CYCLIC DISTRIBUTED GARBAGE COLLECTION

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Helena Cristina Coutinho Duarte Rodrigues
November 1998

To Luís and my family

Acknowledgements

I warmly thank my supervisor, Richard Jones, for his care, encouragement and patience. I greatly appreciate the guidance he has given me over the course of my studies.

I thank everyone in the computing laboratory for providing such a pleasant working environment. Especially, all the members of the TCS group, and in particular Howard Bowman, for their encouragement of my participation in the TCS seminars.

I have made many friends during my stay in Canterbury. I warmly thank them all. To my office mates Eduardo Rojas-Vegas, Eduardo Albuquerque and Carlos Ferraz for a wonderful office (latin) atmosphere. To the group that joined everyday for a relaxing and joyful lunch break. To Jason and Vince for being so wonderful office mates and for all the fun we have together. To Catarina, João Corte-Real, Maria, Paula, Regina, Júlia and Pedro, Paul and Valéria, for their care, support and constant friendship. To Manel and Helena and Filipe, Miguel and Francisco for making me feel more close to Portugal.

I warmly thank Geraldina for her friendship and constant companionship.

I thank my family and all my friends in Portugal for always being so supportive and caring.

Finally, I thank Luís for all his love and encouragement.

I acknowledge the financial support of JNICT, Portugal, without whom this work would not have been possible.

Abstract

With the continued growth of distributed systems as a means to provide shared data, designers are turning their attention to garbage collection, prompted by the complexity of memory management and the desire for transparent object management. Garbage collection in very large address spaces is a difficult and unsolved problem, due to problems of efficiency, fault-tolerance, scalability and completeness. The collection of distributed garbage cycles is especially problematic. This thesis presents a new algorithm for distributed garbage collection and describes its implementation in the Network Objects system. The algorithm is based on a *reference listing* scheme, which is augmented by *partial tracing* in order to collect distributed garbage cycles. Our collector is designed to be flexible, allowing efficiency, promptness and fault-tolerance to be traded against completeness, albeit it can be also complete. Processes may be dynamically organised into groups, according to appropriate heuristics, in order to reclaim distributed garbage cycles. Multiple concurrent distributed garbage collections that span groups are supported: when two collections meet they may either merge, overlap or retreat. This choice may be done at the level of different partial tracings, of processes or of individual objects. The algorithm places no overhead on local collectors and does not disrupt the collection of acyclic distributed garbage. Partial tracing of the distributed graph involves only objects thought to be part of a garbage cycle: no collaboration with other processes is required.

Contents

List of Figures

Chapter 1

Introduction

The development of computer and high-speed network technology has led to the introduction of computing systems composed of large numbers of processors connected by high-speed networks that appear to users as a single computing system; these processes may co-operate in solving computational problems. Such distributed systems can make effective use of loosely coupled multiprocessor systems. In these systems, the network provides communication amongst several processors, which do not have access to a common physical memory. Distribution allows multiple users to access the system simultaneously, regardless of their physical location. More importantly, it allows users to share data. Another benefit of distributed systems is fault-tolerance: a distributed system is able to continue despite the failure of one process.

An essential part of tomorrow's computing world will be workers and organisations carrying out cooperative tasks interacting via shared information. The need for dara sharing is well known in applications such as interactive computer aided design, office information systems, engineering databases, medical imaging systems, geographical information processing systems, biological information systems, and others. Information is shared either concurrently or at different times, thus it must be available at different locations and may *persist* beyond completion of a particular application.

Many recent distributed systems have been developed with *objects* as their main structuring concept, as this offers a suitable paradigm for distributed computing. An object-based programming language encourages a methodology for designing and creating an application as a set of autonomous and co-operative objects, whereas a distributed operating system permits a collection of workstations or personal computers to be treated as a single entity. These two concepts together are the basis for *distributed object-based programming systems* (Chin and Chanson 1991).

In general, the objects in such systems are *dynamically* created, and persistent, *i.e* conceptually they live forever. Such address spaces are usually composed of a large number of objects — Large Address Spaces. The implementation of such systems must deal with scalability problems. Accessing an object entails finding its references by navigation from the active part of the system — *roots*. Objects are considered *garbage* if they are not reachable from such roots. Garbage objects should be removed from the system. This can be done either via manual memory management, or automatically via *Garbage Collection* (GC) (Wilson 1992, Jones 1996).

A program that uses explicit deallocation rather than garbage collection needs to keep track of object reachability to know when to deallocate an object. Keeping track of object reachability in a large shared address space where different programs written by different programmers exchange references to objects, becomes impractical and error-prone as the number of programs increases. This is also true for uniprocessor systems. This is because objects are potentially shared among independent threads of control, none of which can have a comprehensive view of the overall object graph, and because objects may outlive the thread of control that created them. Explicit object deallocation requires all programs to agree explicitly when to deallocate an object.

Distributed garbage collection is a difficult problem. It should collect *all* the garbage of a system but still be prompt, that is, rates of collection should match rates of allocation of objects. Moreover, a distributed garbage collector must deal with lost, out of order and duplicated messages, process crashes, long lasting network failures, and problems of scalability.

There are many algorithms for distributed garbage collection in the literature (Plainfossé and Shapiro 1995). Each of these algorithms solves some problems but leaves other problems unresolved. For example, there are algorithms that collect cycles of garbage using some form of complementary tracing but require global synchronisation (Hudak and Keller 1982, Augusteijn 1987, Derbyshire 1990, Juul and Jul 1992) making the algorithm unscalable. Other algorithms are efficient at passing references, but are not fault-tolerant (Bevan 1987, Watson and Watson 1987, Piquer 1991, Dickman 1992). Other are scalable and deal with process and message failure, but do not handle cyclic data (Shapiro, Dickman and Plainfossé 1992, Plainfossé and Shapiro 1992, Birrel, Evers, Nelson, Owicki and Wobber 1994, Maheshwari and Liskov 1994). Others migrate objects until an entire garbage cyclic structure is eventually held within a single process where it can be collected by the local collector (Shapiro, Gruber and Plainfossé 1990, Maheshwari and Liskov 1995), but migration is communication-expensive.

However, unreclaimed garbage is particular undesirable in long-lived systems, especially persistent systems, where even small amounts of uncollected garbage can accumulate over time to cause a significant storage loss.

In the next sections we will introduce the background from which garbage collection is investigated in this thesis. First we introduce distributed object systems and different solutions for object sharing. We intend to briefly acquaint the reader with the different fields of garbage collection in large address spaces. Next, we describe in more detail the RPC-based computational model, as this is the target of our system.

We also present the generic goals of distributed garbage collection, and state our primary goals. Finally we explain the structure of this thesis.

1.1 Why Garbage Collection?

Heap allocation is required for objects that may survive the procedure that created them. If these objects are passed to further procedures or functions it may be impossible for the programmer or compiler to determine at compile-time at which point it is safe to deallocate them. The prevalence of sharing and delayed execution of suspensions means that some programming languages have unpredictable execution orders. For them garbage collection is mandatory (Jones 1996).

Garbage collection has been a research topic for more than 40 years (McCarthy 1960, Collins 1960). It was first investigated in the domain of symbolic programming languages, for example Lisp. Applications written in such languages had complex data graphs. In such applications, memory management is an intricate problem. Today, object-oriented languages and systems face the same problem. With the advent of distributed and persistent systems, the need for GC has increased even further. In such systems, manual memory management becomes a complex task as the number of objects, references and users scales up.

Automatic garbage collection is to be preferred to user-controlled memory management for many reasons. Programmer-controlled memory management is error-prone. The programmer tends to make two mistakes. One mistake is that he fails to free a resource when it is no longer used. This leads to storage leaks and performance degradation. The second mistake is that he returns a resource that is still in use, leading to dangling references. Both mistakes are difficult to detect and recover from, especially in systems managing persistent data. Garbage collection relieves the programmer from the burden of discovering memory management errors by ensuring they cannot happen. A considerable proportion of development time may be spent on bugs of this kind (Rovner 1985). Object-oriented or object-based programming languages typically allocate a greater proportion of program data structures in the heap and generate complex data structures. This only increases the complexity of explicit memory management.

Consequently, garbage collection also provides a better division of responsibility. The task of programming becomes easier and productive increases when memory management is no longer a concern. Programs become shorter and simpler.

Other issues relate to abstraction and modularity. Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies (Wilson 1992). If objects must be deallocated explicitly, some modules must be responsible for knowing when other modules are not interested in a particular object, to prevent one module from causing the failure of another through space leaks or premature reclamation of storage. This introduces nonlocal book-keeping: the behaviour of a module is no longer independent from the context in which it is used. This reduces abstraction and extensibility, because when new functionality is implemented, the book-keeping code must be updated. Manual reclamation is often tightly coupled to the application, making any further modification difficult.

All these reasons apply equally to distributed object-based systems. Moreover, as we have already said, explicit garbage collection is a complex task in distributed systems where objects are highly shared among different programs. The design of an efficient distributed algorithm for managing distributed data is complex: local collectors must be coordinated to consistently keep track of changing references between address spaces.

Distributed garbage collection contributes to transparency in distributed systems: just as modern distributed systems support transparent, uniform placement and invocation of both local and remote objects, so should they also support transparent object management, including reclamation.

With the advent of persistent programming languages and database systems that provide general purpose programming capabilities, garbage collection issues are becoming more relevant to designers of systems that manage persistent data (Franklin, Copeland and Weikum 1989).

Persistent Object stores, also known as a stable heap, are found in many object databases, persistent programming languages and environments, and distributed shared memory systems. Garbage collection is an important issue in persistent object stores. First, the object graphs of applications over persistent stores are complicated, which makes manual storage management increasingly difficult and error-prone, often resulting in dangling pointers and storage leaks. This is because objects are potentially shared among independent programs that cannot have a comprehensive view of the overall object graph, and because objects outlive the program that created them.

Second, garbage collection is necessary to support the property of persistence by reachability. This approach provides true orthogonality of object types and persistence — objects of any type become persistent and operations can be applied to an object regardless at whether it is persistent or not. Any object that is reachable from a persistent object becomes persistent itself (Atkison, Bailey, Chisholm, Cockshott and Morrison 1983).

Third, compaction and clustering of objects improves efficiency. Databases may contain gigabytes of data. Garbage is expensive. If disk blocks contain a large percentage of garbage, disk I/O traffic may be drastically increased. Deferring garbage collection may adversely affect performance (Franklin et al. 1989).

Garbage collection does not, however, guarantee perfect utilisation of memory; the programmer may still, for example, construct ever-expanding data structures that fill the address space. Furthermore, there are costs involved in garbage collection which, although comparable with the cost of manual memory management (Zorn 1992), are non-trivial and which might exceed the cost of doing no recycling of memory in small, short-lived applications. However, garbage collection has advanced rapidly and is now a robust, mature technology (Wilson 1992, Jones 1996).

1.2 Distributed Object-based Programming Systems

Distributed object-based programming system attempts to hide the underlying distribution thus giving the programmer the illusion of a non-distributed system. Using this model, programmers interact with a single conceptual system which fully manages distribution. Examples of such systems include Emerald (Jul, Levy, Hutchinson and Black 1988), IK Platform (Sousa, Sequeira, Zúquete, Ferreira, Lopes, Pereira, Guedes and Marques 1993), Network Objects (Birrel, Evers, Nelson, Owicki and Wobber 1993), CORBA (Vinoski 1993), JAVA Remote Method Invocation protocol (Gosling and McGilton 1995), Microsoft DCOM, Thor (Liskov, Day and Shrira 1992) and Larchant (Ferreira and Shapiro 1996).

The main advantage of distributed object-based programming systems is a simple conceptual framework that normally translates to a simple programming environment. The programmer does not need to understand the complexity necessary to manage distribution, deal with partial failures, optimise the placement of objects, or locate computations. In a distributed system all of these are intended to be performed automatically — and transparently — by the support system.

The most important challenges for garbage collection in these systems are that they feature:

Concurrency A distributed system provides inherent concurrency, *i.e.* it is possible to have more than one part of an application running at the same time. In particular, we may have the application program and the garbage collector running at the same time.

Asynchrony A large class of problems in distributed systems can be cast as executing some notification or reaction when the global state of the system satisfies a particular condition. Thus, the ability to construct a global state and evaluate a predicate over such a state constitutes the core of solutions to many problems in distributed systems (Babaoglu and Marzullo 1993).

The global state of a distributed system is the union of the states of the individual processes. Given that the processes of a distributed system do not share memory but instead communicate solely through the exchange of messages, a process that wishes to construct a global state must communicate with the other processes through message exchanges, which are expensive and unreliable.

Partial failures Distributed systems may be partitioned by break-downs of processors or communication links.

Availability In theory, distributed systems can be more reliable than centralised ones, since if a machine crashes others may keep functioning. This property should not be ignored if we want to increase availability and reliability. When independent failure is properly harnessed by replicating functions on independent components, multiple components failures are required before system availability and reliability suffer (Schroeder 1993).

Scalability Distributed systems may be augmented easily by any number of processors.

Remote communication at the programming language level may be accomplished through any number of paradigms including message-passing, *e.g.* Remote Procedure Calls (RPC) (Birrel and Nelson 1984, Jul et al. 1988, Sousa et al. 1993, Vinoski 1993, Birrel et al. 1993, Gosling and McGilton 1995), transactions (Ozsu, Daylal and Valduriez 1994, Liskov et al. 1992) and distributed shared memory (Nitzberg and Lo 1991, Ferreira 1996). This results in different computation models and solutions for data sharing in distributed systems as RPC-based systems, Object Oriented Database Management systems and Distributed Shared Memory systems respectively.

Also, the relationship between the processes and the objects of a distributed object-based programming system characterises the composition of the objects. Processes may either be separate and temporarily bound to the objects they invoke, or they may be coupled and permanently bound to the objects in which they execute. These two approaches correspond to the *passive object mode* and the *active object model*, respectively (Chin and Chanson 1991):

Passive Object Model Passive objects store data and the computational thread of control is external to them. Once a passive object is no longer referenced from any other object it is garbage and its memory is free to be re-allocated.

Active Object Model Whenever an object controls its computational thread it is called an active object. Their management is more complex than the passive one, because reachability and state may need to be analysed simultaneously. A passive garbage object wastes space only, while an active garbage object consumes processing power and may also waste unbounded amounts of memory.

All these factors influence the job of a garbage collector. In the next sections we will introduce Actor systems, a computational model of active objects, RPC-based systems, Object Oriented Database Management systems and Distributed Shared Memory systems and outline their garbage collection job. Only the Actor model is of Active Object model type. In the remainder of this section and the rest of this thesis we are only concerned with the Passive Object model.

1.2.1 Actors

Actor systems (Agha 1986) are of Active Object Model type. In an Actor system each object contains a thread of control and a message queue, as well as encapsulated behaviour and state, including references to other Actors. Actors exchange messages between each other and this is the only way that one Actor can influence the actions of another Actor. The processing of messages by the embedded thread within an Actor may cause the Actor to change its subsequent behaviour.

The key distinction, for the purpose of garbage collection, between Actor systems and passive objects systems is that Actors contain a thread of control at all times. Traditionally, the definition of 'roots' used by garbage collection algorithms includes the stack associated with every thread in the system. If this was done in an Actor system, every object would have to be considered as live, which is inappropriate. One the other hand, an Actor A which holds a reference to another Actor B that is live might send a message to B which contains a self-reference. In this case, B would then hold a reference to A, and since B is live A must also be live. If A was not considered a root at the time of the garbage collection, A would not be reachable from any root and hence would be unsafely discarded. Consequently, Actors with at least one active behaviour or with a non-empty message queue are also included in the system roots.

Such Actors are called *active*.

Conceptually, an Actor can be considered garbage if its absence from the system cannot be detected by external observation, apart from its consumption of memory and processor resources. Kafaru *et al.* (Kafaru, Washabaugh and Nelson 1990) have given the definition of liveness of objects in the field of Actor garbage collection, that has become a standard: An Actor may be defined as garbage if it lacks either one (or both) of the properties below:

Computable the Actor is active or can become active hereafter.

Reachable the Actor can send information to, or receive information from, a root.

Garbage collection in such systems concentrates in finding efficient techniques for determining the liveness of objects following the above definition of the system roots. These kinds of system are not addressed in this thesis.

1.2.2 RPC-based Systems

One possibility in a programming language to support distributed computing is to provide a distributed heap with parts at different processes; each individual object resides at a single process, but it can refer to objects at other processes. They communicate by Remote Procedure Call (RPC). RPC (Birrel and Nelson 1984) is a basic communication mechanism that forms the basis for the client-server model (Jul et al. 1988, Linington 1992, Sousa et al. 1993, Vinoski 1993, Birrel et al. 1993).

Mutator processes perform local computations independently of other mutators in the system, although they may periodically exchange messages and allocate objects in local heaps. These mutator messages transfer data, which may include references to objects. The mutator sending the message is referred to as the *sender*, the mutator receiving it, the *receiver*. The object to which a reference in the message points may be on yet another process, usually called the *owner*. On receipt of the message, the receiver's mutator may store the reference in a local object, thus creating a new inter-process reference. In some systems, mutator messages may also transfer objects from one process to another; this is called *migration*.

The rôle of garbage collection in such systems is usually divided into:

Local garbage collection is performed in individual processes. It regards inter-process references as roots for garbage collection, in addition to the local roots. It is responsible for detecting and deallocating local garbage. Further, depending on the exact scheme employed, the local garbage collectors may be required to store extra information and do extra work to assist the distributed garbage collection.

Distributed garbage collection is a protocol to exchange information between local garbage collectors. It is responsible for detecting distributed garbage and make it be recognised as garbage by the local collectors, as well as protecting objects reachable from a remote root against local collection.

Garbage collection in such systems mainly addresses problems of inter-process communication, global synchronisation, scalability and fault-tolerance, while achieving safety and completeness. These systems are the main target of this thesis.

1.2.3 Object-Oriented Database Management Systems

Object-Oriented Database Management Systems (OODBMS) provide persistent storage of objects with complex inter-relationships (Ozsu et al. 1994). They support atomic transactions (Tanenbaum 1992), a mechanism that allows client applications to group a set of reads and writes to objects as an atomic unit.

In a client-server system, objects reside in a stable heap on secondary storage. Application clients navigate by starting at some *persistent root* object and may access the objects in the heap through a memory cache. Persistence is determined by reachability from the persistent root. A client fetches objects from the server, and keeps them in a local cache. It works upon these objects by copying data between objects, removing data from objects and creating new objects, in its *private space*. In other words, objects are gathered from their servers and the transaction works upon them at the client.

Servers keep a *log* where read, new and modified objects are written. The log is maintained in secondary storage or in main memory, but replicated, in order to allow recovery after a crash. When the transaction commits, modifications are installed into the stable heap.

The address space of such systems is maintained in the stable heap, usually called a persistent object store. Garbage collection in such systems is usually implemented by a server-based garbage collector. This is because object-oriented database technology takes the view that data resides mostly on secondary storage, with main memory being used as a temporary scratch buffer.

Garbage collection in such systems supports *persistence by reachability*. The rôle of garbage collection is to reclaim storage allocated to objects that are useless because they are not reachable from the persistent root or any application variables. The roots for the local collection at the persistent store include its persistent root, application roots and references from other persistent stores, if they exist. To allow concurrency, the roots also include the modified versions and new objects in the log that are yet to be installed.

Garbage collection in such systems mainly addresses problems of concurrency, recovery and disk traffic. It borrows some ideas from garbage collection on RPC-based systems. We discuss an adaptation of the solution presented in this thesis for such systems.

1.2.4 Distributed Shared Memory

The concept of distributed shared memory (DSM) provides a shared memory abstraction for a physically distributed memory architecture. The simple abstraction provided to the application programmer by the DSM model has made it the focus of recent study and implementation efforts (Nitzberg and Lo 1991).

DSM systems maintain the illusion of a distributed shared memory by synchronising data access and moving objects between processes when required, transparently to applications. The address space is distributed amongst the processes. Processes either have *no*, *read* or *write* access to data. Conceptually, data can be replicated on multiple processes to increase data locality, reducing access times. Each process can access any memory location in the shared address space at any time and read or write values altered by any other process. Objects' replicas are kept consistent by a *consistency protocol* (Tanenbaum 1992).

This model can be extended to distributed applications with persistent objects (Ferreira 1996), providing the illusion of a shared address space across the network, including secondary storage. This model offers transparent distribution and persistence. Applications have uniform access to any object in the system independently of its location. The model hides both the distinction between local and remote data, and the distinction between short-term and long-term storage. Applications navigate through the shared store by following pointers in virtual memory. The system moves the necessary data between main and secondary storage or between the main memory of remote sites, according to application needs.

Garbage collection in such systems also supports *persistence by reachability*. By traversing the objects graph starting from *persistent roots*, the collector is able to distinguish live objects from garbage objects which can then be safely collected.

The most interesting problem for garbage collection is consistence interference. Garbage collection algorithms must not compete with applications for holding consistent object replicas. Such competition would interfere with application's consistency needs. For example, if the collector on some process requires access to a consistent object, that would prevent an application from writing into another replica of that same object at the same time (Ferreira 1996).

We find in the garbage collection literature both server-based and client-based garbage collectors for persistent object stores. The former are used by object-oriented database technology (section 1.2.3). The latter are used by distributed shared memory technology extended with persistent objects where applications need high performance data manipulation in main memory.

Garbage collection in such systems mainly addresses problems of scalability, efficiency, disk traffic and consistence interference. It also borrows some ideas from garbage collection on RPC-based systems. We believe that these systems also may benefit from the ideas presented in this thesis.

1.3 RPC-based System Model

The main goal of this description of the RPC-based model is to establish the environment in which the cyclic distributed garbage collection algorithm executes (later, in chapter 8 we will describe the implementation of this model in the Network Objects system).

Our cyclic distributed garbage collector is presented for a classical distributed system, that is, RPC communication, no shared memory, partial failures, and unreliable and costly messages.

Process Model

Each process has an independent object space. It may contain any number of threads. It performs local computations independently of other processes in the system.

Processes may fail. Processes are fail-stop, that is, they will either deliver the correct result or no result at all. Processes recover from crashes eventually, but objects are lost in crashes.

Network Model

Communication between different processes occurs via message passing. Communications channels are potentially unreliable. Consequently, messages may be lost, duplicated or arrive out of order.

Processes may be disconnected temporarily because of a network failure. Process crashes cannot be differentiated from long term communication failures.

Memory Model

We assume a large scale object space distributed amongst a set of processes in a distributed system. Each address space supports a large number of objects. An object may contain any number of references to other objects. The implementation of a reference is not considered for now (in section 8 we will describe the implementation in the Network Objects system).

We distinguish between a *local* reference (to an object known to be in the same process) and a *remote* one (to an object thought to be in another process).

Mutator Model

Mutators modify the pointer graph: they create objects, and assign and delete references. Reference assignments modify objects' reachability. Any distributed garbage collection algorithm must detect the objects which are not remote referenced from any other processor. For this, every remote pointer operation must be considered:

1. Creation of an o-reference

   A process P where an object o resides, the *owner*, transmittes an o-reference to another process Q. Process Q has now a remote reference to o.

2. Transmission of an o-reference

   A process Q, which already has an o-reference to an object on another process P, transmittes the o-reference to a third process R. This operation differs from creation because the owner of the object (P) is not involved. So, it does not necessarily know that a new o-reference was created. Process R has now a remote reference to o.

3. Deletion of an o-reference

   A process q, holding an o-reference to an object located on a remote process P, discards it.

1.4 Distributed Garbage Collection Goals

In this section we describe the issues in designing a garbage collector for large persistent and/or distributed address spaces. We have identified efficiency, concurrency, fault-tolerance and scalability as the main issues, apart from safety and completeness, for large address spaces. Persistence introduces other issues like low I/O traffic, recovery and clustering, but they are not considered in this thesis.

Safety

Only garbage should be reclaimed.

Completeness

All objects that are garbage at the start of a garbage collection cycle should be reclaimed eventually. In particular, it should be possible to reclaim distributed cycles of garbage.

Concurrency

Distributed garbage collection should not require the suspension of mutator or local collector processes. Concurrency allows the collector to work in small mutator pauses making it possible for several processes to change the connectivity of the graph simultaneously in an autonomous way. However, inconsistencies in the object graph may be introduced. This leads to the need for some form of synchronisation between mutator and collector actions in order to avoid live objects being missed by the collector (safety), and in order to allow progress of the collector (liveness).

In a distributed environment, this problem is more serious given the asynchrony of distributed systems. Consider for instance the following example. Process A holds the last reference to object x, sends a copy of it to process B, then deletes this reference. Suppose B collects before receiving the reference to x, and A collects after having removed it. Then it would appear that x is unreachable although a reference to it is in transit.

Concurrency may interfere with garbage collection algorithm termination: consecutive changes of the object graph may delay termination of the collector in a large address space system. This is true for algorithms that need to visit every object in the system.

Another issue concerning concurrency is the possible existence of multiple collectors. In this situation, global synchronisation between the different collectors may be required.

Fault-tolerance

In a distributed system partial failures occur. Partial failures include crashes of individual nodes and failures in message delivery. Crashes are fail-stop, therefore the only consequence of a crash is temporary disconnection, loss of volatile memory, and halting of computation. Messages can be delayed, lost, duplicated, and delivered out of order, or there might be a network failure, in which a group of nodes becomes virtually disconnected from the rest. The memory management system should be robust, *i.e.* work efficiently and be safe, in spite of message delay, loss or duplication, or process failure. It should also prevent the dangling references that are caused by failures.

Failures in message delivery can be dealt with by using a generic reliable message protocol, but this is a costly solution that often requires multiple round trips per reliable message. The goal then is to design a garbage collector where messages are idempotent (so that duplicated messages are harmless), and non-essential (so that the loss of a message does not violate correctness, and is expected to be taken care of by later messages or on demand).

A robust garbage collection scheme must cope with unavailable nodes of the system:

- Wherever possible, garbage should be reclaimed despite the unavailability of parts of the system, without interaction with the crashed nodes.

- The garbage collection algorithm must adapt its behaviour to the situation, exhibiting graceful degradation of service, in order to guarantee safety and liveness: processes need to disregard references from processes that have failed, since they would otherwise be unable to collect garbage objects that were 'referenced' by such processes; if a process has not communicated for a long time and does not respond to repeated query messages, other processes assume that it has failed. Failure detection based solely on time-outs is inconsistent because (for instance) a transient overload could cause some processor be considered terminated, whereas it continues to execute. This introduces the problem of dangling references that may interfere with safety: a process may hold references to deallocated objects, and these references must be prevented from corrupting objects in the referenced processes. Consistent failure detection is however harder to achieve (Ricciardi and Birman 1993).

Failures and their recovery must be handled efficiently and should scale. Additional overheads due to fault-tolerance must be limited and mainly incurred when failures are present.

Persistent servers are however expected to recover from failures (see Recovery below).

Scalability

Distributed garbage collection algorithms should scale to networks of many processors without incurring non-linear cost overheads due to computation and synchronisation (Ferreira and Shapiro 1996). This requires that the chosen mechanisms have minimum dependence on limited resources that do not grow as the system gets bigger. In particular, the collection of the whole graph in a single phase is clearly not scalable.

Efficiency

There are two main issues concerning efficiency:

Promptness Collector efficiency: garbage should be reclaimed promptly. Having the collector run concurrently with the user application may not resolve the efficiency problem: it disperses a long garbage collection pause into shorter ones, but it does not reduce the total work to be done before garbage can be collected. If a garbage collection takes too long, the garbage collection may effectively fail as the system may run out of storage. If too much garbage accumulates, and must be paged to disk, the system may slow down even more.

Correctness of concurrent solutions often requires costly synchronisation methods. Moreover, global synchronisation between multiple garbage collectors may also be required in distributed environments, contributing to the system overhead. In particular, a consistent view of the object graph is very expensive (Babaoglu and Marzullo 1993)

The garbage collector may need to manage specific data to guarantee safety or fault-tolerance aspects. This may be specially significant for efficiency when some extra data, like log records, needs to be stored on stable storage.

User Application Overhead performance overhead due to garbage collection must be minimised. Two factors may slow down applications: the extra load experienced by the system and the potential synchronisation between mutators and garbage collectors.

Low I/O traffic

As we have already pointed out persistent object stores may consist of a huge object graph. Algorithms which frequently analyse most or all of the objects in the system are not feasible. Disk I/O is costly: only a small part of the heap may be cached in main memory. Garbage collection has to minimise disk traffic.

Recovery

In persistent object stores the effects of committed transactions survive crashes. In this case the safety requirement that an object remain in existence as long as it is accessible must be satisfied even if the only node that holds a reference to that object is down or unavailable. This requirement is needed since the heap is stable and once the crashed node recovers it should contain valid references in order to avoid objects being corrupted.

Some of the information that supports garbage collection must survive crashes too, while the rest can be recomputed on recovery. Updates of the garbage collection information may therefore incur stable-storage writes in addition to those required for durability of transactions (Tanenbaum 1992). The challenge is to reduce the amount of garbage collection information that must be kept stable, without incurring a long wait on recovery to re-establish the remaining information.

Also, the possibility of recovering implies that it no longer means that when an objects becomes garbage, it remains garbage.

Clustering

Clustering — the action of putting together related objects — is important. The performance of persistent object stores is often dominated by disk access. Clustering data that is likely to be accessed together is a major consideration in such systems. Garbage collection must take clustering (rather than merely compaction) into account (Franklin et al. 1989).

1.5 Cycles of Garbage

Cycles on uniprocessor systems are common, both at the application level and at the system level (Jones 1996). Cycles are typically created by programmers when they use back-pointers or they aim to express domain-specific problems in a natural manner.

When using a system that does not provide cyclic distributed collection, programmers must either modify their style, or break cycles explicitly by deleting pointers. However, it is not always apparent which pointer should be cut. Manual intervention is both burdensome and unsafe. We know of no good large-scale methodology of avoiding cycles.

We believe that cycles also occur in large shared address spaces (Godard 1994). In distributed systems for example, in client-server systems, objects that communicate with each other remotely are likely to hold references to each other, and often this communication is bidirectional (Wilson 1996). Many object-based systems are long running (persistent stores), so floating garbage is particularly undesirable as even small amounts of uncollected garbage may accumulate over time to cause significant memory loss (Maheshwari and Liskov 1995). Since cyclic distributed garbage collection is not widely available, there are few applications that make full use of distributed garbage collection. As in uniprocessor systems, programmers tend to either modify their style, or break cycles explicitly by deleting pointers. As we have already shown, this is even more complex in distributed systems

In distributed systems cycles may also be formed as a consequence of replication (Louboutin and Cahill 1995). Hypertext documents often form large, complex cycles (Maheshwari and Liskov 1997a). Recently, programming models for mobile computing applications seem to be a potential source for distributed cycles, as they allow arbitrary transmission and copies of data graphs that preserve sharing and circularities (Bharat and Cardelli 1995).

Some solutions for distributed garbage collection trade off completeness, that is, the ability to collect *all* garbage in a system, including distributed cycles of garbage, for weaker inter-process synchronisation constraints and a higher degree of concurrency under the assumption that distributed cycles are rare (Bevan 1987, Watson and Watson 1987, Shapiro et al. 1992, Ferreira 1996, Birrel et al. 1993). Such acyclic techniques only work if cycles are rare enough to be neglected. This approach may be acceptable if servers are short-lived, if sufficient memory is available to support the storage leaks and any additional paging cost due to memory fragmentation is bearable.

We do not make any assumption about topology of the overall distributed object graph, and more specifically about the rarity of distributed cycles. That is, we do not ignore them. However, we assume that local and acyclic distributed garbage are formed more frequently, hence they should be given the higher priority for reclamation.

1.6 Outline of the Thesis

The contribution of this thesis lies in the design of a distributed garbage collection algorithm that accounts for the collection of distributed garbage cycles.

Our goal is an expedient and complete, scalable, efficient and fault-tolerant cyclic distributed garbage collector for large address spaces (Rodrigues and Jones 1996, Rodrigues and Jones 1998).

As we argue, compromises inevitably must be made between these goals. For example, scalability, fault-tolerance and efficiency may only be achievable at the expense of completeness, and concurrency introduces synchronisation overheads.

We propose a garbage collection scheme that collects cycles on RPC-based systems without compromising the efficient reclamation of local and distributed garbage, it requires little synchronisation with applications and avoids global synchronisation. Additionally, it provides a technique that can be adapted to some solutions for garbage collection in persistent stores that are usually found in Object Oriented Databases Management systems and Distributed Shared Memory systems.

In chapter 2 we describe uniprocessor garbage collection. This description will help the reader to understand why the simple extension of these techniques to distributed environments does not match our goals.

In chapter 3 we survey the main techniques for partitioned collection. We will focus on techniques for RPC-based distributed systems as our work targets these systems. We will also present extensions for Object Oriented Databases Management systems and Distributed Shared Memory systems.

In chapter 4 we introduce our basic cyclic scheme. We do not account for concurrency, scalability, completeness or fault-tolerance. In chapter 5 and chapter 6, we describe the advanced features of our algorithm such as scalability, completeness and concurrency. In chapter 7 we present a proof of correctness of several aspects of our algorithm.

In chapter 8 we describe the implementation of our system over the Network Objects system.

Finally, in chapter 9 we conclude and discuss how we have met our primary goals, and present some ideas for future work.

Chapter 2

Classical Uniprocessor Algorithms

In this chapter, we briefly overview classical uniprocessor garbage collection techniques since most distributed garbage collectors are built upon them. For a more complete description of such techniques, readers are recommended to refer to (Wilson 1992, Jones 1996).

In section 2.1 we define the basic principles of garbage collection and introduce some terminology. Then we describe the three classical techniques for garbage collection: reference counting (section 2.2), mark-sweep and copying collection (section 2.3). These techniques are first described in the *stop-the-world* mode, that is, they suspend all user computation during garbage collection. This latency is sometimes unbearable for real-time or interactive applications which have strong responsiveness requirements.

We discuss, in section 2.4, advanced uniprocessor garbage collection techniques that decrease user program pause times: incremental and generational garbage collection. Section 2.4.1 introduces incremental garbage collection techniques, which allow the cost of garbage collection to be spread incrementally throughout the computation. Section 2.4.2 overviews generational garbage collection, a paradigm that has proved effective at reducing garbage collection pause times by segregating objects into regions according to their age, and concentrating garbage collection effort in a single collection cycle on just one region of the heap.

Finally, section 2.4.3 addresses garbage collection issues specific to environments in

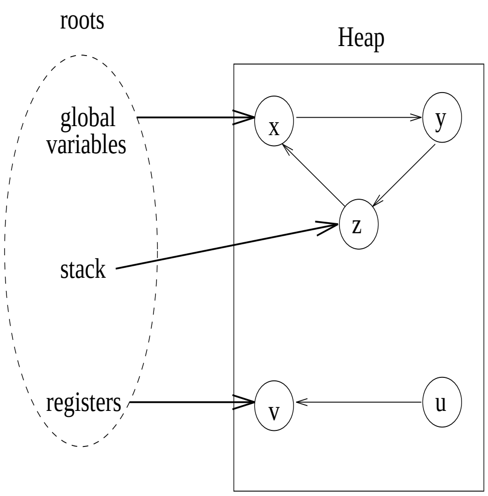


Figure 1: Heap and Roots.

which there is no support from the language compiler.

2.1 Principles

2.1.1 Live and Garbage Objects

Most high-level programming languages are able to allocate storage in a dedicated area called the *heap*. An individually allocated piece of data in the heap will be called an *object*. An application dynamically creates, in any order a number, of objects in the heap. An object embodies a mixture of regular data and *internal pointers* or *references* to other objects. The whole set of objects allocated in the heap forms a directed (potentially cyclic) graph whose nodes are the objects and whose arcs are references to heap objects. Each object in the graph may be referenced by a number of parents and may refer to a number of descendants

The values that an application can directly manipulate are those held in processor registers, the application's stack and global variables (static area). Such locations that hold references to objects in the heap form the set of *roots* of the computation. Other objects are reachable indirectly by following chains of internal pointers. The roots and heap are shown in figure 1.

By definition an object reachable from a root is *live*, that is, an object in the heap is live if its address is held in a root, or there is a pointer to it from another live heap object. More formally, $\rightarrow$ is defined as the 'refers-to' relation (Jones 1996): for any object or root M and any heap object N, $M \to N$ if and only if M holds a reference to N. The set of live objects in the heap is the *transitive closure* of the set of the roots under this relation, *i.e.* the least set[1] *live* where:

$$live = \{N \in Objects \mid (\exists r \in Roots.r \to N) \vee (\exists M \in live.M \to N)\}$$

When an object is no longer referenced from other reachable objects it becomes *unreachable* and cannot become *reachable* again[2], at least for well-behaved programs. It is called *garbage*. Since a garbage object remains garbage forever, it should be reclaimed in order to reuse the corresponding memory for further allocation.

Consider figure 1. Objects x, z and v are live since they are directly reachable from a root. Object y is also live since it is referenced by the live object x. Object u is garbage since it is neither reachable from any root and nor referenced from a live object. But note that if this were an actor system u might be live.

2.1.2 Garbage Collection

Manual reclamation of dynamically managed storage is often unsatisfactory. The alternative is to still allow the programmer to request dynamically allocated storage to be reserved but no longer ask him/her to determine when that memory is no longer required: it is recycled automatically. Garbage collection is the automatic reclamation of dynamically heap-allocated storage after its last use by a program.

The garbage collection literature distinguishes the *mutator* and *collector* rôles (Dijkstra, Lamport, Martin, Scholten and Steffens 1978). The mutator encompasses all application activities. Its sole rôle is to change or mutate the connectivity of the graph of active data structures in the heap. The collector detects and reclaims garbage objects.

Conceptually, garbage collection operates in two distinct phases. *Garbage detection* tries to distinguish the set of garbage objects from the set of live objects, whereas *garbage reclamation* disposes of memory occupied by objects previously detected as garbage. In practice, garbage detection and garbage reclamation can be interleaved temporally and the garbage reclamation technique is usually strongly coupled to the garbage detection

[1] Mathematical note: such a least set exists by Tarski's theorem, which says that any equation of the form $S = fS$, where f is a monotonic operation on sets, has a least fixed point.

[2] This is not true on persistent stores in the presence of recovery.

technique: an object's liveness may be determined either directly or indirectly. Direct methods require that a record be associated with each object in the heap, recording all references to that object from other heap objects or roots. The most common direct method is *reference counting*. It stores a count of the number of references to an object, its *reference count*, in the object itself. In its simple form, these records must be kept up to date as the mutator alters the connectivity of the graph in the heap. When the record reaches zero, the object is immediately made available for recycling.

Indirect or *tracing* collectors typically determine the set of live objects whenever a request by the mutator for more memory fails. They actually detect garbage objects by infering that they are not members of the set of live objects. The collector starts from the roots and, by following pointers, visits all reachable objects. These objects are considered to be live and all memory occupied by other objects is made available for recycling in a second phase.

It is difficult to compare different garbage collection algorithms either in principle or in practice. While formulae for algorithmic complexity can be determined, their constants and implementation details often have substantial impact on actual performance (Jones 1996). We do not deeply address this problem in this thesis. We aim at making a simple description of the three classical methods of storage reclamation: reference counting, mark-and-sweep and copying. As the techniques and ideas behind these algorithms form the basis of many more complex schemes, including distributed garbage collection schemes, it is important to understand how they work, and their strengths and weaknesses.

2.1.3 Safety and Liveness properties

There are two goals that we have to take into account when choosing a garbage collection algorithm: it must reclaim every garbage object as soon as possible without corrupting the integrity of references. The *liveness* property guarantees that all garbage is eventually reclaimed, and the *safety* property ensures that only garbage objects are reclaimed.

Garbage collection should be *comprehensive*: garbage should not be allowed to float unreclaimed in the heap. However collectors vary in their approach to comprehensiveness collection with different efficiency tradeoffs: most collectors based on reference counting cannot reclaim garbage cycles. A system that uses a tracing garbage collector delays garbage detection to the next collection: garbage collection introduces a *latency* between the moment an object becomes garbage and the moment it is eventually reclaimed. Generational collectors (as explained in section 2.4.2) and other partitioned collectors (as explained in chapter 3) concentrate their efforts in a single collection cycle on just one partition of the heap, rather than collecting the entire heap.

2.2 Reference Counting

Algorithms based on reference counting have been adopted for many languages and applications (for example, early versions of the Smalltalk object-oriented language (Goldberg and Robson 1983) and Modula-2+ (DeTreville 1990)). It is also the method used by the operating system Unix to determine whether a file may be deleted from the file-store.

The basic idea of the reference counting algorithm is to count the number of references to each object from other live objects (Collins 1960). Each object has an additional field, the *reference count*, denoting the number of references to it. When a new object is created, a single reference points to it, and its reference count is set to one. Each time a reference is duplicated the object's reference count is increased by one. When a reference to an object is deleted, its counter is decreased by one. Therefore, the reference counting algorithm preserves the invariant that the value of an object's reference count is always equal to the number of references to it.

When a reference count drops to zero, the reference counting invariant implies that there are no remaining references to the corresponding object. This means that the object is no longer required by the mutator and it can be safely reclaimed. For instance, in figure 2-(i) object u's reference counter is equal to zero. Therefore u is unreachable and u can be reclaimed. Upon reclamation of object u, v's reference counter is decremented by one, from two to one.

One advantage of this algorithm is that it is simple to understand and straightforward to implement. It is also a naturally incremental technique. Garbage detection

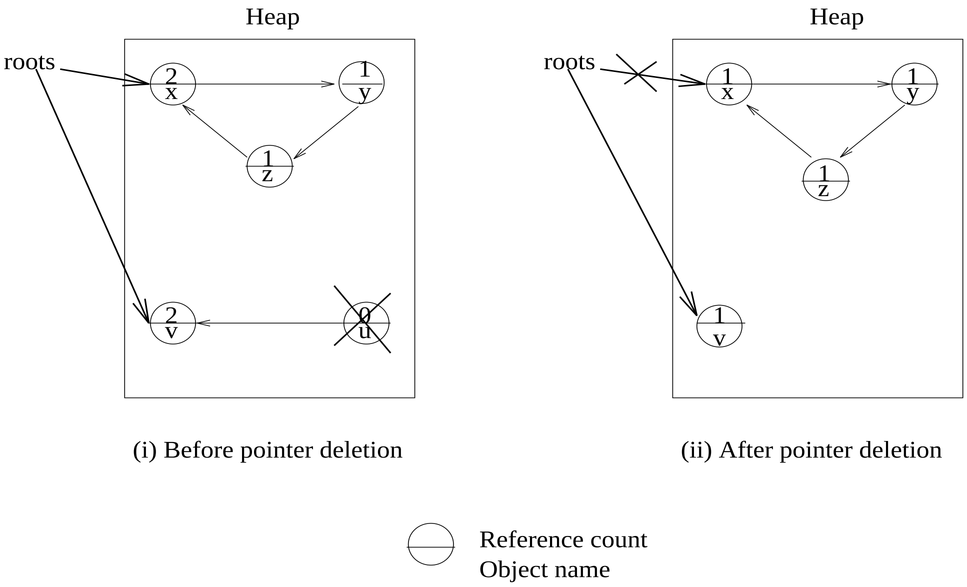


Figure 2: Reference Counting Algorithm.

and reclamation are performed concurrently with the mutator, distributing the memory management overheads throughout the computation. This contrasts with (non-incremental) tracing schemes in which the mutator is suspended while the algorithm runs. If the mutator has strong responsiveness requirements, short pauses may be important. Nevertheless, there are circumstances, for the simple algorithm described above, in which counter updating can suspend the mutator for a long while: the cost of deleting the last pointer to a sub-graph depends on its size. If any of the descendant's counters happen to drop to zero then their own descendants must also be recursively decremented. This is likely to occur with long data structures such as linked lists. Therefore, the deletion of a single pointer may result in a large amount of reclamation activity. Weizenbaum proposed a method to ameliorate the recursive freeing: Weizenbaum's lazy freeing (Weizenbaum 1963). Pointers in any reclaimed object with reference counter equal to zero are only deleted when that object's memory is again allocated. This lazy method is as efficient as the original method — the same instructions are used, but have moved from deletion to allocation of an object — but the algorithm is not so vulnerable to delays caused by cascades of object releases.

Another advantage of reference counting is its good locality of reference. It offers a good temporal locality of reference because an object is immediately reclaimed as soon it becomes garbage, that is, as soon as its reference count drops to zero. It also offers a good spatial locality of reference because a garbage objects is detected and reclaimed — as soon its reference count drops to zero — without access to objects in other pages of the heap.

However, the reference counting algorithm suffers from a number of disadvantages. First is the high processing cost paid to update reference counts. The cost is proportional to the amount of work done by the mutator because reference counts must be updated whenever references are assigned or deleted. This extra code imposes a severe overhead to the mutator. It also restricts the portability of the garbage collector since garbage collector efficiency is usually achieved by compiler support of the extra code. This overhead may be reduced by taking every safe opportunity to not adjust reference counts. This problem is addressed by variants of this algorithm such as Deferred Reference Counting (Deutsch and Bobrow 1976). Furthermore, it may exhibit poor locality of reference in the sense that an old target object must have is reference count decremented. Also, it may impose extra work on activities as simple as traversing a list, because it may require the list cells to be written on disk (in system with virtual memory), to update their reference counts, even if their value were not altered.

Another problem relates to the extra space in each object to store the reference count and reference count overflow. In the worst case this field should be large enough to hold the total number of pointers contained in the heap. Since there are usually only a small number of references between objects, a small number of bits could be used. Some authors have even suggested restricting the reference count field to a single bit. One-bit reference counting concentrates reclamation efforts on the unshared objects that typically make up the majority of the heap (Friedman and Wise 1977, Wise 1993). However, some "popular" objects may be referenced by many different objects. This can be handled safely by leaving the counter 'stuck' at its maximum value: it cannot be reduced since the true count of pointers to the object may be greater than its reference count. Hence, overflows of reference counts results in increasing conservatism.

Reference counting algorithms work badly with concurrency as each reference count must be protected by a lock. This is a substantial disadvantage.

Finally, the major problem of simple reference counting algorithms is their inability to reclaim cycles of unreachable objects. This algorithm is not complete. This problem appears because each object in a garbage cycle is referenced (at least) from its predecessor in the cycle. Therefore, each object in a cycle has a count of at least one even if there are no more references to any of these objects outside the cycle. Figure 2-(ii) illustrates a garbage cycle composed of objects x, y and z after deletion of two root pointers.

Consequently, reference counting is effective only if the mutator cannot create cyclic data structures. As we have shown in section 1, albeit in the context of distributed garbage collection, garbage cycle reclamation is an important requirement for many systems. Several authors have suggested combining reference counting with other garbage collection algorithms that handle cyclic data structures (Weizenbaum 1969). These solutions consist of using reference counting until the heap has exhausted. At this point a global garbage collector would be invoked in order to reclaim cyclic data structures and restore reference counts in the case that small reference counts are used.

However, some effort has been invested on solving the problem of reclaiming garbage cycles without using global garbage collection. Some of this work is specific to functional programming languages (Friedman and Wise 1979) or it relies on information from the programmer (Bobrow 1980). This work suggested that all objects should be assigned to groups by the programmer and that these groups rather than individual objects should be referenced count. In this way, intra- but not inter-group cycles could be reclaimed. David Brownbridge and others investigated the possibility of distinguishing cycle-closing pointers from other pointers (Brownbridge 1985). However, these proposals are either incorrect or inefficient in the general case.

Other proposals are generally applicable like the work by (Christopher 1984) and (Lins 1990). These algorithms are hybrid collectors. Most cells are freed by reference counting but garbage cycles are reclaimed by a mark-and-sweep collector. The idea behind these algorithms is to determine dynamically which data structures are only referenced by those data structures' internal pointers. Lins' algorithm picks an object that may be member of a cycle and performs a local mark-and-sweep on the object's transitive closure. In a first phase it removes reference counts that are due to pointers internal to the sub-graph. Any non-zero reference counts in the traced subgraph can only be due to external references and are considered live.

The technique developed by Lins has showed to be promising in the context of distributed garbage collection because it exhibits some locality: in the best case, only cyclic garbage is traced. In chapter 3 we describe the adaptation of some of the techniques cited above and other techniques for distributed garbage collection.

2.3 Tracing

Tracing techniques use the reachability property to distinguish live from garbage objects. There are two basic types of tracing algorithms: *Mark-and-Sweep* and *Copy*. Tracing algorithms occasionally traverse the reference graph, from the *roots*, to determine which objects are reachable. An object is live if it can be reached from a root by following pointers. Each object encountered during the traversal is marked as live and the remaining unmarked objects are considered as garbage.

Section 2.3.1 describes the Mark-and-sweep collector, a tracing technique which happens in two distinct phases.

Section 2.3.2 introduces the Copying collector, a different tracing technique which merges the garbage detection and garbage reclamation phases.

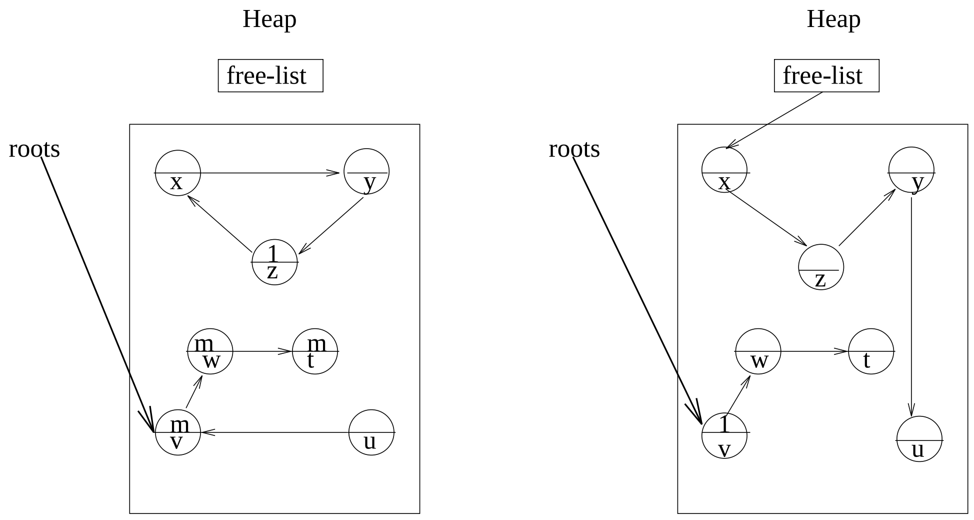
2.3.1 The mark-and-sweep collector

Under this scheme, objects are not reclaimed immediately they become garbage, but remain unreachable and undetected until all available storage is exhausted. A mark-and-sweep algorithm has two phases (McCarthy 1960). The first phase, known as marking, identifies all reachable objects. The second phase, the sweep, reclaims all unmarked objects.

The marking phase traverses all objects reachable from roots and marks them by setting, for example, a bit in each object visited. This phase ends when there are no more reachable but unmarked objects. Termination is enforced by not traversing from objects that have already been marked.

During the sweep phase, the memory is swept to find all unmarked objects, and typically to insert their memory in the *free-list*. Marked objects are unmarked in order to make them ready for the next collection.

Mark-and-sweep collectors have some advantages over reference counting. The most



(i) mark phase (ii) sweep phase

Figure 3: Mark-and-sweep Algorithm.

important of these is that no special action needs to be taken to reclaim garbage cycles (this also is true for copying collectors). They also have much lower overheads on the user program than reference counting: the overall elapsed time of a tracing system will be better.

The interface between the user program and a non-incremental tracing garbage collector is also much simpler than that of reference counting system. Under the later, care must be taken to ensure that reference count invariants are maintained. The simplicity of interface of tracing collectors makes them easier to maintain.

On the other hand, the simple version of the mark-and-sweep collector is a *stop-the-world* collector: computation is halted while the garbage collector runs. The pauses caused by this algorithm may be substantial. We introduce some methods of reducing pause times in section 2.4.1 and 2.4.2, when we discuss generational and incremental techniques respectively.

The simple mark-and-sweep algorithm presented above also tends to fragment memory, as does reference counting, scattering objects across the heap. In a virtual memory system such fragmentation may lead to loss of locality between associated objects of a data structure and result in excessive swapping of pages to and from secondary storage. In a real memory system some benefits of caching can be lost. Fragmentation makes allocation more difficult as suitable spaces must be found in the heap to store new objects.

This problem can be ameliorated using a two-level allocator such as that used by the Boehm-Weiser collector (Boehm and Weiser 1988). An additional compaction phase can be also performed at the expense of a significant overhead to the garbage collector. We refer the reader to (Jones 1996) for a description of several styles of compaction.

The complexity of a mark-and-sweep collection is usually measured as being proportional to the size of the entire heap rather than to the volume of surviving data because the sweep phase must examine the whole heap. Analysis must also consider the algorithm's virtual memory and cache behaviour. More sophisticated implementations of the mark-and-sweep algorithm reduce the cost of sweep phase and improve the virtual memory behaviour of both phases.

Some implementations store mark-bits in a separate bitmap table rather than placed them in the objects that they mark, for instance (Boehm and Weiser 1988). Mark bits have several advantages for the virtual memory system. If the bitmap is comparatively small, it can be held in RAM so that reading or writing mark-bits will not incur page faults. Furthermore, no heap object need be written to during the marking phase. Page faults will only be incurred by the garbage collector when pointers need to be traced. Also, in the sweep phase live objects do not need to be accessed at all, although garbage objects may have to be linked into a free-list.

The efficiency of non-incremental garbage collection may be improved if the sweep phase is done in parallel with mutator execution. This is possible because the mutator cannot interfere with the collector's sweep phase since the mark-bits of live objects are invisible to the user program. We refer the reader to (Jones 1996) for a description of techniques of *lazy sweep*.

A different problem with mark-and-sweep is that it requires a strong synchronisation between phases. That is, generally, the mark and sweep phases cannot be interleaved since all reachable objects must be marked before starting the sweep phase (this synchronisation is relaxed in (Queinnec, Beaudoing and Queille 1989)). This contrasts with reference counting where, as we have already said, the two phases are interleaved and an object is immediately reclaimed as soon as its reference count drops to zero — temporal locality of reference. This feature makes reference counting more attractive for distributed systems since its communications are local to the objects involved in an

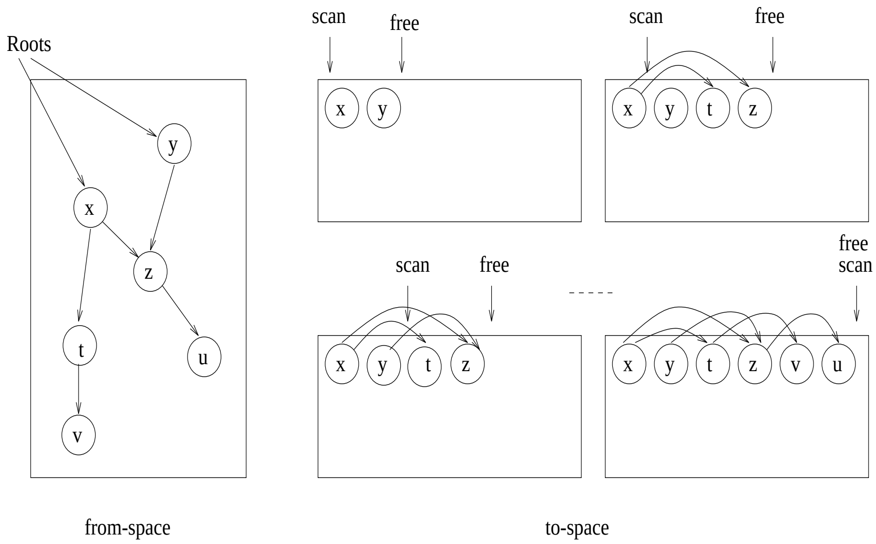


Figure 4: Cheney's Algorithm.

update.

2.3.2 The copying collector

The second class of tracing algorithm is that of copying collectors (Minsky 1963, Cheney 1970). Copying collectors merge garbage detection with garbage reclamation in a single phase. This kind of algorithm divides the heap into two disjoint *semi-spaces* called *from-space* and *to-space*. During normal mutator execution objects are allocated in from-space. When there is not sufficient space to meet allocation requests, the algorithm copies reachable objects to to-space. Unreachable objects are left in from-space. Once the copying is completed, the roles of the two spaces are reversed. This transition is called the *flip*.

Cheney's algorithm (Cheney 1970) is a well known technique and it is usually presented as the simplest form of copying reachable objects from from-space to to-space. Its major advantage is that it is iterative, hence elegantly avoiding recursive call costs, stack space overhead and stack overflow.

The copying collection is done iteratively using two pointers: a *scan* pointer and a *free* pointer (see figure 4). Objects immediately reachable from the roots are copied to to-space. Free points now to the first free address in to-space, and scan points to the first object in to-space. The object pointed by scan is scanned for references into from-space. Each object reached is copied to to-space. In addition, the references in the scanned object are updated to refer to the new copy, and a forwarding pointer (pointing to to-space) is left in the object's old location. The free pointer is then advanced and the scan continues to the next object. Eventually the scan pointer reaches the free pointer. This means that all the objects that have been copied have also been scanned for descendents and that the algorithm is finished.

The complexity of copying is proportional to the size of the active data structure rather than the size of the heap. This makes copying particularly attractive if the surviving data is a small proportion of the total heap. This is typical of many functional and object-oriented programming languages. However, this measure of complexity is too simplistic. The constants in the complexity formula are also important (Jones 1996). For instance, the cost of copying an object is likely to be more expensive than simply testing and setting a mark-bit, particularly if the objects is large. Although mark-and-sweep must sweep the entire heap, in practice its real cost is dominated by the mark phase. Furthermore, lazy sweep techniques and bitmaps (see section 2.3.1) can reduce significantly the cost of the sweep phase.

Copying collectors have the nice effect of compacting the heap since objects are copied contiguously in to-space. This reduces heap fragmentation — by compacting live objects into the bottom of to-space — and improves allocation costs — new memory is allocated simply by incrementing the free space pointer. Compacting the active part of the heap onto fewer pages should reduce the size of the program's working set, that is the locality of reference of the user program. Consequently, it may perform better than mark-and-sweep with the virtual memory system, although reorganising data in the heap may be undesirable in some environments. Unless care is taken with this regrouping, the spatial locality of the resulting structures may be poor. We refer the reader to (Jones 1996) for a description of regrouping strategies.

There is another issue of spatial locality, the locality characteristics of the garbage collector itself. An immediate cost of copying garbage collection is the use of two semi-spaces: the address space required is doubled, compared to mark-and-sweep collectors. A copying garbage collector will touch every page in to-space and from-space in each collection cycle. Consequently it may suffer more page faults than mark-and-sweep for a fixed size of heap, as it uses twice as many pages.

Although copying garbage collection has predominated in the past — its advantages of compaction, cheap allocation, low complexity and easier incorporation into incremental and generational systems gave it the advantage over mark-and-sweep garbage collection — recent studies suggest that the choice between mark-and-sweep and copying collectors may depend as much on the behaviour of the client program as on the inherent properties of the garbage collection algorithm.

2.4 Advanced Techniques

2.4.1 Incremental Garbage Collection

The aim of incremental garbage collection is to avoid the pauses incurred by stop-the-world garbage collectors. In such collectors small units of garbage collection are interleaved with small units of mutator execution. Each garbage collection pause time is smaller than in the stop-the world garbage collection.

The mutator and collector can also run concurrently. The usefulness of this mode is that collection adds no pauses on top of time-slicing. In the rest of this section, and when the difference is not relevant, we will use the term incremental to designate both incremental and concurrent collectors. Several algorithms were originally designed for multi-processors but are easily adapted for serial machines.

The simplest of incremental techniques is reference counting, which is naturally incremental for all operations except the deletion of the last pointer to a sub-graph (see section 2.2). However, it is expensive, it is closely coupled to the user program and it is unable to reclaim garbage cycles. These drawbacks discourage its use. It is therefore desirable to make tracing techniques incremental.

There are two potential conflicts between the mutator operations and the collector. First, care must be taken to ensure that the collector makes sufficient progress to prevent the user program from running out of memory before the collection cycle is complete — mutator starvation. Several policies have been used to balance processing between collector and mutator in way that avoid such mutator starvation. For instance, Baker (Baker 1978) tunes the rate of collection to the rate of consumption of memory. The idea is that a small amount of marking or copying can be done at each allocation. Others, for instance (Appel, Ellis and Li 1988) avoid the problem by triggering garbage collection whenever the amount of free memory falls below a certain *threshold*, avoiding mutator starvation.

Second, the main issue of incremental tracing is how to ensure the correct execution of the collector when it competes, asynchronously, with the mutator for the same data. This introduces a consistency problem: while the collector is tracing the graph of reachable objects, the graph may change while the collector "isn't looking". This may lead the collector to failing to find (i) all garbage objects in a garbage collection cycle and (ii) all reachable objects and conclude wrongly that some live objects are garbage.

Concerning situation (i), consider an object o reachable from the root. The mark phase reaches the object and marks it live. Afterwards the mutator discards the pointer from the root to the object o. Since the object has been already marked live, it will not be collected by the sweep phase. The reclamation of the object is only postponed until the next garbage collection. Such unreclaimed garbage objects are called *floating garbage*.

Situation (ii) may occur when the mutator concurrently to the collector detaches a reachable object o from a non-traversed part of the graph and attaches it to an already traversed part of the graph. In this way, the mutator may hide object o from the marking process. At the end of the marking phase, object o is not marked as live and therefore will be reclaimed, unsafely, by the sweep phase.

To avoid situation (ii) some synchronisation is needed between mutator and collector to indicate that the connectivity of the graph has changed. It is not necessary for the mutator and the collector to share an identical view of the graph. The consistency requirement can be relaxed to allow the collector to work with a conservative approximation of the graph of live objects (Wilson 1992). If the mutator changes the graph of reachable objects, garbage objects may or may not be reclaimed at the end of the garbage collection cycle depending on whether or not they have already been marked live by the garbage collector. As consistency requirements are relaxed, the collector's view of the graph becomes more conservative, and more floating garbage accumulates.

Before going into more details concerning synchronisation, it is useful to see how incremental garbage collection can be described by the abstract *tricolour marking* algorithm (Dijkstra et al. 1978).

Dijkstra's algorithm required the mutator to communicate with the collector by colouring objects black, grey or white.

Black indicates that an object and its immediate descendents have been reached by the collector. The garbage collector has finished with a black object and need not visit it again. At the end of garbage collection all live objects are black.

Grey indicates that an object has been reached by the collector, but its immediate descendents may not have been, or its connectivity to the rest of the graph has been altered by the mutator behind the collector's back. Once a grey object has been scanned its descendents are coloured grey and it becomes black.

White indicates that an object has not yet been visited by the garbage collector and may be garbage at the end of the tracing phase.

A garbage collection cycle terminates when all reachable objects are coloured black, and hence when there are no grey objects left. Any objects left white at this point are garbage and can be reclaimed.

Intuitively, the traversal proceeds in a wavefront of grey objects, which separates the white objects from the black objects that have been passed by the wave — that is, there are no pointers directly from black objects to white ones. The importance of this invariant is that the collector must be able to assume that it is "finished with" black objects, and can continue to traverse grey objects (Wilson 1992). If the mutator creates a pointer from a black object to a white one, it must somehow notify the collector that its assumption has been violated. Therefore, the collector must be capable of keeping track of graph changes resulting from mutator activity, and re-trace parts of the graph adequately. This ensures that the collector is aware of every significant change concerning the pointer graph.

Figure 5 demonstrates this need for synchronisation. Suppose that object x has been completely scanned (and therefore blackened); its descendents (y and z) have been reached and greyed. Now, suppose that the mutator copies the pointer from y to u into x, copies the pointer from x to z into y and deletes the pointer from y to u. The only pointer to u is now in object x, which has already been scanned by the collector. This violates the invariant we have stated: black object x pointing to white object u. If the tracing continues without any synchronisation, y will be blackened, z will be reached again and u will never be reached at all, and hence will be unsafely reclaimed.

We describe below the two basic approaches to synchronising the collector with

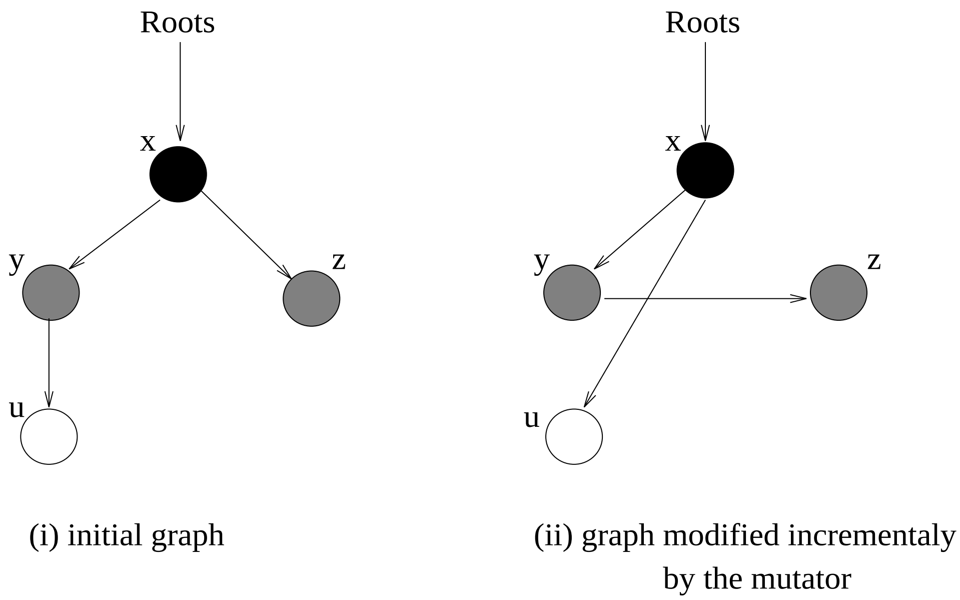


Figure 5: Concurrent Mutator Activity.

the mutator: *read barrier* and *write-barrier*. Different approaches lead to different incremental algorithms that may be judged through several parameters. The degree of conservatism is an important parameter because floating garbage fragments the heap, increasing the effective residency of the program. The pauses incurred on mutator activity are a second parameter. Incremental collection should delay computation only briefly at each step. Pause time depends on how much work is done by the synchronisation action — the *barrier*. Incremental collectors may also contain uninterruptible sections: processing the root set and checking for termination of a garbage collection cycle. If pauses are too great, the incremental nature of the algorithm is compromised. The time and space costs of the barrier also depend on its selectivity and frequency, and how it is implemented.

Write-barrier

Whenever the mutator attempts to write a pointer into a black object, the write-barrier trap or records the write so that the object can be visited or revisited by the collector. In order to falsely reclaim a live object, a white object must become invisible to the collector but still be reachable by the mutator. For this to happen, both of the following two conditions must hold at some point during the marking phase:

1. *A pointer to the white object is written into a black object.* If this condition does not hold, there will not be any black-white pointer during the marking phase. In this case, there must be a path to each reachable white object from a (black) root that passes through a grey object. The marking phase will eventually reach the white object from the grey one.

2. *The original reference to the white object is destroyed.* If a pointer to a live white objects is written into a black object during the marking phase and the original reference to that white object is not lost, the white object will still be reached by the marking phase through that original reference unless this pointer is destroyed.

Write-barrier methods are classified as either *snapshot-at-the-beginning* or *incremental-update*. When a pointer to a white object is written into a black object, snapshot-at-the-beginning collectors prevent the loss of the original reference to the white object (because it might have already been copied into black objects), while incremental-update collectors catch the change to the connectivity of the graph (Wilson 1992). Snapshot-at-the-begining algorithms are more conservative than incremental-update ones. No objects that become garbage in one garbage collection cycle can be reclaimed in that cycle. Actually, they allow the tricolour invariant to be broken, temporarily, during incremental tracing. Rather than preventing the creation of pointers from black objects to white ones, they ensure that the original path to the object is not lost, because all overwritten pointer values are saved and traversed.

There are many incremental collectors that use a write-barrier. For example, (Steele 1975, Dijkstra et al. 1978, Boehm, Demers and Shenker 1991) use incremental-update write-barriers, and (Yuasa 1990) uses a snapshot-at-the-begining barrier.

Read-barrier

A read-barrier ensures that the mutator never sees a white object: whenever a mutator attempts to access a white object, the object is immediately visited by the collector and coloured grey or black; since the mutator cannot read pointers to white objects, it cannot write them into black objects.

Software read barriers are generally considered to be too expensive. Read barriers may also be implemented with support from the operating system's virtual memory protection mechanisms to trap access to protected pages. Appel-Ellis-Li's collector (Appel et al. 1988) uses this last approach. Zorn's measurements suggest that its performance may be inferior to software methods, although different architectures and operating systems vary considerably (Zorn 1990).

The expense of read-barriers means that they are rarely used with non-moving collectors. They are instead used with copying garbage collectors to trap mutator accesses to *to-space*. The best well known collectors using a read-barrier are (Baker 1978, Appel et al. 1988, Nettles, O'Toole, Pierce and Haines 1992).

2.4.2 Generational Garbage Collection

Generational collectors use the ages of objects to optimise the collection of younger, smaller partitions. They aim at reducing the garbage collection pause time by decreasing the amount of memory that has to be collected. They take advantage of the following empirical observations (Lieberman and Hewitt 1983, Ungar 1984):

1. Newly created objects have a higher chance of becoming garbage than those that have already survived many collections.

2. There are more references from new objects to older objects than the other way round. Older objects may refer to newer ones only if they have been updated. Mutations are comparatively infrequent in many systems.

Objects are segregated into *generations* based on how long they have survived. We talk of just two generations, *old* and *new*, but the scheme can be extended to any number of generations. Since the new generation is where most garbage is created, it is collected more frequently. Objects that survive a certain number of collections are moved to a less-frequently collected partition.

In order for this scheme to work, it must be possible to collect the younger generation(s) without collecting the old one(s). The collector must be capable of finding pointers into the young generation(s). This requires the use of a write-barrier similar to the one found in incremental collection (see section 2.4.1) to keep track of such inter-generational pointers. Each potential pointer write in the heap must be accompanied by some extra bookkeeping in case an inter-generational pointer is being created. The important point is that all references from old to younger generations must be located at collection time, and used as roots for the collection. If the above assumptions hold, there would be few such references.

Generational collectors only keep track of pointers from the old generation to the younger generation (the converse would be expensive as there are typically many more references from new to old than from old to new generations (Wilson 1992)). Consequently, when the old generation is collected, the new and old generations are traced together, starting from their roots. This also contrasts with other partitioned schemes, where any partition may be collected at any time.

2.4.3 Conservative Garbage Collection

Tracing algorithms need to traverse the reference graph. For this purpose, the garbage collector must be able to find references inside any object, in registers, the stack, the heap or any other memory area. In other words it must distinguish pointer from non-pointer data. This co-operation is usually difficult to implement in *unco-operative* environments or programming languages such as C (Kernighan and Ritchie 1990) or C++ (Ellis and Stroustrup 1990), which do not provide the necessary runtime type information.

A possible solution, for these cases, consists of either a pre-processor (Edelson 1992) or the compiler (Samples 1992) statically generating type information. Normally this is accomplished by maintaining tags. For example, pointers might be constrained to have a fixed bit pattern in the low-order bit positions. This kind of solution typically slows down some operations on integers, and brings a performance penalty for application programs that rarely or never make use of garbage collection.

Another solution receives no help at all from the compiler and assumes that anything that might be a pointer is a potential pointer unless it can be proved otherwise. It is called *conservative* garbage collection.

The Boehm-Weiser collector (Boehm and Weiser 1988) is a conservative collector with no reliance on co-operation from the compiler, and that has no knowledge of the stack, registers or heap object layout. This approach relies on the use of a mark-and-sweep collector. In order to determine accessibility, it treats any data directly accessible to the program as a potential pointer. The allocator ensures that given such a data value, it is possible to determine whether it points to a valid object or not. If so, it is assumed that the data value in fact was a pointer, and that the object it points to is accessible. Similarly, it treats any data inside the objects as potential pointers, to be followed if they, in turn, point to valid data objects.

The *Mostly Copying Garbage Collector* (Bartlett 1988, Bartlett 1989) is a conservative garbage collector that still assumes no knowledge of stack or registers layouts, but it does assume that all pointers in the heap can be found accurately. The collector is a hybrid conservative and copying collector. The algorithm divides all accessible objects in the heap in two classes: those which might be referenced from the stack or registers (the root set), and those which are not. The former objects are treated conservatively and are left in place, and the later objects are copied into a compact area of memory.

The main disadvantage of such collectors is the risk of misidentifying data as heap pointers (*e.g*, considering an integer as a pointer), thereby leading to the consideration of garbage objects as being reachable. This implies that memory is retained, which could otherwise be recycled — a *space leak*.

2.5 Summary

In this chapter we briefly surveyed uniprocessor garbage collection techniques.

There are two fundamental garbage collection strategies: reference counting and tracing. There are two tracing collectors: mark-and-sweep and copy. *Stop-the-world* collections are not suitable for real-time or interactive applications as they suspend all user computation during garbage collection.

Reference counting algorithms are inherently incremental and are scalable. However, they do not collect cycles of garbage. Consequently, tracing collectors had to be made incremental. We described two techniques for synchronisation between mutator and collectors in order to provide safe garbage collection: a *read-barrier* and a *write-barrier*.

Another technique that decreases mutator pause times is generational garbage collection. The heap is divided in several generations. Young generations are collected more frequently as young objects tend to become garbage more rapidly than objects that survive several collections.

Finally, we described a technique for garbage collecting unco-operative environments, which do not provide runtime type information. Such collectors all are called conservative collectors as they may misidentify data as heap pointers, thereby leading the collector to consider garbage objects as being reachable.

Chapter 3

Distributed Garbage Collection Techniques

In this chapter we review the most relevant solutions for distributed garbage collection. We will emphasise distributed garbage collection solutions that collect cycles of garbage, showing the extent to which they meet the goals we have introduced in section 1. We also describe extensions of these techniques for persistent stores like Object-Oriented Database Management Systems and Distributed Cached Stores.

The key to achieving an expedient, scalable and fault-tolerance distributed garbage collection is to preserve the *property of locality*. The first step is to use a partitioned model, dividing a large address space into several partitions that can be collected somewhat independently. We describe this model in section 3.1. Reclamation of garbage is typically done by a local collector that traces a single partition independently. Partitions track remote references by storing all incoming references in an *entry-table*, and all outgoing references in an *exit-table*. This is an abstract model; different systems may use different implementations of this model.

Unlike non-partitioned collectors (we describe an example in section 3.2 — *global tracing*), partitioned collectors collect local (to the partition) garbage without any cooperation of the rest of the distributed system. The local collection makes a conservative assumption that references in the entry-table are reachable, and counts them as roots for the local collection.

Broadly speaking, techniques for distributed garbage collection fall under the same two paradigms as uniprocessor techniques: tracing and reference counting (recall section 2). In the following sections we survey a number of distributed garbage collection techniques that fall into one of this paradigms and show how they preserve the property of locality.

In order to analyse these techniques we have recast the terminology used by their authors into the one we present in section 3.1. This helps to compare between algorithms and allows better understanding of their fundamental contributions.

3.1 Partitioned *vs* Non-partitioned Collection

A first step in the direction of the distributed garbage collection goals of expediency, scalability and fault-tolerance is to preserve the following property:

Property of Locality *The collection of garbage should not require the co-operation of any process other than those containing the garbage.*

This property is desirable for scalability and fault-tolerance in distributed systems. An algorithm that preserves such a property is scalable because it does not need the co-operation of every process in the system. Consequently, it does not require any protocol that demands the co-operation of all the distributed system. Fault-tolerance is achieved because progress may be made even if some processes of the system are down. The collection of distributed garbage can only be delayed by those processes containing the garbage.

The key to preserving such a property is to sub-divide the address space into separate areas and collect each area independently. This idea was first proposed by Bishop (Bishop 1977) in the context of a virtual memory system, and is now generalised to describe all decentralised distributed garbage collectors. This concept is supported by the following advantages of partitioned approaches over non-partitioned ones:

Scalability The collector does not need to wait for the collection of the entire address space. Only a subset of a potentially huge set of objects needs to be considered at any point by a collector. Partitioned collection greatly improves the locality of reference of the collection algorithm. In RPC-based systems local computation is used and message passing avoided. In persistent stores I/O operations are reduced.

This makes the collection more efficient.

Promptness Separate collections give control over how frequently to traverse different parts of the object graph. Some parts of the graph may remain unchanged for a long time, so that traversing them repeatedly is a waste, while other parts may change rapidly, providing a rich source of garbage.

Mutator Overhead The disruption caused by interfering with the application during garbage collection is reduced.

On the other hand, partitioned schemes are not complete:

Completeness The liveness of objects reachable from outside the partition cannot be decided locally, hence these objects are conservatively considered as roots by the partition garbage collection.

In the remainder of this chapter we concentrate on solutions for collecting inter-partition garbage. We introduce a model for partitioned garbage collection. We describe a general scheme — a hybrid of tracing and reference tracking (following the terminology in (Maheshwari 1993)) — that is used to create arbitrary sized partitions that can be collected separately and concurrently.

3.1.1 Model for Partitioned Garbage Collection

The exact nature of partitions may vary, since different models of distributed object-based programming systems allow different implementations. On a RPC-based system, each site is a partition. On a persistent server, a set of logically related persistent objects is a partition. Finally, in a cached distributed persistent store, each unit of memory cached is a partition.

To correctly collect one partition without entirely scanning the others, information must be kept about object pointers that cross partition boundaries. We distinguish between intra-partition references (to an object known to be in the same partition) and inter-partition ones (to an object in another partition). For garbage collection purposes, an inter-partition reference is described by an *exit-item* in the source partition and an

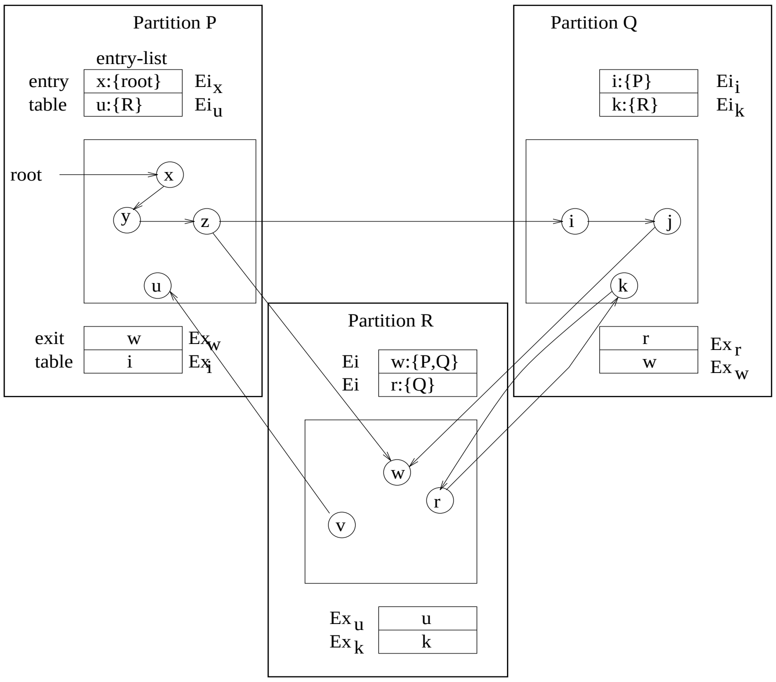


Figure 6: Partitioned Garbage Collection Model

entry-item in the target partition[1].

Entry-items are collected in a structure called the *entry-table* and exit-items are collected in a structure called the *exit-table*. They are represented in figure 6. Partitions are identified by capital letters P, Q,..., etc. Objects are identified by lower case letters y, z,..., etc. An exit-item for object z is noted by Ex_z and the corresponding entry-item by Ei_z.

Additionally, we group together all inter-partition references in a partition that point to the same object. We model this by having each exit-table store a single item for each outgoing inter-partition reference. The importance of this will become clear during the rest of this section.

The entry-table of a partition is maintained by co-operation between the local collector and an inter-partition reference tracking protocol.

[1]Generally, in RPC-based systems, an inter-partition reference is represented as a local reference to a structure called a *surrogate*, which in turn contains necessary remote information. This remote information may, in turn, point to the entry-item which points to the actual object, or point directly to the object.

In garbage collection schemes for persistent object stores, entry and exit-items are simply auxiliary data structures; they are not seen by application code.

We will use the entry-item/exit-item abstraction whenever we may abstract from system-dependent issues. We will explicitly describe such issues whenever it is significant for our scheme.

Intra-partition collection

Local garbage is collected independently in each partition. Usually a tracing algorithm is used because of the inability of the reference counting technique to reclaim circular garbage and efficiency considerations (recall section 2.2).

The root set used for local garbage collection consists of *local roots* — the local root set — that is, objects usually designated as roots (stack, registers and global variables), plus *global roots* — the global root set — that is, the entry-table: these are objects known outside this process; consequently, it is not possible to decide locally if they are garbage or not. As we have already said, each entry-item refers to an object that is alleged to be referenced from other partitions.

Once an inter-partition reference to some object is created, it is no longer possible to determine locally whether or not it is still reachable from a root. The local garbage collector must therefore conservatively consider it to be a global root. Until it is shown otherwise, all local objects and exit-items reachable from this root are considered to be live, as are any objects and exit-items reachable from local roots.

An intra-partition collection updates the exit-table: unreached exit-items are removed.

Inter-partition Collection

The information maintained by the inter-partition protocol constitutes a conservative snapshot of the actual object graph, built incrementally as the overall object graph evolves. The union of the local root set and global root set is a superset of the *actual root set*[2]. The actual root set of a given partition contains only the roots from which live objects, and only live objects of that partition, are reachable. The actual root set of a partition is the union of the local root set and the set of global roots that are live, that is, reachable from some root in the system. Only the inter-partition protocol can determine whether or not a given global root is still referenced from outside the partition.

The inter-partition reference tracking protocol is responsible for maintaining the entry and exit-tables, that is, how they are created and how reachability information is

[2]Following the terminology in (Louboutin and Cahill 1997).

propagated. When the mutator creates a new inter-partition reference, or deletes one, extra messages may or may not be required to update the target entry-table. Following the terminology in (Maheshwari 1993), we call the part of the inter-partition protocol that takes care of creation of references the *increase protocol*, and the one that takes care of deletion the *decrease protocol*. The increase protocol ensures the safety property that live objects will not be collected; it is therefore run by the mutator when creating a new inter-partition reference in such a way that prevents the premature reclamation of objects. As we are going to see next, this is an issue related to the global order of events in distributed systems.

On the other hand, the decrease protocol ensures the liveness property that garbage will ultimately be reclaimed; therefore, it can be delayed and executed in the background. If, in a partition, we represent every inter-partition reference to the same object as a single item in the exit-table, then the decrease protocol must only be performed when the last intra-partition reference is deleted.

Different schemes result in differences in the message passing protocol and fault-tolerance.

For the rest of this chapter we assume the mutator model described in sections 1.3, except for sections 3.6 and 3.7. For simplicity, we consider that one process corresponds to one partition. From now on, in this chapter, we will treat partition and process as synonymous.

3.1.2 Road-map to the Remainder of this Chapter

In the following sections we describe different distributed garbage collection techniques following the taxonomy shown in figure 7.

Section 3.2 discusses distributed algorithms based on tracing. We argue that the pure form of tracing is unacceptable for distributed systems as it is a non-partitioned algorithm. We also describe an intermediate tracing solution that preserves the property of locality for local garbage (section 3.3).

Section 3.4 describes a number of reference tracking techniques for inter-partition garbage collection, and compares them with respect to message passing and fault tolerance.

(Louboutin 1997) suggested that these schemes are called *reactive* schemes as they

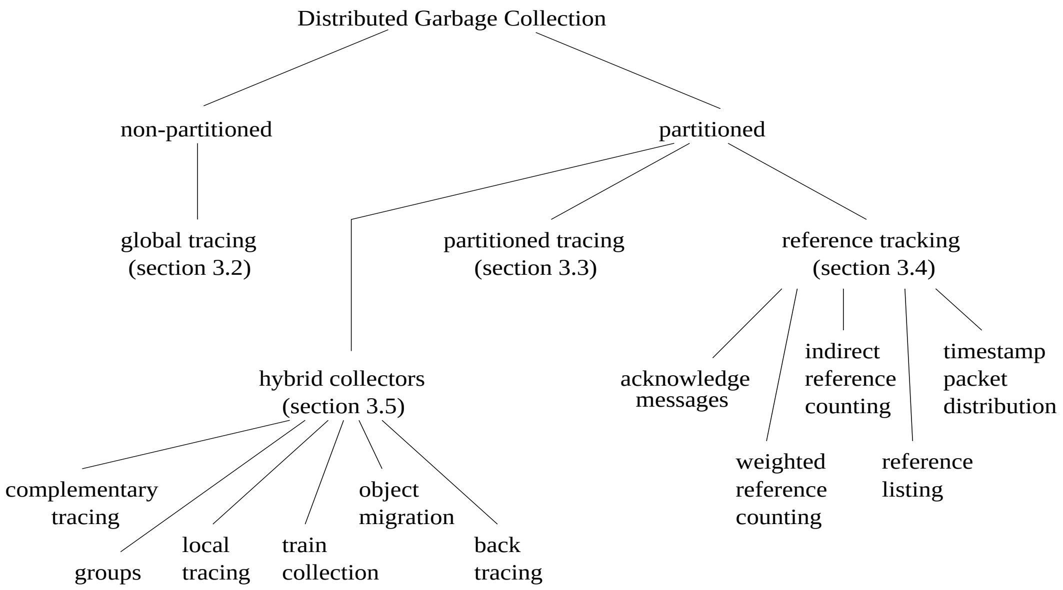


Figure 7: Partitioned *vs* Non-partitioned Distributed Garbage Collection

"react" directly to some events of the mutator. These approaches make it possible to identify a garbage object from information received from objects directly adjacent to it in the global object graph. Collection can therefore proceed without need of co-operation from all sites in the system. They preserve the property of locality.

The most typical reference tracking based algorithms are those based on variations of uniprocessor reference counting (Lermen and Maurer 1986, Bevan 1987, Watson and Watson 1987, Piquer 1991, Dickman 1992) or reference listing (Birrel et al. 1994, Shapiro et al. 1992, Plainfossé and Shapiro 1992, Maheshwari and Liskov 1994, Ferreira and Shapiro 1996) that are not intrinsically complete, that is, that are not able to collect cycles of garbage. We also describe a new approach for reference tracking that is intrinsically complete (Louboutin and Cahill 1997, Louboutin 1997).

Finally, we address hybrid approaches for distributed cyclic garbage collection. Section 3.5 describes a variety of schemes that collect inter-partition cycles of garbage. These schemes have the common feature of having been designed for the partitioned model. They usually combine a form of reference tracking with a second approach in order to collect cycles of garbage. We classify several approaches in this category depending to what extent they preserve the property of locality for cyclic garbage.

3.2 Global Tracing

Global distributed tracing algorithms have the ability to collect garbage cycles. The majority of the tracing collectors known are of the mark-and-sweep type (Hudak and Keller 1982, Augusteijn 1987, Derbyshire 1990, Juul and Jul 1992).

Distributed mark-sweep algorithms visit all globally reachable objects. In the mark-phase each process marks the objects reachable from its local roots. Each object is scanned and for each remote reference found a *marking message* is sent to the target process. The process receiving a marking message marks the corresponding object and continues the marking phase. When every process has marked all the reachable objects it owns and there are no marking messages in transit, the sweep phase starts. The sweep phase may be done by each process independently from any other process.

Making a mark-and-sweep algorithm distributed adds the problem of global synchronisation. The main problem is to synchronise the distributed mark-phase with independent sweep phases. During the mark-phase processes receive and send marking messages. The collector in each process can be resumed if it receives a marking message for an object it owns. Therefore, processes are alternatively contributing to global marking — active state — and waiting for a marking message — idle state. The active processes are those which are carrying out marking, and only they can send messages. An active process goes from active to idle when it finishes its marking. An idle process can only be reactivated on receipt of a message. When all the processors are idle, the global mark-phase is said to be terminated since no further marking is possible.

This problem is even more difficult if the global marking is concurrent with the mutator, since the mutator may turn an idle process to active as a result of the synchronisation with the collector (section 2.4.1) (Hudak and Keller 1982, Augusteijn 1987, Derbyshire 1990, Juul and Jul 1992).

Global synchronisation can be detected by any existing distributed termination detection protocol (Dijkstra and Scholten 1980, Mattern 1987, Mattern 1989, Tel and Mattern 1993). However, different distributed tracing algorithms have presented their own protocols, since the existing solutions may be either very expensive for the garbage collection problem (Augusteijn 1987) or unco-operative with synchronisation between mutator and collector.

Finally, distributed tracing algorithms require the cooperation of all processes in the network before it can reclaim any objects. This technique, is, thus, neither scalable nor fault-tolerant.

Below, we present a set of solutions that use the approach described above. These schemes have the common feature that garbage is only reclaimed with the co-operation of all processes in the system.

Hudak and Keller (Hudak and Keller 1982) present a concurrent distributed *marking-tree* collector based on the Dijkstra's concurrent mark-and-sweep collector 2.4.1. Marking (mark an object, scan it and mark its descendents) comprises spawning a mark-task. In addition, a marking-tree is simultaneously built to provide mechanisms for cooperation and proper termination. The algorithm terminates when there are no more grey objects in the system (*cf.* 2.4.1). Termination is detected by having each mark-task spawn an uptree-task, which propagates upward in the marking-tree. When the root receives an uptree-task for each mark-task it has spawned, the marking phase terminates. Each process can then proceed to the sweep-phase independently.

This work leads to a large space overhead in providing space for recording the marking-tree. It also halts the mutator for a long period when a remote reference to a potential white object is written into a black object as a mutator needs to execute (rather than just spawn) a mark-task on the white object. Thus the mutator must halt until the remote object has been marked and has spawned an uptree-task.

Augusteijn (Augusteijn 1987) also presents a concurrent distributed collector based on the Dijkstra's concurrent mark-and-sweep collector. The marking phase operates by colouring objects. When there are no more grey objects the phase has terminated (*cf.* section 2.4.1). When a remote object must be greyed, a request message is sent to the owner process. The main problem is the detection of a global state in which there are no grey objects. This global state is detected by a termination detection algorithm as follows.

Each process can be active-disquiet, passive-quiet or passive-disquiet. Initially, all processes are active-disquiet. A process will turn from an active-disquiet into a passive-quiet state when it has no more local grey objects and it has received an acknowledgement for each request message it has sent. Once passive, it remains so, but it changes its state from quiet to disquiet on receipt of a request message. This request message can only be sent by another disquiet process. This means that once all processes are quiet, they remain so in a stable state. This state is determined by having each process send to a synchroniser process a message informing it of the change from active to passive.

A process may also turn to a disquiet state when a remote mutator sends a request message to preserve the "no references from black objects to white ones". This may happen when a grey object sends a message to a black object with potential white descendents. If its descendents are not local, a request message must be sent, possibly disturbing quiescence. This is no problem, since the sender must be disquiet (it holds a grey object). The mutator sending the message needs to wait until the objects in remote nodes are shaded. This may disrupt the mutator.

The global distributed tracing algorithm presented in (Derbyshire 1990) uses a similar termination detection protocol.

3.3 Partitioned Tracing

In this scheme, the only information kept by the entry-table is which local objects are referenced by one or more remote references. Such objects are distinguished by the presence of an entry-item; the entry does not contain any other information. The entry-item is created when a process first sends a mutator message containing a reference to the object. When the reference is passed on to other processes, the entry-item is not affected. Thus, it cannot be removed without help from a global tracing (Hughes 1985, Juul and Jul 1992, Ladin and Liskov 1992).

The Emerald System

The Emerald garbage collection scheme (Juul and Jul 1992) consists of two sets of collectors, which are applied concurrently. The global scheme is achieved by concurrent mark-and-sweep collectors on each process, which cooperate as one global garbage collector across the entire network. This global collector tries to achieve completeness even though various parts of the distributed system may be temporarily unavailable. The local scheme consists of an independent partial local collection on each process that collects local garbage. These local collectors do a more expedient collection of local garbage without being complete. Note that, in this solution, global tracing is needed for collecting both acyclic and cyclic distributed garbage. This is because the inter-process collector is not reference counting-based.

Any site in the system may initiate a global garbage collection cycle. All messages in the system are identified with the the current cycle number, making it possible for the receiver to join the current cycle.

The algorithm is based on a concurrent variant of the mark-and-sweep collector. The mark-phase is done concurrently with the mutator using an *object-fault* mechanism similar to a page-fault mechanism in a virtual memory system (Appel et al. 1988). This mechanism implements a read-barrier — the mutator processes can only access black objects. All grey objects are protected (*cf.* section 2.4.1). The object protection mechanism ensures that whenever the mutator attempts to access a grey object, a fault occurs, causing the collector to mark and traverse, that is colour the object black, which entails shading all the objects reachable from it. Unlike Augusteijn's algorithm, this distributed version of the mark-and-sweep algorithm allows a black object to temporarily hold a reference to some remote object that has not yet actually been shaded at its own site. Each site maintains a set of non-resident grey objects that makes it possible to postpone the actual shading of these remote objects. The remote shading involves sending a message to the remote site. When the site at which the remote grey object is located acknowledges the message, the reference denoting the remote object may be removed from the non-resident grey set. A black object is prevented from invoking a remote white object because objects that are remotely invoked are implicitly marked and traversed before the invocation is actually performed.

Marking is complete when there are no longer any grey objects in the system. Every process in the system needs to cooperate to determine when the mark-phase has finished. For this purpose the algorithm uses a a two-phase commit protocol based on acknowledgement messages and pairwise availability. This protocol is robust to process and message failures, but depends on each process being aware of all other processes in the system, hence not being scalable.

As local and global garbage collection operate concurrently on the same objects, conflicts may arise. To prevent the two collectors from conflicting with each other, their activities become mutually exclusive on a given site.

This algorithm violates the desirable property of locality for distributed garbage: all processes in the system need to co-operate in order to collect distributed garbage.

Tracing With Timestamps

The algorithm described in (Hughes 1985) is similar to the basic distributed mark-and-sweep algorithm except that it uses timestamps instead of mark bits. The main idea of this algorithm is that the timestamp of a live object keeps increasing while the timestamp of a garbage object eventually stabilises. A global timestamp threshold is computed. When this threshold exceeds the timestamp of an object, that object is garbage and can be collected.

This algorithm performs many global garbage collections in parallel. Each processor makes a contribution to all currently active global garbage collections every time it performs a local garbage collection. Each process has a clock that is used to record the time when the garbage collection started locally — the *GC-time*. A local collection propagates the roots' timestamps to the exit-items. The local roots are timestamped with the *GC-time*, while entry-items retain the timestamps last put into them. When an entry-item is created, it is time-stamped with the local process' current timestamp. The local collection in a process is expected to timestamp an exit-item with the largest local timestamp of any root from which is reachable. To ensure this, references in the roots are selected for tracing in decreasing order of timestamp.

At the end of the local collection, exit-items' timestamps are sent to the corresponding entry-items on the target processes in *timestamping* messages (similar to the marking messages of the basic distributed mark-and-sweep algorithm). When a process receives a *timestamping* message, it updates the timestamps of the corresponding entry-items to the maximum of their current value and that received in the message. When a process increases the timestamp of an entry-item, it records the fact that it has not propagated the increased timestamping. For this purpose, each process maintains a timestamp called *redo* whose value is equal to the greatest timestamp already locally propagated. Thus, when an entry-item's timestamp is increased, the *redo* is set to the entry-items's old timestamp if that is lower than its current value. When a process has processed the *timestamping* message, it sends back an acknowledgement to the sender. When the sender has received all the acknowledgements of all messages it has sent, it can update its own *redo* to the local *GC-time*, provided it did not receive *timestamping* messages from other processes itself.

It can be shown that an entry-item timestamped below the global minimum of the *redo*'s of all processes — the threshold — is garbage. However, it is tricky and costly to compute this threshold at any time. Such computation depends on any global termination algorithm (Dijkstra and Scholten 1980, Rana 1983, Mattern 1987, Mattern 1989, Tel and Mattern 1993).

This algorithm does not preserve the property of locality: it does require all processes to co-operate in order to collect distributed garbage. Additionally, if a process that crashes does not recover, the entire global tracing will eventually come to a halt, as the global minimum *redo*'s will be stuck at the crashed process's value.

Logically Centralised Tracing

Ladin and Liskov (Ladin and Liskov 1992) describe a variant of distributed tracing. The idea is to compute the global accessibility of objects on a single centralised service. This service is used to store information about the inter-process references. Each process performs asynchronous local collections and communicates periodically with the central service providing it with its inter-process references. From time to time, each process asks the central service about the reachability of its objects that are not locally reachable. The answer may indicate that an object is no longer remotely reachable, thus it can be deleted.

Since each process garbage collects asynchronously, the service never has a consistent view of the reachability of every object. Thus, the central service adopts a conservative approach that can be used safely. For this purpose, it uses a timestamp protocol involving loosely synchronised clocks at each process and a bounded delay for messages in transit. Messages containing references to objects also contain the time at which they are sent.

To reclaim cycles they also timestamp objects as Hughes. Each public object is timestamped with the latest time at which it was accessible from some process, and the algorithm is based on the premise that timestamps of accessible objects continue to increase, while those of garbage objects eventually become constant. These timestamps are also kept in the central service and the threshold is also computed in the central service (avoiding a termination detection protocol). In contrast with (Hughes 1985), only garbage objects in cycles will be identified and collected by this technique. Most objects will be found to be inaccessible earlier (without using timestamps).

In this service the processes do not have to communicate with each other for the purpose of garbage collection. The communication with the service can be performed in the background. The drawback, however, is that the server, albeit replicated, can become a bottleneck in a large system. Also, the processes have to transfer a fair amount of information to the server in order to have it detect all garbage.

Although this algorithm ameliorates some drawbacks of Tracing with Timestamps, it still requires the co-operation of all processes in the system in order to collect distributed garbage.

3.4 Reference Tracking

The reference counting algorithm is promising to distribute for the following reasons:

- It is performed in small steps interleaved with the mutator, allowing concurrency without the need of synchronisation and hence presenting lower communication costs.

- There is no need to scan global data structures.

Distributed reference counting is a simple extension to uniprocessor reference counting. Each entry-item stores a count of the number of exit-table items that point to it. Duplicating or deleting a reference to an object requires, as part of the increase and decrease protocol, *increment* and *decrement messages* to be sent to the owner of the object in order to increment or decrement, respectively, the corresponding entry-item's reference count.

Distributed reference counting, however, poses some problems: it is vulnerable to out-of-order delivery of reference counter manipulation messages, leading to premature reclamation of live objects — *race conditions*. There are two types of race conditions, both possibly leading to the unsafe reclamation of a reachable object. We call them *decrement/increment* and *increment/decrement* race conditions.

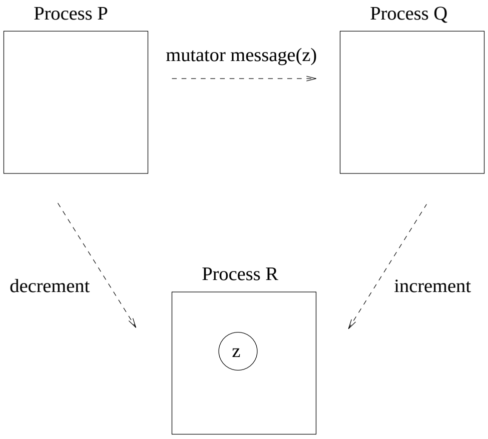


Figure 8: *Decrement/increment* race condition

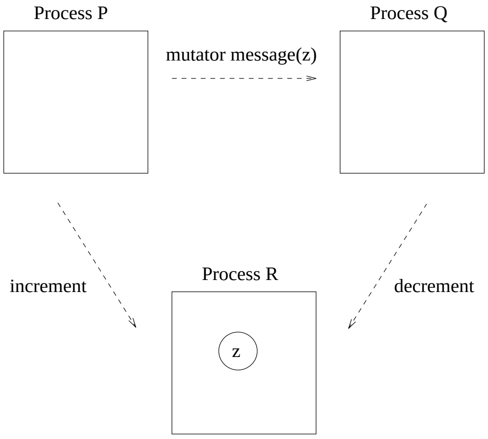


Figure 9: *Increment/decrement* race condition

Figure 8 illustrates the decrement/increment race condition. Suppose process P holds a reference to object z in process R, and sends a message to process Q containing a reference to z. Process Q receives this message and sends an increment message to R concerning object z; concurrently, process P deletes its reference to z and sends the corresponding decrement message to R. If the decrement message arrives first at R, then z is considered to be unreachable and is unsafely reclaimed.

On the other hand, suppose process P sends the increment message before sending the reference to z to Q. This also results in a potential race condition. Figure 9 illustrates the increment/decrement race condition. Suppose that Q receives the message and immediately discards it. Therefore it sends a decrement message to R. Once again, if the decrement message arrives first at R, z is unsafely reclaimed.

This scheme is also less fault-tolerant then the first. If the increment message to R succeeds, but the mutator message does not arrive to Q, the reference count of Ei_z would be erroneously incremented, compromising the liveness of the algorithm.

We model the transmission of a z-reference as an *incomplete transmission*.

Incomplete transmission The transmission of a z-reference is incomplete while the owner of z does not receive the corresponding *increment message*. A potential race condition may occur when a *decrement message* is sent during this period.

Race conditions can be resolved by imposing a global order on message delivery, which is expensive. Moreover, distributed reference counting is not resilient to message failures because increment and decrement messages are not idempotent and must neither be duplicated nor lost.

A number of variations of the standard distributed reference counting algorithm improve resilience to either race conditions, message failures or process failure. These variants can be grouped in the following categories: acknowledgement messages, weighted reference counting, indirect reference counting and reference listing.

3.4.1 Acknowledgement Messages

Acknowledgement messages suppress those potential race conditions described in the section above.

Lermer and Maurer (Lermen and Maurer 1986) describe a communication protocol based on acknowledgement messages that provides a correct distributed reference counting algorithm. This protocol assumes that point-to-point communication links can be modeled as infinite length FIFO queues (and hence do not allow loss or duplication of messages). The protocol is based on four kinds of messages: delete messages, acknowledge messages, copy messages and acknowledge request messages; and two attributes per object in a processor, giving the number of acknowledge references to that object and the number of 'incomplete' references to it. A copy of a reference to an object passed to a process P is complete when P receives a copy message (with the reference) and an acknowledgement message from the process holding the object (this message acknowledges the increment of the object's reference counter). This process sends this acknowledge message after receiving an acknowledge request message from the process doing the copy. A process P can only delete a reference to an object if that reference is an acknowledged reference. This algorithm eliminates race conditions, and allows a partial detection of lost messages, at the expense of a significant overhead in message traffic for each reference copied.

This algorithm is still not completely resilient to message failures — increment and decrement of reference counts is still a non-idempotent operation and messages cannot be lost or duplicated — nor, as with standard reference counting, to process failure. As references (counts) cannot be identified with processes, it is not possible to associate them with crashed processes, hence to be disregard.

Birrel *et. al.* (Birrel et al. 1994) describe a variant of the reference counting technique for reclaiming Network Objects (Birrel et al. 1993). We explain this technique in greater depth in section 3.4.4, but we introduce it here because it also uses acknowledgement messages to prevent race conditions.

References to Network Objects are created as a side-effect of marshaling references in remote invocations. The potential race condition between concurrent copy and deletion of the same reference is avoided by preventing the remote reference from being reclaimed in the sender process until this process receives an acknowledgement from the target process indicating that the operation has been completed, that is, the target process has already reported the creation of the new reference to the owner of the object. Following the terminology in (Ladin and Liskov 1992, Maheshwari 1993), the transmitted reference is kept in a *translist*. The references in the translist are seen as roots for the local collection, preventing the collection of the exit-item correspondent to the transmitted remote reference.

This extra acknowledgement is only needed when a reference is sent as a result of a remote method invocation. If it is sent as an argument, the method's return serves as the acknowledgement that the operation is completed. It seems that performance is not seriously affected by this extra message, because the wait for the acknowledgement is asynchronous with the mutator: for safety, the object is kept locally reachable until the acknowledgement arrives. However it still doubles the number of messages for the worst case.

3.4.2 Weighted Reference Counting

Bevan (Bevan 1987) and Watson (Watson and Watson 1987) independently proposed a new algorithm — Weighted Reference Counting — that eliminates increment messages, and hence the potential race conditions.

To each reference is associated a weight and to each object a standard reference counter. The algorithm should maintain the following invariant:

> The reference counter of an object is equal to the sum of the weights of the references to it.

When a remote reference is first created, a weight equal to the reference counter of the object is assigned to it. Each time a reference is duplicated to another process, its weight is halved and the remainder is sent to the target process. When the process receives it, it associates the weight with the new reference. Thus the sum of the weights is kept unchanged. The increment message is thus not necessary.

When a process deletes a remote reference, it sends a decrement message with the associated weight to the target process. When the process receives the message, it subtracts the received weight from the object reference counter. The object may be reclaimed, if it is not reachable locally, when its reference counter becomes zero.

This algorithm has extra space associated with each remote reference. Bevan (Bevan 1987) proposed using weights that are power of two, in order to store only the logarithm in the references. In addition, this algorithm has the following problem: the number of times a reference may be sent to another process is limited by its initial weight. This problem can be solved with the use of an extra indirection object. This solution may create remote indirections (if the indirection object is not created on the same processor holding the object pointed by the reference being duplicated) which are expensive, and also affect the access to the data.

Weighted Reference Counting avoids race conditions and improves communication overhead, but is not resilient to message loss or duplication, or process failures, in order to ensure safety and completeness.

Dickman (Dickman 1992) improves on Weighted Reference Counting in two aspects: resilience to message loss and indirection objects. He defines a new weaker invariant that is compatible with message loss. The new invariant states:

> The reference counter of an object is greater or equal to the sum of the weights of the references to it.

A lost or miss-ordered message does not violate this weakened invariant. In contrast, a duplicated decrement message remains problematic because it could make the sum of the weights of the references to an object greater than the object's reference counter.

This algorithm avoids the use of indirection objects when weights cannot be split, by using a special *null weight* value. In this case, the reference counter is always greater than the sum of the partial weights, thus preventing the object from being reclaimed at all. In this case, this algorithm generates memory leaks. The author assumes that some cyclic distributed tracing collector is used in conjunction with the Optimised Weighted Reference Counting, in order to reclaim such objects and cycles of garbage.

3.4.3 Indirection, and Strong-Weak Pointers

Piquer (Piquer 1991) suggests an algorithm that improves on Weighted Reference Counting by avoiding the indirection objects at the expense of some memory overhead. It also avoids the use of increment messages by maintaining a distributed reference counter for each remotely referenced object. Increments are performed locally, therefore without communication. This is achieved by always maintaining enough information on each process to do the increments locally.

The key observation is that the process that sends a reference already has an entry-item that protects the remote entry-item at the owner of the object. If the remote reference sent to the target process is made to protect the exit-item at the sender, the entry-item at the owner will be protected indirectly. The basis of the algorithm is to maintain a tree structure representing the diffusion tree of a reference throughout the system. For this purpose, remote references are extended with two fields: the identification of the process that sent the reference and a counter recording the number of times the reference was copied from the local process. The latter is incremented every time the reference is copied. This means that a new (remote) reference was created to the target object. A reference may be deleted if its reference counter is zero. When this happens the process sends a decrement message to the process from which the reference came. In its turn, this process decrements the counter of the reference it holds.

As for the other proposals based on reference counting, Piquer's proposal is not resilient to message or process failures. However, this technique may be used in conjunction with another variant of reference counting. The work on SSP chains (Shapiro et al. 1992, Plainfossé and Shapiro 1992) combines *reference listing* — a fault-tolerant variant of reference counting (see section 3.4.4) with indirection.

A further problem with the indirection method is that if the receiver accesses the reference, it is indirected through the sender. The work in (Shapiro et al. 1992, Plainfossé and Shapiro 1992) proposes the use of *weak* and *strong pointers*. An exit-item encapsulates two pointers: a strong and a weak one. The strong pointer indicates the next entry-item in the above diffusion tree. It is used only for garbage collection. The weak pointer short-cuts ahead of the strong pointer and allows direct access to the object. As an optimisation, the strong pointers can be short-cut in the background to point directly to the object after the owner has created a corresponding entry-item.

One drawback of using strong-weak pointers is that every reference included in mutator messages actually occupies the size of two references. As pointed out by (Maheshwari 1993), if the mutator message is carrying an object that contains references, the object would have been marshaled into a different format wherein the contained references are twice as big.

3.4.4 Reference Listing

Reference Listing (Shapiro et al. 1992, Plainfossé and Shapiro 1992, Birrel et al. 1994, Maheshwari and Liskov 1994, Ferreira and Shapiro 1996) differs from standard reference counting in the way the reference counter of a remotely referenced object is managed. Instead of maintaining a simple reference counter, recording the number of remote references for each object, the entry-item for an object keeps a *list* of the process identifiers that refer to the object. The following safety invariant is maintained in such systems:

> If process P refers to an object z in process Q, then P is in Ei_z's reference list.

Increment and decrement messages are replaced, respectively, by insert and delete control messages, which include the process' identity. To preserve the invariant, whenever a reference is copied, the process acquiring the new remote reference must be inserted in the target object's reference list. When a process P no longer refers to a remote object, the owner must remove it from the object's reference list. There are two ways of doing this. P may send a delete message to the owner (Birrel et al. 1994) or may periodically send the complete list of references that it holds for objects in the owner (Shapiro et al. 1992, Plainfossé and Shapiro 1992, Maheshwari and Liskov 1994). A failed deleted message needs to be remembered and retried, while a lost list is compensated for by the subsequent one.

Race conditions may be avoided by using acknowledgement messages (Birrel et al. 1994) (as explained in section 3.4.1) or using any other variation of reference counting that avoids the sending of synchronous insert messages; (Shapiro et al. 1992, Plainfossé and Shapiro 1992) uses indirection and weak-strong pointers (as explained in section 3.4.3).

Reference Listing has two important advantages over the standard reference counting variants: it improves resilience to message and process failures, albeit at the expense of some memory overhead. Insert and delete control messages are idempotent, in contrast with the increment or decrement messages in standard reference counting, and therefore can be retried on failure. However, a straggler delete message is potentially unsafe. Suppose that an insert message was sent after the delete message to re-create the entry-item. If the delete message arrives before the insert message, the entry-item may be deleted for good. One way to avoid this problem is to use timestamped insert and delete messages. A process stores the timestamp from the insert message in the entry-item. A delete message is effective only if it is timestamped higher than the entry-item (Shapiro et al. 1992, Plainfossé and Shapiro 1992, Birrel et al. 1994).

Tolerance of process failures relies on the ability of each process to compute the set of processes holding references to an object it owns, by looking through the object's reference list, so it can prompt one of those to proceed with any communication.

3.4.5 Timestamp Packet Distribution

The work presented in (Louboutin and Cahill 1997) is based on the work described in (Schelvis 1989) that is often overlooked in the literature because of its complexity, and message and space overhead. It describes a new approach to cyclic garbage collection that entails reconstructing the vector-times (Dependency Vector — DV) that characterise the causal history of relevant events of the mutator processes' computation. These events are those that result in modifications to the inter-process paths in the *global object graph*. The global root graph is formed by the global roots and the local roots of each individual process. It is shown that knowing the causal history of these events makes it possible to identify garbage objects that are not identifiable by means of per-process garbage collection alone.

The global graph's edge-destruction events' dependency vectors are constructed by propagating increasingly accurate approximations of these vectors along the paths of the global root graph.

This algorithm preserves the property of locality. Detection of garbage only requires the co-operation of those processes that contain the cycle. The underlying reference listing scheme is responsible for repeatedly circulating approximations of the dependency vector until the complete transitive closure, that is, the full vector-time of events responsible for the creation of all the paths to an entry-item, has been determined. Whenever an entry-item receives a dependency vector, a new approximation can be computed. If this newly computed dependency vector is the actual full vector-time and indicates that the entry-item is no longer reachable from an actual root (recall section 3.1), the entry-item is removed.

This algorithm is very complex, however, and, as pointed out by (Maheshwari and Liskov 1997a), its space overhead is large. It requires full reachability information between all entry-items and exit-items, and each entry-item Ei_i stores a set of vector timestamps; each vector corresponds to a path Ei_i is reachable from. At present we argue that some issues need more clarification, in particularl, the maintenance of the global root graph in the presence of mutator concurrency.

3.5 Hybrid Collectors

Distributed reference counting based algorithms cannot collect cycles of garbage spanning processes. Collecting interprocess cycles of garbage is, however, an important issue especially for long running distributed systems (*e.g.* distributed databases), where floating garbage is particularly undesirable as even small amounts of uncollected garbage may accumulate over time to cause significant memory loss (section 1.5).

Figure 10: Locality Spectrum

Several techniques have been proposed for cyclic distributed garbage collection. In this section we will provide a brief description of each, and discuss how they meet our primary goals of completeness, efficiency and fault-tolerance as we stated in section 1.

The degree of *locality* is the metric we have chosen for judging distributed garbage collection. For cyclic distributed collection, this is defined as follows (Maheshwari 1993):

Property of Locality *The collection of distributed cycles should not require the co-operation of any process other than those containing the cycle.*

For the same reasons described in section 3.1, we judge cyclic distributed garbage collection techniques on the extent to which they meet this property. We summarise a set of cyclic distributed garbage collection techniques found in the literature in figure 10 (the enumeration represents the section in which they are described). Complementary Tracing methods do not preserve the property of locality. As we move to the right, the techniques improve, in some way, the locality of the algorithms.

Some hybrid approaches are based on the choice of suspect objects: objects that may be garbage. Those objects are identified by heuristics. Heuristics are not accurate because, although are able to identify garbage objects as suspects, they may identify live objects as well. This fact may result in the performance of wasted work that directly influences factors like message passing, local computation overhead and space overhead. It is then necessary to trade locality against these factors.

We have identified two heuristics in the literature: *local reachability* (Bishop 1977, Shapiro et al. 1990, Gupta and Fuchs 1993, Rodriguez-Riviera 1995) and *distance heuristic* (Maheshwari and Liskov 1995).

Local reachability An objects is suspected of being garbage if it is not reachable locally.

This heuristic is weak if we consider, as is suggested by (Maheshwari and Liskov 1995, Gupta and Fuchs 1993), that in a long-lived distributed systems it is likely that objects are not locally reachable, but still live. (Gupta and Fuchs 1993) suggests that a suspect object must not have been used for some time period. But, again, in long-lived distributed systems it is likely that live objects may not be accessed for long periods.

Distance heuristic The *Distance Heuristic* (Maheshwari and Liskov 1995) is based on the "distance of objects" from a root. The distance of an object is the length of the shortest path from any root to the object, that is, the minimum number of remote references in any path from any root to the object.

Suspects are found by *estimating* distances. A distance field is associated with each entry-item. When a new process is added to an entry-item list, its distance is conservatively set to one. A root is modeled as an entry-item with zero distance. The local collector propagates distances from entry-items to exit-items. Changes in the distances of exit-items are sent to target processes, where they are reflected in the corresponding entry-items. The estimate of the the distance of a cyclic distributed garbage objects keeps increasing without bound; that of a live object does not.

The following theorem is defined: If all processes containing a cycle perform at least one local collection in a certain period of time, called a round, then n rounds after the cycle became garbage, the estimated distances of all objects in the cycle will be at least n.

Therefore, they select a *suspicion threshold* distance, D. Objects with estimated distances greater than D are highly likely to be garbage. Such objects (entry-items) are candidates for suspect objects. The choice of the threshold depends on the expected distance of live objects. Although estimated distances of live objects may temporarily deviate significantly from their actual distances, this is not expected to be common. The threshold can be chosen to be a small multiple of the expected maximum distance.

3.5.1 Complementary Tracing

The idea of complementary tracing is to provide periodically a global tracing collector (cyclic collector) to collect circular garbage (Dickman 1992).

This algorithm violates the desirable property of locality: all processes in the system need to co-operate in order to collect distributed garbage cycles. Indeed, the drawbacks of this scheme are the same as those of global tracing itself (section 3.2), except with reduced severity because the responsibility of tracing as a complementary scheme is to collect cycles of garbage.

3.5.2 Tracing in Groups

Lang *et. al.* (Lang, Quenniac and Piquer 1992) present an algorithm for tracing within groups of processes. They combine reference counting and mark-and-sweep in order to reclaim inter-group cycles of garbage. A group is a set of processes that may overlap or include other groups. Multiple collections on different groups can run in parallel.

A group collection begins with group negotiation. The next phase — initial marking — distinguishes inter-group from intra-group references. For this purpose they use a technique inspired by Christopher (Christopher 1984). At the end of this phase all the entry-items of processes within the group are marked with respect to the group. The marks on entry-items depend on whether they are referenced from inside or from outside the group. In the following phase — local propagation — local collectors propagate the marks of the entry-items towards the exit-items. Then, in the next phase — global propagation — the group garbage collector propagates the marks of the exit-items towards the entry-items they reference, when within the group. The preceding two phases are repeated until marks of entry or exit-items of the group no longer evolve. When this is completed, any dead cycles can be collected. Objects referenced from outside the group are considered to be reachable. Group stabilisation — when there are no more marks to propagate locally — can be detected, in the absence of failures, by any distributed termination detection protocol (the paper does not describe how it can be done).

This algorithm approximates the property of locality in the sense that the collection of intra-group cycles does not need co-operation of other processes in the system, but only of the members of the group. In this way, it tolerates process failures in the sense that if some process is down (or unreachable due to communication problems) the set of accessible processes can still form a group, hence the group collection is not blocked due to a crashed process. However, the new group collection is restarted almost from scratch. Also, a small distributed cycle can be collected quickly by a small group instead of having to wait for a global tracing.

Although it is suggested that a process may be removed or added during garbage collection, knowing which processes should be grouped (and when) in order to reclaim the maximum amount of inter-process cycles of garbage is still a difficulty of this algorithm. Some groups could be large enough so that cycles can be collected, but the large they are the longer the group collection takes. The authors propose a tree-like hierarchy of embedded groups. Multiple group collections can be activated at the same time. Groups may overlap, though this puts more burden on local collections. This ensures that each cycle is covered by some group, but the smallest group covering, for example, a two-process cycle may contain many more processes.

Unfortunately, this algorithm is very difficult to evaluate because of the lack of detail presented. Dynamic configuration of processes into groups that succeed in collecting circular garbage is not trivial.

3.5.3 Local Tracing

Local tracing techniques basically combine reference counting based techniques (acyclic algorithm) with distributed tracing (cyclic algorithm). Usually the cyclic algorithm is triggered at a low rate and most garbage is assumed to be reclaimed by the acyclic one.

Jones' and Lin's approach (Jones and Lins 1993) combines the ideas behind weighted reference counting with mark-and-sweep in order to collect interprocess cycles of garbage, as was first proposed in (Lins and Jones 1993). This idea may be seen as based on the trial deletion technique proposed by Vestal (Vestal 1987).

The mark-and-sweep algorithm does not trace the whole distributed graph. Instead it traces locally from an object suspected to be part of a garbage cycle. Every time a reference to a shared object is deleted, it is inserted on a control-set of suspect objects. When a distributed garbage collection is triggered, it picks an object from the control-set and the object's transitive referential closure — the suspect subgraph — is inspected.

The suspect graph is traced in order to decrement the object's counter (previously copied). At the end of tracing, if the object's counter has dropped to zero, it means that this object belonged to a garbage cycle and it can be safely reclaimed.

Tracing may discover objects that cannot be proved to be garbage. Thus it may involve live objects, leading to wasted work. The efficiency of this algorithm depends on the accuracy of the heuristic for choosing the seed object. So, this algorithm approximates the property of locality. It is thus more scalable than global tracing because it eliminates the need for global synchronisation. However, it has three main disadvantages. First, this algorithm assumes that local collections are also reference counting-based; second, each phase of the local mark-and-sweep must be sure that the preceding phase has finished before starting; finally, the corresponding phases of different distributed collections must synchronise in order to allow concurrency.

Maeda *et. al.* presented (Maeda, Konaka, Ishikawa, TomoKiyo, Hori and Nolte 1995) a new algorithm that borrows the ideas of Jones' and Lins' algorithm. The main advantage is that it does not require the local collections to be reference counting-based. However, as above, if multiple processes on the same cycle initiate separate local tracings, the collection of the cycle will fail.

3.5.4 Train Collection

Hudson *et al.* have adapted their Mature Object Space 'train' algorithm for distributed garbage collection (Moss, Munro and Hudson 1996, Hudson, Morrison, Moss and Munro 1997). This algorithm is complete, non-disruptive, incremental and scalable.

Like the other partitioned collectors, it divides the address space into a number of disjoint partitions called *cars*. To collect cyclic garbage that spans more than one car, cars are grouped together in *trains*. Partitions are grouped in processes. Each car resides on a single process but a train may span more than one process. By ensuring that all the cars in a train are collected by copying the reachable objects into other trains, cyclic garbage will be left behind and can be collected when marshaled into the same train.

This algorithm also uses a reference listing scheme (recall section 3.4.4). It maintains for each entry-item, corresponding to an object o at process P, a set of those cars that have pointers to o at P — the *remembered set* of o. The protocol that implements the insert and delete messages introduced in section 3.4.4 is asynchronous, only assuming FIFO channels between any two processes. When a reference to an object o is transmitted from a process P to process Q, process P informs the process holding o. As noticed by (Birrel et al. 1994), this approach is not fault-tolerant. The sender may notify the process holding the object, but, for some reason, process Q may not receive the reference to o. In this case, Q will be inserted in o's *remembered set*, compromising the liveness of the protocol. The same protocol is used to update the references to an object that has been moved to a different car/train.

Joining a train requires a distributed termination protocol, that only involves the processes that belong to the train the process wants to join. Those processes form a ring that is identified and managed by the process that creates the train — the *master*. A process that wants to join the train communicates with the train's *master*. Leaving the ring is more subtle. The algorithm provides a technique that propagates the *leave* intention around the ring. The technique works for any number of simultaneous deletions from the ring. However, it depends on the fact that messages flowing around the ring cannot pass one another.

This algorithm depends on being able to detect when there are no references into a train from outside of the train, allowing the whole train to be reclaimed at once. For this purpose, it uses a distributed termination protocol that, as above, only involves the processes that belong to the train. The basic idea in detecting that there are no references into a train is to pass a *token* around the train's ring. The protocol described in (Hudson et al. 1997) accounts for objects being created in the train or added to the train during detection.

This algorithm shares the features of any partitioned solution for distributed garbage collection. It is scalable in that it is decentralised, uses asynchronous communication, and has no protocols that demand the involvement of all nodes. It reclaims objects incrementally without global knowledge of reachability. However, it requires an *object substitution protocol* to ensure that all old references to an object are updated to refer to the new copy, when it is moved to another car/train. This seems to add a significant message overhead to the system.

The number of trains and the creation of new trains may influence the degree of locality of this algorithm. A garbage train may include more than one garbage cycle. Consequently, collection of garbage cycles may delay the collection of other garbage cycles. Moreover, this implies that the *token* technique may visit processes not belonging to a garbage cycle in other to be able to collect that cycle. However, the extent of these problems may only be known after measurements of real applications. Intuitively, this algorithm exhibits a good locality, however at a cost of the message overhead required by the substitution protocol.

Finally, although this algorithm is fault-tolerant in the sense that it does not require the participation of all processors in the system, the authors suggest an extension to tolerate process failure and communications failures.

3.5.5 Object Migration

The idea of object migration is to consolidate a distributed garbage cycle on a single process in order to transform a distributed cycle into a local cycle that can be easily reclaimed by a tracing local collector.

This idea was first proposed by Bishop (Bishop 1977). In his thesis he proposes that a local collector be broken into two parts, in order to find which objects are only referenced remotely. These objects are then considered to be migrated to the process from which they are being referenced. This will bring the benefit of consolidating an unaccessible interprocess cycle into a single process where it can be reclaimed. This technique was followed by (Maheshwari and Liskov 1995, Shapiro et al. 1990, Gupta and Fuchs 1993).

Migration techniques have, however, some practical problems: they may tend to migrate live objects along with garbage and they may need to migrate objects multiple times before they converge on the same node. Migration of live objects is undesirable because it wastes process and network bandwidth, and also interferes with load balancing. The definition of heuristics may help to distinguish which objects are likely to belong to a garbage cycle. Some schemes use the "local reachability" heuristic for identifying suspect. Such suspects are migrated either immediately or if they are not invoked for long periods (Gupta and Fuchs 1993, Shapiro et al. 1990).

A recent work (Maheshwari and Liskov 1995) uses the "distance heuristic". It proposes to limit migration to those objects with distances above some threshold because they have a high probability of being garbage. This technique reduces highly the probability of migrating live objects, reducing wasted heavy work.

The consolidation of a garbage cycle may be very inefficient if it involves migrating objects multiple times before they converge on the same process. The work presented in (Maheshwari and Liskov 1995) also presents a simple way of selecting one of the processes containing a garbage cycle as the destination, avoiding multiple migrations.

This algorithm is fault-tolerant and scalable because it does preserve the property of locality: the collection of a cycle only requires the co-operation of those processes containing the cycle. The better the heuristic that identifies candidates for migration, the greater the probability of migrating only garbage objects. However, it still presents some problems:

- Migration requires support for object migration. Some heterogeneous systems either do not allow migration or make it rather cumbersome.
- Migrating an object is a communication-intensive operation, not only because of its inherent overhead but also because of the time necessary to prepare an object for migration and to install it in the target process. It may also interfere with other object management goals such as load balancing (Shivaratri, Krueger and Singhal 1992).

3.5.6 Back-Tracing

Recent works (Fuchs 1995, Rodriguez-Riviera 1995, Maheshwari and Liskov 1997a) propose an original technique based on *back-tracing*, instead of forward tracing, in order to collect interprocess cycles of garbage. Call the traditional reference graph the *forward reference graph* (FRG). The *inverse reference graph* (IRG) is obtained by switching the direction of all the references in the FRG (Fuchs 1995). Back-tracing, as opposed to forward tracing, follows the references in that inverse graph. When an entry-item is suspected to be garbage, the references that point to it are recursively back traced. The back-tracing continues until the closure of all the objects from which the suspect entry-item can be reached is found — the suspect subgraph. If this closure does not contain any root, then all objects in the closure are reclaimed. This algorithm does not need global synchronisation and scales well to a distributed system of many processes since it only involves the processes containing the garbage cycle.

However, this algorithm imposes an extra overhead on the local collector in order to determine the local backwards references. It also needs some kind of synchronisation to allow concurrency between the mutator and collector, and between multiple back-tracings running at the same time in the same cycle in order to avoid duplication of efforts. Finally, although the collection of a garbage cycle only needs the co-operation of the processes that contain the cycle, and thus preserves the *property of locality*, the efficiency of the algorithm depends on heuristics in order to avoid wasted work.

The solution presented in (Rodriguez-Riviera 1995) computes the backward references during the local garbage collection. However, local objects may be traced more than once, which imposes a great overhead on the local collector. To resolve the first synchronisation problem, this solution uses a barrier against new references or method invocations on remotely referenced entry-items to detect modification in the back-references after the last local collection. Then, a second pass through the suspect subgraph, done to inspect the barrier, will state if the back-tracing is still valid or not. This second step requires the state of the first back-tracing to be recorded in a single token-message or maintained in the processor that has started the back-trace (Rodriguez-Riviera and Russo 1997). The former solution is fault-tolerant in the sense that, if a processor is known to have crashed, it is just ignored (references from the crashed process are deleted); if not, a network failure is assumed and the token is safely discarded. Later, another suspect would start another back-tracing. The later solution is less fault-tolerant because if the starter process fails, the back-trace must be abandoned. However, recording the back-tracing state in a single message may lead to huge messages if the suspect graph includes too many objects.

Fuchs (Fuchs 1995) does not suffer from this problem since he assumes that there are only remote references, and uses Piquer's or Birrel's algorithm. Consequently, an entry-item always knows about new references to it. But this situation is unrealistic.

Maheshwari and Liskov (Maheshwari and Liskov 1997a) present an efficient method for computing local backward references that uses Tarjan's algorithm (Tarjan 1972). The method computes the backward references during the local forward trace during a local garbage collection for every suspect entry-item. Each local object is only traced once.

To resolve the first synchronisation problem, they use the same barrier as (Rodriguez-Riviera 1995). However, in contrast with (Rodriguez-Riviera 1995) it does not perform a second pass through the suspect sub-graph. This has the advantage of not needing to record the state of the back-trace. Instead, it creates a chain of activation frames for each call on each entry or exit-item — a call returns *garbage* if it reaches an item that has been already visited or returns *live* if it reaches a root. A back-trace is *active* at an item if it has a call pending there. Thus, the algorithm safely ensures that if there is any overlap in the periods when a barrier is performed in an item and when a back-trace is active there, the trace will return *live*.

As for the second synchronisation problem, (Rodriguez-Riviera 1995) and (Maheshwari and Liskov 1997a) do not present any solution. However, (Maheshwari and Liskov 1997a) argues that, using the *distance heuristic*, it is likely that one suspect item will cross the distance threshold first.

Fuchs (Fuchs 1995) present an algorithm that uses a total partial order in the back-tracings identifiers. If two different back-tracings arrive on the same object, the one with the 'biggest' identifier will proceed, and the other one is blocked until the higher priority back-tracing terminates. If the encounters are in order of decreasing priority, his solution may still lead to repeated work.

Fuchs (Fuchs 1995) and Maheshwari and Liskov (Maheshwari and Liskov 1997a) suggest the use of back-tracing in conjunction with the "distance heuristic". This decreases significantly the probability of performing wasted work. Rodriguez-Riviera (Rodriguez-Riviera and Russo 1997) suggest the use of the "local reachability" heuristic in conjunction with *generational back-tracing* and *back-tracing factoring*. The idea is to improve the accuracy of the "local reachability" heuristic by taking into account the ages of the objects. He tries to decrease the number of redundant back-tracings by collecting less frequently objects that have survived collections. Back-tracing factoring improves this heuristic by propagating the output of a failed back-tracing to every object involved in the back-tracing.

The three algorithms presented are fault-tolerant. If processes crash, the first two algorithms abort very cheaply (this is not true for the solution in (Rodriguez-Riviera and Russo 1997), as we said). They are also resilient to duplicated messages, relying on the idempotency of the algorithm operations and message identifiers. Acknowledgements protect them against loss of messages.

Back-tracing algorithms are promising for cyclic distributed garbage collection, although the problem of synchronising multiple back-tracings running at the same time in the same cycle in order to avoid duplication of efforts still persist. Also, pathological configurations — for example, the mutator may infinitely create new back paths for a back-tracing — may compromise the liveness of the algorithm. A possible solution is to abandon the back-tracing, but this would lead inevitably to wasted work. However, this configuration is not likely to happen.

Only measurements of the behaviour of such algorithms in real applications may give a better understanding of their effectiveness.

3.6 Garbage Collection in Distributed Shared Memory

In Distributed Shared Memory systems garbage collection is provided by adding functionality to the Distributed Shared Memory service, rather than built on top of it. The work presented in (Ferreira and Shapiro 1996, Ferreira 1996), supports persistence by reachability in a distributed shared address space transparently and efficiently. The main issue in these system is coherence interference. This work addresses this issue while being scalable and efficient.

Garbage collection for persistent distributed shared memory systems borrows many ideas from the algorithms we have described. However, they must be extended to account for multiple replicas of objects. In addition, while in RPC-based systems there is one partition per process, in such systems partitions and processes are orthogonal.

The shared address space spans every process in a distributed system and it is inherently large. Consequently, a scalable solution should be based in a partitioned model of the address space. The solution in (Ferreira and Shapiro 1996, Ferreira 1996) approximates a global trace with a series of non-synchronised, per partition, local traces. Each partition (a bunch) is collected independently at the process where it is cached. In addition, if a partition is replicated, each one of its replicas is also collected independently. Inter-partition garbage is collected using the reference listing protocol described in section 3.4.4.

The intra-partition collection does not compete with applications for coherent data, there is no synchronisation between collectors and mutators or between different collectors, and the garbage collection messages are asynchronous and exchanged in the background. The price to pay for these features is some degree of conservativeness, and some messages need to be delivered in causal order.

In order to collect inter-partition cycles of garbage, this algorithm performs a group collection in partitions cached in the same processor. Here a group collection is simpler than in (Lang et al. 1992) because this algorithm uses reference listing instead of reference counting. In this way, references external to the group are easily determined by not considering entry-items only reachable from inside the group as roots for the group collection. However, such groups may not contain all of a garbage cycle.

3.7 Garbage Collection in Object-Oriented Database Management Systems

Garbage collection in Object-Oriented Database Management systems borrows many ideas from the algorithms we have described. Usually they are extended to deal with the specific safety problems posed by transactional systems. We survey here some of the work in the literature. However, we mainly focus on how high-level design decisions, such as partitioned *v.s.* non-partitioned garbage collection, affects garbage collection.

The work on automatic garbage collection in these systems was mainly developed on a server-based basis for multiple client-single server architectures (Franklin et al. 1989, Kolodner and Weihl 1993, Yong, Naughton and Yu 1994, Cook, Wolf and Zorn 1994, Amsaleg, Franklin and Gruber 1995, Moss et al. 1996, Cook, Klauser, Wolf and Zorn 1996). This is because here garbage collection takes the view that data resides mostly on secondary storage, with main memory being used as a temporary cache buffer. They focus on garbage collection of persistent stores, which are the core of Object-Oriented Database Management systems. The only work we know for multiple client-multiple server architectures is (Maheshwari and Liskov 1994).

The collector presented in (Kolodner and Weihl 1993) is an incremental copying collector and is correct in the presence of concurrency, concurrency control, and crash recovery. This work was mainly concerned with devising correct algorithms, in the face of concurrency and/or failures (Moss et al. 1996). This is a non-partitioned approach, thus, like other non-partitioned schemes, this collector makes random access to the heap and requires the traversal of the entire persistent address space in order to collect any garbage.

Other works were mainly concerned with efficiency (Yong et al. 1994, Amsaleg et al. 1995, Moss et al. 1996, Ng 1996, Maheshwari and Liskov 1997b). All of them approach the garbage collection problem using a partitioned scheme. As in the algorithms described in this chapter, partitioned collection of a persistent store alone does not collect inter-partition cycles of garbage.

(Yong et al. 1994) compared incremental copying, reference counting, and partitioned collection in Object-Oriented Database Management systems and found partitioned collection to perform the best. The partitioned scheme involves multiple clients and a single server. It uses a *remembered set* for each partition that holds the identification of each object holding a reference to that partition. The remember set is created and maintained by a *write barrier*. Every object in the remember set needs to be fetched and scanned before tracing a partition.

In contrast, the work in (Maheshwari and Liskov 1997b) provided an efficient method that allows partitions to be collected independently. They remember the objects in the partition that are referenced from other partitions in an *entry-table*. To keep track of which partitions reference an object they use a scheme akin to the *reference listing* protocol described in section 3.4.4. They also record information about outgoing references from a partition in an *exit-table*. Additionally they describe a global tracing scheme for collecting inter-partitions cycles of garbage.

PMOS by (Moss et al. 1996) is the persistent version of the DMOS described in section 3.5.4. They collect cycles of garbage and address efficient maintainance of inter-partition references. As noticed by (Maheshwari and Liskov 1997b), collecting a partition (a *car*) may involve accessing multiple target partitions.

Work by (Cook et al. 1994) investigates heuristics for selecting a partition to collect when a garbage collection is necessary. Their results show that the partition selection policy can significantly affect application performance and proposed a new policy based on the observation that when a pointer is overwritten, the object it pointed to is more likely to become garbage.

The garbage collection algorithm in Thor (Maheshwari and Liskov 1994) has been designed for a multiple client-multiple server architecture. Clients cache in memory objects that are being accessed. The complexity added to the algorithms above is that the persistent store is distributed, consequently, garbage collection of the persistent store has to account for references in other servers. Also, clients may acquire references to servers through the fetching of objects in other servers.

This algorithm is a fault-tolerant version of the reference listing scheme (recall section 3.4.4) that handles fetches of objects into clients and commits of transactions. Every server keeps entry-tables for clients — garbage collection between client and servers — and for other servers — garbage collection between servers.

3.8 Summary

Distributed garbage collection poses a challenging problem: reclaiming all data structures while achieving efficiency, scalability, completeness, fault-tolerance and safety. Several proposals have been made to design a distributed garbage collection that fulfils all these requirements. The great number of incomplete proposals reflects the difficulty of the problem.

The most suitable algorithms are those based on reference counting. They can be made fault-tolerant to message and process failures but they cannot reclaim cycles of garbage.

The second family, tracing-based techniques, ensures better liveness but most of them make strong assumptions on the reliability of the network. They require all processes to co-operate in the distributed collection. Therefore, those techniques cannot progress if a single process is crashed.

The drawbacks of global tracing are not so severe if it only runs infrequently, and its responsibility is limited to collecting circular garbage. However, all processes must be up together for tracing to complete. This violates the desirable property that the collection of a cycle not depend on processes other than those that contain the cycle. This drawback can be alleviated if the tracing is confined to a suspect subgraph. However existing solutions are either not fault-tolerant, or do not allow concurrency between different tracings, or do not provide methods to confine the sub-tracing to suspect objects.

Another method to alleviate the drawbacks of global tracing is to trace within groups. The problem with group tracing is configuring groups in order to collect inter-partition cycles. Cycles may never be covered by any group and collection of larger cycles may delay the collection of smaller ones.

Migration schemes for collecting cyclic garbage have the locality property. Since migration is expensive it is important to use a good heuristic for finding suspects. The distance heuristic alleviates unnecessary migration. However, some systems do not support migration due to security or autonomy constrains or due to heterogeneous architectures. Forced object migration may also result in load-imbalance.

Recently, new techniques have been proposed: Back tracing, Train Collection, and Causal Dependencies (Timestamp Packet Distribution based). These techniques exhibit good locality in collecting cycle garbage and look promising for distributed systems. However, real measurements would be necessary to know to what extent the message and space overheads of the Train Collection technique and the Causal Dependencies technique are a problem.

Back tracing is fault-tolerant, concurrent and scalable. However, how to control multiple back tracing in the same subgraph that could lead to repeated work is still an open issue.

Chapter 4

A Cyclic Distributed Garbage Collector

In this chapter we describe a garbage collection technique for large address spaces that has the potential to collect garbage cycles of objects efficiently.

In a RPC-based system, each process maintains its local address space as a partition (recall section 1.3). In this and the following chapters we will use *process* as synonymous for *partition*. Our description is based on the partitioned model we provided in section 3.1 and takes into account the mutator model described in section 1.3 for RPC-based systems.

Our technique is designed to work with a partitioned solution for distributed garbage collection: reference listing (recall section 3.4.4). In this chapter and the following one we assume a safe reference listing protocol is provided. We augment the reference listing scheme with Partial Tracing (PT) in order to collect inter-process garbage cycles.

A partial tracing is a tracing that only involves a subset of the processes in the system. This definition will be made clear in the next sections. This technique provides an efficient, scalable and fault-tolerant solution for RPC-based systems and shows promise for garbage collection of persistent stores. We want to provide completeness while not compromising our primary goals of efficient reclamation of local and distributed acyclic garbage, low synchronisation overheads, avoidance of global synchronisation, and fault-tolerance. All these aspects raise interesting problems in terms of safety.

As we have already pointed out, the challenge in collecting inter-process garbage cycles is to preserve locality, that is, to involve only those processes containing the cycle. Our method belongs to the category of methods that approximate this property, in the sense that it relies on heuristics for identifying objects that may belong to a garbage cycle. A group of processes co-operates in the detection of garbage cycles. Group membership is determined by heuristics that improve inter-process garbage cycle collection. Collection operates in three phases. First, it identifies a subgraph suspect of being a garbage cycle: subsequent efforts are confined to this subgraph alone. This phase also defines the group of processes that will collaborate to collect cycles. The second phase determines whether objects of this subgraph are actually garbage. Finally, the last phase makes those garbage objects discovered available for reclamation by local collectors.

This technique has the luxury of using techniques that are too costly if applied to all objects or to uniprocessors, but are acceptable if applied only to a subset of distributed objects we call *suspects*. More precisely, it may be seen as framework within which other heuristics may be used.

This chapter first presents an overview of our solution. Then, it describes our primary goals and outlines some strategies, in the light of the overview, to meet them. The following sections describe in detail the different phases of a partial tracing. First we give a very simple description of the different phases without considering termination. Then, we present the termination protocol. Finally, in section 4.6, we discuss heuristics to discover *suspect* objects and to improve the algorithm's discrimination, and hence its efficiency.

4.1 General Overview

The reference listing algorithm reclaims acyclic inter-process garbage. However, it is incomplete because does not collect inter-process garbage cycles. For example, consider the inter-process cycle illustrated in figure 11. It crosses processes A, B, C and D. The entry-items for objects a, c, e and g are considered roots of A, B, C and D processes' local collection respectively. This condition does not allow the collection of exit-items reachable by the garbage objects. In this case, the corresponding entry-items' entry-list is never emptied by the reference listing protocol (recall section 3.4.4), that is, a cycle of

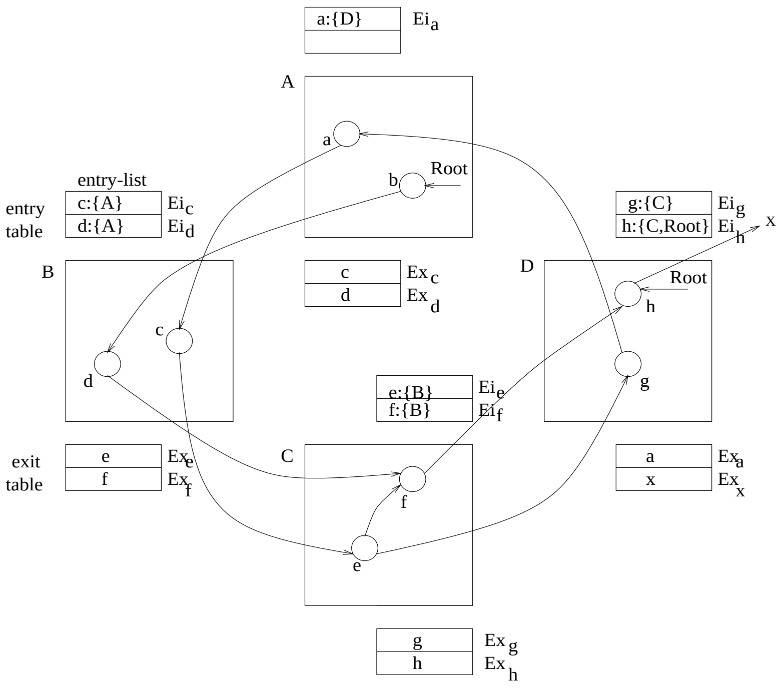


Figure 11: Inter-process Garbage Cycle

garbage is self-supporting with respect to reference listing (more precisely, with respect to every form of reference counting).

Our scheme is based on the fact that an inter-process cycle of objects is garbage if it is only reachable from global roots (recall section 3.1) internal to the cycle, that is, only from entry-items only reachable from exit-items on the cycle. Note entry-items Ei_a, Ei_c, Ei_e and Ei_g in figure 11. It first identifies a distributed subgraph that may be garbage; secondly, it discovers whether members of this subgraph are actually garbage by determining if they are reachable from any global root external to the subgraph. Finally, it makes any garbage objects available for reclamation by local collectors.

The distributed collector requires that each item in processes' entry and exit-table has a colour — red, green or white — and that initially all items are white. Red means that an item may be garbage; green means that we cannot conclude that an item is garbage (although it may be). Entry-items also have a *red-list* of process names, akin to their entry-list.

Partial tracing is initiated at *suspect* objects: objects suspected of belonging to a distributed garbage cycle. A new partial tracing may be initiated by any process not currently part of a tracing. There are several reasons for choosing to initiate such an activity: the process may be idle, a local collection may have reclaimed insufficient space, the process may not have contributed to a distributed collection for a long time, or the process may simply choose to start a new distributed collection whenever it discovers a suspect object.

The partial tracing operates in three phases:

Mark-red phase We identify a red set of objects reachable from an object heuristically suspected of belonging to a garbage cycle, starting from the corresponding entry-item. This phase determines implicitly which global roots are reachable only from the suspect subgraph, and forms a group of processes that will collaborate in the subsequent phases.

Scan phase We try to isolate self-contained red subgraphs, that is, garbage cycles: the mark-red phase may lead to the discovery of entry-items in the suspect red subgraph that are reachable from outside this subgraph. These items must be considered live[1]. We perform a group collection that aims at marking green any red object reachable from outside the red subgraph, that is, red objects reachable from a non-red global root (recall that the global root set includes local roots of each process and process's entry-table). A group collection involves a local trace in each process. However, to trace a group: (*i*) those red entry-items only reachable within the suspect subgraph are not considered as members of the local roots, and (*ii*) tracing continues across boundaries internal to the group, when red exit-items are marked green. The scan phase 'rescues' any red object that may be live.

Sweep phase Any objects remaining red are garbage. We make them available for collection by the local collector.

There is an important detail in the design of our system that concerns the reachability between entry and exit-items. A first approach invokes a trace from entry to exit-items every time reachability information is needed. In section 5.1 we show that we can always cache this information, and we describe a solution proposed in (Maheshwari and Liskov 1997a) that efficiently computes the required information in chapter 8.

[1] Although, they might be garbage.

The mark-red phase is able to use such information. However, the scan phase will still perform a recursive trace until we introduce the concurrent model in section 6.

4.2 Goals and Outline of Solutions

In this section we describe the specific problems we want to solve and we outline our corresponding solutions. Our solution is directed at RPC-based systems, although it shows some potential for garbage collection of persistent stores. Our main goals are then efficiency, scalability and completeness, and fault-tolerance. In section 9.2.4 we describe a possible implementation in persistent stores.

4.2.1 Scalability and Completeness

Our scheme combines reference listing (across process boundaries) with a subgraph tracing scheme. Reference listing does not collect inter-process cycles, thus it is not complete. However, it is scalable: it allows independent local garbage collection and only involves processes containing objects suspected as garbage when collecting inter-process acyclic garbage. Partial tracing is also a scalable technique since it only involves processes that may contain garbage objects and does not require the co-operation of all processes in the system.

Ideally, a cyclic garbage collector would reclaim all cycles of garbage objects. We claim that our solution has the potential to reclaim all garbage cycles in a large address space without need for global synchronisation. Our solution provides some degree of adaptability and can take advantage of heuristics to improve completeness, including hints from the user program. However, as we show in the next chapter, our first system design decision is to trade completeness for promptness.

Collecting cycles by tracing within arbitrary groups may be a heavyweight mechanism in that it may never collect all cycles. Suppose there are disjoint cycles between pairs of sites A, B, C, and D. Then if A and B and C and D always pair up, cycles between B and C will never be collected. The problem is more serious in a larger network. Our scheme has a first phase that defines which processes should be involved in collecting cyclic garbage. Thus, we use heuristics to form groups opportunistically. Ideally, this heuristic would guarantee that B and C eventually would pair up.

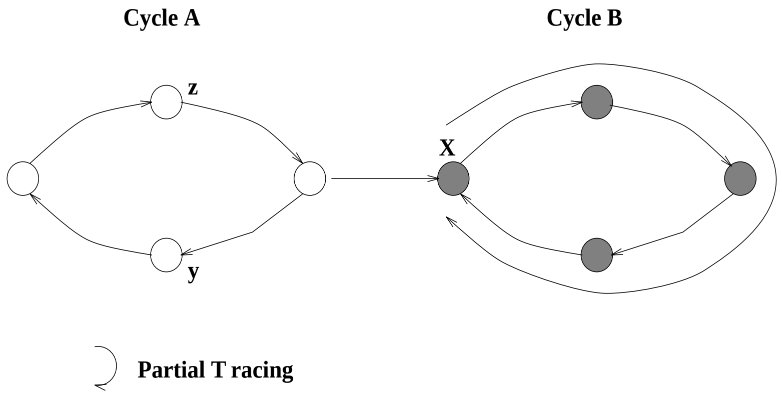


Figure 12: Cycle Dependency

The key to this problem is to allow processes to co-operate, simultaneously but independently, in the detection of garbage cycles. In the case of disjoint garbage cycles, and assuming that each one would be covered by a separate partial tracing, this would be possible and would result in the independent collection of each one. For example, two processes containing a two-process cycle may also contain part of a bigger cycle. Those two processes may participate, independently and simultaneously, in the collection of the two cycles, without having to synchronise to organise the collection of each different cycle.

Moreover, by allowing each partial tracing to work independently and to use private information, different partial tracings may operate on the same cycle (this will be explained in section 5.1) and then co-operate in the collection of that cycle.

Some problems may also arise concerning shape of cycles of garbage. Cycles may reference other cycles, possibly of different dimensions (number of edges and objects). Depending on the chosen suspects, collection of one cycle may either delay or abort collection of other one. This is because the mark-red phase may not trace a whole set of connected garbage objects. Delays may be acceptable assuming cyclic garbage spanning processes is generated slowly. In its turn, aborted collections lead to ineffective and wasted work.

Consider figure 12. It represents a graph formed by two cycles A and B. For simplicity, each circle represents an object and each object is allocated to a different process. We call A and B *connected* cycles. Cycle B is dependent on cycle A, because cycle B is reachable from cycle A, even if cycle A is garbage. X's liveness depends on information from cycle A. The partial tracing shown in the figure will fail without

A's cooperation. This cooperation may be through the reference listing scheme or through a partial tracing on A. One the other hand, cycle A may be collected without cooperation of processes containing cycle B. Objects on cycle A are not accessible from objects on cycle B, thus their liveness may be determined without cooperation of cycle B. However, a partial tracing initiated by any process containing cycle A may involve cycle B as well. In this case, all the represented processes would form a group and co-operate in the simultaneous collection of cycle A and B.

A slightly different problem arises from the fact that, z and y, being suspects of the same cycle, may initiate two different partial tracings on the same cycle. Although they are allowed to co-operate, the different phases of each partial tracing might not be synchronised, thus preventing the collection of cycle A. These restrictions make even more difficult the selection of suspect objects.

We accept that our solution trades completeness for promptness. However, we allow co-operation of different partial tracings and mark-red phases to provide some degree of adaptability in the way that our system may decide when to stop mark-red, and possibly restart another partial tracing, based on some heuristics or hints from the user program. We may also require the synchronisation of the begining of scan phase of each co-operative partial tracing. These avenues are better discuss in section 9.1.

4.2.2 Efficiency

As stated in section 1.4, efficiency is concerned with mutator overheads and collector promptness. There are two fundamentals difficulties concerning the mutator overhead of tracing partitioned schemes. First, how are low synchronisation overheads to be ensured, in order to be unnoticed by the user independently of the size of the address space, and second, how to avoid the need for global barriers.

To achieve the first goal, our scheme runs concurrently and asynchronously with the mutator. To achieve the second, our scheme reduces the need for synchronisation with the mutator in two ways. Firstly, it only requires mutator cooperation when accessing remote objects, $i.e$, in the concurrent model, local mutator activity does not incur in any additional overhead to ensure safety. Synchronising action is associated only with new inter-process reference creation, and invocation of remote methods. Action is needed only if a new inter-process reference to a red object is created and whenever any remote method is invoked. However, as we describe, these control actions are very cheap. Secondly, once red objects are suspected of being garbage, the probability of their being mutated or new inter-process references to them being created is small. Clearly, the better the heuristic for identifying suspect objects the lower these overheads are.

Our scheme achieves promptness in two ways. First, it does not compromise the reclamation of local and acyclic distributed garbage. Second, it approximates the property of locality: the first phase uses a strong heuristic to define suspect subgraphs and those processes that should co-operate in subsequent efforts. This improves the probability of success — reclaiming garbage — for the subgraph tracing, and reduces the need for global synchronisation.

4.2.3 Fault-tolerance

A garbage collection scheme should be safe and complete in the presence of failures. In this thesis we assume fail-stop process semantics and accept communication failures and/or delays. Whenever a process fails, all contained objects are inaccessible.

As we described in section 3, inter-process reference listing is fault-tolerant in the sense that the detection of distributed garbage needs the cooperation of only those processes that the garbage was reachable from. Thus, if a processor is temporarily unavailable, or otherwise slow in doing local collection, it will prevent the collection of only the garbage that is reachable from its objects.

Our scheme approximates this feature: the collection of a garbage cycle is likely to only require the cooperation of processes that contain the garbage cycle. In this way, our system allows garbage to be collected despite unavailability of parts of the system.

Idempotent remote operation and a system of acknowledgements, between the processes involved in the partial tracing, are the basis of our fault-tolerant scheme with respect to messages failures. We deal with lost, duplicated and out of order messages.

4.3 Mark-red Phase

The aim of the mark-red phase is to trace from an object suspected of being garbage, thereby defining an inter-process subgraph of objects that may be a garbage cycle. The key insight behind this idea is to alleviate the drawbacks of global tracing by confining the efforts of tracing to a subset of the address space, approximating the property of locality.

The mark-red phase only identifies objects suspect of being garbage. An entry-item corresponding to a suspect object is marked red. Entry and exit-items traced from this suspect entry-item are also suspected of forming an inter-process cycle of garbage, and hence marked red. At the end of mark-red phase the set of red items may be a superset of the set of garbage items, that is, some red items may be referenced by references external to the cycle and hence may be live. This condition will be discovered by the scan phase.

In the following sections we describe the basic techniques for conducting the mark-red phase. For the moment we ignore concurrency, scalability and fault-tolerance. We include an example for illustrating this phase and present the algorithm.

4.3.1 Mark Steps and *Red-list*

A initiator process — the process initiating the partial tracing — chooses a suspect object and marks red the corresponding entry-item. The mark-red phase is a tracing technique. Traced entry and exit-items are coloured red. It takes two kinds of step:

Local-step that goes from reddened entry-items to exit-items reachable from them in the same process. Exit-items are reddened if not already red. For the present, local objects are traced recursively in order to reach exit-items. Local traced objects are marked red to allow termination of local steps.

All reddened exit-items execute a remote-step.

Remote-step that goes from reddened exit-items to the corresponding entry-items on the target process. We call a remote step a *mark-red request*. Entry-items are reddened if not already red, and the source process is inserted in the *red-list* of the target entry-item. The *red-list* records which references are internal to the red closure by using the identifier of the remote process (akin to the reference list protocol). The *red-list* for an entry-item Ei_x contains those processes that have a red exit-item Ex_x. The formal definition of *red-list* is:

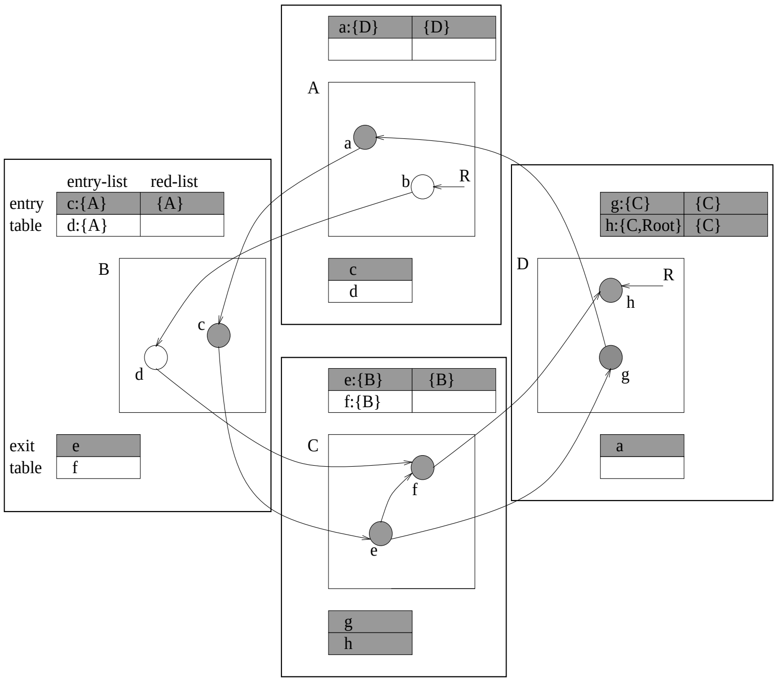


Figure 13: Mark-red phase identifies a subgraph suspect of being garbage

Definition 4.1

$$Ei_x.\text{red-list} =$$

$$\{p \in \text{processes} : Ex_x \in \text{exit-table}(p) \wedge \text{colour}(Ex_x) = \text{red}\}$$

All reddened entry-items execute a local-step.

The mark-red phase consists of taking local and remote-steps alternately. Consider figure 11 on page 82. Assume the suspect object is a in process A. For the present, let us consider an object to be suspect if it is not referenced locally, other than as an entry-item. A mark-red trace will start at entry-item Ei_a and will take a local step to exit-item Ex_c. From there, it will generate a mark-red request to entry-item Ei_c in process B and so on.

The result of the mark-red phase is illustrated in figure 13. Shaded objects and shaded entry and exit-items denote red items. White denotes white items. Note that object f in process C is not garbage although it has been marked red, as has the corresponding entry-item: its liveness will be detected by the scan phase.

4.3.2 Mark-red Algorithm

The local and remote-steps described in section 4.3.1 are implemented by the algorithm below. A local-step (mark-red) generates a remote-step — mark-red request — for each red exit-item. The mark-red request is sent to the corresponding entry-item. In its turn, every mark-red request received (handle-marked-request) generates a local-step.

```
mark-red(entry-item Ei_x) =
  if colour(Ei_x) not red
    colour(Ei_x) = red
    for exit-item Ex_y in local-transitive-closure(Ei_x) do
      if colour(Ex_y) not red then
        colour(Ex_y) = red
        send-markred-request(thisprocess,Ei_y)
end

handle-markred-request(process P, entry-item Ei_y) =
  entry-table[Ei_y].red-list = entry-table[Ei_y].red-list ∪ {P}
  mark-red(Ei_y)
end
```

Until now, we ignored the problem of distributed termination detection. To detect it, we use a scheme based on acknowledgements (not shown in the pseudo-code above) (Dijkstra and Scholten 1980): every mark-red-request generated waits for an *acknowledgement*. When the process that initiated the partial tracing has received acknowledgements for all mark-red request it has sent, the mark-red phase has terminated. It then reports to all processes reached during this phase, called the *participants*, the begining of the next phase. To allow the initiator to determine the set of participants, each participant appends its identifier to the acknowledgement of a mark-red request. This acknowledgement system will be explained in more detail in section 4.5.

4.4 Scan and Sweep Phase

The scan phase aims to isolate red cycles of garbage by colouring any red accessible object green to prevent it from being reclaimed. In this section we describe the basic technique for conducting the scan phase. It ignores mutator concurrency, scalability and fault-tolerance. We include an example illustrating this phase and present the algorithm.

4.4.1 Scan Steps

The scan phase is performed concurrently on each participant. In each participant, first we determine the members of the global root set that are not members of the suspect subgraph. We will call them the *local-scan-root-set.* The *local-scan-root-set* does not include any entry-items that may be internal to the red sub-graph. Red entry-items whose entry and red-lists are equal are directly reachable only from the red sub-graph, and are kept red. Red entry-items whose entry and red-lists differ must be accessible from outside the suspect sub-graph, and so are marked green. We give the following definition:

Definition 4.2 local-scan-root-set *is a set, in each participant, consisting of the local roots, non-red entry-items and red entry-items whose entry and red-lists differ.*

As for mark-red, the scan phase is a tracing technique. Traced entry and exit-items are marked green. It takes three kinds of step:

Initial-step Mark green any red entry-item in the *local-scan-root-set.* Mark green any red exit-item that is reachable from the *local-scan-root-set.* For the present, traced local objects are coloured green to allow termination of local steps (akin to the mark-red phase).

All greened exit-items execute a remote-step.

Remote-step Propagate the green colour from greened exit-items to the corresponding entry-item in the target process. Mark green the entry-item if red. We call a remote step a *scan request.*

All greened entry-items execute a local-step.

Local-step Propagate the green colour from greened entry-items to those locally reachable exit-items. Mark green the exit-items if red. For the present, local objects are traced recursively in order to reach exit-items. Traced local objects are also marked green to allow termination of local steps.

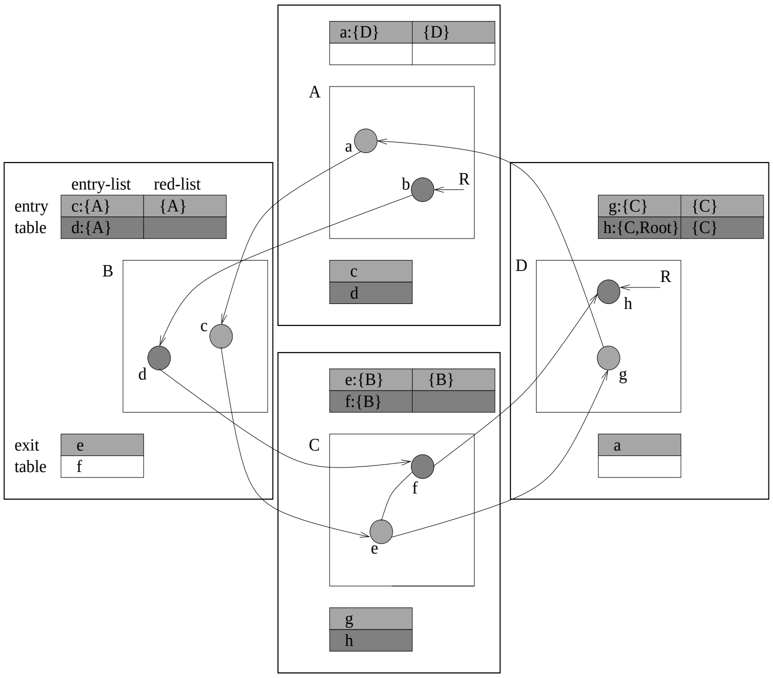


Figure 14: Scan phase 'rescues' any red objects that may be live

All greened exit-items execute a remote-step.

The scan phase consists of taking an initial-step in each participant followed by alternate remote and local steps, until all potentially live red objects have been marked green. Continuing our example, each process performs an initial-step. For that, it determines the *local-scan-root-set*. Entry-item Ei_f is not red, so is marked green and becomes a member of the *local-scan-root-set*. Object f and exit-item Ex_h are marked green by process C's initial-step. A remote-step from Ex_h marks green Ei_h in process D, which becomes a root for a local-step. h is marked green by the local-step. Figure 14 illustrates the situation at the end of the scan phase. Darker grey objects and items represent green ones. Red objects are ready for collection through the sweep phase.

4.4.2 Scan Algorithm

The initial, remote and local-steps described above are shown below. Initial-step (main-scan) and local-step (scan) generate a remote step — scan request — for each exit-item marked green, and a request is sent to the corresponding entry-item. In its turn, every scan request received (handle-scan-request) will generate a local-step.

```
main-scan() =
  for Ei_y in local-scan-root-set
    if colour(Ex_y) is red then
      colour(Ei_y) = green
  for exit-item Ex_y in
            transitive-closure(local-scan-root-set) do
    if colour(Ex_y) is red then
      colour(Ex_y) = green
      send-scan-request(Ei_y)
end

handle-scan-request(entry-item Ei_y) =
    scan(Ei_y)
end

scan(entry-item Ei_x) =
  if colour(Ei_x) is red
    colour(Ei_x) = green
    for exit-item Ex_y in local-transitive-closure(Ei_x) do
      if colour(Ex_y) is red then
        colour(Ex_y) = green
        send-scan-request(Ei_y)
end
```

Each *participant* detects termination of the distributed computation generated by its initial-step using the same scheme we introduced for the mark-red phase. However, in order to proceed to the next phase, each *participant* has to detect that all participants have terminated. We describe a solution in section 4.5: group termination is detected by the process that initiated the partial tracing — the Initiator.

4.4.3 Sweep Phase

At the end of the scan phase, any remaining red entry-items must be part of inaccessible sub-graphs, and can thus be safely reclaimed.

The sweep phase is performed in each participant independently. Our scheme is designed not to interfere with the reference listing scheme, which is responsible for the collection of entry and exit-items (recall section 3.4.4). Such red entry-items are not removed immediately in order to maintain referential integrity between exit and entry-items. Thus, we keep the red entry and exit-items. However, in the next local collection, red entry-items corresponding to a finished (scan phase) partial tracing will not be used as roots in the local collection. Consequently, objects belonging to the garbage cycle will be collected the next time the containing processes do a local collection. Additionally, the sweep phase resets the colour of green items to white.

When red exit-items are deleted, the corresponding entry-list is updated by the reference list scheme. When the entry-list is empty, the entry-item may then be removed.

4.5 Termination

In this section we will address the problem of termination detection. We will define the distributed termination detection problem, present a solution initially proposed by (Augusteijn 1987), and describe a report phase for both mark-red and scan phase. For the present, we do not consider concurrency, scalability or fault-tolerance. In chapter 5 we present variants of our solution which cope with the advanced features of our system.

4.5.1 Distributed Termination Protocol

A partial tracing is a multi-phase algorithm. It needs to determine the end of each phase in order to progress from one phase to the following one, namely, from mark-red to scan and from scan to sweep. This requires some kind of synchronisation between processes co-operating in a given partial tracing. This need for synchronisation is much less restrictive than the need for synchronisation in global tracing solutions: it only involves processes involved in a partial tracing; this approximates the property of locality.

In a distributed system where processes communicate only via messages, in general no process has a consistent and up-to-date view of the global state. As a result, it is difficult to decide whether or not the global state is one in which a distributed computation has terminated. This is particularly true in our context, where some processes may have finished their local steps, while others are still working. New remote steps (mark requests[2]) may be generated and result in new local steps. As a consequence, 'finished' processes may later have to compute local steps again.

[2] We will use the notation *mark requests* to refer to both mark-red requests and scan-requests

However, termination is a *locally stable* property. Locally stable properties are those for which once the property becomes true, the state of the processes over which the property holds do not change with respect to the property, *i.e.*, the property never becomes false again (Marzullo and Sabel 1994).

Returning to our problem, it is not possible for a process to decide whether it will later generate new mark requests. Therefore, it is always assumed that for each process a local *condition of stability* exists (Tel and Mattern 1993). When this condition holds, no local steps will be generated by the process, and no initiative of the process itself will falsify the condition of stability. It now follows that if a global state is reached in which that condition of stability is satisfied, simultaneously, in every process and no mark requests are in transit, the computation is terminated.

Several classes of solutions to the termination detection problem are known. (Tel and Mattern 1993) identified those based on *probes* (Dijkstra, Feijen and van Gasteren 1983) and those based on *acknowledgements* (Dijkstra and Scholten 1980) as the most important ones. We adopt an acknowledgement based approach because it deals with mark requests in transit. Additionally, our system is opportunistic — it identifies dynamically a suspect subgraph and those processes that will co-operate in collecting distributed garbage. Consequently, we have chosen an acknowledgement based termination detection protocol that does not require the processes involved to be known *a priori*. This is one of the features of our scheme that make it scalable. Our mark-red phase detects on-the-fly processes that will co-operate in the partial tracing.

In order to define the required condition of stability, we first describe some requirements of our system. We require every mark request to be acknowledged. We introduce the notation *grey-marked*[3] to identify exit-items that have generated mark requests which have not yet been acknowledged. Those exit-items are inserted in a *grey-set*. It is required that whenever a process acknowledges a mark-request it identifies the exit-item. To achieve this, whenever a process receives a mark request it inserts the source exit-item in a *reply-set*. *Grey-set* and *reply-set* are defined below. Intuitively, we can infer that if no process has local steps to perform and all grey-sets are empty, then the distributed computation is finished.

[3] *grey-marked* actually means grey-red-marked or grey-green-marked depending on which phase we are in.

Following (Augusteijn 1987) we introduce three possible states for a process: *active-disquiet*, *passive-disquiet*, or *passive-quiet*. We additionally introduce a new state, *Inactive*, to model the situation when a process, not yet participating in a phase, receives a mark request. We also introduce a process condition to describe when a process may, by itself, generate mark requests in respect to a instantiation of a phase:

a dynamic process may generate mark requests by itself; it may also generate mark requests as a consequence of receiving mark requests from other processes;

a non-dynamic process only generates mark requests as a consequence of receiving mark requests from other processes.

Initially every process is inactive. It may participate in a phase by receiving a < *start-phase* > event or by receiving a mark request. An inactive process receiving a < *start-phase* > message enters the phase and its state turns to active-disquiet. An inactive process receiving a mark request turns to:

- active-disquiet if it is a **dynamic** process. When it has no more local steps to perform and its grey-set is empty, it changes to passive-quiet. If it is subsequently reactivated by receiving mark-requests, it switches to passive-disquiet and then reverts back to passive-quiet when it has performed all its local steps and its grey-set is again empty.
- passive-disquiet if it is a **non-dynamic** process. When it has no more local steps to perform and its grey-set is empty, it changes to passive-quiet.

Thus, it is clear that once a process has become passive it remains so.

The state transaction diagram for each phase is illustrated in figure 15. The events illustrated in the figure are summarised in figure 16 on page 99.

When a disquiet process receives a mark request it sends the acknowledgement immediately, but when the receiver is quiet it becomes disquiet and it delays the acknowledgement until it becomes quiet again. In this case, from the receiver's point of view, the process sending the mark request is responsible for any further mark requests, *i.e*, it is the process responsible for detecting the termination of computing activity generated by such requests. We call it the]em parent. In its turn, the sending process must

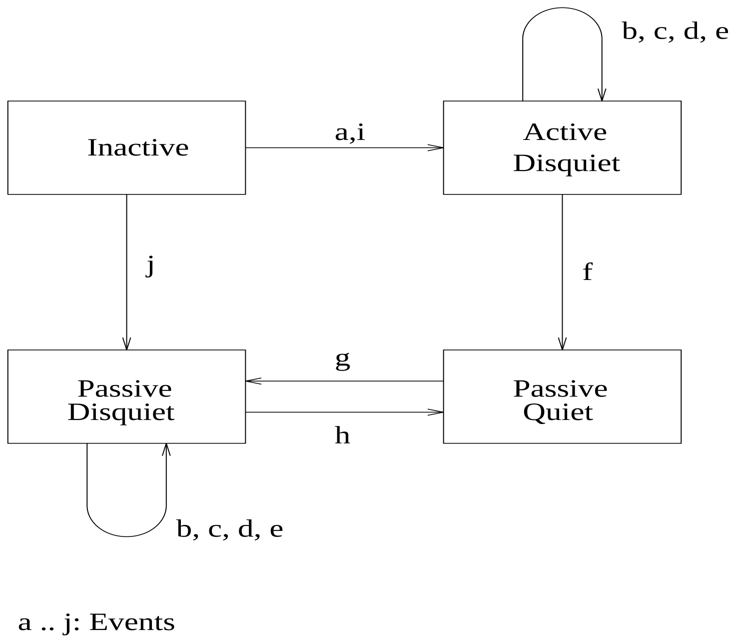


Figure 15: State transition diagram for termination detection.

be a disquiet process because it has not received all the acknowledgements for every mark request it has generated. It may have a process responsible for it or it maybe the responsible process itself, if it has initiated the phase, that is, if it is an active-disquiet process. Clearly a process may be switched from quiet to disquiet a number of times before termination is detected, and on each occasion it may have a different process responsible for it. Given that only a disquiet process may send a mark request, this scheme ensures that, if any process is disquiet, there is at least one process which is active-disquiet.

The stability condition that must be satisfied simultaneously in every process X is thus:

$$X \text{ is not active-disquiet;}$$

We now define formally the events that model the two phases and the termination protocol. The local state of a process p_i with respect to termination is defined by the following state variables:

- $p_i.condition \in \{dynamic, non\text{-}dynamic\}$.

- $p_i.state \in \{inactive, active\text{-}disquiet, passive\text{-}quiet, passive\text{-}disquiet\}$.

- p_i.*grey-set*: Set of exit-item; *grey-set* $= \emptyset$ iff all acknowledgements have been received; an exit-item recorded in *grey-set* represents an unacknowledged request.
- p_i.*local-steps*: Integer; *local-steps* $= 0$ iff there are no local steps to perform.
- p_i.*reply-set*: Set of exit-item; an exit-item recorded in *reply-set* represents the exit-item to be acknowledged.
- $p_i.parent \in \{self, other, none\}$.
  - *self* : p_i is responsible for mark requests it has generated;
  - *other*(p_j) : there is $p_j (i \neq j)$ such that p_j has sent a mark request to p_i and p_i has not yet acknowledged such request;
  - *none* : p_i is passive-quiet.

Before defining the invariant that must hold in each state in order to detect termination, we first make the following definition:

Definition 4.3 *The relation Ancestor(p_j, p_i) holds for two processes p_j and p_i if and only if* $p_i.\text{parent} = p_j$.

Now, consider $Ancestor^*$ as the reflexive transitive closure of relation $Ancestor$, that is $Ancestor^*(p_j, p_i)$ holds if and only if one of the following holds:

- $Ancestor(p_j, p_i)$
- or there is p_k such that $Ancestor(p_k, p_i)$ and $Ancestor^*(p_j, p_k)$;

Invariant 4.4 then holds:

Invariant 4.4

$$\forall p_i \cdot p_i.\text{state} = \text{passive-disquiet} \Rightarrow$$
$$\exists p_j.(i \neq j) \wedge \text{Ancestor}^*(p_j, p_i) \wedge p_j.\text{state} = \text{active-disquiet}$$

This invariant means that if there is a *passive-disquiet* process (and hence termination is not achieved) there must be an *active-disquiet* process. From this it follows that if no process is *active-disquiet*, the phase has terminated.

	Event	Action
a	start-phase	• initialise *local-steps* • $parent = self$ • $state =$ active-disquiet
b	send *mark request*(*exit-item*)	• $grey\text{-}set = grey\text{-}set \cup \{exit\text{-}item\}$
c	receive *mark request*(*exit-item*) from process p	• $local\text{-}steps++$ • send *acknowledgement*(*exit-item*) to p
d	receive *acknowledgement*(*exit-item*)	• $grey\text{-}set = grey\text{-}set \setminus \{exit\text{-}item\}$
e	perform *local step*	• $local\text{-}steps--$
f	$grey\text{-}set = \emptyset \wedge local\text{-}steps = 0$	• $parent = none$ • $state =$ passive-quiet
g	receive *mark request*(*exit-item*) from process p	• $local\text{-}steps++$ • $parent = other(p)$ • $reply\text{-}set = reply\text{-}set \cup exit\text{-}item$ • $state =$ passive-disquiet
h	$grey\text{-}set = \emptyset \wedge local\text{-}steps = 0$	• for exit-item in *reply-set* send *acknowledgement*(*exit-item*) to *parent* • $parent = none$ • $state =$ passive-quiet
i	receive *mark request*(*exit-item*) from process $p \wedge$ **dynamic**	• initialise *local-steps* • set $parent = self$ • send *acknowledgement*(*exit-item*) to p • $state =$ active-disquiet
j	receive *mark request*(*exit-item*) from process $p \wedge$ **non-dynamic**	• $local\text{-}steps = 1$ • $parent = other(p)$ • $reply\text{-}set = reply\text{-}set \cup exit\text{-}item$ • $state =$ passive-disquiet

Figure 16: State changes for termination detection

4.5.2 Report phase

We have presented a solution that allows a process initiating a distributed computation to detect termination of such computation. Next we complete our distributed termination protocol describing a report phase for both mark-red and scan phases.

Mark-red

The mark-red phase is initiated by a single process, hence in a mark-red phase instantiation there is only one **dynamic** process and consequently there is only one active-disquiet process. As soon as the initiating process turns to passive-quiet, the mark-red phase is complete. Processes join the mark-red phase when they receive a mark-red request. These processes are **non-dynamic** because they do not generate mark-red requests on their own account. Consequently their first state transition is to passive-disquiet.

In order to proceed to the scan phase, the initiator needs to report the end of the mark-red phase to processes that were involved in it — the *participants*. For the initiator to know the group of participants, each participant appends its identity, and the identity of those processes to which it has sent requests, to the acknowledgement of a mark-request. This feature is the key to our opportunistic scheme for identifying the suspect subgraph dynamically. Initially, it is not necessary to know which processes will be involved in a partial tracing. Consequently, our system guarantees that only processes involved in a partial tracing will co-operate in garbage cycle collection. This approximates the property of locality.

Scan

The scan phase starts concurrently in each process holding a part of the suspect subgraph (recall section 4.4). In this case, all participants are **dynamic** because they will generate mark-green requests independently through the local initial-step. Consequently, all participants will be active-disquiet at the begining of this phase.

A process will turn to active-disquiet when it receives the report message from the Initiator. However, a process may receive a scan request from another process that has already entered the scan phase, without having received the report message itself. In this case, it will enter the scan phase and become active-disquiet.

Following our termination protocol, each participant may enter the sweep phase when it knows that every participant has changed to passive-quiet. The protocol requires every participant to inform the initiator process when it changes from active-disquiet to passive-quiet. In its turn, the initiator process will inform the participants of the end of scan phase after having received the state change information from all the participants and after it has changed from active-disquiet to passive-quiet itself.

4.6 Heuristics

The basic partial tracing algorithm we have described in the last three sections presents two potential challenges in order to provide efficiency, scalability and promptness:

1. Which objects should be suspects? Suspects should be chosen with care both to maximise the amount of garbage reclaimed and to minimise redundant computation or communication.
2. What should be the extent of mark-red? Limiting the extent of mark-red to just garbage items would make our algorithm preserve the property of locality and improve promptness.

4.6.1 Heuristics for Suspect Objects

The global cost of our algorithm depends on how frequently it is run. In particular, acyclic garbage will be collected by the acyclic collector, so the greater the delay, the more likely that acyclic garbage will have collected itself. Also, repeated and wasted work would be minimised if our algorithm did not work on live objects.

Until now we have adopted the *Locally Reachable* heuristic. But, as we have already said, this heuristic is very simplistic and may lead to undesirable wasted and repeated work. It may repeatedly identify an object as a suspect even though it is reachable from a remote root. Rather, our algorithm should be seen as a framework: any better heuristic could be used.

Heuristics for finding objects determine on what extent our algorithm approximates the property of locality. The closer we approximate this property, the better the probability of only triggering a distributed garbage collection on garbage objects. In this way we reduce the frequency with which our algorithm is run and reduce the overhead of a particular partial tracing by minimising the number of scan requests. We also benefit from the fact that mutators do not work on garbage objects, hence reducing the synchronisation actions overhead.

The distance heuristic described in section 3.5 is suitable for finding suspect objects, because it allows the identification of objects belonging to a garbage cycle with a high probability of being correct. This increases the probability of a partial tracing working in a garbage subgraph. Simpler heuristics may be used in conjunction with the "Generational Heuristic" (Rodriguez-Riviera and Russo 1997). Instead of starting a partial tracing every time an object is found suspect, we start only at those suspect objects that have not been subject to a distributed collection recently. This reduces the number of times the algorithm is run.

We may also locally decide between grouping the suspect objects in one partial tracing and tracing independently from each suspect object. At first sight, the second choice would provide better promptness because it would involve a smaller group of processes and objects. However, with the current solution, the collection of a garbage cycle would be compromised if different suspects in the same cycle start an independent collection.

In section 9.1 we analyse the cost of our algorithm.

4.6.2 How far to go?

The mark-red trace may include more processes than necessary because a garbage cycle may point to chains of garbage or live objects. Hence, a practical requirement on the mark-red phase is to limit its spread to suspect objects. In this way, we may avoid a mark-red trace from spreading to live objects by using the same heuristic that chose suspect objects. This restricts the number of red objects that might to be rescued by the scan phase. Recall figure 13 on page 89. Note that object h in process D is locally reachable, hence live. Based on this information, the mark-red trace should terminate at Ei_h: the represented garbage cycle points to a chain of live objects.

The aim of the mark-red phase is to mark red a subgraph suspected of belonging to a garbage cycle. It does not make any decision about objects' liveness. Consequently, the red subgraph need not include the whole set of garbage objects. It suffices that the red subgraph includes only a subset of the set of garbage objects sufficient to make progress— *i.e.* a *conservative* approximation[4]. Indeed, early termination of this phase trades *conservatism* (tolerance of floating garbage) for expediency, and bounds on the size of the graph traced, and hence, as before, on the cost of the trace.

This policy decision can be taken statically by prior negotiation or dynamically by mark-red. It may be determined by the collector itself or by the user program, globally or on a per-process or even per-object basis. Heuristics based on geography, process identity, distance from the suspect originating the collection, minimum distance from any object known to be live, or time constraints may be used to restrict the extent of mark-red.

In section 9.1 we make a qualitative analysis of our algorithm based on which heuristics are chosen for suspect identification and mark-red phase extent.

4.7 Summary

We have described a basic algorithm for garbage collection on distributed large address spaces that is scalable, efficient and fault-tolerant, albeit not complete.

It combines the reference listing scheme with an incremental, three-phase, partial tracing to reclaim distributed garbage cycles. Our algorithm operates in three-phases. The first, mark-red, phase identifies a distributed subgraph that may be garbage, to which subsequent efforts are confined. The mark-red phase also dynamically identifies groups of processes that will collaborate to reclaim cyclic distributed garbage. The second, scan, phase determines whether members of this subgraph are actually garbage. Finally the sweep phase makes any garbage objects available for reclamation by local collectors.

Fault-tolerance and efficiency are achieved by requiring the co-operation of only those processes forming the group: progress can be made even if other processes in the system fail. Global synchronisation is avoided by partitioning the distributed system into groups, with multiple groups simultaneously but independently active for garbage collection: communication is only necessary between members of the group.

[4]Equally it does not matter if we mark too much.

Moreover, two kind of heuristics are defined, whose goals are to improve the algorithm's discrimination and hence its efficiency: heuristics for suspect identification that try to maximise the amount of garbage reclaimed and to minimise redundant computation or communication; and heuristics for the extent of mark-red that try to approximate the property of locality and improve promptness.

Chapter 5

A Scalable Cyclic Garbage Collector

In this chapter we identify some deficiencies of the solution presented in the previous chapter. We give a detailed description of techniques for improve scalability. There are two aspects of concurrency: collector/collector and mutator/collector concurrency. The first is concerned with scalability and the second with efficiency. We address the scalability aspect in this chapter and leave mutator/collector concurrency to the next chapter.

5.1 Scalability

As we have already stated in section 4.2.1, the property of locality is the key to scalability. Whether our solution approximates this property depends primarily on the heuristic for finding suspect objects. More accurate heuristics give better approximations as we explained in section 4.6. However, the solution we have presented in the previous chapter has deficiencies in scalability and completeness in large address spaces.

In practice, in a large address space, there will be multiple suspect entry-items, which may generate numerous concurrent and/or overlapping partial tracings, *i.e.*, several partial tracings may be triggered concurrently at the same or different processes and/or multiple partial tracings may be active on entry-items in the same cycle.

Until now we have ignored concurrency between different partial tracings. A first solution might require a partial tracing to retreat when it meets a different partial

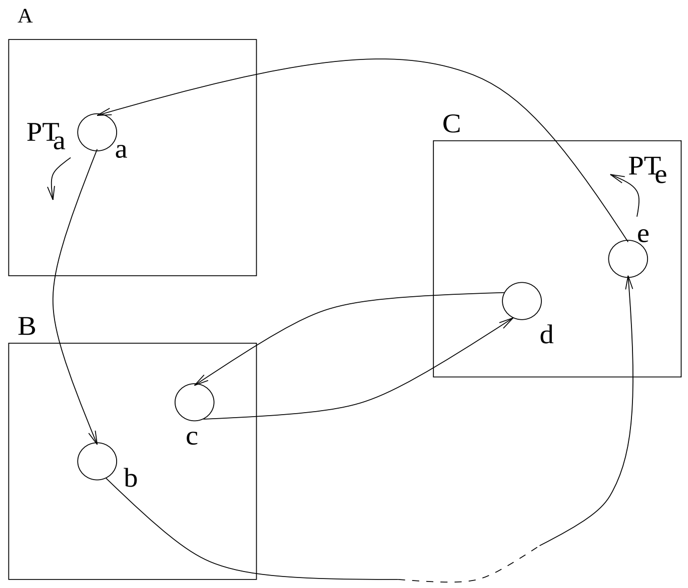


Figure 17: Multiple Partial Tracings

tracing or involves a process already participating in a different partial tracing. This will not achieve the completeness goals we presented in section 4.2.1:

- Retreating introduces the problem of live-lock: different partial tracings may indefinitely interrupt or restart, but never complete, preventing the collection of cyclic garbage.
- The collection of disjoint cycles is compromised. The collection of a larger cycle may delay the collection of small cycles spread across the same processes.

These two situations are shown in figure 17. Cycle ($a \rightarrow b \rightarrow \ldots \rightarrow e \rightarrow a$) would not be collected if partial tracings a and e start and retreat indefinitely. Also, collection of the cycle ($c \rightarrow d \rightarrow c$) could be delayed until the end of the larger cycle collection. In section 5.3 we will describe techniques for allowing partial tracings to co-operate in the collection of overlapping cycles. It will be shown in the next sections that our solution does not always ensure completeness. However, although our first design choice is to trade completeness for promptness, our solution does have the potential to be complete at the cost of stronger synchronisation (see section 9.1).

Recall the two problems arising from dependent cycles introduced in section 4.2.1. Concurrency of partial tracings is directly related to completeness: concurrent and independent partial tracings could co-operate in the collection of cycles A and B in figure 12 on page 85. Moreover, ideally a partial tracing should succeed independently of garbage cycles' dependencies.

We would benefit from a solution allowing disjoint partial tracings to proceed independently, ignoring events associated with a partial tracing with a different identifier. In this case, the mark-red phase would produce disjoint red closures. For example, the garbage cycle ($c \rightarrow d \rightarrow c$) in figure 17 could be collected independently from the larger one.

A different situation arises if different partial tracings are simultaneously active in the same suspect cycle, or in connected cycles. This means that they would eventually meet in an entry or exit-item, and interfere with each other. Consider figure 17. PT_a and PT_e are simultaneously active in the same suspect cycle. PT_a eventually reaches entry-item Ei_e and PT_e eventually reaches entry-item Ei_a (entry items are not represented in the figure for simplicity). In this situation, as we have already said, live-lock must be avoided, hence any retreat should be avoided.

We offer two solutions for partial tracings that may overlap:

Overlapping partial tracings The different partial tracings are allowed to proceed concurrently and independently in every element of the suspect subgraph. Each partial tracing would ignore any another. In effect, the partial tracings retain their own identity but *overlap*.

Co-operative partial tracings Overlapping partial tracings are allowed to proceed and co-operate in collection of garbage cycles. A partial tracing working in a part of such a subgraph may contribute to partial tracings working in other parts of the same subgraph. The result may be seen as the union of every partial tracing active in the same subgraph. However, this would mean that every partial tracing's phase termination would be dependent on other partial tracings's phase termination and every event of each partial tracing would contribute in some way to an event of the other partial tracing.

A solution for overlapping partial tracings would be essentially the same as presented in the last chapter. This requires that the partial tracings do not share any state (the colour and red-list information held in the entry and exit-tables). This could be achieved

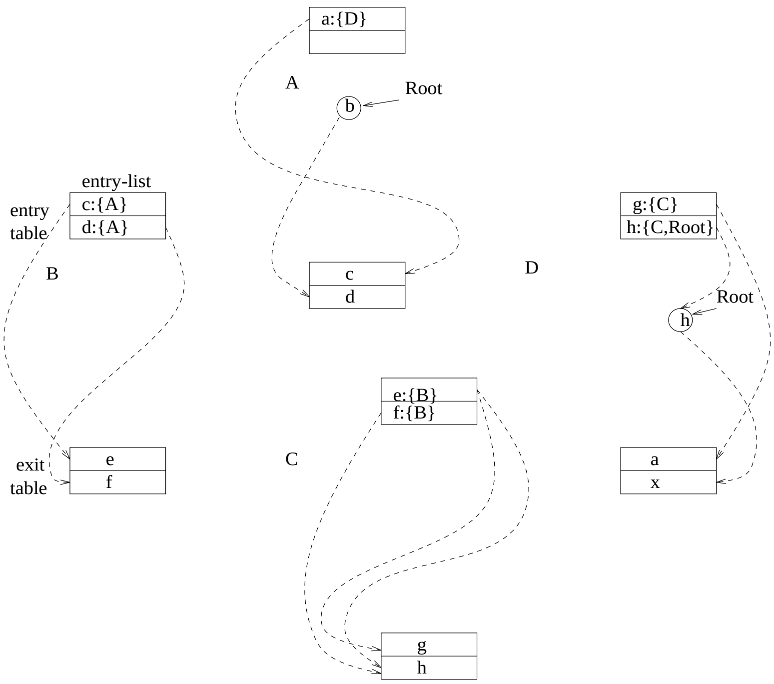


Figure 18: Entry-item/Exit-item reachability

by maintaining a copy of this state information for each partial tracing, and have all garbage collection messages signed with the identity of their partial tracing. The obvious drawback is that, while it is scalable and complete, it is neither time- nor space-efficient as it leads to repeated work.

Let us concentrate on the solution based on co-operative work. First we describe the target graph for partial tracings in section 5.2; our aim is to implement efficient mark-red and scan local and remote steps, and scan initial steps, and to decrease space overhead. Then we describe a solution that accounts for multiple partial tracings; we define mark-red and scan phase' steps, the information that is necessary for each partial tracing to proceed when there is some co-operation between partial tracings, and how that information is built during a run. We also describe the new conditions for termination and how they are achieved by every co-operative partial tracing — the report phase.

5.2 *Cut-references* Graph

Multiple overlapping partial tracings require every object traced to be signed with the identifier of the partial tracing it is involved with, and coloured. This would lead to a considerable space overhead. As we mentioned in section 4.1, we can always determine reachability between entry and exit-items. We illustrate this idea, with respect to the figure 11 on page 82, in figure 18: the entry and exit-items of each site taken together form a distributed graph that we name the *cut-references* graph; every entry and exit-item becomes a vertex of this graph. Every outgoing path from an entry-item, which reaches an exit-item (via some local objects), becomes a single edge in the same graph ((Maheshwari and Liskov 1997a)). A remote edge is represented by the correspondence entry-item/exit-item, for example exit-item(c)/entry-item(c). Notice that we are only interested in suspect items, so we just require such information for those items. In this figure we show the *cut-references* of suspect items. The *cut-reference* graph is connected to the ordinary graph through ordinary references. For example, entry-item Ei_d is reachable from the local root at process A. Inter-process garbage collection can therefore be described as performing garbage collection of the *cut-references* graph. If two partial tracings are to overlap, they will will encounter each other in a common entry or exit-item.

Now, multiple partial tracings only require entry and exit-items to be signed and coloured. We benefit because both less space is needed and because local steps are cheaper, as explained later (this solution will introduce more constraints on dealing with mutator concurrency as we will show in section 6).

We now describe the computation of the *cut-references graph*. For now we consider suspect items to be defined by the *Local reachability* heuristic. Notice that, independently of which heuristic we use, a necessary condition is that those suspect objects are not locally reachable. A better heuristic would only reduce the number of suspect items. The requirement that suspect entry and exit-items must not be locally reachable still holds.

The computation of the *cut-reference* graph may be performed at any time. However because the mutator actions may change the reachability of entry and exit-items, the more often the *cut-reference* graph is computed, the more accurate it will be.

We divide the computation of the *cut-references graph* into two phases. In the first phase, we require a method that identifies suspect objects. That is, this method must determine whether objects are reachable locally. This will inform our system about which entry and exit-items are suspect. In the second phase, we compute the *cut-reference* graph, that is, which suspect exit-items are reachable from each suspect entry-item.

The first method, **identify-suspects**, can be implemented by tracing from the local roots in each process, excluding objects only reachable remotely (as we describe in section 8, this may be done by the local collector). After the **identify-suspects** method, the following post-condition holds:

Post-condition 5.1

$$[\textbf{identify-suspects}]$$
$$\{(\forall Ei_z \cdot \text{suspect}(Ei_z) \Rightarrow \neg\text{path}(Roots, z)) \wedge$$
$$(\forall Ex_y \cdot \text{suspect}(Ex_y) \Rightarrow \neg\text{path}(Roots, Ex_y))\}$$

After **identify-suspects**, entry and exit-items are suspect if and only if they are not reachable locally. For every suspect entry-item Ei_z, a second method, **compute-graph**, computes the list of all suspect exit-items Ex_y recursively reachable from Ei_z. This information is recorded in $Ei_z.exits$. In the absence of mutator concurrency the following post-condition is always true.

Post-condition 5.2

$$[\textbf{identify-suspects}]$$
$$\{(\forall Ei_z \cdot \text{suspect}(Ei_z) \Rightarrow \neg\text{path}(Roots, z)) \wedge$$
$$(\forall Ex_y \cdot \text{suspect}(Ex_y) \Rightarrow \neg\text{path}(Roots, Ex_y))\}$$
$$[\textbf{compute-graph}]$$
$$\{\forall Ei_y, Ex_z \cdot (\text{suspect}(Ei_y) \wedge \text{suspect}(Ex_z) \wedge \text{path}(Ei_y, Ex_z)) \Rightarrow Ex_z \in Ei_y.\text{exits}\}$$

However, as we said, due to mutator activity between two computations of the *cut-references* graph, suspect information may change. In this case, we say that the *cut-references* graph is not accurate.

The *cut-reference* graph represented in figure 18 would be described by the following set of variables:

$$
\begin{aligned}
Ei_a.exits &= \{Ex_c\} \\
Ei_c.exits &= \{Ex_e\} \\
Ei_e.exits &= \{Ex_g, Ex_h\} \\
Ei_g.exits &= \{Ex_a\}
\end{aligned}
$$

Additionally we define three further components of each entry and exit-item:

mark holds the identifier (see below) of the first active partial tracing to reach that entry or exit-item.

marks holds the identifiers of other partial tracings simultaneously active in the same entry or exit-item.

colour holds the colour of the entry or exit-item. An entry or exit-item not involved on any partial tracing is white. The mark-red phase paints suspect items red, and the scan phase paints all live items green.

5.3 Multiple Partial Tracings

Partial tracings simultaneously active in the same suspect cycle or connected cycles may meet each other during each other's garbage collection cycle. We aim at defining a co-operation between overlapping partial tracings. If the mark-red and scan phases of simultaneous and connected partial tracings were to overlap, they may never terminate because of race conditions between mark-red and scan requests. Thus, these partial tracings are not allowed to proceed if they meet for the first time in different phases. We only allow different partial tracings to co-operate when they meet at an entry or exit-item for the first time in the mark-red phase. When they meet, the protocol for co-operation will be established. Care must be taken to ensure that co-operative partial tracings do not interfere with each other, *i.e* for the same reason as above, the different

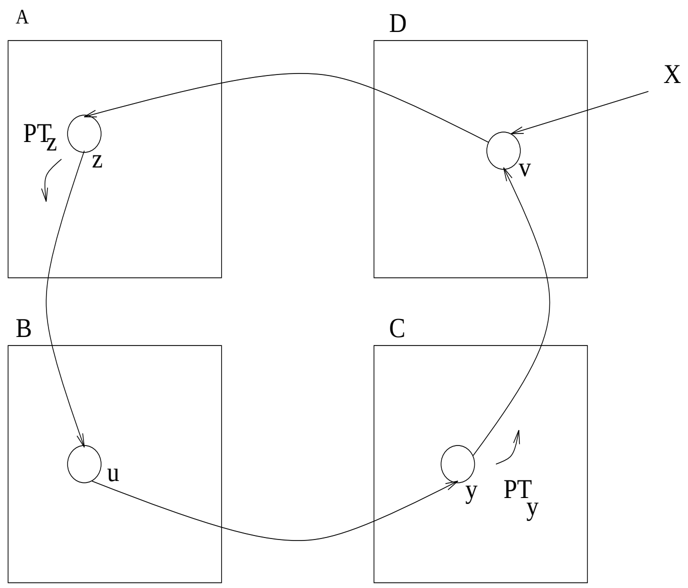


Figure 19: Multiple Partial Tracings Co-operation

phases of each partial tracing should not overlap. This co-operation is informally defined next.

Co-operation between simultaneously active partial tracings relies on the fact that each partial tracing has enough information to proceed. When the mark-red phase of a partial tracing PT_y meets the mark-red phase of another PT_z, through a mark-red step, we will say that PT_z is *dependent* on PT_y and PT_y is *responsible* for PT_z. The red sub-graph defined by the responsible partial tracing will be conceptually merged with the dependent tracing's red sub-graph. Consequently, in subsequent phases, a dependent partial tracing will not consider references to its red sub-graph from a responsible tracing's red sub-graph to be external references. For that to be possible, red entry-items with the field *mark* holding the responsible tracing's identifier are not considered roots for the dependent's initial scan step. Moreover, in order for any remote reference, from a red exit-item with the field *mark* holding the responsible tracing's identifier, to be considered a reference internal to the dependent tracing's red subgraph, it must have the source process inserted in the corresponding entry-item in the target process. For that, by the time a mark-red remote step was taken, it should have inserted the sending process's identifier into the target entry-item's red list, even if that item already belongs to another partial tracing. In this way a reference from the responsible tracing's red sub-graph will be considered to be an internal reference.

Co-operation must be extended to the scan phase. Consider the simple example in figure 19. Suppose that the sub-graphs were merged and PT_z finishes its scan phase and proceeds to the sweep phase without any co-operation from PT_y. Consequently, PT_z does not count red external references to its red sub-graph from PT_y's red sub-graph. Consider now the external reference from X, which would cause the entry-item for v to be repainted green during PT_y's scan phase. Consequently, PT_z's subgraph should be repainted green. If PT_z has proceeded to the sweep phase before PT_y has completed its scan phase, live objects would be reclaimed unsafely. We conclude that PT_z must wait for PT_y's scan phase to terminate before it proceeds to the sweep phase.

If partial tracings do meet for the first time in different phases, they should retreat. Notice that we want to avoid this situation. Partial tracings simultaneously active on the same cycle or connected cycles are dependent on each other. This means that dependent partial tracings will fail to collect garbage as we explained in section 4.2.1. This situation is particularly undesirable for partial tracings simultaneously active in the same sub-graph because, if we do not synchronise the beginning of each scan phase, completeness is compromised. In the example of figure 19, suppose that PT_z encounters PT_y at y, defines the required co-operation and returns. If PT_z starts the scan phase before PT_y's mark-red reaches object z, PT_y will retreat and the two partial tracings will fail. A similar situation may occur in two connected cycles, A and B, if the dependent cycle, for example A, finishes its mark-red phase before B's mark-red has encountered it. The partial tracing active on A will fail. However, in this situation we do not compromise completeness, because a partial tracing at B would eventually collect B or meet A.

5.3.1 Initiating a partial tracing

We now define the behaviour of the collector more precisely. A suspect entry-item Ei_z at process P may initiate a partial tracing if it is not already involved in another one. We call process P the initiator process. Recall that, during the mark-red phase, a red sub-graph is formed by entry and exit-items identified by that partial tracing. A group of processes that we call the participants is also formed. While in this phase, a partial tracing may meet other partial tracings and establish a dependency/responsibility relationship. In this case they are called *co-operative* partial tracings. For every partial tracing, we capture this information in an object of type *PTobj*.

Definition 5.3

$$PTobj = (Id,\ Initiator,\ Participants,\ EI,\ EX,\ Dependents,\ Responsibles)$$

where the components are defined as follows:

Id is a unique identifier. A partial tracing will be identified by the starting suspect entry-item. We consider that entry-items are unique and identify the initiator process[1]. We use the notation PT_z for Ei_z.

Initiator is the initiator process.

Participants is the set of the partial tracing's members (recall section 4.5.2).

EI is the set of entry-items such that $Ei.mark = Id$ or $Id \in Ei.marks$.

EX is the set of exit-items such that $Ex.mark = Id$ *or* $Id \in Ex.marks$.

Dependents is the set of partial tracings that are dependent on this partial tracing.

Responsibles is the set of partial tracings that are responsible for this partial tracing.

A partial tracing can be defined by the tuple on definition 5.3. This partial tracing has images (approximations) in each participant. When a partial tracing with identity z — PT_z — visits a participant process P for the first time in a collection cycle, it constructs a new partial tracing object of type $PTobj$ — pto_{Pz}. The partial tracing images have concrete representations as those partial tracing objects. The whole partial tracing information is distributed across PT_z's images in each participant.

There is a time, as we see below, that the whole partial tracing information must be known by the initiator. We defined the partial tracing value as the union of every image in each participant.

For $P, Q \in PT_z.Participants$ and the corresponding pto_{Pz} and pto_{Qz}, we define the union of the two images as $union(pto_{Pz}, pto_{Qz})$.

[1]If we are considering network partitions, the pair (entry-item,collection number) could be used for identifying each partial tracing.

Definition 5.4 union$(pto_{Pz}, pto_{Qz}) = pto_{Rz}$

Where,

- $pto_{Rz}.Id = pto_{Pz}.Id = pto_{Qz}.Id$
- $pto_{Rz}.Initiator = pto_{Pz}.Initiator = pto_{Qz}.Initiator$
- $pto_{Rz}.Participants$ is not defined because $PT_z.Participants$ is determined at the end of mark-red phase at $PT_z.Initiator.$
- $pto_{Rz}.EI = pto_{Pz}.EI \;\cup\; pto_{Qz}.EI$
- $pto_{Rz}.EX = pto_{Pz}.EX \;\cup\; pto_{Qz}.EX$
- $pto_{Rz}.Dependents = pto_{Pz}.Dependents \cup pto_{Qz}.Dependents$
- $pto_{Rz}.Responsibles = pto_{Pz}.Responsibles \cup pto_{Qz}.Responsibles$

The whole partial tracing value is defined as:

$$PT_z = \bigcup_{P \in PT_z.Participants} pto_{Pz}$$

During a run (mark-red $\longrightarrow$ scan $\longrightarrow$ sweep), most communication between partial tracings is handled through the local partial tracing objects and does not require remote communication.

For the present, we are not interested in when the union of the partial tracing objects in every participant will be effectively performed, we will come back to this later. Instead, next we describe the different phases of a partial tracing, accounting for co-operative partial tracings: how a partial tracing object is constructed in every phase of a partial tracing and which information different partial tracings need to exchange between themselves and their participants in order to proceed to the next phase.

From now on, we identify a partial tracing object PT_z on each participant by using the notation PT_z. When relevant, instead of using PT_z, we may use the image of PT_z in a particular participant P — pto_{Pz}.

5.3.2 Mark-red Phase

Recall the mark-red steps from section 4.3.1. We now redefine the mark-red steps for PT_y in order to account for co-operative partial tracings. Every participant executes alternate local (ML) and remote steps (MR), colouring items that it reaches. It performs a local mark-red step from each entry-item Ei_a newly marked red, where $Ei_a.mark = PT_y$, to each exit-item Ex_b in $Ei_a.exits$ as follows:

(ML.1) If Ex_b is white, then it is reddened and its mark set to PT_y, that is, $Ex_b.mark = PT_y$: we call Ex_b red_y.

(ML.2) If Ex_b is already red_y, then no further action is necessary.

(ML.3) If Ex_b is red_z where $z \neq y$, then two partial tracings have met in the same phase. We merge the partial tracings and say that z is *dependent* on y and y is conversely *responsible* for z. PT_y is appended to $Ex_b.marks$, PT_y is added to the $PT_z.Responsibles$, and PT_z to the $PT_y.Dependents$. Both these interactions take place between the partial tracing objects in this process — no messages are sent.

(ML.4) If Ex_b is green, it must have been marked by another group operating in a later phase so the red wave-front retreats from this object.

A remote step executed by PT_y propagates a colour from an exit-item Ex_b in a participant P to entry-items Ei_b in a remote process Q. A new PT_y image pto_{Qy} is constructed in Q to represent this partial tracing (unless one already exists for this partial tracing as a result of an earlier mark-red request in this collection cycle).

(MR.1) If Ei_b is white or is red_y, P is added to $Ei_b.red\text{-}list$ and Ei_b is marked red_y.

(MR.2) If Ei_b is red_z and $z \neq y$, P is still added to $Ei_b.red\text{-}list$. Once again two partial tracings have met and, as in the local step, PT_y is appended to $Ei_b.marks$ and to $PT_z.Responsibles$, PT_z to $PT_y.Dependents$ in process Q; no messages are exchanged.

(MR.3) If Ei_b is green, no further action is taken and the mark-red phase retreats.

Figure 20 shows an example in which two objects, y in process A and z in process D, have initiated independent distributed collections which have met at Ei_u in process

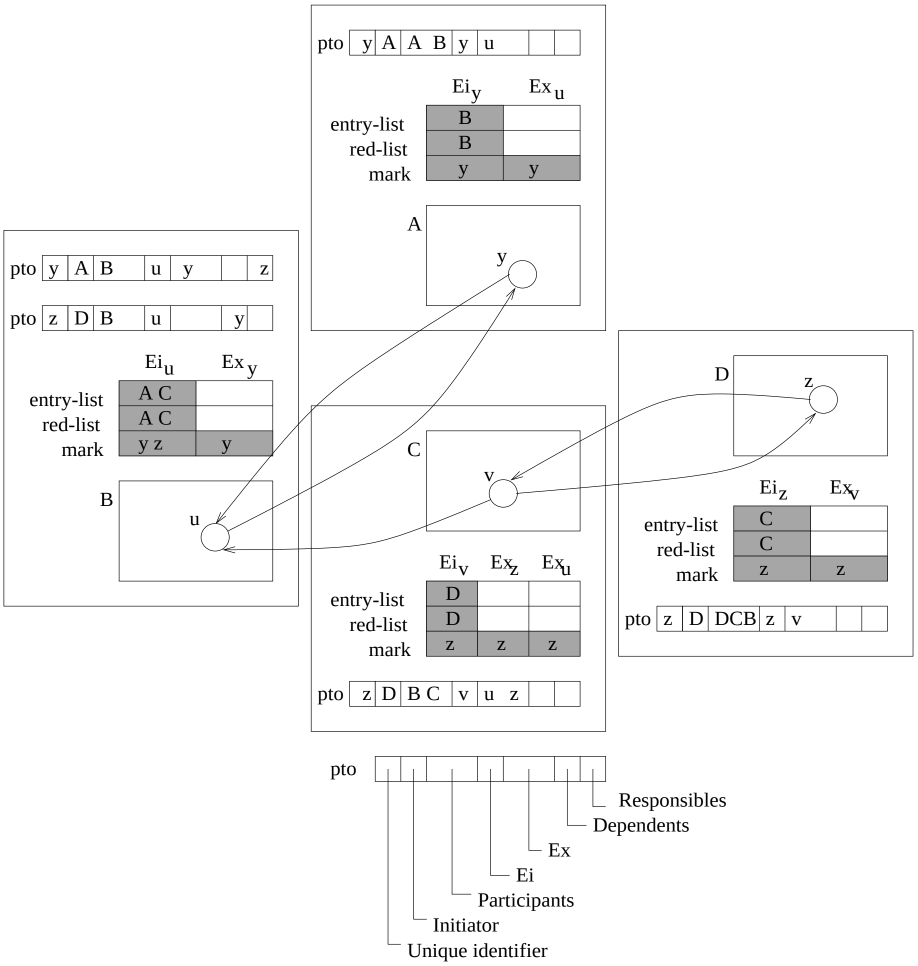


Figure 20: End of the mark-red phase

B. Note that process B contains pto's for both partial tracings PT_y and PT_z. When PT_z performs the remote step from Ex_u at C to Ei_u at B, the following action is taken at B: $PT_y.Responsibles = PT_y.Responsibles \cup \{PT_z\}$ and $PT_z.Dependents = PT_z.Dependents \cup \{PT_y\}$. More precisely, $pto_{By}.Responsibles = pto_{By}.Responsibles \cup \{PT_z\}$ and $pto_{Bz}.Dependents = pto_{Bz}.Dependents \cup \{PT_y\}$.

Mark-red Termination

As we described in section 4.5, termination of the mark-red phase for a single partial tracing PT_y is detected by its initiator when it receives acknowledgements for all the mark-requests it has generated. At this time, the termination protocol described in section 4.5 guarantees that no item in $PT_y.EI \cup PT_y.EX$ will receive a mark-red request generated by PT_y itself. Those items may only receive mark-red requests generated by other partial tracings trying to enter some co-operation. If the corresponding participant had already entered the scan phase, such requests may be safely refused (ML.4 and MR.3) because the mark-red phase does not need to visit the complete referential transitive closure of a suspect object. Otherwise, co-operation between the two partial tracings would be engaged and the mark-request properly acknowledged.

We conclude that termination of the mark-red phase of co-operative partial tracings are independent of each other. Consequently, we can detect mark-red termination as described in section 4.5.

5.3.3 Scan Phase

Recall the definitions of scan steps and the *local-scan-root-set* in section 4.4.1. We redefine them now to account for co-operative partial tracings. Any partial tracing recorded in $PT_y.Responsibles$ must co-operate with PT_y. A red_y-subgraph is alive if it is accessible from a root or from outside the merged subgraphs. The scan phase must therefore take into account the scan-requests generated by every co-operative partial tracing.

Each partial tracing determines the liveness of its own red subgraph. However, it also has to take into account requests from co-operative partial tracings. We present a set of rules that must be obeyed by each participant of a partial tracing in order to proceed to the scan phase. These rules ensure safety and termination of mark tracings.

However, they do not of themselves ensure completeness as we have already mentioned in section 5.3. Nevertheless, as we discuss in section 9.1, our systems do allow a complete solution.

1 A process $P \in PT_y.Participants$ may safely enter the scan phase when it receives the report message (recall section 4.5) from $PT_y.Initiator$. Recall that the acknowledgement system allows the Initiator to know the identity of each participant.

2 When $P \in PT_y.Participants$ receives a scan request from PT_z where $PT_z \in PT_y.Responsibles$, it may safely green the target entry-item. However it may not have received the report message yet from $PT_y.Initiator$. By rule 1, it should wait for that message before it enters the scan phase. To avoid race conditions between PT_y mark-red requests and the responsible process's scan requests, a mark-red only paints red white entry-items.

3 The *local-scan-root-set*(PT_y) in each $P \in PT_y.Participants$ is formed by:

- P's local roots, as suspect information may have changed since the last time it was computed.
- white entry-items, as they do not belong to the suspect sub-graph,
- green entry-items, as they have already been found to be live,
- any red entry-item marked by either PT_y or $PT_u \in PT_y.Responsibles$ whose entry and red-lists differ, as they are reachable from outside the suspect sub-graph, and
- any other red entry-item marked by another $PT_u \notin PT_y.Responsibles$, as they are not part of PT_y's suspect subgraph. Recall that $PT_u \in PT_y.Responsibles$ is a co-operative partial tracing. Thus, references from the PT_u's suspect sub-graph are not considered as external references to PT_y's suspect sub-graph.

Again, after an initial step (SI) to colour green any entry or exit-item reachable from the *local-scan-root-set*, the scan phase proceeds by an alternating series of local (SL) and remote scan steps (SR). The initial scan step of each PT_y greens any objects directly reachable from the *local-scan-root-set* that PT_y had previously visited: these will be the starting points for the 'rescue' trace.

(SI.1) Mark green any red entry or exit-item E in the *local-scan-root-set* for which $E.mark = PT_y$. Mark green any red exit-item Ex_b for which $Ex_b.mark = PT_y$ and which is reachable from the *local-scan-root-set*. These are $green_y$.

The local scan phase step for PT_y propagates the green colour from a $green_y$ entry-item Ei_a to those exit-items Ex_b in the same process reachable from Ei_a that PT_y had previously visited in the mark-red phase:

(SL.1) Green Ex_b if it is red, reachable from a green Ei_a, and either $Ex_b.mark = PT_y$ or $PT_y \in Ex_b.marks$. That is, we green only those exit-items reddened by co-operative partial tracings.

The remote step from a green exit-item Ex_b propagates the green colour to the corresponding entry-item Ei_b:

(SR.1) If Ei_b is red and $Ei_b.mark = PT_y$ or $PT_y \in Ei_b.marks$, mark Ei_b green.

(SR.2) If Ei_b is red but neither $Ei_b.mark = PT_y$ nor $PT_y \in Ei_b.marks$, retreat.

(SR.3) If Ei_b is not red, retreat.

(SR.4) Request a local step from every greened Ei_b if $Ei_b.mark$ has entered the scan phase.

Remote steps do not invoke local steps directly. Rather, the partial tracing object that 'owns' the entry-item (identified by its *mark*) will execute a local step once it has started its scan phase. Note that an entry-item may be part of more than one partial tracing (if the length of its *marks* list is greater than one). If a partial tracing receives a scan-request before it receives the instruction to start the scan phase, it simply marks the entry-item green but does not yet take a local step[2].

[2]Actually, the greening operation may be queued up instead of being taken immediately. It may be better to leave the item red, as it can accept more mark-red requests, leaving more opportunities for co-operation.

A PT_y may only proceed to the sweep phase when it has finished the scan phase and when every co-operative partial tracing in $PT_y.Responsibles$ has also finished its scan phase. Next we explain how the termination protocol is modified to cope with co-operative partial tracings.

Scan Termination

Intuitively, we may conclude that the scan phase of a partial tracing cannot finish while it is possible for that partial tracing to receive a scan request from another partial tracing on which it is a dependent. This scan request may be generated by the responsible partial tracing or as a consequence of a third partial tracing on which the responsible partial tracing in turn is dependent. Consequently, the scan phase of a partial tracing is only terminated when the scan phase of every partial tracing on which it is transitively dependent has terminated.

Recall the termination protocol presented in section 4.5. In the non-scalable model, any partial tracing object member of PT_z, after having changed to passive-quiet, may be re-activated by a remote step (*i.e.* a mark-request) from another partial tracing object that is also member of PT_z. This means that there is at least one process which is active-disquiet, hence that the tracing phase has not terminated. Our solution accounting for co-operative partial tracings introduces a new action that may re-activate a process in the passive-quiet state. This is a consequence of a remote step executed by a $PT_y \in PT_z.Responsibles$. This means that the co-operative PT_y has not finished its scan phase, that is, there is at least one process involved in PT_y which is active-disquiet. This process may be an element of $PT_y.Participants$ or an element of $PT_u.Participants$ where $PT_u \in PT_y.Responsibles$.

We now define the relation *Dependent*, which identifies for a given partial tracing those partial tracings that are directly dependent on it. Given a PT_z,

Definition 5.5 $Dependent(PT_z, PT_y) \equiv PT_y \in PT_z.Responsibles.$

Now, let $Dependent^*$ be the reflexive transitive closure of relation *Dependent*, that is $Dependent^*(PT_z, PT_y)$ holds if and only if one of the following holds:

- $PT_z = PT_y$

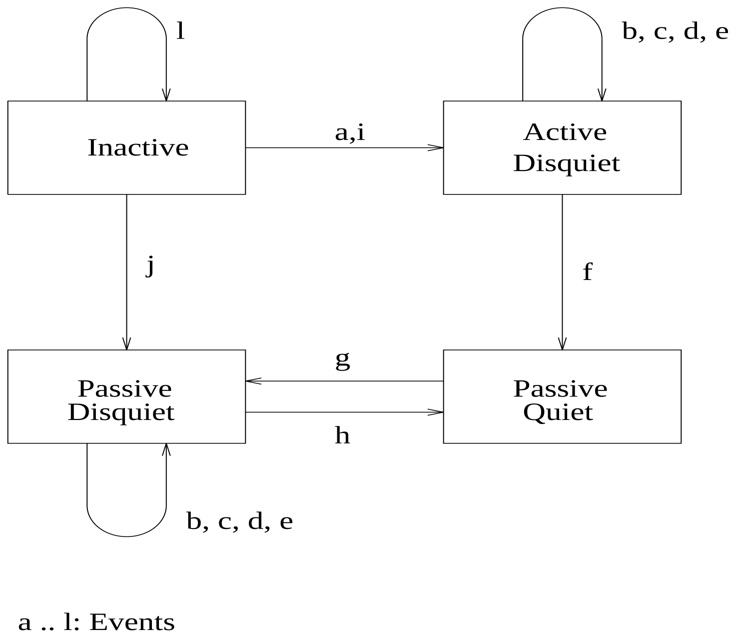


Figure 21: State transition diagram for termination detection of PT_z accounting for co-operative partial tracings.

- $Dependent(PT_z, PT_y)$
- or there is a PT_u such that $Dependent(PT_z, PT_u) \wedge Dependent^*(PT_u, PT_y)$;

The new state transition diagram for termination detection of a PT_z's scan phase is shown in figure 21. The events illustrated in the figure are summarised in figure 22. Compared with figure 15 on page 97, notice that there is one new event, l — related to actions from co-operative partial tracings — and event i is modified to account for co-operative partial tracings. That is, an *Inactive* but **Dynamic** process may only start the scan phase when it receives a scan request if that request is sent by PT_z. These events are related to actions from co-operative partial tracings. Events c and g may now be originated by co-operative partial tracings.

Intuitively, the condition of stability defined in section 4.5:

$$X \text{ is not active-disquiet;}$$

must be now satisfied for every process X in $PT_z.Participants \cup PT_y.Participants$ for all PT_y where $Dependent^*(PT_z, PT_y)$ holds. When such a state is known by each participant of a partial tracing, it may enter the sweep phase safely.

	Event	Action
a	start-phase	• initialise *local-steps* • *responsible* = *self* • *state* = active-disquiet
b	send *scan request*(*exit-item*)	• *grey-set* = *grey-set* $\cup$ {*exit-item*}
c	receive *scan request*(*exit-item*) from $p \in PT_z.Participants \cup PT_y.Participants$ where $PT_y \in PT_z.Responsibles$	• *local-steps* + + • send *acknowledgement*(*exit-item*) to p
d	receive *acknowledgement*(*exit-item*)	• *grey-set* = *grey-set* $\setminus$ {*exit-item*}
e	perform *local step*	• *local-steps* − −
f	*grey-set* = $\emptyset$ $\wedge$ *local-steps* = 0	• *responsible* = *none* • *state* = passive-quiet
g	receive *scan request*(*exit-item*) from $p \in PT_z.Participants \cup PT_y.Participants$ where $PT_y \in PT_z.Responsibles$	• *local-steps* + + • *responsible* = *other*(p) • *reply-set* = *reply-set* $\cup$ *exit-item* • *state* = passive-disquiet
h	*grey-set* = $\emptyset$ $\wedge$ *local-steps* = 0	• for exit-item in *reply-set* send *acknowledgement*(*exit-item*) to *responsible* • *responsible* = *none* • *state* = passive-quiet
i	receive *scan request*(*exit-item*) from $p \in PT_z.Participants$ $\wedge$ **dynamic**	• initialise *local-steps* • set *responsible* = *self* • send *acknowledgement*(*exit-item*) to p • *state* = active-disquiet
j	receive *scan request*(*exit-item*)*from* $p \in PT_z.Participants$ $\wedge$ **non-dynamic**	• *local-steps* = 1 • *responsible* = *other*(p) • *reply-set* = *reply-set* $\cup$ *exit-item* • *state* = passive-disquiet
l	receive *scan request*(*exit-item*) at entry-item Ei_a from $PT_y \in PT_z.Responsibles$	• *local-scan-root-set*(PT_z) = *local-scan-root-set*(PT_z) $\cup$ $\{Ei_a\}$ • send *acknowledgement*(*exit-item*)

Figure 22: State changes for termination detection of PT_z accounting for co-operative partial tracings.

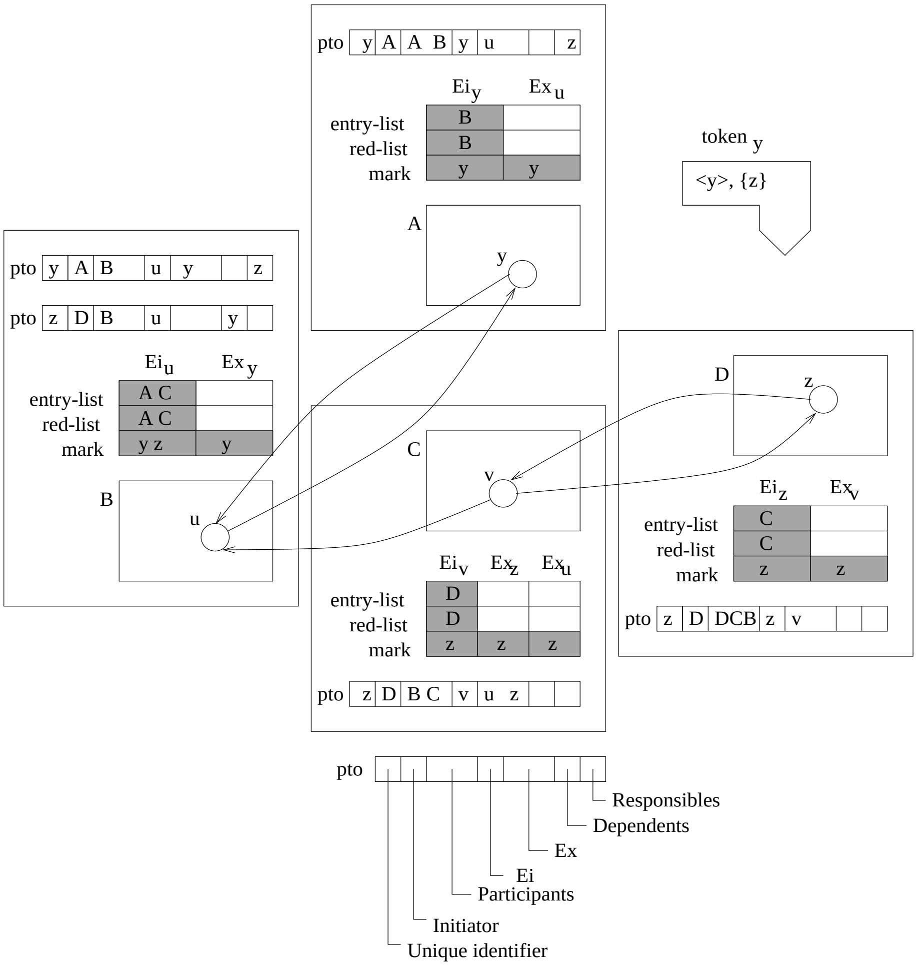


Figure 23: End of the scan phase

We have defined a new global state in which termination is achieved. We now require a report phase that allows every process involved in a partial tracing to be notified of such a global state. For that, it is useful to have some definitions relating to the state of every process and such a global state.

Definition 5.6 *Given* $P \in PT_z.Participants$*,* $stable(P) \equiv \neg active\text{-}disquiet(P)$*.*

We also say that a PT_z is *partial-terminated* if and only if all its participants are stable:

Definition 5.7 $partial\text{-}terminated(PT_z) \equiv \forall P \in PT_z.Participants \cdot stable(P)$*.*

PT_z's report phase implements the algorithm to compute $partial\text{-}terminated(PT_z)$ for $PT_z.Initiator$. We described it in section 4.5.2: every participant $P \in PT_z$.

Participants sends a report message to $PT_z.Initiator$ when $stable(P)$. $PT_z.Initiator$ detects $Partial\text{-}terminated(PT_z)$ when it has received report messages from all its participants and it is stable itself. Additionally, every participant pto_{Pz} reports (in the same report message) to the initiator a list of all partial tracings on which it is dependent: $pto_{Pz}.Responsibles$. At this time the initiator knows the identity of every partial tracing PT_z on which it is dependent (recall definition 5.4 on page 115):

$$PT_z.Responsibles = \bigcup_{P \in PT_z.Participants} pto_{Pz}.Responsibles$$

Figure 23 shows the example at the end of the scan phase. pto_{By} in process B has reported to pto_{Ay} in process A that initiated the collection that PT_y is dependent on PT_z.

The predicate $partial\text{-}terminated(PT_z)$ is a locally stable predicate (akin to a locally stable condition — recall section 4.5). Once this property becomes true, the state of the partial tracing over which the property holds will not change with respect to the property, that is, the property never becomes false again during this collection. Once a process becomes *passive*, it never changes to *active-disquiet* again (recall section 4.5) during this collection. Our termination property for PT_z when accounting for co-operative partial tracings is that all partial tracings on which it depends are partially terminated.

Definition 5.8 *$terminated(PT_z)$ is defined as follows:*

$$terminated(PT_z) \equiv (\forall PT_y \cdot Dependent^*(PT_z, PT_y) \Rightarrow partial\text{-}terminated(PT_y))$$

Consequently, we define a report phase algorithm — the **Token Algorithm** — that, given PT_z, determines if, for all PT_y where $Dependent^*(PT_z, PT_y)$, $partial\text{-}terminated(PT_y)$ is true. It is only at this moment that we need remote communication between co-operative partial tracing objects. Furthermore, this communication is only between initiators. The basic idea of the **Token Algorithm** is to calculate $Dependent^*$. We adopt the simple protocol of passing a token around a ring formed by each initiator of the co-operative partial tracings (Rana 1983), so that when a token has returned to the initiator that created it, the scan phase is known to be complete. The requirement is that a PT_y is only added to *Dependent** when *partial-terminated*(PT_y).

As soon as PT_z.*Initiator* partially terminates, it constructs a token. The token has two parts:

terminated a list that represents *Dependent** in so far as it has been calculated; initially this is $< PT_z >$. The head of the terminated list is the partial tracing that started the token.

next a set that holds initiators not yet visited; initially this is PT_z.*Responsibles*.

Propagation of the token around the ring is simple:

Starting the token($\mathbf{PT_z}$): The starting condition is that *partial-terminated*(PT_z). If PT_z.*Responsibles* $\neq \emptyset$ then create a *terminated* list containing element PT_z and create a *next* set, PT_z.*Responsibles*; PT_z.*Initiator* may send this token to any element of the *next* set.

Receiving the token($\mathbf{PT_y}$): if PT_y receives token, it either passes it on or retains it according the following rules:

Rule1: if not *partial-terminated*(PT_y) then PT_y.*Initiator* retains the token until *partial-terminated*(PT_y), at which time it sends the token to any element of the token's *next* set according to rules 2 and 3.

Rule2: if *partial-terminated*(PT_y) then PT_y.*Initiator* sends the token (see below). If $PT_y = PT_z$, that is, PT_y is the head of the *terminated* list, the scan phase has terminated. The initiator reports this to its participants (recall section 4.5.2).

Rule3: as an optimisation, if *terminated*(PT_y) then remove all $PT_u \in PT_y$. *Responsibles* from the *next* set and append them to *terminated* list, as since, if *terminated*(PT_y), all PT_y's responsibles must have already terminated. This may happen when a partial tracing that has also started a token has received it back by the time another token belonging to another initiator arrives.

Sending the token($\mathbf{PT_y}$): remove PT_y from the *next* set and append it to the *terminated* list. If any $PT_u \in PT_y$.*Responsibles* is not in the *terminated* list, then insert PT_u it into the *next* set. Then proceed according the following rules:

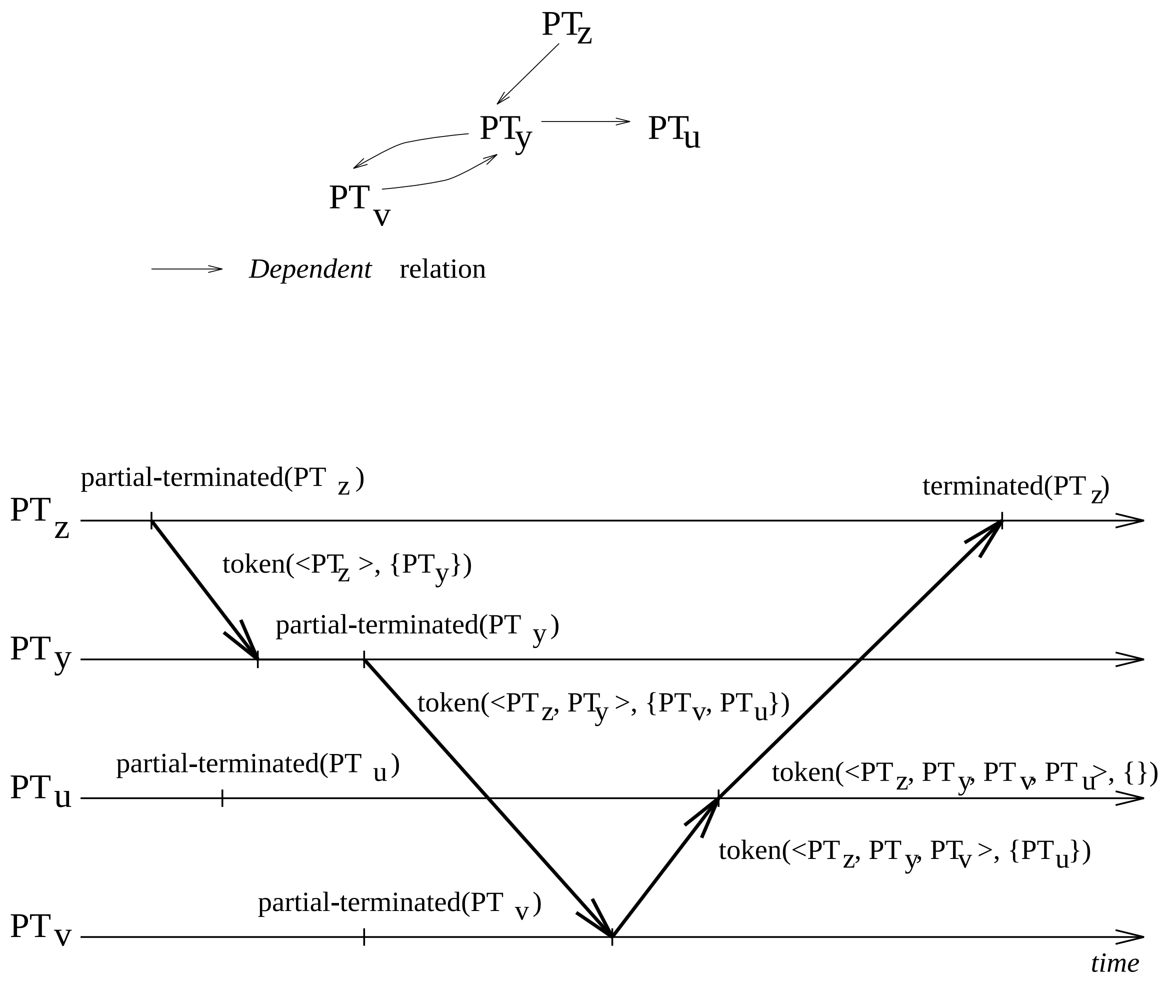


Figure 24: Token Algorithm.

Rule4: if the *next* set is empty then send the token to head of the *terminated* list.

Rule5: if the *next* set is not empty then send the token to any member of the *next* set.

In order to proceed to the sweep phase, each initiator has to initiate a token. PT_z.*Initiator* initiates the token when *partial-terminated*(PT_z) itself. When PT_z.*Initiator* receives the token back it may proceed independently to the sweep phase; the suspect sub-graph held by the corresponding group is collected.

Figure 23 on page 124 shows the token sent by pto_{Ay} at process A — the initiator — to its responsible, pto_{Dz} at process D; pto_{Dz} will return the token with an empty *next*-set to the head of the *terminated*-list, PT_y. As pto_{Dz} has an empty *responsibles* set, it does not need to wait for any other partial tracing to terminate.

Consider figure 24. It shows a more general example. We show a *Dependent* relation and the token initiated at PT_z, when *partial-terminated*(PT_z). PT_z is dependent on PT_y. Consequently, it appends PT_y to the *next* set, and sends it to the first element of the *next* set, in this case PT_y. After having received the token, PT_y must wait until *partial-terminated*(PT_y). At this moment it appends itself to the *terminated* list and appends its responsibles to the *next* set. These actions are repeated by every partial tracing receiving the token. Notice that PT_v does not append PT_y to the *next* set, because it is already in the *terminated* list. Following our algorithm this means that *partial-terminated*(PT_y). When PT_u receives the token, it sends it immediately to PT_z because *partial-terminated*(PT_u) and $next = \emptyset$. When PT_z receives the token back, it may proceed to the sweep phase.

5.4 Example

In this section we present an example. We aim to illustrate in more detail the start of a partial tracing, the creation of a partial tracing object, the mark-red phase steps, the scan phase steps, the co-operation between different partial tracings and the corresponding distributed termination detection protocol. We consider the graph in figure 25.

Suppose that x, u and r initiate PT_x, PT_u and PT_r respectively, independently. We describe in table 26 the most relevant events for one possible sequence of events of PT_x, PT_u and PT_r. The result of mark-red is shown in figure 25.

Observe the event "PT_x: remote_step from Ex_u at B to Ei_u at D". Ei_u receives a mark-red request. Ei_u is red_u. Two partial tracings, PT_x and PT_u, have met and must establish a responsible/dependent relation. (MR.2) is applied. B is added to Ei_u.*red-list*; PT_x is appended to Ei_u.*marks* and inserted to pto_{Du}.*Responsibles*; PT_u is inserted in pto_{Dx}.*Dependents*.

Observe that pto_{Ax} generated a mark-request from A to B. After have received the mark-red request, pto_{Bx} generated two mark-red requests: one from B to A and another from B to D. After pto_{Bx} have received those mark-red requests acknowledgement, pto_{Bx} acknowledged the mark-red request, sent by pto_{Ax}, to pto_{Ax} (event

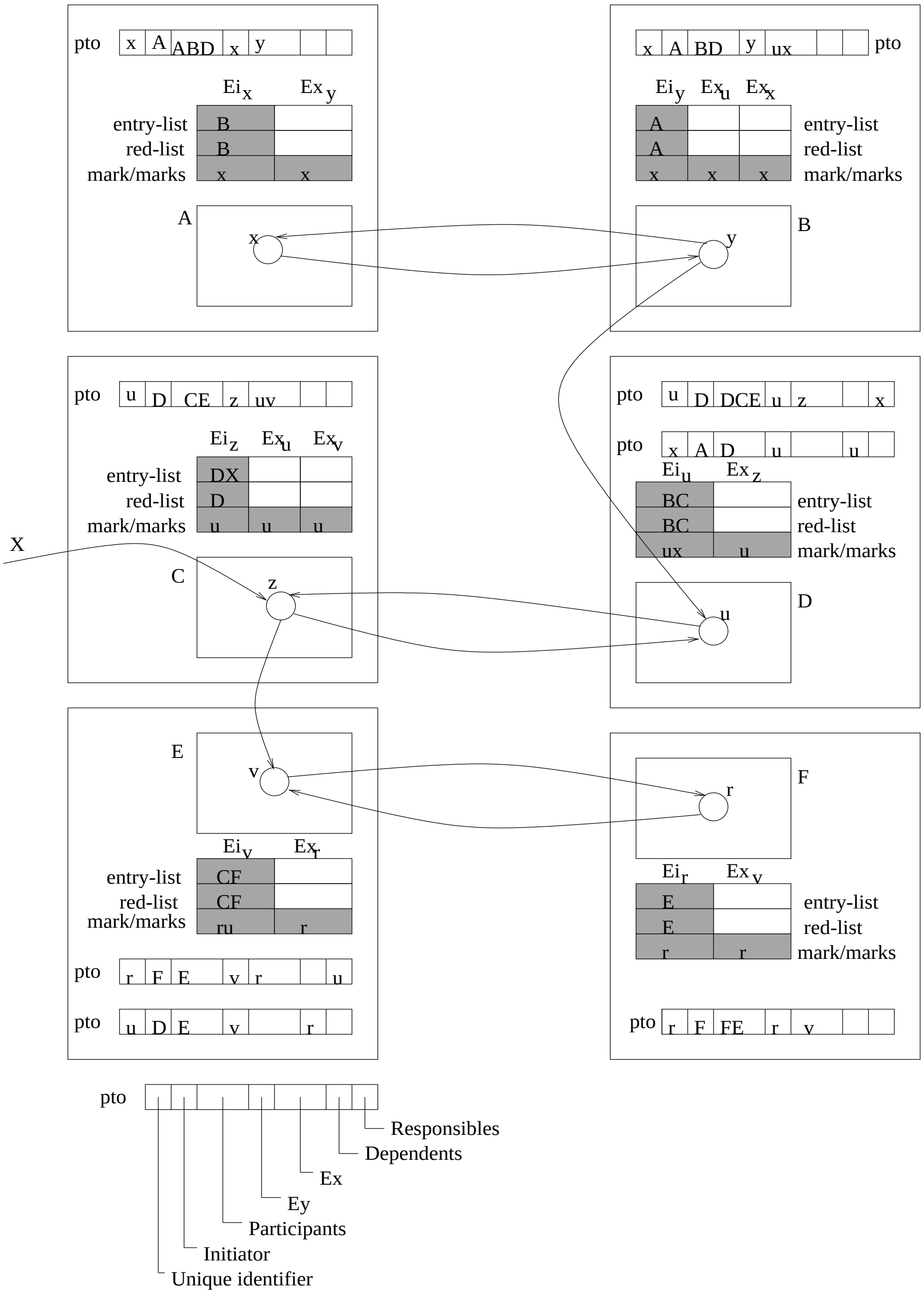


Figure 25: End of the mark-red phase

Event	Action
x initiates PT_x at A	• $pto_{Ax} = (PT_x, A, \{A\}, \{x\}, \emptyset, \emptyset, \emptyset)$ • $Ei_x.mark = x$
u initiates PT_u at D	• $pto_{Du} = (PT_u, D, \{D\}, \{u\}, \emptyset, \emptyset, \emptyset)$ • $Ei_u.mark = u$
r initiates PT_r at F	• $pto_{Fr} = (PT_r, F, \{F\}, \{r\}, \emptyset, \emptyset, \emptyset)$ • $Ei_r.mark = r$
PT_x: local-step from Ei_x to Ex_y	• (ML.1) $pto_{Ax} =$ $(PT_x, A, \{A\}, \{x\}, \{y\}, \emptyset, \emptyset)$ • $Ex_y.mark = x$
PT_u: local-step from Ei_u to Ex_z	• (ML.1) $pto_{Du} =$ $(PT_u, D, \{D\}, \{u\}, \{z\}, \emptyset, \emptyset)$ • $Ex_z.mark = u$
PT_r: local-step from Ei_r to Ex_v	• (ML.1) $pto_{Fr} =$ $(PT_r, F, \{F\}, \{r\}, \{v\}, \emptyset, \emptyset)$ • $Ex_v.mark = r$
PT_x: remote-step from Ex_y at A to Ei_y at B PT_x: local-step from Ei_y to Ex_u PT_x: local-step from Ei_y to Ex_x	• (MR.1) $Ei_y.\textit{red-list} = \{A\}$ • (ML.1) $Ex_u.mark = x$ • (ML.1) $Ex_x.mark = x$ • $pto_{Bx} = (x, A, \{B\}, \{y\}, \{u, x\}, \emptyset, \emptyset)$
PT_x: remote-step from Ex_x at B to Ei_x at A	• (MR.1) $pto_{Ax} =$ $(PT_x, A, \{A\}, \{x\}, \{y\}, \emptyset, \emptyset)$ • $Ei_x.\textit{red-list} = \{B\}$
PT_x: remote-step from Ex_u at B to Ei_u at D	• (MR.2) $Ei_u.\textit{red-list} = \{B\}$ • $Ei_u.marks = \{x\}$ • $pto_{Du} = (u, D, \{D\}, \{u\}, \{z\}, \emptyset, \{x\})$ • $pto_{Dx} = (x, A, \{D\}, \{u\}, \emptyset, \{u\}, \emptyset)$
PT_u: remote-step from Ex_z at D to Ei_z at C	• (MR.1)
PT_u: local-step from Ei_z to Ex_u	• (ML.1)
PT_u: local-step from Ei_z to Ex_v	• (ML.1)
PT_r: local-step from Ei_r to Ex_v	• (ML.1)
PT_r: remote-step from Ex_v at F to Ei_v at E	• (MR.1)
PT_r: local-step from Ei_v to Ex_r	• (ML.1)
PT_r: remote-step from Ex_r at E to Ei_r at F	• (MR.1)
PT_u: remote-step from Ex_v at C to Ei_v at E	• (MR.2)
PT_u: remote-step from Ex_u at C to Ei_u at D	• (MR.1)
$\vdots$	
PT_x at A receives mark-red acknowledgement from B	• $pto_{Ax} =$ $(x, A, \{A, B, D\}, \{x\}, \{y\}, \emptyset, \emptyset)$
PT_u at D receives mark-red acknowledgement from C	• $pto_{Du} =$ $(u, D, \{D, C, E\}, \{u\}, \{z\}, \emptyset, \{x\})$
PT_r at F receives mark-red acknowledgement from E	• $pto_{Fr} =$ $(r, F, \{F, E\}, \{r\}, \{v\}, \emptyset, \emptyset)$

Figure 26: Mark-red phase events

"PT_x at A receives mark-red acknowledgement from B). This acknowledgement informs PT_x's initiator of its participants: A, B and D. This information is recorded in $pto_{Ax}.Participants$.

When receiving the mark-red acknowledgements, pto_{Ax} instructs A, B and D to enter PT_x's scan phase, pto_{Du} instructs D, C and E to enter PT_u's scan phase and pto_{Fr} instructs F and E to enter PT_r's scan phase.

The initial step of PT_u at C discovers Ei_z whose entry and red-list differ. The initial step generates remote steps to Ei_v, Ei_r and Ei_u which are coloured green, as well as the corresponding exit-items. Note that PT_u remote step from Ex_v to Ei_v at E will colour Ei_v. The algorithm will request a local step from Ei_v if/when the partial tracing identified by $Ei_v.mark$ entered the scan phase. The result of scan phase is shown in figure 27.

When every member of PT_x's, PT_u's and PT_r's participants finishes its initial scan step, it informs the corresponding initiator, pto_{Ax}, pto_{Du} and pto_{Fr} respectively, of the corresponding responsible partial tracings. Observe that pto_{Fr} receives from pto_{Er} the information that PT_u is responsible for PT_r. Note the *Responsibles* fields in pto_{Er} (in which the responsible/dependent relation was established) and in pto_{Fr}.

In order to proceed to the sweep phase, each initiator has to initiate a token. PT_r.*Initiator*, process F, initiates *token*$_r$ as shown in figure 27. The *token*$_r$ is sent to PT_u's initiator, process D, because PT_r is dependent on PT_u. PT_u may initiate a token, *token*$_u$, independently as it is dependent on PT_x. Figure 28 shows our example dependent relation and a possible sequence of steps in order to detect termination. Lighter lines describe PT_u termination detection and darker lines describe PT_r termination detection.

Notice that if *token*$_r$ arrived at PT_u's initiator after *terminated*(PT_u), it would be immediately sent back to PT_r's initiator.

5.5 Synchronised Merging

We do not claim that the solution presented up to now in this chapter is complete. As we remarked in section 5.1 (*c.f.* figure 19 on page 112), a partial tracing may finish its mark-red phase, and consequently turn to the scan phase, without the co-operation of

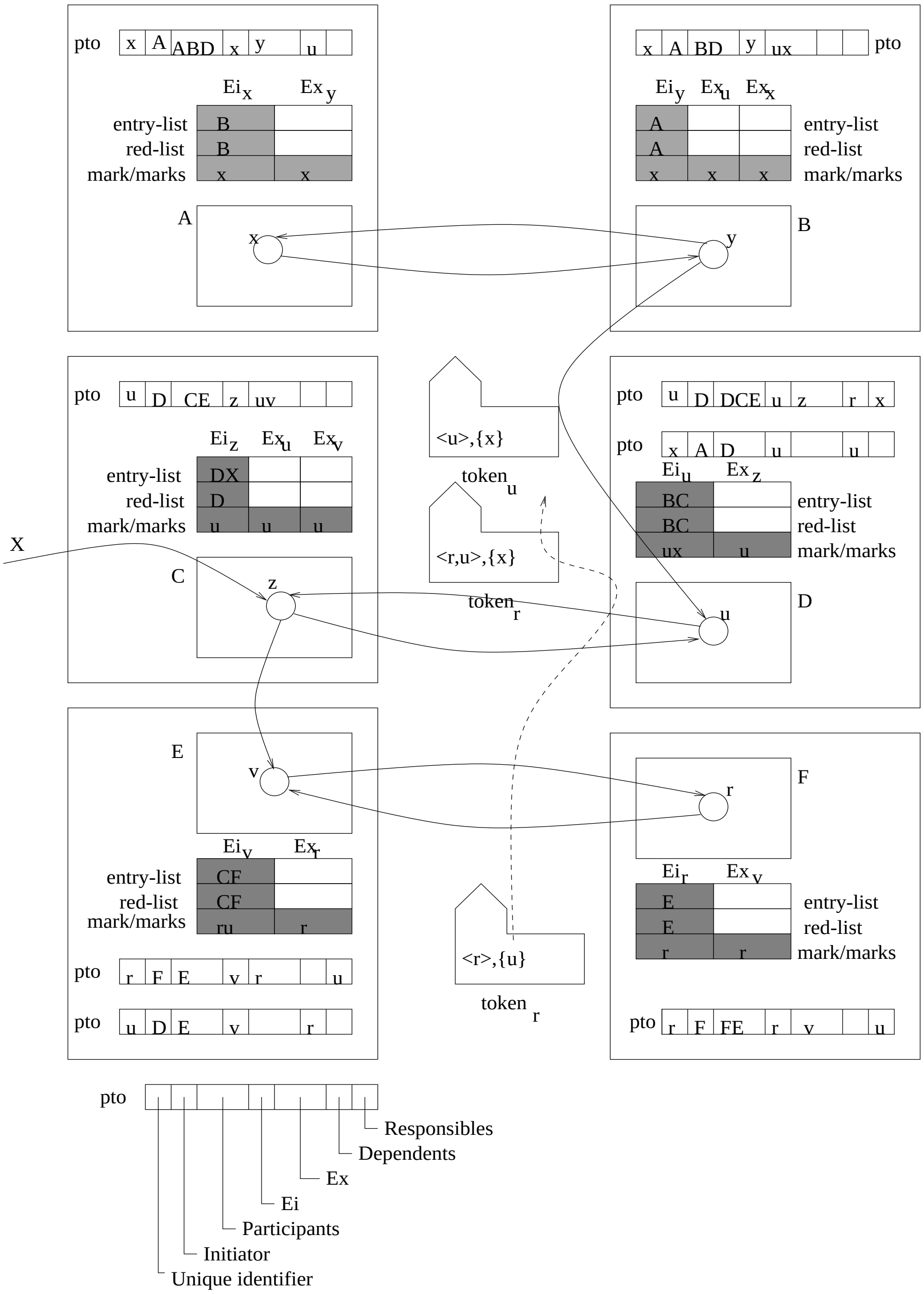


Figure 27: End of the scan phase

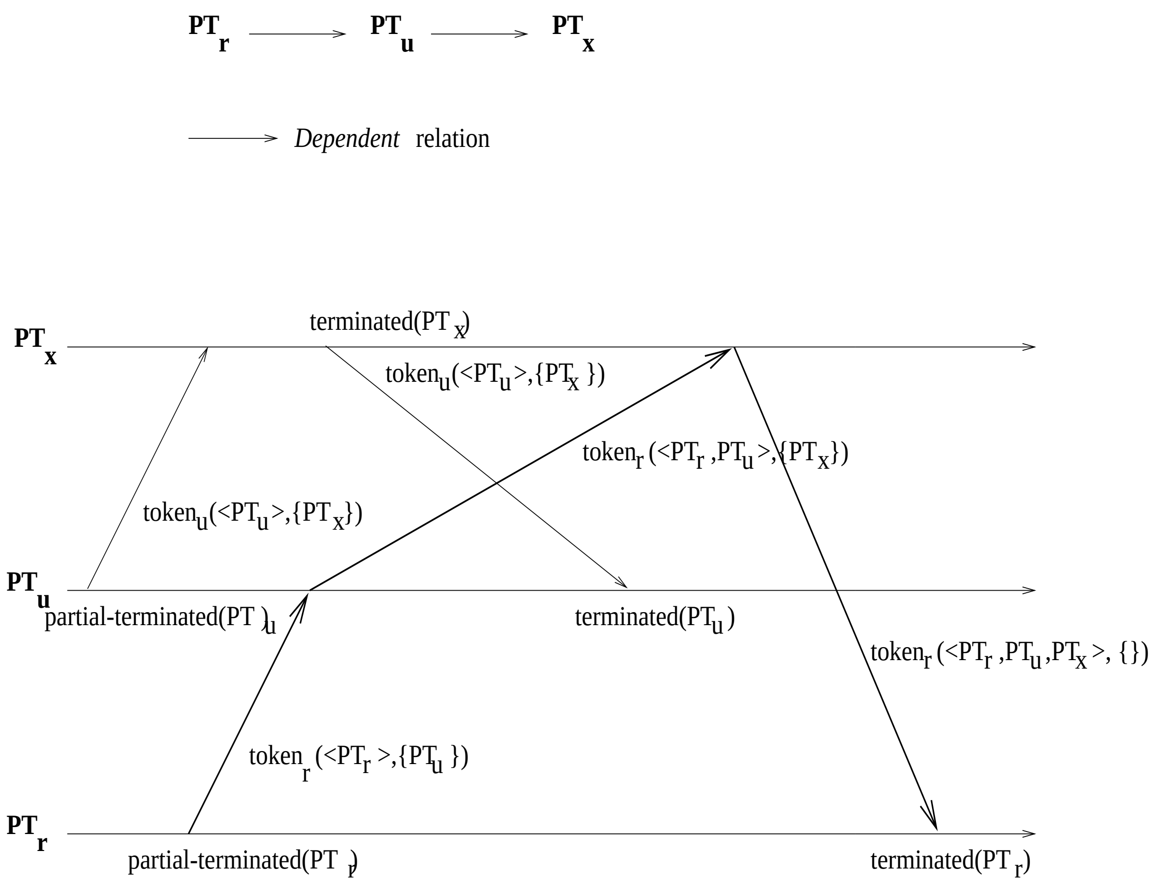


Figure 28: Distributed termination detection

other partial tracings active in the same cycle. This leads to the undesirable situation in which the cycle is not collected. The reason is that the garbage cycle is neither covered by a single partial tracing nor by a set of co-operative partial tracings. Thus, there may always be a reference external to the suspect subgraph. The condition for a garbage cycle to be collected is that it must be covered by a partial tracing — *single-group* — or by a set of co-operative partial tracings — *super-group*.

We claim, however, that our system does provide adaptability, because the mark-red phase only determines entry and exit-items suspected of being garbage. Consequently, it does not make any decision about such items' liveness. As we have just said, our system offers the choice between long-running overlapped collections and more frequent faster collections over small groups.

However overlapping partial tracings leads to repeated work and space overhead. Between these two opposing solutions, a compromise that still achieves completeness is possible, at the cost of another level of synchronisation: synchronisation of the beginning of the scan phases — *Synchronised Merging* We describe this solution next.

Now, we show how the start of the co-operative partial tracings scan phase can be synchronised to obtain a complete solution.

We define the relation *Responsible* (akin to definition 5.5 on page 121). The aim of this relation is to identify, for a given partial tracing, those partial tracings for which it is directly responsible. Given PT_z,

Definition 5.9 $\mathit{Responsible}(PT_z, PT_y) \equiv PT_y \in PT_z.\mathit{Dependents}.$

Now, consider $\mathit{Responsible}^*$ as the reflexive transitive closure of relation *Responsible*, that is $\mathit{Responsible}^*(PT_z, PT_y)$ holds if and only if one of the following holds:

- $PT_z = PT_y$
- $\mathit{Responsible}(PT_z, PT_y)$
- there is PT_u such that $\mathit{Responsible}(PT_z, PT_u) \wedge \mathit{Responsible}^*(PT_u, PT_y)$;

Given a PT_z, the synchronisation of the begining of PT_z's scan phase and the begining of PT_y's scan phase, where $\mathit{Responsible}^*(PT_z, PT_y)$, is implemented by the Token Algorithm described in section 5.1. We give the following definitions:

Definition 5.10 *partial-terminated-mark-red(PT_z) is true if and only if PT_z.Initiator has received all the acknowledgements for the mark-red requests it generated.*

Now, we require that every initiator should know which partial tracings it is responsible for. This may be determined by the same acknowledgement system that allows the initiator to be aware of all participants. For that, the acknowledgement of a mark-request by an entry-item belonging to another partial tracing should return the partial tracing's identity. The acknowledgement system would propagate it to the initiator. At the stage where *partial-terminated-mark-red*(PT_z), PT_z's initiator knows the identity of every partial tracing it is responsible for (recall definition 5.4 on page 115):

$$PT_z.Dependents = \bigcup_{P \in PT_z.Participants} PT_{zP}.Dependents$$

The following definition captures the state where the synchronisation of the start of PT_z" scan phase and the start of PT_y's scan phase, where *Responsible*$^*(PT_z, PT_y)$, is achieved:

Definition 5.11 *terminated-mark-red(PT_z) is true if and only if, for all PT_y where Responsible$^*(PT_z, PT_y)$, partial-terminated-mark-red(PT_y) is true.*

The **Token Algorithm** detects such a state. PT_z's initiator retains the token until *partial-terminate-mark-red*(PT_z). Then, if the head of the token's *terminated* list is PT_z, *terminated-mark-red*(PT_z). Otherwise, PT_z's initiator (i) removes itself from the *next* set to the end of to the *terminated* list, (ii) inserts any of its dependent partial tracings that are not members of the *terminated* list into the *next* set, and (iii) passes the token to any member of the *next* set. If this set is empty, all PT_y where *Responsible*(PT_z, PT_y) have terminated and the token is returned to its owner, the head of the *terminated* list.

Intuitively, we conclude that if the system synchronises the start of the scan phase of the multiple partial tracings active in the same cycle, that cycle will be eventually covered by a *super-group* and eventually collected. We prove that this solution is complete in section 7.3.

In the absence of measurements we cannot conclude that this complete solution is preferable to the first that we presented. Although it achieves completeness, it also presents an extra synchronisation phase.

5.6 Summary

We have presented a scalable solution for garbage collection on distributed large address spaces. Scalability is achieved in our system by having each process collected independently from the rest of the system through the reference listing protocol. Additionally, cycles of garbage are collected by the three-phase partial tracing. This is a scalable solution, because it does not involve the whole distributed system. The mark-red phase forms groups of processes dynamically that will co-operate in the collection of garbage cycles. Our solution is incomplete in the sense that a partial tracing may not cover the whole transitive closure of a suspect subgraph. However, the merger of concurrent partial tracings ensures completeness.

We transformed the garbage collection of a distributed suspect subgraph to the garbage collection of the corresponding distributed suspect *cut-references* graph. We benefit from less space overhead, and from cheap local steps. When two partial tracings meet in an entry or exit-item, they establish a protocol for the co-operation through partial tracing objects in each process. This co-operation is extended until the end of scan phase.

The establishment of the *Dependent/Responsible* relation between two partial tracings is a fundamental aspect of our algorithm. The termination detection protocol is dependent on such a relation. The scan phase of a partial tracing is only terminated when the scan phase of every partial tracing on which it is transitively dependent has terminated.

Chapter 6

Mutator Concurrency

Until now, we assumed that a partial tracing executes without intervening mutations. In practice, while a partial tracing is executing, a mutator may change the object graph so that a partial tracing no longer has a consistent view of the distributed graph. We give now a detailed description of techniques for preserving safety and liveness in the presence of concurrency and show how we benefit from features that reduce the costs introduced by the need for synchronisation. This problem was discussed in section 2.4.1, in the context of uniprocessor garbage collection. The parameters by which we can judge incremental algorithms are their degree of conservatism, synchronisation overheads and termination cost.

We present our solutions for synchronisation between the mutator and a partial tracing here. In chapter 7 we prove that our solution is safe and live. We also show that mutator synchronisation actions do not interfere with the termination detection protocol described in section 4.5.

6.1 Synchronisation

Recall that the mark-red phase only aims at identifying those entry and exit-items suspected of belonging to a garbage cycle. It does not make any decision about entry and exit-items' liveness. Thus, the mark-red phase does not need to be accurate. In contrast to the mark-red phase, the scan phase must be complete with respect to the red suspect subgraph, since it must ensure that all live red entry and exit-items are repainted green[1].

Mark-red Phase

The mark-red phase does not have any need for synchronisation with mutators. One benefit of this is that the mark-red phase does not delay mutator activity, since it does not require any synchronisation. Additionally, we gain cheap termination because there is no need to account for mutator actions that may violate the local condition of section 4.5 and there is no need for uninterruptible actions of mark-red to check for termination. Consequently, the mark-red phase preserves the termination protocol invariants, irrespective of concurrency.

Scan Phase

Mutator activity may prevent a root (any member of the *local-scan-root-set*) from ever being seen by the scan phase, because of scan phase initial steps not being synchronised in each process, and processes rapidly exchanging and deleting references between spaces. This may happen because a mutator message could arrive after the initial step had been taken. We illustrate this problem in figure 29 and figure 30 (for simplicity, we represent entry and exit-items as common objects, and we do not represent the corresponding objects). Process A is involved in a partial tracing. Suppose that, after the initial step has finished at process A, a mutator message arrives at object a. As a result of the method invocation, Ex_b is now reachable from the *local-scan-root-set* at process A. Also, suppose mutator and local collector activity at process B results in the deletion of the external reference to object a. Consequently, without any co-operation from the mutator, exit-item Ex_b would be missed by the scan phase and wrongly reclaimed by the subsequent sweep phase.

Also consider figure 30. Suppose that process A transmits a b-reference to process C. Entry-item Ei_b receives an insert message (recall section 3.4.4) such that $Ei_b.\mathit{entry\text{-}list} = Ei_b.\mathit{entry\text{-}list} \cup \{C\}$. In this way, Ei_b's red-list and Ei_b's entry-list differ. If the insert message arrived after initial step has been taken at B and b does not receive a scan request from A — Ex_b is garbage or the b-reference at A is deleted — Ex_b would be

[1] It may also suggest that garbage items are live, hence it is conservative

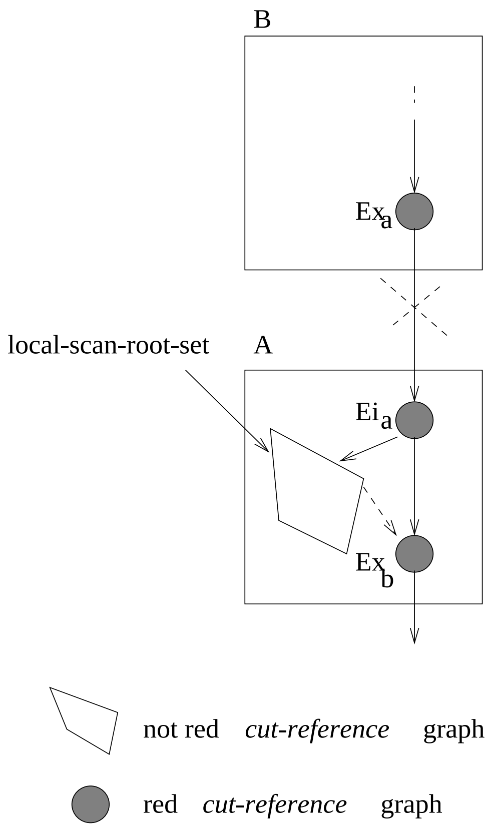


Figure 29: Reference mutations — local copy (dotted lines).

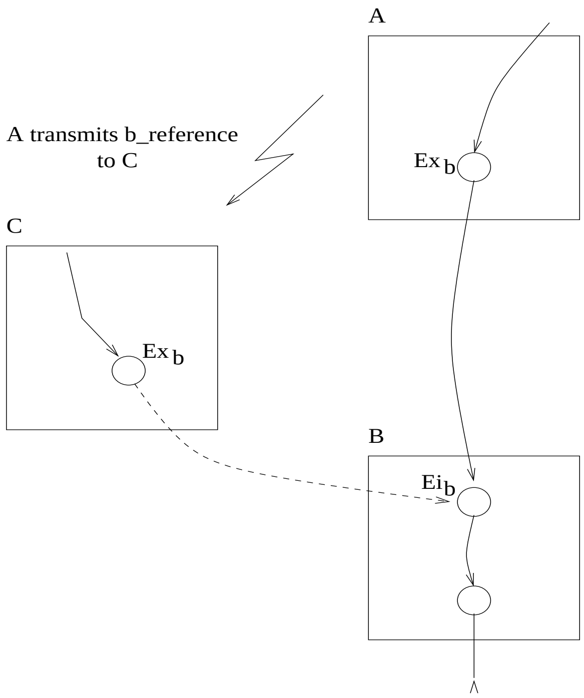


Figure 30: Reference mutations — remote copy (dotted lines).

missed by the scan phase and wrongly reclaimed by the following sweep phase.

We first consider the effect of mutator actions on the scan phase initial step and then on scan phase remote steps. Recall that the *local-scan-root-set* is part of the global root set in each processor. The initial step starts from the *local-scan-root-set* and colours green all local entry and exit-items reached from it. A local mutator may only change the connectivity of the graph by overwriting references to objects. Such writes can be detected by a *write barrier* (section 2.4.1). Subsequent scan requests at entry-items are propagated (local step) atomically to the corresponding exit-item. Although local steps could also be protected with a *write barrier*, we believe that such a implementation would be expensive. We describe our implementation in section 8.

On termination of the initial step in a process P, the local red subgraph will be isolated from the live object graph (green or white) held in that process. The reason is intuitively explained as follows: the initial step colours green any local entry or exit item reachable from the *local-scan-root-set*, that is from references external to the red suspect subgraph graph. Thus any remaining red entry-item Ei_z and its red descendent exit-items were not initially reachable from the *local-scan-root-set*.

A red entry-item Ei_z at process P not belonging to the *local-scan-root-set* at the beginning of the initial step has its entry-list equal to its red-list. By our algorithm, this means that all processes Q holding an exit-item Ex_z must have coloured red Ex_z with a mark-red local step. If z is not being transmitted to any process, that is, any process Q holding a z-reference is not involved in an incomplete transmission operation (recall section 3.4), only actions through such red exit-items — members of the red suspect *cut-reference* graph in other processes — may change the reachability of the red entry-item Ei_z and its red descendent exit-items after initial step has finished at P. Consequently, this reachability may only be changed if:

- process Q makes a remote invocation on red Ei_z's corresponding object;
- process Q transmits the z-reference through a red exit-item Ex_z;
- process Q transmitted the z-reference before Ex_z has been coloured red and z-reference is in transit;

In order to ensure that no item at process P is missed by the scan phase, we ensure that the following invariants always hold after the end of the scan phase initial step at P:

Invariant 6.1 *After SI(P), no red exit-items are reachable by the local-scan-root-set at P.*

Invariant 6.2 *After SI(P), no red entry or exit-items become new members of the local-scan-root-set.*

We also define the auxiliary invariant:

Invariant 6.3 *A red entry-item Ei_z, where Ei_z.red-list $=$ Ei_z.entry-list, cannot receive a message that inserts a process into Ei_z.entry-list unless there is a scan request in transit to that Ei_z.*

To preserve the above invariants, mutator actions through red exit-items require synchronisation operations (akin to *read* and *write* barriers described on chapter 2) — before the mutator can perform such an operation, it must activate the garbage collector (scan phase) to perform some action.

We introduce a *read barrier* on red exit-items: **Remote Barrier**.

Remote Barrier When a mutator at process Q invokes or transmits a remote reference to an object z at process P through a red exit-item Ex_z, it performs a remote step (scan request) to Ei_z at P. The remote step colours green Ei_z and all exit-items locally reachable from Ei_z atomically, before the mutator can change the connectivity of red exit-items.

Notice that, as a consequence of the scan phase not being synchronised in each process, an exit-item Ex_z that corresponds to an entry-item Ei_z may have been coloured green by the initial step at Q or by a scan remote step (scan request). In this case, a scan remote step must have been executed to colour Ei_z green, but one of the mutator actions described above may be performed on Ei_z before the remote step reached Ei_z because of network latency. For now we assume that messages are to arrive in the same order as they are generated in point to point connections. This may be effected, as we show in chapter 8, by only colouring green an exit-item after colouring green the corresponding entry-item. In this way, the following invariant is maintained:

Remote-step invariant 6.4 $\forall\, Ex_z.(green(Ex_z) \Rightarrow green(Ei_z))$

The **Remote Barrier** preserves the invariant 6.1 at process P by not allowing the mutator to read references to objects corresponding to red entry-items. In this way, the mutator cannot change the reachability of red exit-items. The *Remote Barrier* is effectively a *read barrier* (Baker 1978).

Invariant 6.2 is preserved by the **Remote Barrier** and invariant 6.3. The reason is intuitively as follows: to be a new member of the *local-scan-root-set* after the initial step, a red entry-item Ei_z must receive an insert message adding a reference to its *entry-list*, so that Ei_z*.red-list* $\neq$ Ei_z*.entry-list*. If such a message was sent after the source Ex_z has been painted red, the **Remote Barrier** ensures that Ex_z is greened. By invariant 6.4, Ei_z is green. Consequently the new member of the *local-scan-root-set* and the exit-items from which it is reachable are green.

We now have to make sure that such an insert message is not sent before Ex_z is marked red. For that we introduce the following restriction on mark-red.

Mark-red Restriction Do not perform remote steps through exit-items corresponding to references being transmitted. Following the terminology in (Ladin and Liskov 1992), do not perform remote steps through references in the *translist*[2].

Cut-references

Until now, scan phase initial and local steps ignored the cut-reference graph as it may have changed since the last computation. Because of this the initial step has to include local roots in the *local-scan-root-set* as the reachability of the local graph, and consequently the reachability of suspect items may have changed. For the same reasons, local steps have to perform an (atomic) trace from entry to exit-items, as reachability between those items might also have changed.

However, we would benefit substantially from a system that kept *cut-references* up to date. That is, an initial step would compute which items were reachable from outside

[2]We have chosen to model our problem using the *translist* model, since it is implemented by the Network Objects system. However, there is a correspondence between this problem and the race conditions problem described in section 3.4. Consequently, other solutions could be applied.

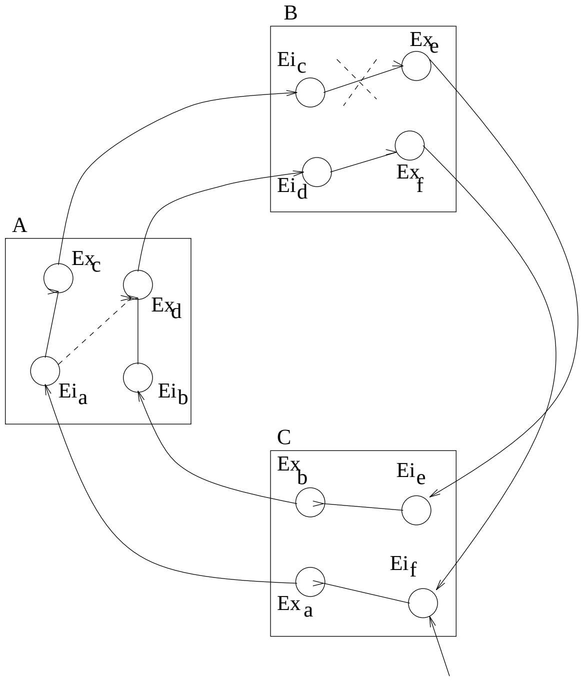


Figure 31: Reference mutations (dotted lines) and *Cut-references* graph.

the suspect *cut-references* graph, and a local step would simply follow a *cut-reference* accurately.

First, let us illustrate how a non-accurate *cut-reference* graph may lead to an unsafe scan phase. Consider figure 31 (again, for simplicity, we represent entry and exit-items as common objects, and we do not represent the corresponding objects). Suppose that an external mutator message arriving on process A at object a creates a new path from object a to object d in process B and that this is followed by the deletion of a reference in the old path to d at process B. The *cut-references* graph is no longer accurate. Also suppose that the *cut-references* graph is involved in a partial tracing so that all entry and exit-items become red. Now suppose process B computes a new *cut-reference* graph to reflect the deletion, but process A does not update its *cut-reference* graph to reflect the new reference. In this way, a scan local step on entry-item Ei_a, generated by the external reference to entry-item Ei_f, will miss exit-item Ex_d.

Cut-references — Dirty Barrier

Let us define the *local-scan-root-set* assuming an accurate *cut-reference* graph. By post-condition 5.1 on page 110, the nodes of a *cut-reference* graph are not reachable locally.

Consequently, one may remove the local roots from the *local-scan-root-set* defined on page 119: it suffices for the initial step to check that the red suspect *cut-references* graph is not reachable from external entry-items (any non-red entry-item or red entry whose entry and red-lists differ). Moreover, by post-condition 5.2 on page 110, it would be enough for the initial step as well as for every local step to follow the *cut-references* graph, because, if there is a path between a suspect Ei_z and a suspect Ex_y, then $Ex_y \in Ei_z.exits$. As we said, after this simplification, the initial scan step and the local scan steps are simpler and cheaper. Recall that we do not perform a scan from the local roots anymore. Moreover, the local scan steps just follow the references in the *cut-references* graph.

In this way, the *local-scan-root-set* for a PT_y would be formed by:

- white entry-items, as they do not belong to the red suspect sub-graph,
- green entry-items, as they have already been found to be live,
- any red entry-item marked by either PT_y or any $PT_u \in PT_y.Responsibles$ whose entry and red-list differ, as they are reachable from outside the suspect sub-graph, and
- any other red entry-item marked by other PT_u such that $PT_u \notin PT_y.Responsibles$, as they are not part of PT_y's suspect subgraph.

However, as we showed, if suspect items turn to be live, the *cut-references* of those items may change due to creation and deletion of local references[3]. We ignore deletions since doing so does not affect safety, but simply makes the algorithm overly conservative. Also they do not affect collection of garbage cycles because deletions are reflected in the next computation of *cut-references*. On the other hand, reference creations must be handled so that the scan phase does not miss live exit-items (as illustrated by the example in figure 31.).

A partial tracing needs co-operation from the mutator in order to provide the scan phase's initial step with accurate information about which *cut-references* may have changed in the system and to always provide *cut-references* for safe local steps.

[3]None of these events can occur unless those items are still alive.

By post-condition 5.1 on page 110, we conclude that in order to create a new path to a suspect exit-item, the mutator must have traversed an old path to it. This traversal must have included traversing an inter-process reference to a suspect entry-item, since post-condition 5.1 states that suspect entry-items and suspect exit-items are not locally reachable. Consequently, only external mutator actions may change their connectivity. In the example (figure 31 on page 143), the mutator must have traversed the reference to object b. In this way, it is possible to protect the *cut-references* graph by a read-barrier on suspect entry-items: a **Dirty Barrier**.

Dirty Barrier When a mutator traverses a remote reference to an object z at process P, if P has a suspect entry-item Ei_z for z, it dirties Ei_z and the exit-items in $Ei_z.exits$.

The following post-condition is then true.

Post-condition 6.5

$$\textbf{Dirty Barrier}(Ei_z)$$

$$\{\text{dirty}(Ei_z) \wedge (\forall Ex_y \in Ei_z.\text{exits} \cdot \text{dirty}(Ex_y))\}$$

Dirty entry-items are cleared when an entry-item's *exits* are re-computed (as described in section 5.2). For simplicity, we assume that if the mark-red phase visits a process during such a computation, it waits until it terminates. This condition may be easily relaxed by allowing mark-red to read the old copy while a new copy is being computed. We proceed as follows. After method **identify-suspects**, all dirty information is cleared. After this point, and until the next *cut-references* graph computation, every remote invocation on suspect objects will trigger a **Dirty Barrier**.

As we have described in section 5.2, **compute-graph** computes the list of all suspect exit-items reachable from each suspect Ei_z. Assuming concurrency, the following post-condition holds:

Post-condition 6.6

$$\{\exists Ex_y \cdot (suspect(Ei_z) \wedge suspect(Ex_y) \wedge path(Ei_z, Ex_y)\}$$
$$\textbf{compute-graph}(Ei_z)$$
$$\{Ex_y \in Ei_z.exits\}$$

This post-condition asserts that if there is a path between a suspect entry-item Ei_z and a suspect exit-item Ex_y before method **compute-graph**'s iteration for Ei_z, Ex_y will be a member of Ei_z's *exits*. This is ensured by any 'snapshot-at-the-beginning' incremental tracing.

Compute-Graph is performed concurrently with the mutator **Dirty Barrier**. When the component *exits* is computed for an entry-item Ei_z, if during this computation a **Dirty Barrier** is triggered on Ei_z such that $dirty(Ei_z)$, all exit-items in $Ei_z.exits$ must be dirtied at the end of the computation. Thus,

Post-condition 6.7

$$\textbf{compute-graph}(Ei_z)$$
$$\{\text{dirty}(Ei_z) \Rightarrow (\forall Ex_y \in Ei_z.\text{exits} \cdot \text{dirty}(Ex_y))\}$$

During **compute-graph** iteration for Ei_z (recall that dirty information is cleared after **identify-suspects**), if a **Dirty Barrier** is applied on Ei_z, we remember Ei_z and dirty the exit-items in *exits* at the end of **compute-graph**.

We are now ready to define a set of invariants that must hold in every process P, between the last computation of the *cut-reference* and the initial step at P. Consequently, we are able to redefine a new *local-scan-root-set*, and scan initial step (SI) and local steps (SL) in order to preserve safety. In section 7 we will prove that **Dirty Barrier** ensures safety in the presence of mutator concurrency.

Invariant 6.8

$$\forall Ex_y \cdot (\text{path}(\text{Roots}, Ex_y) \wedge \text{suspect}(Ex_y) \Rightarrow \text{dirty}(Ex_y))$$

If a suspect exit-item is locally reachable then it must be *dirty*.

Invariant 6.9

$$\forall Ei_z, Ex_y \cdot \text{path}(Ei_z, Ex_y) \wedge \text{suspect}(Ex_y) \Rightarrow$$
$$Ex_y \in Ei_z.\text{exits} \vee \text{dirty}(Ex_y)$$

If there is a path between an entry-item and a suspect exit-item, then either there must be a *cut-reference* from the entry-item to that exit-item, or that exit-item is *dirty*.

The *local-scan-root-set* of a PT_a, taking account of co-operative partial tracings, may now be defined as the set of:

- white entry-items, as they do not belong to the suspect sub-graph,
- green entry-items, as they have already been found live,
- every entry-item Ei_z where *dirty*(Ei_z), since no Ex_y (where $Ex_y \in Ei_z.exits$) may be suspect any longer,
- every dirty red exit-item Ex_y, as it may no longer be a suspect,
- every red exit-item Ex_y that have been found non-suspect by the last computation of **find-suspects**,
- any red entry-item marked by either PT_a or any $PT_b \in PT_a.Responsibles$ whose entry and red-lists differ, as they are reachable from outside the suspect sub-graph, and
- any other red entry-item marked by other PT_c where $PT_c \notin PT_a.Responsibles$, as they are not part of PT_a's suspect subgraph.

We can now redefine the scan phase initial step (SI) and scan phase local steps (SL) using an accurate *cut-references* graph. The **Dirty Barrier** ensures that no local roots or external entry-items are missed by the scan phase initial step in any participant. It also ensures that the *cut-reference* between entry and exit-items is consistent.

(Conc-SI.1) Mark green any red entry or exit item E of the *local-scan-root-set* for which $E.mark = PT_a$. For all entry-item Ei_z in the *local-scan-root-set*, green any red exit-item Ex_y where $Ex_y \in Ei_z.exits$ and $Ex_y.mark = PT_a$: they are *green*$_a$.

Notice that there may be new members added to the *local-scan-root-set* by external mutator messages while the initial step at P is being executed. Consequently, this process must be repeated for any new members. Termination is ensured because there is a finite number of red items that may be added to the *local-scan-root-set*.

The local scan phase step for PT_a propagates the green colour from a $green_a$ entry-item Ei_z to those exit-items $Ex_y \in Ei_z.exits$ that PT_a had previously visited in the mark-red phase:

(Conc-SL.1) Green $Ex_y \in Ei_z.exits$ if it is *red* and either $Ex_y.mark = PT_a$ or $PT_a \in Ex_y.marks$.

As we prove in chapter 7, invariants 6.8 and 6.9 ensure the safety of scan phase initial and local steps.

6.2 Termination

In section 4.5 we described a distributed termination protocol for detecting termination of each phase of the distributed partial tracing. However we did not account for synchronisation actions due to to mutator-collector concurrency during the scan phase.

Correct termination detection requires that each remote step (scan request), and subsequent local and remote steps, has an *active-disquiet* process ultimately responsible for it. We need to show that the **Remote Barrier** preserves this condition.

In order to show that the **Remote Barrier** preserves the termination detection protocol, we analyse each case of every possible process state and show that a passive process cannot execute the **Remote Barrier**, thus there must always be a *active-disquiet* process responsible for it.

Inactive P has not taken the initial step. This means that it has not received the report phase message yet. However, it might install a **Remote Barrier**, when the mutator invokes a remote object or transmits a remote reference, in any red exit-item Ex_z because P does not know the state of the target process. Since P is a **dynamic** process because it is a participant, it eventually turns to *active-disquiet*. The exit-item Ex_z should then be inserted on the *local-scan-root-set* as a new root for the initial step.

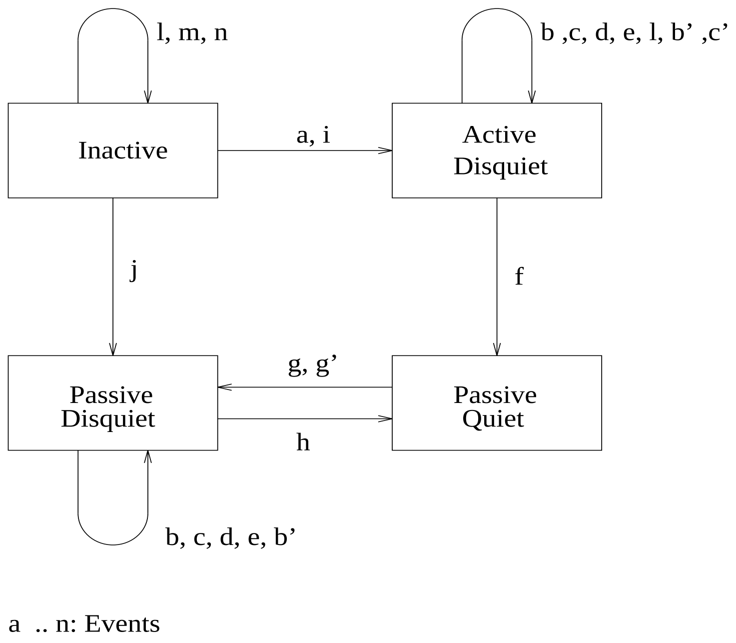


Figure 32: State transition diagram for termination detection of PT_z accounting for mutator concurrency.

Active-disquiet P has not yet received all the acknowledgements for the scan requests generated by its initial step, thus it will take responsibility for any *Remote Barrier* scan request.

Passive-disquiet P has already completed its initial step. Any remaining red exit-item $Ex_y \in P$ can only have been reachable from a red entry-item $Ei_z \in P$. By invariant 6.4 we conclude that all exit-items Ex_z in all other processes Q must still be red. Consequently, a mutator message to $Ei_z \in P$ must have been sent through a red $Ex_z \in Q \neq P$. In this case a **Remote Barrier** must have been executed on Ex_z. As a result, Ei_z and Ex_y must have been coloured green.

Passive-quiet Similar arguments to above.

We conclude that a **Remote barrier** must ultimately have an active-disquiet process responsible for it. Note that the responsible process may be the one that generated the barrier.

We may now complete the state transition diagram introduced in section 4.5 (figure 15). The new events reflect the execution on a red exit-item and reception on a red entry-item of a **Remote Barrier**, and reception of an acknowledgement for each possible state. The new transaction diagram and new events are described in figures 32 and 33. Events b', and c' and g' represent the execution on exit-items and reception on entry-items of the **Remote Barrier** respectively. The actions are the same as sending and receiving scan requests.

6.3 Summary

In this chapter we have presented techniques for preserving safety and liveness in the presence of mutator/collector concurrency. We conclude that since the mark-red phase does not need to be accurate, it does not delay mutator activity and it does not require any synchronisation.

The scan phase must be accurate. Mutator activity may prevent a root from being seen by the scan phase. Moreover, the *cut-references* graph may change since it last computation and consequently the reachability of suspect items may also change.

We defined two synchronisation actions in order to make our algorithm safe in the presence of mutator concurrency. The **Remote Barrier** ensures safety by not allowing the mutator to read references to objects corresponding to red entry-items. In this way the mutator cannot change the reachability of red exit-items.

The **Dirty Barrier** keeps the *cut-references* graph up to date. It ensures that it is always provides accurate information about those *cut-references* that may have changed in the system to the scan phase's initial-step. In this way, the algorithm can always perform safe local steps, directly, from entry to exit-items.

	Event	Action
a	start-phase	• initialise *local-steps* • $responsible = self$ • $state =$ active-disquiet
b	• send *scan request*(*exit-item*)	• $grey\text{-}set = grey\text{-}set \cup \{exit\text{-}item\}$
b'	• trigger a **Remote Barrier**(*exit-item*)	
c	• receive *scan request*(*exit-item*)	• $local\text{-}steps{+}{+}$ • send *acknowledgement*(*exit-item*) to p
c'	• receive **Remote Barrier**(*exit-item*) from $p \in PT_z.Participants \cup PT_y.Participants$ where $PT_y \in PT_z.Responsibles$	
d	receive *acknowledgement*(*exit-item*)	• $grey\text{-}set = grey\text{-}set \setminus \{exit\text{-}item\}$
e	perform *local step*	• $local\text{-}steps{-}{-}$
f	$grey\text{-}set = \emptyset \wedge local\text{-}steps = 0$	• $responsible = none$ • $state =$ passive-quiet
g	• receive *scan request*(*exit-item*)	• $local\text{-}steps{+}{+}$ • $responsible = other(p)$ • $reply\text{-}set = reply\text{-}set \cup exit\text{-}item$ • $state =$ passive-disquiet
g'	• receive **Remote Barrier**(*exit-item*) from $p \in PT_z.Participants \cup PT_y.Participants$ where $PT_y \in PT_z.Responsibles$	
h	$grey\text{-}set = \emptyset \wedge local\text{-}steps = 0$	• for exit-item in *reply-set* send *acknowledgement*(*exit-item*) to *responsible* • $responsible = none$ • $state =$ passive-quiet
i	receive *scan request*(*exit-item*) from $p \in PT_z.Participants \wedge$ **dynamic**	• initialise *local-steps* • set $responsible = self$ • send *acknowledgement*(*exit-item*) to p • $state =$ active-disquiet
j	receive $scan\ request(exit\text{-}item) from$ $p \in PT_z.Participants \wedge$ **non-dynamic**	• $local\text{-}steps = 1$ • $responsible = other(p)$ • $reply\text{-}set = reply\text{-}set \cup exit\text{-}item$ • $state =$ passive-disquiet
l	receive *scan request*(*exit-item*) on entry-item Ei_a from $PT_y \in PT_z.Responsibles$	• $local\text{-}scan\text{-}root\text{-}set(PT_z) = local\text{-}scan\text{-}root\text{-}set(PT_z) \cup \{Ei_a\}$ • send *acknowledgement*(*exit-item*)
m	trigger a **Remote Barrier**(*exit-item*)	• $local\text{-}scan\text{-}root\text{-}set = local\text{-}scan\text{-}root\text{-}set \cup \{exit\text{-}item\}$
n	receive a **Remote Barrier**(*exit-item*) on entry-item Ei_a	• $local\text{-}scan\text{-}root\text{-}set(PT_z) = local\text{-}scan\text{-}root\text{-}set(PT_z) \cup \{Ei_a\}$ • send *acknowledgement*(*exit-item*)

Figure 33: State changes for termination detection of PT_z accounting for mutator concurrency

Chapter 7

Proof of Correctness

In this chapter we outline a proof of the correctness of some aspects of the partial tracing algorithm. First we summarise our model, rewriting the steps, definitions, invariants and post/conditions relevant for the proof. Then, we proceed to the proof itself. We divide the proof into two sections: safety (section 7.2) and liveness (section 7.3). When proving safety, we assume that the distributed tracing is correct providing it terminates. We first use a simplified model that does not account for mutator concurrency or concurrent partial tracings, and we assume that distributed termination is correctly detected. Then, we extend the proof to the distributed termination protocol. We prove that once a partial tracing's initiator has received a report message from every participant, the scan phase has terminated, that is, there can be no mark requests in transit. Consequently, no red object can receive a scan request in the partial tracing's sweep phase. Next, we extend the proof to the mutator concurrency model and prove that mutator actions do not affect termination detection and safety. Finally, we extend the proof to the case where different partial tracings may be simultaneously active in the same or a connected sub-graph. In this case we show that two partial tracings may co-operate without compromising each other's safety and liveness properties.

When proving liveness, we reason about the termination of mark-red and scan tracings, and show that all garbage objects eventually become suspect objects and that garbage cycles within a suspect subgraph are eventually collected by our system.

7.1 Summary of the Model

Mark-red phase

Definition 4.1

$$Ei_x.\text{red-list} =$$
$$\{p \in \text{processes} : Ex_x \in \text{exit-table}(p) \land \text{colour}(Ex_x) = \text{red}\}$$

Mark-red steps

For a partial tracing PT_y, a local step goes from each red entry-item Ei_a, where $Ei_a.mark = PT_y$, to each exit-item Ex_b in $Ei_a.exits$ as follows:

(ML.1) If Ex_b is white, then it is reddened and its mark set to PT_y, that is, $Ex_b.mark = PT_y$: we call it red_y.

(ML.2) If Ex_b is already red_y, then no further action is necessary.

(ML.3) If Ex_b is red_z where $z \neq y$, then two partial tracings have met in the same phase. We merge the partial tracings and say that z is *dependent* on y and y is *responsible* for z. PT_y is appended to $Ex_b.marks$, PT_y is added to the $PT_z.Responsibles$, and PT_z to the $PT_y.Dependents$. Both these interactions take place between the $PTobj$'s in this process — no messages are sent.

(ML.4) If Ex_b is green, it must have been marked by another group operating in a later phase so the red wave-front retreats from this object.

A remote steps executed by PT_y propagates colours from an exit-item Ex_b in a participant P to entry-items Ei_b in a remote process Q:

(MR.1) If Ei_b is white or red_y, P is added to $Ei_b.red\text{-}list$ and Ei_b is marked red_y.

(MR.2) If Ei_b is red_z and $z \neq y$, P is still added to $Ei_b.red\text{-}list$. Once again two partial tracings have met and, as in the local step, PT_y is appended to $Ei_b.marks$ and to $PT_z.Responsibles$, PT_z to $PT_y.Dependents$ in process Q; no messages are exchanged.

(MR.3) If Ei_b is green, no further action is taken and the mark-red phase retreats.

Scan phase

Scan steps

An initial-step:

(SI.1) Mark green any red entry or exit-item E in the *local-scan-root-set* for which $E.mark = PT_y$. Mark green any red exit-item Ex_b for which $Ex_b.mark = PT_y$ that is reachable from the *local-scan-root-set*. These are $green_y$.

The local scan phase step for PT_y propagates the green colour from a $green_y$ entry-item Ei_a to those exit-items Ex_b in the same process reachable from Ei_a that PT_y had previously visited in the mark-red phase:

(SL.1) Green Ex_b if it is red, reachable from green a Ei_a, and either $Ex_b.mark = PT_y$ or $PT_y \in Ex_b.marks$. That is, we green only those exit-items reddened by co-operative partial tracings.

The remote step from a $green_A$ exit-item Ex_b propagates the green colour to the corresponding entry-item Ei_b:

(SR.1) If Ei_b is red and $Ei_b.mark = PT_y$ or $PT_y \in Ei_b.marks$, mark Ei_b green.

(SR.2) If Ei_b is red but neither $Ei_b.mark = PT_y$ nor $PT_y \in Ei_b.marks$, retreat.

(SR.3) If Ei_b is not red, retreat.

(SR.4) Request a local step from Ei_b if $Ei_b.mark$ has entered the scan phase.

Advanced scan steps

The advanced scan steps follow the *cut-references* graph. It suffices for the initial scan step to check that the red suspect *cut-references* graph is not reachable from external entry-items (any non-red entry-item or red entry whose entry and red-lists differ). Moreover, it is enough for the initial step, as well as for every local step, to follow the *cut-references* graph, because, if there is a path between a suspect Ei_z and a suspect Ex_y, then $Ex_y \in Ei_z.exits$.

(Conc-SI.1) Mark green any red entry or exit item E of the *local-scan-root-set* for which $E.mark = PT_a$. For all entry-item Ei_z in the *local-scan-root-set*, green any red exit-item Ex_y where $Ex_y \in Ei_z.exits$ and $Ex_y.mark = PT_a$: they are $green_a$.

(Conc-SL.1) Green $Ex_y \in Ei_z.exits$ if it is *red* and either $Ex_y.mark = PT_a$ or $PT_a \in Ex_y.marks$.

Termination detection

Invariant 4.4

$$\forall p_i \cdot p_i.\text{state} = \text{passive-disquiet} \Rightarrow$$
$$\exists p_j.(i \neq j) \wedge \text{Ancestor}^*(p_j, p_i) \wedge p_j.\text{state} = \text{active-disquiet}$$

Cut-references graph

Post-condition 5.1

$$[\textbf{identify-suspects}]$$
$$\{(\forall Ei_z \cdot \text{suspect}(Ei_z) \Rightarrow \neg\text{path}(Roots, z)) \wedge$$
$$(\forall Ex_y \cdot \text{suspect}(Ex_y) \Rightarrow \neg\text{path}(Roots, Ex_y))\}$$

Post-condition 5.2

$$[\textbf{identify-suspects}]$$
$$\{(\forall Ei_z \cdot \text{suspect}(Ei_z) \Rightarrow \neg\text{path}(Roots, z)) \wedge$$
$$(\forall Ex_y \cdot \text{suspect}(Ex_y) \Rightarrow \neg\text{path}(Roots, Ex_y))\}$$
$$[\textbf{compute-graph}]$$
$$\{\forall Ei_y, Ex_z \cdot (\text{suspect}(Ei_y) \wedge \text{suspect}(Ex_z) \wedge \text{path}(Ei_y, Ex_z)) \Rightarrow Ex_z \in Ei_y.\text{exits}\}$$

Advanced termination detection

Definition 5.5 *Dependent*$(PT_z, PT_y) \equiv PT_y \in PT_z.Responsibles.$

Definition 5.6 *Given $P \in PT_z.Participants$, $stable(P) \equiv \neg active\text{-}disquiet(P)$.*

Definition 5.7 $partial\text{-}terminated(PT_z) \equiv \forall P \in PT_z.Participants \cdot stable(P)$.

Definition 5.8

$$terminated(PT_z) \equiv (\forall PT_y \cdot Dependent^*(PT_z, PT_y) \Rightarrow partial\text{-}terminated(PT_y))$$

Mutator concurrency

Invariant 6.1 *After SI(P), no red exit-items are reachable by the local-scan-root-set at P.*

Invariant 6.2 *After SI(P), no red entry or exit-items become new members of the local-scan-root-set.*

Invariant 6.3 *A red entry-item Ei_z, where Ei_z.red-list $=$ Ei_z.entry-list, cannot receive an insert message unless there is a scan request in transit to that Ei_z.*

Remote-step invariant 6.4 $\forall\ Ex_z.(green(Ex_z) \Rightarrow green(Ei_z))$

Mark-red Restriction Do not perform remote steps through exit-items corresponding to references being transmitted. Following the notation in (Ladin and Liskov 1992), do not perform remote steps through references in the *translist*.

Post-condition 6.5

Dirty Barrier(Ei_z)

$\{\text{dirty}(Ei_z) \wedge (\forall Ex_y \in Ei_z.\text{exits} \cdot \text{dirty}(Ex_y))\}$

Post-condition 6.6

$\{\exists Ex_y \cdot (suspect(Ei_z) \wedge suspect(Ex_y) \wedge path(Ei_z, Ex_y)\}$

compute-graph(Ei_z)

$\{Ex_y \in Ei_z.exits\}$

Post-condition 6.7

$$\textbf{compute-graph}(Ei_z)$$
$$\{\text{dirty}(Ei_z) \Rightarrow (\forall Ex_y \in Ei_z.\text{exits} \cdot \text{dirty}(Ex_y))\}$$

Invariant 6.8

$$\forall Ex_y \cdot (\text{path}(\text{Roots}, Ex_y) \wedge \text{suspect}(Ex_y) \Rightarrow \text{dirty}(Ex_y))$$

Invariant 6.9

$$\forall Ei_z, Ex_y \cdot \text{path}(Ei_z, Ex_y) \wedge \text{suspect}(Ex_y) \Rightarrow$$
$$Ex_y \in Ei_z.\text{exits} \vee \text{dirty}(Ex_y)$$

7.2 Safety

Safety Invariant

The partial tracing algorithm must satisfy the following safety invariant:

Safety property 1 *No live objects are reclaimed.*

In the sweep phase only red entry-items are collected, so the invariant is equivalent to:

Safety property 2 *At the begining of the sweep phase no red entry-items involved in a PT_a are reachable from any root inside or outside the group containing the suspect subgraph.*

7.2.1 Partial tracing algorithm

In this section we prove that the partial tracing algorithm works safely in conjunction with the reference listing protocol. We consider neither mutator-collector concurrency, nor multiple partial tracings nor termination detection.

We want to prove that at the begining of the sweep phase of PT_a

$$\forall Ei_y(red_a(Ei_y) \Rightarrow \neg live(Ei_y)) \tag{1}$$

Suppose that $red_a(Ei_y) \wedge live(Ei_y)$ at the begin of sweep phase. Thus, Ei_y must be reachable from outside the group or from a live exit-item inside the suspect sub-graph. Notice that an entry-item is never locally reachable, although the corresponding object may be locally reachable itself. Thus,

$$\forall Ei_y \cdot (red_a(Ei_y) \wedge live(Ei_y) \Rightarrow \tag{2}$$

$$(\exists Q \notin PT_a.Participants \wedge \exists Ex_z \in Q.exit\text{-}table \wedge path(Ex_z, Ei_y) \tag{3}$$

$$\vee(\exists R \in PT_a.Participants \wedge \exists Ex_u \in R.exit\text{-}table \wedge path(Ex_u, Ei_z) \wedge live(Ex_u)) \tag{4}$$

Now suppose that (3) holds but (4) does not. By the reference listing protocol,

$$path(Ex_z, Ei_y) \Rightarrow Q \in Ei_y.entry\text{-}list \tag{5}$$

That is, Q must have been inserted in $Ei_y.entry\text{-}list$ when $path(Ex_z, Ei_y)$ was created (recall that we assume a safe reference listing protocol). Also,

$$Q \notin PT_a.Participants \Rightarrow \neg red_a(Ex_z) \tag{6}$$

because Ex_z was not visited by PT_a's mark-red phase. If it had been visited, Q would have been made a member of $PT_a.Participants$ by the acknowledgements system.

By definition 4.1,

$$(path(Ex_z, Ei_y) \wedge \neg red_a(Ex_z)) \Rightarrow Q \notin Ei_y.red\text{-}list \tag{7}$$

That is, since exit-items are unique, Q cannot be a member of $Ei_y.red\text{-}list$ because it does not hold a red exit-item to Ei_y. Consequently,

$$(path(Ex_z, Ei_y) \wedge Q \notin PT_a.Participants) \Rightarrow Difference(Ei_y.entry\text{-}list, Ei_y.red\text{-}list) \neq \emptyset \tag{8}$$

From this it follows that Ei_y must have been painted green by the scan initial step. Thus, $live(Ei_y)$ and $red_a(Ei_y)$ is a contradiction. Now suppose proposition (4) holds, that is, there is an exit-item Ex_u belonging to $R \in PT_a.Participants$ from which Ei_y is reachable and $live(Ex_u)$. Since Ei_y was not painted green by the scan phase initial step,

$$red_a(Ei_y) \wedge (Ei_y.red\text{-}list = Ei_y.entry\text{-}list) \tag{9}$$

because, entry-items whose red and entry-lists are painted green by the initial scan step.

So, since $path(Ex_u, Ei_y)$, $R \in Ei_y.entry\text{-}list$. Hence, $R \in Ei_y.red\text{-}list$. Thus, $red_a(Ex_u)$ by definition 4.1. By hypothesis Ex_u is live. Thus,

$$live(Ex_u) \Rightarrow \tag{10}$$

$$(\exists r \in Roots(R) \wedge path(r, Ex_u)) \vee \tag{11}$$

$$(\exists Ei_r \in R.entry\text{-}table \wedge path(Ei_r, Ex_u) \wedge live(Ei_r)) \tag{12}$$

Ignoring concurrency, if (11) holds, Ex_u would have been painted green by the scan initial-step (SI.1). So $live(Ex_u)$ and $red_a(Ex_u)$ is a contradiction. Thus (12) must hold.

We have $path(Ei_r, Ex_u)$. Once we are at the begin of sweep phase, Ei_r must be red. If not, Ex_u would have been painted green by the initial step. Thus any red 'live' entry-item is only reachable from other global roots which are red. Any red 'live' entry-item is reachable from the actual roots or from outside the red subgraph. This contradicts our definition of liveness, hence there are no red live entry-items at the start of the sweep phase.

7.2.2 Distributed Termination Protocol

Now we outline the proof of the distributed termination detection protocol described in section 4.5. We want to show that the distributed termination protocol detects safely the end of the mark-red and scan phases, in order for a partial tracing to proceed to the scan and sweep phases respectively. Consequently, no object may receive a mark-red or scan request at the begining of scan and sweep phases respectively. In order to safely proceed to the next phase, each process P involved in a PT_a must receive a report message from PT_a's initiator. As we stated in section 4.5.2, this means that there are no $P \in PT_a.Participants$ such that $P.state = active\text{-}disquiet$. This property is captured by the following theorem:

Theorem 7.1

$$(\forall P \in PT_a.\text{Participants } P.\text{state} \neq \text{active-disquiet}) \Rightarrow$$
$$(\forall P \in PT_a.\text{Participants} \cdot P.\text{grey-set} = \emptyset \wedge P.\text{local-steps} = 0)$$

The proof of this theorem is based on the validity of invariant 4.4. If we apply this invariant to the set of PT_a's participants, it states that:

$$\forall P \in PT_a.Participants\ (P.state = passive\text{-}disquiet \Rightarrow$$
$$\exists Q \in PT_a.Participants \wedge Ancestor^*(Q,P) \wedge Q.state = active\text{-}disquiet)$$

The proof of this invariant is based on the state transition diagram on figures 15 on page 97 and 16 on page 99. The proof follows by induction on the number n of ancestors of a participant P. This number is finite and it is at least one, because, from the state transition diagram, it follows that:

- at the start, there is always at least one *active-disquiet* process,
- two disquiet processes never establish a *Ancestor* relation, because a disquiet process receiving a scan request immediately acknowledges it. The responsibility is then inherited by the process responsible for the receiver disquiet process.

Case $n = 1$, by definition, P is the only ancestor of P and so P cannot be *passive-disquiet*. Thus, the implication holds.

Case $n = m + 1$. It follows from the state transition diagram that,

$$P.state = passive\text{-}disquiet \Rightarrow \exists Q \in PT_a.Participants$$
$$(Q.state = active\text{-}disquiet \vee Q.state = passive\text{-}disquiet) \wedge Ancestor(Q,P))$$

Let us assume that $P.state$ = *passive-disquiet*. If $Q.state$ = *active-disquiet*, then, as from $Ancestor(Q,P)$ we have $Ancestor^*(Q,P)$, the result we want to prove holds immediately.

Otherwise, $Q.state$ = *passive-disquiet* and, since $Ancestor(Q,P)$, the number of ancestors of Q is less or equal to m, and then by the induction hypothesis,

$$\exists R \in PT_a.Participants$$
$$R.state = active\text{-}disquiet \wedge Ancestor^*(R,Q))$$

So, the participant R has the desired property: R is active-disquiet and, because $Ancestor^*(R,Q))$ and $Ancestor(Q,P))$, $Ancestor^*(R,P))$. □

Now, we proceed with the proof of theorem 7.1. Suppose that $\forall P \in PT_a.Participants\cdot$ $P.state \neq$ *active-disquiet* and $\exists P \in PT_a.Participants$ such that $P.grey\text{-}set \neq 0$ or $P.local\text{-}steps \neq 0$, that is, there may be a scan-request in transit. From the state transition diagram, this implies that $P.state$ = *passive-disquiet* $\vee$ $P.state$ = *active-disquiet*. If $P.state$ = *passive-disquiet*, by invariant 4.4, there must be an *active-disquiet* participant. So we have a contradiction. □

7.2.3 Mutator Concurrency

In this section we extend the proof to the model allowing for mutator concurrency. We prove that:

1 The **Remote Barrier** allows correct distributed termination detection of the scan phase.

2 Safety property 2 on page 157 holds.

The proof that the **Remote Barrier** preserves the invariant 7.1 follows immediately from the state case analysis in section 6.2. We showed that a **Remote Barrier** can only be triggered by an *active-disquiet* process. Hence, at least that process is responsible for all subsequent requests generated by the barrier. From this it follows that theorem 7.1 is trivially true.

Next we prove safety property 2. Recall the proof of the partial tracing algorithm on page 157. Let us turn to prove proposition 1 on page 158 and account for mutator concurrency. To show that concurrent mutator activity preserves this proposition, and consequently the safety property, we prove first invariants 6.1, 6.2, 6.3, 6.8 and 6.9.

Proof of Invariant 6.1 This invariant states: After SI(P), no red exit-items are reachable from the *local-scan-root-set* at P.

Suppose that after the initial step at process P there is an exit-item Ex_y such that *red*(Ex_y). By mark-red local steps, Ex_y must be reachable from a red entry-item Ei_z, where Ei_z.*red-list* = Ei_z.*entry-list*. By mark-red remote steps and definition 4.1, all Ex_z from which Ei_z is reachable must be red. In order for Ei_z to be referenced from the *local-scan-root-set*, some process P holding a red Ex_z for Ei_z must have invoked Ei_z's corresponding object. In that case, a **Remote Barrier** must have been triggered in Ei_z through Ex_z and painted Ei_z and all exit-items in Ei_z.*exits* green, including Ex_y. Consequently, the invariant is true. □

Before proving invariant 6.2, we prove the auxiliary invariant 6.3.

Proof of Invariant 6.3 Such invariant states that: A red entry-item Ei_z, where Ei_z.*red-list* = Ei_z.*entry-list*, cannot receive an insert message unless there is a scan request in transit to that Ei_z.

We have Ei_z such that Ei_z.*red-list* = Ei_z.*entry-list*. By mark-red remote steps and definition 4.1, all Ex_z from which Ei_z is reachable must be red. By the mark-red restriction, at the time every Ex_z was marked-red, $Ex_z \notin$ *translist*. This implies that there were no messages in transit holding a reference to Ei_z's corresponding object. Consequently, for Ei_z to receive an insert message, some process P holding a red Ex_z for Ei_z must have transmitted a reference to Ei_z to another process. But, in this case a **Remote Barrier** would have been triggered on Ex_z and a scan request would have been sent and received (by invariant 6.4) at Ei_z.

Proof of Invariant 6.2 This invariant states: After SI(P), no red entry or exit-items become new members of the *local-scan-root-set*.

Suppose that after the initial step at process P there is an entry-item Ei_z such that *red*(Ei_z). Hence, Ei_z.*red-list* = Ei_z.*entry-list*. For Ei_z to become a new member of the *local-scan-root-set*, it must receive an insert message such that Ei_z.*red-list* $\neq$ Ei_z.*entry-list*. By invariant 6.3, there must be a scan request in transit to Ei_z. Hence, Ei_z and all exit-items in Ei_z.*exits* will be painted green. Consequently the invariant is true. □

Proof of Invariant 6.8 This invariant states that:

$$\forall Ex_y \cdot (path(Roots, Ex_y) \wedge suspect(Ex_y) \Rightarrow dirty(Ex_y))$$

This invariant is true immediately after **identify-suspects** because, by post-condition 5.1, $path(Roots, Ex_y) \wedge suspect(Ex_y)$ is a contradiction. After **identify-suspects**, only external messages may change the reachability of suspect exit-items, so there must have been Ei_z such that $path(Ei_z, Ex_y)$. By post-condition 6.6, $Ex_y \in Ei_z.exits$.

In order to make a new path to Ex_y, an external mutator message must have arrived at Ei_z after **identify-suspects**. By post-condition 6.5 and post-condition 6.7, $dirty(Ei_z) \wedge dirty(Ex_y)$.

We conclude that the invariant is always true. □

Proof of Invariant 6.9 This invariant states that:

$$\begin{aligned} \forall Ei_z, Ex_y \cdot path(Ei_z, Ex_y) \wedge suspect(Ex_y) \Rightarrow \\ Ex_y \in Ei_z.exits \vee dirty(Ex_y) \end{aligned}$$

By post-condition 5.1, Ex_y was not reachable locally in the last **identify-suspects**. If $path(Ei_z, Ex_y)$ existed before **compute-graph**, by post-condition 6.6, $Ex_y \in Ei_z.exits$. If not, as above, after **identify-suspects** only external messages may change the reachability of suspect exit-items, so there must have been Ei_u such that $path(Ei_u, Ex_y)$. By post-condition 6.6, $Ex_y \in Ei_u.exits$. In order to make a new path to Ex_y, an external mutator message must have arrived at Ei_u after **identify-suspects**. By post-condition 6.5 and post-condition 6.7, $dirty(Ei_z) \wedge dirty(Ex_y)$.

We conclude that the invariant is always true. □

Recall the partial tracing algorithm proof on page 157. Here we want to prove the property:

$$\forall Ei_y \cdot (red_a(Ei_y) \Rightarrow \neg live(Ei_y))$$

Suppose that $red_a(Ei_y) \wedge live(Ei_y)$ at the begin of sweep phase. Then,

$$\forall Ei_y(red_a(Ei_y) \wedge live(Ei_y) \Rightarrow \tag{13}$$

$$(\exists Q \notin PT_a.Participants \wedge \exists Ex_z \in Q.exit\text{-}table \wedge path(Ex_z, Ei_y) \vee \tag{14}$$

$$(\exists R \in PT_a.Participants \wedge \exists Ex_u \in R.exit\text{-}table \wedge \tag{15}$$

$$path(Ex_u, Ei_z) \wedge live(Ex_u)) \tag{16}$$

Suppose (14) holds, but ((15) $\wedge$ (16)) do not, and suppose $path(Ex_z, Ei_y)$ existed before the initial scan step. As we showed in the non-concurrent model (see page 157), Ei_z is a member of the *local-scan-root-set* and painted green by scan initial step. Thus, $red_a(Ei_y) \wedge live(Ei_y)$ is a contradiction.

Now, suppose $path(Ex_z, Ei_y)$ did not exist before the initial scan step. Once $red_a(Ei_y)$, by invariant 6.2, Ei_y could not be a new member of the *local-scan-root-set* after initial step. A **Remote Barrier** on Ei_y preserves the invariant.

Now suppose (15) and (16) hold, but (14) does not hold. As we have already shown, Ex_u is red and by hypothesis Ex_u is live. Hence

$$live(Ex_u) \Rightarrow \tag{17}$$

$$(\exists r \in Roots(R) \wedge path(r, Ex_u)) \vee \tag{18}$$

$$(\exists Ei_r \in R.entry\text{-}table \wedge path(Ei_r, Ex_u) \wedge live(Ei_r)) \tag{19}$$

Suppose (18) holds, but (19) does not. Thus, $\exists r \in Roots(R) \wedge path(r, Ex_u)$.

Suppose such a path existed before the initial step at R. If $\neg suspect(Ex_u)$, Ex_u would have been a member of the *local-scan-root-set* and painted green. If $suspect(Ex_u)$, by invariant 6.8, Ex_u is dirty, and so Ex_u would have been a member of the *local-scan-root-set* and painted green. Thus, $red_a(Ei_y) \wedge live(Ei_y)$ is a contradiction.

If such path did not exist before the initial step at R, by invariant 6.1, and assuming safety of 'Advanced local steps', Ex_u would have safely been greened by a **Remote Barrier**.

Now, suppose that (19), but (18) does not hold. As we have shown in the proof of the partial tracing algorithm on page 157, Ex_u must be red. Thus red live items are only reachable from other red live items, and not from any root. So they are not live.

To conclude this proof, we have to prove that the 'Advanced local steps' are safe. That is, after initial step at process P:

$$\forall Ei_z, Ex_y \cdot red(Ei_z) \wedge red(Ex_y) \wedge path(Ei_z, Ex_y) \Rightarrow Ex_y \in Ei_z.exits$$

From invariant 6.9, $Ex_y \in Ei_z.exits \vee dirty(Ex_y)$. By definition of the *local-scan-root-set*, if $dirty(Ex_y)$, Ex_y would have been painted green by the initial step. Thus, we have that $Ex_y \in Ei_z.exits$.

We conclude the proof that no red objects are live at the begining of the scan phase.

7.2.4 Co-operative partial tracings

In this section we extend the proof to the scalability model, that is, to the model that allows co-operative partial tracings. We prove the following safety properties:

Co-safety.1 The **Token Algorithm** safely detects termination of scan phase. That is, when the initiator PT_a receives back the token it has initiated, $token_a$, PT_a may safely switch to the sweep phase.

Co-safety.2 Safety property 2 on page 157 holds.

First we prove property Co-safety.1, which is described by the following theorem:
Theorem 7.2

$$\forall PT_a(\text{receive-token}_a(token_a) \Rightarrow \text{``no scan requests in transit for } PT_a\text{''})$$

If PT_a receives back the token it initiated, $token_a$, there are no more requests in transit for PT_a.

The proof of this theorem is based on the proof of the following theorem:

Theorem 7.3

$$\forall PT_a \cdot (\forall P \in \mathit{PARTICIPANTS}(PT_a) \cdot P.\text{state} \neq \text{active-disquiet}$$
$$\Rightarrow \text{"no scan requests in transit for } PT_a\text{"})$$

Where,

$\mathit{PARTICIPANTS}(PT_a) = \bigcup_{PT_b \in \text{Dependent}^*(PT_a)} PT_b.\text{Participants}$

If no participant of PT_a and no participant of any PT_b transitively responsible for PT_a is active-disquiet, there cannot be any scan request in transit for PT_a.

We first prove the following invariant:

Invariant 7.4

$$\forall P \in PT_a.\text{Participants } (P.\text{state} = \text{passive-disquiet} \Rightarrow$$
$$\exists PT_b \cdot \text{Dependent}^*(PT_a, PT_b) \wedge \neg \mathit{partial\text{-}terminated}(PT_b))$$

The proof follows by induction on the number n of partial tracings in the *Dependent** relation of PT_a.

Case $n = 1$, that is $\mid \mathit{Dependent}^*(PT_a, PT_b) \mid = 1$, PT_a is the only partial trace responsible for PT_a. The formula is trivially true by definition of $\mathit{partial\text{-}terminated}(PT_a)$.

Case $n = m + 1$. It follows from the state transition diagram that,

$$\mathit{P.state} = \mathit{passive\text{-}disquiet} \Rightarrow \exists Q \in PT_b.\mathit{Participants}\cdot$$
$$(\mathit{Dependent}(PT_a, PT_b) \wedge (Q.\mathit{state} = \mathit{active\text{-}disquiet} \vee Q.\mathit{state} = \mathit{passive\text{-}disquiet}))$$

Let us assume that $\mathit{P.state} = \mathit{passive\text{-}disquiet}$. If $Q.\mathit{state} = \mathit{active\text{-}disquiet}$, then $\neg\mathit{partial\text{-}terminated}(PT_b)$ by definition 5.7. As from $\mathit{Dependent}(PT_a, PT_b)$ we have $\mathit{Dependent}^*(PT_a, PT_b)$, the result we want to prove holds.

Otherwise, $Q.\mathit{state} = \mathit{passive\text{-}disquiet}$, and since $\mathit{Dependent}(PT_a, PT_b)$, by the induction hypothesis,

$$\exists PT_c \cdot \mathit{Dependent}^*(PT_b, PT_c) \wedge \neg \mathit{partial\text{-}terminated}(PT_c)$$

Then,

$$\exists PT_c \cdot \mathit{Dependent}^*(PT_a, PT_c) \wedge \neg \mathit{partial\text{-}terminated}(PT_c)\Box$$

Let us prove theorem 7.3. By the state transition diagram, if there are scan-requests in transit for PT_a, there must be P in PT_a or $PT_b \in PT_a.\mathit{Responsibles}$ where $P.\mathit{state} = \mathit{passive\text{-}disquiet} \vee P.\mathit{state} = \mathit{active\text{-}disquiet}$. By invariant 7.4 this leads to a contradiction. Thus, this theorem holds.

To complete this proof we have to prove theorem 7.2. Recall that in each individual PT_b scan report phase, the initiator determines the condition $\mathit{partial\text{-}terminated}(PT_b)$ (definition 5.7). So, it suffices for the **Token Algorithm** to detect this condition for all PT_b such that $\mathit{Dependent}^*(PT_a, PT_b)$ holds. PT_a receives token_a back when $\mathit{token}_a.\mathit{next}$ is empty. Thus, we show that

$$\forall PT_a \cdot (\mathit{token}_a.\mathit{next} = \emptyset \Rightarrow \tag{20}$$

$$(\forall PT_b \cdot \mathit{Dependent}^*(PT_a, PT_b) \Rightarrow \mathit{partial\text{-}terminated}(PT_b))) \tag{21}$$

From its definition and $\mathit{Dependent}^*$ definition (page 121), the **Token Algorithm** builds PT_a's $\mathit{Dependent}^*$ such that

$$\begin{aligned}
&\forall PT_b.\mathit{Dependent}^*(PT_a, PT_b) \Rightarrow\\
&\quad(\mathit{partial\text{-}terminated}(PT_b) \wedge PT_b \in \mathit{token}_a.\mathit{terminated})\\
&\quad\vee(\mathit{partial\text{-}terminated}(PT_b) \wedge PT_b \notin \mathit{token}_a.\mathit{terminated} \wedge\\
&\qquad(\exists PT_c.\mathit{Dependent}^*(PT_c, PT_b) \wedge PT_c \in \mathit{token}_a.\mathit{next}))\\
&\quad\vee(\neg \mathit{partial\text{-}terminated}(PT_b) \wedge (PT_b \in \mathit{token}_a.\mathit{next} \vee\\
&\qquad(\exists PT_c.\mathit{Dependent}^*(PT_c, PT_b) \wedge PT_c \in \mathit{token}_a.\mathit{next}))
\end{aligned}$$

Suppose that there is PT_b such that $Dependent^*(PT_a, PT_b)$ and $\neg partial\text{-}terminated(PT_b)$ and $token_a.next = \emptyset$. By definition of the **Token Algorithm**, PT_b is only inserted in $token_a.terminated$, if $partial\text{-}terminated(PT_b)$. Thus, $PT_b \notin token_a.terminated$. Consequently, by the above proposition, $PT_b \in token_a.next \vee (\exists PT_c.Dependent^*(PT_c, PT_b) \wedge PT_c \in token_a.next)$. In either case, $token_a.next \neq \emptyset$. Thus, by the assumption of proposition 20, we have arrived at a contradiction.

Next we prove property Co-safety.2. We rewrite propositions 3 on page 158 and 4 on page 158 accounting for co-operative partial tracings. We introduce the term $super\text{-}group_a$ for the union of $PT_b.Participants$ such that $Dependent^*(PT_a, PT_b)$.

$$\forall Ei_y(red_a(Ei_y) \wedge live(Ei_y) \Rightarrow \tag{22}$$

$$(\exists Q \notin super\text{-}group_a \wedge \exists Ex_z \in Q.exit\text{-}table \wedge path(Ex_z, Ei_y)) \tag{23}$$

$$\vee(\exists R \in super\text{-}group_a \wedge Ex_u \in R.exit\text{-}table \wedge path(Ex_u, Ei_z) \wedge live(Ex_u)) \tag{24}$$

Suppose that (23) holds but (24) does not. As we have already shown, by the reference listing protocol, proposition 5 on page 158 holds. Also,

$$Q \notin super\text{-}group_a \Rightarrow (\forall PT_b \in super\text{-}group_a \cdot \neg red_b(Ex_z)) \tag{25}$$

because, Ex_z was not visited by by the mark-red phase of any member of $super\text{-}group_a$. By definition 4.1,

$$(path(Ex_z, Ei_y) \wedge (\forall PT_b \in super\text{-}group_a \cdot \neg red_b(Ex_z))) \Rightarrow Q \notin Ei_y.red\text{-}list \tag{26}$$

That is, Q cannot be a member of $Ei_y.red\text{-}list$ because it does not hold a red exit-item signed by any $PT_b \in super\text{-}group_a$ to Ei_y. Consequently, proposition 8 on page 158 holds. From this it follows that Ei_y must have been painted green by scan initial step. Thus, $live(Ei_y)$ and $red_a(Ei_y)$ is a contradiction. In this case, proposition (24) must hold, that is, there must be an exit-item Ex_u belonging to $super\text{-}group_a$ from which Ei_y is reachable and $live(Ex_u)$.

We rephrase the conclusion from 9 on page 159 as "Ex_u must be red_b for some PT_b and by hypothesis Ex_u is live". But, by *mark-red* step, $PT_b \in PT_a.Responsibles$ and hence a member of the *responsibles* set of the initiator of PT_a (every $P \in PT_a.Participants$ reports a list of responsibles it is aware of to its initiator at the end of mark-red phase). PT_a does not enter sweep phase until PT_b (and other responsible partial tracings) are *partial-terminated* (recall **Token Algorithm**). Assuming correctness of **Token Algorithm**, PT_a has detected this condition (once it is at the begining of sweep phase). Consequently, Ei_y cannot receive one scan request through Ex_u.

7.3 Liveness

We want to show that our system cannot deadlock or livelock. Mark-red and scan phases do not have any critical regions or exclusive holding of resources. Consequently, they cannot deadlock.

The mark-red phase does not need to visit the complete transitive referential closure of suspect entry-items, hence it is guaranteed that it terminates. On the other hand, the scan phase must paint green all red live objects. We guarantee that it performs a finite number of steps because there are a finite number of red items. Also, the token algorithm performs a finite number of steps. Finally, the colouring process is monotonic, that is, the colour of an item may change only from white to red and from red to green. This prevents the responsible's scan phase and the dependent's mark-red phase from chasing each other, that is, it avoids race conditions between mark-red and scan requests. We conclude that our system also does not livelock.

Now, we show that a complete solution can be achieved, that is, all garbage objects are eventually collected. The completeness argument is that every garbage object is eventually a suspect object. Both, the *locally reachable* and *distance heuristic* are complete heuristics as every garbage object is non-locally reachable, and the distance of every garbage object increases infinitely. Hence, assuming that every process performs a local collection regularly, every garbage object will become non-locally reachable or every garbage object will eventually cross the distance threshold.

Now we define which conditions must be met for a garbage object (member of a garbage cycle) to not be collected.

Consider a distributed garbage cycle. Suppose that PT_z is initiated at any object z which is member of that cycle. For an object z to not be collected, there must be an external reference to the cycle. That is, there must be an object y that is not involved in PT_z, and z is transitively reachable from y. Object y must be garbage, otherwise z would not be garbage. In this case, we may say that y will eventually be a member of a partial tracing, for example PT_y. Conceptually, one of these three situations will happen eventually:

1 PT_y covers z, and there are no external references to PT_y's suspect subgraph. z will be eventually collected by PT_y.

2 PT_z transitively dependent on PT_y, and there are no external references to PT_y's suspect subgraph. PT_y will eventually succeed and collect object y. If *Dependent* (PT_z, PT_y), there are no more external references to z. Consequently z will be collected by PT_z. If *Dependent** (PT_z, PT_y), there is PT_u such that *Dependent** (PT_z, PT_u) and *Dependent*(PT_u, PT_y). PT_u will eventually succeed, and by induction on the number of PT_z's responsibles, object z will eventually be collected by PT_z.

3 PT_y is transitively responsible for PT_z and PT_z is transitively responsible for PT_y, and there are no external references to PT_z's suspect subgraph and PT_y's suspect subgraph. If *Responsible*(PT_y, PT_z) and *Responsible*(PT_z, PT_y), PT_y and PT_z eventually meet each other. If the beginning of PT_y's scan phase and PT_z's scan phase are synchronised, there are no more external references to z. Consequently z will be collected by PT_z. Now assume that:

   (a) If there is PT_u such that *Responsible** (PT_y, PT_u) and *Responsible*(PT_u, PT_z), PT_u will eventually meet PT_z. As *Responsible** (PT_y, PT_u), we conclude that PT_y will eventually meet PT_z.

   (b) If there is PT_v such that *Responsible** (PT_v, PT_z) and *Responsible*(PT_v, PT_y), PT_v will eventually meet PT_y. As *Responsible** (PT_z, PT_v), we conclude that PT_z will eventually meet PT_y.

   By induction on the number of partial tracings between PT_z and PT_y, if the beginning of PT_y's scan phase and PT_z's scan phase are synchronised, there are no more external references to z. Consequently z will be collected by PT_z.

7.4 Summary

In this chapter we presented a proof of several aspects of our algorithm. We first used a simplified model that does not account for mutator concurrency or concurrent partial tracings, and we assumed that distributed termination is correctly detected. Then, we extended the proof to the other aspects. We showed that the distributed termination protocol detects safely the end of the mark-red and scan phases, in order for a partial tracing to proceed to the following phase.

Then, we extended the proof to the mutator concurrency model and proved that mutator actions do not affect termination detection and safety. Finally, we showed that two partial tracings may be simultaneously active in the same or a connected sub-graph, that they may co-operate without compromising each other's safety property.

Finally, we showed that our algorithm is live in the sense that all garbage cycles within a suspect subgraph are eventually collected, and that eventually all garbage objects are suspect objects, thus, that eventually all garbage cycles are covered by one or more co-operative partial tracings.

Chapter 8

Implementation over Network Objects

We have implemented and tested a prototype of the concurrent version of the cyclic garbage collector algorithm presented in chapter 4. The implementation was carried out to test its feasibility and identify its weaknesses and strengths. We mainly focus on implementation strategies related to our efficiency goals.

We chose the Network Objects system as a vehicle because it provides an implementation of the reference listing protocol (section 3.4.4), on which our system is based. However, our algorithm can be adapted to other systems that provide an acyclic garbage collector based on reference listing.

First we give an overview of the architecture of Network Objects system, mainly focusing on the local and acyclic distributed garbage collector, remote invocation and marshaling of network objects. This is important, to understand the implementation strategies of our system. Then, we briefly report on our prototype implementation. We start by describing the modules of the implementation and overviewing some solutions. Then, we explain each solution, reporting the main problems encountered.

We also discuss the implementation of extensions, mainly to cover those aspects of our system introduced in section 5.

8.1 An Overview of Network Objects

Network Objects is a distributed object-based programming system for Modula-3 that allows the development of programs that communicate over a network, while hiding the details of network programming (Birrel et al. 1993).

Network Objects provide a means to incorporate remote procedure call (Birrel and Nelson 1984) in an object-based programming style. An object consists of a data record and a set of methods that can be invoked on the object. The process that allocated the network object is called its *owner*, and the instance of the object at the owner is called the *concrete* object. A network object can be transmitted between processes by reference and then shared by processes in a distributed system. Processes holding a reference to a concrete object are called *clients*. The client and owner can run on different machines or in different processes on the same machine. Network objects may be transmitted from one client to another as well as from the owner to a client.

Network Objects allows a concrete object to be accessed from another process in the same way as if it was local to that process. A remote reference in the client actually points to a *surrogate* object. The surrogate contains an handler and has methods which perform remote procedure calls on the concrete object in the remote owner process.

References for a network object may be marshaled from one process to another during method invocation as arguments or results. A network object is marshaled by transmitting its *wireRep*, which consists of a unique identifier for the owner process — ProcessID — plus the index of the object at the owner — ObjID.

Each process maintains an *object table* that contains references to all its surrogates and all its concrete objects for which some process holds a surrogate. If a concrete network object is present in the object table, the runtime system says that it is *exported*, otherwise is *unexported*.

The heap of a Modula-3 program is managed by garbage collection. Network objects are managed by a distributed garbage collector based on the reference listing scheme presented in section 3.4.4 (Birrel et al. 1993). In the following sections we will explain how the the partitioned model presented in chapter 3 is implemented in the Network Objects System, including the local collector for Modula-3.

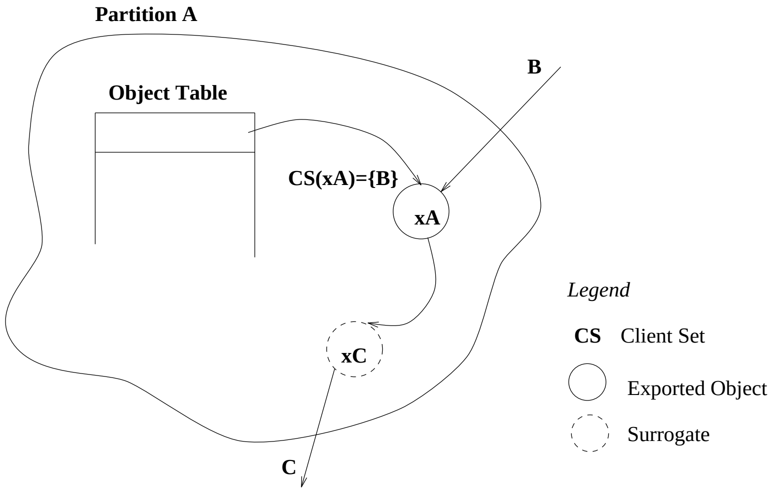


Figure 34: Network Object Model for Garbage Collection

8.1.1 Implementation of the Garbage Collection Model

The Network Object Model for garbage collection divides the distributed system global address space into several partitions that are held by the different processes[1] of the distributed system. Each process (or the corresponding partition) can be identified unambiguously, and we identify processes (partitions) by upper-case letters, *e.g.* A, B, ..., and objects by lower-case letters suffixed by the identifier of the process to which they belong, *e.g.* xA, xB, ...

From the garbage collector's point of view, mutators periodically exchange messages in addition to performing local computations independently of other mutators in the system. They allocate objects in the local heaps and transfer data between processes, which may include references to objects. An object referenced from another process is called an *exported* object. Each process is garbage collected independently from the other processes.

The Network Objects Model for garbage collection, represented in figure 34, consists of:

- A set of processes containing objects that may point to objects on the same processes or to objects in other processes.
- *Surrogate* objects that record an outgoing reference. A process may hold at most

[1] A Modula-3 process.

one surrogate for a given object, in which case all references in the process to that object point to the surrogate. In the Network Objects implementation a surrogate object has other roles apart from garbage collection. It also encapsulates a marshaler as we shall see.

A surrogate implements an exit-item described in chapter 3.

- An *object table* for each process that records references from outside the process to objects within the process. If process B has a surrogate pointing to an object in process A, the object table at A has an entry for the referenced object (see figure 34). To ensure this, a concrete object is entered into its owner's table when it is first marshaled; it remains there until the distributed garbage collector detects the deletion of its last surrogate.

  The object table also contains entries for all surrogates that exist in the process. It maps the wireRep for a remote object to the unique local surrogate for that object.

  The entry table and exit table described in chapter 3 are implemented by the Network Objects's object table.

- A *Client-set* for each exported object which contains identifiers for all the processes that have a surrogate for that object. The following invariant is always maintained (see figure 34):

  NO Invariant 8.1 *If there is a surrogate for object xA at client B, then $B \in xA$.*Client-set.

  The client-set implements the entry-list described in chapter 3.

8.1.2 Local Garbage Collection

Local collections are based on tracing from local roots — the stack, registers, global variables and also the object table. We shall refer to the object table entries as *OT roots*. The object table is considered a root by the local collector in order to preserve objects reachable only from other processes.

The Modula-3 local collector is an adaptation of the algorithm presented in (Bartlett 1988) and modified to be incremental, generational, and VM-synchronised by John DeTreville. The relevant features of this local collector for our implementation are its conservative and incremental nature. The local collector is a mostly copying collector (presented in section 2.4.3) that uses a page-wise *black-only* read-barrier supported by the operating system's memory protection hardware (Appel et al. 1988) to trap heap accesses by the mutator.

The primary differences between the mostly copying algorithm and the classical copying algorithm (recall section 2.3) lie in how it finds its root set and how it organises the heap.

The heap of a Modula-3 process consists of a number of pages that may appear anywhere in the heap. Associated with each page is a space identifier, which identifies the "space" that objects on the page belong to: *previous*, *current* or *free*. *Previous* and *current* spaces are equivalent to *from-space* and *to-space* of a classical copying collector respectively. The *free* space represents the free pages. This level of indirection means that there are two ways to move an object from the *previous* to *current* space. It can be copied to a fresh *current* space page as in a classical copying algorithm, or the space identifier of the page containing it can be changed to *current*. Since pages do not have contiguous addresses, they are held in a list for each space.

As described in section 2.4.3 the mostly copying algorithm is a conservative collector. It assumes no knowledge of registers, stack or global variables layouts, but it does assume that all pointers in the heap can be found accurately. Using this distinction, the local collector will divide all accessible objects in the heap into two classes: those which might have a direct reference from the root set, and those which do not. The former objects are left in place — the space identifier is changed for that page — and the latter objects are copied into a compact area of memory.

The mutator works on pages identified with *current* space. The local collector operates in three phases. When a garbage collection cycle is initiated, in a first phase, the *current* space becomes *previous* space, that is, the space identifier of pages in *current* space is changed to *previous*. Next, the root set is scanned conservatively for potential pointers in the heap. When a pointer into a heap page is found, that page is "promoted" to the *current* space, that is, the space identifier of the page is changed to *current*. The page is also added to the tail of the list of pages in *current* space. Thus, the objects which might have references in the root set are now part of the *current* space, but their address has not changed. Promoting pages is a conservative action, because garbage objects may also be retained on promoted pages.

During the second phase, all pages in the *current* space list are scanned for references into the *previous* space. Each object reached is copied into a fresh *current* space page (recall section 2.3) that will also be queued and scanned. When the scanning is complete, in a third phase, pages in *previous* space are freed, that is, they are appended to the free-list for new allocations, and garbage collection is completed.

Until now, we have assumed a *stop-the-world* local collection, hence we did not account for incremental mutator activity (recall section 2.4.1). The algorithm described above is augmented with a page-wise *black-only* read-barrier supported by the operating system's memory protection facilities (Appel et al. 1988) in order to synchronise the mutator and the collector. The result is showed in figure 35. We use the following syntax: '=' for assignment, '==' for equality.

Recalling the tricolour abstraction introduced in section 2.4.1, the following invariant is maintained during the local collection:

NO Invariant 8.2 *The mutator is only allowed to see black objects.*

When the mutator allocates memory for an object, it checks the occupancy of the heap. If it is bigger then a certain *threshold* (recall section 2.4.1), some amount of garbage collection, depending on the collector's state, is performed.

In figure 35 we represent three states for a local garbage collection cycle. The Modula-3 garbage collector has more states, but they are irrelevant for our purpose. Whenever a local garbage collection starts — state zero — the space identifier of each page in the *current* space is changed to *previous* (as we said, this means that the *current* space becomes the *previous* space). Also, pages referenced from the stack are promoted to the *current* space — `promote(page)`. The promoted pages are inserted in a queue

[2] We simply assume a new page as a result of scanning an object. A nil page will be return if none object is copied to the *current* space. The new page space is set to *current*.

```
State == zero, one, two
Space == current, previous, free
State = zero
current_queue = empty

gc() =
    State == zero => for page where page.space == current do
                        page.space = previous
                     for R in Roots do
                         promote(page(R))
                     State = one
    State == one  => page = pop(current_queue)
                     scanPage(page)
                     if current_queue == empty then
                         State = two
    State == two  => for page where page.space = previous do
                         page.space = free
                     State = zero
    if current_queue ≠ empty then
      for page in current_queue do
        protect(page)

fault_handler() =
    forever
        thread, page = WaitForTrappedThread()
        LOCK memory
            scanPage(page)
        ResumeThread(thread)

promote(page) =
      if heap_bottom ≤ page ≤ Heap_top
         and page.space == previous then
          page.space = current
      push(page, current_queue)

scanPage(page) =
    if gray(page) then
        if protected(page) then
            unprotect(page)
        for obj on page do
            newpage = scan(obj)^2
            if newpage ≠ nil then
              push(newpage, Tospace_queue)

allocate(n) =
    LOCK memory
          ⋮
        if threshold then
            gc()
```

Figure 35: Modula-3 local collector algorithm

to be scanned for pointers in the previous space. The mutator is then resumed by the collector.

When the collector resumes the mutator, it sets the virtual-memory protection of the unscanned area's page to 'no access' (coloured grey) — `protect(page)`. Whenever the mutator attempts to access an object on a protected (grey) page, the page-access trap is triggered and the fault is caught by the collector — `fault-handler()`. The collector scans the objects on that page (colouring the page black). New *current* space pages are then enqueued and protected as well. Then, the collector unprotects the page and resumes the mutator. To the mutator all objects appeared to be in the *current* space.

In state one the collector pops a page from the current space queue and scans it. If the queue is empty, the collector moves to state two in the next call, otherwise, it stays in state one. Again, pages in the current space queue are protected (greyed) to catch mutator accesses on grey objects.

Finally, in state two — no grey pages — pages left in previous space are freed.

8.1.3 Network Objects Runtime System

The Network Objects runtime system is implemented by a *special object* that provides methods to support, among other things, distributed garbage collection. There is one such concrete object per process. In addition there are potentially many *special object* surrogates used to invoke corresponding methods in different processes, that is, when the runtime system of two different processes interact, they will do so through the invocation of each others *special object* methods.

8.1.4 Remote Invocation and Marshaling of Network Objects

Argument and results values are communicated by *marshaling* them into a sequence of bytes, transmitting the bytes from one process to another, and then unmarshaling them into values in the receiving process, in a remote method invocation.

In this section, we will describe the implementation of remote methods invocation and network object's marshaling. For each network object there must be a client and server stub to support remote method invocation using Remote Procedure Call (Birrel and Nelson 1984). The client stub is a local surrogate object. The client process actually invokes its methods, which in turn implement the remote invocation. On the server side, the stub consists of a single procedure for that object — the *Dispatcher* — that is called to dispatch remote invocations.

Next we describe a simplified sketch of the procedure calls performed by a client to make a remote method invocation to a method of `obj`. A remote object invocation can be viewed as an exchange of messages between client and server. The messages are exchanged via a connection `c` established between the client and the server. The contents of the message includes the number that identifies the method to be invoked at the server-side, and the arguments of that method. On completion of the remote invocation the server sends a message with the method results.

```
BEGIN SurrogateMethod
  <marshal to c the number of the method>
  <marshal to c the method arguments>
  result := AwaitResult(c);
  <unmarshal from c the method results>
END SurrogateMethod
```

Next we consider the server-side stub, which consists of a dispatcher procedure for each network object. The dispatcher is called by the Network Object runtime system when it receives a remote object invocation for a network object. The dispatcher procedure is responsible for unmarshaling the method number and any arguments, invoking the concrete object's method, and marshaling any results.

Here is a simplified sketch of a typical dispatcher for an object `obj` through a connection `c`:

```
BEGIN Dispatcher
   <unmarshal from c the method number>
   <unmarshal from c the method arguments>
   <call the appropriate method of obj>
   <marshal to c the method result>
 END Dispatcher
```

8.1.5 Acyclic Garbage Collection

The acyclic garbage collector is responsible for always safely maintaining the invariant 8.1. In this section we will describe the acyclic garbage collector operations responsible for maintaining the object table entries and the client-sets of exported objects.

The *insert* and *delete* control messages introduced in section 3.4.4 are implemented in the Network Objects system by a *dirty-call* and a *clean-call* respectively.

The potential race conditions between concurrent transmission and deletion of a same network object (its wireRep) is avoided (recall section 3.4.4) by preventing the remote reference from being reclaimed at the sender process. This is done by making sure that the object's *client-set* remains non-empty while its wireRep is being transmitted. When the sending process P is the object's owner, this is accomplished by putting P into the object's *client-set* until P has received the *dirty-call*. When P is not the object's owner, it must have a surrogate for it. This surrogate is kept reachable until it is known that the object's owner has received the *dirty-call* (recall section 3.4.1). This happen when P receives an acknowledgement from the receiver process. In this overview we skip implementation details, and concentrate on the implementation of the dirty-call mechanisms since it is important for our prototype implementation.

Our system is then implemented above a safe reference listing protocol.

dirty call Whenever a process A exports a network object O to process B (as a result of A marshaling O to B), (i) if A is the owner of O, O is inserted in A's object table; when B receives the imported *wireRep*, it creates a surrogate for O and sends a *dirty call* to A; upon receipt of the *dirty call*, B is inserted in O's *client-set*, (ii) otherwise, when B receives the imported *wireRep*, it creates a surrogate for O and sends a *dirty call* to the owner of O; upon receipt of the *dirty call* by the owner of O, B is inserted in O's *client-set*.

The following sequence of instructions implements the above actions for an object `obj` with wireRep `wireRep`. `MarshalObj` is executed at process A. It sends object `obj`'s wireRep `wireRep` to process B. Notice that A may be either the owner of `obj` or have a surrogate to `obj`. `concrete(obj)` returns true if `obj` is a concrete object.

```
BEGIN MarshalObj
  if concrete(obj) then
    <insert obj in objTbl>
  <send wireRep to receiving process>
END
```

Process B executes `UnmarshalObj`. It creates a surrogate for the new object and sends a *dirty-call* to the owner of the object identified by the wireRep — `ProcessID.specialObj(ProcessID(wireRep)).dirty-call(wireRep)` corresponds to the remote invocation of the `dirty-call` method of the owner's *special object*.

```
BEGIN UnmarshlObj
  <receive wireRep>
  if not objtbl.find(wireRep, obj) then
    <create surrogate with wireRep>
    specialObj(ProcessID(wireRep)).dirty-call(wireRep)
END
```

When the owner of `obj`[3] receives the *dirty-call* it inserts process B in `obj`'s *client-set*.

```
BEGIN DirtyCall
  <insert sending process in obj.client-set>
END
```

clean call Whenever a surrogate for O is reclaimed by the local garbage collector at process B, B sends a *clean call* to O's owner. Upon reception of the *clean call*, B is deleted from O's *client-set*. When an object's *client-set* becomes empty, the reference to the object is removed from the object table so that the object can be reclaimed subsequently by its owner's local collector.

[3] As we explained it can be the sending process or other process to which the sending process holds a surrogate.

8.2 Prototype Implementation

In this section we describe the implementation of the partial tracing over the Network Objects system. We start by taking an overview of the prototype. At a high-level, for each process we define a set of threads that implement the partial tracing algorithm. At a lower-level, we sketch the implementation of policies inherent in the algorithm and, at a third level, we describe which implementation strategies we have adapted to the runtime system of Network Objects. Then we look in more detail at the implementation of the described prototype components.

We defined a *tracing* and a *sweep* thread that implement the mark-red, scan and sweep phases, plus additional threads acting as *event handlers* driven by external events. The event handlers are implemented as methods of a *DGCobj* (Distributed Garbage Collector object) object that we have implemented as a part of the runtime system object — the *special object*. Different DGC objects in different processes communicate with each other by remote procedure call.

Each partial tracing is implemented by a *PTobj* (Partial Tracing object)[4]. The *PTobj* state holds the state described in definition 5.3 on page 114 and the *PTobj* methods are invoked by the DGCobj methods to perform the required action for each event handler.

The termination of each phase is detected by the distributed termination detection protocol (recall section 4.5). The variables that define the termination condition in each process are implemented as part of the *PTobj* state and are accessed by the *PTobj* methods when invoked by the termination detection protocol event handlers.

The concurrency barriers, as described in section 6 are implemented in the Network Objects runtime system. The partial tracing algorithm intercepts the *transmission* and *remote method invocation* of a network object to implement the barriers.

Finally several implementation strategies were designed to support different requirements of the partial tracing algorithm. These include, suspect identification, implementation of mark-red and scan steps, implementation of mutator co-operation and, finally, as extensions to our prototype, the computation of the *cut-reference* graph. The implementation strategies we have defined require mainly the co-operation of the Network

[4] In our implementation we just have one *PTobj* per Network Objects process, since we have not implemented the advanced cyclic garbage collector

Objects local collector. We have modified it in order to obtain the necessary information for the partial tracing algorithm.

8.2.1 Partial Tracing

In this section we briefly describe each *DGCobj* handler and its interaction with the partial tracing state implemented by a *PTobj*, as they implement the partial tracing algorithm described in chapter 4. We represent the system architecture in figure 36. Next, we explain every element represented as well as every interaction, and which system feature they implement. As we have already said communication between processes is implemented by remote procedure call.

We first introduce the specification of *PTobj*:

```
PTobj =
 State
   Id = Initiator: processId
   Participants: set of processId
   parent: processId
   reply-set: set of wireRep
   local-steps: number
   grey-set: set of wireRep
   stack: stack of memory address
 Methods
   addStackMRrequest(obj: object): boolean
   init-scan(participants: set of processId): boolean
   addStackSrequest(obj: object): boolean
   insert-greySet(wrep: wireRep): boolean
   remove-greySet(wrep: wireRep): boolean
   insert-replySet(wrep: wireRep): boolean
   incLocal-steps(): boolean
   decLocal-steps(): boolean
```

Mark-red Phase

The `mark-red thread` implements mark-red local steps. It transmits the red colour locally from concrete objects to surrogates. In our implementation, local objects are

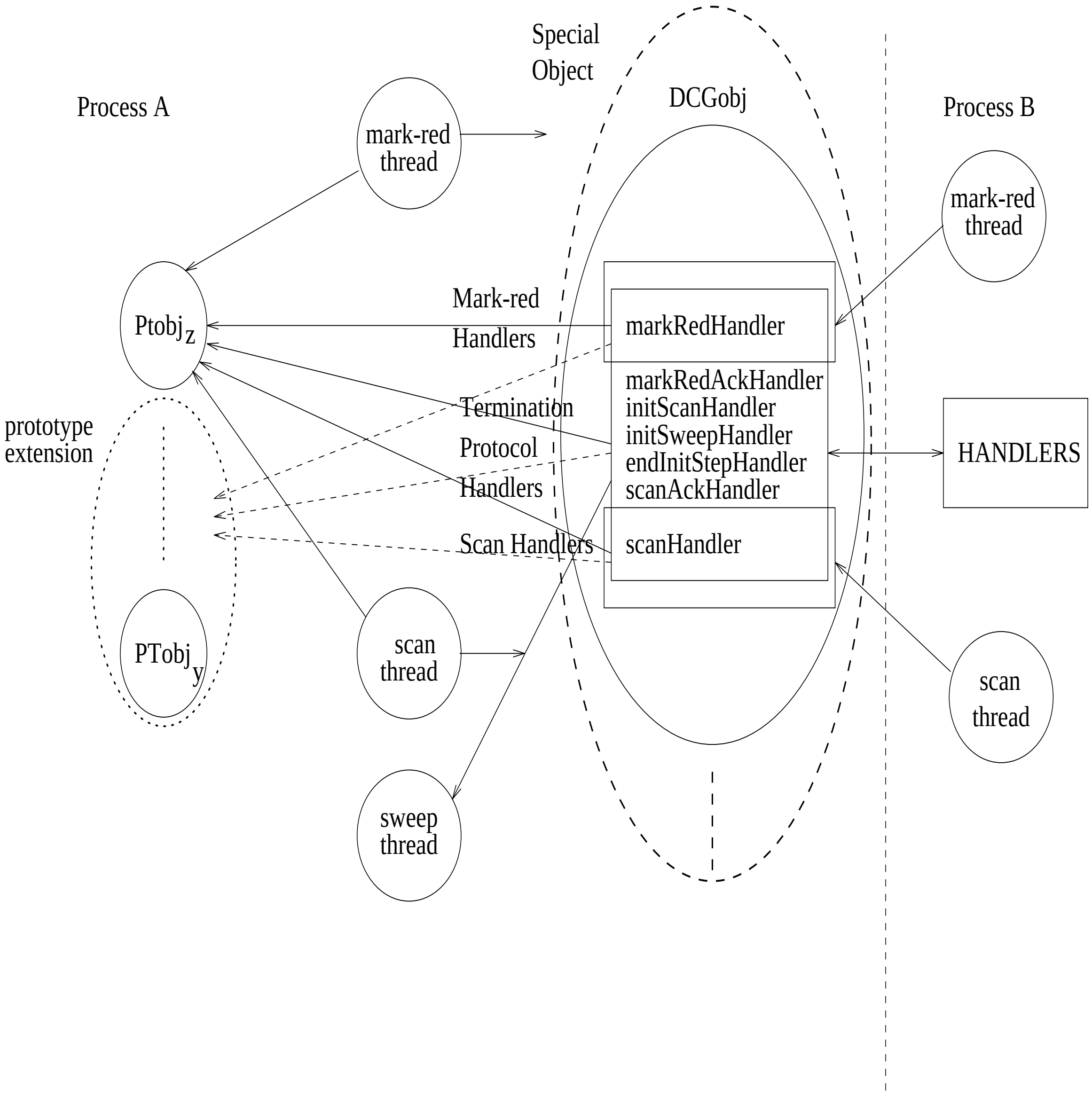


Figure 36: System Architecture

recursively traced. In contrast, our solution described on section 6 would simply follow the *cut-references* graph. Because this phase simply constructs a conservative estimate of those objects that might be garbage, and therefore need not be accurate, red marks can be disseminated without synchronisation with the mutator. The simplified algorithm of `mark-red thread` is described next.

```
BEGIN mark-red thread
   forever
     while stack not empty do
       pop(obj)
       colour(obj, red)
       push(descendants(obj))
     for all surrogate s in objtbl do
       if red(s) then
         PTobj.insert-greySet(WireRep(s))
         if DGCobj(ProcessID(s)).markRedHandler(s) then
           PTobj.remove-greySet(WireRep(s))
     PTobj.decLocal-steps()
     if PTobj.local-steps == 0 and PTobj.greySet empty then
       if thisProcess == PTobj.Initiator then
         for all P in PTobj.Participants do
           DGCobj(P).initScanHandler(PTobj.Participants)
       else
           DGCobj(PTobj.Responsible).
             markRedAckHandler(PTobj.Participants, PTobj.reply-set)
END
```

The recursive trace is implemented with a stack. The object seed of each local step is pushed onto the stack. The mark-red trace repeatedly pops objects from the stack until it is empty. Objects are marked red and any unmarked descendent is pushed in the stack. When the stack is empty, a remote step to the concrete object is executed for each red surrogate. For efficiency reasons, we chose to batch remote steps to the same target process.

Notice that `insert-greySet`, and `decLocal-steps` contribute to implementing the *termination detection protocol* (recall section 4.5). Termination detection protocol handlers, `initScanHandler` (described in the context of scan phase) and `markRedAckHandler`, implement the mark-red report-phase and acknowledging of a mark-red request respectively. Notice that, if `markRedHandler` replies `true`, the surrogate is removed from the `PTobj.greySet`. In this way, we reduce the number of explicit acknowledgements.

```
BEGIN markRedAckHandler(participants: ProcessID,
                        reply-set: set of wireRep)
  PTobj.Participants = PTobj.Participants ∪ participants
  for all wrep in reply-set
    PTobj.remove-greySet(wrep)
  if PTobj.local-steps == 0 and empty PTobj.greySet then
     if thisProcess == PTobj.Initiator then
       for all P in PTobj.Participants do
         DGCobj(P).initScanHandler(PTobj.Participants)
     else
         DGCobj(PTobj.Responsible).
            markRedAckHandler(PTobj.Participants, PTobj.reply-set)
END
```

A remote step is implemented as a remote procedure call of the `markRedHandler` by the mark-red thread. The target object is pushed into the stack. Consequently, `local-steps` is incremented. Additionally, the sending process is inserted into a structure called the *red-set* (akin to *client-set* (recall section 4.3)).

```
BEGIN markRedHandler(obj: object)
  if PTobj.Responsible ≠ NIL then
    reply = true
  else
    reply = false
    PTobj.insert-replySet(WireRep(obj))
    responsible = sending process
  PTobj.incLocal-steps()
  obj.red-set = obj.red-set ∪ sending process
  PTobj.addStackRequest(obj)
  return reply
END
```

Mark-red Restriction

As we have described in section 6 a mark-red remote step should not be performed through a surrogate corresponding to a reference being transmitted. As we have described in section 8.1, the Network Object system keeps track of incomplete reference transmissions on behalf of the acyclic collector, until the object's owner has received the corresponding *dirty-call*. In the same way, the mark-red thread discards remote-steps for surrogates in such conditions.

Scan Phase

A scan step may be a initial step, a remote step or a local step.

`initScanHandler` initiates the scan phase in every participant, that is, it initiates the initial step by inserting into the mark stack (akin to mark-red phase), the *local-scan-root-set* (in section 8.3 we describe the computation of the *local-scan-root-set*).

```
BEGIN initScanHandler(participants)
  PTobj.Participants = participants
  PTobj.incLocal-steps()
  PTobj.addStacksRequest(local-scan-root-set)
END
```

The initial step is a local tracing. It must be accurate, that is, it must paint green all surrogates reachable from the *local-scan-root-set* defined in section 5.1. Recall that this version does include the local roots and performs a recursive tracing from the defined *local-scan-root-set*. In contrast, the initial step defined in section 6 only follows the *cut-reference* graph.

We perform an independent local tracing from the *local-scan-root-set*. We have adopted Mostly Parallel garbage collection (Boehm et al. 1991) — an *incremental update* algorithm — because it does not require compiler modifications to implement the write-barrier and so can be used to support different languages, such as Modula-3.

The basic idea of the Mostly Parallel Garbage Collection algorithm is the following. The traditional tracing operation of stop-and-collect collectors is performed in parallel with the mutator. The mutator and collector are synchronised by the use of a write barrier that catches all the mutator writes while the collector is running. The write-barrier is implemented using a set of *virtual memory dirty bits*, which are automatically set whenever the corresponding pages of virtual memory are written to. The virtual memory bits are updated to reflect mutator writes. After tracing is complete, the mutator is halted and tracing is restarted from all marked objects that lie on dirty pages. At this point, all objects reachable from the *local-scan-root-set* are marked green, which is the safety condition for our system.

This option introduces, however, synchronisation requirements between the local collector and initial step as they both make use of the operating system virtual memory system. Thus, we believe that implementing the initial step in the system local collector could be a better compromise.

`mark-red thread` and `scan thread` implement recursive tracing using a stack from seed objects to surrogates, disseminating the colour. New object seeds increment `PTobj.local-steps`. When the stack is empty the same variable is decremented. A remote step is implemented as a remote procedure call of the `scanHandler` by the scan thread. The target object is then pushed into the stack. Consequently, the surrogate at the sending process is inserted into `PTobj.greySet`.

A local step is also implemented by the `scan thread`. If the initial step has not already finished, the target concrete object is a new root object for initial step and the remote step is immediately acknowledged — `scanAckHandler`. If initial step has already finished, the concrete object is a new object seed, is pushed into the stack and `PTobj.local-steps` is incremented. As we have already said, the scan phase must be accurate. In this way, mutator actions should also perform synchronisation actions. However, due to the asynchronous nature of local steps (also generated by external mutator messages — recall section 6) it would be inefficient for the Mostly Parallel garbage collection algorithm to implement such synchronisation. This is because, for every local step implementation, the memory would have to be protected and mutator actions trapped. The simplest method of propagating marks from concrete objects to surrogates is to 'stop the world' in the process and perform a standard recursive trace from the concrete object. We expect that this would not cause excessive delay because it is likely that objects reachable from a live concrete object are already known to be live. Recall that local steps defined in section 6 only follow the *cut-references* graph.

When a process terminates its initial-step — `PTobj.local-steps = 0` and `PTobj.greySet` is empty — it reports to the initiator — `PTobj.Initiator` — through `endInitStepHandler`. This state is detected by `scan thread` when `local-steps = 0` and `greySet` is empty (akin to `mark-red thread`) or by `scanAckHandler` (akin to `markRedAckHandler`).

```
BEGIN endInitStepHandler(participant: processID)
  PTobj.doneParticipants = PTobj.doneParticipants ∪ participant
  if PTobj.doneParticipants == PTobj.Participants then
    for all P in PTobj.Participants do
      DGCobj(P).initSweepHandler()
END
```

The initiator detects the end of scan phase when it `PTobj.doneParticipants` is the same as `PTobj.Participants`. The scan phase report phase is implemented by the invocation of the `initSweepHandler` in every participant.

We here skipped the description of `scan thread`, `scanHandler` and `scanAckHandler`, as they are similar to the corresponding mark-red phase ones. `initSweepHandler` instructs every participant to proceed to the sweep phase.

Sweep Phase

The `sweep thread` is responsible for communicating to the local collector which *object table* entries should be discarded from the global root set. In our implementation we have coloured red and green concrete objects rather than *object table* entries. Our solution consists of instructing the local collector to discard *object table* entries that point to red concrete objects. However, the entries themselves are only removed when the *client-set* of such concrete objects is empty. In this way, we do not interfere with the *reference listing* scheme (recall section 3.4.4): discarding *object table* entries causes the cycle to be deleted the next time the containing processes do a local trace. Discarded *object table* entries are then removed through regular *clean-calls*. This solution can be implemented in the Network Objects system because a concrete object's *client-set* is implemented as an auxiliary structure and not as a state variable of the object.

8.2.2 Suspect Identification

We describe our implementation strategy for identifying suspect entry-items. We focus here on the *locally reachable* heuristic.

In the partitioned model (recall section 3.1) the roots for a local collection include the local root set — the local roots — and the global root set — the entry-table. In the Network Object system the entry-table is implemented by a monolithic *object table* (section 8.1.1), which is then considered part of the root set by the local collector. An unmodified local collector would always mark all the exported objects and surrogates. As a result, exported objects and surrogates would be always locally reachable and therefore never collected. To solve this problem we have implemented a modified version of the Modula-3 local collector (section 8.1.2) and a modified object table. This new version implements a two-phase local collection that defines two kind of objects:

soft objects are not locally reachable other than from the global root set — the *object table*. They are suspect objects, and candidates for a partial tracing. A *soft* mark identifies a seed for a partial tracing, while a *red* mark identifies an object as a member of a suspect subgraph (recall chapter 4).

hard objects are locally reachable. Consequently, they are not candidates for a partial tracing.

The two phase local collector proceeds as follows:

- A first tracing is performed from the local roots. This first root set does not include the *object table*. The objects reached by this first tracing are marked *hard*. The object table is marked *soft*.
- A second tracing is performed starting from the object table. This second tracing completes the first. Any object reached by this second tracing is marked *soft* if it is not already marked *hard*.

Special actions are needed to prevent the first tracing from including the object table in the first root set. The modified local collector treats the object table as a special root. To allow this, we modified the object table as shown in figure 37: the object table state includes a fixed size array *hash table* of pointers to network objects. This allows the collector to identify unambiguously which pages the object table uses[5].

[5] Here, for simplicity, we assume the object table fits in one page. In our implementation, this restriction does not apply. We just ensure that object table pages do not contain any other objects.

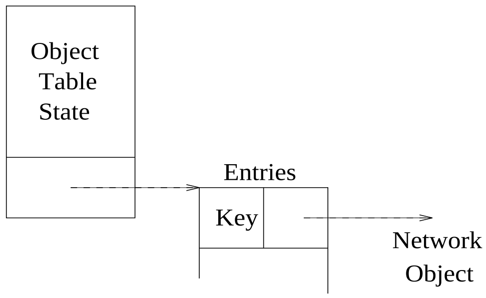


Figure 37: Object: Object Table

Now we require the allocator to allocate the object table in a separate page. No other objects share this page with *object table*, and the collector proceeds as follows:

1 at the begin of a local collection promote the object table page. In this way we prevent tracing from the objects in the *object table* in the first phase.

2 at the end of the first phase, start a second trace from the *object table*. The objects reached by this second trace are marked *soft* if not reached by the first trace.

The two phases of the local collection execute concurrently with the mutator. Certain actions have to be taken in order to maintain invariant 8.2 on page 177 — *the mutator is not allowed to see black objects* — when the *object table* is accessed by the mutator. The unmodified local collector (section 8.1.2) would protect the *object table* page when promoted, but this would result in a page scan by the fault-handler (see figure 37) whenever the *object table* fault is trapped. This would result in the same problem mentioned above: every exported object, even those only reachable from the *object table*, would be marked *hard*. We resolve this problem as follows. As above, the *object table* page is promoted at the begining of the local collection, but it is not protected. Instead, each entry in *Entries* is protected with a *black-only* read barrier implemented in the *object table* methods. Hence, in terms of the tricolour abstraction for incremental collection, *object table* entries are *grey*. We illustrate this situation in figure 38 and describe it in figure 39. When an *object table* search method is invoked, the entry found in *Entries* is scanned and the object pointed by it is copied to the *current* space (coloured *grey* in terms of the tricolour abstraction) — `objTbl_method()`. In its turn, the entry is coloured *black*, and hence allowed to be read by the mutator. The network object pointed to this entry is marked *hard* during the first phase.

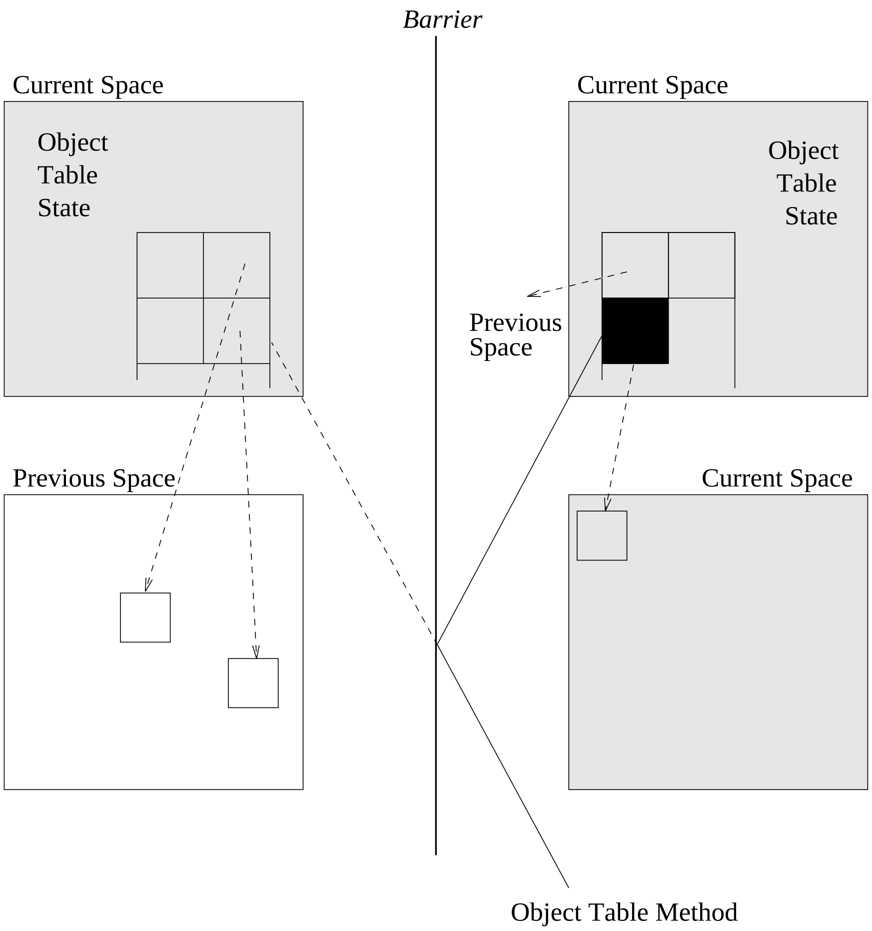


Figure 38: Object Table Barrier

```
State == zero, one, two, three
Space == current, previous, free
State = zero
current_queue = empty

gc() =
    promote_objtbl()
    State == zero  => for page where  page.space = current do
                         page.space = previous
                      for R in Roots do
                          promote(page(R))
                      State = One
    State == One   => page = pop(current_queue)
                      scanPage(page)
                      if current_queue == empty then
                          State = two
    State == Two   => scanPage(objTblpage)
                      if current_queue == empty then
                          State = three
    State == Three => for page page.space = previous do
                         page.space = free
                      State = zero
     if current_queue ≠ empty then
      for page in current_queue do
        protect(page)

promote_objTbl() =
    objTblpage.space = current

objTbl_method() =
      ⋮
    entry = findEntry();
    newpage = scan(entry)
    push(newpage, current_queue)
      ⋮
```

Figure 39: Modula-3 local collector algorithm for suspect identification

At the end of first phase, *object table* pages are scanned. *Grey* entries are coloured black in terms of the tricolour abstraction, while the corresponding network objects are marked *soft* in terms of suspect identification. The omitted details are as described in figure 39.

8.2.3 Remote Barrier

As we have described in section 6 a **Remote Barrier** is executed whenever a mutator at process A invokes or transmits a remote reference to a remote object at process B through a red exit-item. In the Network Objects system model this means whenever a mutator at process A invokes or transmits a remote reference to an object yB at process B through a red surrogate yB, a scan request is sent to yB entry in B's *object table*.

We combine the implementation of the **Remote Barrier** described in section 6 with the implementation of remote methods invocation and the implementation of the distributed garbage collector operations described in section 8.1.4 and 8.1.5 respectively, thus not incurring any extra remote messages. We explain how in the remaining of this section.

Method Invocation

A client does not directly invoke the methods of a remote object. Instead, it invokes the corresponding methods of a surrogate object. Our implementation intercepts the surrogate methods invocation and sends a scan request piggy-backed on the remote method invocation whenever this surrogate is **red**.

The Network Objects system generates the stubs for remote method invocation automatically. We modified those in order to provide piggy-backed scan requests. Recall the simple sketch of a typical surrogate method in section 8.1.4. The modified sketch to account for piggy-backed scan requests is:

```
BEGIN SurrogateMethod
  if red(self) then
    PTobj.insert-greySet(WireRep(self))
    <marshal to c (barrier = true)>
    <marshal to c process(self)>
    <marshal to c PTobj.Id>
```

```
    else <marshal to c (barrier = false)>
    <marshal to c the number of the method>
    <marshal to c the method arguments>
    result = AwaitResult(c);
    <unmarshal from c the method results>
  END SurrogateMethod
```

`red(self)` tests the colour of the surrogate and `process(self)` identifies the sending process. The sending process is the responsible for such request.

`insert-greySet(self)` implements the communication with the *termination detection protocol*, that is, it communicates a new element for `greySet` (recall section 4.5). Finally, the boolean value `barrier` indicates a scan request for object `obj` at the server side.

Recall the sketch of a typical dispatcher described on section 8.1.4. At the server side the encoded information is interpreted as follows.

```
BEGIN Dispatcher
   <unmarshal from c barrier>
   if barrier == true then
     <unmarshal from c process>
     <unmarshal from c Id>
     <execute barrier with Id from process>
   <unmarshal from c the method arguments>
   <call the appropriate method of obj>
   <marshal to c the method result>
 END Dispatcher
```

When unmarshaling the method arguments, if `barrier = true` a barrier should be executed. This is equivalent to a scan local step described in section 8.2.1. *Id* identifies the partial tracing and *process* the responsible for such request.

Transmission of a Reference

As we have already described, network objects are transmitted from one process to another during method invocation as arguments or results of remote method invocations. Our implementation intercepts such actions on the `MarshalObj` and `UnmarshalObj` procedures described on section 8.1.5. When a process A, with a `red` surrogate for an

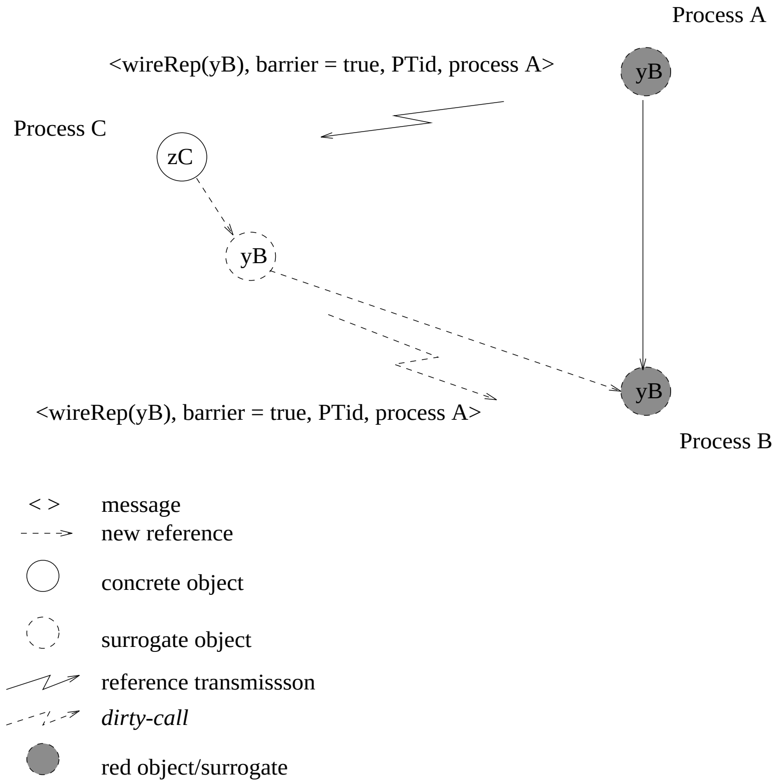


Figure 40: Implementation of **Remote Barrier** for transmission of a reference.

object `obj` with wireRep `wireRep` at process B, marshals it as an argument or result of a remote method invocation to process C, we piggy-back a scan request to `obj` at process B (the owner) in the transmission message and acyclic garbage collector operations: the *dirty-call* to `obj`. We illustrate this procedure in figure 40 and present a simple sketch next:

```
BEGIN MarshalObj
  if concrete(obj) then
    <insert obj in objTbl>
    msg = <wireRep, barrier = false>
  else (* surrogate *)
    if red(obj) then
      PTobj.insert-greySet(WireRep(obj))
      msg = <wireRep, barrier = true, PTobj.Id, process(obj)>
    else msg = <wireRep, barrier = false>
  <send msg to receiving process>
END
```

Recall section 6, **Remote Barriers** are only executed through red surrogates. Consequently, we concentrate on reference transmissions through surrogates. `red(obj)` tests the colour of the surrogate. When marshaling a reference through a red surrogate, our implementation instructs the receiver process to send a scan request piggy-backed on the *dirty-call* to the object owner. The sending process is the responsible of such scan request. `insert-greySet(obj)` implements the communication with the *termination detection protocol*, that is, it communicates a new element for `greySet` (recall section 4.5). As above, the boolean value `barrier` indicates a scan request for object `obj` at the owner process.

When unmarshaling the network object reference, the receiver process instructs the *dirty-call* to the owner process to execute a barrier, if `barrier = true`. However, it may happen that the reference was transfered to the owner process. In this case, `objtbl.find(wireRep, obj)` and `concrete(obj)` return true, and the barrier should be executed immediately on `obj`. As above, this is equivalent to a scan local step described in section 8.2.1. `Id` identifies the partial tracing and `process` the responsible for such request.

```
BEGIN UnmarshalObj
  <receive msg>
  with msg
    if barrier == true then
      if objtbl.find(wireRep, obj) then
        if concrete(obj) then
          <execute barrier with Id from process>
      else
        <create surrogate with wireRep>
        specialObj(ProcessID(wireRep)).dirty-call
                   (wireRep, barrier = true, Id, process)
    else
      if not objtbl.find(msg.wireRep, obj) then
        <create surrogate with msg.wireRep>
        specialObj(ProcessID(wireRep)).dirty-call
                   (wireRep, barrier = false)
END
```

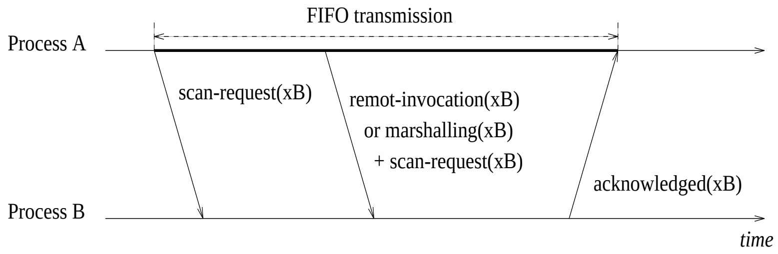


Figure 41: Time-line showing the need for repeated piggy-backing of scan requests on barrier execution.

If `barrier` = true, a *dirty-call* should execute a barrier. As above, this is equivalent to a scan local step described in section 8.2.1. *Id* identifies the partial tracing and `process` the responsible for such request.

```
BEGIN DirtyCall
  if objtbl.find(wireRep, obj) then
    <insert sending process in obj  client-set>
    if barrier == true then
      <execute barrier with Id from process>
END
```

Remote-step Invariant

In section 6 we introduced a restriction of point to point communication in order to maintain the invariant 6.4. We assumed that messages are to arrive in the same order as they were generated in point-to-point communications, that is, point to point communication channels are FIFOs. In this way, a mutator message sent through a *green* surrogate would always arrive after the scan request generated when that surrogate was painted green. As we no longer assume messages are delivered in the order they were sent in point-to-point communications, we implement it by repeatedly piggy-backing scan requests onto mutator messages until one such request is acknowledged. This is illustrated on figure 41 for two processes, A and B, and a scan request to object xB. Any remote invocation of xB, or transmission of xB to other processes on the bold time-line, requires a piggy-backed scan request.

In the absence of failures, one request will be eventually acknowledged, resulting in the effective greening of the surrogate.

8.3 Prototype Extensions

8.3.1 *Cut-references* Graph

In this section we discuss the implementation of *cut-references* graph in the Network Objects system, described in section 5.3.

Several authors have proposed techniques for computing the *cut-references* graph in the context of back-tracing algorithms (Rodriguez-Riviera 1995, Maheshwari and Liskov 1997a). The solution proposed in (Maheshwari and Liskov 1997a) has some advantages over the solution proposed in (Rodriguez-Riviera 1995). Locally reachable surrogates of suspect concrete objects (akin to variable *exits*) may be computed bottom up during a depth-first local collection using Tarjan's algorithm (Tarjan 1972). This method requires objects to be traced just once, in contrast with the solution described in (Rodriguez-Riviera 1995) that may require objects to be traced many times.

However, as mentioned in the same work, breadth-first copying collectors need to perform a separate trace from suspect objects in order to compute the *cut-references* graph. We argue that this would not be expensive because suspect objects are expected to be few and such traces would be able to proceed concurrently with the mutator at low cost. Moreover, the frequency with which the *cut-references* graph is computed may be controled: if suspect items are not dirty, compute the cut-references only for new suspects.

Recall section 5.3. We introduced **identify-suspects** that computed suspect objects. As we have already shown in section 8.2.2, such a method may be implemented by the Modula-3 local collector, albeit slightly modified. **compute-graph** may now implement Tarjan's algorithm concurrently with the mutator. The **Dirty barrier** we have introduced in section 6 allows one partial tracing to use the *cut-references* graph safely, as we have proved in section 7. As we will explain next, the **Dirty Barrier** is only applied when the mutator invokes a remote method and it is inexpensive.

8.3.2 Dirty Barrier

The implementation of the **Dirty Barrier** is simple and inexpensive. In the Network Objects system we would need a new field for every concrete object that would be cleared every time **identify-suspects** is performed and set every time the concrete object is invoked: the *dirty* field. Such invocations may only be made by an external mutator, as we have shown in section 6. The typical dispatcher described on section 8.1.4 would be modified as follows:

```
BEGIN Dispatcher
   <dirty obj>
   <unmarshal from c barrier>
   if barrier = true then
     <unmarshal from c process>
     <unmarshal from c Id>
     <execute barrier with Id from process>
   <unmarshal from c the method arguments>
   <call the appropriate method of obj>
   <marshal to c the method result>
 END Dispatcher
```

`<dirty obj>` would set the object's dirty *field* until the next local collection.

8.4 Summary

The main goal of this chapter is to identify the potential problems and propose solutions when mapping the system we have described in chapters 4 and 5 onto the Network Objects system.

We presented an overview of the Network Objects system, namely its implementation of the model described in chapter 3, and implementation details that were important to the implementation of our system. These included the Modula-3 local collection algorithm, methods for marshaling, unmarshaling and remote invocation, and finally the acyclic garbage collection operations.

We described our prototype implementation. The prototype description included the system architecture and interaction between the different components. We looked, in more detail, at the implementation strategies of several aspects of our system, namely, how to identify suspect objects and how to implement the mutator co-operation over the remote invocation system.

We also discussed the viability of implementing the extensions to our system described in chapter 5, namely the implementation of the *cut-references* graph and dirty barrier.

Chapter 9

Conclusions and Future Work

In this chapter we summarise our primary goals for garbage collection in distributed systems, for which we proposed an innovative solution and offer a qualitative analysis of our algorithm. We conclude this chapter with some perspectives for future research.

9.1 Discussion

We have addressed the following fundamental goals faced by cyclic distributed garbage collection: correctness, efficiency, scalability, completeness and fault-tolerance. A common thread to our solution, described in chapters 4 and 5, is that we approximate the property of locality. That is, we rejected global tracing as a means for cyclic garbage collection in a large, distributed address space, because it is neither scalable nor fault-tolerant. Instead, we have adopted a solution that tries to confine the collection of a distributed garbage cycle to those processes that contain it.

Our approach combines tracing within one partition and reference listing (across partition boundaries) with *partial tracing* (within a group of partitions) in order to collect distributed garbage cycles. We use heuristics to form groups of processes dynamically (the mark-red phase) that co-operate to perform partial traces of subgraphs suspected of being garbage.

We support multiple, independently-initiated distributed garbage collections and allow the collection of garbage cycles that span groups.

Our scheme operates in three phases:

Mark-red phase We identify as red the inter-process transitive closure of an object heuristically suspected of belonging to a garbage cycle, starting from a suspect entry-item. We also form a group of processes that will collaborate in the subsequent phases.

Scan phase We try to isolate self-contained red subgraphs, *i.e.*, garbage cycles. We perform a group collection that aims at painting green any red object reachable from outside the red subgraph, *i.e.*, red objects reachable from a non-red global root. A group collection involves a local trace in each partition. However, to trace a group: (*i*) red entry-items are not considered as members of the local roots, and (*ii*) tracing continues across boundaries internal to the group, when red exit-items are repainted green.

Sweep phase We make remaining red objects available for collection by the next local collection, because they must be garbage.

In this section we offer a qualitative analysis of our algorithm. We discuss several aspects of our system in order to analyse how it meets the goals stated in section 4.2.

9.1.1 Scalability and Completeness

The first feature that makes our system scalable is its lack of need for global synchronisation. A partial tracing is potentially scalable in that it uses asynchronous communication and has no protocols that demand the involvement of all processes in the system. Each partial tracing only needs the co-operation of those processes that either participate in the partial tracing itself or participate in some responsible partial tracing. This co-operation is engaged through the initiators of each partial tracing. Synchronisation actions imply communication between each partial tracing participant and its initiator, and between an initiator and its responsibles' initiators.

We showed that our algorithm is complete (section 7.3). It allows different actions to be taken when two concurrent partial tracings meet:

Overlapping of suspect subgraphs. Different partial tracings are active in the same subgraph. This allows long-running complete collections, although at a cost of wasted work and space overhead.

Synchronised merging of suspect subgraphs. Different co-operative partial tracings are active and synchronised in the same subgraph. They allow complete collections at space and synchronisation low costs. Although, depending on the graph topology, they may compromise promptness.

Merging of suspect subgraphs. Different co-operative and independent partial tracings are active in the same subgraph. They allow more expedient collections although compromising completeness.

This policy decision may be determined by the collector itself or by the user program, globally or on a per-process or even per-object basis. Heuristics based on geography, process identity, distance from the suspect originating the collection, minimum distance from any object known to be live, or time constraints may be used to restrict the extent of mark-red or the decision whether to merge with, overlap or retreat from other distributed collections. In the absence of knowledge of the problem being computed, it is unclear what action should be taken when two groups meet. A merger may not always be desirable. Instead it may be preferable to run multiple overlapping partial tracings. For example the best compromise may be to combine simultaneously occasional long-running but complete collections over very large groups with more frequent faster completing collections over small object graphs. Our algorithm offers the implementer the choice between completeness and promptness at the level of partial tracings, processes and individual objects. Partial tracings can decide whether or not to merge, processes can decide whether to allow partial tracings to merge, to overlap or to retreat from one another, and objects can decide on merger or retreat.

The efficiency of this algorithm is greatly affected by this policy decision. We analyse its efficiency when discussing promptness next.

9.1.2 Efficiency

We have identified three efficiency concerns: message complexity, space complexity, promptness/progress and mutator overhead.

Message Complexity

The message complexity of our system depends entirely on number of interprocess edges and the topology of the graph. We can measure this by considering the number of times an inter-process reference might be traversed, where each traversing is a message.

Call the number of inter-process edges in the subgraph visited by mark-red e, and the number of participants in this group n. Note that $e \leq$ the number of edges in the transitive referential closure of the suspect objects, because the mark-red phase does not need to visit the complete transitive referential closure of suspect entry-items.

- The mark-red phase for each group issues e mark-request remote procedure calls, by definition.
- The number of mark-red acknowledgement calls depends on whether the request is sent to a quiet or a disquiet participant, and this in turn depends on the topology (degree of sharing) of the subgraph. An acknowledgement from a disquiet participant can be piggy-backed onto the remote procedure call acknowledgement; that from a quiet participant requires a separate call. Thus, between $n-1$ (one acknowledgement for each participant-creating request) and e (one per edge) calls are required.
- Each acknowledgement message has a length $\leq n$, the maximum number of processes to which the request message can have been forwarded.

Thus the number of remote procedure calls $\mathcal{C}_{MR}$ caused by mark-red is:

$$e + n - 1 \leq \mathcal{C}_{MR} \leq 2e$$

- Scan phase initiation requires $n-1$ messages: one message from the initiator to each participant. Additionally, scan phase initiation for synchronised co-operative partial tracings (recall section 5.5) requires d_z calls for each PT_z where $d_z = |Dependent^*(PT_z)|$.
- The number of scan requests sent depends on the accuracy with which suspects are identified. In the best case, no requests are sent but each participant must report termination to the initiator; in the worst case, the number of remote procedure

calls is the same as that for mark-red[1]. Let p be the probability that our suspect identification heuristic is accurate.

- Scan termination for co-operative partial tracings requires d_z calls.

Thus the expected number of remote procedure calls $\mathcal{C}_{SC}$ caused by the scan phase is:

$$(1-p)e + 2(n-1) + 2d_z \leq \mathcal{C}_{SC} \leq 2(1-p)e + (1+p)(n-1) + 2d_z$$

- The sweep phase requires $n-1$ messages.

The total number of remote procedure calls $\mathcal{C}_{pt}$ required is:

$$(2-p)e + 4(n-1) + 2d_z \leq \mathcal{C}_{pt} \leq 2(2-p)e + (2+p)(n-1) + 2d_z$$

The cost of our algorithm is determined by the parameters n, e, d_z and p. p depends on our choice of suspect; n, e and d_z are partly determined by the topology of the subgraph and the dynamics of distributed collections but can also be controlled by policy decisions on the extent of mark-red's coverage of the graph. Because little is known of the demographics of distributed objects, flexibility is a key goal of our collector. Our collector can be seen as a framework within which policy decisions can be implemented. Policy guides the choice of suspects, the choice of processes forming each partial tracing and the merger of partial tracings.

The better the heuristic the greater the chance p that our algorithm traces only garbage subgraphs thereby:

1 decreasing the number of times a partial tracing is run,

2 limiting the mark-red trace to just garbage items,

3 reducing the number of messages for the scan phase to the best case, and

4 decreasing the chance of wasted and repeated work.

As we have already said in section 4.6 a more sophisticated heuristic — the *distance heuristic* — may improve the algorithm's efficiency.

[1]The intermediate case occurs when a subset of the red sub-graph is found to be live.

p and n can be controlled by bounding the amount of work done by mark-red. Recall that this phase needs only make a conservative estimate of the transitive referential closure of suspect objects — it need not visit the whole closure. This policy decision can be taken statically by prior negotiation or dynamically by mark-red.

Back-tracing Algorithm

Let us turn to the analysis of the algorithm presented in (Maheshwari and Liskov 1997a). This is an algorithm of the same class as ours, as they identify heuristically objects suspect of being garbage. Back-tracing, as opposed to forward tracing, follows the "refers-to" relation on the *inverse reference graph* (IRG) (recall section 3.5.6). It starts a back-tracing on the transitive closure, of this new relation, of a suspect object in order to find out if it is transitively reachable from a root.

Call the number of inter-process edges in the transitive closure of a suspect entry-item e, and the number of involved processes n. In its first phase, back-tracing involves two messages for each inter-process reference it traverses — one for the call when tracing back to the root, and another for its response when returning. Finally the report phase involves a message from the initiator to each participant. Thus, independently of the suspect choice, the total number of calls required is

$$\mathcal{C}_{bt} = e + (n - 1)$$

$\mathcal{C}_{pt}$ is greater than $\mathcal{C}_{bt}$. However they are both $O(n)$. Moreover, if multiple collections start on several objects of a suspect subgraph, say m, $\mathcal{C}_{pt}$ will be the same because the multiple partial tracings will co-operate. On the other hand, $\mathcal{C}_{bt}$ will turn to

$$\mathcal{C}_{bt} = me + m(n - 1)$$

As in the partial tracing algorithm, the better the heuristic the greater the chance that the back-tracing algorithm traces only garbage subgraphs thereby decreasing the number of times a back-tracing is run and decreasing the chance of wasted and repeated work. Moreover, this algorithm guarantees that if a back-tracing is started in a garbage structure, only garbage objects will be traced. On the other hand, our system may trace live objects if a garbage structure points to live data. As we said, our system relies on the heuristic to find suspect objects. The more accurate the heuristic the greater the chance that our algorithm traces only garbage subgraphs thereby limiting the mark-red trace to just garbage items.

In addition to an increasing message complexity, multiple collections active in the same cycle lead to a greater amount of repeated and wasted work. We expect a partial tracing to be more efficient than a back-tracing, and vice-versa, depending on the topology of the applications graph. We come back to this analysis when discussing promptness next.

Space Complexity

Compared to other algorithms based on reference-listing (recall section 3.4.4), our algorithm requires extra space for the red-lists and for the *cut-reference* graph. However, this space is not proportional to the application's graph.

Consider the space occupied by the *cut-reference* graph. After a local collection, the structure *exits* retains, for each suspect entry-item Ei_z, a list of the suspects' exit-items reachable from Ei_z. Call the number of suspect entry-items n_{ei}, and the number of suspect exit-items n_{ex}. The space occupied by the *cut-reference* graph is $O(n_{ei} \times n_{ex})$. This extra space is only required between the time an entry-item becomes suspect and the time the entry-item is collected or turns to non-suspect. Moreover, this extra space is the same required by the back-tracing algorithm, as it also builds a suspect *cut-references* graph.

Compared to the back-tracing algorithm, our algorithm requires extra space for the red-lists. During a partial tracing, a red entry-item Ei_z, member of a suspect sub-graph, retains a list of which processes have a red Ex_z. Call the number of red entry-items r_{ei}, and the number of red exit-items r_{ex}. The space occupied by the red-lists is $O(r_{ei} \times r_{ex})$. Recall, however, that Ei_z.*red-list* $\subseteq$ Ei_z.*entry-list*. Thus, the red-list may be implemented in the entry-lists just by setting a bit on the entry-list's elements.

Promptness/Progress

Our algorithm achieves promptness in two ways. First, it does not compromise the reclamation of local and acyclic distributed garbage: local collection and acyclic distributed collection are not hindered. Second, the promptness of the cyclic distributed collection is potentially improved by restricting the target object graph to suspect objects graphs — property of locality. Moreover, as we have already shown in section 4.6, the amount of work done by mark-red can be bounded, hence potentially improving the promptness of our system. We need, however, to make a stronger case about the likely behaviour of our algorithm: how much progress it makes, how great can be the ineffective and wasted work, which are the good cases and which are the bad cases.

Consider a distributed garbage cycle and suppose that PT_z is initiated at any object z, which is member of that cycle. For that cycle to not be collected there must be an external reference to it; that is, there must be an object y that is not involved in PT_z, and z is transitively reachable from y. We have showed that our system is complete: y must be garbage, otherwise z would not be garbage; a partial tracing will be initiated at y eventually. However we want to show that its efficiency depends greatly on the suspect choice. Recall that one of these three situations will happen eventually:

1 PT_y covers z, and there are no external references to PT_y's suspect subgraph.

2 PT_z is transitively dependent of PT_y, and there are no external references to PT_y's suspect subgraph.

3 PT_y is transitively responsible for PT_z and PT_z is transitively responsible for PT_y, and there are no external references to PT_z's suspect subgraph and PT_y's suspect subgraph.

As we showed, z is eventually collected independently of which situation occurs. Suppose however that PT_z initiates at z and terminates before PT_y has initiated at y, and that PT_z does not cover y. As there is an external reference to PT_z's suspect subgraph, PT_z will fail in collecting z. We show this situation in figure 42. We show in the figure two connected cycles. The lines represent an arbitrary number of objects. We explicitly represent objects x, z and y. Before one of the above situations can happen, any partial tracing initiated in the bold cycle in the figure will fail to collect z, hence not making any progress. Also, if two partial tracings, PT_x and PT_y, start and PT_y finishes before PT_x encounters y, PT_y will fail because there is an external reference to PT_y's suspect subgraph. These two situations lead to wasted and repeated work.

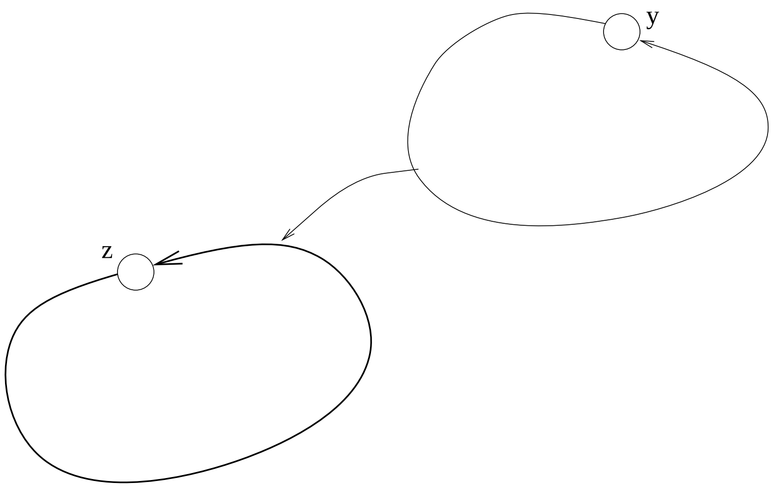


Figure 42: Connected garbage structures

This happens because, as we said in section 4.2.1, the mark-red phase may not trace the whole set of connected garbage objects.

Notice that these situations do not occur in the back-tracing algorithm. A back-trace that starts at a garbage object will always succeed in the absence of failures. Hence, we could expect a better promptness/progress in this situation.

We may bound the wasted and repeated work of the partial-tracing algorithm, based on the fact that if the bold cycle in figure 42 is garbage, the lighter cycle in the same figure must be garbage. Consequently, by our completeness argument, every object in the garbage structure is eventually suspect and may potentially initiate a partial tracing.

We now define two parameters that determine, as we show below, the success of our algorithm, that is, its promptness:

$\mathcal{T}$ is an estimated time interval during which all garbage objects within a garbage structure become suspect. Assuming that our system is implementing the *distance heuristic*, $\mathcal{T}$ would be the time needed for every object in the garbage structure to cross the distance threshold. Assuming that processes perform local garbage collections regularly, we expect that every garbage object will cross the threshold in a bounded time.

$\mathcal{T}_{pt}$ is an estimated time for a mark-trace to cover the whole transitive closure of a suspect object.

Now consider that object z becomes a suspect object. It then starts a PT_z. It may reach some object that, although being garbage, may have not become suspect. The solution would be to follow the following rule:

rule a start PT_z at the time given by adding $\mathcal{T}$ to the time z became a suspect.

At this time, all objects in the garbage structure are likely to be suspect.

Now suppose that PT_z's mark-red phase has finished and PT_z's initiator is going to start PT_z's scan phase. As we have already said, it fails because there is an external reference to PT_z's suspect sub-graph. We show now how our algorithm can reduce the probability of such a problem to happen using the values of $\mathcal{T}$ and $\mathcal{T}_{pt}$. The rule is:

rule b delay the beginning of PT_z's scan phase by $\mathcal{T}_{pt}$.

Recall that, by rule a, when PT_z started, the objects in the garbage structure are likely to be suspect, hence are likely to have started a partial tracing. In the example, PT_y. $\mathcal{T}_{pt}$ is an estimated time for a mark-trace to reach the transitive closure of a suspect object. If PT_z delays the beginning of the scan phase by $\mathcal{T}_{pt}$, it is likely that by the time PT_z initiates the scan phase PT_y has encountered PT_z and defined a dependent/responsible relation. We may generalize this assumption and say that the more accurate the estimations of $\mathcal{T}$ and $\mathcal{T}_{pt}$, the greater the probability of different garbage structures to be covered by one, or more, partial tracings.

These values may be estimated after measurements of real applications and may be tuned each garbage collection cycle. That is, these values may be adapted as the system evolves.

How can these values be estimated? We expect our system fails to collect a suspect cycle because garbage, not involved in the partial tracing, points to garbage, and not because the cycle garbage is live. In our example, PT_z always fail until there is a partial tracing, for example PT_y, that also covers the PT_z's suspect sub-graph, or meets PT_z or deletes the external references to PT_z's suspect sub-graph. We may measure the time between a PT_z's failure and the PT_z's success. As we expect that this time is the time necessary for the whole garbage structure to be covered by one, or more, partial tracings, we accept it as an estimation of $\mathcal{T} + \mathcal{T}_{pt}$. Expecting that, in a real application, one garbage structure may be formed often, we could use the estimated time to tune future collection cycles.

Notice that PT_z's failure or success may be measured by which action — delete or no delete — is applied to the initiator object, the object z, every time PT_z is finished. We believe this may improve the success of a partial tracing and then the promptness of the system.

On the other hand, we are also interested in avoiding unnecessary multiple collections, for example, collections initiated in the bold cycle in figure 42, that increase both d (see above) and the number of processes where the collections have to meet. To this end, the system could only propagate distances over a certain threshold through mark-red requests. In this way, when the objects' distances cross the distance threshold, the corresponding objects would be already involved in a partial tracing. In this way we could avoid synchronising the start of scan phase as we would reduce the risk of multiple distributed collections in the same garbage cycle. It is important to say however that if the system neither propagates distances through the mark-red phase nor synchronises the begin of scan phase, multiple collections on the same cycle may abort indefinitely (recall section 4.2.1).

We consider now the behaviour of our algorithm when handling large and complex structures, for example, chains of multiple garbage structures such as the ones represented in figure 42.

Recall that if we delay the beginning of a partial tracing by $\mathcal{T}$, there is a large chance that all objects in the garbage structure turn to suspect. This situation may lead to one in which multiple partial tracings would start and co-operate in the collection of large structures.

As we have already said (recall section 4.6), heuristics can be used for restricting the extent of the mark-red phase. Depending on the shape of the garbage structure, this may compromise the algorithm's progress as the mark-red phase may not include the whole connected garbage structure, hence failing to collect the garbage structure. Only measurements of real applications can say how much this type of heuristic may improve the promptness of the algorithm.

Consider now the back-tracing algorithm. It is difficult to understand how the algorithm behaves in such garbage structures. If multiple back-tracings start, they may lead to repeated and wasted work. On the other hand, multiple back-tracings may

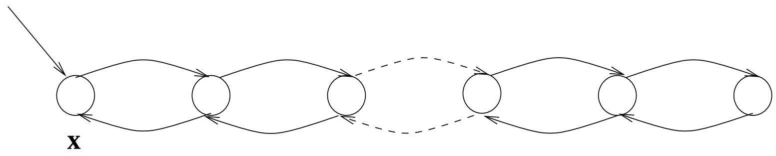


Figure 43: Double linked list

independently collect any part of the garbage structure. In the absence of measurements, it is difficult to know which action would be taken.

Moreover, a back-tracing has to find a root or a garbage object in order to determine the liveness of a suspect object. The process/object that initiated a back-tracing must receive back an answer of type "live" or "not live". Back-tracing may always stop at any time. However, if the back-tracing algorithm gives up at any path, it returns "live" to the initiator. In this way it cannot decide about the liveness of suspect objects. Thus, we say that it lacks of the property of flexibility, hence make difficult the use of heuristics to improve promptness of garbage collection of large and complex structures.

Examples Consider now some examples of typical structures such as double linked lists and searching trees, with back references, of an arbitrary size. We describe how we expect our algorithm to behave with those structures.

We show in figure 43 a double linked list of N elements, in which each element is located in a different process, and E inter-process references. After the deletion of the list entry reference, every object will become a suspect object as a consequence of multiple local collections in each process. The behaviour of the partial tracing algorithm depends on which garbage object starts a partial tracing:

- Suppose that object x initiates a partial tracing PT_x. It propagates the mark-red to the other objects before they start a partial tracing. This may be achieved for example, as we said, by propagating the distances through the mark-red phase. The whole structure would be covered by PT_x and entirely collected. The message cost corresponds to the best case of the message complexity formula.

- Suppose that object x initiates a back-tracing algorithm. It propagates the back-tracing messages to the other objects before they start a back-tracing. The whole structure would be covered by PT_x and entirely collected. The message cost corresponds to the best case of back-tracing message complexity.

In this situation, the back-tracing algorithm would present a lower cost than the partial tracing algorithm.

- Suppose that several objects initiate a partial tracing. In the worst case, every object starts a partial tracing. In this situation every partial tracing must co-operate in order to collect the garbage structure. If the co-operation is not set, the collection fails. The message cost corresponds to the worst case of the message complexity formula and represents a wasted effort. If the co-operation is set, and this depends on the values estimated for $\mathcal{T}_{pt}$, the whole structure would be covered by the co-operating partial tracings and entirely collected. The cost for each partial tracing is defined by the best case of the message complexity formula. The total cost depends on the number of partial tracings co-operating on the collection of this structure, d, that is the number of participants that have initiated a partial tracing. The message complexity formula would be:

$$2d^2 + \sum_{i=1}^{d} e_i + 4(n_i - 1)$$

  The formula is an overestimate as some of the token passing that implement the distributed termination protocol could be short circuited. e_i and n_i are related to each co-operative partial tracing. Generally, for each partial tracing $e_i \ll E$ and $n_i \ll N$. For this particular case, for each partial tracing $d = N$ because every object started a partial tracing. However, as we said when reasoning about multiple simultaneous collections, we may expect $d \ll N$.

- Suppose that several objects initiate a back-tracing algorithm. In the worst case, every object starts a back-tracing. The whole structure is entirely collected. The message cost corresponds to the worst case of the message complexity. The message complexity formula would be:

$$\sum_{i=1}^{m} e + (n - 1)$$

  For each initiated back-tracing, $e = E$ and $n = N$. In this particular case $m = N$. As in the partial tracing algorithm, it is expected $m \ll N$.

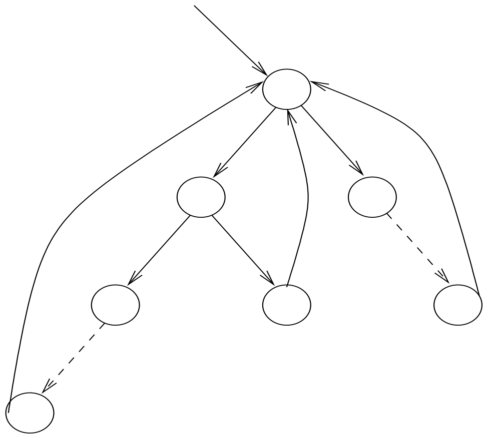


Figure 44: Searching tree with back references

Comparing the two formulae, $\sum_{i=1}^{d} e_i \ll \sum_{i=1}^{m} e$ and $\sum_{i=1}^{d} 4(n_i-1) \ll \sum_{i=1}^{m}(n-1)$. The partial tracing algorithm also presents the cost $2d^2$ that in this particular situation equals $2N^2$. But, as we said, we expect $d \ll N$ in the majority of the cases. Thus, even if m and d have the same order of complexity, it is likely that the partial tracing algorithm would present a better message complexity.

How can $\mathcal{T}_{pt}$ be estimated? The system may estimate $\mathcal{T}_{pt}$ as the time needed to propagate the mark-red tracing through a conservatively estimated (large) list's length. In this way, if every partial tracing delays the beginning of the scan phase by $\mathcal{T}+\mathcal{T}_{pt}$, the partial tracings initiated in every object of the linked list are likely to meet.

We show in figure 44 a searching tree, with back references to the root, of N nodes, each located in a different process, and E inter-process references. After the deletion of the reference to the root of the, every object will become a suspect object as a consequence of multiple local collections in each process. In the same way, the behaviour of the partial tracing algorithm depends on which garbage object starts a partial tracing.

The analysis of the behaviour of the partial tracing and back-tracing algorithm in the collection of this structure is essentially the same as the analysis of the double linked list. This comes from the fact that, as in the double linked list, we may find a path between any two different objects in the structure. It can happen that an object in the tree initiates a partial tracing and propagates it to the other objects in the tree. In this case, the whole structure is entirely collected by that partial tracing. Alternatively,

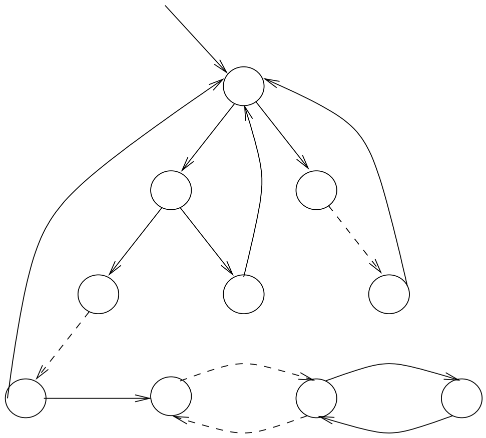


Figure 45: Tree nodes pointing to cyclic garbage structures

several objects may start a partial tracing. In this case, the partial tracings must co-operate in collecting the garbage structure.

A different behaviour would arise if the nodes of the tree structure point to any arbitrary structure such a linked list, as represented in figure 45.

Every partial tracing initiated in a linked list, pointed by a node in the tree, will fail. The wasted effort is measured by the worst case in the message complexity formula. However, recall that the whole tree structure is garbage. A partial tracing initiated in any node of the tree would cover the linked list or would meet any partial tracing that had initiated in the linked list. The system may estimate $\mathcal{T}_{pt}$ as the time need to propagate the mark-red tracing through a conservatively estimated (large) length of the path between the root and a leaf of the tree. In this way, if every partial tracing delays the beginning of the scan phase by $\mathcal{T} + \mathcal{T}_{pt}$, the partial tracing initiated in the linked list is likely to be encountered by a partial tracing initiated in some node of the tree.

A back-tracing initiated in any element of the tree or linked list structures would discover the target suspect sub-graph to be garbage. Again, the back-tracing algorithm cost depends on the number of objects that initiated a garbage collection cycle. The cost of the back-tracing algorithm increases substantially as the number of objects that initiate simultaneous back-tracings increases.

Mutator Overhead

Mutator overhead due to garbage collection must be minimised. Our system performs garbage collection concurrently and asynchronously with respect to mutators. All partial tracing steps are performed concurrently with the mutator. In particular, the scan phase initial step requires atomic execution with respect to *cut-references* updating, but this does not affect the mutator activity.

As we stated in section 4.2.2, mutator actions may generate synchronisation actions in order to ensure safety. Our system has two synchronisation points as we described in section 6: a **Dirty Barrier** every time a remote object is invoked and a **Remote Barrier** every time a red remote object is invocated through a red exit-item or transmitted. As our implementation showed, these barriers are cheap. The *Dirty Barrier* only involves setting an entry-item Ei_z, and every exit-item where $Ex_y \in Ei_z$.exits, dirty. Dirty items are considered members of the *local-scan-root-set* for the scan phase initial step.

The **Remote Barrier** causes the mutator to generate a scan request only on the first occasion in a collection cycle that a message is sent from a red exit-item. This scan request is piggy-backed onto the reference listing protocol operations or mutator messages (see section 8.2.3). Additionally, the probability of mutating objects corresponding to red entry or exit-items decreases with better heuristics.

In order to scan phase to terminate, all scan requests must be acknowledged. Mutator actions on red exit-items spawn new scan requests, however. The number of red entry and exit-items is finite, hence, a finite number of scan requests is generated. Consequently, mutator actions do not delay scan phase termination.

9.1.3 Fault-tolerance

Until now we have assumed that processes do not crash, and that they communicate by messages which are guaranteed to be delivered. As we have already said, in a distributed system a message may be lost, duplicated or delivered out of order. Processes may become disconnected for a long period of time and may also crash.

In this section we extend our partial tracing to realistic distributed system behaviour. We assume that messages may be lost, duplicated or delivered out of order. We further assume that crashes are fail-stop, therefore the only consequence of a crash is disconnection, loss of volatile memory and the halting of computation. In this section we do not address recovery, that is, all objects contained in a process at the time of the crash are assumed to be deleted.

Our partial tracing is built on top of the reference listing mechanism described in section 3.4.4. Consequently we inherit the following features (Birrel and Nelson 1984):

- a protocol for maintaining entry and exit-items which is robust in the face of lost, duplicated and out of order messages.
- a protocol for detecting those objects that are referenced by crashed processes.
- a protocol for handling dangling references from objects to processes that appear to have crashed.

Message failure

We identify four types of messages in our system: *mark request messages*, *acknowledgement messages*, *report messages* and *token messages*. These messages may be lost, duplicated or delivered out of order, without compromising the safety of our system.

mark requests messages(*entry-item*) perform two kind of actions: (i) insert source exit-item in the *grey-set*; and (ii) send a mark-request to the target entry-item: colour the entry-item and, for the mark-red phase case, insert the source process in the entry-item's *red-list*; if the target process is passive, insert the source exit-item in the *reply-set*.

These actions are idempotent because set insertion is an idempotent operation, and we do not perform any removal operations. The acknowledgement system protects our system from lost messages.

acknowledgement messages(*exit-item*) remove the exit-item from the *grey-set*. This is an idempotent message because set removal operations are idempotent, and, for a particular partial tracing, an item is never re-inserted into a *grey-set*. The loss of a message is detected because an exit-item is not removed from the *grey-set* until the corresponding acknowledgement is received.

report messages inform the initiator of the end of the initial step and respective responsibles. By their nature, these messages are idempotent. The initiator is aware of the participants in a partial tracing and, consequently, the loss of such a message will be noticed by it.

token messages may be safely retried. The initiator process just discards any copy it is already aware of.

Finally, these messages may be delivered in any order. There are no potential race conditions between them. Race conditions with mutator messages are safely handled by our concurrency model as we have proved in chapter 7.

Messages delay is handled by the acknowledgement system. A non-acknowledged request will make a partial tracing PT_z to fail. A message from PT_z's cycle n may interfere with PT_z's cycle $n+1$. A sequence numbering for each message handles such delays.

Process failures

As we showed, our system is fault-tolerant to message failures. Additionally, we tolerate process crashes in the sense that a partial tracing may start even if other processes in the system are down. Hence, we trade completeness against promptness.

We do not handle a failed participant. We propose such a research as a future work.

9.2 Future Work

9.2.1 Prototype Implementation

An obvious step for future work is to improve the current prototype implementation. It should be extended with the scalable version described in section 5.1.

With the advent of new object-based distributed programming systems, including mobile computing technology, applications may make more use of cyclic distributed garbage collection. As an example, researchers are turn their attention to distributed garbage collection, including cyclic garbage collection, for distributed standards such as CORBA (Vinoski 1993), as recent "Request for Proposals" have shown (Carlini 1997), and to the Java Remote Method Invocation protocol (Gosling and McGilton 1995).

9.2.2 Performance Evaluation

Another future direction is to analyse the performance of our system. We do not know of any performance measurements of cyclic distributed garbage collection algorithms.

Unfortunately, since cyclic distributed garbage collection is not currently widely available, there are few applications that make full use of distributed garbage collection. Hopefully, the number of these applications will start growing once distributed garbage collection is widely spread.

We decided not to produce synthetic benchmarks for the following reasons. As (Wilson 1995) states, to build a benchmark important characteristics of workload must be known. In our case, the probabilities of relevant characteristics of the behaviour of distributed applications must be known. However, for the reasons above we do not have that knowledge. Also, synthetic benchmarks usually make false assumptions, for example that memory allocation by programs is random. (Wilson 1995) shows that real programs do not generally behave randomly. They are designed to solve actual programs, and the method chosen to solve these problems has a strong effect on their pattern of memory usage. Until much deeper understanding of program behaviour is available, the only reliable method for simulation is to use real applications.

These statements enforce the need for studying real distributed applications and using them to test garbage collection systems. In the distributed memory management case, this means that there is an urgent need for studies of topology or demographics of distributed object systems.

These studies should answer questions like: At what rate is garbage, including cycles, formed? How common are distributed cycles? What is their shape? Is there some degree of locality, or are they randomly spread through the distributed system?

If it is the case, as stated by (Wilson 1995), that the patterns of memory usage are related with a specific method, our system would benefit from such a study. The mark-red phase could be supported by hints from the programmer or compiler, or from an event history, improving the efficiency of our cyclic algorithm.

9.2.3 Fault-tolerance

Our system is fault-tolerance to message failures and process failures: it handles safely loss, duplication and out of order delivery of messages; and it does not need the co-operation of all processes in the system to perform cyclic collections.

Another issue related to fault-tolerance needs further research, however. This issue is how to handle a failed participant, that is, garbage collection must remain safe and live under adverse conditions.

We intend to study more deeply the relation between our system and fault-tolerant reference listing, to produce a more fault-tolerant cyclic distributed garbage collector. We intend to produce a solution based on identifying every garbage collection cycle message with a sequence number generated by the initiator process. This technique would allow our system to always be safely able to turn participant processes on non-participant processes.

9.2.4 Related Areas

Partitioned garbage collection in persistent stores has much in common with independent collection in RPC-based distributed systems. Large persistent object stores are usually divided into partitions that are collected independently, for example (Maheshwari and Liskov 1997b, Printezis, Atkinson, Daynes, Spence and Bailey 1997).

To trace a partition independently of the others, each partition must remember references to its objects from other partitions and use them as roots. So, our partitioned garbage collection model described in chapter 3 may be adapted to garbage collection in persistent object stores.

Partitioned garbage collection introduces two problems in the context of garbage collection in persistent object stores (Maheshwari and Liskov 1997b). One is performance: maintaining information about inter-partition references has a space and time overhead. The other is completeness: tracing from inter-partition references does not collect garbage cycles that span partitions. Again, the second problem has much in common with the completeness problem for garbage collection in RPC-based systems.

We shall concentrate on the second problem. The work of (Maheshwari and Liskov 1997b) proposes a global marking scheme to ensure the collection of cycles of garbage that span partitions. Global marking is piggy-backed on partitioned collection, but cyclic garbage can still only be collected when the whole persistent store is marked. We propose PTs on suspect objects (entry and exit-items may be assigned with a distance). We will outline in which way our PT system may improve completeness while preserving the localised nature of partitioned garbage collection and improving promptness.

The global marking is piggy-backed on regular tracing of partitions. A *marked trace* marks from persistent roots, application roots and marked entry-items. Objects reached during this trace are marked, while inter-partition references reached are recorded to be marked when the target partition is garbage collected.

Our scheme may improve promptness in two ways. A first solution would use the mark-red phase as a group-formation heuristic. Mark-red phases may be piggy-backed on partition collection starting from suspect objects and build a group of partitions that may contain garbage cycles. A group marking could then be performed on a group of partitions that have resulted from the mark-red phase. Such information could even contribute to partition selection policies.

Second, we could try a more dynamic solution and perform PTs as we have described in chapter 5:

- Mark-red and scan phases could be piggy-backed on partition collection. A modified partition collection could implement Tarjan's algorithm, and propagate red and green marks from entry to exit-items. At the end of a collection red and green marks for inter-partition references would be updated, providing red or green roots for other partitions and updated *red-lists*. Additionally, the participants must be recorded in some PT object akin to $PTObj$ of definition 5.3.

- For detecting termination, in each phase, a mark bit would be assigned to every participant partition to denote whether the red or green marks of its items had been propagated. Marking-red or scanning a partition causes it to become marked, but may cause other partitions to become unmarked, because more entry-items are marked and such mark must be propagated. Termination would be detected when all participants would had been marked.

- PTs that encounter each other in the mark-red phase would be allowed to join. Different policies could be adopted as we have discussed in section 9.1 depending on how far completeness may be relaxed.

- Synchronisation with mutator activity and scan phase can be avoided during one partition collection, if garbage collection is run as an *optimistic transaction* (Tanenbaum 1992), that aborts in case of concurrent writes by an application (Shapiro and Ferreira 1995). However, synchronisation is still needed to detect reachability changes of red items (local reachability as well as inter-partition reachability) in a partition P after the scan *Initial-step*(P) has been taken, since the

Initial-step is not synchronised in every partition. Recall that after the *Initial-step*, red exit-items are only reachable from outside the partition. This allows us to check at commit time the transaction's *read-write log*. If an object corresponding to a red entry-item has been read, the entry-item is marked green. For this check to be allowed to be done lazily at commit time, termination must only be detected when all partitions stay unmarked after the log has been checked.

- As in (Maheshwari and Liskov 1997b), colour information may be maintained on entry and exit-items that are stored as regular persistent objects, as well as mark bits. They would be updated after tracing a partition. This, in conjunction with transaction recovery methods, would allow the restoration of the state of a PT after recovery.

The main benefits of applying a PT system instead of a global marking are:

- termination is fast. Only the scan phase needs mutator synchronisation. There are a finite number of red objects, so a partition can only be unmarked a finite number of times. Additionally, if the PT is working in garbage objects, partition unmarking will not happen.
- PTs work only in a subset of the object store. Consequently we achieve better promptness.

From this analysis we conclude that, although needing further investigation, our scheme looks promising for garbage collection on persistent object stores.

Bibliography

Agha, G. (1986). *Actors: a Model of Concurrent Computation in Distributed Systems*, MIT Press.

Amsaleg, L., Franklin, M. and Gruber, O. (1995). Efficient incremental garbage collection for client-server object database systems, *Proceedings of the VLDB International Conference on Very Large Data Bases*.

Appel, A. W., Ellis, J. R. and Li, K. (1988). Real-time concurrent collection on stock multiprocessors, *ACM SIGPLAN Notices* **23**(7): 11–20.

Atkison, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. (1983). An approach to persistent programming, *Computer Journal* **26**(4): 360–365.

Augusteijn, L. (1987). Garbage collection in a distributed environment, *PARLE'87 Parallel Architectures and Languages Europe*, Vol. 259 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 75–93.

Babaoglu, O. and Marzullo, K. (1993). Consistent global states of distributed systems: Fundamental concepts and mechanisms, *in* S. Mullender (ed.), *Distributed Systems*, Addison-Wesley, pp. 55–96.

Baker, H. G. (1978). List processing in real-time on a serial computer, *Communications of the ACM* **21**(4): 280–294. Also AI Laboratory Working Paper 139, 1977.

Bartlett, J. F. (1988). Compacting garbage collection with ambiguous roots, *Technical Report 88/2*, DEC Western Research Laboratory, Palo Alto, California. Also in Lisp Pointers 1, 6 (April–June 1988), pp. 2–12.

Bartlett, J. F. (1989). Mostly-Copying garbage collection picks up generations and C++, *Technical note*, DEC Western Research Laboratory, Palo Alto, CA. Sources available in ftp://gatekeeper.dec.com/pub/DEC/CCgc.

Bevan, D. I. (1987). Distributed garbage collection using reference counting, *PARLE Parallel Architectures and Languages Europe*, Vol. 259 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 176–187.

Bharat, K. A. and Cardelli, L. (1995). Migratory applications, *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*.

Birrel, A. D. and Nelson, B. J. (1984). Remote procedure call, *ACM Transactions on Computer Systems* **2**(1): 39–59.

Birrel, A., Evers, D., Nelson, G., Owicki, S. and Wobber, E. (1993). Network objects, *Technical report SRC 115*, Digital Systems Research Center.

Birrel, A., Evers, D., Nelson, G., Owicki, S. and Wobber, E. (1994). Distributed garbage collection for network objects, *Technical report SRC 116*, Digital Systems Research Center.

Bishop, P. (1977). Computer systems with a very large address space and garbage collection, *Technical Report MIT Rep, LCS/TR–178*, Laboratory for Computer Science, M.I.T., Cambridge, Mass.

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

Boehm, H.-J. and Weiser, M. (1988). Garbage collection in an uncooperative environment, *Software Practice and Experience* **18**(9): 807–820.

Boehm, H.-J., Demers, A. J. and Shenker, S. (1991). Mostly parallel garbage collection, *ACM SIGPLAN Notices* **26**(6): 157–164.

Brownbridge, D. R. (1985). Cyclic reference counting for combinator machines, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, Vol. 201 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 256–272.

Carlini, G. (1997). Anyone interested in reviewing a dgc rfp for CORBA?, gclist (gclist@iecc.com).

Cheney, C. J. (1970). A non-recursive list compacting algorithm, *Communications of the ACM* **13**(11): 6–8.

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

Christopher, T. (1984). Reference count garbage collection, *Software Practice and Experience* **14**(6): 503–507.

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

Cook, J., Klauser, A. W., Wolf, A. and Zorn, B. (1996). Semi-automatic, self-adaptive control of garbage collection rates in object databases, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD, pp. 377–388.

Cook, J., Wolf, A. and Zorn, B. (1994). Partition selection policies in object database garbage collection, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD, pp. 371–382.

Derbyshire, M. H. (1990). Mark scan garbage collection on a distributed architecture, *Lisp and Symbolic Computation* **3**(2): 135 – 170.

DeTreville, J. (1990). Experience with concurrent garbage collectors for Modula-2+, *Technical Report 64*, DEC Systems Research Center, Palo Alto, CA.

Deutsch, L. P. and Bobrow, D. G. (1976). An efficient incremental automatic garbage collector, *Communications of the ACM* **19**(7): 522–526.

Dickman, P. (1992). Optimising weighted reference counts for scalable, fault-tolerant distributed object-support systems, Unpublished.

Dijkstra, E. W. and Scholten, C. (1980). Termination detection for diffusing computations, *Information Processing Letters* **11**: 1–4.

Dijkstra, E. W., Feijen, W. and van Gasteren, A. (1983). Derivation of a termination detection algorithms for distributed computations, *Information Processing Letters*.

Dijkstra, E. W., Lamport, L., Martin, A., Scholten, C. and Steffens, E. (1978). On-the-fly garbage collection: An exercise in cooperation, *Communications of the ACM* **21**(11): 965–975.

Edelson, D. R. (1992). Precompiling C++ for garbage collection, *Proceedings of International Workshop on Memory Management, St. Malo, France*, Vol. 637 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Ellis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*, Addison-Wesley.

Ferreira, P. (1996). *Larchant: ramasse-miettes dans une mémoire partagée répartie avec persistance par atteignabilité*, PhD thesis, L'Université Pierre & Marie Curie - Paris VI.

Ferreira, P. and Shapiro, M. (1996). Larchant: Persistence by reachability in distributed shared memory through garbage collection, *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS), Hong Kong*.

Franklin, M., Copeland, G. and Weikum, G. (1989). What's different about garbage collection for persistent programming languages, *Technical Report ACA-ST-062-89*, MCC Information Center, 3500 W. Balcones Center Drive, Austin, TX 78759-6509.

Friedman, D. and Wise, D. S. (1977). The one-bit reference count, *BIT* **17**(3): 351–359.

Friedman, D. and Wise, D. S. (1979). Reference counting can manage the circular environments of mutual recursion, *Inf Process. Lett.* **8**(1): 41–45.

Fuchs, M. (1995). Garbage collection on an open network, *IWMM95*, Vol. 986 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 251–265.

Godard, I. (1994). Re: Collecting distributed cycles of garbage, USENET comp.object.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

Gosling, J. and McGilton, H. (1995). The java language environment: A white paper, Available from http://www.javasoft.com/whitePaper.

Gupta, A. and Fuchs, W. K. (1993). Garbage collection in a distributed object-oriented system, *IEEE Transactions on Knowledge and Data Engineering* **5**(2): 257–265.

Hudak, P. R. and Keller, R. (1982). Garbage collection and task deletion in distributed applicative processing systems, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pa.*, pp. 68–78.

Hudson, R. L., Morrison, R., Moss, J. E. B. and Munro, D. S. (1997). Garbage collecting the world: One car at a time, *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications*, Vol. 32 of *ACM SIGPLAN Notices*, ACM, pp. 162–175.

Hughes, R. J. M. (1985). A distributed garbage collection algorithm, *Proceedings of the 1985 FPCA*, Vol. 201 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 256–272.

Jones, R. E. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons. With a chapter on Distributed garbage collection by R. Lins.

Jones, R. E. and Lins, R. D. (1993). Cyclic weighted reference counting without delay, *in* A. Bode, M. Reeve and G. Wolf (eds), *PARLE'93 Parallel Architectures and Languages Europe, Munich*, Vol. 694 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Jul, E., Levy, H., Hutchinson, N. and Black, A. (1988). Fine-grained mobility in the Emerald system, *ACM Transactions on Computer Systems* **6**(1): 109–133.

Juul, N.-C. and Jul, E. (1992). Comprehensive and robust garbage collection in a distributed system, *IWMM92*, Vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Kafaru, D., Washabaugh, D. and Nelson, J. (1990). Garbage collection of actors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, ACM.

Kernighan, B. W. and Ritchie, D. M. (1990). *The C Programming Language*, Prentice Hall.

Kolodner, E. and Weihl, W. (1993). Atomic incremental garbage collection and recovery for a large stable heap, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD, pp. 177–186.

Ladin, R. and Liskov, B. (1992). Garbage collection of a distributed heap, *International Conference on Distributed Computing Systems, Yokahama*.

Lang, B., Quenniac, C. and Piquer, J. (1992). Garbage collecting the world, *ACM Symposium on Principles of Programming, Albuquerque*, pp. 39–50.

Lermen, C.-W. and Maurer, D. (1986). A protocol for distributed reference counting, *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, pp. 343–350.

Lieberman, H. and Hewitt, C. (1983). A real-time garbage collector based on the lifetimes of objects, *Communications of the ACM* **26(6)**: 19–29. Also report TM–184, Lab. for Computer Science, M.I.T., Cambridge, Mass.,July 1980.

Linington, P. F. (1992). Introduction to the open distributed processing basic reference model, *Open Distributed Processing*, Elsevier Science Publishers B. V. (North Holand).

Lins, R. D. (1990). Cyclic reference counting with lazy mark-scan, *Technical Report 75*, The University of Kent at Canterbury Computing Laboratory, The University, Canterbury, Kent. Also Information Processing Letters 44(4):215–220, 1992.

Lins, R. D. and Jones, R. E. (1993). Cyclic weighted reference counting, *in* K. Boyanov (ed.), *Procedings of WP & DP'93 Workshop on Parallel and Distributed Processing*. Also Computing Laboratory Technical Report 95, University of Kent, December 1991.

Liskov, B., Day, M. and Shrira, L. (1992). Distributed object management in Thor., *Proc. Int. Workshop on Distributed Object Management*, Edmonton(Canada), pp. 1–15.

Louboutin, S. and Cahill, V. (1995). On comprehensive global garbage detection, *Proceeding of the European Research Seminar on Advances in Distributed Systems (ERSADS '95)*, INRIA/IMAG, Grenoble, pp. 208–213. Also technical report TCD-CS-95-11, Dept. of Computer Science, Trinity College Dublin.

Louboutin, S. R. Y. (1997). *A Reactive Approach to Comprehensive Global Garbage Detection*, PhD thesis, University of Dublin, Trinity College.

Louboutin, S. R. Y. and Cahill, V. (1997). Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events, *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS)*, IEEE Press.

Maeda, M., Konaka, H., Ishikawa, Y., TomoKiyo, T., Hori, A. and Nolte, J. (1995). On-the-fly global garbage collection based on partly mark-sweep, *Proceedings of International Workshop on Memory Management, Kinross, UK*, Vol. 986 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Maheshwari, U. (1993). *Distributed garbage collection in a client-server, transactional, persistent object system*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of technology.

Maheshwari, U. and Liskov, B. (1994). Fault-tolerant distributed garbage collection in a client-server objected-oriented database, *Proceedings of the third International Conference on Parallel and Distributed Information Systems*, pp. 239–248.

Maheshwari, U. and Liskov, B. (1995). Collecting cyclic distributed garbage by controlled migration, *Proceedings of the Symposium on Principles of Distributed Computing.*

Maheshwari, U. and Liskov, B. (1997a). Collecting distributed garbage cycles by back tracing, *Proceedings of the Symposium on Principles of Distributed Computing.*

Maheshwari, U. and Liskov, B. (1997b). Partitioned garbage collection of a large object store, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD.

Marzullo, K. and Sabel, L. S. (1994). Efficient detection of a class of stable properties, An earlier version of this paper appears in the *Proceedings of the 5th International Workshop on Distributed Systems*, October 1991, Spring-Verlag LNCS Vol. 579.

Mattern, F. (1987). Algorithms for distributed termination detection, *Distributed Computing* **2**: 161–175.

Mattern, F. (1989). Global quiescence detection based on credit distribution and recovery, *Information Processing Letters* **30**(4): 195–200.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, *Communications of the ACM* **3**: 184–195.

Minsky, M. L. (1963). A Lisp garbage collector algorithm using serial secondary storage, *Technical Report Memo 58 (rev.)*, Project MAC, MIT, Cambridge, MA.

Moss, J. E. B., Munro, D. S. and Hudson, R. L. (1996). Pmos: A complete and coarse-grained incremental garbage collector for persistent object stores, *Proceedings of the seventh Workshop on Persistent Object systems.*

Nettles, S. M., O'Toole, J. W., Pierce, D. and Haines, N. (1992). Replication-based incremental copying collection, *in* Y. Bekkers and J. Cohen (eds), *Proceedings of International Workshop on Memory Management*, Vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, Carnegie Mellon University, USA.

Ng, T. C. T. (1996). *Efficient garbage collection for large object-oriented databases*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of technology.

Nitzberg, B. and Lo, V. (1991). Distributed Shared Memory: A survey of Issues and Algorithms., *IEEE Computer* pp. 52–60.

Ozsu, M. T., Daylal, U. and Valduriez, P. (1994). An introduction to distributed object management, *Distributed Object Management*, Morgan Kaufmann Publishers.

Piquer, J. (1991). Indirect reference counting: A distributed garbage collection algorithm, *in* Aarts *et al.* (ed.), *PARLE'91 Parallel Architectures and Languages Europe*, Vol. 505 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Plainfossé, D. and Shapiro, M. (1992). Experience with fault-tolerant garbage collection in a distributed Lisp system, *Proceedings of International Workshop on Memory Management, St. Malo, France*, Vol. 637 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Plainfossé, D. and Shapiro, M. (1995). A survey of distributed garbage collection techniques, *Proceedings of International Workshop on Memory Management, Kinross, UK*, Vol. 986 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Printezis, T., Atkinson, M., Daynes, L., Spence, S. and Bailey, P. (1997). The design of a new persistent object store for PJama, *Technical report*, Department of Computer Science, University of Glasgow, Glasgow G12 8QQ.

Queinnec, C., Beaudoing, B. and Queille, J.-P. (1989). Mark during sweep rather than mark then sweep, *PARLE'89 Parallel Architectures and Languages Europe*, Vol. 365 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Rana, S. P. (1983). A distributed solution to the distributed termination problem, *Information Processing Letters* **17**: 43–46.

Ricciardi, A. M. and Birman, K. P. (1993). Process membership in asynchronous environments, *Technical Report TR 93-1328*, Department of Computer Science, Cornell University, Ithaca NY (USA).

Rodrigues, H. and Jones, R. E. (1996). A cyclic distributed garbage collector for network objects, *in* O. Babaoglu and K. Marzullo (eds), *Tenth Iternational Workshop on Distributed Algorithms (WDAG)*, Vol. 1095 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Rodrigues, H. and Jones, R. E. (1998). Cyclic distributed garbage collection with group merger, *European Conference on Object-Oriented Programming (ECOOP98)*, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

Rodriguez-Riviera, G. (1995). Cyclic distributed garbage collection without global synchronization, PhD Preliminary Examination Report.

Rodriguez-Riviera, G. and Russo, V. (1997). Cyclic distributed garbage collection without global synchronization in CORBA, Presented at International Workshop on Memory Management OOPSLA'97.

Rovner, P. (1985). On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language, *Technical Report CSL-84-7*, Xerox PARC, Palo Alto, Ca.

Samples, A. D. (1992). Garbage collection-cooperative c++, *in* Y. Bekkers and J. Cohen (eds), *Proceedings of International Workshop on Memory Management*, Vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin.

Schelvis, M. (1989). Incremental distribution of timestamp packects: A new approach to distributed garbage collection, *ACM SIGPLAN Notices* **24**(10): 37–48.

Schroeder, M. D. (1993). A state-of-the-art distributed systems computing with BOB, *in* S. Mullender (ed.), *Distributed Systems*, Addison-Wesley, pp. 1–26.

Shapiro, M. and Ferreira, P. (1995). Larchant-rdoss: a distributed shared persistent memory and its garbage collector, *in* J.-M. Hélary and M. Raynal (eds), *International Workshop on Distributed Algorithms (WDAG)*, Vol. 637 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 198–214.

Shapiro, M., Dickman, P. and Plainfossé, D. (1992). Robust, distributed references and acyclic garbage collection, *Proceedings of the Symposium on Principles of Distributed Computing*.

Shapiro, M., Gruber, O. and Plainfossé, D. (1990). A garbage detection protocol for a realistic distributed object-support system, *Rapports de Recherche 1320*, INRIA-Rocquencourt. Also in ECOOP/OOPSLA'90 Workshop on Garbage Collection.

Shivaratri, N. G., Krueger, P. and Singhal, M. (1992). Load Distributing for Locally Distributed Systems, *Computer* **25**(12): 33–44.

Sousa, P., Sequeira, M., Zúquete, A., Ferreira, P., Lopes, C., Pereira, J., Guedes, P. and Marques, J. A. (1993). Distribution and persistence in the IK platform: Overview and evaluation, *Computing Systems* **6**(4): 391–424.

Steele, G. L. (1975). Multiprocessing compactifying garbage collection, *Communications of the ACM* **18**(9): 495–508.

Tanenbaum, A. S. (1992). *Modern Operating Systems*, Prentice Hall.

Tarjan, R. (1972). Depth first search and linear graph algorithms, *SIAM Journal of Computing*.

Tel, G. and Mattern, F. (1993). The derivation of distributed termination detection algorithms from garbage collection schemes, *ACM Transactions on Programming Languages and Systems* **15**(1): 1–35.

Ungar, D. M. (1984). Generation scavenging: a non-disruptive high performance storage reclamation algorithm, *ACM SIGPLAN Notices* **19**(5): 157–167. Also published as ACM SIGPLAN Notices 19, 5 (May 1984) and ACM Software Engineering Notes 9, 3 (May 1984).

Vestal, S. C. (1987). *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*, PhD thesis, University of Washington.

Vinoski, S. (1993). Distributed object computing with CORBA, *C++ Report* pp. 33–38.

Watson, P. and Watson, I. (1987). An efficient garbage collection scheme for parallel computer architectures, *PARLE'87 Parallel Architectures and Languages Europe*, Vol. 259 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 432–443.

Weizenbaum, J. (1963). Symmetric list processor, *Communications of the ACM* **6**(9): 524–544.

Weizenbaum, J. (1969). Recovery of reentrant list structures in slip, *Communications of the ACM* **12**(7): 370–372.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques, *Proceedings of International Workshop on Memory Management, St. Malo, France*, Vol. 637 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Wilson, P. R. (1995). Dynamic storage allocation: A survey and critical review, *Proceedings of International Workshop on Memory Management, Kinross, UK*, Vol. 986 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Wilson, P. R. (1996). Distr. gc general discussion for faq, gclist (gclist@iecc.com).

Wise, D. S. (1993). Stop-and-copy and one-bit reference counting, *Technical Report 360*, Indiana University, Computer Science Department.

Yong, V.-F., Naughton, J. and Yu, J.-B. (1994). Storage reclamation and reorganisation in client-server persistent object stores, *Proceedings of the ICDE International Conference on Data Engineering*, pp. 120–133.

Yuasa, T. (1990). Real-time garbage collection on general-purpose machines, *Journal of Software and Systems* **11**(3): 181–198.

Zorn, B. (1990). Barrier methods for garbage collection, *Technical Report CU–CS–494–90*, University of Colorado at Boulder, Department of Computer Science, Boulder, Colorado.

Zorn, B. (1992). The measured cost of garbage collection, *Technical Report CU–CS–573–92*, University of Colorado at Boulder, Department of Computer Science, Boulder, Colorado.