than 100 live ones) and garbage collecting these sites takes very little time. We also found a high pointer density in objects of these sites. Finally, an important result is that the number of distributed cycles is very different from the number found by the study of UWO websites: almost no cycle exists and all of them go through at least a root file.

Finally, we note that the DGC required at most three rounds of local GCs before global stability. The first round propagates all marks and reclaim garbage, the second round stabilizes the system, and the third round forwards stable information to all nodes, which then breaks distributed garbage cycle.

6.7 Conclusion

In this chapter, we have presented a new way to use distributed garbage collection algorithms: garbage collecting the Web. We have formulated web maintenance as a memory management problem by establishing a precise semantic correspon-

dence between concepts in primary memory and in the Web.

Our experiments were led in four different environments: all sites of the university, users' websites in a single lab, Java documentation, and networked users' websites in the department. This allowed us to gather data on website structures (number of objects, links, dangling links, ...) and find unreferenced objects even in the Java documentation. From these results, we can claim that garbage collection is needed in the world of Web management.

This project also shows that it is now possible to think of the Web as a legitimate computing platform, which manipulates documents instead of binary data. Many components are already available (objects, pointers, write barriers in the form of web authoring tools, ...), we have now added the possibility to use a garbage collector.

Chapter 7 details our design and implementation of the collectors we used in our experiments. In Chapter 8, we propose to reuse this design for the creation of a software development kit to easily implement local and distributed collectors. This is the foundation of a Web-based experimentation platform for garbage collection research. We will also propose a list of tools which would give this platform complete support for studying and experimenting with stand-alone and distributed garbage collectors.

Finally, we believe this work shows that DGC algorithms can not only be used to collect garbage memory and detect leaks, but can also be used in many areas where specific algorithms are needed to discover the relationships between different entities. As well as managing documents on the Web, this work can be used to manage documents in many types of information systems, based on XML for example.

Chapter 7

Designing and Implementing W3GC

This chapter details the design and implementation of one possible architecture of a garbage collector for the Web. This is an instance of W3GC that we describe in Chapter 6.

In order to implement W3GC, we rely on the design method based on a semantic separation between stand-alone GC and local collectors for DGCs (this method is described in Chapter 5). At the same time, this application is meant to be a direct test of the design method. This chapter describes the design and implementation of a M&S garbage collector, a Distributed Reference Counting scheme, and a distributed collector called "Garbage Collecting the World" (see [51] for details).

Section 7.1 presents a general view of the design. Section 7.2, Section 7.3, and Section 7.4 show the actual design of a garbage collector for a single web site as well as a distributed garbage collector for a set of sites. Section 7.5 presents several practical issues we encountered while implementing the tool.

7.1 Designing garbage collectors for the Web

The design of a garbage collector for the Web differs from that of a garbage collector for primary memory. Properties such as timing and lifetime are usually separated by several orders of magnitude. Whereas objects in memory are mutated several times per second, objects in a Web environment can be mutated several times per week or at best days. In order to design a garbage collector for the Web, we rely on the study of semantic correspondence between WWW and memory presented in Section 6.3. Models and methods described in Chapter 5 also support this activity.

On more practical matters, we use object-oriented programming (Java) to implement this garbage collection tool. Object-oriented design allows us to reproduce in actual design the algorithmic models we presented in Chapter 5. Inheritance is of particular interest as it emphasizes the special nature of certain collectors such as hybrid DGCs which rely on distributed reference counting or listing collectors. A practical design should rely on the class for such a collector to inherit from the class describing a DRC algorithm. Composition can also be used to model such collectors as generational, CMM, and so on. This observation proves useful in practice because it directs the attention of a designer to the essence of an algorithm, and could lead to a design with a better structure.

In this section, we describe important challenges we faced while designing and implementing a garbage collector and a distributed garbage collector to manage web sites. We chose a Mark-and-Sweep algorithm and the DGC known as "Garbage Collecting the Web" by Lang, Piquer and Queinnec [51] (although we do not handle groups: there is just one group). These choices were made based on two criteria: dead cycle reclamation and simplicity. We do not claim that these choices were the only ones we could possibly make, we simply had to choose collectors in order to illustrate this work.

As detailed in Chapter 2, the GCW collector is a cyclic distributed GC which uses a distributed mark-and-sweep mechanism on top of a distributed reference counting algorithm. It relies on mark propagation, both locally and between nodes, to discover distributed garbage cycles. HARD marks are propagated from local roots, while SOFT marks indicate either a garbage cycle or an object that has not been marked HARD yet. A phase of detection is over when all marks have been propagated. A DTD algorithm is used to assess the end of the phase.

7.1.1 Stand-alone GC or DGC?

The choice between a stand-alone GC and a DGC depends on the application. Personal websites can be managed by a stand-alone GC. Several websites located at different places in the world obviously require a distributed collector. One might observe that webcrawlers (see [46] for a presentation of webcrawlers and web robots in general), which recursively visit web documents to gather data, are not really distributed but transport themselves from website to website. We could use the same idea to verify and reclaim dead garbage. However, this triggers the question of security: what is a webcrawler allowed to do? Furthermore, local permissions might become too complicated to handle. A local GC, completely controlled by the local author, is likely to prove much safer.

We now investigate two special situations. In the first one, the website is part of a set of websites but its maintainer chooses not to participate in the DGC. In this case, a stand-alone collector will still function properly. It has to be clear, though, that any garbage cycle containing at least one web object located on this site will never be reclaimed.

The second situation is more subtle. We consider different web servers in the same corporate intranet environment, or on the same computer using a multi-hosting feature of the web server. At a low level, the same file system can be used, by means of NFS for example. In this case, choosing a stand-alone algorithm

with a correspondence table between the base URIs and the actual directories might prove more efficient because it avoids unnecessary network messages. As mentioned in Section 6.4.4, an environment with users might still require a DGC to allow users to manage their own sites.

7.1.2 DGC server

A DGC "server" is needed to handle GC messages sent by remote nodes. While local GCs are not running continuously, a process still has to be resident and constantly available to handle DGC messages. Indeed, local GCs run asynchronously and can send requests to remote nodes (to decrement a counter for example). Another solution would be to write requests to a remote website known by all local GCs. Requests would be then be read by these collectors when they start up. This solution has the disadvantage of relying on a central element (the "repository of requests") and is thus not a good solution for reliability and scalability.

We can implement this resident process with an independent daemon or with a module to the web server. While the former solution allows for simplicity and flexibility, the latter brings performance by making it natural to piggy-back garbage collection messages onto HTTP messages. However, this latter case brings up issues such as permission problems. Indeed, the HTTP server accesses websites only for reading. A DGC module requires write access to certain directories. It would be natural to think that local GCs information should be stored within each website's directory. However, it might make more sense to maintain this information in a separate directory writable by the HTTP server.

7.1.3 Choosing a collection technique

Garbage collection algorithms have peculiarities that can be mapped to notions in the WWW. It is important to understand those differences in order to choose an appropriate technique. In this context, we believe that a Mark-and-Sweep

GC is the best choice as a stand-alone collector. Indeed, it is the least intrusive of collectors and is quite simple to implement. Furthermore, performance or real-time considerations are not an issue here. This means that sophisticated solutions such as generational algorithms or even copying collectors would require a complicated implementation strategy for very little benefits.

Choosing a DGC proves more complicated and depends heavily on the specifics of the system. Furthermore, it is not clear yet whether one distributed collector is better suited than others for a specific situation. We can easily rule out non-cyclic collectors because the Web environment does have many cycles (as explained in [64]). Hybrid collectors are, as usual, the safest choice. Experiments would be needed to decide what hybrid algorithm is best suited.

7.1.4 Our implementation

We present the choices we made for two implementations:

- stand-alone garbage collector for personal websites.
- DGC for a set of websites.

First, we decided to use Java as a language and Java RMI for network communications. Java offers a rich library of classes to assist us with many of the tasks we have to perform. Of course, any language could have been chosen.

We focus on a Web DGC for an intranet environment. For example, we used our code to obtain statistics on a set of websites managed at our university. We also imagine that our DGC can be used between several users. Without any support from the system administrators, they could form a group of websites to be managed by the collector. This can be done by regularly invoking (via a `cron` job for example) a local garbage collector, which also participates in the DGC. Even if the files are part of the same filesystem, it would make sense to use a DGC rather than a GC to be able to cope with permission problems. Local GCs are

run with specific user's permissions and are thus allowed to potentially modify the site (depending on the collection algorithm). A non-intranet environment might require a different approach because of its need for scalable and dynamic features. However, this would not change the main ideas exposed in this work.

Along the lines of the design choices described above, W3GC can currently only find references in HTML files, not in Javascript code for example. However, our approach is extensible, because we used an object-oriented style of design, and we will see in the next sections that adding more types is not a problem. When we detect garbage, we decide to list them (in a file or in an email). This is because we want to see if certain web pages should be re-attached or not. As mentioned before, we propose two implementations: a stand-alone GC and a DGC, so the choice at this level is a matter of what implementation to apply to a particular case. Of course, we offer both possibilities to illustrate our work. Our DGC server will be implemented as an independent daemon for simplicity reasons.

We now describe our choice of garbage collection techniques. For the stand-alone collector, we choose a *Mark-and-Sweep* algorithm because of its flexibility and transparency. It has the advantage not to move objects, which is a useful feature in a Web context to ensure availability at all time, while still detecting all garbage documents.

To manage several sites, we use a distributed garbage collector. Most DGCs have been designed to function with a local Mark-and-Sweep algorithm. We thus have a large panel of possibilities. Our choice was made according to one criterion: ability to reclaim cycles. Our goals are to illustrate our work on a DGC for the Web as well as provide observations and experience about a non-trivial DGC implementation. In the following, we explain the decision process involved in our choice of a DGC:

- DRC or DRL collectors are simple collectors but they can not reclaim cycles. This is the reason why we could not use those techniques.

- DMOS is an interesting algorithm but quite complex.
- Hybrid algorithms are usually chosen for their simplicity and ability to handle cycles. Among all choices, we preferred "Garbage Collecting the World" [51] because of its flexibility and simple concepts. GCW is based on a DRC algorithm and uses a sort of distributed Mark-and-Sweep algorithm to handle distributed garbage cycles.

In the following sections, we describe the design of each component of our collectors: mark-and-sweep (Section 7.2), DRC (Section 7.3) which is the basis for GCW (Section 7.4).

7.2 Stand-alone collector

Our stand-alone collector is of type Mark-and-Sweep. Its model – using the specifications we explained in Chapter 5 – is detailed in Section B.3. This shows concepts that we need and proposes algorithms to implement. In the Web environment, we have to specify two of those concepts:

- A **Reference** (an `Address` in the SAMS model) is represented by a local URI (without `http` keyword otherwise it is a remote pointer), an `IMG` reference, an `APPLET` reference, and so on.
- An **Object** is an HTML file, an image file, an applet directory and so on. Obviously, the method `listRefs` which finds out what are the references included in an object will be of particular importance because web pages are not regular memory objects.

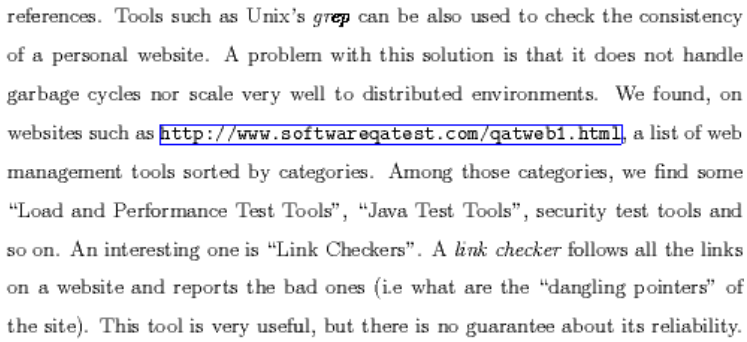
7.2.1 listRefs

Let us study this operation more closely. A first observation is that we have a precise knowledge of the types of values included in an object. Each value is tagged with its type using the HyperText Markup Language [23]. Consequently, we don't need to be conservative as in C or C++. When we encounter a tag such as `<IMG ...>` or `<A HREF=...>`, we know what we are dealing with.

A second observation is the **property of scannability**. This property explains the nature of a particular web object. For example, an HTML file is scannable, i.e we can parse it and find references to other web objects, while a GIF file is usually not scannable. This property is essential for `listRefs`. To find out the type of a web object, we rely on two elements: the tag and the MIME type. Indeed, a tag `IMG` tells us that an image is referenced, but a tag `A HREF` simply gives us a reference to a web object (text, image, html, binary, etc.). Consequently, we need to rely on the extension of the name of the file like any web browser does to find out what type this object really is. Once we know the type, we simply check if it is scannable or not. In the current implementation, we handle html files as well as various non-scannable types such jpg or gif. We remark that pdf documents (see Figure 7.1) may contain references to web pages. In this case, they could be considered scannable if a suitable `listRefs` function can be provided. This is important because it shows the flexibility of our mechanism: a non-scannable object can become scannable *if* appropriate support is provided.

It is also interesting to note that, due to the particular nature of the environment, `listRefs` and the GC are implemented differently than in primary memory with respect to references. In primary memory, an object is traced and as soon as a pointer is found, tracing continues recursively on a new object. Remaining pointers in the object that was being scanned are treated later (when the recursive trace returns). In a Web environment, this can be problematic because it would mean leaving files open. The trace would open the file, find a URL, trace

(a)



references. Tools such as Unix's *grep* can be also used to check the consistency of a personal website. A problem with this solution is that it does not handle garbage cycles nor scale very well to distributed environments. We found, on websites such as http://www.softwareqatest.com/qatweb1.html, a list of web management tools sorted by categories. Among those categories, we find some "Load and Performance Test Tools", "Java Test Tools", security test tools and so on. An interesting one is "Link Checkers". A *link checker* follows all the links on a website and reports the bad ones (i.e what are the "dangling pointers" of the site). This tool is very useful, but there is no guarantee about its reliability.

(b)

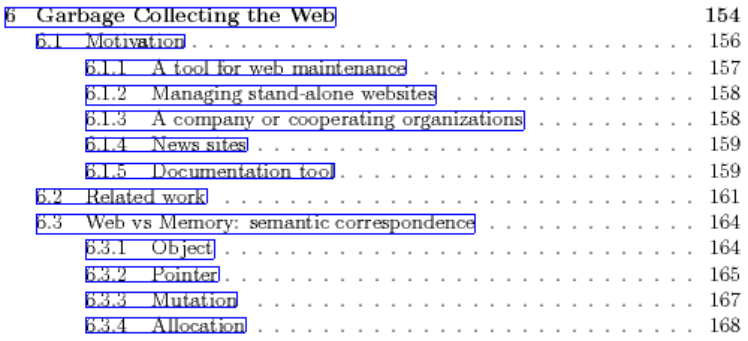


Figure 7.1: *Snapshots of a pdf file. (a) extract of the current chapter, which contains a URL. (b) PDF references inside the same document (this thesis).*

recursively the corresponding web object. This is a potential risk because the allowed number of open files is limited, and there is no guarantee that the depth of recursion will not lead to a situation where the maximum number of open files has been reached. We solve the problem by opening the file, finding all references, closing the file, and, *only then*, tracing recursively.

7.2.2 UML models

We decided to use a class-based object-oriented design and we selected UML to model the base classes we created to simulate a primary memory environment (i.e. references, objects, garbage collector). Of course, such a design is not unique, and we merely describe our view. Also, we would like to point out that the design methodology does not have to be UML, and the design style does not have to be object-oriented. We made this choice because we believe that Java is offering a good library of features to handle the present problem. This naturally led to using objects and classes to create the collector.

Web Objects and HTTPObject

In the Web environment, `Object`s from the stand-alone mark-and-sweep model are actually called `WebObject`s. We have to handle many types of web objects. Some are scannable, some are not. We decided to create an abstract `WebObject` class and to use virtuality of methods so that the GC handles only `WebObject`s. The class `WebObject` contains a table of types and we use the Prototype design pattern (for more information about design patterns, see [32]) to create an object of the appropriate subclass.

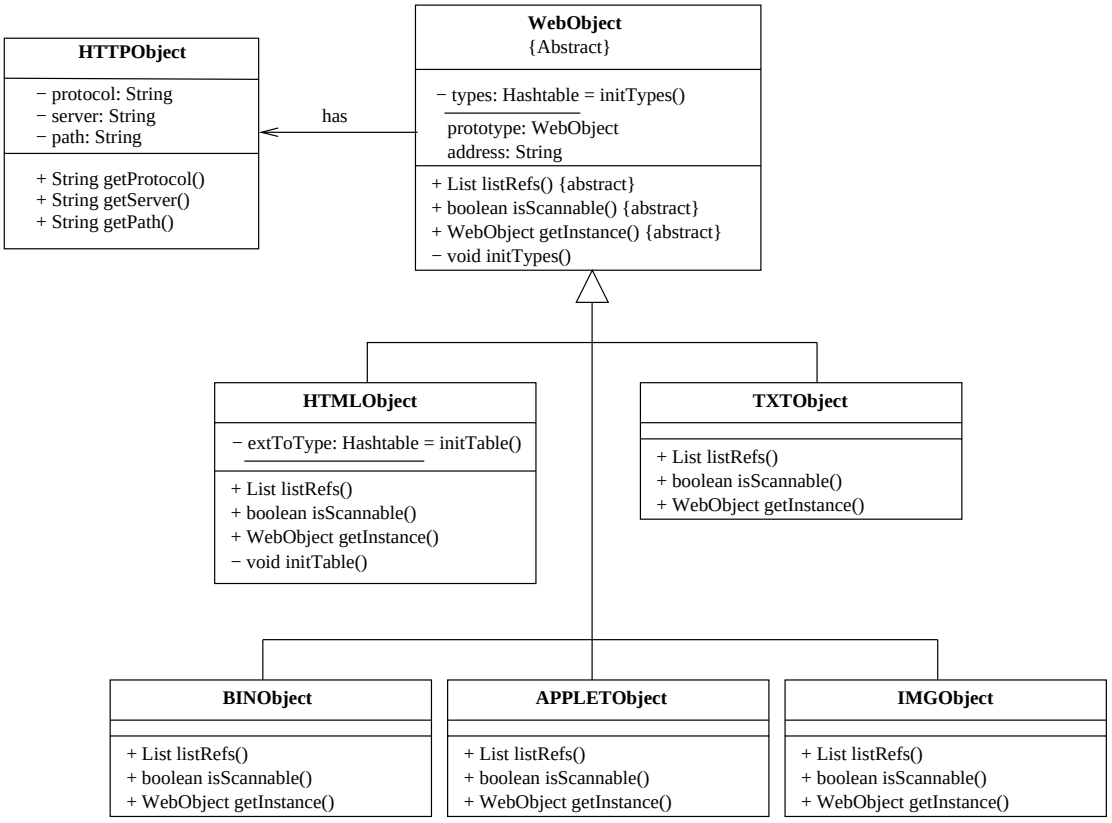


Figure 7.2: *WebObjects and HTTPObject.*

Subclasses each represent a particular type of web object and will have a method `listRefs` to scan themselves for references, which leads to the creation of new web objects. The simple class `HTTPObject` is used by `WebObject`s to store their own addresses (URIs). It represents the `Reference`s of the model. In the Web environment, we distinguish three components: *Protocol* (e.g `http`, `ftp`, `file`, ...), *Server* and *Path*. Figure 7.2 shows both `WebObject` and `HTTPObject`.

Miscellaneous

Apart from the `HTTPObject` and `WebObject` ADTs in the model, we find `MS` (Mark-and-Sweep) and `MarkSheet` (to store marks used by `MS`), described in Figure 7.3 with their relationships to other classes. Note that MS has been renamed

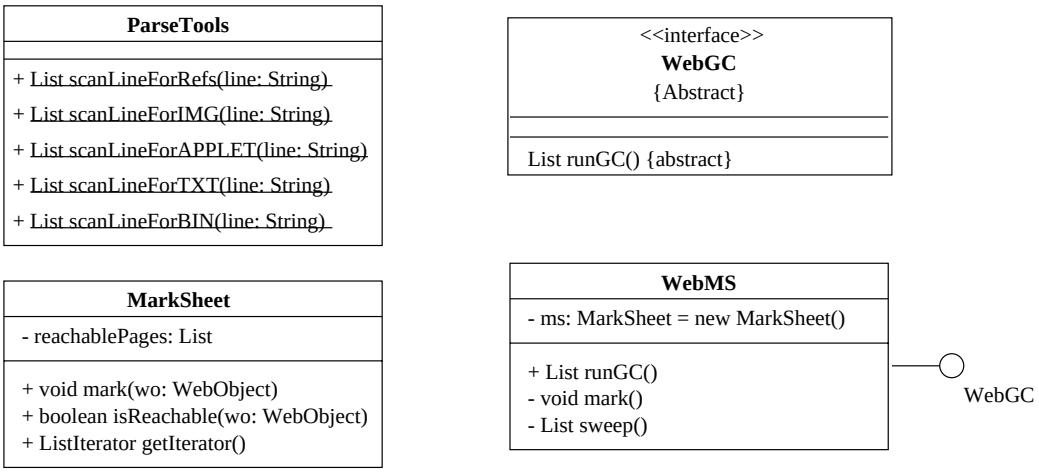


Figure 7.3: *MarkSheet and MS.*

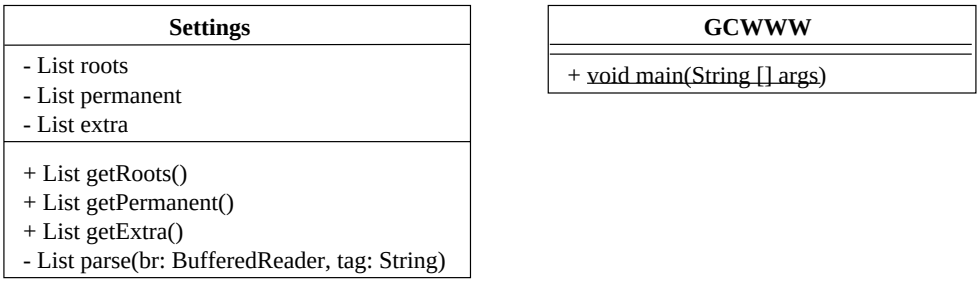


Figure 7.4: *Extra classes.*

into WebMS. Figure 7.4 show classes specific to our application. They are used for tasks such as parsing the configuration file, creating the WebGC object, running the GC, and so on. Finally, Figure 7.5 summarizes the relations between the classes.

7.2.3 Conclusion

The application uses the classes defined above to create a GC for the Web. The "sweep" part of the algorithm has been truncated to simply return a list of "references" of garbage web pages. This application can be used for personal web

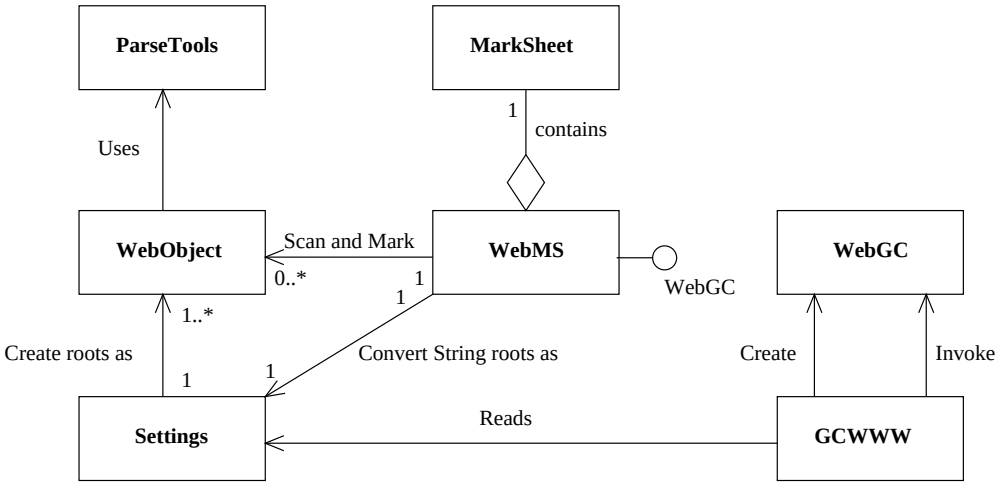


Figure 7.5: *Classes relations.*

sites without any communication with other web sites (although we can report external dangling pointers). This is why it is called a *stand-alone collector*.

In the next section, we design a DRC scheme to serve as a basis for GCW. This intermediate step was convenient to debug our implementation and test our mapping model (see Chapter 5) with DRC and WebMS.

7.3 DRC

The model for this DGC (Section C.1) describes a simple organization. We make several observations:

- The Generic GC shows that **entry items** and **exit items** should be implemented. Entry items are used to hold the counters and exit items make it easier to handle remote pointers.
- Those items are likely to have a very long lifetime (maybe several years) because we are in a Web context.
- Web pages - unlike regular objects in traditional environments such as

CORBA - contain references to remote pages directly. There exists no indirection in the sense of opaque addressing in the web environment.

Potential lifetimes of the items imply two scenarios: either entry and exit items are made persistent (in order to avoid losses if/when host machines crash or get turned off), or there exists an underlying mechanism to save and restore information when crashes occur thus taking care of entry and exit items. Furthermore, when the local GC works with web pages, it has to map addresses to exit items in order to know what entry item is concerned by the mutation. Consequently, items should be scanned at each run of the collector in order to guarantee a correct view of local items.

At a lower level, a resident piece of software needs to be present for each web site to handle protocols required by the DRC. This is the *DGC server* (see Section 7.1.2), which handles decrement requests from remote nodes. Unlike local GCs, the server should always be available. This means that, if an underlying fault-tolerant mechanism exists, entry and exit items can live in memory inside this server. The other solution is to use files to represent these items, which is what we chose in our implementation, because our experimentations did not require such a sophisticated mechanism. However, we propose to integrate, as a future work (see Section 9.2), a distributed garbage collector in a web authoring tool. It is conceivable, at this time, to use such a technique.

We also note that our server is a separate process on the host machine. This has the implication that the HTTP server and the Web DGC server are not the same piece of software, and a mapping between them is required. More precisely, we have to maintain two types of address, an RMI one and an HTTP one. Both are equivalent, but need to be identified depending on the situation. The GC parses web files to find references, which are HTTP-formatted. In order to understand whether they are part of the group of nodes managed by the DGC, a translation is operated using a mapping table. If the reference has a

correspondence to an RMI address, it is used to communicate with the remote DGC server.

We observe that a more transparent solution would be to create "httpg" servers (http servers handling "g"arbage collection), which would integrate a DGC server. However, the architecture we use in our implementation allows the garbage collector to function without requiring **any** modification on the HTTP server or the web browser. It can be activated and deactivated whenever desired and, except for the item files, no difference can be perceived. This makes a more transparent solution.

7.3.1 Creating the local GC

We used the mapping model to create a local GC from our stand-alone `WebMS` and the DRC's generic GC. We call the resulting local GC `LocalMSDRC`. The model in Section D.16 shows what elements have to be added and/or modified:

- `EntryItem`s will be used by the local GC as extra roots.
- `ExitItem`s will also be handle by the local GC, but only to send reachability information to remote nodes.
- Network primitives (such as "sending a message").
- Network protocols ("decrement a remote counter").

We also chose to use the DGC server as a intermediate between the local GC and the rest of the world. Any remote message is first sent to the local server, before being forwarded to the actual remote node recipient. This is a precaution which allows us to control what is being sent in order to keep track of these actions in case it is needed (we will see in the next section that this is required for the GCW distributed collector). Consequently, our architecture looks like Figure 7.6.

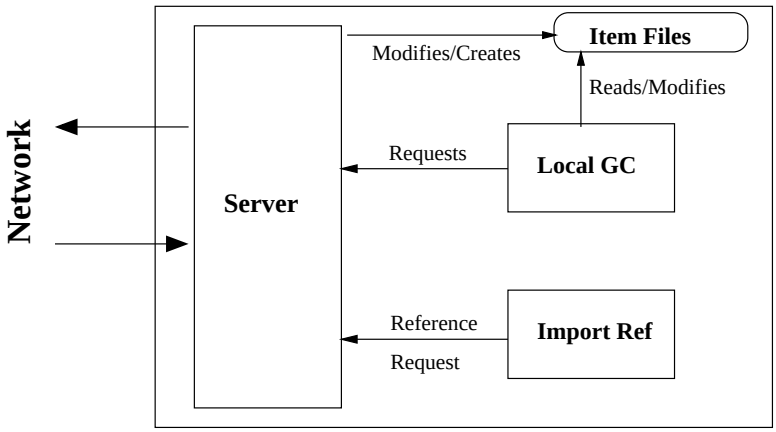


Figure 7.6: *DRC architecture.*

7.3.2 Classes

The classes required by DRC are shown in Figure 7.7 (opaque addressing), Figure 7.8 (server), and Figure 7.9 (local GC).

7.3.3 Reference Import

This external application can be used with any DGC and allows authors to request the authorization to include a remote URI in their pages. When the request is received, the DGC server checks the validity of the request and updates its vector of entry items (creation or increment). The answer to this request is a reference to the entry item. The class used to handle this action is shown in Figure 7.10.

In our architecture, this is accessible as an external program that is normally called explicitly. While this may be sufficient for simple websites, this is obviously not a good solution in general, because it requires a human intervention, which is error-prone. This can be solved by using a "daemon" that scans the local website regularly to find unregistered remote pointers and call the appropriate operation. Unfortunately, the delay in timing (between the moment the reference is published and the moment the reference is actually requested to the remote

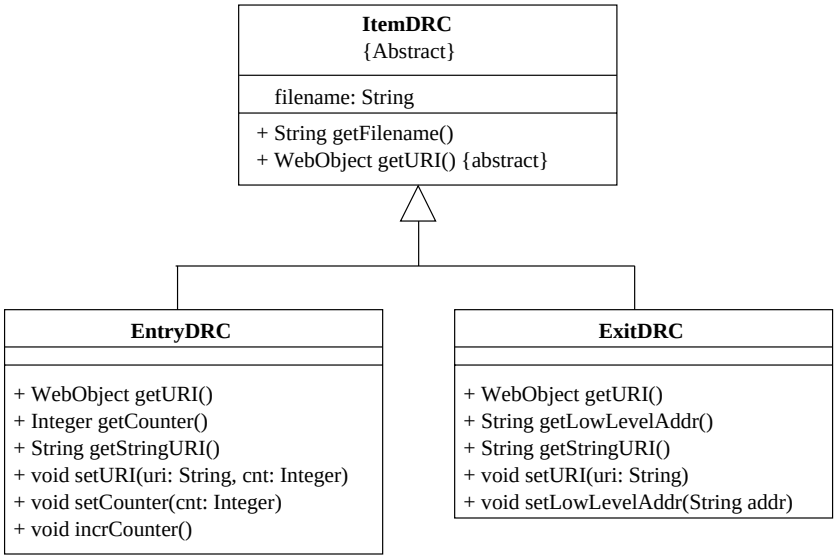


Figure 7.7: *Opaque addressing.*

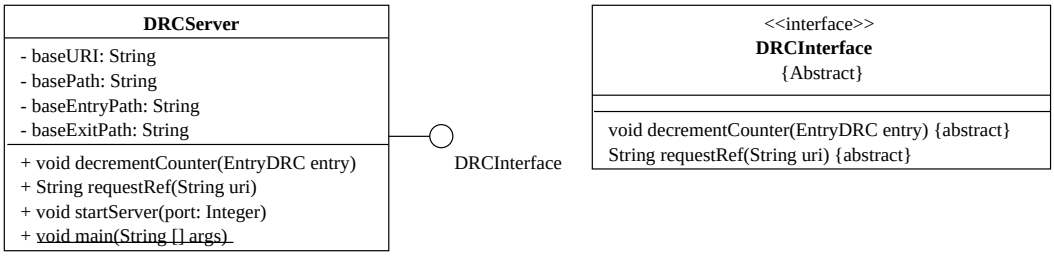


Figure 7.8: *DRC Server.*

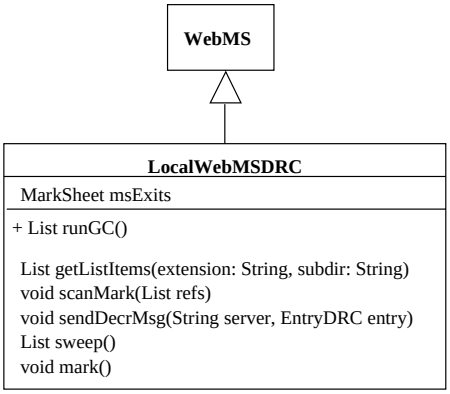


Figure 7.9: *DRC Local GC.*

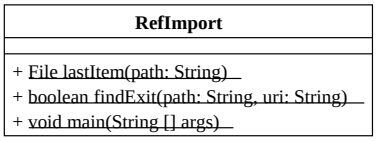


Figure 7.10: *Import references.*

server) might be problematic. It might happen that the reference is published, but the remote page is removed because no *known* remote reference to it was detected. Then, when the daemon notifies the remote server, the answer is that the page does not exist anymore. The author is warned by the daemon that a reference, which was recently added, became dangling.

A better solution would be to integrate the reference import software to a web authoring tool and rely only on this tool to take care of the links. The consequence is that a reference is requested when the author is trying to add it to the file, not at some random future time.

7.3.4 Specialization/Notes on algorithm implementation

In such an environment, keeping exit items costs disk space. This is not much, but it would be possible to avoid this by having exit items only when a local GC is run. To achieve that, we need to replace the decrement message upon discovery of the state of garbage for an exit item by a message similar to `LIVE` messages in the SSPC distributed collector [81]. For each remote site, we send the list of exit items that are still alive and counters are decremented on entry items corresponding to exit items that are *not* in the list received. With the DGC we describe next (GCW), it is not such a good solution, because marks on exit items are important and, from one local GC to another, they have to be kept around for stability detection.

An orthogonal solution would simply be to compress the files representing the items. However, in practice, we find that the cost is reasonable, because, in terms of timing and memory space, the scale of the Web is quite different from primary memory. Secondary storage is usually several times larger than main memory, and is not usually the limiting factor. The space cost of maintaining files for items is probably negligible. Time complexity is also likely to be somewhat negligible because web sites are manipulated by humans, which do not expect

speeds equivalent to primary memory. Furthermore, it would be reasonable for a local GC to be run once a week at a time when few people are likely to use the website (although the GC **never** requires the HTTP daemon to stop, ensuring availability of the website).

Finally, we observe that using opaque addressing is quite necessary to avoid many messages. If one node had 20 pointers to one remote object, 20 messages instead of just one would be necessary to specify decrements.

7.4 GCW

Based on a DRC algorithm, the GCW collector allows the reclamation of distributed garbage cycles. The DRC algorithm we designed in Section 7.3 can not handle cycles. We thus chose the GCW collector to work on top of this DRC. GCW uses protocols and algorithms that could be called an "asynchronous distributed Mark-and-Sweep" to identify and break distributed garbage cycles.

We first study the model of the GCW collector and identify the elements that should be specifically adapted to the Web environment. This model is a subset of the collector's features: groups and failure handling are not treated here. This is specified in the general description of the model. It is important to understand that these features are not modeled, in case the designer refers to the original paper [51]. However, if needed, it is fairly straightforward to derive a new model from the existing one.

We also observe that entry and exit items are reused from the DRC model. The GCW model simply specifies extra data, this allows us to reuse the classes we defined previously for those ADTs. An important element is the Distributed Termination Detection algorithm. The model specifies its purpose but no specific technique is given, leaving this choice to the designer. The remaining task is to adapt the local M&S already created for the DRC to the GCW collector's generic GC.

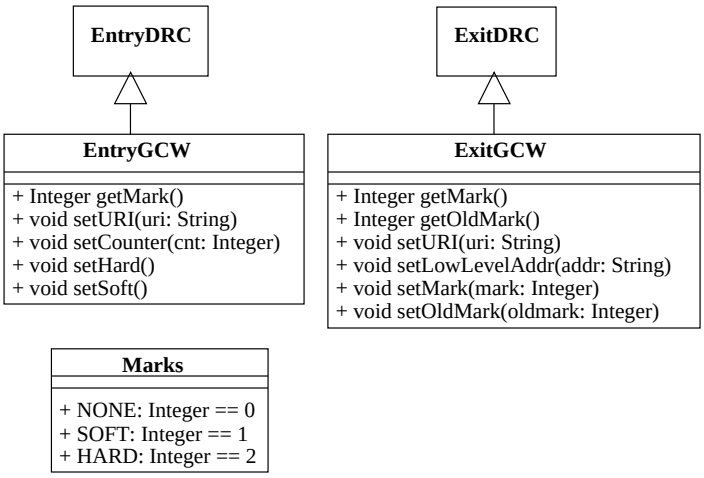


Figure 7.11: *Opaque Addressing for GCW.*

To achieve this, we study more specifically the following elements:

- Integration of local mark propagation to the local GC
- Computation of local stability
- Network protocols handling

We also use the mapping model described in Section D.1.

7.4.1 Classes

The classes required by GCW are shown in Figure 7.11 (opaque addressing), Figure 7.12 (server), and Figure 7.13 (local GC). We also update `RefImport` created for DRC to use GCW's items instead of DRC's items (Figure 7.14).

7.4.2 The Distributed Termination Detection algorithm

We chose to use the algorithm that is used by the DMOS distributed collector (see [37] and [8]). It uses a ring of nodes with a token passing mechanism. We see

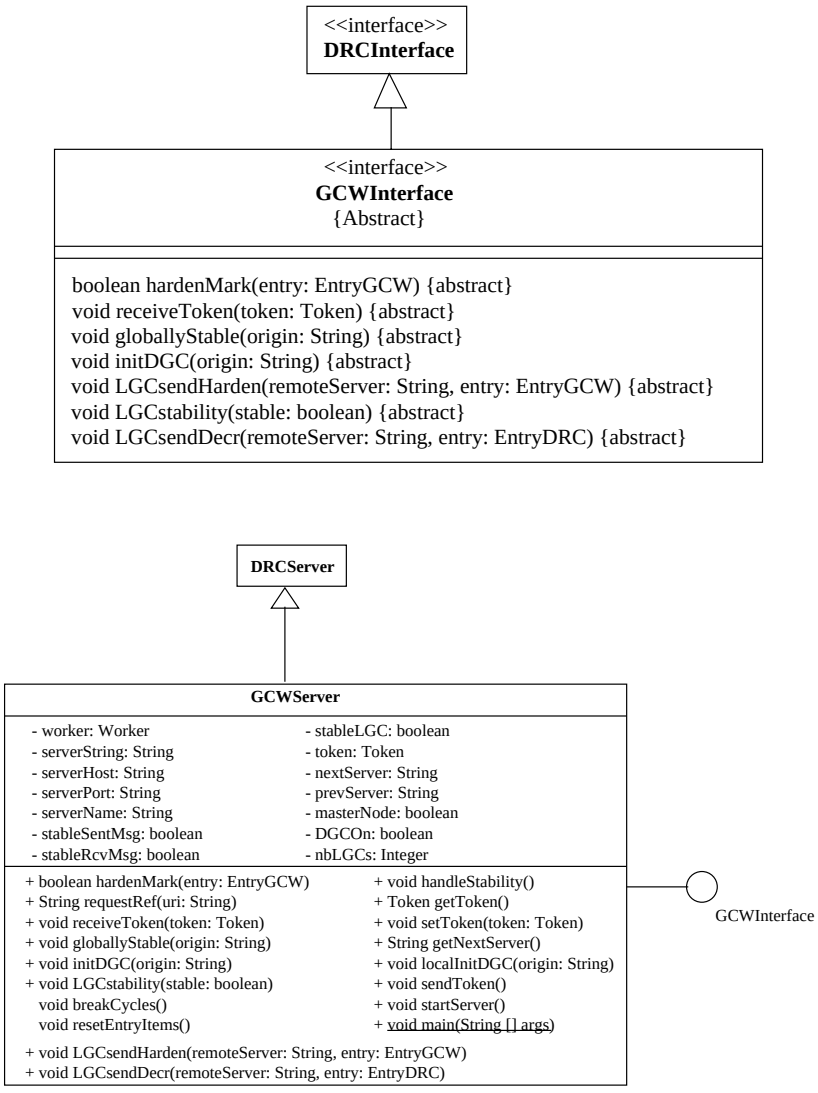


Figure 7.12: *GCW Server.*

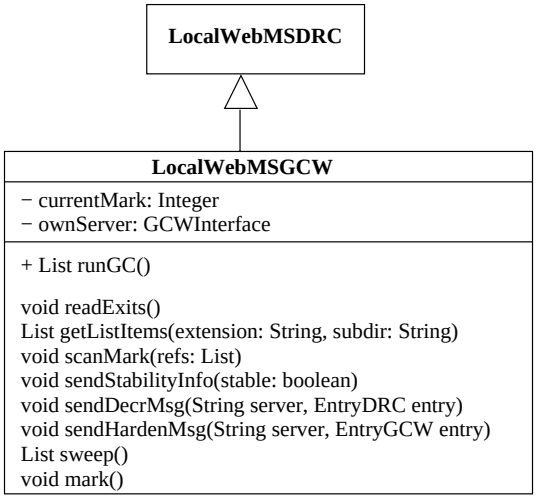


Figure 7.13: *Local GC for GCW.*

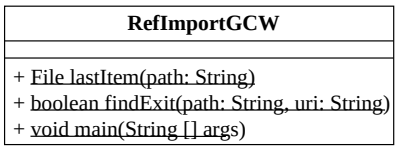


Figure 7.14: *Import references for GCW.*

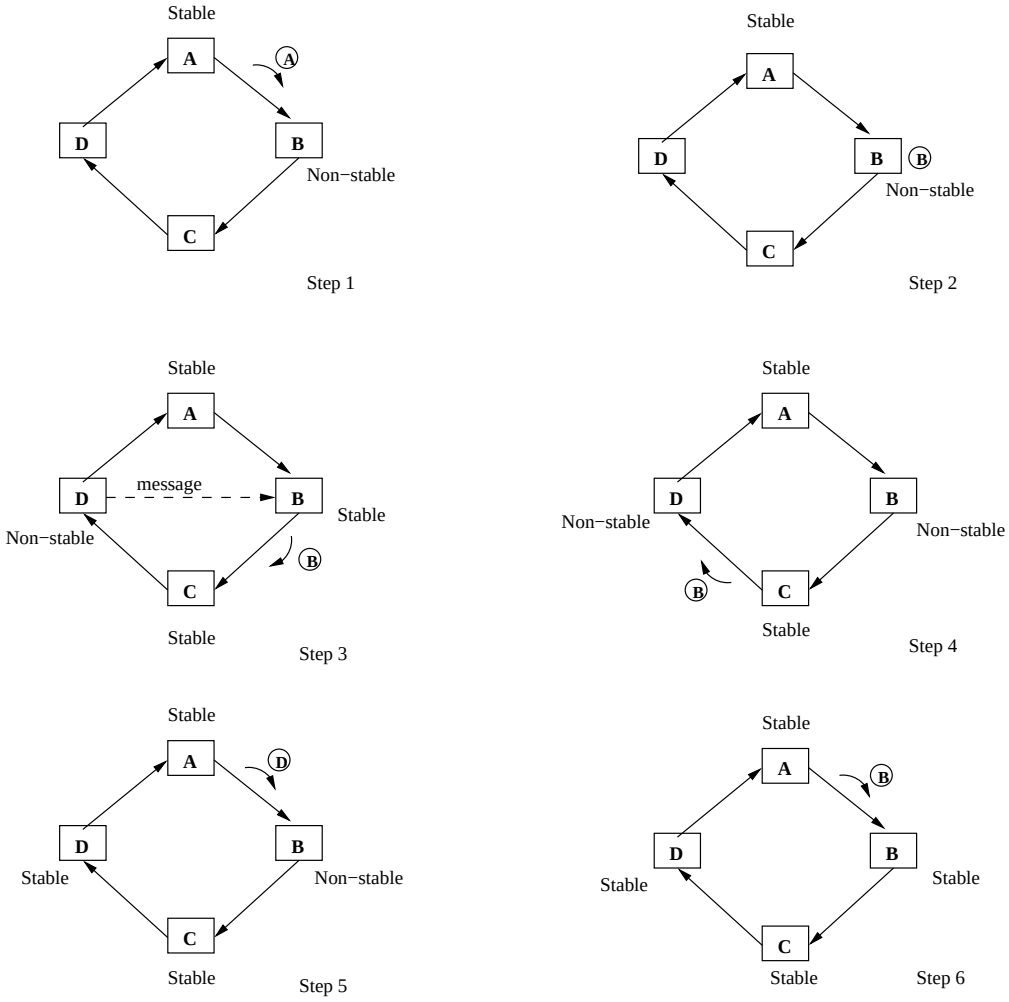


Figure 7.15: *Distributed Termination Detection algorithm example.*

from the GCW model that the purpose of this DTD is to find out global stability of marking, and that the criterion to pass on the token is local stability.

Example

In this example, we use four nodes and discover global stability of the GCW algorithm using a token passing DTD algorithm. The token is initialized at node `A` (this is an arbitrary choice).

The following describes each step in Figure 7.15:

1. `A` becomes stable and sends the token with value `A`. Node `B` is not stable when the token is received.

2. At node `B`, the token takes the value `B`, because the node is not stable.

3. Once `B` becomes stable, the token is sent towards C, which is stable at that time. At the same time, node `D` sends a message to `B` which will destabilize it. That is why `D` is noted as non-stable.

4. `B` becomes unstable, `D` is still unstable because it has not been visited by the token yet. Node `C` passes the token right away – without changing its value – because it is stable.

5. At node `D`, the token takes the value `D`, because the node was unstable. Node `D` is marked stable and the token is sent to node `A` which is stable and forwards it to `B`. Node `B` is non-stable and thus keeps the token for a while.

6. Once `B` is stable, it sends the token – with value `B` – to `C`, which forwards it to `D`, which forwards it to `A`, which forwards it back to `B`. Once the token comes back to `B`, distributed termination, and thus global stability, has been detected.

DTD for W3GC

We need to adapt the local collector to this algorithm. Here is the list of elements we add:

- DTD initialization.
- Token protocol.

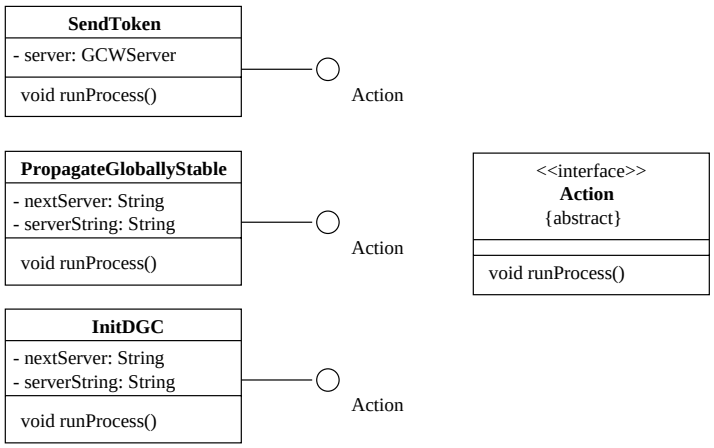


Figure 7.16: *Action abstract class.*

- Stability discovery computed on three criteria: stability after local GC, stability based on what messages are received, and on what messages are sent.
- New reference import. Marks on new entry items should be HARD to ensure safe termination.
- Global stability protocol.

The model in Section C.2 describes the details of the collector's conditions of termination for the DTD.

Classes

Classes shown in Figure 7.16 and Figure 7.17 are used to support the DTD algorithm. An `Action` is run asynchronously from the process. `SendToken` is such an action.

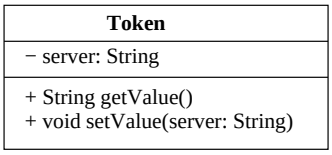


Figure 7.17: *Token class.*

7.5 Implementation issues

We present here some of the practical issues we encountered while implementing and debugging the collectors described in the previous sections.

7.5.1 Marking objects

A mark-and-sweep algorithm requires a mechanism to mark objects as live when they are visited. For example, in a primary memory collector, it is possible to set a bit in the header part of an object or a bitmap could be updated. In our environment, we can imagine modifying the file representing the object:

```
<!-- Marked -->
<HTML>
...
</HTML>
```

Of course, the problem is that this mark has to be cleared at some point (after the sweep phase or just before a new mark phase). This means that two disk accesses are required (one for marking, one for clearing the mark). This solution is problematic in that it changes file access and modification times of the file, and places greater reliance on file system robustness e.g. in case of power failure. Furthermore, it has the problems of modifying user data and changing file system modification dates.

Our solution uses only primary memory. Indeed, although objects might stay alive for years, a GC is short-lived. There is no need to keep any M&S information

in a file. We create an object called `MarkSheet` (as specified in the algorithmic model) which contains the list of addresses of live objects as we visit them. To "clear the bits", we simply get rid of the list.

7.5.2 Entry and exit items

Unlike mark bits, entry and exit items (see Section 2.5.1 for details) are likely to live for several years. Therefore, they need to be recorded in persistent store. We use a special directory located in the main directory of each website called `.dgcwww` with subdirectories `Entries` and `Exits`. Each item will have a unique name created using a counter (`entry1.ent`, `entry2.ent`, ...).

We note that, unlike in primary memory, these items are used only for garbage collection purposes. They are not required to access objects during mutation and visitors of the websites are still using direct access. We do not make any modification to the existing web objects, nor do we catch HTTP messages on-the-fly to treat them in a different way. Our solution is completely independent of the existing structure of web pages. Consequently, whenever an object is visited and a remote address is found, we must look for the corresponding exit item to mark it as live. Because this item is on disk, it is more efficient to import exit items in memory before a garbage collection starts.

We work with a model which stops local mutation to execute a local collection, so no exit item can be created while the GC is running. However, we can imagine that, if this was not the case, they could be imported on demand when a remote address is found but the exit item does not exist in memory yet (note that it is an error when a remote address exists without the corresponding exit item).

7.5.3 GCW-specific issues

A first problem comes from the fact that GCW is based on a distributed reference counting algorithm. When global stability for GCW occurs, we break distributed

garbage cycles by setting the counter of garbage entry items to zero. This is safe, because we know it is garbage. The subsequent local GC will reclaim zeroed entry items and will not trace from these items. It will consequently not trace the corresponding exit item(s) in the dead cycle, thus reclaiming them. Unfortunately, local GCs are not synchronized, and exit items on a node could be reclaimed *after* corresponding entry items are deleted on another node. When an exit item is reclaimed, a decrement message is sent. In the present case, it is sent to decrement the counter of an entry which may no longer exist. In this situation, the decrement message was as legitimate as reclaiming entry items. The only problem is that reporting errors becomes a difficult task, distinguishing between this situation and a genuine wrong DGC request.

A simple solution is to assume that errors at this level will never occur and ignore any decrement message sent to a non-existing entry item. Possibly, a response would be sent announcing the problem, but this should not be a high priority error message. Another solution might be to use a special mark for exit items that are SOFT when global stability has been detected and treat them differently when they are reclaimed by the next local GC. This is safe, because if an exit item was not supposed to be SOFT, it means that either it is pointed to by a local root-reachable object or by a HARD entry item. In these cases, global stability would not have been reached.

Another GCW issue is initialization. In our implementation, we do not set up any sophisticated negotiation structure to start a distributed garbage cycle detection phase. Instead, we declare one node to be the "master" node, whose only task – as a master node – is to start a detection phase. This solution is usually sufficient in environments where failures are fatal (i.e. the whole system stops once a node fails), and is thus adapted to intranet contexts. On the internet, a true distributed negotiation algorithm should be selected.

```
<W3GC_logs>
<date>2002/4/29(15:50:32)</date>
<runGC>
<mark>
<lk_int>/scl/people/chicha/public_html/index.html</lk_int>
<examine>
<file>/scl/people/chicha/public_html/index.html</file>
<trace>
<links>
<ext_lk>http://www.csd.uwo.ca</ext_lk>
<ext_lk>http://www.uwo.ca</ext_lk>
<int_lk>/scl/people/chicha/public_html/fun.html</int_lk>
<int_lk>/scl/people/chicha/public_html/yan2.jpg</int_lk>
<int_lk>/scl/people/chicha/public_html/me.html</int_lk>
<int_lk>/scl/people/chicha/public_html/fun.html</int_lk>
<int_lk>/scl/people/chicha/public_html/aa.html</int_lk>
<int_lk>/scl/people/chicha/public_html/interop.html</int_lk>
<int_lk>/scl/people/chicha/public_html/dangling_link.html</int_lk>
<int_lk>/scl/people/chicha/public_html/mail.gif</int_lk>
</links>
...
```

Figure 7.18: *Extract of a log file. We record dates, file examined, links found in the file, and so on.*

7.5.4 Debugging

Debugging our implementation was sometimes complicated. This motivated us to instrument our code, and generate event logs (see Figure 7.18 for an example). We also recorded regular snapshots of the test websites in a special subdirectory and used those files to follow each event from one snapshot to another to identify incorrect behavior.

Not having any tool at our disposal resulted in many tedious manual verifications. We checked snapshot consistency by verifying the following list of elements:

- Uniqueness of exit items with respect to a given remote pointer.
- One-to-one connection between entry and exit items.

- Correct counters on entry items.
- Only garbage cycles are identified as such (safety property) and broken.
- Correct reclamation of garbage files and exit items after garbage cycles are broken.
- Dead cycles created *during* a DGC phase should be alive at the end of this phase. Indeed, to ensure safety, entry and exit items should always be created with a HARD mark.

Certain design and implementation issues described in this chapter came to light while we were debugging our implementation. Although it might not be obvious at first, debugging such an application can be quite tedious without proper tools. This is the reason why we created the experimentation platform we describe in Chapter 8.

7.6 Conclusion

In this chapter, we have reported on our experience with implementing two garbage collectors for the Web. A Mark-and-Sweep algorithm has been implemented to handle single websites, and a hybrid distributed collector has been implemented for an intranet distributed context.

We also highlighted several issues and observations such as the need for a log mechanism, that we believe will be useful for future implementers of W3GC and distributed garbage collectors in general.

Chapter 8

A Platform for Experiments on DGCs

Tools to support research in garbage collection are quite rare. Experimentation environments can not be found easily, and it is even a challenge to find appropriate benchmarks for certain contexts (as we have discovered in Chapter 4). In recent years, a few tools appeared (see [61], [20] or [77]) to help with studies of garbage collectors. However, the set remains quite sparse, especially for distributed garbage collection. For years, garbage collection algorithms have been used to create software development tools (leak detectors, for example). Unfortunately, the favor was never returned, and GC research severely lacks tools such as appropriate debuggers, benchmarks, visual tools, and so on.

In this chapter, we propose to use the Web as an experimentation platform for both uniprocessor and distributed garbage collection research. The Web possesses interesting characteristics for experiments on these topics. We explain these in Section 8.1. Although we do not present a complete experimentation platform including software tools such as debuggers, profilers, and so on, we studied several fundamental elements. First, we derive a software development kit (see Section 8.2) from classes we presented in Chapter 7. In this section, we discuss

mutators and investigate logs and snapshots to keep track of important events during allocation, mutation and garbage collection. In order to illustrate the use of the experimentation platform, we report on a practical experiment (Section 8.4) about interoperability of garbage collectors in a distributed environment. This concept was explained in Section 5.4 in a previous chapter.

8.1 DGC research experiment platform

In this section, we discuss the need for a software platform targeted to Distributed Garbage Collection research. We explain how the Web environment is particularly well suited for this task and discuss tools that would be useful to add to this experimentation platform.

8.1.1 DGC research

Since 1960, garbage collection research has produced many efficient algorithms and interesting studies. Unfortunately, it lacks many tools usually provided to other areas of software development. Debuggers, profilers, standard benchmarks would be welcome in this essential area. There are efforts in this direction (Zorn's allocation benchmarks [36], Boehm's GCBench [9], GCSpy from Jones *et al* [76]), but these remain in limited number. Fortunately, empirical studies have been conducted to give researchers certain information about GC and mutator behaviors. We suspect, however, that programs tested do not reflect the complete spectrum of allocation and mutation behaviors one can find in a computational environment (as seen in Chapter 4).

After twenty years of algorithmic studies, distributed garbage collection only starts emerging in mainstream systems (such as Java RMIs [59] or PerDis [77]). This domain of memory management clearly lacks appropriate testing environments. It is difficult to find programs to test a collector with. Of course, one

reason is that distributed computing is still young and interesting applications only begin to appear. We would like to propose support tools for distributed garbage collection right away in order to avoid having to wait more than forty years like we did for uniprocessor garbage collection.

In this chapter, we describe a preliminary study of the problem. We propose current results and future directions for the development of an experiment platform for DGC research. Such an environment should be easy to customize as well as flexible, because the main activity – experimenting with garbage collectors – is a complicated task and should not be "polluted" by the details of manipulating the environment. While implementing a GC algorithm could be considered as a useful research work (especially for distributed collectors), the underlying environment should not constitute an obstacle but rather provide necessary support.

8.1.2 Web-based experimentation platform

Our work on a distributed garbage collection mechanism for the Web had an interesting outcome. We observed that, once we provided appropriate basic classes to model a primary memory context, we had access to a convenient platform to implement collectors. Objects and references on the Web can be manipulated in the simplest way. Furthermore, in a primary memory environment, many difficult-to-control parameters such as stack, cache behavior, virtual memory behavior, and so on, have to be taken into account. In a Web context, every single aspect of the memory simulation can be configured, created, or deleted *as necessary* . For example, the Web environment does not have any concept of "stack", but we could simulate one if our research requires it (e.g. by using a special directory to include "local" objects and an HTML file to maintain a stack order).

One challenge coming from the use of the Web is that real-life GCs usually deal with thousands or millions of objects. Our environment is unfortunately limited by the capacity of the file system. Timing/Performance studies would be

needed to assert the usefulness of the environment to a certain scale. However, it can certainly handle a thousand or a million network nodes. To lead experiments on DGCs, this is useful because performance of DGCs is often evaluated with the number of network messages involved by the algorithm. This is still valid whether the heap is represented as a set of files or as a block of memory (if we consider primary-memory programs largely using virtual memory, the difference becomes almost insignificant).

We also note that the Web offers several interesting features such as large scale, complexity and flexibility. These characteristics are very useful for research in distributed garbage collection. This is why we believe that such an artificial environment may be of benefit to perform certain experiments in both uniprocessor and distributed garbage collection contexts. To summarize, this platform has several advantages:

- Ease of implementation.
- Complete control over the graph of objects (no interference from a compiler or an interpreter).
- Human timings rather than computer timings, which allows for a better fine-grained understanding of certain behaviors.

This results in numerous possible applications:

- Testing and debugging collectors at an algorithmic level.
- Investigating high-level programming issues. This is especially useful with DGCs as they rarely focus on low-level problems, relying on local collectors for such tasks.
- Easy creation of tools such as testing tools, algorithm animation tools and so on (the Web already benefits from certain statistics tools that could be modified to be integrated in our environment).

- Teaching about memory management becomes more interesting with a platform that is both useful as an independent product and useful as an experiment environment.

8.1.3 Application contexts

We distinguish three axes of study using the Web environment for GC and DGC experimentation: one website, a controlled set of websites, and the WWW.

On a single website, one can test uniprocessor garbage collection strategies. It is also possible to experiment with different solutions to adapt uniprocessor code to a DGC's generic GC. Such high-level experimentation would help to quickly decide on what solution to choose for a final implementation in an actual environment.

Experimenting with DGCs on a collection of websites controlled by a single organism is likely to be the target application of the architecture we describe in this chapter. Indeed, such an environment would be close to an actual distributed application. With the advances related to the Semantic Web [6], it makes sense to foresee that a collection of websites interacting with each other would **be** a distributed application. In this case, experimenting with techniques to maintain link integrity is essential and we would benefit from an immediate testbed.

The WWW is very large and experimenting in such a chaotic environment is obviously complicated, if not impossible. However, as DGC techniques increase their scalability potential, larger and larger testbeds will be necessary. We envision that, at each step of the way, the WWW can provide sufficient support for experimentation. Experimenting in this context becomes a question of limiting the testbed rather than creating one.

8.1.4 Support tools

To support research about DGC in the Web, we propose three tools:

- a Software Development Kit to help implement collectors on the Web.
- Mutators. As we will see, these are needed because we can not rely on regular web mutation.
- Logs to keep track of various heap and network events.

This last tool is an important feature we have to provide for this experimentation platform. Preliminary experiments with W3GC convinced us that debugging and studying the behavior of a DGC by hand is not efficient (although feasible for small problems such as garbage collecting one user's website). Even though the log mechanism we used was quite precise, it was nevertheless made for human reading (using full sentences). We need a format that can be easily process by a computer. Chilimbi, Jones and Zorn [20] propose a trace format to keep track of heap-allocation events. We believe this interesting work could be reused and extended to keep track of GC and network events in a DGC context. The actual syntax for this format does not really matter – as outlined in Chilimbi, Jones and Zorn's work [20]. However, the success of XML makes this language a good candidate.

We can imagine many programs that could exploit a computer-formatted trace. Most of them are already in use in some other areas of computer programming. We simply propose a new dimension for those tools: memory management. Among them, we believe the following ones are of particular interest:

- **Verifier** of coherence between two snapshots of the heap by "running" the trace. Because we focus only on specific events (allocation, mutation and garbage collection), this activity should prove useful in pinpointing errors. Furthermore, this program does not have to use a sophisticated algorithm; a simple and correct technique can do the job.

- A **converter** between Trace and human-readable representation would be useful for debugging. We did not study the format and grammar of this "human readable" representation yet. This is future work.
- A **debugger** which can be used to follow the code as well as results between snapshots.
- **Algorithm animation**. It is important to be able to show the behavior of GC algorithms in class or during conferences. We can also imagine a tool that replays part of the events on a snapshot, to help us study consequences of different parameters of an algorithm. GCSpy [76] is the first example of such a tool for garbage collection.
- A **statistics extractor** could use traces of DGC runs to gather data in order to study behaviors and certain aspects of performance (number of messages for example).

Most of these tools can be used to achieve fair comparisons between two algorithms. Because distributed collectors usually rely on distributed termination detection algorithms, our environment could certainly help with empirical studies by providing a simple, yet powerful, mechanism to observe executions of such algorithms.

8.2 Software Development Kit

We lay the foundations of our experimentation platform by providing a Software Development Kit that abstracts Web elements. Using the semantic correspondence established in Section 6.3, we created three categories of classes: memory, GC, DGC. The first category is essential as it provides the basic blocks to manipulate Web documents as memory objects. The two other categories build upon the first one to provide automatic management of these documents.

An example of usage of this SDK can be found in Chapter 6 and Chapter 7 which describe a garbage collection mechanism to maintain web sites. We also note that this SDK is destined to work together with the design method we present in Chapter 5 to ease the implementation of distributed garbage collectors.

Finally, we provide – as part of this SDK – an implementation of a family of mutators that can be used to test and debug DGC implementations.

8.2.1 Memory elements

Most of the classes defining the SDK have been described in detail in Chapter 7. We simply list them here as a reminder to support our discussion about the experimentation platform. We effectively define an API that could help in the implementation of GCs for the Web and, thus, for our experimentation platform. We use an object-oriented approach to allow extension and easy implementation with languages such as Java which provide a rich library for file and network manipulation.

- `WebObject` represents an object on the Web. It is an abstract class, because we can potentially handle several types of objects. This is similar to languages where types are associated to values. On the Web, we can usually associate a web document to a particular type.

- `HTMLObject` handles HTML objects. It derives from `WebObject`. This class is particular in that an `HTMLObject` is able to scan itself to find what references are available. This type of Web object thus has the property of scannability. This is likely to be the only class that we will use with this platform, because we focus on GC research rather than W3GC research. However, adding objects with special characteristics could prove interesting and this flexibility is thus provided.

- `ParseTools` is a utility class, which exports functions to parse HTML files and find references they contain.
- `HTTPObject` represents references on the Web. It is similar to pointers in primary memory environment. An HTTPObject is basically a URL, and contains access protocol, server name, and path to a Web object.

These classes define an abstraction that can be used to implement various applications as well as memory management tools (region-based, reference counting, garbage collection). We observe that this SDK can be extended to include more memory elements such as stack, regions, and so on.

8.2.2 Garbage collection elements

For the purpose of experimentations, we implemented a Mark-and-Sweep, a Generational, a DRC, and the GCW garbage collectors. Please refer to Chapter 7 for a detailed description of their design. Of course, we integrated them into the SDK, as they can at least offer interesting examples of techniques to implement GCs for the Web. We distinguish several features that have to be handled in such GCs.

Basic functions

These functions are common to all GCs, and are fundamental functional components:

- **Handle a type of link**. Although the type of link is useful to know in a Web environment, the experimentation platform only deals with HTML files initially. This should be sufficient for many experiments with DGCs.
- **Scan** an object for links. In a *scannable* object, we try to find references to other objects, which may be scannable or not. Initially, this property of Web objects is not exploited on the experimentation platform.

- **Follow link**. From an `HTTPObject`, the referenced object is found.
- **Action on garbage**. It is a matter of policy to know what to do with garbage objects. As we have seen before, in primary memory, garbage is either reclaimed (added to a free list or given back to the OS) or marked for "lazy reclamation" (see, for example, [39]). On the Web, we have many choices; however, our experimentation platform is used for GC research, and we will remove all garbage objects, as in primary memory.

Specific functions

Collectors have specific needs, which can be expressed using our design models described in Chapter 5. We describe essential elements for several stand-alone collectors in the following list:

- A *mark-and-sweep* algorithm needs to store its marks. In a traditional environment, marks can be stored in the header of an object or in an external bitmap. On our platform, this is also an implementation choice. For example, we can reuse the class `MarkSheet`, developed for the tool described in Chapter 7.
- A *mark-and-copy* algorithm requires a **move** operation and a way to store or leave behind "forwarding pointers".
- A *reference counting* scheme is based on the ability to maintain a counter per page.
- A *generational scheme* needs an organization of the space into generations. Depending on the variant, it also needs *remembered sets* or *cards* data structures. In our two-generation design, we used specific files to manage young generation and remembered sets (remsets). This allows us to avoid moving objects and reuse our mark-and-sweep collector for each generation.

Other specific operations can be determined by the policy chosen to handle garbage objects: *remove*, *move*, *list in a file*, *email the list*, and so on. These specific functions should be handled by the implementation of the collector. We do not provide support for these, but, once implemented, it is possible to integrate them in the SDK (for example, `MarkSheet` has been added).

8.2.3 DGC elements

Distributed garbage collectors (such as [81], [37], [51]) often use opaque addressing. We thus need to provide supporting classes for this concept. The classes listed here are very basic, and needs to be derived into appropriate types depending on the DGC chosen. For example, we implemented GCW and had to derive these items to include a counter and a mark. Furthermore, if the mutator is configured to use these entities, local collectors and network communication will be easier to implement. However, the platform does not require it and this choice is left open.

- `EntryItem` only contains a reference to the object it represents. Inherited classes may add data such as a counter.
- `ExitItem` contains a reference to the remote entry item it represents.

Distributed Termination Detection algorithms are important to DGCs and we implemented one to support the "Garbage Collecting the World" [51] distributed collector. This is based on a token-passing mechanism and requires several classes. `Token` that is a singleton (one object only), `Worker` that launches a thread to asynchronously send the token using the interface `Action` and the class `SendToken`. DTDs are complex and diverse, we consider it outside the scope of this thesis to provide a standard API to implement DTDs in a DGC environment. A general model for DTD has been studied in [15] and [8] studies DTDs for distributed garbage collection.

In order to build DGCs, we can rely on the models and method described in Chapter 5. The basic idea is to use stand-alone collectors and transform them to simulate a DGC's generic GC. Extra operations may be added to handle a DTD and network protocols. In our implementation, we used Java RMIs [59] to handle network protocols. We also created a DGC "server" class for incoming network messages. The specifics of the server depend on the chosen DGC algorithm. However, object-oriented design can also used at this level, a DRC server can be a base class for many other DGC servers (such as GCW for example), because they handle decrement messages, which is a very common operation.

8.2.4 Mutator

One advantage of our experimentation platform is to have a "human scale". It becomes very easy to examine any part of the heap and the structure of any object. However, one drawback of the Web environment is that mutations are very slow. Usual web mutators are a person or a program. However, modifications and creations are made infrequently compared to mutators in primary memory. Experiments and tests of collectors might not prove very useful or representative in these conditions.

An artificial mutator created specifically for this platform could provide necessary support for useful research and testing. We thus propose a customizable *mutator* to create and modify objects at a more comfortable rate for real DGC research (comparable to mutation in primary memory environment). It already proved very useful to test and debug our initial implementation of W3GC to manage website by simulating authors of websites. We note that, although it has been implemented to work with our collectors, it is not a necessary component of the platform. This mutator could be replaced by any other mutation process (even manual). Our collectors would still be able to function adequately.

In [92], it is shown that synthetic allocation is rarely representative of actual

allocation behaviors. We need to take this fact into account and create a realistic mutator which would rely on recording actual program behaviors and simulating them in this environment. For example, Richer [79] recorded the structure of two websites at particular dates and used an allocation simulator in the PerDis system [77] to study allocation behavior of the Web. Until we obtain sufficient data about allocation and mutation patterns of distributed applications, we can rely on synthetic mutators to work with DGC implementations.

In this thesis, we created a customizable, artificial mutator which allows to observe the behavior of garbage collectors in the context of different flavors of mutation. The mutator we propose here is a preliminary work, which can be used for debugging and simple experimentation. More sophisticated mutation algorithms should be implemented to help perform more interesting tests. In particular, distributed mutation patterns or actual distributed applications could be selected. Our goal here is to show an example of what can be done.

We choose a mutator which would simulate a web author. In order to create web documents automatically, we define "units of text". These elements are the bricks – added, removed or modified – used to create or change web documents. A *unit of text* contains a number of lines of text and a number of references. We note that, in order to simplify parsing, each reference is located on a single line and nothing else is on such lines.

The mutator is parameterized by the following aspects:

- a number of lines per unit.
- a number of references (local or remote) per unit.
- the number of lines at the beginning of the file that should never be modified or deleted. This is to guarantee a certain number of lines in a given file. This allowed us to test websites containing files with possibly no contents and others with guaranteed contents. In practice, this is of little consequence.

- a percentage of chance to choose a local reference versus a remote reference when mutation requires a new unit of text.
- a percentage of chance to delete or add a new unit in an object when modifying an object.
- a percentage of chance to create artificial distributed garbage cycles. Preliminary tests showed that simple random modifications were not sufficient to create distributed cycles, we thus added a module to explicitly "allocate" web objects in an already distributed garbage cyclic structure.

We have three flavors of our mutator: "user", "news" and "project". Of course, any number of such mutators can be created. These mutation models use the following values:

Style	**Lines /unit**	**References /unit**	**Protected lines**	**LocalRef ratio**	**Add/Delete ratio**
News	2	1	2	10%	10%
Project	4	1	0	70%	20%
User	3	1	4	50%	5%

These values are arbitrary and try to reflect *one* possible behavior pattern in each style. This is not meant to be anything more than an example. Empirical studies would be useful to choose more realistic values. The percentages are explained like this: a news website is much more likely to reference remote pages than local pages, thus 10% of chance to choose a local reference, and we consider it usually has little chance to delete a unit from an article. Project pages usually refer to other pages in the project and rarely delete information. User pages usually contain as many references to remote pages as to local ones, and units are rarely removed.

We observe that these random modifications do not allow fair comparisons between collectors if used directly. An acceptable course of action is to run the

mutator using the ratio and record all events (using our log format for example). These recorded events and associated snapshots can then be used to replay the mutation while a new GC is invoked.

We also provide an abstract class called `Mutation` which takes care of common functions such as the general mutation loop. We parameterize general mutation with three values:

- `THRESHOLD_GC`: call frequency for the GC.
- `THRES_GARBAGE_CYCLE`: frequency of artificial cycle creation.
- `THRES_CREAT`: frequency for new object/page creation.

Modifications are done at every loop because we would like the mutator to show a lot of activity. As observed, this mutation loop contains code to call the garbage collector. Choosing when to call the collector is a matter of policy (or is possibly dependent on the DGC used, if any). Disk space comes cheap these days and it is impractical to try to fill up disk space before calling the GC. So, unlike memory environment, the GC can not be called when "memory" runs out. Instead, we call it after a certain number of mutation loops (chosen with `THRESHOLD_GC`):

```
// Main mutator loop
cntLoops = 0;

while (allowedToRun) {

   cntLoops++;

   // Run GC
   if ((cntLoops % THRESHOLD_GC) == 0) {
      makeSnapshot();
      List garbage = gc.runGC();
      continue;
   }
```

```
            // Mutation: artificial cycle
            if ((cntLoops % THRES_GARBAGE_CYCLE) == 0) {
               createNewCycle();
               continue;
            }

            // Mutation: webpage creation
            if ((cntLoops % THRES_CREAT) == 0) createNewPage();
            modifyPage();

         } // end while
```

Another important feature of our mutator is the artificial creation of distributed garbage cycles. We implemented a simple mechanism to create two-object cycles. However, any type of cycle can be created depending on the behavior we want to debug or study. That is a future direction for this tool.

Finally, when modifying a page and adding a reference, we need to choose this page and reference. A preliminary version of the mutator was randomly selecting files from the directory. This is obviously a problem if the file is no longer reachable. For example, making garbage objects reachable again could break the opaque addressing mechanism by making live again a remote reference that disappeared several local collections ago (along with the corresponding exit item). The solution is a function which randomly selects references only from reachable objects.

8.3 Logs and snapshots

Logging information about the behaviors of the collectors and the mutator allows us to debug, understand and observe. In this section, we report important events to log for debugging implementations and study behaviors. We observe that logs are dependent on the type of operations. For example, we want to know when a mark phase starts for a mark-and-sweep algorithm, but we need to know if we collect a young generation area if we work with a generational scheme.

We also remark that the level of details of logs can be different depending on the purpose of the logs (debugging, algorithm animation, and so on). However, a better solution would be to record events with maximum details and let the tool using the logs decide what to display. A more serious question is obviously what to report. For each event, a lot of data exists; choosing what to report can be a complex task. We did not address the question in this thesis, although we report the choices we made when we debugged our W3GC implementation.

Our experience reports on a stand-alone mark-and-sweep and generational algorithm as well as on DRC and GCW distributed collectors. We also present the snapshot mechanism we used to record intermediate states of the heaps.

Mutation

We consider the mutator as a tool which helps outline and emphasize the behavior of collectors. Although the purpose of our platform is not the study of mutation behavior, certain mutation events should be recorded in order to understand garbage collection behaviors. We record the following events (we specify elements that are reported with the event within brackets):

- *Create a new object* (filename, List of references)
- *Remove a reference* (filename, position in the file)
- *Add a reference* (filename, reference)
- *Get a local reference* ()

In order to simulate a mutator which also manipulates remote objects, we use a server for the mutation, for which we record the following events:

- *Export a reference* (remote server, reference, entry_item)
- *Get a remote reference* (remote server)

- *Create new exit* (exit name, reference to entry)
- *Artificial cycle creation* (server_origin, server_destination, (exit_origin, exit_destination), (entry_origin, entry_destination))

Garbage Collectors

Few common characteristics exist among stand-alone collectors. We record the initial **root set** and the **list of garbage objects** found and dealt with at each collection.

Mark-and-Sweep

We found several events useful to record in order to debug and observe the order of visit of objects during the collection. We record:

Marking start and end
Element is already marked
Element is found and marked
List of references found in Web objects
Path of the object visited by the collector
Object can not be visited (image for example)
Sweep starts and ends

M&S-based Generational GC

This generational collector is based on a mark-and-sweep algorithm (each generation uses such a GC). Consequently, we log the same events plus the following:

What generation is collected
Objects marked from the remset
Links not followed because they point to the old generation
Objects promoted at the end of collection

Allocation and mutation are different in a generational environment, so we also need to log such events as:

Newly allocated objects
Reference added to a remset

Distributed collectors

We record the following elements common to distributed collectors relying on opaque addressing: **list of entry items used as roots by the local GC** and **list of garbage exit items**. The order in which entry items are visited might be important and thus must be specified in the logs as well.

DRC

A distributed reference counting scheme generates very few events. We record: **Counter increment**, **Counter decrement** and **Counter decrement for abusive increment**.

The latter event might be needed depending on the way references are exported. Normally, a mutator should scan its exit items to find out if a particular reference on a remote object is already available. If this is the case, the event we listed will not occur. Another solution can be used if the mutator does not know the reference it desires but asks a remote node for a type of objects. This happens with CORBA where clients can request an object with a specific ability. In this case, the remote node exports a reference to this object and thus increments the counter of the corresponding entry item (for safety). Once the reference is received, an exit item is set up at the recipient node. However, if such an item was already present, it means that the counter on the entry item was "abusively" incremented and needs to be set to the proper value, hence this event resulting from a DECREMENT message.

GCW

This DGC is more complex than a distributed reference counting scheme. Being based on this technique, it records the same events and adds new ones.

Details about the root set. Order in which local roots, soft entry items and hard entry items are visited.

Local propagation. We need to know what mark is propagated from roots and entry items to exit items (SOFT or HARD).

Finalize an exit item. With respect to its mark, three actions are possible: (NONE) remove exit and send a decrement message, (HARD) propagate the hard mark to the corresponding entry item, (SOFT) do nothing.

Local stability information. We record the status of a node after a local GC (stable or not).

Global stability. We simply record this situation as soon as it is discovered.

DTD. When we need a termination detection algorithm (which is the case for most DGCs), we record the events of the detection process. For the GCW collector, we log the following elements: *initialization* (when, what is done locally, ...), *local stability* information, *propagation of stability information*.

Snapshot

Snapshots of the heap are very useful for debugging and algorithm animation, because they record the history of data structure evolution. Pointers are added and deleted several times between collections, and recording the state of the heap allows us to trace more easily specific sections of an execution.

A snapshot of the heap records all web documents and extra information such as entry items and exit items. It also records GC-specific structures such as `remset` files for a generational GC (see Section 8.4). Depending on the size of the site, snapshots might represent an important job to do. In practice, we do not believe it to be a problem, because a test platform for garbage collection research

should only use HTML files, which are usually pretty small.

Recording the time of a snapshot indicates the checkpoints of the execution. This helps for debugging as well as observing behaviors. We remark that the contents of the snapshots can be quite different depending on the collectors and behavior of the mutator. This is because different techniques may require different structures of the heap. For example, a generational collector needs to save different generations. Also, collectors usually use specific "maintenance files" (remembered sets for example). Consequently, each local GC should provide its own format for the snapshot (what to save, what are the names of the files, and so on). We note, however, that interconnection of data structures are preserved because this is not dependent on the collection algorithm, but on mutation.

8.4 Interoperability experiments

One of the products of this thesis is the definition of interoperability of garbage collectors in a distributed garbage collection environment. We consider complex distributed computations where measurements have been done and memory activity is known. For each node, we choose the best local memory management according to foreseen activity. This may result in an heterogeneous system, using different stand-alone collectors. With respect to this topic, our work in this thesis was to define methods and tools making it possible to choose a DGC and create a cooperation framework between these chosen collectors.

In this section, we present our preliminary study on possible software experiments on the topic of GC interoperability. We believe the W3GC-based experimentation platform can help provide interesting statistical observations as well as implementation experience reports. This work is only a preliminary study and actual experiments have not been performed. We report on our practical experience with the implementation of actual collectors. We remark that debugging tests, using the mutators described in Section 8.2.4, were made that helped us

refine our implementation, which is one of the rare heterogeneous implementations of a distributed collector that we are aware of. Consequently, we believe this practical experience report to be of value. Furthermore, as we will see in Section 9.2, experimentation with DGCs are quite awkward because mutation patterns for distributed applications do not exist yet.

In this implementation experiment, we reuse our implementation of Mark-and-Sweep, DRC and GCW (see Chapter 7) and add a generational stand-alone collector. We also provide an adaptation of this GC to work as a local GC in the "Garbage Collecting the World" environment. The following paragraphs present our experience with implementing a generational GC in a Web environment, and adapting it to work with GCW, and experiments about heterogeneous configurations within a GCW context using mark-and-sweep and generational collectors.

Stand-alone Web-based generational GC

Our implementation of a stand-alone generational GC for the Web makes use of the SDK defined in Section 8.2. We note that, even though we tried to provide a flexible environment to easily choose the garbage collector to experiment with, generational GCs are somewhat invasive due to their need to allocate objects in a special area called the *nursery*. This requires changes to the mutator code and involved creating mutators using a "generation-aware allocator". We also would like to point out that we reused our Mark-and-Sweep collector to handle each of the generations. We use only two generations in our example: a young generation and an old one. For the young generation, the mark-and-sweep collector was made aware of generations and remembered sets. However, because a collection of the old generation looks at the whole heap, we could reuse directly our mark-and-sweep collector – without modification – in this context.

A traditional distributed memory setting uses opaque addressing, which means that when an object is moved, its entry item is updated and nothing has to be

done for remote objects referencing this now moved object. Unfortunately, we do not have this luxury in the Web environment, where pages are referenced directly. Indeed, a natural idea would have been to use a young object directory and an old object directory and move files from one directory to another. To avoid moving objects, we decided to list them in a file. The filename of any young object will be added to the "young generation" file.

This has also the advantage of allowing very fast object promotion. In our case, we simply empty the file. Indeed, any file whose path is not recorded in the youngGen file is considered old. To promote objects from young to old, we just need to empty the file. This assumes that an object has to survive only one GC to be considered old. If several GCs are required, some kind of counter or several generations are needed to solve the problem.

Similarly, we use a remembered set and save it in a file. When a reference is added to a file, we check if the file is old and if the reference is to a young file. If this is the case, we add the name of the referenced young file to the `remset` file. This file is emptied once young objects are promoted. It is interesting to note that we can simplify the code for newly created objects. We initialize objects with text and references to "simulate" a real environment. In this case, there is no risk of old-to-young reference, because the object is the youngest one so far. This means that we do not need to call the routine verifying that references inserted in the object are to young objects.

Another observation is that once a reference to an object has been added to the remembered set, we do not remove it before the next complete GC. Precision would require the entry of a remembered set be removed as soon as the corresponding old-to-young reference is deleted, allowing the next collection in the young generation to possibly collect more objects. However, handling this problem would involve, for example, a counter of references in the remembered set. This would result in a more precise, less conservative solution but it would be

less optimal as well. Indeed, instead of using a barrier on the module which adds references, we also would have to set one up to control references removal. This would involve extra information to maintain for each entry in the remembered set, which can be costly.

Local Generational GC for GCW

Like for the Mark-and-Sweep GC we implemented for the Web, we need to adapt our stand-alone generational GC to the needs of the DGC. Using the design method we developed in Chapter 5, we list the different elements that we have to modify or create to allow our WebGen to become a LocalGenGCW (see Section D.2 for a model of the mapping). The first remark we make is that a collection of the old generation is actually a collection of the entire heap. This allows us to reuse our LocalMSGCW without any modification.

Adapting a generational algorithm to work with a DGC which normally requires a full visit of the heap necessarily calls for compromises. Two points have to be treated: reclamation of exit items and mark propagation. In our simple solution, we decided to let each collection of the old generation do all the work. However, we use young generation collections to help. Indeed, even if we can not reclaim exit items after an young generation collection (an exit item can be pointed to by an old object), we can still reclaim dead entry items (whose counter is zero) and propagate marks. A possible solution to be able to reclaim exit items after young generation collections would involve managing a remembered set for those items. The design and implementation we describe is a simple experiment to understand what practical problems we have with GC interoperability. Consequently, we did not implement any improvement such as the one we just described.

To propagate marks, young generations are useful. Indeed, HARD marks can be propagated to remote nodes right away (once an item is HARD, it can

not become SOFT again) thus speeding up global propagation. Of course, local stability can only be decided after a full GC of the heap, because all exit items are not seen by a young collection. We need to be careful not to let the young collection interfere in the process of stability evaluation. Particularly, marks on the exit items should be handled carefully. The design we propose skips initialization of such marks at each run of the young GC, thus preserving the "oldmark" field. This means that from one old GC to the other, "oldmark"s are the same. Also, the only modification we make on exit items is to possibly *harden a mark*. A special marker should be updated to inform the old GC that messages have been sent by the young GC. The next old GC may find an apparent stability, but should take appropriate action (here, change the value of the token) to take the young GC actions into account.

We note that we do not evaluate the reachability of exit item objects because they also depend on the old generation. We could optimize this aspect by implementing a remembered set for each exit item to find if an object in the old generation points to the item, if not, the young collection can safely evaluate reachability.

Furthermore, propagating marks poses a question: what mark can we propagate from a remembered set? For young generations, remembered sets are part of the root set, consequently a natural approach would be to propagate a HARD mark. This is conservative but safe. However, we might want to propagate a SOFT mark to avoid slowing down the dead cycle detection. We still ensure safety because reachable objects will not be reclaimed and stability detection is done only by an old generation collection so stability will not be detected when an item is marked SOFT whereas it should be marked HARD. It would be useful to run some tests and obtain statistics about what situation is the most interesting for a given application: propagating HARD marks to speed up stability or SOFT marks to increase the number of cycle detections.

Possible software experiments

Although we did not perform software experiments beyond designing and implementing our W3GC, we describe here an experiment that we might wish to make in the near future. The purpose is to study the behavior of an heterogeneous distributed garbage collection system in an intranet environment. An interesting test configuration would be to use 100 network nodes, dispatched on 20 machines. In our department, we can use computers based on Intel Pentium III with 256MB of RAM and Sparc workstations. It would also be possible to run the experiment on a cluster of machines.

We believe the following garbage collection configurations would allow for useful results:

- 100% of the nodes are using a Mark-and-Sweep algorithm.
- 100% of the nodes are using a Generational algorithm.
- 50% – 50%
- 25% – 75%
- 75% – 25%

Using various distributed application patterns, we could study the differences in terms of performance with these scenarios. This would help system designers understand what would be the best strategy with respect to the types of applications they wish to run.

8.5 Conclusion

This chapter describes a possible extension of our Web-based garbage collection mechanism. Relying on the simplicity of this platform, we propose to use a set of practical tools to create an experimentation platform for garbage collection

research. Although this work is still in progress, we provide observations about different components of this platform: a software development kit to help implement garbage collectors, a mutation mechanism, and a list of important events to record to observe the behavior of garbage collectors. We used this platform, in conjunction with the design method described in Chapter 5 to implement a generational local GC for GCW, for which we had already implemented a local Mark-and-Sweep collector. This work illustrates, in practice, the notion of interoperability we described in this thesis.

Chapter 9

Conclusion

9.1 Summary

Garbage collection has reached a certain maturity and has gained acceptance in widely used environments. Automatic memory management becomes vital for complex applications, because it is difficult to track down object usage. In new contexts such as distributed systems, garbage collection should prove even more useful, because objects can now be referenced from other processes. Manually keeping a record of all references becomes overly complicated, and resources allocated to design, implement and debug applications include a large part for memory management. Creating distributed applications can be complex without the extra burden of managing memory; adding such an important task could actually slow down the development of these systems.

We believe distributed garbage collection should play a central role in the distributed world. Currently, only distributed reference counting is implemented in certain systems. As networks become faster, applications will export more objects and manipulate them on several nodes without taking any special action about performance. Cycles of objects will then become frequent and DRC algorithms will no longer be appropriate. For several years, DGC research has provided

interesting algorithms handling cycles. Unfortunately, they have not gained acceptance in the real world yet. We believe that one reason is the difficulty to understand and evaluate these collectors.

In this thesis, we propose to study interactions between garbage collection entities and the overall strategy as a mean to better understand the nature of distributed garbage collectors. The study we provide explores uniprocessor collectors and multiprocessor techniques before addressing distributed GCs. We conclude that these interactions appear in the form of requirements expressed by the DGC that local collectors have to fulfill. A DGC is a collection of local GCs that communicate in some way, usually with network messages. These requirements are listed with more or less success in the literature because there exists no standard mechanism to record them. For that purpose, we introduced the Generic Garbage Collector, which acts as a template describing the "perfect local collector" from a DGC point of view.

Our work on interactions in garbage collectors led us to consider an optimization technique for the tracing process we find in GCs such as mark-and-sweep and mark-and-copy. While these collectors are not composed of several processes, we organize the heap into regions and create logical entities to take care of the tracing process. We call this technique Localized Tracing Scheme. When appropriate parameters are chosen, it is possible to improve either caching or paging behavior with the same algorithm. The heap is divided into regions. The core of this technique is to traverse as many objects as possible in a single region, discarding pointers to far objects. Pointers to far objects are saved in a special structure called trace queue. There is one trace queue per region. This technique effectively preserves the working set for a longer time than usual, and thus reduces the number of cache misses and page faults. Experiments showed up to 75% improvement when the LTS is optimized for paging behavior.

Our observations on GC/DGC interactions also led us to a design method for DGCs. We propose the sketch of a design process in which this method could be integrated. The purpose of this method is to help designers create a memory management solution that is the most adapted to their needs and resources. Instead of relying on distributed collection algorithms that dictate seemingly inflexible rules, we propose to list characteristics of all actors of the system, in order to study the best compromise. We created templates to help list these characteristics (for stand-alone and distributed collectors) and extended the template of the Generic GC to integrate it to the design method. Models created from these templates are useful to understand how local collectors can be created according to this context. Furthermore, most distributed systems use a homogeneous memory management model. However, we have seen, with projects such as CMM [3], that different parts of an application may require different memory management techniques. In this case, distributed collectors have to handle several different types of local collectors. Our method allows to naturally solve this problem using the notion of Generic GC as a contract for each type of collector to fulfill to help the distributed collector in its task.

In this thesis, we also explored the possibility of using a distributed garbage collection mechanism to manage web pages. This project had two aspects: (1) it is an interesting and non-trivial environment to test our design method, (2) the Web environment is currently the largest distributed system available. One of the problems in distributed garbage collection today is the lack of implementations. We believe that more implementation experiences are necessary to record various challenges people can face in a practical context. The Web is an environment containing documents and links between those documents. Garbage collection can be used to detect unlinked documents and avoid dangling links (known as the HTTP "Error 404"). In this context, history is repeating itself. Authors are reluctant to use automatic tools to manage their documents, although they

agree to use development environments to create web pages and URLs between them. We hope that our research on the different aspects of the problem will help garbage collection gain acceptance in this widely used distributed environment.

In the course of this thesis we have made a number of implementations and experiments, including:

- The *LTS* in C for the Aldor language. Experiments were led in the context of virtual memory intensive applications.
- Using Java, a uniprocessor Mark-and-Sweep collector, a Distributed Reference Counting DGC, and the "Garbage Collecting the World" algorithm. These implementations allowed the following experiments in the context of the *World Wide Web*: UWO websites, single websites in our laboratory, network of users websites in the department, Java API documentation.
- A Generational GC for the distributed collectors described above. This work is a preliminary test of our results on *GC interoperability* in a distributed context. This also confirms that W3GC as a *platform for experiments* is useful for GC research.

Through the work described in this thesis, we discovered several important properties in the world of distributed garbage collection.

First, local garbage collectors are intimately linked to specific stand-alone GCs and distributed GCs. More precisely, any given local collector is **linked** to one stand-alone GC and one DGC. Although this property seemed to have been implicitly known in the literature, it has never been explicitly studied. Through the introduction of the "Generic GC", which represents a DGC's needs and provides a template to adapt a stand-alone GC, this thesis proposes an explicit link between these three entities (local, stand-alone and distributed GCs).

An important property of future computer applications is **heterogeneity** of memory management needs. This thesis proposes a possible solution through the study of interactions between garbage collection entities and the definition of the generic GC.

Finally, we emphasize and study a property of the **World Wide Web** which is its similarity to a traditional distributed memory environment. This thesis establishes a semantic correspondence of notions in both worlds and creates a mechanism to implement and use distributed garbage collectors in this context to help with link integrity management in hypertext environments.

9.2 Future directions

We now present future research directions related to the topics we discuss in this thesis. Two of these directions have already been detailed in the report because we made preliminary studies of them.

Future direction: a tool for the design

This future direction is oriented toward software development rather than research. Our design method uses templates to list characteristics of collectors and thus create their models. We believe that these models could be reused in many cases, allowing, in particular, the reuse of mapping solutions to create local collectors. A future useful development would be the creation of a tool based on these templates to record design models. We can imagine that such a tool would store mapping solutions in a database and would allow searches based on specific keywords. For example, we could look for "handling local propagation for a migration-based DGC". The answer would retrieve all such mappings found in the associated database. If this tool were to be integrated to a design tool, for UML or OMT for example, a well-known syntax could be used to formalize fur-

ther the templates and allow designers to create their DGC solutions in a familiar environment.

Future direction: correctness proof model

Little work has been done on a formal model for garbage collection, supporting correctness proof of GC and DGC algorithms.

In [8], Moss presents a method to transform uniprocessor garbage collectors into distributed GCs by using distributed termination detection algorithms. One advantage of this technique is that correctness proofs are made easier, because they rely on existing proofs for the chosen DTD algorithms. Our technique – based on the notion of Generic GC – can provide similar leverage because it divides the task of proving correctness into different sections and relies upon assumptions that non-proved parts are correct. This allows reusability of proofs and formal models.

As explained in Section 2.5.5, the work published by Ungureanu [89] is an interesting approach to the problem of proving DGC correctness. It defines building blocks to express algorithms and prove their correctness. However, the resulting calculus may not be flexible enough to efficiently handle heterogeneous systems (i.e. different local collection techniques participating to a single DGC), gathering low-level and high-level elements in the same model.

We believe that the Generic Garbage Collector could help simplify the process of correctness proofs. The GGC was designed to separate concerns in a DGC environment. In the same way, the GGC can be used to organize correctness proofs of distributed collectors into two independent steps: (i) proving the correctness of a DGC algorithm assuming its local collectors correctly behave like the generic collector, (ii) for each local collector, prove that the emulation of the generic GC is correct. This architecture also allows to add a new type of collector to the system with very little cost. Proving the correctness of the new DGC system

(which would now be an heterogeneous DGC) would simply be a proof that the newly integrated collection properly emulates the generic GC.

Finally, we observe that, in the literature, it is generally understood that correctness of collectors means completeness (all garbage is reclaimed) and safety (only garbage is reclaimed). While the latter is ensured in all GCs encountered in this thesis, the former (completeness) is obviously not guaranteed by reference counting mechanisms. However, stand-alone and distributed reference counting (and listing) algorithms are generally accepted as correct. This inconsistency is easily fixed by integrating, in the description or model of each DGC, a list of properties it claims to support. Consequently, this DGC would be correct if and only if each of the listed properties is proved correct. This distinction would help clarify the characteristics of DGCs, and improve acceptance by non-experts.

Future direction: experimentation platform

Chapter 8 reports on our preliminary work on a Web-based experimentation platform for garbage collection research. While we implemented a distributed collector for the Web, we observed that this platform had interesting potential for many experiments. Its architecture allows both small and large scale experimentation, and, with proper support, GC and DGC implementation proves quite simple (there is no compiler interference).

In our preliminary work, we described a software development kit to implement collectors, a customizable mutation mechanism to observe garbage collection behavior with respect to different allocation and mutation patterns, a list of events we found important to record during our first experiments with the collectors. We implemented the following collectors: uniprocessor mark-and-sweep, uniprocessor generational GC, distributed reference counting, and the DGC known as "Garbage Collecting the World". With this experimentation platform, we could test our theory about interoperability being facilitated by our design method. The

M&S and Generational methods were adapted into local collectors for "Garbage Collecting the World". What remains to be done for this platform is: other implementations, tools to exploit the logs, and a comprehensive study on mutation patterns in distributed environments.

This study will be essential to our platform because understanding DGC behaviors can only be done if appropriate mutators are available. On this topic, the current state of the art is very limited (and we believe this thesis brought a non-negligible piece by identifying the Web as a suitable environment).

Future direction: experiments

During this thesis, we have led experiments about the LTS using Aldor (see Chapter 4) and about the structure of websites using W3GC (see Chapter 6). While we obtained interesting results, we had envisioned several other tests.

For the LTS, we obtained results at virtual memory level. We plan on pursuing these experiments on four axes:

1. We defined a family of tracing algorithms in Section 4.2.2. Experiments are required to explore the different possibilities. These would allow us to deduct recommendations about the optimal tracing scheme with respect to given allocation and mutation patterns.

2. Although results for cache level tests were not very encouraging, we believe that the LTS can be used for a certain range of applications. A future work would be to define these mutation and allocation patterns and perform experiments on those.

3. The LTS appears to help at different levels of the memory hierarchy. A future work would be to experiment with large applications on hand-held computers. Although memory is becoming larger, common sizes are still quite small (32MB or 64MB of RAM) compared to desktop machines. We

can imagine an application using a remote server for its virtual memory. Pages could be swapped over the network. This obviously requires new strategies on the mutator side, but the LTS already proposes a solution on the collector side.

4. Finally, testing the LTS on real-life contexts would help us assess the practical value of this tracing optimization.

Our experiments with W3GC have been oriented towards empirical results. It would be interesting to place the collector in a context of real use, by integrating it to a piece of web authoring software, for example. We could observe the behavior of users and report their critiques. This would certainly be valuable to the tool by itself, but we can imagine that certain aspects of the environment might also trigger new ideas in the field of primary memory distributed garbage collection.

Furthermore, website mutations should be recorded over a long period of time to allow more realistic experiments. Using a logging technology, such as the one described in Section 8.3, we can replay these mutations faster and study the behaviors of various garbage collection mechanisms.

Finally, emerging technologies such as the Semantic Web [6] would certainly benefit from the results of this thesis, as it is likely that such environments require sophisticated object reference management. The Web environment is used more and more as a platform to support many types of applications. Automatic (i.e. non-human) users can set up URLs to various documents; such links should be managed efficiently. Distributed garbage collection is likely to be a solution to this vital problem.

Future directions in Garbage Collection

While this thesis focused on certain areas of garbage collection, we have still been able to observe certain aspects of the topic in general. We believe that future

research and development work in the field of GCs will/should include the three following topics.

Standard **benchmarks** are difficult to find for uniprocessor GCs and simply do not exist for distributed collectors. We had to develop a synthetic test suite for the LTS, because we did not find any benchmark that could display appropriate features. In particular, GCs are usually tested with applications that entirely fit in RAM. While most applications do, this may not be the case in all areas. Applications in certain fields involve important computations, which require very large heaps. Little work has been done in this area and appropriate benchmarks are missing. Furthermore, distributed computing is a young application field, and it is difficult to find distributed applications to test a DGC. We believe that the Web-based experimentation platform, described in this thesis, is a step in the right direction. However, there is still a lot of work to do in this area.

One observation we made while testing the LTS and W3GC is that little research has been done on allocation and mutation patterns. Even if benchmarks exist for uniprocessor GC, it is a complex task to understand the behaviors of these programs with respect to allocation and mutation. In [17], an interesting study was led, but it misses an important component which is the evolution of the heap structure, showing how objects are mutated. We believe that measurements of various applications both at allocation and at mutation level would be beneficial. This survey should include distributed applications. Algorithm animation tools such as GCSpy [76] would certainly help with this study.

Finally, implementations of DGCs are rare. We hope that the results of this thesis will motivate further work in the area. The next step is to implement various collectors in environments such as Java RMIs or .NET, and report problems, solutions, and observations. In summary, we need a survey of DGC implementations in real-life environments. We believe that this work is one of the necessary conditions to a popular development of distributed garbage collection.

Bibliography

[1] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Collection schemes for distributed garbage. In *Proceedings of International Workshop on Memory Management*, University of London, UK, 1992.

[2] C. Anderson and J. Lennon. Maintaining link integrity to external web sites in a Hyperwave-based learning environment.
http://ddi.cs.uni-potsdam.de/HyFISCH/ZieleWerkzeug/. HyperwaveDokumentation/LinkIntegrityLennon.htm
Valid on 2002/07/19.

[3] Giuseppe Attardi and Tito Flagella. A customisable memory management framework. Technical Report TR-94-010, International Computer Science Institute, Berkeley, 1994. Also Proceedings of the USENIX C++ Conference, Cambridge, MA, 1994.

[4] Henry Baker. Henry Baker's garbage collection page.
ftp://ftp.netcom.com/pub/hb/hbaker/home.html
Valid on 2002/04/12.

[5] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988. Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

[6] Tim Berners-Lee. *Weaving the Web*. Harper, San Francisco, 1999.

[7] Sanjay Bhudia. Implementing a cyclic distributed garbage collector for a heterogeneous system with space failures.
http://www.doc.ic.ac.uk/ ajf/Teaching/Projects/ Distinguished99/SanjayBhudia.pdf
Valid on 2002/07/19.

[8] Steve Blackburn, Rick Hudson, Ron Morrison, J. Eliot B. Moss, David Munro, and John Zigman. Starting with termination: A methodology for building distributed garbage collection algorithms. In *24th Australasian Computer Science Conference (ACSC 2001)*, 2001.

[9] Hans-Juergen Boehm. GCBench.
http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench
Valid on 2002/07/19.

[10] Hans-Juergen Boehm. Mark-and-sweep vs. copying collection and asymptotic complexity.
http://www.hpl.hp.com/personal/Hans_Boehm/gc/complexity.html
Valid on 2002/07/19.

[11] Hans-Juergen Boehm. Reducing garbage collector cache misses. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, 2000.

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

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

[14] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[15] Jerzy Brzezinski, Jean-Michel Helary, and Michel Raynal. Distributed termination detection : General model and algorithms. Technical Report RR-1964, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, 1993.

[16] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, 1981.

[17] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[18] Michelle Cartwright. Empirical perspectives on maintaining web systems: A short review. In *6th IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2000)*, 2000.

[19] Yannis Chicha. Aldor documentation generation tool.
http://www.aldor.org/projects/aldordoc
Valid on 2002/07/19.

[20] Trishul Chilimbi, Richard E. Jones, and Benjamin Zorn. Designing a trace format for heap allocation events. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, 2000.

[21] William D. Clinger. Source code for selected GC benchmarks .
http://www.ccs.neu.edu/home/will/GC/sourcecode.html
Valid on 2002/07/19.

[22] World Wide Web Consortium. Extensible markup language (XML).
http://www.w3.org/XML
Valid on 2002/07/19.

[23] World Wide Web Consortium. Hypertext markup language.
http://www.w3.org/MarkUp
Valid on 2002/07/19.

[24] World Wide Web Consortium. Xhtml 1.0: The extensible hypertext markup language.
http://www.w3.org/TR/xhtml1
Valid on 2002/07/19.

[25] World Wide Web Consortium. Xml linking language (xlink) version 1.0.
http://www.w3.org/TR/xlink
Valid on 2002/07/19.

[26] Intel Corporation. Itanium architecture.
http://www.intel.com/ebusiness/pdf/prod/itanium/ds010401.pdf
Valid on 2002/07/19.

[27] P. Danzig, D. DeLucia, and K. Obraczka. Massively replicating services in autonomously managed wide-area internetworks. Technical Report 93-541, Computer-Science Department, University of Southern California, 1994.

[28] Edsgar 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):965–975, November 1978.

[29] Bruce Eckel. Thinking in Java.
http://www.mindview.net/Books/TIJ
Valid on 2002/07/19.

[30] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[31] The Apache Software Foundation. The Apache software foundation.
http://www.apache.org
Valid on 2002/07/19.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[33] Google, Inc. Google.
http://www.google.com
Valid on 2002/07/19.

[34] MIT Programming Methodology Group. MIT Thor system.
http://www.pmg.lcs.mit.edu/Thor.html
Valid on 2002/07/19.

[35] Network Working Group. HyperText Transfer Protocol – HTTP/1.1.
RFC n.2616.
http://www.ietf.org/rfc/rfc2616.txt
Valid on 2002/07/19.

[36] Dirk Grunwald and Benjamin Zorn. Malloc benchmarks.
ftp://ftp.cs.colorado.edu/pub/cs/misc/MallocStudy
Valid on 2002/07/19.

[37] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA'97 Proceedings*, 1997.

[38] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In *IWMM'92 Proceedings*, 1992.

[39] R. John M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1084, November 1982.

[40] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, March 1975.

[41] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

[42] Richard E. Jones. Richard Jones's garbage collection page.
http://www.cs.ukc.ac.uk/people/staff/rej/gc.html
Valid on 2002/07/19.

[43] Bálint Joó. Yet another Paraldor web page.
http://www.ph.ed.ac.uk/ bj/paraldor/WWW
Valid on 2002/07/19.

[44] Michael Kanellos. Intel offers details on future Itanium chips.
http://news.com.com/2100-1001-272320.html
Valid on 2002/07/19.

[45] F. Kappe. Maintaining link consistency in distributed Hyperwebs. In *Proceedings INET '95*, pages 15–24, 1995.

[46] Martijn Koster. The web robots pages.
http://www.robotstxt.org/wc/robots.html
Valid on 2002/07/19.

[47] UWO Symbolic Computation Lab. Aldor documentation.
http://www.aldor.org/documentation.html
Valid on 2002/07/19.

[48] UWO Symbolic Computation Lab. Aldor homepage.
http://www.aldor.org
Valid on 2002/07/19.

[49] UWO Symbolic Computation Lab. Aldor user guide.
http://www.aldor.org/AldorUserGuide
Valid on 2002/07/19.

[50] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[51] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *POPL'92 Proceedings*, pages 39–50, 1992.

[52] Fabrice LeFessant, Ian Piumarta, and Marc Shapiro. An implementation for complete asynchronous distributed garbage collection. In *PLDI'98 Proceedings*, 1998.

[53] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[54] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, 1995. Later appeared in Distributed Computing, Springer Verlag, 1996.

[55] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[56] Ryan L. McFall and Matt W. Mutka. Automatically finding and repairing broken links between XML documents. Technical Report MSU-CPS-98-38, Department of Computer Science, Michigan State University, East Lansing, Michigan, December 1998.

[57] Sun Microsystems. Javadoc tool home page.
http://java.sun.com/j2se/javadoc
Valid on 2002/07/19.

[58] Sun Microsystems. The Java(tm) programming language.
http://java.sun.com
Valid on 2002/07/19.

[59] Sun Microsystems. Java(tm) remote method invocation.
http://java.sun.com/products/jdk/rmi
Valid on 2002/07/19.

[60] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[61] MMnet. The UK memory management network.
http://www.mm-net.org.uk
Valid on 2002/07/19.

[62] David A. Moon. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, 1984.

[63] Luc Moreau. Hierarchical distributed reference counting. In *Proceedings of International Workshop on Memory Management*, pages 57–67, 1998.

[64] Luc Moreau and Nicholas Gray. A Community of Agents Maintaining Links in the World Wide Web (Preliminary Report). In *The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents*, pages 221–235, London, UK, March 1998.

[65] J. Gregory Morrisett, Mattias Felleisen, and Robert Harper. Abstract models of memory management. In *Record of the 1995 Conference on Functional Programming and Computer Architecture*, 1995.

[66] J. Gregory Morrisett, Mattias Felleisen, and Robert Harper. Abstract models of memory management. Technical Report CMU–CS–95–110, Carnegie Mellon University, January 1995. Also published as Fox memorandum CMU–CS–FOX–95–01.

[67] Cynthia J. Morton. L.O.S.T. in cyberspace.
http://gnacademy.tzo.org/ lost
Valid on 2002/07/19.

[68] Basem A. Nayfeh and Kunle Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA-21)*, 1994.

[69] Numerical Algorithms Group, Inc. The FRISCO project.
http://www.nag.co.uk/projects/FRISCO.html
Valid on 2002/07/19.

[70] Object Management Group, Inc. OMG's CORBA website.
http://www.corba.org/
Valid on 2002/07/19.

[71] The Mozilla Organization. mozilla.org.
http://www.mozilla.org
Valid on 2002/07/19.

[72] Michael Philippsen. Cooperating distributed garbage collectors for clusters and beyond. *Concurrency: Practice and Experience*, 12(7):595–610, May 2000. Also published in 8th Int. Workshop on Compilers for Parallel Computers CPC'2000, Aussois, France.

[73] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE'91 Parallel Architectures and Languages Europe*, 1991.

[74] José M. Piquer. Indirect mark and sweep: A distributed GC. In *IWMM'95 Proceedings*, 1995.

[75] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, ILOG, Gentilly, France, and INRIA, Le Chesnay, France, 1995.

[76] T. Printezis and R.E. Jones. GCspy.
http://www.dcs.gla.ac.uk/ tony/gcspy.www
Valid on 2002/07/19.

[77] The PerDis project. Perdis: Persistent distributed store.
http://www-sor.inria.fr/projects/perdis
Valid on 2002/07/19.

[78] Ravenbrook. The memory management reference.
http://www.memorymanagement.org
Valid on 2002/07/19.

[79] Nicolas Richer and Marc Shapiro. The memory behavior of the WWW, or: The WWW considered as a persistent store. In Graham Kirby, editor, *Int. W. on Persistent Obj. Sys.*, volume 2135 of *Lecture Notes in Computer*

Science, pages 169–184, Lillehammer (Norway), September 2000. Springer-Verlag. *http://www-sor.inria.fr/publi/TMBotWoTWCaaPS_pos2000.html* **Valid on 2002/07/19**.

[80] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[81] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, INRIA, November 1992. Also available as Broadcast Technical Report 1.

[82] Marc Shapiro, David Plainfossé, Paulo Ferreira, and Laurent Amsaleg. Some key issues in the design of distributed garbage collection and references. In *Unifying Theory and Practice in Distributed Systems*, Dagstuhl (Germany), September 1994.

[83] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[84] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[85] Darko Stefanović, J. Eliot B. Moss, and Kathryn S. McKinley. Age-based garbage collection. Technical report, University of Massachussets, April 1999. preliminary version of a paper to appear in OOPSLA'99.

[86] Robert Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, Sept 1973.

[87] Kenjiro Taura and Akinori Yonezawa. An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 264–275, 1997.

[88] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[89] Christian Ungureanu and Benjamin Goldberg. Formal models of distributed memory management. In *ICFP'97 Proceedings*, pages 280–291, 1997.

[90] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87 Parallel Architectures and Languages Europe*, pages 432–443, 1987.

[91] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, University of Texas, USA, 1992.

[92] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[93] Yahoo!, Inc. Yahoo!
http://www.yahoo.com
Valid on 2002/07/19.

[94] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, 1990.

Appendices

The following appendices describe uniprocessor and distributed garbage collectors according to the models presented in Chapter 5. We also illustrate our design method with mapping models between uniprocessor and distributed garbage collectors. In some cases, we propose several solutions for a single scenario (GC,DGC).

In this part of the report, Appendix B lists models for many well-known uniprocessor techniques, Appendix C describes four distributed garbage collectors, and Appendix D concludes with various scenarios.

Appendix A

Glossary

In this appendix, we explain the terminology and acronyms used in this thesis. Terms are presented in alphabetical order.

- **Allocation**: action to reserve memory space in the heap. It is usually performed by the **mutator**.
- **Collector**: the garbage collection part of the process.
- **Deallocation**: action to declare previously reserved memory space as "unused" and ready for a new allocation. In some cases, this memory space can be returned to the operating system.
- **Explicit memory management**: Technique of memory management which relies on programmers to take appropriate actions from their own initiative. In particular, a deallocation (e.g. `free`) primitive is provided.
- **Free list**: list of memory blocks available for allocation. This is an internal free list managed by the process. It is **not** managed by the OS.
- **Garbage object**: object that is not reachable from the **roots** and, thus, can be **deallocated**.

- **Heap**: area of memory used by the program to dynamically allocate objects. This space is where live and garbage objects are found. Objects in this area are accessed by direct addressing (pointers).
- **Lamport clock**: Well-known algorithm for synchronizing and ordering events in a distributed system. See [50] for details.
- **Live object**: object that can still be used by the process. It is reachable from the **roots**.
- **Memory management**: Activity which defines how to handle the creation and destruction of objects manipulated by the program.
- **Mutator**: part of the process which does actual work, as opposed to memory management.
- **Reachability**: an object is reachable if there exists at least one path of pointers from the roots to this object (see Section 2.3.2 for a detailed explanation).
- **Read barrier**: extra code added to the primitives used to read the values of objects.
- **Registers**: Processor registers. These elements are very close to the processor itself to allow extremely fast access but are not part of normal memory. Values that are frequently used are usually placed in the registers.
- **Roots**: Set of pointers found in the stack, the registers and the static area. In a distributed setting (see Section 2.5), it also includes entry and exit items (see Section 2.5 for a definition of this terminology). Roots are known to refer to live objects. They are the starting point for the collection process.

- **Scalability**: Property of distributed systems which defines the ability to increase the size of the network (in the number of nodes) while still providing acceptable features and performance.
- **Stack**: special memory area used to handle function calls. Local variables and returned values are placed in this area in a LIFO manner.
- **Static area**: the memory area used to allocate global variables prior to program execution.
- **Weak pointer**: Special type of pointer that does not prevent the garbage collector from reclaiming an object.
- **Write barrier**: extra code added to the primitives used to modify the values of objects.
- **XLink**: An XLink is an extended version of a URL, designed to be used in XML documents for sophisticated linking (e.g. use of attributes, possibility to reference any part of a document, and so on). See [25] for more details.

Appendix B

Models for common Memory Management techniques

This appendix describes models for common memory management techniques in a uniprocessor environment. These models have been created by following the templates described in Section 5.2.1.

B.1 Explicit Memory Management

We do not detail any high-level model for this technique. The primitives are functions to allocate and deallocate memory spaces. In C, we use the well-known **malloc** and **free**. At our level, we focus on how generic GCs can be mapped to the different techniques. For explicit management, we believe that the only solution is a special layer representing the generic GC. This is why we do not provide any model.

B.2 Reference Counting

Basics

- **ID:** Reference Counting.
- **Algorithm class:** *Reference Counting.*
- **Focus class:** *Object-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** A counter of references is associated with each object. When a reference to the object is created the counter is incremented, when a reference disappear, it is decremented. If the counter reaches zero, the object is reclaimed. This technique does not handle garbage cycles.

Data structures

1. - **ID:** ObjectHeader
   - **Description:** Information associated with each object. It is a counter.
   - **Contents:** `{ int counter }`

Data

1. - **ID:** POPULAR
   - **Description:** value used to flag a popular object. When an object is popular, its counter never increases and never decreases. This object becomes eternal.
   - **Contents:** `MAX_INTEGER`

Algorithms

1. - *ID:* DecrCnt
   - *Description:* Decrement object reference counter, possibly triggering more decrements.
   - *Input:* object o, *Output:* none, *Side-effects:* counter for o is decremented and, it reaches zero, the object is reclaimed. In this case, DecrCnt is called recursively on children objects.
   - *Data and Data Structures IDs:* ObjectHeader, POPULAR
   - *Contents:*

```
DecrCnt o
   if (o.ObjectHeader.counter == POPULAR) return
   o.ObjectHeader.counter--
   if (o.ObjectHeader.counter == 0)
     for each reference r in o
        DecrCnt r
```

2. - *ID:* IncrCnt
   - *Description:* Increment object reference counter.
   - *Input:* object o, *Output:* none, *Side-effects:* counter for o is incremented.
   - *Data and Data Structures IDs:* ObjectHeader, POPULAR
   - *Contents:*

```
IncrCnt o
   if (o.ObjectHeader.counter == POPULAR) return
   o.ObjectHeader.counter++
   if (o.ObjectHeader.counter == MAX_INTEGER)
      o.ObjectHeader.counter = POPULAR
```

B.3 Mark-and-Sweep

Basics

- **ID:** Mark & Sweep.
- **Algorithm class:** *Tracing non-copying.*
- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** The first phase is to mark all objects reachable from the roots (stack, registers, global variables). The second phase is to "sweep" (i.e remove) all non marked objects. It is important to note that this M&S algorithm uses two kinds of objects: fixed-size and mixed-size. Fixed-size objects are small and gathered contiguously on pages. Mixed-size objects are bigger objects and can span several pages.

Data structures

1. - **ID:** Page
   - **Description:** A page can be of any size. It is usually recommended to use a size corresponding to the system page size.
   - **Contents:** `{ PAGETAG tag }`
2. - **ID:** FixedArea
   - **Description:** Array listing the different available sizes of fixed-size objects. This is mainly useful for allocation.
   - **Contents:** `Array[#FixedSizes] of { FixedPage firstPage }`

3. • **ID:** FixedPage
   • **Description:** page of fixed-size objects. The number of objects stored depends on the size of one object. Everything (data structure + objects) has to fit on one page.
   • **Contents:**

```
{ PAGETAG tag = FIXED, // can't be anything else
  size_t size,         // size of one object
  FixedPage nextPage,  // next page of objects of this size
  Bitmap bits,         // bits for marking
  Pointer freeList,    // local free list
  Pointer objects      // Area where objects are stored
}
```

4. • **ID:** PageMap
   • **Description:** to keep track of large objects (which can span several pages)
   • **Contents:**

```
{ PAGETAG tag = PAGEMAP, // this is the page map!
  PAGETAG tags[]         // the map itself
}
```

5. • **ID:** MixedObject
   • **Description:** information for a mixed object
   • **Contents:**

```
{ Pointer begin,   // nil if the beginning is not on the same page
  Pointer end,     // nil if the end is not on the same page
  Byte mark        // for the GC mark phase
}
```

6. • **ID:** MixedObjectArea
   • **Description:** header for a mixed objects page
   • **Contents:**

```
{ PAGETAG tag = MIXED, // can't be anything else
  MixedObject info[]   // table of mixed objects information
}
```

Data

1. • **ID:** PAGETAG
   • **Description:** type of a page
   • **Contents:** `{ FREE, MIXED, FIXED, PAGEMAP, FOLLOW, FIRST }`

2. • **ID:** FixedSizes
   • **Description:** List of sizes for small objects
   • **Contents:**

```
{ sizeof(Pointer), 2*sizeof(Pointer), 4*sizeof(Pointer),
  8*sizeof(Pointer), 16*sizeof(Pointer), 32*sizeof(Pointer) }
// this can be changed to whatever is needed and it is possible
// to add more values (e.g 64*sizeof(Pointer))
```

Actions

1. • *ID:* MarkSweep
   • *Description:* Main algorithm.
   • *Input:* none, *Output:* none, *Side-effects:* garbage objects are removed from the heap.
   • *Data Structures IDs:* List (for the roots).

- *Contents:*

```
MarkSweep()
  r = getRoots()
    // assumed to return a list of ranges of addresses
  mark(r)
  sweep()
```

2.
- *ID:* mark
- *Description:* Scan the addresses in the given list and mark each encountered object. It is a conservative scheme in that there is no way to be 100% sure that a sequence of bytes is actually a pointer.
- *Input:* List of address ranges, *Output:* none, *Side-effects:* live objects are marked.
- *Data Structures IDs:* Page, FixedArea, FixedPage, MixedObject
- *Contents:*

```
mark(r: List (addr_beg, addr_end))
  for each range in r
     for (tmp := range.addr_beg; tmp < range.addr_end;
          tmp += sizeof(Pointer))
        ptr := *tmp
        if (not looksValid(ptr)) then continue
        page := getPageFromPtr(ptr)   // simple arithmetic
        pageno := getPageNumber(page)

        takeLastObj := false
        type := pagemap[pageno].tag
             // first (either page of fixed size objects
             // or first page of a large object) or follow?
        while (type = FOLLOW)  // get to the first page
             takeLastObj := true  // ptr is on a large object
                                  //  beginning  at the end of
                                  // the first page
             pageno := pageno - 1
             type := pagemap[pageno].tag
```

```
page := getPageFromNumber(pageno)
tag := page.tag

if (tag = FIXED) then
    obj_begin := (ptr - page.objects) / page.size
    if (not isMarked(obj_begin,page)) then
       markSmallObj(obj_begin, page)
         // simple arithmetic to set the right
         // bit in page.bitmap
       new_range := (obj_begin,obj_begin+page.size)
       mark(new List(new_range))
         // recursive call with only one range
         // in the list
else // tag = MIXED
    if (takeLastObj) then
      if (page.info[LAST].mark = false) then
        page.info[LAST].mark := true
        ptr_end := getEndObject(page,
                                page.info[LAST].begin);
           // follow the page until the end of
           // the object is found
        new_range := (page.info[LAST].begin,ptr_end);
        mark(new List(new_range))
           // recursive call with only one range
           // in the list
    else // takeLastObj = false
      index := getBeginObject(page, ptr)
           // object contained within the page
      if (page.info[index].mark = false) then
        page.info[index].mark := true
        new_range := (page.info[index].begin,
                      page.info[index].end)
        mark(new List(new_range))
           // recursive call with only one range
           // in the list
```

3.
   - *ID:* sweep
   - *Description:* reclaim all non marked objects
   - *Input:* none, *Output:* none, *Side-effects:* non-marked objects are reclaimed.
   - *Data Structures IDs:* MixedObject, FixedSize

- *Contents:*

```
sweep()
  step_pages := 1
  for (p := first_page; p <= last_page; p := p + step_pages)
     if (p.tag = FIXED) then
        atLeastOneMarked := false;
        for each obj in p.objects
           if (isMarked(obj,p)) then
              unsetMark(obj,p) // uses p.bits
              atLeastOneMarked := true
           else
              insert(obj,p.freeList)
        if (not atLeastOneMarked) then
           unlink from FixedPage set
           reclaim(p) // i.e add p to global free list
     else // p.tag = MIXED
        pageno := getPageNumber(p)
```

B.4 Localised Mark-and-Sweep

Basics

- **ID:** Localized Mark&Sweep.
- **Algorithm class:** *Tracing non-copying.* Inherit from Mark-and-Sweep.
- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** This is a variant of a Mark&Sweep algorithm. It tries to improve marking performance by improving caching and paging behavior. The heap is divided into regions and each region is completely marked before marking another one. Mark queues are used to handle "out-of-the-region" pointers.

  Important: the purpose of the algorithm is to improve caching and paging

behavior by deciding which objects should be seen in what order. This is what makes this algorithm special. Any adaptation has to respect the organization in regions and the order of visit of the heap objects.
Note: in this model we describe only specific elements that are different from the traditional Mark&Sweep model.

The elements detailed here are used to present the GC rather than give a complete in-depth description (this is done below with data structures, data, and actions). This is intended to be a sort of "identity card". We see here that LMS is a tracing non-copying, heap-focused, uniprocessor GC. The statement *Inherit from Mark-and-Sweep* brings a precision about the algorithm class of the collector. We also observe that the description can be used to provide an important clue about the usage of this GC: its "essence" relies on a specific heap organization that should be respected by all adaptations.

Data structures

1.
   - **ID:** TraceQueue
   - **Description:** Stores pointers to the corresponding region. This acts like a DGC environment entry item.
   - **Contents:**

```
Pointer queue[MAX_SIZE_QUEUE]
int head
int count
int region
```

   - **Operations:**
     - `enqueue`. Input: Pointer, Output: none, Effect: new element in the queue or, if the queue is full, context switch to a new region.

- `dequeue`. Input: none, Output: Pointer, Effect: one element removed from the queue.
- `isEmpty`. Input: none, Output: Boolean, Effect: none.
- `isFull`. Input: none, Output: Boolean, Effect: none.
- `whatRegion`. Input: none, Output: region the queue is linked to, Effect: none.

The `TraceQueue` data structure is the first component we detail here. LMS is based on a regional organization of the heap. Trace queues are the basis of the collector as they enable the delay of object marking. We give here all the details about them. Other data structures such as the header of objects, memory maps and so on, should be shown as well for a complete description of the GC. However, we show only TraceQueue as it is what makes the scheme different from Mark-and-Sweep, we can use an inheritance mechanism where LMS inherits all from MS and describe only what is different. This "OO-ish" organization would allow for simple reuse of models (using UML, for example).

Data

1. - **ID:** queues
   - **Description:** All queues gathered in a contiguous area. This organization is not needed (could be a list), but is likely to be more efficient.
   - **Contents:** `TraceQueue queues[Nb_Regions]`
2. - **ID:** `MAX_QUEUE_SIZE`
   - **Description:** Queues have a limited size. This is used for the size of the array representing a trace queue at low-level. `MAX_QUEUE_SIZE` can change overtime if we include a mechanism to allow dynamic evolution of trace queues.

- **Contents:** `MAX_QUEUE_SIZE == a value`

We note that one variable and one constant are shown here. As we can see the "Contents" parts are quite small, because these are usually simple entities. We attach more importance on the description which explains the purpose of each piece of data.

Actions

1. - *ID:* rootsDispatch
   - *Description:* Initial work. Take all roots and place them into the proper queue.
   - *Input:* stack, static area, registers, *Output:* none, *Side effect:* `queues` is filled with the roots.
   - *Data Structures IDs:* TraceQueue
   - *Contents:*

```
rootsDispatch(stack, static area, registers)
  roots := all pointers from stack, static area and registers.
  for each r in roots
     reg := regionOf(r)
     if (isFull(queues[reg]))
       dealWithQ(queues[reg])
       scanMark(r,reg)
     else
       enqueue(r,queues[reg])
```

2. - *ID:* dealWithQ
   - *Description:* Visit all references in a given queue.
   - *Input:* a trace queue q, *Output:* none, *Side effects:* recursively mark and visit all objects pointed to by pointers in q.
   - *Data and Data Structures IDs:* TraceQueue.

- *Algorithm:*

```
dealWithQ(q)
    while (not q.isEmpty())
        p = q.dequeue();
        if (not marked(*p)) scanMark(p,q.whatRegion());
```

3. - *ID:* scanMark
   - *Description:* Mark an object and scan it for more pointers.
   - *Input:* Pointer p and a region *currentRegion*, *Output:* none, *Side effects:* recursively mark objects or enqueue them if not in region *currentRegion*.
   - *Data and Data Structures IDs:* TraceQueue, Pointer, queues
   - *Algorithm:*

```
scanMark(Pointer p, int currentRegion)
   if (p is not valid) return
   if (marked(*p)) return
   mark *p
   for each p' in *p
      reg := regionOf(p')
      if (reg == currentRegion)
         scanMark(p',reg)
      else
        if (isFull(queues[reg]))
           dealWithQ(queues[reg])
           scanMark(p',reg)
        else
           enqueue(p',queues[reg])
```

4. - *ID:* LMS
   - *Description:* Main GC function.
   - *Input:* none, *Output:* none, *Side effects:* Enqueue roots. Mark objects. Sweep garbage thus filling free list.

- *Data and Data Structures IDs:* TraceQueue, queues, free list.
- *Algorithm:*

```
LMS()
   rootsDispatch()
   while (queues are not empty)
      for each q in queues
         dealWithQ q

   Sweep() // regular sweep phase
```

We can see in this example that the main function (called LMS) calls other functions such as dealWithQ also described in this model. The syntax of the algorithm description is left open. When the template is used, it might be necessary to decide about some rules, but it can be based on conventions already used by the group of designers.

The tracing part of the algorithm is very useful to describe (even when it is not the main feature of the GC). Most of today's DGCs rely on local propagation of information and understanding the tracing model of a stand-alone DGC will help creating an appropriate local GC.

B.5 Mark-and-Copy

Basics

- **ID:** Mark-and-Copy.
- **Algorithm class:** *Tracing copying.*
- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*

- **Description:** Starting from the roots, the algorithm follows all possible paths of pointers. Each object found is copied to another part of the memory. The algorithm modeled here uses semi-space organization. The 'From' space contains all the objects. The 'To' space is where they are copied. A forwarding mechanism is used to take care of objects referenced by several objects.

Data structures

1. - **ID:** ObjectHeader
   - **Description:** Information about each object.
   - **Contents:** `{ boolean forwarded, size_t size }`

Data

1. - **ID:** indexTo
   - **Description:** pointer for free space in the 'To' area
   - **Contents:** `Pointer indexTo`

Actions

1. - *ID:* CopyObject
   - *Description:* copy one object over to the 'To' part
   - *Input:* pointer p to an object, *Output:* none, *Side-effects:* o is moved to the `To` space.
   - *Data & Data Structures IDs:* ObjectHeader, indexTo

- *Contents:*

```
CopyObject p
    newaddr := indexTo
    copy(p, newaddr, header(p).size)
    header(p).forwarded := true
    *p := newaddr // saved in the first bytes of o
    indexTo := indexTo + size
```

2. - *ID:* MarkCopy
   - *Description:* most important function. Trace all objects and copy them over to the 'To' part.
   - *Input:* (low address, high address). This is the range of addresses to examine, *Output:* none, *Side-effects:* live objects are moved to the 'To' area.
   - *Data & Data Structures IDs:* ObjectHeader
   - *Contents:*

```
MarkCopy(a1,a2)
  For each valid pointer p in [a1,a2]
     if (header(*p).forwarded)
        update(p,*p)
        // here, *p contains the new address of the object
     else
        CopyObject(p)
        MarkCopy(p,p+header(*p).size)
```

3. - *ID:* MarkCopyGC
   - *Description:* main function
   - *Input:* none, *Output:* none, *Side-effects:* live objects are moved to the 'To' area. garbage objects are left in the 'From' area. Roles of the area are reversed.
   - *Data & Data Structures IDs:* indexTo.

- *Contents:*

```
MarkCopyGC()
   [r1,r2] := getRoots()
   indexTo := pointer to the beginning of the 'To' area.
   MarkCopy(r1,r2)
   reverseToFrom()
```

B.6 Generational GC

Basics

- **ID:** Generational Copying GC.
- **Algorithm class:** *Generational.*
- **Focus class:** *Region-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** This collector relies on the observation that objects tend to die young. Therefore, the heap is divided into a young and an old generation. Objects are allocated in the young generation which is collected very often. The old one is rarely collected. Because the young generation is small, collection pauses will be short. We describe the particular case of two generations: a young one and an old one.

Data structures

1. - **ID:** RemSet
   - **Description:** Remembered set. List of referents to objects in a given generation.
   - **Contents:** `List(RemSetEntry) entries`

- **Operations:**
  - `isEmpty`. Input: none, Output: boolean (entries.isEmpty()), Effect: none.

2. - **ID:** RemSetEntry
   - **Description:** This structure contains a reference to an object, and the address of its referent.
   - **Contents:** `{ Pointer obj, Pointer origObj }`

3. - **ID:** Generation
   - **Description:** Area of memory holding objects of a given age.
   - **Contents:** `{ int rank, int age, Pointer begin, Pointer end }`
   - **Operations:**
     - `fillRatio`. Input: Generation g, Output: Fill ratio for g, Effect: none.

Data

1. - **ID:** YGRemSet
   - **Description:** Young Generation Remembered Set.
   - **Contents:** `RemSet YGRemSet`

2. - **ID:** youngGen
   - **Description:** Young generation
   - **Contents:** `Generation youngGen`

3. - **ID:** oldGen
   - **Description:** Old generation
   - **Contents:** `Generation oldGen`

4. • **ID:** FILL_THRESHOLD
   • **Description:** Value that determines if a collection is needed.
   • **Contents:** `FILL_THRESHOLD == a value`

Actions

1. • *ID:* collectYoung
   • *Description:* Collection of the young generation.
   • *Input:* none, *Output:* none, *Side-effects:* young garbage is reclaimed.
   • *Data & Data Structures IDs:* Generation, youngGen, YGRemSet.
   • *Contents:*

```
collectYoung()
  for each r in root
    if ((youngGen.begin <= r) and (r < youngGen.end))
      scan_and_promote(r) // Mark&Sweep or Mark&Copy
  for each p in YGRemSet
    if ((youngGen.begin <= p) and (p < youngGen.end))
      scan_and_promote(p) // Mark&Sweep or Mark&Copy
```

2. • *ID:* collectOld
   • *Description:* Collection of the old generation. It is actually a collection of the whole heap.
   • *Input:* none, *Output:* none, *Side-effects:* all garbage is reclaimed. `youngGen` becomes automatically empty.
   • *Data & Data Structures IDs:* YGRemSet.
   • *Contents:*

```
collectOld()
   call normal collector (Mark-and-Sweep or Mark-and-Copy)
   promote all live young objects
   empty YGRemSet
```

3. - *ID:* GenGC
   - *Description:* Main garbage collection function. It calls either `collectYoung` or `collectOld`. It is called when memory in the young generation is running out. We add a verification to call the old collection if necessary.
   - *Input:* none, *Output:* none, *Side-effects:* part of all garbage is reclaimed.
   - *Data & Data Structures IDs:* none.
   - *Contents:*

```
GenGC()
   if (fillRatio(oldGen) > FILL_THRESHOLD)
     collectOld()
   else
     collectYoung()
```

B.7 Mature Object Space

Basics

- **ID:** Mature Object Space (MOS).
- **Algorithm class:** *Age-based and Incremental.*
- **Focus class:** *Region-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** Old generations in a generational scheme are usually big. Collecting them leads to a disruptive process. MOS proposes a way to place a bound on collection time by limiting the size of space treated at each collection of the old generation. It will no longer be collected at once.

Instead, small portions will be collected, thus providing an incremental scheme.

Data structures

1.
   - **ID:** Car
   - **Description:** Basic block holding objects in the old generation.
   - **Contents:**

```
{
  int rank, Pointer begin,
  RemSet remset, Train owner,
  Car next, Car prev
}
```

2.
   - **ID:** Train
   - **Description:** Gather cars to reclaim cycles, created to hold related data structures.
   - **Contents:** `{ List(Car) cars, RemSet remset }`
   - **Operations:**
     - * *ID:* updateRemSet
       * *Input:* none, *Output:* none, *Side-effects:* the field `remset` is computed from the remsets of all the train's cars.
     - * *ID:* reclaimTrain
       * *Input:* a train t, *Output:* none, *Side-effects:* Reclamation of a train as a whole.

3.
   - **ID:** RemSet
   - **Description:** Remembered set. List of referents to objects of this car.
   - **Contents:** `List(RemSetEntry) entries`

- **Operations:**
  - *ID:* isEmpty
  - *Input:* none, *Output:* boolean (entries.isEmpty()), *Side-effects:* none.

4. - **ID:** RemSetEntry
   - **Description:** This structure contains a reference to an object, the address of its referent, and the address of the car of the referent.
   - **Contents:** `{ Pointer obj, Pointer origObj, Pointer origCar }`

5. - **ID:** Generation
   - **Description:** Area of memory holding objects of a given age.
   - **Contents:** `{ int rank, int age, Pointer begin, Pointer end }`

Data

1. - **ID:** nextCar
   - **Description:** Designate the next car to deal with.
   - **Contents:** `Car nextCar`

2. - **ID:** SIZECAR
   - **Description:** Size of a car, each car has the same size.
   - **Contents:** `SIZECAR == a value`

3. - **ID:** YGRemSet
   - **Description:** Young Generation Remembered Set.
   - **Contents:** `RemSet YGRemSet`

4. - **ID:** youngGen
   - **Description:** Young generation
   - **Contents:** `Generation youngGen`

Actions

1.
   - *ID:* collectCar
   - *Description:* Collection of a car. This function is called each time a collection of the old generation is needed.
   - *Input:* none, *Output:* none, *Side-effects:* All live objects are moved from the car to others. The chosen car is reclaimed at the end. The train owning the car may be reclaimed too.
   - *Data Structures IDs:* Car, Train.
   - *Contents:*

```
collectCar()
   // get car to collect. Cars are usually ordered from
   // the oldest to the youngest.
   c := nextCar
   nextCar := nextCar.next

   // Collect!
   if (checkTrainToReclaim(c.owner))
      reclaimTrain(c.owner)
   else
      collectFromRoots(c)
      collectFromOtherTrains(c)
      collectFromOtherCars(c)
      reclaimCar(c)
```

2.
   - *ID:* checkTrainToReclaim
   - *Description:* If the remset for the train is empty, there is no external pointer to objects in this train. This means these objects are not reachable, the train can thus be safely reclaimed.
   - *Input:* a train t, *Output:* boolean, *Side-effects:* none.
   - *Data Structures IDs:* Train, RemSet.

- *Contents:*

```
checkTrainToReclaim(Train t)
  return t.remset.isEmpty();
```

3. - *ID:* collectYoung
   - *Description:* Collection of the young generation. With MOS, there is only two generations.
   - *Input:* none, *Output:* none, *Side-effects:* young garbage is reclaimed.
   - *Data & Data Structures IDs:* Generation, youngGen, YGRemSet.
   - *Contents:*

```
collectYoung()
   for each r in root
      if ((youngGen.begin <= r) and (r < youngGen.end))
            scan_and_promote(r)
   for each p in YGRemSet
      if ((youngGen.begin <= p) and (p < youngGen.end))
            scan_and_promote(p)
```

4. - *ID:* collectFromRoots
   - *Description:* Scan roots pointing to a given car and move objects to a younger train. The following operations are assumed: `rootsToCar`, which gives the list of roots pointer to a car, `moveToTrain`, which moves an object to a given train, and `getYoungerTrain`, which finds a younger train.
   - *Input:* a car c, *Output:* none, *Side-effects:* Root-reachable objects are moved to another train.
   - *Data Structures IDs:* Car, Train.

- *Contents:*

```
collectFromRoots(c)
   for each r in rootsToCar(c)
      moveToTrain(r, getYoungerTrain(c.owner))
```

5. - *ID:* collectFromOtherTrains
   - *Description:* Find references coming to a given car from trains others than the car's owner. Move objects accordingly. The following functions are assumed: `TrainsToCar`, which obtains, from the remset associated to a given car c, all pointers from other trains to an object in c, and `moveToTrain`, which moves an object to a given train.
   - *Input:* a car c, *Output:* none, *Side-effects:* Objects reachable from other trains are moved to those trains.
   - *Data Structures IDs:* Train, Car.
   - *Contents:*

```
collectFromOtherTrains(c)
   for each (p,t) in TrainsToCar(c)
      moveToTrain(p,t)
```

6. - *ID:* collectFromOtherCars
   - *Description:* Find references coming to a given car from trains others than the car's owner. Move objects accordingly. The following functions are assumed: `CarsToCar`, which obtains, from the remset associated to a given car c, all pointers from other cars to an object in c, and `moveToCar`, which moves an object to an arbitrary car in the same train.
   - *Input:* a car c, *Output:* none, *Side-effects:* Objects reachable from other cars in the same trains are moved to any car in the train.

- *Data Structures IDs:* Car.
- *Contents:*

```
collectFromOtherCars(c)
   for each p in CarsToCar(c)
      moveToCar(p)
```

Appendix C

Models for common Distributed Garbage Collectors

C.1 Distributed Reference Counting

Basics

- **ID:** DRC.
- **Algorithm class:** Distributed Reference Counting.
- **Inherits from:** none.
- **Description:** This is a standard, simple distributed reference counting algorithm. There is no optimization.
- **Reference paper:** none.
- **Assumptions:**

  1. – *Type:* Opaque addressing.
     – *Comments:* this implies that the mutator takes care of incrementing counters on entry items when an object is exported.

- **Conditions of use:** This algorithm can be used in all environments.
- **Addressed issues:** *Distributed acyclic garbage, safety.*
- **Supplies:**
  1. – **Entity:** Entry.
     – **Description:** Message to decrement opaque addressing items. Note that there is no increment message (more details in the main algorithm).
- **External Algorithms:** none.

Main algorithm

This algorithm is completely localized. Decrement messages allow to maintain counters on entry items. Identification and reclamation of garbage is the responsibility of each local collector. Counters are incremented when the mutator exports an object to a remote node. It is up to the local collector to implement the appropriate procedure to invoke when the counter of an entry item reaches zero.

Protocols

1. - **ID** DECREMENT
   - **Description** Message to decrement the counter of a specific entry item.
   - **Local actions needed** reclamation of an exit item.
   - **Local actions triggered** decrement of a counter.

- **Contents**
  - *Number of sites* 2
  - *Ordered list of messages:* `(A,B,DECREMENT,[entry_item])`
- **Set of sites** {A,B}

Generic GC for DRC

Basics

- **ID:** Generic GC for DRC
- **Algorithm class:** None specific.
- **Focus class:** None specific.
- **Concurrency class:** None specific.
- **Description:** This generic GC expresses very few requirements, which explains why DRC algorithms are widespread in practical environments.

Data structures

1.
   - **ID:** EntryItem
   - **Description:** Proxy to handle incoming pointers.
   - **Contents:** `{ int counter, Pointer object }`
2.
   - **ID:** ExitItem
   - **Description:** For outgoing pointers. It is a proxy for a remote object.
   - **Contents:** `{ RemoteNode node, Pointer remoteEntry }`

Actions

1.
   - *ID:* Increment
   - *Description:* Called by the mutator when exporting a reference to an object.
   - *Input:* Pointer p, *Output:* EntryItem, *Side-effects:* entry item e corresponding to p is found and its counter is incremented. If there is no such e, a new item is created with a counter of 1.
   - *Data Structures IDs:* EntryItem.
   - *Contents:*

```
Increment(Pointer p) {
    e := findEntryItem(p);
    if (e != nil) e.incrCounter();
    else e := createNewEntryItem(p);
    return e;
}
```

2.
   - *ID:* Decrement
   - *Description:* When an exit item is reclaimed on one node, the counter of its corresponding entry item on a remote node should be decremented. This function is called when the DECREMENT message is received. A local GC specific action is possibly taken if the counter reaches 0.
   - *Input:* EntryItem e, *Output:* none, *Side-effects:* e's counter is decremented.
   - *Data Structures IDs:* EntryItem.

- *Contents:*

```
Decrement(Entry item e) {
    e.decrCounter();
    if (e.counter = 0) local_gc_specific_action();
}
```

C.2 GCW

Basics

- **ID:** GCW (Garbage Collecting the World)
- **Algorithm class:** *Augmented DRC/DRL*
- **Inherits from:** DRC
- **Description:** GCW is a distributed reference counting algorithm using mark propagation to reclaim dead distributed cycles. It relies on a DTD algorithm to discover global stability. Although it is not treated in this model, GCW also proposes an organization of the nodes in groups. This allows for scalability and failure handling.
- **Reference paper:** *Garbage Collecting the World*, B.Lang, C.Queinnec, J.Piquer, POPL'92 Proceedings.
- **Assumptions:** none. A complete model of GCW should assume neat failures.
- **Conditions of use:** This technique works best with local tracing GCs.
- **Addressed issues:** *failure handling, distributed cycles*

- **Supplies**

  1. – **Entity:** Entry item decrement.
     – **Description:** message allowing to maintain the counter of pointers on an entry item.
  2. – **Entity:** Hard marking propagation.
     – **Description:** this acknowledges an entry item to be part of a root-reachable distributed chain of pointers.

- **External Algorithms**

  1. – **ID** DTD.
     – **Description** Any DTD algorithm is allowed.
     – **Purpose** Find out global stability according to local stability rule: after a local GC, no exit item has been marked HARDer than during previous GC.

Main algorithm

Basic algorithm is a DRC algorithm:

```
EraseRemotePointer => send decrement message
```

To reclaim garbage cycles:

Uses opaque addressing: entry items and exit items

Initialization: All entry items are marked SOFT

Local propagation: by each GC

Global propagation:

- HARD marks are sent as soon as possible

Global stability detection:

- rely on a DTD algorithm.
- once discovered, garbage cycles contain only SOFT items.
- Asynchronously, each node sets the counters of SOFT entry items to zero. This allows to break garbage distributed cycles.

Protocols

1. - **ID** Decrement entry item
   - **Description** When an exit item has to be reclaimed, a decrement message is sent to the corresponding remote node.
   - **Local actions needed** dead exit item detection.
   - **Local actions triggered** decrement the counter of an entry item. There is a possible reclamation of an entry item if the counter reaches zero (local reference counting collectors), otherwise reclamation is done at a later date when a local collection occurs.
   - **Contents**
     - *Number of sites* 2
     - *Ordered list of messages:* `(A,B,DECR,[entry addr])`
   - **Set of sites** { A = origin, B = destination }
2. - **ID** Harden entry item
   - **Description** An exit item has been marked HARD, the corresponding entry item needs to be informed. This is the global propagation step. It can be batched.
   - **Local actions needed** hard mark assigned to an exit item.
   - **Local actions triggered** entry item marked hard.

- **Contents**
  - *Number of sites* 2
  - *Ordered list of messages:* `(A,B,HARD,[entry addr])`
- **Set of sites** { A = origin, B = destination }

Generic GC for GCW

Basics

- **ID:** Generic GC for GCW
- **Algorithm class:** *Tracing.*
- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** The tracing characteristic of this GC is necessary for local mark propagation.

Data structures

1.
   - **ID:** EntryItem
   - **Description:** Proxy to handle incoming pointers. It inherits from DRC's entry item. We only show the new field: mark.
   - **Contents:** `EntryItem: DRC.EntryItem { Marks mark }`
2.
   - **ID:** ExitItem
   - **Description:** For outgoing pointers. It is a proxy for a remote object. It inherits from DRC's exit item.
   - **Contents:** `ExitItem: DRC.ExitItem { Marks mark, Marks oldmark }`

Data

1. - **ID:** Marks
   - **Description:** Used by items. NONE is used to initialize exit items before a local propagation.
   - **Contents:** `{ NONE, SOFT, HARD }`

Actions

1. - *ID:* LocalPropagation - Solution 1
   - *Description:* Assign a mark to exit items according to reachability from entry items and/or roots.
   - *Input:* none, *Output:* none, *Side-effects:* Marks on exit items are possibly changed.
   - *Data Structures IDs:* EntryItem, ExitItem, Roots
   - *Contents:*
     (a) $\forall$ ex $\in$ ExitItems, oldmark(ex) := mark(ex), mark(ex) := NONE
     (b) $\forall$ en $\in$ EntryItems s.t. mark(en) = SOFT, $\forall$ ex $\in$ ExitsReachableFrom(en), mark(ex) := SOFT
     (c) $\forall$ en $\in$ EntryItems s.t. mark(en) = HARD, $\forall$ ex $\in$ ExitsReachableFrom(en), mark(ex) := HARD
     (d) $\forall$ r $\in$ Roots, $\forall$ ex $\in$ ExitsReachableFrom(r), mark(ex) := HARD

2. - *ID:* LocalPropagation - Solution 2
   - *Description:* Assign a mark to exit items according to reachability from entry items and/or roots.
   - *Input:* none, *Output:* none, *Side-effects:* Marks on exit items are possibly changed.

- *Data Structures IDs:* EntryItem, ExitItem, Roots
- *Contents:*
  (a) $\forall$ ex $\in$ ExitItems, oldmark(ex) := mark(ex), mark(ex) := NONE
  (b) $\forall$ r $\in$ Roots, $\forall$ ex $\in$ ExitsReachableFrom(r),
  mark(ex) := HARD
  (c) $\forall$ en $\in$ EntryItems s.t. mark(en) = HARD,
  $\forall$ ex $\in$ ExitsReachableFrom(en), mark(ex) := HARD
  (d) $\forall$ en $\in$ EntryItems s.t. mark(en) = SOFT,
  $\forall$ ex $\in$ ExitsReachableFrom(en),
  if (mark(ex) = NONE) then mark(ex) := SOFT

3. - *ID:* Stability detection
   - *Description:* Detects local stability of a node to support termination detection. A node reaches stability when local propagation does not change the marks on exit items. To achieve this, old marks on exit items are necessary (this is an implementation of the properties *t1* and *t2* used below). **Global stability** is detecting via the DTD algorithm, which relies on local stability detection.
   - *Input:* exit items, *Output:* boolean (stable or not), *Side-effects:* none.
   - *Data Structures IDs:* EntryItem, ExitItem, Roots
   - *Contents:*
     - MarksExits(t1) = MarksExits(t2) *where t1 = time begin Propagation and t2 = time end Propagation*
     - **and** ($\exists$ no rcv_msg m, s.t. $m \in$ HARD_MSG
       $\wedge$ mark(entry(m)) $\neq$ HARD)
     - **and** ($\exists$ no sent_msg m', s.t. $m' \in$ HARD_MSG)

C.3 Migration-based

Basics

- **ID:** Controlled Migration.
- **Algorithm class:** *Augmented Distributed Reference Listing.*
- **Inherits from:** DRL.
- **Description:** This DGC has been created for databases and persistent systems, but can be adapted to other systems. The idea is to migrate all objects that are likely to be part of a distributed garbage cycle to a single node, where the local GC can reclaim it. Heuristics, based on estimated distances to a root object, are used to determine if an object is in this case.
- **Reference paper:** *Collecting Cyclic Distributed Garbage by Controlled Migration*, U.Maheshwari, B.Liskov, Proceedings of PODC'95 Principles of Distributed Computing.
- **Assumptions:**
  1. – *Type:* Underlying system requirement: no unnecessary migration.
     – *Comments:* Only garbage objects should be migrated. It is important to maintain good performance in some database systems.
- **Conditions of use:** Migration should be possible (no local conservative systems).
- **Addressed issues:** *scalability, distributed cycles, speed.*

- **Supplies**

  1. – **Entity:** Distance
     – **Description:** Value used to determine if a remotely reachable object is still reachable. If the distance is above a certain threshold, then the object becomes a candidate for migration. The final Destination is then estimated to reduce the number of migrations of the same object.
  2. – **Entity:** Destination
     – **Description:** Value used to determine where an object shall be migrated. The idea is to estimate the node where objects should be evacuated first to minimize the number of migrations of the same object. For example, if the node Z is the final destination node and the object o on node D has to be migrated, it would be more efficient to send o directly to Z rather than sending it to E, where other objects part of the cycle are located, then F then G, and finally Z.

- **External Algorithms:** none.

Notes: We can see two important assumptions. "No unnecessary migration" constitutes a basic requirement that we need to keep in mind while designing the system. It is part of the essence of the algorithm, and, without it, the algorithm does not have the same meaning. "Migration should be possible" is of a more practical nature. If there is no system support for migration, the DGC can not perform properly.

The "Supplies" fields described here are both essential notions used by this DGC algorithm. Distance and Destination allow the collector to discover garbage cycles and know where to migrate its components. We can see that information

supplied to the local GC is very important in this distributed collector. Designers of the local GC should be aware of these values.

Main algorithm

- **DRL** is the basic system used to reclaim non-cyclic garbage and provide some kind of fault-tolerance (but it is not detailed here). *inlists* correspond to entry items and *outlists* to exit items.
- **Local propagation**. Usually performed using the local GC. Distances are computed and propagated from roots and inlists to outlists. Destinations are also computed and propagated by this mechanism.
- **Global propagation**. After a local GC, outlists are sent to their corresponding nodes along with the distances and destinations.
- **Migration**. When the distance value of certain objects is reached, and the destination has been computed, objects are migrated.

Notes: The main algorithm is an overview of the global strategy. Its purpose is to explain the essence of the DGC. We find three elements here. First, the basic strategy is a Distributed Reference Listing, which means that we can use any DRL strategy that will complies with the rest of the algorithm. For example, we could use Moreau's HDRC [63] or Shapiro's SSPC [81]. The second element is the distance that is computed by propagation. The third one is the destination also computed by propagation of information. We need to keep these three important elements in mind to really understand this DGC.

Protocols

1.
   - **ID** OUTLIST
   - **Description** This is a message used to send an outlist to a node, which will use it to update its inlist.
   - **Local actions needed** local_propag
   - **Local actions triggered** receive_OUTLIST
   - **Contents**
     - *Number of sites* 2
     - *Ordered list of messages:* `(A,B,OUTLIST,[outlist])`
   - **Set of sites** A, B

Notes: This DGC features very simple protocols. Local GCs send their "outlist"s (which would correspond to sending exit items) to corresponding nodes. This message is essential to the whole process, because it informs remote nodes about remote reachability of their objects. We see that `local_propagation` (see Generic GC in Section C.3) is the action that will trigger this message and, of course, `receive_OUTLIST` will replace the old "inlist" by the received "outlist".

The message itself (see Contents above) is straightforward. No acknowledgment message is specified in the strategy although it could be implicit. This usually depends on the environment. If messages do not need acknowledgment, this means that the environment is safe. This allows assumptions about failures, thus helping to choose the DRL algorithm needed as a basis here.

Generic GC for Migration-based

Basics

- **ID:** Generic GC for Controlled Migration DGC.
- **Algorithm class:** *Tracing.*
- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** This GC essentially propagates and computes distances and destinations. It triggers migration of objects when necessary.

Notes: The Generic GC is using the model described for Stand-alone collectors. This DGC requires a tracing collector, which is reflected by the algorithm class of this generic collector. Beyond this model, more details about requirements for the local collector can be found in the literature indicated in the "Reference paper" section of the DGC model. Remember that this model does not intend to *teach* about a collector, but to organize its characteristics in a standard way for design and implementation.

Data structures

1. - **ID:** inlist
   - **Description:** List of references to public objects. This corresponds to entry items in other DGC schemes.
   - **Contents:**

```
{ Pointer object,
  List of struct {
     Node node, // where the pointer comes from
     Distance distance,
     Destination destination }
}
```

2. • **ID:** outlist
   • **Description:** Records pointers to remote objects. This is known as exit items in other DGC schemes.
   • **Contents:**

```
List of struct {
  Node node, Pointer object,
  Distance distance, Destination destination }
```

Note: There is nothing surprising here. We already mentioned that most DGCs require local proxies, this is the case here. `inlist`s hold "entry items" and `outlist`s hold "exit items". These data structures have to be added to the stand-alone GC in order to form the local GC.

Data

We first describe two important notions of this DGC algorithm: Distance and Destination. Although their types are not complex, they are listed here to describe their purpose in detail.

1. • **ID:** Distance
   • **Description:** The notion of distance is used to find out what objects are part of garbage cycles. The distance is based on the knowledge of the location of root objects. When an object is pointed to by a rooted object in the same node, the distance between them is 0, when this object is on a node A, the distance is 1. If a rooted object o_1 on node A points to o_2 on node B and o_2 points to o_3 on node C, the distance for o_3 is 2. Distances are propagated from one node to another, and the distance value assigned to an object is the smallest distance to that object. Garbage cycles will hold objects with potentially infinite distances as there will be no root to keep the value from increasing as this goes from one node to another.

- **Contents:** `typedef int Distance;`

2. - **ID:** Destination
   - **Description:** Destinations are evaluated according to heuristics about the cycle being detected. Information about the possible final destination for a suspected garbage cycle is propagated from node to node, using the destination files in inlist and outlist. When `Threshold2` (see below) is reached during propagation, it means that the final destination is likely to have been discovered. Migration can occur.
   - **Contents:** `typedef NodeID Destination;`

3. - **ID:** Threshold
   - **Description:** Chosen value that will serve as a threshold to decide whether or not objects are to be migrated because they might belong to a garbage cycle.
   - **Contents:** `Threshold = A_CHOSEN_THRESHOLD`

4. - **ID:** Threshold2
   - **Description:** Chosen value that will serve as a threshold to decide when the final estimated destination for the objects of this garbage cycle has been determined and propagated. Once this is done, migration can be done. This value is thus very important.
   - **Contents:** `Threshold2 = A_CHOSEN_THRESHOLD`

5. - **ID:** Inlist
   - **Description:** Vector of `inlist`s.
   - **Contents:** `Inlist = Vector inlist;`

6. - **ID:** Outlist
   - **Description:** Vector of `outlists`.
   - **Contents:** `Outlist = Vector outlist;`

7. - **ID:** batch_migration
   - **Description:** Vector of objects used to record all objects that should be migrated at once (see reference paper for details).
   - **Contents:** `batch_migration = Vector Object`

Actions

1. - *ID:* receive_OUTLIST
   - *Description:* Function called when an OUTLIST message is received. Its purpose is to update the corresponding inlist using the list that has been received. We assume that the list given as a parameter is sorted by distance. The operation `convert_OUTLIST_TO_INLIST` is not complex, it simply updates the corresponding inlist with new values.
   - *Input:* newly received outlist, *Output:* none, *Side-effects:* inlist was updated.
   - *Data & Data Structures IDs:* inlist, outlist
   - *Contents:*

```
receive_OUTLIST(outlist new_list)
  inlist := getInlist(new_list.object);
  convert_OUTLIST_TO_INLIST(new_list,inlist)
```

2. - *ID:* local_propagation
   - *Description:* Main function. Local propagation can be easily integrated with a tracing collector like Mark-and-Sweep. Its purpose is

to propagate and update distance and destination values from local roots and inlists to outlists. The **outlist** is recomputed at each call of this function. We assume the existence of the low-level operation **migrate(o,A,B)**, which migrates object o from node A to node B. It takes care of all the details of pointer forwarding and copy of data.

- *Input:* none, *Output:* none, *Side-effects:* the outlist is possibly updated with new distances and destinations.
- *Data & Data Structures IDs:* outlist, inlist, Distance, Destination.
- *Contents:*

```
local_propagation()
   // init
   for each node in Nodes { Outlist[node] := empty }
   // Local roots
   propagate_local_roots()
   // inlist
   batch_migration := propagate_inlists()
   // Global propagation
   for each node in Nodes
       if (batch_migration[node] != empty)
              migrateALL(batch_migration[node], current_node, node)
              // This will migrate all objects in the list
              // batch_migration[node], this operation calls
              // 'migrate(o,A,B)'
       send(node, OUTLIST, Outlist[node])
```

3.
- *ID:* propagate_local_roots
- *Description:* helper function for *local_propagation*. Purpose: trace objects from local roots and propagate distance values to the outlist. Local objects have a (non-recorded) distance value of 0 and outlists will be updated with a distance value of 1.
- *Input:* none, *Output:* none, *Side-effects:* root-reachable outlist elements are updated with a distance of 1 and an estimation of the destination.

- *Data Structures IDs:* outlist
- *Contents:*

```
propagate_local_roots()
   For each local root r
       visit all reachable and not already visited objects o
           from r recursively
       If o has a reference REF to a remote object:
          add(Outlist[node(REF)],
              [REF, // reference
               1,   // distance
               max(rank(node(REF)), rank(current_node))]
                  // estimation of destination
          // add maintains Outlist[node] sorted in increasing
          // order of distances
```

4.
- *ID:* propagate_inlists
- *Description:* helper function for *local_propagation*. Purpose: trace objects and propagate distance values to the outlist. Distance values are given by the inlists entries.
- *Input:* none, *Output:* none, *Side-effects:* outlist elements reachable from inlist elements are updated w.r.t their distances and an estimation of the destination.
- *Data Structures IDs:* inlist, outlist.
- *Contents:*

```
propagate_inlists()
   for each node
       batch_migration[node] := empty

   // inlist is sorted in increasing order of distances
   For each node
    For each Inlist[node] entry ei
       propagated_distance := ei.distance
```

```
        if (ei.destination != -1)
          propagated_destination := ei.destination
        else propagated_destination := rank(current_node)

        // Migration mode?
        // -- migration mode level 1: objects are likely garbage,
        // -- destination has to be determined
        migration_mode := (ei.distance >= Threshold)

        // -- migration mode level 2: destination is probably
        // -- determined by now
        actual_migration_mode := (ei.distance >= Threshold2)

        // Tracing
        visit all reachable and not already visited objects o
           from ei.object recursively
        if (actual_migration_mode)
           add(batch_migration[propagated_destination], o)
               // ready to be migrated
        else if (migration_mode) {
          if  (o has a reference REF to a remote object)
            and (REF is not already in Outlist[node(REF)]) {
             add(Outlist[node(REF)],
                 [REF, propagated_distance + 1,
                  max(rank(node(REF)),propagated_destination)])
              // add maintains Outlist[node] sorted in increasing
              // order of distances
          }
        } else {
          if  (o has a reference REF to a remote object)
            and (REF is not already in Outlist[node(REF)]) {
             add(Outlist[node(REF)],
                 [REF, propagated_distance + 1, -1])
              // we don't care about destination at this point
              // add maintains Outlist[node] sorted in increasing
              // order of distances
          }
        }

    return batch_migration;
```

We have shown here most of the required operations, which should be enough to illustrate the "Actions" part of a Generic GC. We often found that algorithms

are well-detailed in the paper presenting the collector. If this is not the case, designers have to do the work and interpret what has not been explicitly detailed. The templates we created make it easier to list the different elements appropriately. Descriptions are important because they explain how each action can be used and in what context. For example, we can see that the operation called `migrate` should be available in order for `local_propagation` to work properly.

C.4 Cyclic version of SSPC

Basics

- **ID:** Cyclic SSPC
- **Algorithm class:** *Augmented DRL*
- **Inherits from:** SSPC.
- **Description:** SSPC (Stub-Scion Pair Chain) is a distributed reference listing garbage collector. It uses timestamps and timing thresholds to handle late messages and detect crashed spaces (known as nodes in other DGCs). This cyclic extension uses timestamps to reclaim distributed cycles by propagating constantly increasing time marks to reachable objects, while garbage objects are marked with constant timestamps. A "time server" is used to compute a global threshold to reclaim the garbage.
- **Reference papers:**
  - "An implementation for complete asynchronous distributed garbage collection", Fabrice LeFessant, Ian Piumarta and Marc Shapiro, PLDI'98.
  - "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection.", Marc Shapiro, Peter Dickman and David Plainfossé, Rapport de Recherche INRIA, 1992.

- **Assumptions:**

  1. – *Type:* Global timing mechanism.
     – *Comments:* Use of a Lamport clock. See *External algorithms* for details.

- **Conditions of use:** This scheme is not really scalable for two reasons: 1. it uses a DRL and 2. it relies on a server. For these reasons, this DGC is better suited for local distributed systems. It is important to note that this DGC allows and handles failures.

- **Addressed issues:** *fault tolerance, distributed cycles.*

- **Supplies:**

  1. – **Entity:** scion.
     – **Description:** A timestamp is provided for garbage cycle detection.
  2. – **Entity:** all scions.
     – **Description:** A list of remote live stubs is provided to help update remote reachability information.
  3. – **Entity:** space.
     – **Description:** A global minimum timestamp (called `globalmin`) is provided to each space for comparison with timestamps on scions and stubs.

- **External Algorithms**

  1. – *ID:* Globalmin update.
     – *Description:* A dedicated space, called detection server, computes the global threshold use to reclaim garbage cycles. This value is

set using 'localmin' values sent by all spaces participating to the detection of distributed garbage cycles. This operation is called each time a new localmin value is received from a space via a LOCALMIN message (see *Protocols*).

- *Input:* space and value of received localmin, *Output:* none, *Side-effects:* possible new value for globalmin.
- *Data Structures IDs:* localmin[]: array to store localmin values
- *Contents:*

```
GlobalminUpdate(space, localmin_val)
   localmin[space] := localmin_val
   globalmin := min(localmin[])
```

2. - **ID:** Lamport clock.
   - **Description:** Distributed clock to establish consistent times among all nodes of a network.
   - **Purpose:** A common global timing system is needed to reclaim garbage cycle. A Lamport clock is used on each relevant message (i.e the ones corresponding to date propagation) to provide this global time.
3. - **ID:** Negotiation.
   - **Description:** distributed algorithm to establish a global consensus.
   - **Purpose:** Find out what spaces agree to participate to distributed garbage cycles detection.

Main algorithm

The algorithm is based on a traditional reference listing technique, where opaque addressing is used to list the identifiers of the spaces referring to a given object. In

this DGC, a specific terminology is used for opaque addressing entities: a **scion** corresponds to an **entry item** and a **stub** corresponds to an **exit item**.

Initialization

- Set up detection server
- Negotiation to obtain the list of participating spaces (see *External Algorithms*).

Local propagation

- Dates are propagated from scions to stubs and the current date is propagated from local roots to stubs.
- The localmin value is computed according to these newly propagated dates after each local propagation.

Remote propagation

- After a local GC, dates assigned to stubs are sent to corresponding scions.
- The localmin value is sent to the "detection server".

Globalmin

- Computed by the detection server.
- Minimum of all localmin values sent to the server.
- It represents the threshold used to reclaim garbage cycles.

Localmin

- This value is a threshold computed locally. It represents the minimum date that has to be protected on a given space.
- Computed after each local propagation and sent to the detection server.

Protocols

1. - **ID** LOCALMIN
   - **Description** Message sending the localmin value of a space to the Detection Server.
   - **Local actions needed** compute_localmin
   - **Local actions triggered** receive_localmin
   - **Contents**
     - *Number of sites* 2
     - *Ordered list of messages:*

       `(A,DetectionServer,LOCALMIN,[gc_date,localmin])`

       `(DetectionServer,A,ACK,[gc_date,globalmin])`
   - **Set of sites** {A,DetectionServer}

2. - **ID** STUBDATES
   - **Description** This message propagates timestamps between participating spaces. This message is also used by the original SSPC algorithm, because it allows to specify what stubs are still reachable after a local collection on the sender space. This is used to update scions.
   - **Local actions needed** local gc
   - **Local actions triggered** receive_stubdates
   - **Contents**
     - *Number of sites* 2
     - *Ordered list of messages:*

       `(A,B,STUBDATES,[date, List({stub_id, stubdate})])`
   - **Set of sites** A, B

3.
   - **ID** THRESHOLD
   - **Description** Date of the last STUBDATES message received. This is used to update the cyclicthreshold value.
   - **Local actions needed** Receive ACK for LOCALMIN
   - **Local actions triggered** receive_threshold
   - **Contents**
     - *Number of sites* 2
     - *Ordered list of messages:* `(A,B,THRESHOLD,[date])`
   - **Set of sites** A, B

4.
   - **ID** ACK
   - **Description** regular acknowledgment message.
   - **Local actions needed** none
   - **Local actions triggered** none specific
   - **Contents**
     - *Number of sites* 2
     - *Ordered list of messages:*`(A,B,ACK,[message_id])`
   - **Set of sites** A, B

Generic GC for the Cyclic version of SSPC

Basics

- **ID:** Generic GC for Cyclic SSPC.
- **Algorithm class:** *Tracing.*
- **Focus class:** *Heap-focused.*
- **Concurrency class:** *Uniprocessor.*
- **Description:** This is a tracing GC, because local GCs are required to propagate dates from scions to stubs.

Data structures

1. - **ID:** Scion
   - **Description:** This is the equivalent of an entry item. We can use two types of structures: either a scion contains a list of referents with corresponding timestamps or there exists one scion per referent to this object. The second solution seem to have been chosen in the reference paper.
   - **Contents:**

     ```
     { Pointer obj, SpaceID referent,
       Date scionstamp, Date scion_date, Date olddate }
     ```

2. - **ID:** Stub
   - **Description:** This is the equivalent of an exit item.
   - **Contents:**

     ```
     { SpaceID spaceID, Scion scion, Date stub_date, Date olddate }
     ```

3. - **ID:** CyclicThresholdToSend
   - **Description:** Stores cyclicthreshold for each space at different dates. Each date is the date of a GC. Indeed, cyclicthresholds are saved in cyclicthresholdtosend before each local GC. A particular entry is removed when an ACK message is received either for this date or for a more recent one. The values of the most recent removed entry are sent to the corresponding spaces in THRESHOLD messages. This set is ordered in increasing dates (oldest date first).
   - **Contents:** `Map(Space, [Date date, Date cyclicthreshold]);`

4. - **ID:** CyclicThreshold
   - **Description:** Last GC date received from each space in a STUB-DATES message. It is used in a msg THRESHOLD that triggers a "purge" operation of the protected_set.
   - **Contents:** `Date CyclicThreshold[Space]`

5. - **ID:** Space
   - **Description:** Data Structure representing a "Space".
   - **Contents:** `{ SpaceID id, List Stub stubs, List Scion scions }`

6. - **ID:** ProtectNow
   - **Description:** It is a vector, indexed on spaces, that record the minimum of stub olddates. Only stubs for which stubdate is different from olddate are used in the computation. Furthermore, only stubs referencing objects on a given space can be used. For example, to compute ProtectNow[space1], only stubs linked to scions on space1 are used.
   - **Contents:** `Date ProtectNow[Spaces]`

7. • **ID:** ProtectedSet
   • **Description:** Array, indexed on spaces, which records a list of different "protect_now" values that need to be protected. This structure is used to compute 'localmin'. It stores, per space, a list of protect_now dates at different times. Values are kept until we are told (via a THRESHOLD message) that a protect_now value at a certain date has been taken into account (i.e propagated). On the corresponding space, local propagation must have occurred and localmin must have been computed and sent. It protects all dates, per space, that have not been treated.
   • **Contents:**
   `Stack([Date protect_now, Date gc_date]) ProtectedSet[Spaces]`

Data

1. • **ID:** stubs
   • **Description:** stubs for the current space
   • **Contents:** `List(Stub) stubs`

2. • **ID:** scions
   • **Description:** scions for the current space
   • **Contents:** `List(Scion) scions`

3. • **ID:** spaces
   • **Description:** Set of spaces known at a particular space. It contains various information about protected timestamps.
   • **Contents:** `List(Space) spaces`

4. • **ID:** globalmin
   • **Description:** Threshold value used by each local GC to know whether or not a stub should be scanned according to its stubdate. It is computed by the "detection server"
   • **Contents:** `Date globalmin`

5. • **ID:** localmin
   • **Description:** Threshold value computed by each local GC and sent to the "detection server" to compute `globalmin`.
   • **Contents:** `Date localmin`

6. • **ID:** local_roots
   • **Description:** List usually composed of pointers found on the stack, in the registers, and static area.
   • **Contents:** `List(Pointer) local_roots`

7. • **ID:** current_date
   • **Description:** date managed by a Lamport clock.
   • **Contents:** `Date current_date`

8. • **ID:** cyclicThresholdToSend
   • **Description:** set of cyclic thresholds.
   • **Contents:** `CyclicThresholdToSend cyclicThresholdToSend`

9. • **ID:** cyclic_threshold
   • **Description:** current cyclic threshold for this space
   • **Contents:** `CyclicThreshold cyclic_threshold`

10. • **ID:** protected_set
    • **Description:** protected set for this space

- **Contents:** `ProtectedSet protected_set`

11. - **ID:** protect_now
    - **Description:** protect_now set for this space
    - **Contents:** `ProtectNow protect_now`

Actions

1. - *ID:* receive_STUBDATES
   - *Description:* This operation updates the vector of scions on a given space using a list of reachable stubs. The algorithm was taken directly from the paper.
   - *Input:*
     - `space`, ID of the current space.
     - `gc_date`, date of last local GC, used to update the threshold.
     - `stub_set`, vector of stubdates to update scions.
     - `threshold`, value used to reclaim garbage scions.
   - *Output:* none.
   - *Side-effects:* scions are updated with more current dates. Scions that are no longer referenced by any stubs are removed (by not being added to the new set of scions).
   - *Data & Data Structures IDs:* space, scion, cyclic_threshold, SpaceID.
   - *Contents:*

```
receive_STUBDATES(space, gc_date, stub_set, threshold)
    // space is a SpaceID
    increase cyclic_threshold[space] to gc_date
    old_scion_set := space.scions
    space.scions := {}
```

```
for all scion in old_scion_set
    (found, stub_date) := find(scion, stub_set)
    if found or scion.scionstamp > threshold
      if scion.scionstamp < threshold
        increase scion.scion_date to stub_date
      space.scions.add(scion)
```

2.
   - *ID:* receive_THRESHOLD
   - *Description:* Update of the threshold used to detect garbage cycles.
   - *Input:*
     - `space`, ID of the current space.
     - `cylic_threshold`, new date.
   - *Output:* none.
   - *Side-effects:* old protected dates are removed, thus allowing corresponding scions and stubs to be removed if they have not been updated with a new timestamp.
   - *Data & Data Structures IDs:* protected_set, protect_now, space.
   - *Contents:*

```
receive_THRESHOLD(space, cyclic_threshold)
    (protect_now, gc_date) := protected_set[space].top()
    while (gc_date <= cyclic_threshold)
         (protect_now, gc_date) := protected_set[space].pop()
         (protect_now, gc_date) := protected_set[space].top()
    // at this point, all ``too old'' protect_now values
    // have been removed
```

3.
   - *ID:* local_propag
   - *Description:* Local propagation of timestamps and computation of localmin. We assume the existence of the operation `mark_from_root` used to mark objects and propagate dates from a given pointer.

- *Input:* none, *Output:* none, *Side-effects:* Timestamps are propagated from roots and scions to stubs. Localmin is computed after propagation.
- *Data & Data Structures IDs:* cyclic_threshold, cyclicthresholdtosend_set, cyclicthresholdtosend, protected_set, protect_now, localmin, stub, scion, local_roots, spaces, current_date.
- *Contents:*

```
local_propag()
    // timestamp propagation
    increment current_date
    cyclicThresholdToSend.push(current_date,cyclic_threshold[])

    for all r in local_roots
       mark_from_root(r, current_date)
    for all scion in sorted_scions(scions)
       if (scion.scion_date < globalmin)
          scion.pointer := nil
       else
          if scion.scion_date = NOW
             mark_from_root(scion.obj,current_date)
          else
             mark_from_root(scion.obj,scion.scion_date)

    // localmin computation
    for all space in spaces
       for all stub in space.stubs
          if stub.stub_date > stub.olddate
            decrease protect_now[space] to stub.olddate
          stub.olddate := stub.stub_date
       protected_set[space].push(protect_now[space],current_date)
       protect_now[space] := current_date
       send(space, STUBDATES, current_date,
            { for all stub in space.stubs,
               (stub.stub_id, stub.stub_date) })

    localmin := min(protected_set[])
    send(server, LOCALMIN, current_date, localmin)
```

4.
   - *ID:* decrease to
   - *Description:* arithmetic operation.
   - *Input:* two values (dates or int) A and B, *Output:* none, *Side-effects:* A updated with B's value, if B is smaller than A.
   - *Data Structures IDs:* none
   - *Contents:*

```
decrease A to B
   if (A > B) then A := B
```

5.
   - *ID:* increase to
   - *Description:* arithmetic operation.
   - *Input:* two values (dates or int) A and B, *Output:* none, *Side-effects:* A updated with B's value, if B is greater than A.
   - *Data Structures IDs:* none
   - *Contents:*

```
increase A to B
   if (A < B) then A := B
```

Appendix D

Some mappings between Generic GC and stand-alone GCs

D.1 GCW and Mark-and-Sweep

Basics

- **Stand-alone GC ID:** Mark-and-Sweep
- **Generic GC's DGC ID:** GCW
- **Description:** Mark-and-Sweep is particularly suited to serve as a local GC for GCW because this distributed collector has been designed with a local tracing GC in mind (as can be observed in Section C.2 describing its generic GC).

Mapping/Creation of extra data structures

1. - **ID:** EntryItemsVector
   - **Description:** set of entry items.
   - **Affected algorithms:** mark, sweep.

- **Contents:** `Vector(EntryItem)`

2. - **ID:** ExitItemsVector
   - **Description:** set of exit items.
   - **Affected algorithms:** mark, sweep.
   - **Contents:** `Vector(ExitItem)`

Specialization of algorithms

1. - *Algorithm ID:* MarkAndSweep
   - *Description:* We add code to propagate marks from local roots and entry items to exit items. We also add a few instructions for stability detection. Solution 2 of the Generic model has been arbitrarily chosen here. We assume an operation `reclaimExit` for exit items which also sends a decrement message.
   - *Data Structures IDs:* EntryItemsVector, ExitItemsVector, EntryItem, ExitItem, Marks.
   - *Contents:*

```
MarkAndSweep()
   // -------- Init exit items
   For all ex in ExitItemsVector
      ex.oldmark :=  ex.mark
      ex.mark := Marks.NONE
   stableNode := true; // by default the node is stable

   // -------- Mark phase
   CurrentMark := Marks.HARD
   For all r in roots { MarkScan(r) }
   For all en in EntryItemsVector
      if (en.counter = 0) continue
      if (en.mark = Marks.HARD) MarkScan(en.object)
   CurrentMark := Marks.SOFT
   For all en in EntryItemsVector
```

```
      if (en.counter = 0) continue
      if (en.mark = Marks.SOFT) MarkScan(en.object)

   // -------- Check node stability
   // -------- and sweep exit items
   For all ex in ExitItemsVector
      if (ex = Marks.NONE) {
         // we get rid of the exit item and
         // thus do not need to check it
         reclaimExit(ex)
         continue
      }
      if (ex.oldmark != ex.mark)
        { stableNode := false; break; }

   // -------- Sweep objects
   sweep() // regular sweep phase for normal objects

   // -------- Sweep EntryItemsVector
   // Because network messages are asynchronous, counters
   // can be decremented at any time during the GC. Sweeping
   // entry items as late as possible increases the chance of
   // reclaiming a larger number of garbage at this collection.
   for all en in EntryItemsVector
       if (en.counter = 0) reclaim(en)
```

2.
   - *ID:* MarkScan
   - *Description:* Mark an object and scan it for pointers. Specific action is taken when reaching exit items. We assume `hardenMark` which sends a message to "harden" marks on remote entry items.
   - *Data Structures IDs:* Fixed size area, Bitmap, Mixed size area, ExitItemsVector, CurrentMark
   - *Contents:*

```
MarkScan(Pointer p)
  // -------- check
  if (not valid(p)) return
  // -------- Exit item
```

```
if (isExitItem(p)) {
  if (p->mark = HARD) return;
    // if the mark is already HARD, there is
    // nothing more we can do.
  p->mark := CurrentMark;
  if ((p->mark = HARD) and (p->oldmark != HARD))
    hardenMark(p->remoteEntry, p->node);
    stableNode := false; // because message sent
  return;
}
// -------- Regular M&S
if (isFixed(p)) setMark(bitmap(FixedArea(p)))
else setMark(mixedSizeHeader(p))
for all p' in *p { MarkScan(p) }
```

D.2 GCW and Generational

Basics

- **Stand-alone GC ID:** M&S-based generational GC
- **Generic GC's DGC ID:** GCW
- **Description:** This mapping relies on the young generation collection to globally propagate hard marks in a timely manner. Concretely, it assists the old GC by handling certain hard marks and not interfering with the rest of the process which is handled by the old GC.

Mapping/Creation of extra data structures

1. - **ID:** EntryItemsVector
   - **Description:** set of entry items.
   - **Affected algorithms:** mark, sweep.
   - **Contents:** `Vector(EntryItem)`

2. - **ID:** ExitItemsVector
   - **Description:** set of exit items.
   - **Affected algorithms:** mark, sweep.
   - **Contents:** `Vector(ExitItem)`

Specialization of algorithms

1. - *Algorithm ID:* collectYoung
   - *Description:* Collection of the young generation. From the remset, we propagate HARD marks. `scan_mark_and_propagate` is assumed, it traces objects and if it encounters an exit item, it possibly updates the mark to HARD and globally propagate it. If this is the case, a flag is set to let the old collection know in order to properly compute stability.
   - *Data Structures IDs:* EntryItemsVector, EntryItem, Marks, YGRemSet.
   - *Contents:*

```
collectYoung()
 for each r in root
   if ((youngGen.begin <= r) and (r < youngGen.end))
    scan_mark_and_propagate(r,Marks.HARD)
 for each p in YGRemSet
   if ((youngGen.begin <= p) and (p < youngGen.end))
    scan_mark_and_propagate(p,Marks.HARD)
 for each e in EntryItemsVector
   if ((youngGen.begin <= e.object)
       and (e.object < youngGen.end))
     if (e.mark == Marks.HARD)
       scan_mark_and_propagate(e.object,Marks.HARD)
 for each e in EntryItemsVector
   if ((youngGen.begin <= e.object)
       and (e.object < youngGen.end))
```

```
        if (e.mark == Marks.SOFT)
          scan_mark_and_propagate(e.object,Marks.SOFT)
```

2.
   - *Algorithm ID:* collectOld
   - *Description:* Collection of the entire heap. Stability is also computed using young collection information.
   - *Data Structures IDs:* EntryItemsVector, ExitItemsVector, EntryItem, ExitItem, Marks.
   - *Contents:*

```
collectOld()
   call normal collector (Mark-and-Sweep or Mark-and-Copy)
       propagating marks from roots and entry items.
   promote all live young objects
   empty YGRemSet
   Compute stability using young generation information
       (a young GC may have destabilized the node).
```

D.3 GCW and MOS: Solution 1

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** GCW
- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping shows the simplest and least optimal solution.

Specialization of algorithms

As we have seen in the Generic GC description, there are two important aspects that the local GC has to deal with: mark propagation and local stability detection. To do this, the Generic GC relies on a single full collection of the heap. Unfortunately, MOS never looks at the heap entirely at once. As it is an incremental collector based on a generational technique, each collection will work either on the young generation or on the young generation and a car.

This solution directly uses the algorithms described in the Generic GC. No adaptation is required, and a function is regularly called to discover node stability and propagate marks by scanning all the objects. There are two solutions to call this function: **explicit call** or **timing thread** regularly waking up, blocking all other activities, and running the function. It is important to note that, most of the time, it will be possible to adapt the stand-alone GC to the Generic GC by using this technique (a special function to be called when a job has to be done). This allows to integrate the mapping into existing schemes quickly and easily. Unfortunately, this might not lead to an efficient result and may break the essence of the stand-alone collector.

D.4 GCW and MOS: Solution 2

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** GCW
- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping stores marks on each remembered set.

Mapping/Creation of extra data structures

1. - **ID:** RemSetEntry
   - **Description:** Extension of MOS's `RemSetEntry` data structure to add marks.
   - **Affected algorithms:** collectCar.
   - **Contents:**

     ```
     { Pointer obj, Pointer origObj, Pointer origCar, Marks mark }
     ```

Specialization of algorithms

Marks are maintained on remembered sets. Each collection of a car should propagate marks from the car's remset to other cars' remsets. This can be seen as if the heap were distributed with a car representing a node and marks are propagated from car to car. A "phase" needs to be defined in order to help evaluate the stability of the entire heap. At the beginning of this phase, all cars are tagged and new cars will be considered "permanent" and will not be involved in the evaluation of local stability. The final marks on exit items depend on the local marking phase. This solution could be rather conservative, and could slow down the entire DGC process, but it has the advantage to respect MOS' spirit.

D.5 GCW and MOS: Solution 3

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** GCW

- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping uses a remset for each exit item and traces objects back to the origin of the paths. In the worst case, it is as disruptive as Solution 1, but could be cheap depending on the local graph of objects.

Mapping/Creation of extra data structures

1. - **ID:** ExitItem
   - **Description:** For outgoing pointers. It inherits from GCW's exit item and add a remset.
   - **Contents:** `ExitItem: GCW.ExitItem { MOS.RemSet remset }`

Specialization of algorithms

- *Technique:* Each exit item is backtraced to either a root, an entry item, or nothing (if unreachable). In the best cases, a HARD mark is found quickly and the mark is assigned and propagated globally. In the worst case, the mark is SOFT. This backtracing is regularly called (by scheduling the call within the round-robin mechanism used to choose cars to collect).
- *Main Algorithm:*

```
for each exit item e
   rs := e.remset
   e.mark := backtrace(rs.allReferrents())
```

- *Additional algorithms:*

```
backtrace(List(Pointer) refs) {
  if (there is a root in refs) return HARD
  if (there is a HARD entry in refs) return HARD

  mark := NONE
  for all r in refs
    if (r is an entry item) {
      if (r.mark > mark) mark := r.mark
      continue
    }
    c := car(r)
    l := listref(r,c)
    if (l = nil) continue
    m := backtrace(l)
    if (m = HARD) return HARD
    if (m > mark) mark := m

  return mark
}

listref(Pointer r, car c) {
  l := nil
  for each p in c.remset.allObjects()
    if (r reachable from p) l := cons(p,l)
  return l
}
```

- *Comments:* As can be seen, this solution attempts to backtrace exit items in order to find out what marks to associate with them. In the worst case, the graph of objects is linear and all cars must be visited before finding the mark. Another worst case can be seen when the final mark of the exit item is SOFT, in this case all possible paths are visited. It may be advisable to use a technique similar to the one described in Chapter 4 and discover all marks per car rather than backtracing objects up to a root or an entry item thus possibly losing opportunity for locality of treatment. We observe that when a HARD mark is found, the backtrace stops, thus making the

process more efficient. We note that MOS moves objects from car to car or car to train with the effect of gathering related objects. Performance will thus likely improve over time, as paths are likely to span a smaller number of cars.

D.6 GCW and MOS: Solution 4

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** GCW
- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping associates a mark with each car and a regularly called function propagates these marks conservatively to exit items. This is the fastest solution because it does not visit objects, only remembered sets.

Mapping/Creation of extra data structures

1.
   - **ID:** Car
   - **Description:** This extension of a car adds a mark.
   - **Contents:** `Car: MOS.Car { GCW.Marks mark }`

Specialization of algorithms

A local propagation function is regularly called (by scheduling it within the round-robin mechanism used to choose cars to collect) to propagate marks. Starting from roots and entry items, a mark is associated with each car by looking at its

remembered set. If objects of this car are referenced either by a root, a HARD entry, or an object in a HARD car, the car becomes HARD. If an exit item is referenced by a HARD car, it is marked HARD.

We note one problem with this solution: cycles. Objects in several cars can form a cycle. In this case, it is not possible to assign a definitive mark to a given car, except if at least one HARD car references it (which will happen often). If no HARD car is known to reference the car and its mark depends on the mark of another car – not visited yet –, then this car is listed in an "incomplete" list. Once all cars have been seen once, we treat cars in the list with a complexity of $O(n^2)$ if n is the number of cars in the list. Finally, cars remaining in the list are part of a cycle. In this case, for each car, we examine the marks assigned from cars not in the list, and the highest mark (SOFT or NONE) is assigned to all cars part of the cycle. We note that a HARD can not be assigned at this point, because it would have been detected earlier.

This solution may be too conservative, practical experiments would be required to assess it. We note that MOS moves objects from car to car or car to train with the effect of gathering related objects. As for solution 3 (see Section D.5), this is likely to improve efficiency, not in terms of performance this time, but in terms of precision. Independent objects have less chance to be in the same car, increasing the chance for the car to obtain a more precise mark.

D.7 GCW and RC

Basics

- **Stand-alone GC ID:** Reference Counting
- **Generic GC's DGC ID:** GCW
- **Description:** Local RC needs to show a solution to map a tracing-like

local GC. A specific "GCW" function will be used to propagate marks and compute stability.

Mapping/Creation of extra data structures

1. - **ID** ExitItem
   - **Description** For outgoing pointers. It is a proxy for a remote object. We have to specify this data structure, because we add a counter of references to it.
   - **Affected algorithms:** none.
   - **Contents**

```
{ int counter, Marks mark, Marks oldmark,
  RemoteNode node, Pointer remoteEntry }
```

Specialization of algorithms

Algorithms described in the Generic GC will be used as such. No adaptation is required. Operations from the model is integrated in a specific function that should be called regularly. It propagates marks and computes local stability (see DGC's model in Section C.2). There are three solutions to call this function:

- Explicit call at the discretion of the programmer.
- Timing thread regularly woken up, which blocks all other activities and executes the function.
- Callbacks associated to specific objects. When they are reclaimed, the function is called. This solution is application-dependent.

D.8 Cyclic SSPC and Mark-and-Sweep

Basics

- **Stand-alone GC ID:** Mark-and-Sweep
- **Generic GC's DGC ID:** Cyclic SSPC
- **Description:** This version of SSPC works well with a tracing algorithm, and is actually designed to work with such a local GC. The mapping in this case is obvious because this DGC was created to work with this particular stand-alone collector. There is only a work of integration and not a work of algorithm mapping.

Specialization of algorithms

We do not present a mapping model for this scenario. A similar work was done with M&S and GCW in Section D.1, please refer to it for an example of integration. We see here that mapping or integration solutions can be reused from one model to another. This case is particularly simple, but we also provide a more complex example with Section D.5 and Section D.12 for instance.

D.9 Cyclic SSPC and RC

Basics

- **Stand-alone GC ID:** RC
- **Generic GC's DGC ID:** CyclicSSPC
- **Description:** Local RC needs to show a solution to map a tracing-like local GC. A specific "SSPC" function will be used to propagate timestamps

and compute the localmin value. See an example of adaptation of data structures in Section D.7.

Specialization of algorithms

Algorithms described in the Generic GC will be used as such. No adaptation is required. Operations from the model is integrated in a specific function that should be called regularly. It propagates timestamps and computes localmin values (see DGC's model in Section C.4). There are three solutions to call this function:

- Explicit call at the discretion of the programmer.
- Timing thread regularly woken up, which blocks all other activities and executes the function.
- Callbacks associated to specific objects. When they are reclaimed, the function is called. This solution is application-dependent.

D.10 CSSPC and MOS: Solution 1

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** CSSPC
- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping shows the simplest and least optimal solution.

Specialization of algorithms

As we have seen in the Generic GC description, there are two important aspects that the local GC has to deal with: timestamps propagation and localmin computation. To do this, the Generic GC relies on a single full collection of the heap. Unfortunately, MOS never looks at the heap entirely at once. As it is an incremental collector based on a generational technique, each collection will work either on the young generation or on the young generation and a car.

This solution directly uses the algorithms described in the Generic GC. No adaptation is required, and a function is regularly called to propagate timestamps and compute localmin by scanning all the objects. There are two solutions to call this function: **explicit call** or **timing thread** regularly waking up, blocking all other activities, and running the function. It is important to note that, most of the time, it will be possible to adapt the stand-alone GC to the Generic GC by using this technique (a special function to be called when a job has to be done). This allows to integrate the mapping into existing schemes quickly and easily. Unfortunately, this might not lead to an efficient result and may break the essence of the stand-alone collector.

D.11 CSSPC and MOS: Solution 2

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** CSSPC
- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping stores timestamps on each remembered set.

Mapping/Creation of extra data structures

1. - **ID:** RemSetEntry
   - **Description:** Extension of MOS's `RemSetEntry` data structure to add timestamps.
   - **Affected algorithms:** collectCar.
   - **Contents:**

     ```
     { Pointer obj, Pointer origObj, Pointer origCar, Timestamp ts }
     ```

Specialization of algorithms

Timestamps are maintained on remembered sets. Each collection of a car should propagate timestamps from the car's remset to other cars' remsets. This can be seen as if the heap were distributed with a car representing a node and timestamps are propagated from car to car. A "phase" needs to be defined in order to help evaluate the stability of the entire heap. At the beginning of this phase, all cars are tagged and new cars will be considered "permanent" and will not be involved in the evaluation of localmin. The final timestamps on stubs depend on this local marking phase. The solution is likely to be too conservative, and could slow down the entire DGC process, but it has the advantage to respect MOS' spirit.

D.12 CSSPC and MOS: Solution 3

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** CSSPC

- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping uses a remset for each stub and traces objects back to the origin of the paths. In the worst case, it is as disruptive as Solution 1, but could be cheap depending on the local graph of objects.

Mapping/Creation of extra data structures

1. - **ID:** Stub
   - **Description:** For outgoing pointers. It inherits from CSSPC's stub and add a remset.
   - **Contents:** `Stub: CSSPC.Stub { MOS.RemSet remset }`
2. - **ID:** CurrentDate
   - **Description:** This global variable holds the most recent time currently found in backtracing a particular stub.
   - **Contents:** `Date CurrentDate`

Specialization of algorithms

- *Technique:* Each stub is backtraced to either a root, a scion, or nothing (if unreachable). In the best cases, a root is encountered and the timestamp NOW is directly assigned to the stub. In the worst case, no root is quickly found and timestamps are taken from scions. This backtracing is regularly called (by scheduling the call within the round-robin mechanism used to choose cars to collect).

- *Main Algorithm:*

```
for each stub s
   rs := s.remset
   CurrentDate := 0
   backtrace(rs.allReferrents())
   s.stubdate := CurrentDate
```

- *Additional algorithms:*

```
backtrace(List(Pointer) refs) {
  if (there is a root in refs)
    CurrentDate := Date.NOW
    return

  for all r in refs
    if (r is a scion) {
      if (r.sciondate > CurrentDate) CurrentDate := r.sciondate
      continue
    }
    c := car(r)
    l := listref(r,c)
    if (l = nil) continue
    backtrace(l)
    if (CurrentDate = Date.NOW) return
}

listref(Pointer r, car c) {
  l := nil
  for each p in c.remset.allObjects()
    if (r reachable from p) l := cons(p,l)
  return l
}
```

- *Comments:* As can be seen, this solution attempts to backtrace stubs in order to find out what timestamps to associate with them. In the worst case, the graph of objects is linear and all cars must be visited before finding a timestamp. Another worst case can be seen when the final timestamp of

the stub is not NOW, in this case all possible paths are visited. It may be advisable to use a technique similar to the one described in Chapter 4 and discover all timestamps per car rather than backtracing objects up to a root or a scion thus possibly losing opportunity for locality of treatment. We observe that when a root is encountered, the backtrace stops, thus making the process more efficient. Unlike the mapping GCW-MOS, the fact that MOS moves objects from car to car or car to train with the effect of gathering related objects has little impact on performance, because timestamps are all different and we can not stop until we found all of them.

D.13 CSSPC and MOS: Solution 4

Basics

- **Stand-Alone GC ID:** MOS
- **Generic GC's DGC ID:** CSSPC
- **Description:** As specified in its model, MOS is a region-focused GC. It never visits the entire heap, which makes a direct use of the generic GC solution impossible. A true work of adaptation is required in this case. This mapping associates a timestamp with each car and a regularly called function propagates these timestamps conservatively to stubs. This is the fastest solution because it does not visit objects, only remembered sets.

Mapping/Creation of extra data structures

1. - **ID:** Car
   - **Description:** This extension of a car adds a mark.
   - **Contents:** `Car: MOS.Car { Date timestamp }`

Specialization of algorithms

A local propagation function is regularly called (by scheduling it within the round-robin mechanism used to choose cars to collect) to propagate timestamps. Starting from roots and scions, a mark is associated with each car by looking at its remembered set. The most recent timestamp of the referents (car or root) is chosen for the car. Stubs are assigned the maximum timestamp among their referents. This solution may be too conservative, practical experiments would be required to assess it.

We note one problem with this solution: cycles. Objects in several cars can form a cycle. In this case, it is not possible to assign a definitive timestamp to a given car, except if at least one car with a NOW timestamp references it. If no such car is known to reference this car and its timestamp may depend on the timestamp of another car – not visited yet –, then this car is listed in an "incomplete" list. Once all cars have been seen once, we treat cars in the list with a complexity of $O(n^2)$ if n is the number of cars in the list. Finally, cars remaining in the list are part of a cycle. In this case, for each car, we examine the timestamps assigned from cars not in the list, and the more recent timestamp is assigned to all cars part of the cycle. We note that a NOW timestamp can not be assigned at this point, because it would have been detected earlier.

D.14 Controlled Migration and MC

Basics

- **Stand-alone GC ID:** MC
- **Generic GC's DGC ID:** Controlled Migration
- **Description:** The Mark-And-Copy algorithm is a tracing collector. Adaptation is not tricky. Indeed, the reference paper of the DGC specifies that

"a copying collector can be used as long as one root is traced completely before the next untraced root is copied".

Mapping/Creation of extra data structures

1. - **ID:** CurrentDistance
   - **Description:** Extra data to keep the current distance in memory rather than propagating it as an extra parameter to each function.
   - **Affected algorithms:** MarkCopyGC.
   - **Contents:** `Distance CurrentDistance`
2. - **ID:** CurrentDestination
   - **Description:** Extra data to keep the current destination in memory rather than propagating it as an extra parameter to each function.
   - **Affected algorithms:** MarkCopyGC.
   - **Contents:** `Destination CurrentDestination`

Specialization of algorithms

1. - *ID:* MarkCopyGC
   - *Description:* main function. Modified.
   - *Data Structures IDs:* CurrentDistance, CurrentDestination, Object, inlist.
   - *Contents:*

```
MarkCopyGC
   // From roots
   [r1,r2] := getRoots()
   indexTo := beginToArea
   MarkCopy(r1,r2)
```

```
// deal with inlist. Reminder: inlist is sorted.
for each entry in inlist
   CurrentDistance := entry.distance
   CurrentDestination := entry.destination
   p := entry.object
   if (header(*p).forwarded) update(entry,p)
   else
     Copy p
     MarkCopy(p,p+header(*p).size)

// At the end ...
reverseToFrom()
```

2.
   - *ID:* MarkCopy
   - *Description:* Most important function. Trace all objects and copy them over to the 'To' part.
   - *Data Structures IDs:* Object, outlist.
   - *Contents:*

```
MarkCopy(a1,a2)
  For each valid pointer p in [a1,a2]
     // valid means that it is actually a pointer and it
     // points in the From area or to a stub

     if (header(*p).forwarded) update(p,*p)
     else {
        // *** Extra code ***
        for each remote pointer REF in (*p)
          if (REF not in outlist)
            add(Outlist[node(REF)],
                [REF, CurrentDistance+1,
                 max(CurrentDestination, rank(node(REF)),
                     rank(current_node))
                      // estimation of destination
                ]
        // *** End extra code ***

        Copy p
        MarkCopy(p,p+header(*p).size)
```

D.15 Controlled Migration and Explicit

Basics

- **Stand-alone GC ID:** Explicit management
- **Generic GC's DGC ID:** Controlled Migration
- **Description:** This mapping is special in the sense that local memory management is not using a garbage collector. Traditional memory primitives such as `malloc` and `free` are used by the programmer, who is then responsible for allocation *and* deallocation.

Specialization of algorithms

Algorithms described in the Generic GC will be used as such. No adaptation is required. This will be part of a function that should be called regularly. This function computes and propagates estimations of distances and destinations (see DGC's model in Section C.3). There are two solutions to call this function:

- Explicit call at the discretion of the programmer, or
- Timing thread regularly woken up, which blocks all other activities and executes the function.

D.16 DRC and Mark-and-Sweep

Basics

- **Stand-alone GC ID:** Mark-and-Sweep.
- **Generic GC's DGC ID:** DRC.
- **Description:** This adaptation proves quite simple. Opaque addressing entities are added to the heap and the Mark-and-Sweep algorithm should be modified to handle them.

Specialization of algorithms

The mark phase should use all the entry items as roots. However, when an entry item has a counter of zero, it should not be considered as root, because it is garbage. The sweep phase reclaims zero-ed entry items and non-traced exit items. When an exit item is reclaimed, a DECREMENT message (see DRC model in Section C.1) should be sent.

VITA

Name: Yannis Chicha

Place of birth: Blois, France

Year of birth: 1973

Post-secondary Education and Degrees:

IUT Informatique (college-like)
Orléans, France
1991-1993 DUT

Ecole Supérieure en Sciences Informatiques (ESSI)
Sophia-Antipolis, France
1993-1996 DEA (MSc-like) and Engineering degree in software development.

The University of Western Ontario
London, Canada
1997-2002 PhD

Related work experience:

Course instructor
The University of Western Ontario
1999

Teaching assistant
The University of Western Ontario
1997-2000

Research assistant
The University of Western Ontario
1997-2002