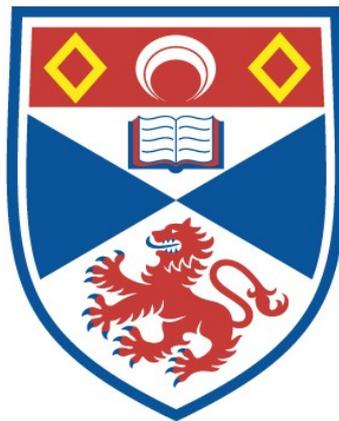


DERIVING DISTRIBUTED GARBAGE COLLECTORS
FROM DISTRIBUTED TERMINATION ALGORITHMS

Stuart J. Norcross

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2004

Full metadata for this item is available in
St Andrews Research Repository
at:
<http://research-repository.st-andrews.ac.uk/>

Please use this identifier to cite or link to this item:
<http://hdl.handle.net/10023/14986>

This item is protected by original copyright

Deriving Distributed Garbage Collectors from Distributed Termination Algorithms

Stuart J. Norcross

PhD Thesis

9th September 2003



School of Computer Science

University of St Andrews

St Andrews

Fife. KY16 9SS

Scotland



ProQuest Number: 10166188

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10166188

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

TH
E560

Abstract

This thesis concentrates on the derivation of a modularised version of the DMOS distributed garbage collection algorithm and the implementation of this algorithm in a distributed computational environment. DMOS appears to exhibit a unique combination of attractive characteristics for a distributed garbage collector but the original algorithm is known to contain a bug and, previous to this work, lacks a satisfactory, understandable implementation.

The relationship between distributed termination detection algorithms and distributed garbage collectors is central to this thesis. A modularised DMOS algorithm is developed using a previously published distributed garbage collector derivation methodology that centres on mapping centralised collection schemes to distributed termination detection algorithms. In examining the utility and suitability of the derivation methodology, a family of six distributed collectors is developed and an extension to the methodology is presented.

The research work described in this thesis incorporates the definition and implementation of a distributed computational environment based on the ProcessBase language and a generic definition of a previously unimplemented distributed termination detection algorithm called Task Balancing.

The role of distributed termination detection in the DMOS collection mechanisms is defined through a process of step-wise refinement. The implementation of the collector is achieved in two stages; the first stage defines the implementation of two distributed termination mappings with the Task Balancing algorithm; the second stage defines the DMOS collection mechanisms.

Acknowledgements

A number of people have helped directly and indirectly with the work described in this thesis. In particular I would like to thank:

Ron Morrison, my supervisor, for his guidance and patience, for the opportunities he has given me and for providing an excellent research environment.

David Munro for all his help and enthusiasm, for all the quality drinking time and for introducing me to research in the first place.

Ian and Alison Norcross, my parents, for their constant support and encouragement throughout my time in St Andrews.

I would also like to thank: Graham Kirby and Alan Dearle for many useful conversations and for always providing a different viewpoint; Aled Sage for all his help from back in the 'early days' right up until the end; the many visitors to the Persistent Programming Group in St Andrews; and everyone in the School who has helped me during my time as a postgraduate.

Thanks to Ron Morrison, David Munro and Graham Kirby for the constructive criticism and insight that they have imparted during the writing of this thesis. Their assistance has been invaluable.

Finally I would like to thank my examiners Paul Watson and Alan Dearle for their helpful suggestions as to how the thesis should be improved, for the interest they have shown in the work and for all their time and effort in examining this thesis.

This work was funded by the EPSRC Distributed Information Systems Initiative under grant GR/M 74931: "Collecting Distributed Garbage using the DMOS Family of Algorithms".

Declarations

I, Stuart John Norcross, hereby certify that this thesis, which is approximately 66,000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

date 14/02/04 signature of candidate _____

I was admitted as a research student in September 1999 and as a candidate for the degree of Doctor of Philosophy in September 2000; the higher study for which this is a record was carried out in the University of St Andrews between 1999 and 2003.

date 14/02/04 signature of candidate _____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 14/02/04 signature of supervisor _____

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

date 14/02/04

signature of candidate _____

Contents

| | | |
|-------|--|----|
| 1 | Introduction..... | 1 |
| 1.1 | An Abstract Distributed System | 5 |
| 1.1.1 | System Model | 6 |
| 1.1.2 | Unreachability: A Stable State..... | 7 |
| 1.1.3 | Desirable Properties for a Distributed Garbage Collector | 9 |
| 1.1.4 | The Distributed Garbage Collection Problem | 11 |
| 1.2 | A Distributed Garbage Collector Derivation Methodology | 12 |
| 1.2.1 | The Derivation Methodology from Blackburn et al. | 13 |
| 1.2.2 | A Separation of Concerns | 13 |
| 1.3 | The DMOS Collector..... | 14 |
| 1.4 | Contribution of the Thesis | 15 |
| 1.5 | Thesis Structure | 16 |
| 2 | Related Work | 17 |
| 2.1 | A Review of Distributed Garbage Collectors | 17 |
| 2.1.1 | Distributed Reference Counting | 17 |
| 2.1.2 | Centralised Control of Distributed Collection | 24 |
| 2.1.3 | Distributed Mark-Sweep..... | 26 |
| 2.1.4 | Distributed Copying Collection | 32 |
| 2.1.5 | Hybrid Distributed Collectors..... | 34 |
| 2.1.6 | Garbage Collecting the World..... | 38 |
| 2.1.7 | The DMOS Collector..... | 40 |
| 2.2 | Distributed Termination Detection | 42 |
| 2.2.1 | A Model for Distributed Termination..... | 43 |
| 2.2.2 | Wave Based Algorithms | 44 |
| 2.2.3 | Credit Recovery | 50 |
| 2.3 | Summary..... | 51 |
| 3 | The Experimental Platform..... | 53 |
| 3.1 | The ProcessBase Language | 53 |
| 3.1.1 | Compliance in ProcessBase | 54 |
| 3.2 | The Distributed ProcessBase Architecture | 56 |
| 3.2.1 | Inter-site Addressing..... | 57 |
| 3.2.2 | The Distributed Object Cache..... | 59 |
| 3.3 | The Distributed ProcessBase Implementation | 62 |
| 3.3.1 | Message Passing in the Distributed VM..... | 62 |
| 3.3.2 | The Local Object Cache Layout | 63 |
| 3.3.3 | The Distributed Object Cache..... | 67 |
| 3.3.4 | Local Garbage Collection | 73 |
| 3.3.5 | Remote Thread Execution | 74 |
| 3.4 | Summary..... | 76 |
| 4 | The Task Balancing Distributed Termination Algorithm..... | 78 |
| 4.1 | A Model for Distributed Termination..... | 79 |
| 4.2 | Task Balancing | 79 |
| 4.2.1 | Termination Detection at the Home Site | 82 |
| 4.2.2 | Implementation Choices | 82 |
| 4.2.3 | An Example Task Balancing Implementation | 90 |
| 4.3 | Why Use TB for DGC Implementation? | 93 |
| 4.4 | Summary..... | 96 |

| | | |
|-------|--|-----|
| 5 | Separating Distributed and Local Collection..... | 98 |
| 5.1 | Forming the Club Rules..... | 102 |
| 5.2 | Distributed Mark-Sweep Collection..... | 103 |
| 5.2.1 | Club Rules for Distributed Mark-Sweep..... | 106 |
| 5.2.2 | Club Rules for a Homogeneous Mark-Sweep Collector..... | 111 |
| 5.2.3 | Separating Local and Distributed Collection..... | 111 |
| 5.2.4 | Local Collection in the Heterogeneous System..... | 114 |
| 5.2.5 | Discussion..... | 116 |
| 5.3 | Distributed Generational Collection..... | 117 |
| 5.3.1 | Two DTA Mappings..... | 119 |
| 5.3.2 | Club Rules for Distributed Generational Collection..... | 121 |
| 5.3.3 | Club Rules for Homogeneous Distributed Generational Collection ... | 125 |
| 5.3.4 | Separating Local and Distributed Collection..... | 126 |
| 5.3.5 | Local Collection in the Heterogeneous System..... | 128 |
| 5.4 | Distributed Reference Counting..... | 129 |
| 5.4.1 | Club Rules for Distributed Reference Counting..... | 130 |
| 5.4.2 | Club Rules for a Homogeneous Reference Counting Collector..... | 131 |
| 5.4.3 | Separating Local and Distributed Collection..... | 133 |
| 5.4.4 | Club Rules for a Heterogeneous Reference Counting Collector..... | 133 |
| 5.4.5 | A Local Mark Sweep Collector..... | 134 |
| 5.4.6 | A Local Semi-Space Copying Collector..... | 134 |
| 5.4.7 | Discussion..... | 135 |
| 5.5 | Summary..... | 136 |
| 6 | Developing a DTA Mapping for DMOS..... | 138 |
| 6.1 | The UMOS Collection Algorithm..... | 139 |
| 6.1.1 | UMOS Safety and Completeness..... | 146 |
| 6.1.2 | Concurrency Issues in UMOS..... | 147 |
| 6.2 | Mapping UMOS to Distributed Termination: The DMOS Algorithm | 149 |
| 6.2.1 | Distributing UMOS..... | 149 |
| 6.2.2 | Train Collection..... | 153 |
| 6.2.3 | Car Collection..... | 156 |
| 6.2.4 | A Summary of the DTA Mappings..... | 167 |
| 6.3 | The Stepwise Refinement of DMOS..... | 169 |
| 6.3.1 | Layer 1: Object Isolation..... | 170 |
| 6.3.2 | Layer 2: Car Reclamation..... | 171 |
| 6.3.3 | Layer 3: Isolated Train Detection..... | 174 |
| 6.3.4 | A DMOS Garbage Collection Cycle..... | 179 |
| 6.4 | Summary..... | 181 |
| 7 | Implementing the DMOS DTA Mappings..... | 183 |
| 7.1 | Cars and Trains in Distributed ProcessBase..... | 184 |
| 7.1.1 | Cars..... | 185 |
| 7.1.2 | Trains..... | 188 |
| 7.2 | Isolated Train and Object Detection with Task Balancing..... | 190 |
| 7.2.1 | Isolated Train Detection..... | 190 |
| 7.2.2 | Isolated Object Detection..... | 197 |
| 7.2.3 | An Optimisation for Task Counting..... | 204 |
| 7.3 | Summary..... | 205 |
| 8 | Implementing DMOS in DPBASE..... | 207 |
| 8.1 | RAL Maintenance..... | 207 |

| | | |
|------------|---|-----|
| 8.1.1 | RAL Updates and Root Reference RAL Entries | 208 |
| 8.1.2 | Adding RAL entries..... | 210 |
| 8.1.3 | Removing RAL Entries | 211 |
| 8.2 | Requesting Train Tasks | 212 |
| 8.3 | Collecting Isolated Trains | 213 |
| 8.4 | Car Collection..... | 215 |
| 8.4.1 | Examining the Local Root Set..... | 216 |
| 8.4.2 | Re-Associating Objects..... | 216 |
| 8.4.3 | Reclaiming a Car | 218 |
| 8.5 | Safety and Completeness of the DMOS Implementation..... | 219 |
| 8.5.1 | Safety | 220 |
| 8.5.2 | Completeness..... | 221 |
| 8.6 | Summary..... | 225 |
| 9 | Conclusions..... | 227 |
| 9.1 | The Mapping Methodology | 227 |
| 9.1.1 | DTA Mappings and Reference Counting Collectors..... | 229 |
| 9.2 | Task Balancing | 230 |
| 9.3 | The DMOS Collector..... | 230 |
| 9.3.1 | DMOS: Reference Counting with Trains | 233 |
| 9.4 | Contribution..... | 234 |
| 9.4.1 | Summary..... | 239 |
| 9.5 | Future Research | 240 |
| 9.5.1 | A Formal Proof for Task Balancing | 240 |
| 9.5.2 | A Modularized Formal Proof of DMOS..... | 241 |
| 9.5.3 | Policy Evaluation..... | 241 |
| 9.6 | Finally | 242 |
| Appendix A | | 244 |
| | An Annotated Implementation of DMOS..... | 244 |
| | DMOS Implementation Pseudo-Code | 248 |
| References | | 260 |

1 Introduction

Automatic storage management in high level languages saves the programmer from the time consuming and error prone task of manually managing the allocation and de-allocation of storage space. Instead, the language runtime systems abstract over the underlying storage mechanisms by dynamically allocating space and automatically reclaiming it when it is no longer used by the application.

This thesis concentrates on the mechanism by which space is reclaimed, known as garbage collection. The work described here concentrates specifically on the derivation and construction of distributed garbage collectors. The aim is to simplify the design and comparability of distributed collectors through modularisation. Central to this work, is the use of distributed termination algorithms in the implementation of distributed garbage collectors.

The discussion of distributed garbage collection in this thesis assumes that the reader is familiar with the more traditional non-distributed garbage collection algorithms. However, in establishing a universe of discourse it is useful to first outline the fundamental model assumptions and techniques underlying all garbage collection.

An object¹ becomes garbage immediately after it is accessed for the last time by the computation. The purpose of a garbage collector is to identify and reclaim these objects². Computing the exact set of garbage objects at any given time in the computation (through static analysis for instance) is at best difficult and

¹ This is object in the loosest sense of the word, which is an identifiable contiguous area of allocated storage space.

² Garbage collectors are typically used to reclaim the space used by objects whose extent exceeds their static scope within the program.

computationally intensive but often not possible. Instead, in systems which guarantee the integrity of references, garbage collectors consider the set of objects allocated by a computation as a rooted directed graph where vertices are objects and directed edges are references. An object is said to be reachable if it can be discovered through the traversal of a path of references from one of the set of root references. In other words, the computation of the transitive closure of the computation from its roots yields the set of reachable objects. Unreachable objects cannot be accessed by the computation and are therefore garbage. Reachability provides a conservative approximation to the set of non-garbage objects and thus allows the calculation of a conservative approximation to the set of garbage objects. This is a conservative approximation because, by the definition of garbage given above, an object that is reachable but which will not be accessed by the application again is garbage.

Wilson [Wil92] defines an abstraction for garbage collection that consists of two parts: garbage identification; and garbage reclamation. Abdullahi and Ringwoods's [AR98] taxonomy of single address space garbage collectors identifies two fundamental techniques for garbage identification;

- Direct identification - also known as reference counting [Col60], is a technique whereby the garbage collector maintains a count of the number of references to each object. When the count reaches zero, the object is garbage since if there exists no reference to the object then it cannot be reachable. Reference counting mechanisms have two key properties. The first is that the work done to reclaim garbage is proportional to the work done by the computation, since the more reference manipulations there are, the more work must be done in maintaining the reference counts. The second is that

cyclic garbage cannot be reclaimed since the reference count for each object in a cycle will never become zero.

- Indirect identification - corresponds more closely to the principle of computing the transitive closure of the object graph. That is, the collector traverses the object graph from the roots of reachability and takes some action for each reachable object it encounters to ensure that the object is not collected. Those objects not encountered by the collector during the traversal are garbage. Broadly speaking, indirect collectors can be categorised as either copying collectors or mark-sweep collectors. Copying collectors segregate the storage space into a number of partitions, and reclaim space with a given partition by creating copies of all reachable objects (in that partition) in some other partition. Any objects not copied are garbage and thus the whole partition may be reclaimed when copying is complete. Copying collectors relocate objects on collection thus compacting the used storage space. The locality properties of stored data may or may not be improved as objects are moved. Mark-sweep collectors trace the object graph and record (mark) any reachable objects. To collect garbage, the space is scanned sequentially and any unmarked objects are reclaimed. During the scan phase of the collection, live objects may be relocated to compact the used space and improve the locality of the data.

As with garbage identification, there are two techniques underlying any garbage reclamation scheme. Either each live object is copied to some part of the managed storage space, where it is guaranteed to be maintained or each garbage object is directly reclaimed and added to a free list. The way in which space is reclaimed is directly associated with the mechanisms by which space is allocated; however no

discussion of allocation is presented here (see [WJN+95] for a review of allocation techniques).

Every garbage collection scheme is based on one or a combination of these two basic techniques. However, the specific nature of individual garbage collection schemes varies widely. Comprehensive reviews of individual uni-processor collection algorithms can be found in [AR98, JL96, Wil92].

Garbage collection algorithms can be seen in use in reclaiming fixed-size double word storage cells in LISP systems as far back as the 1960's. Later, through the 70's and 80's garbage collection was used for the reclamation of non-fixed-size objects in block structured imperative languages, such as the various Algol incarnations; Algol 68 [BLS+71], S-Algol [Mor79] and PS-Algol [ACC82], languages such as Napier-88 [MBC+89] and in functional languages such as ML [MTH89] and its derivatives.

More recent object-oriented systems also incorporate garbage collection techniques from Smalltalk [GR83] of the mid 80's to Sun's Java [GJS+00, LY99] from the late 90's and later Microsoft's C# [DAN02].

These languages support application development for a range of target architectures such as uni-processor architectures, tightly-coupled parallel processor architectures (with physically shared storage) and loosely-coupled distributed processor architectures (with no physically shared storage). The run-time support systems for each of these architectures vary widely in their implementation and complexity. This thesis concentrates on the design and implementation of garbage collectors for automatic storage management in run-time systems for loosely-coupled distributed architectures. Applying the same high level language concepts and storage

management abstractions, particularly garbage collection, in the distributed context has proven to require far more complex language support systems.

To ground the discussion on distributed systems it is necessary to specify the universe of discourse. This is achieved through the definition of an abstract system model.

1.1 An Abstract Distributed System

Lamport [Lam78] describes a distributed system as one consisting of a number of spatially separated processes that communicate through message passing. Given such a definition, the phrase “distributed system” can be used to describe a range of systems from loosely coupled multi-computers (such as a Beowulf cluster [BSS+95]) to tightly coupled (parallel) multi-processor systems. Lamport becomes more specific by defining a distributed system as one where the transmission delay between processes is significantly greater than the delay between events within a single process. [CDK01] and [Sch93] give further properties that define a distributed system, such as multiple computers with inter-connections, concurrency, the lack of a global clock and the occurrence of independent failures.

Distributed automatic storage management is required in distributed systems that exhibit a computationally shared state. Figure 1.1 illustrates such a distributed system. The implementation of such a shared state may be through, for example, a shared name space or a shared address space that operates over the underlying distributed storage systems.

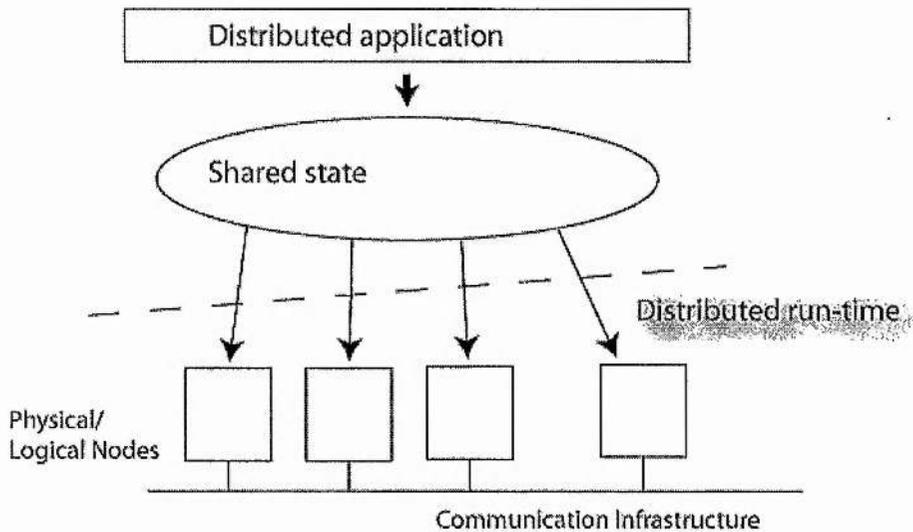


Figure 1.1 - Shared Application State in a Distributed Run-Time

These properties are demonstrated in the system model that follows.

1.1.1 System Model

The system model (taken from [NMM+03]) is defined such that a computation executes over a number of sites where each site acts independently, concurrently and asynchronously. The following assertions are made:

1. Each site has its own local storage and communicates with other sites only through message passing.
2. Local storage is dynamically allocated and automatically (safely) reclaimed.
3. Sites appear to operate correctly, without Byzantine behaviour.
4. There is no bound on the relative rates of computation of the sites.
5. Events at a given site are totally ordered; since messages are delivered only after being sent, events are partially ordered in the system as a whole.
6. Messages are delivered in-order, without omission or corruption.

The distributed computation operates over a set of objects that can be modelled by a directed graph with multiple roots. The graph is mutated by the concurrent computation at each site through a series of explicit object update and object allocation operations. A node of the graph represents an object while a directed edge represents a reference. The object graph is distributed across the sites of the computation and each site maintains zero or more root pointers. An object at a site may contain references to other local objects or to objects on a remote site. An object is said to be garbage after it has been accessed for the last time by the computation.

The set of reachable objects for a computation is defined as the set of all objects that can be reached through some path of references from a root of the computation. The run-time system maintains the property of referential integrity across the (potentially disjoint) graph of reachable objects in the face of two operations; object update (effected by the mutator) and object deletion (effected by the run-time system's storage management mechanisms in reclaiming space). The property of referential integrity is that no object, at any site, can ever contain a dangling reference. Two rules guarantee the referential integrity of the system;

- References are never forged; that is to say references are only ever copied³.
- An object may not be deleted before it becomes garbage.

1.1.2 Unreachability: A Stable State

By maintaining the property of referential integrity over the reachable data, unreachability becomes a stable state. If a reference to an object cannot be found, by tracing a path of references from a root, then it cannot be used in an object update

³ References to new objects represent an exceptional case.

operation. That is, no reference exists to be copied. Thus an unreachable object can never become reachable again.

Referential integrity is maintained without resorting to a centralized service. That is, referential integrity is maintained by an agent at each site operating on local information and cooperating with agents at other sites through asynchronous message passing. To maintain referential integrity on object deletion only unreachable objects may be deleted.

Unreachable objects can be in one of two subsidiary stable states. If either of these states is detected an object can be deemed unreachable, and thus made a candidate for deletion. The first stable state is that of an object referenced by no other object. For some objects this state is detectable with completely local information⁴ and for other objects, a globally consistent view of the (disjoint) distributed object graph is required.

The second stable state for unreachable objects is that of a set of mutually referential objects, where each object in the set is referenced only by other objects in the set. The objects in such a set may be distributed across a number of sites. The members of such a set form a (potentially inter-site) cycle of unreachable objects. Garbage cycles that reside within a single site can be detected through completely local information while to detect inter-site garbage cycles a globally consistent view of the (disjoint) distributed object graph is always required. A distributed garbage collector must detect both of these states and thus provide safe and automatic space reclamation. However, both cyclic and non-cyclic garbage may exist that can be reclaimed without recourse to the distributed garbage collection mechanism and

⁴ Such as objects to which a reference has never been exported to a remote site.

there are potential benefits in allowing this space to be reclaimed independently (from the distributed collection mechanisms) at a site. Separation of local and distributed collection work is central to the work described in this thesis and is achieved through the use of a methodology that allows for a systematic modularisation of distributed garbage collector design and implementation; for instance allowing independent local garbage collection.

1.1.3 Desirable Properties for a Distributed Garbage Collector

A number of desirable properties for distributed garbage collectors are presented in [HMM+97]. Two of these properties are of fundamental importance in any (distributed or centralised) garbage collector;

- Safety – No object should be reclaimed before it would have been accessed for the last time by the application.
- Completeness - The property of completeness ensures that every garbage object is eventually reclaimed. A property of the automatic storage management (ASM) abstraction is that storage space is not exhausted until all available space is filled with live objects. Space leakage, whereby the storage space contains garbage objects that are never reclaimed by the storage management mechanism, breaks the ASM abstraction. Given that the discussion here is based on reachability being used to determine garbage, the abstraction is refined to ensure that the storage space is not exhausted until all storage is filled with reachable objects.

Given the system model described here, there are a number of further properties that one would wish of a distributed garbage collector;

- Non-disruptive – Each invocation of the collector carries out a bounded amount of work, thereby bounding the time taken to collect and the space required.
- Incremental - Space is reclaimed in increments without global knowledge of the system state.
- Non-blocking – The collector does not require synchronisation between sites.
- Scalable – No restrictions on scaling of the distributed system are introduced by the distributed collector. This is achieved by ensuring that communication between sites is asynchronous; protocols do not require the participation of all sites and sub algorithms (within the collector) are not centralised. However, scalability is not a property that can be achieved in isolation. To achieve scalability the collector must demonstrate each of the properties described above.
- Independence – Collection progress can be made at a site independently from other sites.
- Performance – Typically the performance of garbage collectors is measured in terms of metrics based on quantifying throughput or pause time. This thesis concentrates on the simplifying design processes for distributed collector, paying particular attention to the correctness and understandability of collectors. However, each of the above fundamental properties is required if high throughputs and low pause time are to be achieved.

Each of these properties affects the intrusiveness of the distributed garbage collection mechanism both on the implementation of the distributed system and on run-time behaviour.

1.1.4 The Distributed Garbage Collection Problem

The distributed garbage collection problem stems from the desire to apply the high level language abstraction of automatic storage management, more specifically garbage collection, to distributed systems. However, applying the same memory abstractions in a distributed environment requires more complex garbage collection mechanisms.

Many distributed garbage collectors have been published (see [AR98, PS95] for an overview); each with interesting and attractive properties. However, distributed garbage collectors are often difficult to understand. There are several reasons why this is the case. Solutions to the distributed garbage collection problem are necessarily complex, involving the co-ordination of independently operating asynchronous agents, each working with partial information while constructing a safe approximation to a globally consistent view. Achieving any one of the properties that we seek in distributed garbage collectors is difficult enough in a complex distributed system but achieving some or all in combination greatly increases the difficulty of collector design and implementation.

Comparing two independently published collectors is not a trivial task. The assumptions made of the system model can vary widely between two independently published collectors. Different model assumptions for communications infrastructure, fault tolerance, mutator activity and concurrency can lead to a range of subtly different algorithms.

In the face of such complex systems and differing model assumptions, collector designers are forced to produce hand-crafted correctness and completeness arguments. This often adds to the difficulty in understanding individual collectors and in comparing two collectors.

1.2 A Distributed Garbage Collector Derivation

Methodology

A methodology for the derivation of distributed garbage collectors is presented in [BHM+01] and developed in [NMM+03] and describes a structured approach to the derivation of distributed garbage collectors with the aim of improving understandability and comparability in the resultant collectors. The methodology consists of a number of steps (described in Section 1.2.1 below and demonstrated later in Chapter 5) that are followed in order to transform a centralised garbage collector into a distributed garbage collector. The methodology builds on the result from Tel and Mattern [TM93] that all distributed garbage collectors contain an implementation of at least one distributed termination detection algorithm (DTA), a connection that is explored in more detail in Chapter 2.

The result from Tel and Mattern suggests that it is possible to modularise the design of a distributed garbage collector, by incorporating a DTA through a mapping onto a centralised collection scheme. The DTA takes on the role of detecting globally stable states within the distributed collector where the nature of these states is determined by the particular mappings used. It should be noted that the mapping process is not automatic and requires creativity on the part of the distributed collector designer. The benefit of using DTAs in this modularisation is that the field has a rich literature with many well understood algorithms for which exist a number of formal proofs.

1.2.1 The Derivation Methodology from Blackburn et al.

Blackburn et al. ([BHM+01]) suggest that the derivation of distributed garbage collectors can be structured through the mapping of distributed termination algorithms onto known centralized collection schemes as follows:

- Select or derive a distributed termination algorithm that is proven correct.
- Prove safety, and maybe some other properties, of the centralised garbage collector.
- Define an object reclamation mapping, from the centralised garbage collector to the distributed termination algorithm.
- Prove that termination is equivalent to the eventual reclamation of objects.

The methodology starts by making a centralised collector concurrent and then mapping a DTA onto the resultant collector to provide a distributed garbage collection scheme. The resultant distributed collector maintains the properties of the original centralised scheme such as completeness and incrementality.

1.2.2 A Separation of Concerns

The complexity of distributed garbage collectors is addressed through a separation of concerns in the design of the distributed collector. The implementation of distributed termination detection necessary due to distribution is removed from the designer who can instead concentrate on garbage collection.

The derived collectors in Chapter 5 are defined by a set of club rules for each participating site. The club rules allow for a clear distinction between distributed and local collection work. With the rules of participation clearly defined it is possible to concentrate on freeing up the local behaviour for sites. That is, to minimise the constraints placed on each site by the distributed collector and allowing sites a wider

choice of policy in scheduling and controlling collection work. The club rules provide a secondary separation of concerns between implementing the distributed mechanism for distributed garbage identification and the purely local mechanisms by which space is reclaimed.

1.3 The DMOS Collector

DMOS [HMM+97] is a distributed garbage collection algorithm that exhibits all of the attractive properties listed earlier. DMOS is safe, complete, non-disruptive, incremental, non-blocking, independent and scalable. The collector partitions objects by cars and trains [HM92]. Cars are local to a site while trains represent groups of cars that can span multiple sites. DMOS can be considered as consisting of two parts. The first part is called the *pointer tracking protocol* which is an implementation of the Task Balancing DTA [BHM+01, NMM+03] and identifies objects at a site that are not referenced from anywhere in the system. The second part is the train reclamation protocol which is an implementation of a wave based DTA which detects isolated trains.

Blackburn and Zigman [BZ99] identified a bug in DMOS which was due to an unanticipated race condition between the two DTAs. The result of the race condition was that a train could be reclaimed while its cars still contained live objects.

The race condition occurs as a result of an optimisation to the DMOS collector whereby object references are opaque. That is, a reference to an object does not encode the train or car holding that object. Thus, an object can be moved between cars and trains at a site without the need to update all remote references to the object. However, the manipulation of references at a site can have effects on the reachability of trains that cannot be identified at that site. This is because the referencing site does not know the trains holding the objects that the site references.

Only the site holding an object knows the train in which that object is held. Thus, the effects on the reachability of the train holding a particular object due to references to that object is determined by information sent to the site holding the object through the *pointer tracking* protocol. The *pointer tracking* mechanism assumes a fully connected communications network allowing direct site to site communication while the train reclamation mechanism assumes a logical ring topology. Since there is not necessarily any overlap in the communications path between two sites for the two mechanisms, it is possible for train isolation messages to overtake *pointer tracking* messages, thus creating a race condition.

To-date no satisfactory implementation of the DMOS collector has been produced and the interaction of the two DTAs has yet to be suitably defined

The hypothesis being tested in this thesis is that there is benefit in applying modularisation to the design of distributed garbage collectors. Specifically, that an extended version of the mapping methodology can be used to guide the development of a modularised and understandable implementation of DMOS thus yielding an explanation of the interaction of the two collection mechanisms and defining the exact role played by the distributed termination algorithms.

1.4 Contribution of the Thesis

The contribution of this thesis is four-fold. First, the practical application of a previously published derivation methodology is demonstrated through the derivation of six collectors. It is then shown that the methodology can be extended to produce distributed collectors that allow for locally independent collector behaviour through the specification of the club rules for the distributed scheme.

Secondly, a platform for experimenting with the implementation of distributed garbage collectors, which represents an instantiation of the system model described above.

Thirdly, an implementation of the Task Balancing distributed termination algorithm is demonstrated. This is believed to be the first such implementation.

Fourthly, a new implementation of the DMOS collector is presented. Having shown the suitability and flexibility of the derivation methodology, it is used to produce an implementation of the previously published DMOS collection mechanism, the implementation of which has to-date been unsatisfactorily described.

1.5 Thesis Structure

Chapter 2 presents a review of previously published distributed garbage collection algorithms and examines the link to distributed termination detection. Particular attention is given to the DMOS algorithm.

Chapter 3 discusses an experimental platform that represents an instantiation of the abstract system described above. This is the target system for the derived distributed garbage collectors.

Chapter 4 examines the distributed termination problem and explains the Task Balancing DTA and the associated implementation issues.

Chapter 5 presents three example derivations each of which yields two distributed collection mechanisms; one homogenous mechanism and one heterogeneous mechanism that allows for locally independent collector behaviour.

Chapters 6, 7 and 8 discuss the derivation and implementation of a DMOS collector.

2 Related Work

This chapter presents an overview of the two fields that are at the heart of this thesis, namely distributed garbage collection and distributed termination detection. Survey works in the field of distributed garbage collection have been published by Plainfossé and Shapiro [PS95], by Jones and Lins [JL96] and by Abdullahi and Ringwood [AR98]. A taxonomy of distributed termination algorithms is described by Camp and Matocha [CM98].

Non-distributed collection mechanisms are not discussed except when necessary in describing their distributed counter-parts. A comprehensive review of non-distributed garbage collection schemes is given by Wilson [Wil92].

2.1 A Review of Distributed Garbage Collectors

2.1.1 Distributed Reference Counting

Reference counting appears attractive for a distributed system since it is inherently incremental. In a distributed context, the reference count for an object x represents a local view of the number of references to x in the distributed system. This view may be out-of-date due to asynchrony but the view is always safe (typically by being conservative). Any correct distributed reference counting collector must ensure that the reference count for x is non-zero while a reference to x exists. The difficulty in achieving this is due to a site's incomplete view of the state of the distributed computation.

A naïve implementation of a distributed reference counting collector is to simply send *increment* and *decrement* messages to the site holding an object each time a reference to that object is copied or deleted. However such a scheme may incorrectly

determine that an object is garbage due to the race condition that exists for the reference count. This is illustrated in Figure 2.1 below. The diagram shows a representation of a time-line for three sites and the actions of those sites on references to an object x held at site B . The *decrement* message for C 's deletion of its reference to x arrives at B before the *increment* message from A . Thus at point 1 on B 's time-line the reference count for x at B may incorrectly reach zero. The reference count for x remains correct only if the *decrement* message from C arrives after point 2 on B 's time-line.

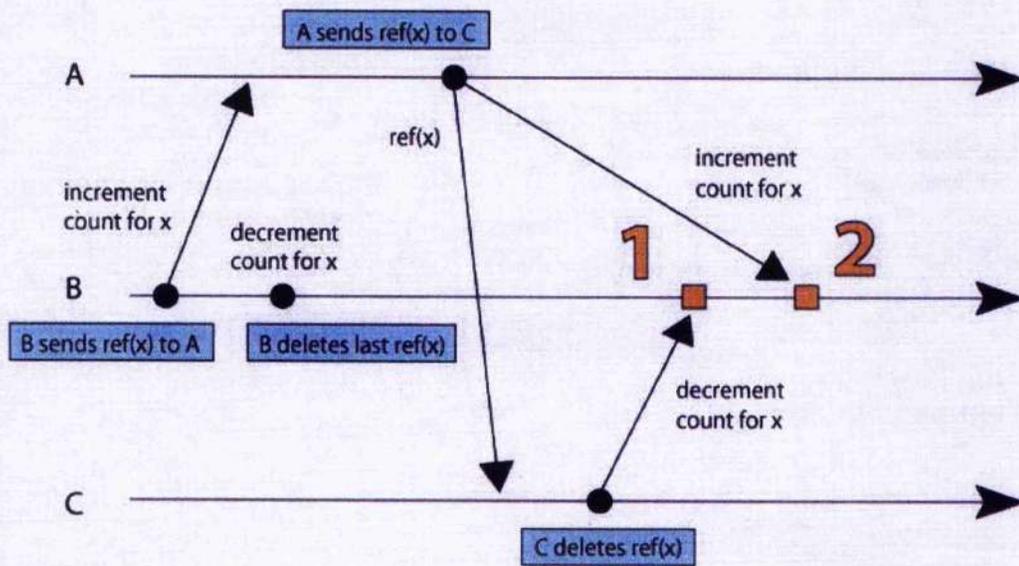


Figure 2.1 - Distributed Reference Counting Race Condition

2.1.1.1 A Reference Counting Protocol

Lermen and Maurer [LM86] describe a distributed reference counting scheme that avoids the message delivery ordering shown in Figure 2.1. When a reference to an object x is sent from site A to site B , A sends an *acknowledge request* message (indicating a reference to x sent to B) to the site holding x . On receiving an *acknowledge request* message the site of x increments the reference count for x and sends an *acknowledge* message to site B . Thus the site B will receive both the

message containing the reference to x and an acknowledgement message from x 's site.

When site B deletes a reference to x , the deletion is communicated to x 's site with a *delete* message. However a *delete* message may only be sent from B to x 's site if the number of *acknowledgement* messages received at B (for x) is equal to or greater than the number of copies of x received at B . The reference count for x is decremented on receipt of a decrement message at x 's site. If the reference count for x becomes zero then the object may be reclaimed.

Thus, at the cost of three messages per reference copy the reference count is never incorrectly determined to be zero.

2.1.1.2 Weighted Reference Counting

The message overhead of Lermen and Maurer's protocol is avoided in Watson and Watson's [WW87] weighted reference counting collector. Weighted reference counting associates an integer *reference count* with each object and an integer *weight* value with each reference to the object. Note that the *reference count* value for an object x does not represent the number of references to x but it serves as an analogue of a traditional reference count. When the *reference count* value for x reaches zero, no reference to x exists.

When an object is created, its *reference count* is set to some non-zero integer value and the *weight* value of the first reference to that object is made equal to the *reference count* value. The collector operates on the premise that the following invariant is maintained:

The sum of the *weight* values for each reference to an object x is equal to the *reference count* for x .

When a reference is copied its *weight* is divided between the original reference and the copy. On the deletion of a reference to an object x , a message containing the weight of the deleted reference is sent to the site holding x . The receiving site then subtracts this weight from the *reference count* for x . If the *reference count* for x reaches zero then x may be reclaimed.

By ensuring that each weight value is always a power of two (allowing equal division of each weight) a logarithmic encoding may be used for weights. For a weight w , the value $\log_2 w$ is stored thus reducing the space over-head incurred by storing a weight value for each reference.

The obvious problem with this scheme is that a reference with a *weight* of one cannot be copied. The solution to this problem is to introduce an *indirection object* to act as a proxy for the original object. This works as follows:

- An object A contains a reference to an object x with a weight of one. This reference to x is to be copied to an object B .
- An indirection object C is created with a non-zero reference count value.
- The reference to x in A is moved to C and in its place is left a reference to C with a weight value that is equal to the reference count value for C .
- The reference to C in A is then copied to B as normal.

Any access to the *indirection object* C must now be redirected to the original object x and reference values cannot be used to determine object identity. This is clearly not ideal in a distributed system.

Weighted reference counting was proposed by both Watson and Watson in [WW87] and by Bevan in [Bev87]. Watson and Watson attribute the initial implementation of

the algorithm to [Wen80] while Bevan attributes the idea for *indirection objects* to Simon Peyton Jones.

2.1.1.3 Indirect Reference Counting

Piquer's [Piq91] indirect reference counting algorithm represents an alternative method to avoiding the message overhead of Lermen and Maurer's protocol. Indirect reference counting maintains a tree structure for each object x which represents the transmission of references from remote sites to x (remote references) through the distributed system. This tree is the equivalent of Dijkstra and Scholten's [DS80] diffusion tree for termination detection, although here it is used to detect the absence of remote references to an object.

Each site that holds a reference to x corresponds to a node of the tree and the node for a particular site is held at that site. A node contains two fields; one field holds a reference to the node's *parent* and the other holds a count of the node's *children*. The *children* count for the node at a site A records the number of sites to which A has sent a reference to x . The root of the tree is the object itself and therefore a *children* count value is associated with each object. The *children* value for an object x is initialised with value one when the first remote reference to x is exported to a remote site. Each time a site sends a reference to x to a remote site the children count for x at the sending site is incremented. Figure 2.2 below illustrates the tree for an object which is referenced from five remote sites.

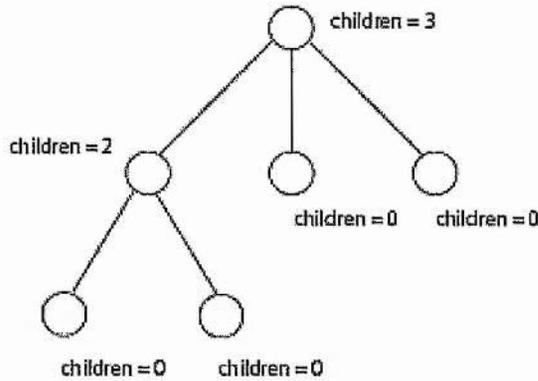


Figure 2.2 – A Diffusion Tree for Indirect Reference Counting

If the *children* value for x reaches zero, the object is no longer referenced from any remote site.

When a site A no longer holds any references to x and the *children* count at A is zero the node at A may be removed from the tree. On deletion of the node at A , a *decrement* message is sent to the site that is A 's parent node. On receipt of a *decrement* message a site reduces its *children* count by one. Only sites with a zero *children* count can be removed from the tree. Thus a site that holds no references to x must wait for its *children* count to reach zero before sending a *decrement* message to its parent.

Cycles are not allowed in the tree structures. Thus if a site A receives a reference to x from a site B and A already has a node of the tree for x then a decrement message is sent back to B immediately. The collector effectively imposes a tree structure, for each object that is referenced from a remote site, over the sites holding the distributed object graph. Sites are added to the tree for x when they receive their first reference to x and a site is only removed from the tree once all of that site's children have been removed.

2.1.1.4 Generational Reference Counting

Goldberg's [Gol89] generational reference counting collector associates a *generation* identifier and a *copy* count with each reference. The first reference to an object has generation zero and any copy of this reference has generation 1. In general a copy of a reference of generation G_i is of generation G_{i+1} .

Each object has an associated *ledger* structure which records the number of outstanding references from each *generation*. The *ledger* contains a reference count for each generation that is known to exist.

When a reference to an object x is copied locally or sent to a remote site, the reference's *copy* count is incremented. When a reference to x is deleted on a remote site A a *delete* message containing the *generation* identifier and the *copy* count for the reference is sent from A to the site holding x .

On receipt of a *delete* message containing generation G_i and count C for object x the ledger for x is modified as follows:

- $ledger_x[G_i] = ledger_x[G_i] - 1$
- $ledger_x[G_{i+1}] = ledger_x[G_i + 1] + C$

When the *ledger* for x contains a reference count of zero for all generations then x may be reclaimed.

2.1.1.5 Reference Listing

Birrell et al. [BEN+93] describe an incremental technique for distributed garbage identification called reference listing. Reference listing, like reference counting, is incomplete and only allows the reclamation of acyclic garbage structures. A reference listing collector is implemented in the Network Objects system from [BNO+93].

Reference listing differs from reference counting in that a site holding an object x maintains a list of the remote sites that hold a reference to x rather than a count of the total number of remote reference to x . When the reference list for x is empty (indicating that no site holds a reference to x) the object may be reclaimed.

2.1.2 Centralised Control of Distributed Collection

2.1.2.1 A Centralised Distributed Garbage Collection Service

Liskov and Ladin [LL86] describe a distributed collector which is logically centralised but physically replicated in order to make it highly available.

Each site implements an independent local collector and local collectors do not communicate with each other. Instead information about inter-site references is recorded by a centralised service. The local collectors communicate with the service to report references to remote objects and to discover those local objects that are accessible from a remote site. Each site maintains the following information:

- *inlist* - a list that records each local object x where a reference to x has been sent to a remote site.
- *trans* - a list of references that have been sent in messages to remote sites.

The local collector uses the *inlist* and the local roots as roots of reachability for local collection. During collection the local collector constructs three sets of data which are then sent to the service:

- *acc* - The set of all remote objects reachable through a path of references from the local roots at the site.
- *paths* - The set of tuples $\langle x, y \rangle$ where x is an object in the local *inlist* and y reachable from x . Where y is reachable from a local root $\langle x, y \rangle$ is not in *paths*.

- *qlist* - The set of all local objects which are in the *inlist* and that are not reachable from a local root.

The service provides a *query* interface which allows a local collector to ask which of the objects in its *qlist* are accessible from other sites. When the result of a query is returned to a site, any *inlist* entries for objects not reachable from a remote site are removed.

The service is made up of a number of replica sites and the local collector at a site communicates with only one replica site. Replicas communicate the data received from local collectors between each other through the exchange of *gossip* messages. Thus the result of a query is calculated by a replica site with global information which is possibly out-of-date. The service returns a result that allows for safe but conservative collection at a site. Each replica periodically runs a cycle detection algorithm over its global view of the object graph to identify inter-site cycles of garbage. This algorithm consists of a local mark-sweep algorithm which operates over the replica's local information about the object graph. When a cycle is detected information about the cycle's component objects is communicated to each of the other replicas through further gossip messages.

Computation of the *acc* and *paths* sets during local collection is potentially computationally very expensive in this scheme. For instance, it is not enough to run a standard marking algorithm at a site since every local path to an object must be discovered. Abdullahi and Ringwood [AR98] cite Rudalics' [Rud90] counter example in explaining why this is the case. This is shown in Figure 2.3 below. Object *x* is accessible via two paths but if the local marker traverses *y* before *w* then only the path from *z* to *y* is sent to the service and not the path from *w* to *z*. Thus *z* and *y* will incorrectly be identified as garbage. Object *x* at site *B* is accessible via

multiple paths and if any of these is omitted the object may be incorrectly identified as garbage.

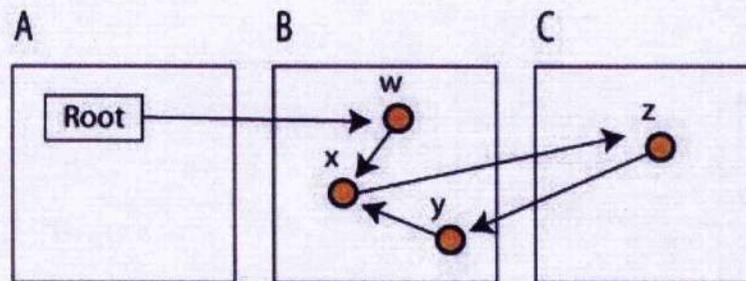


Figure 2.3 - Rudalic's Counter Example

This problem is addressed by Ladin and Liskov [LL92] where a time-stamping algorithm is used, similar to that described by Hughes in [Hug85].

2.1.3 Distributed Mark-Sweep

2.1.3.1 Distributed Concurrent Mark-Sweep

The concurrent mark-sweep collector described by Dijkstra et al. [DLM+78] is the basis for distributed mark-sweep collectors from Hudak and Keller [HK82], Augusteijn [Aug87] and Derbyshire [Der90].

Concurrent mark-sweep is known as *on-the-fly* collection because mutator and collector activity may proceed concurrently. In non-concurrent mark-sweep mutator activity must be suspended while marking is in progress to avoid interference of the mutator and the collector. For instance if a new object x is allocated and a reference to x written into an already marked object, then during the sweep phase x will be reclaimed because it is not marked. Concurrent mark-sweep avoids this problem through the use of a tri-colour marking abstraction. Objects are coloured black, grey or white. Black objects are live, grey objects are potentially live and white objects are unreachable. The interference described above may be restated in terms of this

marking. The problem is caused because a black (marked) object references a white (unmarked) object. Concurrent mark-sweep avoids this problem by ensuring the collector and mutator maintain the invariant that:

No black object contains a reference to a white object.

When an object is allocated it is coloured black. A write barrier ensures that the mutator does not create any black to white references by colouring any white object grey if a reference to that object is written to a black object. It is useful to imagine that as an object is coloured grey a reference to that object is added to a queue.

Note that there may be multiple mutator processes operating over the object graph but there is only one collector process. Within the collector process the mark (garbage identification) and sweep (garbage reclamation) phases are executed in strict serial order.

It is assumed that the operation to test or set an object's colour is atomic. Collection proceeds as follows:

- The mark phase begins by first colouring each root object grey.
- While the queue of grey objects is not empty the collector removes a reference, to an object x say, from the grey queue. The object x is coloured black and each object referenced by x is coloured grey if it is white.
- When there are no more grey objects in the queue the storage space is swept sequentially. Any object which is white is reclaimed and all other objects are coloured white.

Thus every object is guaranteed to survive at least the first collection following its allocation. Marking is guaranteed to complete since eventually there are no grey objects left. At this point the mutator cannot cause the creation of any more grey

objects since all of the white objects that exist are unreachable by the mutator and black objects cannot be coloured grey.

Abdullahi and Ringwood [AR98] describe the centralised concurrent mark-sweep collector as incomplete since a single invocation of the collector reclaims only a subset of the garbage objects that existed before collection began. However this thesis takes a different view of completeness and chooses to define a collector that eventually reclaims a garbage object as complete. This is a fair definition considering that the thesis concentrates on distributed collection mechanisms. Therefore while concurrent mark-sweep is certainly conservative it is still complete. A newly created object is guaranteed to survive the collection cycle in which it was allocated but any garbage objects that survive one collection cycle are collected at the next.

A distributed version of the concurrent mark-sweep collector must address two issues:

1. Propagation of object marking from one site of the distributed object graph to another.
2. Detecting completion of the mark phase.

The first of these problems is solved by simply sending a message from one site to the other, when an inter-site reference is discovered, indicating that marking should proceed from the referenced object (effectively greying the referenced object in the remote site). The solution to the second problem is addressed differently in each of the collectors described below. However both the collector from Hudak and Keller and the collector from Augusteijn use a tree structure (similar to that used for indirect reference counting) to detect termination of the mark phase. Effectively both

collectors make use of Dijkstra and Scholten's diffusion tree distributed termination algorithm, although neither of the collectors states this explicitly.

In the centralised concurrent mark-sweep collector the queue of grey objects (which provides a place-holder for objects that have still to be marked) is shared between the collector and the mutator processes. Hudak and Keller point out that in distributing the collector this structure becomes shared between multiple sites. Thus sharing is more complex and determining when the structure is empty requires the evaluation of a global predicate. Instead of a queue, their collector constructs a tree structure called a *marking tree* which acts as a place-holder for the distributed marking and is used to detect completion of the distributed mark phase.

Hudak and Keller's system model states communication between sites occurs by spawning tasks from one site to another. The collector assumes that a single distinguished root object exists for the distributed object graph. This object is designated as the root of the marking tree. Each node of the tree corresponds to an object that is being marked and records an identifier for node's parent and the number of children of the node.

An object is marked with a *mark task*. A *mark task* for an object x (which is white) creates a node of the tree containing its parent identifier and then colours x grey and spawns a *mark task* for each object referenced by x . For each *mark task* spawned, x 's children count is incremented. Thus the marking is distributed when a *mark task* spawns a *mark task* for a child on at a remote site. A *mark task* completes when the children count for its node of the tree becomes zero. On completion of a *mark task* the object is coloured black and an *up-tree* task is spawned for the node's parent. An *up-tree task* decrements the parent's children count.

A *mark task* that is spawned for an already black object causes an *up-tree task* to be spawned for the parent immediately. Thus the tree collapses back to the root as marking progresses towards completion. When the root object's children count reaches zero marking is complete and all reachable objects are black.

Augusteijn's collector is similar to that of Hudak and Keller. However Augusteijn defines a single synchroniser site which is responsible for detecting completion of the mark phase. The aim of the mark phase is to establish a global state such that there are no grey objects. Each site has a set of root objects and when collection starts these objects are grey. Each site establishes a tree structure which spans across the sites of the object graph reachable from its roots. A site can only be a member of a single spanning tree at any time. If an inter-site reference to an object x on site B is discovered during marking at site A a request message is sent to the site holding the target object and A 's children count is incremented. The object x is coloured grey if it is not black on receipt of the *request* message. If B is not already the child of another site then it becomes a child of A . However if B is a child already a *continue* message is sent to A . On receipt of the *continue* message, A decrements its children count. If a site holds no grey objects, has a children count of zero and is not the child of any site, a *done* message is sent to the *synchroniser*. If a site holds no grey objects, has a children count of zero and is the child of a site A then a *continue* message is sent to A .

Thus each site will send exactly one *done* message to the *synchroniser*, and send one *continue* messages to each parent that it acquires. When the *synchroniser* has received a *done* message from each site, there are no more grey objects globally and marking is complete.

Effectively each site establishes a diffusion tree in marking the sub-graph of the global distributed object graph reachable from the roots at that site. A centralised site is charged with determining when the marking of each of the sub-graphs is complete. Note that the sub-graphs may overlap and in this case the responsibility of the marking of the overlapping portion of the graph is handed-off to the diffusion tree already resident at the site where the overlap begins.

2.1.3.2 Distributed Mark-Sweep with Time-Stamps

Hughes' [Hug85] distributed mark-sweep collector carries out multiple distributed collections in parallel. An independent local mark-sweep collector at each site contributes to all of the currently active global collections every time it performs a local garbage collection.

Each global collection has an associated time-stamp value which is used to mark objects reached during that collection. Mutator activity is suspended while local collection is in progress. The local collector at a site propagates the time stamps of root objects at that site to the objects at that site which hold remote references. After local collection, for each object x which holds a reference to a remote object y where the time-stamp for x was increased during the last collection, the time stamp of x is sent to the site holding y . The time-stamps for remotely referenced objects are thus increased following a local collection.

Each site keeps track of the earliest global collection for which it has more work to do. When no site has work to do for the global collection with time-stamp T then any object with a time-stamp less than T is garbage and may be reclaimed.

There are two key assumptions in Hughes' collector:

- Instantaneous communication - thus no references are ever in-flight between sites and time-stamps are transmitted instantly from one site to another following local collection.
- A globally synchronised clock.

Rana's distributed termination algorithm (from [Ran83]) is used to detect the completion of each of the global mark-phases.

2.1.4 Distributed Copying Collection

Rudalics [Rud86] describes a distributed copying collector. The collector assumes that the distributed computation operates over a distributed graph and that one site of the graph holds the distinguished root object for the graph. Inter-site references are implemented as follows. A reference from an object x at site A to an object y at site B is represented by a local reference in x to a remote reference object at A containing the site identifier for B and the address (at B) of a root object for y . The root object for y at site B holds a local reference to y .

The storage space at each site is divided into three spaces; a root space and semi-spaces called *fromSpace* and *toSpace*. The root space holds the root objects for a site. The semi-spaces are used by the collector for moving (and compacting) local objects at a site. The lower portion of each semi-space is used to store objects, which contain local references. The upper portion of each semi-space is used to hold remote reference objects for inter-site references.

Each site implements a logical semi-space arrangement for root objects by holding them in one of two linked lists; *oldRoots* and *newRoots*. Thus a root may be 'moved' from one logical semi-space to the other without being copied to a new address.

Collection starts at the global root and initially all root objects are in *oldRoots*. Collection begins with a global *SCAN* phase by marking the global root and setting a state variable at the root's site to *SCAN*. Any root object which is marked and held in *oldRoots* is moved to *newRoots*. If the object *x* referenced by the root is in *fromSpace* then *x* is copied to the lower end of *toSpace* and a forwarding pointer is left in its place. The local graph is then traced from the object *x* and any local object is copied to the lower portion of *toSpace* while remote reference objects are copied to the upper portion of *toSpace*. For each remote reference object in *toSpace* a request message is sent to the referenced site and a local message count value is incremented. On receipt of a request message at a site where the state variable is not set to *SCAN*, the state is set to *SCAN* and a *parent* value is set to the site identifier for the sending site. The referenced root object is then marked and collection proceeds as described above.

On receipt of a *request* message at a site where the state value is set to *SCAN* a completion message is immediately sent to the sending site. On receipt of a *completion* message a site decrements its *message count* value. When a site's *message count* value reaches zero, and there are no marked roots in *oldRoots* and no local objects in *fromSpace* which are still to be copied, a completion message is sent to the site's parent. When this state occurs at the site holding the global root then the *SCAN* phase is complete.

Collection continues with a *FLIP* phase whereby the root site broadcasts a *flip* signal to each site. On receiving the *flip* signal a site reclaims any root object in *oldRoots*, makes the *oldRoots* list the new *newRoots* list (and visa versa), unmarks all roots in *oldRoots* and inter-changes the semi-spaces. When a site has done this, an *acknowledgement* message is sent to the root site.

2.1.5 Hybrid Distributed Collectors

Hybrid collectors typically consist of an incomplete collection mechanism (such as reference counting) for the collection of acyclic structures and a complete collection mechanism (such as mark-sweep) for the collection of cyclic garbage structures. The principle behind such collectors is that acyclic structures are more prevalent than cyclic structures. Thus, an inexpensive (in terms of collection overhead) but incomplete collector is used frequently to reclaim acyclic garbage and a more expensive complete collector is invoked less often to reclaim cyclic structures.

An example of such a scheme for a uni-processor system is the hybrid cyclic reference counting mark-sweep collector described by Martinez et al. in [MWL90]. Each time a reference count is decremented for an object x , where the result is non-zero, a local mark-sweep is executed on the sub-graph reachable from x . As the sub-graph is traversed during marking, the reference count for each object that is encountered is decremented and the object is marked as garbage. Next a scan of each object encountered during the mark phase is carried out. Each object with a non-zero reference count is unmarked as garbage and its count is reset. Each object that has a zero reference count is reclaimed.

Lins [Lin92] describes an optimised version of the cyclic reference counting collector that allows the mark-sweep phase for cycle candidates to be delayed. Candidate objects are added to a *control queue* and the mark-sweep for these objects is delayed until such time as the queue is full or the storage space is exhausted. Objects on the queue are coloured black to indicate that they are candidates for cyclic garbage. If the reference count for a candidate is increased after the object is added to the control queue its colour is changed. When an object is removed from

the queue a mark-sweep is executed only if the object is black. Otherwise the next object is removed from the queue.

Lins and Jones [LJ91] describe a distributed version of Lins' lazy cyclic reference counting algorithm. This distributed collector is called cyclic weighted reference counting. In this collector distributed reference counting is achieved through an implementation of the weighted reference counting algorithm. The mark-sweep mechanism for cyclic garbage candidates is similar to Lins' [Lin92] mechanism. However the distributed version is centralised and synchronous. The authors do not give enough detail as to how termination of the mark phase is to be detected for an implementation to be imagined.

Rodrigues and Jones in [RJ96] describe a hybrid distributed collector similar to cyclic weighted reference counting. However this collector uses the reference listing mechanism from [BEN+93] for the collection of acyclic garbage and the distributed mark-sweep collector from [Der90] to identify those objects that are part of cyclic garbage structures from the set of candidates.

2.1.5.1 Distributed Back-Tracing

The collectors from Maeda et al. [MKI+95] and Fuchs [Fuc95] use mechanisms that identify objects which are suspected of being part of cyclic garbage structures. The sub-graph reachable from each suspect is then traced to see if it reaches a root. Fuchs' collector traces references within the sub-graph in reverse to calculate whether or not the cycle is reachable from any root.

The distributed back-tracing collector described by Maheshwari and Liskov in [ML97] is similar in style to the Lins and Jones weighted reference counting collector. However the back tracing collector provides independent local mark-

sweep collection at each site. A reference listing mechanism is used to identify the objects at a site that are referenced from a remote site. Any object referenced from a remote site is treated as a root for local collection.

The reference listing mechanism maintains two sets of references at each site:

- *inrefs* - the set of references to local objects which are referenced from a remote site. Elements of this set are logically tuples which identify the referenced object and a list of referencing sites (called the *source list*).
- *outrefs* - the set of references to remote objects which are referenced from this site.

If site *A* sends a reference to an object *x* at site *B* to site *C* then on receipt of the reference to *x*, *C* adds *x* to its *outrefs* if *x* is not already in *outrefs*. *C* then sends an *insert* message for *x* to site *B*. On receipt of the *insert* message for *x*, *B* adds an entry to its *inrefs* indicating a reference to *x* from *C*. To preserve safety a site *A* maintains its *outrefs* entry for *x* until *B* has received the *insert* message for *x* from *C*.

Following a local garbage collection a site brings its *outrefs* set up-to-date. For each *outrefs* entry that is removed an *update* message is sent to the site holding the referenced object. On receipt of an *update* message at a site *A* from a site *B* for an object *x*, *A* removes *B* from the *source list* in the *inrefs* entry for *x*.

The *inrefs* entry for an object *x* is removed on removal of the last site from the source list entry. An object *x* whose *inrefs* entry is removed will be reclaimed during the next local collection. Reference listing does not allow for the reclamation of inter-site cyclic garbage structures and thus a secondary collection mechanism is required. This mechanism is known as back-tracing.

The distance heuristic from [ML95] is used to identify objects which are suspected as being part of a distributed cycle of garbage. The heuristic is complete and ensures that every garbage object is eventually a suspect.

Back-tracing is based on the principle that if an object is reachable from some root then if each reference is reversed the root is reachable from the object. The key distinction between global marking and back tracing is locality. That is, if a cyclic garbage structure spans only three sites then only those three sites are involved in tracing the cycle.

In back-tracing from a suspect object the algorithm jumps between *outrefs* and *inrefs* as opposed to tracing the reverse of each reference in the cycle. The *inrefs* set already contains enough information to allow the back tracing from an *inrefs* entry to the corresponding remote *outrefs* entry. However in order to jump from an *outrefs* entry to the corresponding local *inrefs* entry from which it is reachable requires that a site record each *inrefs* entry from which a particular *outrefs* entry is reachable. In back-tracing from a suspect object the collector alternately jumps from an *inrefs* entry to an *outrefs* entry and an *outrefs* entry to an *inrefs* entry.

The back-trace for a suspect object returns the value *live* if the trace encounters an *inrefs* entry or an *outrefs* entry for an object that is not suspected of being garbage. This prevents a trace from traversing those parts of the object graph that are known to be live. Since every garbage object is guaranteed to become a suspect every garbage cycle is eventually reclaimed.

If a back-trace indicates that a garbage cycle has been detected then the site that initiated the trace informs each site that was visited. The *inrefs* entries for objects that are part of the garbage cycle are identified as *garbage* and are not used as roots for local garbage collection.

2.1.6 Garbage Collecting the World

Lang et al. [LQP92] describe a hybrid distributed collector that uses a reference counting scheme such as weighted reference counting [WW87] or indirect reference counting [Piq91] to reclaim acyclic garbage and a tracing collector to reclaim cyclic garbage structures. However the tracing collector operates within dynamically formed groups of sites to reclaim inter-site cycles wholly contained within those sites. The aim is to increase the locality of distributed collection (as in the back-tracing collector from [ML97]) and involve only those sites holding a distributed cycle in its collection.

The collector assumes that an inter-site reference is represented by an entry item/exit item pair. That is, a reference from an object x at a site A to an object y at a site B is represented by a reference in x to a local exit item (at A) where the exit item holds a reference to an entry item at B which in turn holds a local reference to y . Each object has only one entry item and each site maintains only one exit item for each remotely referenced object. Each entry item contains a reference count which is maintained by the distributed reference counting scheme. When a site no longer references a particular remote object the exit item is reclaimed and a decrement message (or equivalent) is sent to the remote site. An exit item is reclaimed when its reference count is zero. A local collector reclaims unreachable objects within a site.

The first phase of a distributed collection begins with *group negotiation* whereby the group of sites that will take part in the collection is formed. The initial marking for the collection of a group is provided by the reference counting collector. Objects are initially marked as *hard* (reachable from outside the group or from a root) or *soft* (reachable only from another site of the group). The local collectors at each site propagate the mark values for entry items of the group towards exit items. These

marks are then passed to the corresponding entry items if they are within the group. The local collectors are then responsible for once more propagating the mark values within sites. This process is repeated until *group stability* is reached. That is:

- No new exit item has a mark value that has not already been sent to its corresponding entry item.
- There are no messages in transit.

The tracing of the group is now complete. Any object in the group reachable from a root or from an object outside the group is marked hard. Any object marked soft is reachable only from another object in the group and is thus part of a garbage cycle contained within the group. The group is now disbanded.

A hierarchical structure can be imposed on the groups such that increasingly large groups are traced. In order for the collector to be complete the entire space must eventually be traced.

2.1.6.1 Partitioned Collection

Maheshwari and Liskov [ML97a] present a collector for large partitioned object stores. This collector is not distributed but the partitioning of the store does bear a strong relation to the state partitioning that occurs in a distributed environment. The collector concentrates on providing independent collection within partitions and this is a theme that runs strongly in Chapter 5.

To collect partitions independently the system maintains an *inlist* data structure that records the objects in a partition that are referenced from another partition. Information about the outgoing references from a partition is recorded in an *outlist*. A partition is traced from each entry in its *inlist* and during tracing the *outlist* is updated. The collector defines a third structure called a *translist* which allows

information in *inlists* and *outlist* to be shared efficiently. Thus as the *outlist* for a partition is updated during tracing, the necessary *inlists* are brought up-to-date.

Inter-partition cycles are identified by a global incremental marking phase. Global collection begins by marking the persistent roots of the store. When a partition P is traced (during the collection of P) the marks (for the global trace) are propagated from the roots of P to the entries in P's *outlist*. Thus global tracing is piggy-backed on local tracing within a partition.

2.1.7 The DMOS Collector

The DMOS collector (initially described in [HMM+97]) derives from both the MOS (Mature Object Space) collector [HM92], sometimes known as the Train Algorithm, and the PMOS (Persistent Mature Object Space) collector [MBM+99, MMH96]. MOS is a main memory collector designed to do a limited amount of work each time it is invoked (so it is non-disruptive) and to guarantee that each unreachable data object is collected eventually (and so it is complete). PMOS extends MOS to provide incrementality in a persistent context, while also limiting I/O overhead. DMOS builds upon MOS and PMOS to offer incremental collection for distributed systems.

Objects in DMOS are partitioned by cars within sites and cars are grouped together into trains that span multiple sites. The collector is composed of two interacting collection mechanisms. The first is a car collection mechanism which collects object that are unreachable from outside their car and re-associates reachable objects to other cars and trains in accordance with a set of re-association rules. Car collection is incomplete as it cannot reclaim a cycle of garbage objects that spans multiple cars. A second collection mechanism reclaims entire trains once they become isolated. A train is isolated when each object in the train is referenced only from other objects in

the train. The train collection is complete due to the nature of the re-association rules that govern the collection of individual cars.

The contribution of DMOS is its unique combination of desirable properties for a distributed collector. Specifically, DMOS is:

1. **Safe:** it does not collect live (reachable) objects.
2. **Complete:** it reclaims all garbage, including cyclic garbage that spans sites, within a finite number of invocations.
3. **Non-disruptive:** it bounds the amount of collection work, thereby bounding the time and space requirements, for each invocation.
4. **Incremental:** it reclaims space incrementally without global knowledge of reachability.
5. **Local:** it initiates local collections at each site independently of other sites.
6. **Independent:** it is independent of the specific local collection algorithm employed at each site, though it imposes some requirements on the local collectors.
7. **Decentralised:** it uses no algorithms that rely on a single central site for processing or global synchronisation.
8. **Asynchronous:** it communicates via asynchronous messages, and the collector at a site need only synchronise with another site in one particular case; application computation never need wait for such synchronisation.

DMOS therefore has all of the properties that are prerequisites of scalability; incrementality, locality, decentralisation and asynchrony. The significance of DMOS is that these properties are achieved in combination; in particular, it is difficult simultaneously to realise completeness, incrementality, and decentralisation.

DMOS thus exhibits a unique combination of the properties that are desired of a distributed garbage collector. However, before the work described in this thesis, no satisfactory implementation of DMOS has been produced and as such the properties of the DMOS collector cannot be verified. The nature of interaction of the distributed train collection and local car collection mechanisms has not previously been investigated.

2.2 Distributed Termination Detection

The distributed termination problem was proposed independently by Dijkstra and Scholten in [DS80] and by Francez in [Fra80]. The problem is that of determining when a distributed computation executed by a network of processes has terminated.

Tel [Tel94] states the problem as detecting *terminal configurations* of a distributed computation consisting of a number of distributed processes. He distinguishes between processes which have completed (these are said to be in a *terminal state*) and those processes which can only receive messages from other processes. A computation is also terminated when each site is in this second state and where there are no messages in flight. However in this case each individual process still considers the global computation to be in progress. In this case termination is said to be *implicit*. Termination of a computation is said to be *explicit* if each process is in a *terminal state* when the computation is in a *terminal configuration*.

[Mat89] and [CM86] generalise the distributed termination problem as global quiescence detection. This encompasses a number of problems including the detection of computation termination, distributed deadlock and the end of a phase in a distributed multi-phase algorithm.

Termination detection can also be viewed as providing global predicate evaluation within distributed systems. Predicates that can be evaluated by a DTA are those based on deciding whether or not a particular globally stable state exists for some subset of the shared state within a distributed computation. This is achieved by modelling the subset of the distributed state and the operations over that state as a set of processes in such a way that termination corresponds to a globally stable state. This is explained further in Chapter 5.

Chandy and Lamport [CL85] describe a solution for distributed termination detection based on the construction of distributed snapshots. The state of the distributed system consists of the combination of state of each process and the state of each of the communication channels. The distributed snapshot approach is to determine the global state and whether or not termination holds. The global state approach is adopted by Chandy and Misra [CM86] and by Misra [Mis83].

2.2.1 A Model for Distributed Termination

Camp and Matocha [CM98] define the following system model for distributed termination detection:

- A distributed computation consists of a number of processes that are distributed across a network.
- Processes communicate with each other by passing messages through bi-directional communications channels.
- There is no shared memory.
- There is no global clock.
- Communication is asynchronous and is subject to unbounded latency.

The distributed computation is known as the *basic computation*. Messages sent between processes as part of the basic computation are known as *basic messages*. A process is either *active*, where it is executing part of the distributed computation, or is *passive*, where the process is waiting on a message or has terminated. Each process runs the distributed termination algorithm (DTA). Messages that are passed between sites as part of the DTA are known as *control messages*.

Processes are constrained to behave as follows:

- Each process is initially *active* or *passive*.
- An *active* process may become *passive* spontaneously.
- Only *active* processes can send *basic messages*.
- A process can only change from the *passive* to the *active* state on receipt of a *basic message*.

With this model it is easy to see that detecting that each site is *passive* is not sufficient in order to determine termination of the distributed computation. The computation is terminated only when all sites are *passive* and there are no messages in-flight.

DTA's are typically classified by their type. Broadly speaking there are two types of DTA; wave algorithms and parental responsibility algorithms.

2.2.2 Wave Based Algorithms

A wave algorithm defines a repeatable decision making computation. A wave is started by a process defined as the *initiator* and visits each other process in the distributed system. When the wave returns to the *initiator*, that process is in a position to determine whether or not the computation has terminated. If the computation has not terminated, another wave is started.

2.2.2.1 A Token Ring

A subset of the wave algorithms are ring algorithms. In a ring algorithm a ring topology (typically a Hamiltonian cycle) is imposed on the underlying communications network. *Control messages* are passed only via the ring, while *basic messages* can be sent in any way allowed by the underlying network topology.

Dijkstra, Feijen and van Gasteren [DFG83] describe a ring based DTA. The algorithm assumes the existence of a Hamiltonian cycle which contains all $n+1$ processes P_0 to P_n and that message passing is instantaneous. The wave consists of a token being passed around the ring from the *initiator* process P_0 visiting each process and returning to P_0 . When the token arrives back at P_0 the *initiator* process decides if the computation has terminated. A colour value is carried by the token and held by each process. The token and each process may be either black or white. The token and each process is initially coloured white.

The intuition behind the colouring is as follows:

- A white node is passive and has sent no basic messages since it last held the token.
- A black site is passive but has sent a basic message since it last held the token.
- The token colour carries the accumulated state of the wave. This state consists of a single bit of information identifying whether or not a black site was encountered. If not back site is encountered then a white token will eventually return to the *initiator* site and the terminated state will be detected. However, if a black token is received, a black site was encountered and the result of the wave is to indicate that the computation has not terminated.

While a process P_i is active P_i will hold on to the token. When P_i becomes passive it passes the token to its successor. Thus the wave begins when P_0 becomes passive at which point the token is passed to P_1 . A white process passes the token without changing the token's colour. A process becomes black if it sends a basic message to any other process. A black process changes the token's colour to black before passing it to the next process. A site becomes white on passing the token to its successor. The computation has terminated when the *initiator* receives a white token and the initiator is itself white and passive.

Safra [Dij87] presents a modification to Dijkstra, Feijen and van Gasteren's token ring algorithm which relaxes the restriction on instantaneous message passing. In Safra's scheme each process P_i maintains a message count mP_i which is initially zero. When a process P_i sends a *basic message*, mP_i is incremented. When a *basic message* is received by P_i , mP_i is decremented. The token also carries a message count mT . When a process P_i forwards the token mP_i is added to mT and then mP_i is set to zero. The computation is terminated when the *initiator* receives a white token, the initiator is itself white and passive and $mP_0 + mT = 0$.

Mattern's [Mat87] variation of the Dijkstra, Feijen and van Gasteren algorithm also allows for asynchronous message passing. However in Mattern's scheme each process maintains an array of message counts with one element for each process to which a message has been sent and for each process from which a message has been received. The token also carries an array of count values with one element for each site that has sent or received a basic message. When a process P_i passes on the token the values in the array for P_i are added to the corresponding array values in the token. Each element in the array at P_i is then set to zero. The termination condition

now holds when the *initiator* receives the token and each element in the token's array is zero.

2.2.2.2 Termination Detection with Time-Stamps

Rana's solution [Ran83] to the distributed termination problem allows any process to test for termination. This differs from the algorithms above where only a pre-designated process may initiate detection. Rana's algorithm assumes that processes are connected in a Hamiltonian cycle and assumes the existence of a global clock. Communication between processes is synchronous.

The distributed system consists of n processes and termination detection is achieved through passing a *control message* that contains a time stamp tCM and counter C . When a process P_i becomes passive it records the current time tP_i and sends a *control message* containing $tCM=tP_i$ and $C=1$ to its successor. If an active process receives a control message the message is discarded. If a passive process P_j receives a control message, P_j compares the time in the control message tCM with tP_j . If $tP_j > tCM$ then the message is discarded otherwise C is incremented and passed to P_j 's successor. If a passive process P_i receives a control message where $C=n$ then the computation has terminated.

2.2.2.3 Tree-Based Waves

Francez [Fra80] describes a wave algorithm based on a spanning tree which is constructed over each of the processes in the distributed system. The spanning tree is constructed from the underlying process network without the addition of extra communications channels.

Assume that for each process in the system there exists some terminal state. Each wave is initiated by the process at the root of the tree. When the root process reaches

its terminal state a wave is initiated and a *control message* is passed to each of the root node's successors. The wave propagates through the tree as long as each of the nodes encountered is in a terminal state. As the wave reaches a node the basic computation of that process *freezes*.

Each node P is responsible for informing its parent whether or not each of P 's children are in a terminal state. On encountering a process which is not in a terminal state the wave is propagated no further and a *negative* reply is returned to the parent. If each of the children of a node P are negative then a *positive* reply is returned to P 's parent. When a positive reply reaches the root of the tree then the termination state holds globally for the computation. If a negative result reaches the root then an *unfreezing* wave is initiated to resume the basic computation at each frozen node.

The wave *freezes* each node in the spanning tree and so clearly interferes with the *basic computation*. [Top84] and [FR82] describe tree-based waves which achieve distributed termination detection without *freezing*.

2.2.2.4 Parental Responsibility Algorithms

A parental responsibility algorithm constructs tree structures over the network of processes as basic messages are passed in the distributed system. These structures are known as *computation trees*. Such algorithms get their name from the parent child relationship that is formed between processes as basic message as passed. The definition of the *passive* state is extended so that a site can only become *passive* when each of its children is *passive*. Thus a site is said to be responsible for identifying when its children become passive.

Dijkstra and Scholten [DS80] describe a parental responsibility algorithm for detecting termination of a *diffusing computation* executing over a directed graph of

processes. The distributed computation begins with a process known as the *environment* which sends out messages across the network thus causing remote processes to become active. Each process can receive *basic messages* from its successors in the network and an *active* process can send *basic messages* to its successors. When a process P sends a *basic message* which causes a process Q to become *active* then a parent child relationship is established between P and Q . When a process R becomes *passive* and has no *active* children a *control message* is sent to R 's parent. Receipt of a *control message* may cause the receiving site to become *passive* in which case the receiving site sends a *control message* to its parent. When the *environment* is *passive* the computation has terminated.

Chandy and Misra [CM82] show how the Dijkstra and Scholten algorithm can be adapted to detect computation termination and deadlock in CSP [Hoa78] networks.

Shavit and Francez [SF86] describe a hybrid algorithm which combines multiple *diffusion trees* and a ring algorithm to allow for termination detection of a non-centralised CSP computation.

The distributed computation is composed of a *forest of diffusion trees* where the root of each tree is the *environment* for part of the basic computation. Each tree in the *forest* is constrained so that once it has collapsed it remains so. However this does not mean that a process of a collapsed tree cannot become active again, just that if it does become active it will be part of a different tree. The computation is terminated when each tree in the forest has collapsed. To detect termination, a ring algorithm visits each process which is part of a collapsed tree.

2.2.3 Credit Recovery

Mattern's Credit Recovery algorithm [Mat89] differs from the above algorithms in that it does not fall into either of the categorisations. The algorithm imposes no particular network topology on processes of the system and does not represent a wave based computation. The algorithm assumes a centralised computation and reliable asynchronous communication but does not require ordered delivery.

The basic computation is initialised by an *environment* process which is known to each other process and is responsible for detecting termination. However the *environment* takes no part in the execution of the *basic computation*, i.e. it receives no *basic messages* and sends *basic messages* to other processes only during the initialisation of the computation.

Each process is initially passive. As usual a passive process becomes active on receipt of a *basic message* and only active processes may send *basic messages*. Each active process and each basic message in-flight holds a *share* of a global *credit* value C . At all times the sum of the *credit shares* held by the *environment*, each active process and each *basic message* in-flight, is equal to C .

On initialisation of the computation the *environment* distributes credit shares which total C to each initial process. When a process P_i with credit share csP_i sends a *basic message* P_i keeps half of csP_i and the other half is given to the message. If a process P_j receives a basic message the credit share of the message is added to csP_j . When a process becomes passive its credit share is sent to the *environment* in a control message. When the total *credit shared* held by the *environment* equals C , the computation has terminated.

2.3 Summary

From the descriptions above it is clear that the collectors make different model assumptions for message passing, inter-site addressing, network topology and computational mode. This means that comparison of two distributed garbage collectors is non-trivial and that correctness proofs for different collectors can be difficult to reconcile.

A number of the collectors described above make explicit use of distributed termination algorithms in their collection mechanisms, for instance [Hug85], [Aug87], [Piq91] and [HK82]. In each case the DTA is used to identify some globally stable property of the distributed system.

This thesis examines a technique for developing distributed collectors which make use of existing distributed termination algorithms. The intuition behind this approach is that through a separation of concerns within a derived distributed collector the task of comparing it to another collector is made easier. The inherent modularisation from the use of an existing DTA can also lead to a modularised approach to the construction of correctness proofs. That is, if a proof exists for the DTA then only the additional components of the distributed collector and the interaction of these modules with the DTA need to be proved correct.

The connection between solutions to the distributed termination detection problem and distributed garbage collectors has been well documented. Tel and Leeuwen [TL86] identify the distributed termination problem inherent in any distributed graph marking scheme. They go on to demonstrate the derivation of such marking schemes from DTAs.

On the other hand Tel and Mattern [TM93] show how termination detection for distributed computations can be modelled as an instance of the distributed garbage collection problem. Thus they describe the derivation of distributed termination algorithms from distributed garbage collectors. One such derivation shows how Mattern's credit recovery DTA can be modelled as an instance of Watson and Watson's weighted reference counting collector. Both the DTA and the distributed garbage collector were proposed independently however commercial use of weighted reference counting is protected under US and UK patents.

Blackburn et al. [BHM+01] describe a methodology for the derivation of distributed collectors from distributed termination algorithms. This is not the reverse of the derivation process described by Mattern and Tel. Indeed Mattern and Tel comment that the reverse mapping will only yield a reference counting collector.

The mapping methodology is not automatic and requires creativity in determining the globally stable property of the distributed state that corresponds to the identification of sets of garbage objects. Once this has been done a DTA mapping is established such that termination corresponds to the identification of this globally stable property.

3 The Experimental Platform

Chapter 1 describes an abstract distributed system that serves to ground the discussion on distributed garbage collection. This chapter presents a concrete instantiation of the abstract system that provides a target platform for the experiments in distributed garbage collector implementation. The target platform is a distributed architecture for the execution of ProcessBase applications, called the distributed ProcessBase (DPBASE) system.

3.1 The ProcessBase Language

ProcessBase [DFM+03, MBG+99] is one of a family of languages designed to support process modelling. From the ProcessBase language manual,

“The type system contains the base types integer, real, boolean and string. Higher-order procedures allow code to exist in the value space. Aggregates may be formed using the vector and view types. Both of these allow information hiding without encapsulation. Finally there is an explicit constructor to provide locations.”

Locations in the ProcessBase language are first class entities and provide the only mutable data type. The language and its runtime system provide a number of other key features such as:

- strong typing with an emphasis on static checking;
- type completeness;
- first class procedures;
- an infinite union type with dynamic projection;
- orthogonal persistence;

- threads;
- automatic memory management;
- compliance.

Figure 3.1 below illustrates the layered architecture of the ProcessBase runtime system. The ProcessBase system consists of the language and its object-based runtime environment. That is, in running ProcessBase code (in the form of threads) the interpreter manipulates objects resident in its local object cache. The object cache holds both persistent objects, that have been faulted from the persistent store, and new objects, created by the executing code.

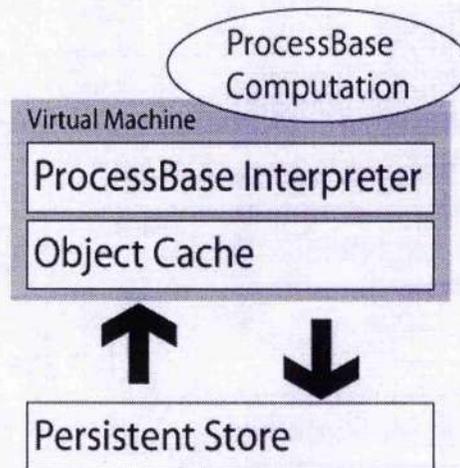


Figure 3.1 - The ProcessBase Runtime Architecture

3.1.1 Compliance in ProcessBase

The ProcessBase language and its runtime system are an instantiation of a compliant systems architecture (CSA). A CSA offers flexibility in the separation of policy and mechanism in a software system, allowing the running application to tailor its environment to its needs. In the ProcessBase CSA, policy governing the operation of the runtime system can be evaluated at the application level through independent up-call and down-call mechanisms. Information is passed 'up' from the VM to the

application through the up-call mechanism. VM instructions raise interrupts that are handled by application level interrupt-handler code. Results returned by handler code for a particular interrupt is passed 'down' to the VM when the handler's execution completes.

VM instructions can be executed explicitly by the application via the down-call mechanism, through library interfaces, which also allows information to be passed 'down' from the executing code to the VM. The down-call mechanism allows extensions to be added to the VM, in the form of additional instructions, and made available to the application via the corresponding library interfaces. The ProcessBase system can be considered as comprising of four parts, as shown in Figure 3.2. Library code is specific to the particular extension defined in the VM and is called by the application code. The compiler generates ProcessBase byte-code which maps onto the VM instructions for both the core interpreter and the interpreter extensions.

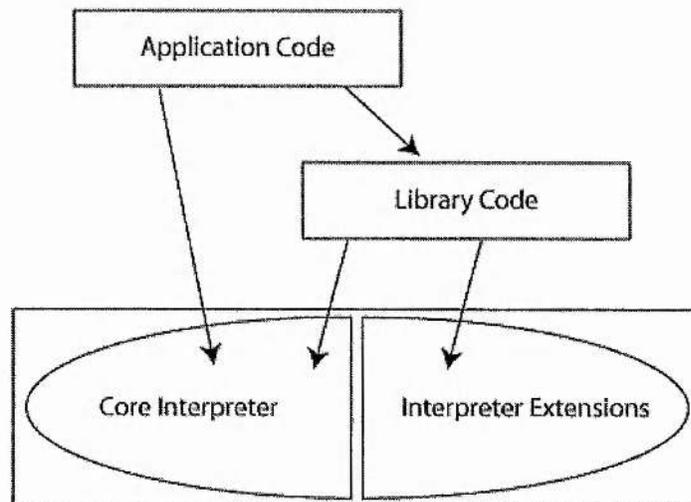


Figure 3.2 - ProcessBase Interpreter Extensions

Compliance in ProcessBase can be used to control the policy governing a number of aspects of the run-time system. For instance, synchronisation, thread control and

scheduling, input/output, recovery and distribution. See [FDH+04, MBG+00] for more details on the CSA and ProcessBase.

3.2 The Distributed ProcessBase Architecture

A ProcessBase program consists of a single computation composed of one or more threads of execution operating in a shared namespace. The ProcessBase compiler maps the shared name-space of the computation into byte-code with a single address space. In executing byte-code the uni-processor interpreter in turn transforms the compiler's address mapping into a mapping to the address space of the local object cache.

In the distributed ProcessBase system the interpreter maps the compiler's single address space across a number of sites. Many different mappings are possible. The distributed ProcessBase system described here represents one such mapping and its implementation.

The model of computation for the distributed ProcessBase system [BDF+01] is shown in Figure 3.3 below. A distributed ProcessBase program consists of a single computation composed of multiple thread closures (T), which execute within a single shared name space across a number of distributed sites. The union of the sites in the distributed system constitutes a distributed virtual machine (VM) for the execution of ProcessBase code. At each site of the distributed VM there exists a ProcessBase interpreter operating over a local object cache. An interpreter is defined as an agent of execution for ProcessBase instructions. The local cache holds new objects created by the local interpreter and both persistent objects faulted from the persistent store and non-persistent objects created by other sites that have been faulted to the local site.

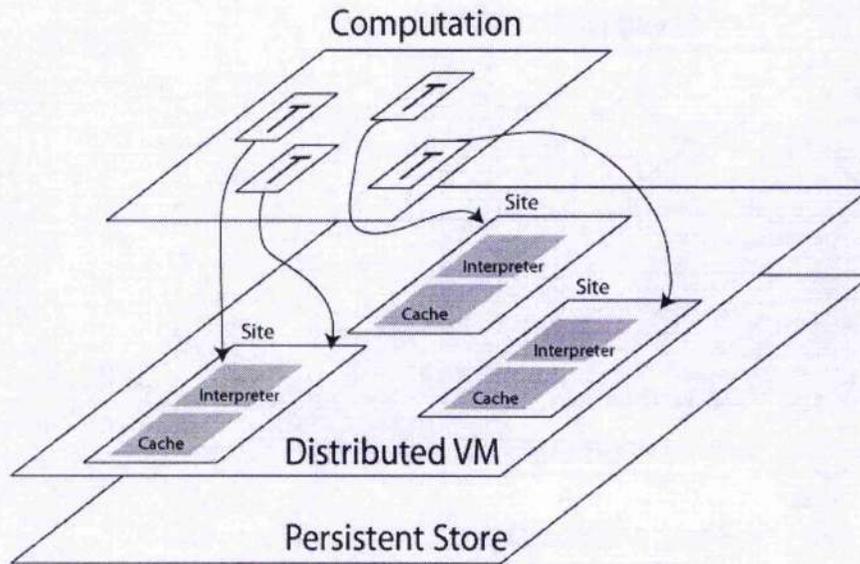


Figure 3.3 - ProcessBase Computational Model

The computation begins with a single thread executing on one site. As this thread executes it can form thread closures that are run locally or are exported to a remote site, thus distributing the computation. Sites communicate only through message passing and a communications channel, providing guaranteed ordered delivery, is maintained between each site.

Messages between two sites are delivered in the order that they were dispatched, they are not lost in transit and messages are dealt with at the target site strictly in the order that they were delivered.

The distributed ProcessBase system represents a concrete instantiation of the abstract distributed system model from Chapter 1.

3.2.1 Inter-site Addressing

The shared namespace of a distributed ProcessBase program is mapped (at run-time, by the distributed VM) onto a distributed graph of objects. Inter-site references between these objects are represented by two part distributed addresses of the form

<site, local id>.

The *local id* part is symbolic and provides one level of indirection for object addresses that allows for independent relocation of objects at a site. Addresses that are entirely local to a site (i.e. that represent references between objects on the same site) are optimised to omit the *site* part of the address and the *local id*, in such addresses, is no longer symbolic but instead is the local cache address (CA) for the object at that site.

A thread running at a site may create objects in its local cache during execution. A thread closure exported from site *S* to site *T* will result in *T* obtaining references to objects local at *S*. On export from *S* to *T*, cache addresses are translated to distributed addresses and are known as remote references at *T*.

In effect, the *<site, local id>* tuples form a distributed shared address space that is mapped onto the address spaces of each of the local caches in the system. The implementation of this mapping is known as the *distributed object cache* for the distributed ProcessBase virtual machine [NFB+01]. Objects are initially allocated in the local cache of their creating site (that is, the local segment of the distributed object cache) and given an address in the distributed address space if and when they become referenced from a remote site. It is the distributed object cache specifically that constitutes the target environment for the distributed garbage collectors described in this thesis.

The architecture of the persistent storage layer in the distributed ProcessBase system is orthogonal to the issues concerned with garbage collection of the distributed cache. The purpose of the distributed ProcessBase system described in this thesis is to provide a target architecture for distributed garbage collectors. With this in mind,

no discussion of the persistence architecture, the persistent storage mechanisms or the persistent object addressing scheme is presented.

3.2.2 The Distributed Object Cache

The distributed addressing mechanism is opaque in terms of an object's location at a particular site. Effectively the symbolic local id only has meaning at the site holding the object. To implement the distributed address (DA) mechanism each site maintains a distributed address to cache address translation table. This table maps the symbolic (*local id*) part of an object's DA to its cache address, or CA, at the local site, and is called the $DA_{\text{sym}} \rightarrow CA$ translation table. When a reference from site S is exported to site T, a DA is constructed and added to the $DA_{\text{sym}} \rightarrow CA$ translation table at S. If the object is moved at S (i.e. given a different CA) only the $DA_{\text{sym}} \rightarrow CA$ table entry is updated, and no other site needs to be informed.

3.2.2.1 Coherency Policy and Object Duplication

The distributed ProcessBase system described here is neutral towards cache coherency policy and program synchronisation mechanism. The garbage collectors operate over a graph of objects irrespective of these mechanisms. It is possible that the combination of the garbage collector, coherency policy and synchronisation mechanisms will be more efficient. See [MFL+01] for one particular example.

To accommodate different coherency policies the system allows for object duplication. A site that holds a remote reference to an object can request a copy of the object from its creator site. Such a copied object is known as *remote resident*. In support of this each site maintains a second address table, called the $DA \rightarrow CA$ address translation table, which maps from a *remote resident's* DA to its CA in the local cache. On discovery of a DA, a site can inspect its $DA \rightarrow CA$ table to see if it

already holds a copy of the object and thereby locate it in the local cache. Table entries are added on object import and (as with the $DA_{sym} \rightarrow CA$ translation table) updated if remote resident objects are moved within the local cache. Table entries are removed as remote resident objects are reclaimed by a local cache collector implemented at each site.

Location objects represent the only mutable data type in the value space of a ProcessBase program. All other language accessible objects (views, arrays, strings etc) are immutable. This means that any coherency mechanism need only operate over location objects, and that other objects can be replicated freely, depending on policy. The approach here is to adopt what is considered the simplest possible coherency mechanism for the experiments in distributed garbage collector implementation, whereby mutable objects (locations) are never replicated. A location L is fixed at its creator site and access to L is afforded to remote sites through a remote dereference and a remote update operation. A site may dereference or update a remote location by sending an asynchronous message to the site holding the location. In this way read and write operations on a location object are serialised through the site holding the location.

The garbage collectors in the distributed object graph operate over a graph of objects irrespective of coherency policy. In this particular scheme, any copied objects at a site are treated as local objects that are not shared with any other site. Effectively the distributed garbage collector operates over only one copy of each object, since this is enough to describe the distributed object graph.

3.2.2.2 Synchronisation

Application thread synchronisation in the DPBASE system is provided by an interpreter mechanism that provides synchronisation on access to individual locations. A site may request a voluntary lock on a location by sending a *lock request* message to the site holding the location object. If the location is not locked then a lock is granted, otherwise the requesting thread is added to a queue. The lock on a location is released by sending a *lock release* message to the site holding the location. On receipt of a lock release message a site grants the lock to the next thread in the queue.

These are voluntary locks because the interpreter does not enforce mutual exclusion on locked location objects. Location update instructions and remote update messages may change the contents of a location even if a voluntary lock has been granted. Mutual exclusion is only guaranteed only if each thread that accesses a location uses the voluntary mechanism. The point of this locking mechanism is to provide the means for an application to enforce access control and update ordering on locations if it wishes. Due to the strict ordering of message delivery and that enforced on message servicing, the last remote update made by a thread to a location that it holds the lock for, is guaranteed to have been made before the next thread is granted the lock.

Each site maintains a lock table that records the thread currently holding a lock and the lock request queue, for each local location object. Entries in the table are added lazily, that is on receipt of the first lock request message, and are removed on receipt of a lock release message if there are no threads in the lock request queue.

3.3 The Distributed ProcessBase Implementation

A site of the DPBASE system corresponds to a physical machine on the network. The distributed VM is initialised by a single *master* site which starts an interpreter process on each site, and then starts the first ProcessBase thread on one of these sites. Each of the sites of the distributed VM, and the byte-code to execute, is specified by an XML VM description file. The number of sites in a particular distributed VM is fixed and is determined on initialisation of the system. Every site maintains one end of a communication channel with each other site. The network of sites is thus fully connected. A site of the distributed VM is identified to each other site and to the ProcessBase application by a globally unique identifier. After initialisation of the distributed VM, the master site is of no particular significance.

The distributed ProcessBase system is implemented on a Linux based Beowulf cluster [BSS+95]. The cluster consists of 64 nodes connected through a switched Fast-Ethernet network. Such an architecture corresponds well with the abstract model from Chapter 1. The nodes of the network are ‘loosely coupled’ in that they are completely independent and communicate only through a reliable network infrastructure.

3.3.1 Message Passing in the Distributed VM

Full-duplex TCP channels provide guaranteed ordered delivery of messages passed between sites. The DPBASE system defines a common message structure as shown in the table below. Each message is identified by a unique identifier.

| Message Field | Description |
|---------------|--|
| Type | The integer identifier for the message type. |
| Id | An identifier for a particular message. For request messages where a reply is required the request id is included in the reply's data section. This allows a site to match up replies it receives with the requests it has sent. |
| From | The identifier for the site that is the source of the message. |
| To | The identifier for the site that is the destination of the message. |
| Data size | The size in bytes of the data section of the message. |
| Data | The message payload. The message payload for each message type is described later in this thesis. |

Table 3.1 - The DPBASE Message Format

3.3.2 The Local Object Cache Layout

In describing the implementation of the DPBASE system, and particularly the distributed object cache mechanisms, it is useful to give an overview of the layout of

the interpreter's local object cache. This layout is unchanged from the single processor 32-bit ProcessBase abstract machine as described in [MBG+99a].

3.3.2.1 Uniform Object Format

Objects in the local object cache conform to a uniform four-byte (word) aligned object format, shown in Figure 3.4.

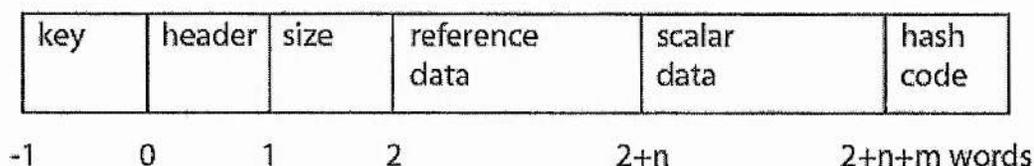


Figure 3.4 - The Uniform Object Format

This illustrates the word offsets and layout for an object in the local cache that contains n references and m words of scalar data.

In the uni-processor ProcessBase system the key word is used to store the Persistent Identifier (PID) for objects that have been faulted from the persistent store, in effect implementing an object to PID mapping in the local cache. Use of the key word in the DPBASE system is explained later. The location of the object's header word in the object cache represents an object's local cache address and bits 0-23 of the header specify the number of pointer fields in the object.

The hash-code field is effectively an additional scalar data field which is carried by every object and is used for experimental purposes.

3.3.2.2 Local Interpreter Objects

An interpreter maintains a local object structure reachable from a root object that is held at a known place in the local cache. Four of the object types in this structure are of importance in describing the DPBASE system. These are:

- The *nil view* object - Each nil view literal declared in the application code is represented by a single nil view object in the interpreter. A reference to a nil view exported from a site A to a site B is translated at site B as a reference to the local nil object.
- The *single character string* - The interpreter maintains a table of single character string objects and any single character string literal in the application code is represented by the corresponding object in the table (this is for efficiency). A reference to a single character string exported from a site A to a site B is translated at B to a reference to the corresponding single character string in B's local single character string table.
- The *stack object* - A thread is executed by the interpreter on two contiguous stacks⁵, one stack for scalar data, called the main stack, and the other for reference data, called the pointer stack. The two stacks for a thread are implemented using a single object called a stack object. Figure 3.5 below illustrates the layout of the stack object maintained for each thread. The pointer stack grows upwards from the first reference field of the stack object while the scalar data stack grows downwards from the last scalar data field of the object.

⁵ The twin stack architecture is derived from the S-Algol abstract machine as described in [BMM80].

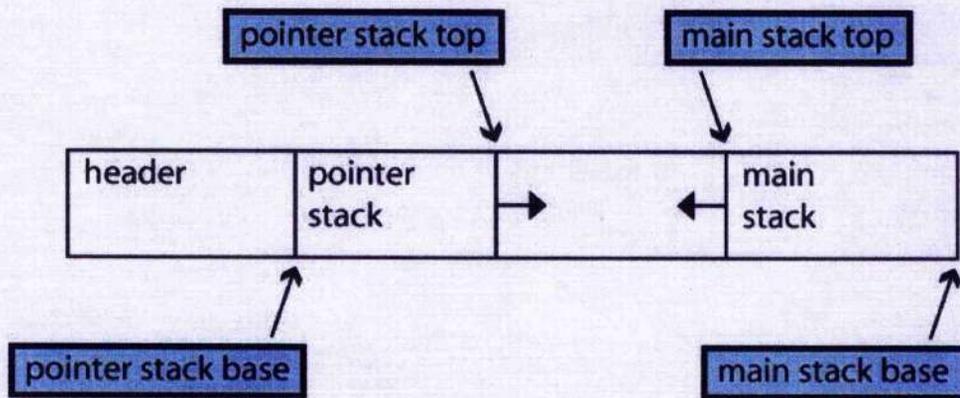


Figure 3.5 - The Layout of a Stack Object

- The *thread control block* (TCB) - The interpreter maintains a TCB for each thread created at that site. The TCB records the execution state for the thread which includes references to the thread's closure (that is, the thread's code vector and environment), a reference to the thread's stack object, indexes for the current frame base and stack top for the pointer and main stacks, the code pointer, and thread status flags. Each TCB also contains an internal (implementation language specific) reference (treated as scalar data in terms of the uniform object format) to an area of per-thread *work space*. Use of the work space is explained later.

The nil view object and the single character strings never become unreachable and as such are never reclaimed by the garbage collector. Stack objects and TCBs become garbage on completion of the thread they are associated with. However, these objects are never shared with a remote site and so can be reclaimed using purely local information. That is, no reference to a TCB or a stack object is ever exported to a remote site, therefore when these objects become unreachable from the local root at their site, they may be reclaimed, without the involvement of a distributed garbage collector.

3.3.3 The Distributed Object Cache

Inter-object references in the DPBASE system are a single word in length and provide object level addressing. There are three distinct object addressing mechanisms used in the distributed object cache. These are discussed in Table 3.2 below.

| Address | Description |
|-----------------------------|--|
| Persistent address (PID) | An object's persistent store address. |
| Local cache address (CA) | The memory address of the start of an object resident in the local cache segment. The cache address for an object <i>o</i> resident at site <i>A</i> is only valid (and only known) at site <i>A</i> . No other site can use this address to access the object. The most significant bit of an object reference is used to distinguish PIDs from CAs. The most significant bit (MSB) is always set for a PID and cleared for a CA. The effect of this implementation choice is that we can distinguish CAs from PIDs. However, a side effect is to limit the local cache address space to 31 bits. |
| Distributed address (DA) | This is a cache-location-independent identifier for a shared non-persistent object. A DA is allocated to an object when a reference to that object is first exported from the site where the object was created. As with CAs the MSB of a DA is always cleared. In order to distinguish between DAs and |

| | |
|--|--|
| | <p>CAs the least significant bit (LSB) of the reference is used.</p> <p>Since the object cache is four-byte-aligned the LSB is clear for every CA reference. This bit is set to identify a DA.</p> <p>When an object is allocated a DA, the DA is recorded in the object's key word. This provides a mapping from objects to DAs, which is the reverse of the mapping provided by the $DA_{sym} \rightarrow CA$ and $DA \rightarrow CA$ address translation tables. The object to DA mapping is required to allow sites to identify remote resident objects and local objects that have been allocated a DA.</p> |
|--|--|

Table 3.2 - Addressing Mechanisms in the Distributed Object Cache

3.3.3.1 Object Copying

Any non-mutable object in the distributed object cache can be replicated across sites. The chosen policy is for an interpreter to obtain a local copy of any non-mutable object before dereferencing it. When an interpreter dereferences an object whose reference is a DA it first checks to see if a copy of the object is already held locally and if not, sends a message to the site holding the object to request a copy. This is known as a *site to site object fault* request. The thread that calls an instruction that causes such a fault request is first blocked and the instruction restarted. The thread scheduler is then instructed to schedule a different thread. The effect of this policy choice on the implementation of the DPBASE system is to make the remote dereference operation for non-mutable objects redundant. Hence it is not implemented.

To prevent multiple fault requests being sent from a particular site for the same object, the interpreter maintains a structure called the *DA wait list* at that site that records the DA and list of requesting threads for each fault request that has been sent and for which no reply has yet been received. Any thread requiring a site to site object fault for a DA that is already in the DA wait list is added to the list of threads waiting on the object fault.

A site to site fault request message can request a single object or a function closure (i.e. two objects) from the target site. The payload consists of the DA of each object requested. The message request id is set to be the thread id for the requesting thread.

On receipt of a site to site object fault request for a non-mutable object, a site creates a copy of the object and sends it to the requesting site. If the requested object is mutable the site replies to the request with a message indicating that the requested object is a location. It is therefore necessary for a site to be able to identify location objects from other (non-mutable objects). This is achieved by setting a bit in the header word for each location object when it is created.

The number of objects requested may be inferred (by the receiving site) from the size of the payload of the request message. If the requested object is a location then the payload of the reply contains only the request id from the request message. Otherwise the payload of the reply consists of the request id from the request message followed by an encoding of the object(s) being sent. An object x is encoded as a word for word copy of the object, including its key word. Any references in x are replaced (in the copy of x) with the DAs of the objects to which they refer, allocating new DAs as necessary.

When the requesting site receives the reply to the site to site fault request one of two things happens. If the reply contains a copy of the requested object then the following actions are taken:

- a local copy of the object is created in the local object cache;
- an address mapping is added to the site's DA \rightarrow CA address translation table for the local copy;
- each of the threads, recorded in the DA wait list structure entry for the DA, is unblocked;
- the DA wait list structure entry for the DA is removed.

On the other hand, if the reply does not contain a copy of the requested object, then the object is a location. To prevent sites from repeatedly sending site to site fault requests for location objects it is necessary that a site can identify location objects from their DA. To this end, each site maintains a third address table, called the *remote location table*, which records the DAs of remote location objects. A site to site fault request is therefore only sent if there is no local copy of the object and the DA is not in the remote location table. On receipt of a site to site fault reply for a location object the following actions are taken:

- the DA for the requested location is added to the remote location table;
- each of the threads, recorded in the DA wait list structure entry for the DA, is unblocked;
- the DA wait list structure entry for the DA is removed.

The point at which a site to site fault request is issued for a particular DA is ultimately a matter of policy.

When the requesting thread is next scheduled the DA translation will result in either the CA of the local copy or an indication that the DA refers to an location.

3.3.3.2 Remote Update and Dereference

Operations on remote locations are facilitated through remote update and dereference mechanisms. Since the interpreter's policy is to obtain a local copy of a non-mutable object before dereferencing a field of that object, the remote dereference and update operations apply only to remote locations.

The interpreter's dereference (and update in the case of locations) instructions are specific to the type of the objects they operate over. That is, the interpreter implements different instructions for operating on locations, infinite unions, vectors and views. For each of these types there are four distinct instructions; reference; double reference (function closures), word (integer and Boolean values) and double word (real values). For instance, a ProcessBase operation that reads a single reference value from a location maps to a specific instruction that dereferences locations containing single reference values.

In the design of the DPBASE system the interpreter's location dereference instructions are extended (from the uni-processor ProcessBase versions) as follows. The approach here is fundamentally the same as that taken in the implementation of the *site to site object fault* request mechanism. However in this case we require that the thread carries some additional state to indicate (when the instruction is executed) whether or not a message has already been sent (in the case of the *site to site object fault* requests this is indicated by the result of the DA translation). It is therefore necessary to define a thread status flag, *location read wait*, that indicates that the last time the thread was scheduled for execution the current instruction was restarted due

to a remote dereference. If the location is remote the instruction first checks the location read wait flag. If the flag is set the location's contents have already been received and can be read from the thread's work space. If the flag is not set, the instruction blocks the calling thread, sets the flag and restarts the instruction. A *location read* request message is then sent to the site holding the location. The message contains the location's DA and an id for the thread that executed the instruction. On receiving a location read message, the site holding the location sends a reply containing the location's contents and the thread identifier contained in the request message. On receipt of a reply message the interpreter writes the location contents into a temporary work space associated with the requesting thread's and unblocks the thread.

The location update instructions are extended as follows. If the location is remote then a *remote location write* message is sent to the remote site containing the value that is to be written to the location. The thread is not blocked.

There are a number of policy choices surrounding the remote update and dereference operations. These policy decisions relate to whether or not sites attempt to preemptively send copies of objects between sites when a remote operation is requested. For instance when a site replies to a *location read* request there is a potential benefit in sending a copy of the referenced object in the reply message; obviously this is only possible if the site holds (a copy of) the object in the first place. The requesting site would benefit from such a policy if that site in turn dereferences the object referenced by the remote location, since it will already hold a copy. A policy, complementary to this first example, is for a site to send a copy of an object when it requests a remote update. This is described as complementary to the first policy since the site holding the location is now in the position of holding a copy of the object

that the location references, thus allowing a copy to be sent along with any reply to a *location read* request. Note that neither of these policies is implemented in the DPBASE system at present.

Both of these policies further extend when function closures are considered as explained below.

3.3.3.3 Object Equality

Object equality in the ProcessBase interpreter is based on reference value. For a DA d referring to an object x and a CA c , at a site s , c equals d if and only if the local cache at s holds a copy of x (with cache address c_2) and c equals c_2 . If s does not hold a copy of x then c cannot be the cache address of a local copy of x and therefore c does not equal d .

3.3.4 Local Garbage Collection

Local garbage collection is provided by a non-incremental mark-compact collector. The local cache collector for a site of the DPBASE system is principally the same as the cache collector in the uni-processor ProcessBase abstract machine. The key additions in the DPBASE system are to take account of the distributed addressing mechanism. Local garbage collection proceeds as follows.

- All threads at the local site are stopped.
- The object graph is then traversed from the local root object and from each object referenced by the $DA_{sym} \rightarrow CA$ table marking all reachable objects. Therefore any object held in the local object cache, reachable from a reference held on the pointer stack of a local thread, or to which a reference has been exported to a remote site, is marked. A single bit in the object header is used for marking.

- The local cache is then compacted using an implementation of the Lockwood-Morris algorithm from [LM78]. The $DA_{sym} \rightarrow CA$ table entry for each object x is treated as an object holding a reference to x and as the local cache is compacted all local references and $DA_{sym} \rightarrow CA$ table entries are updated.
- The $DA \rightarrow CA$ table entry for each unmarked remote resident object is removed during compaction.

This collection mechanism is clearly not complete since there exists no mechanism to safely remove $DA_{sym} \rightarrow CA$ table entries and as such an object can never be reclaimed once a reference to it has been exported to a remote site. Chapter 5 describes a number of distributed collection mechanisms that allow $DA_{sym} \rightarrow CA$ table entries to be safely removed.

3.3.5 Remote Thread Execution

The granularity of distribution for a ProcessBase application in the DPBASE system is a thread. Computation begins with a single thread running on one site and is distributed as new threads are spawned and sent to remote sites for execution. Any void ProcessBase function (that is, a function that returns no value) can be executed as a thread.

3.3.5.1 Thread Closures

A ProcessBase function closure consists of two objects: a code vector, generated by the compiler; and an environment, constructed at run-time, that holds the free variables required for the execution of the code. The transitive closure for a function therefore contains any object reachable from a reference held in either the code vector or its environment. The thread closure that is sent to a remote site on remote

thread invocation contains at least the code vector and environment for the function. It is a matter of policy whether or not any other objects are sent. The default policy is to send any objects referenced by the closure's code vector, which are the string objects for any string literals declared in the function.

A thread closure is sent to a remote site in a *remote thread start* request message. The payload starts with the number of objects contained in the message followed by the code vector, the environment and each of the other objects in turn, encoded as described above for site to site fault reply messages. On receipt of a remote thread start request message a site unpacks each of the objects, adding the appropriate DA → CA table entries, and creates a new local thread to execute the closure. After the thread has been created the site sends a reply message containing the thread id for the new thread.

3.3.5.2 The Remote Thread Library Interface

Distribution of a ProcessBase application is explicit. That is, the application specifies which threads are to be executed remotely and on which sites they are to be executed. To facilitate the distribution of a ProcessBase application, the extension to the VM that allows a thread to be started on a remote site, is exposed to the application level through a library interface (shown below).

```
let remoteThreadStartOp <- opcode 232 [] (int, fun ()) -> int

let remoteThreadStart <- fun (s : int; f : fun()) -> int;
{
    downcall remoteThreadStartOp [] (s,f)
}
```

The function *remoteThreadStartOp* (line 1) defines a down call to an interpreter instruction that packages up a thread closure and sends it to a remote site. The function *remoteThreadStart* (line 3) simply wraps the down-call in a normal ProcessBase function that takes an integer site identifier and a function closure as its arguments.

3.3.5.3 Policy for Remote Update and Dereference with Closures

The interpreter's policy is to send at least the code vector and environment object when a thread closure is sent to a remote site. This policy may be extended to remote dereference and update as follows. When a site receives a *location read* request for a location containing a function closure, copies of the code vector and environment are sent in the reply if the site holds (copies of) the objects. As before a complementary policy to this is to send copies of the code vector and environment objects when a site sends a *remote location write* request. In both cases this can extend to as much of the function closure as a site wants to send.

3.4 Summary

The DPBASE system represents an instantiation of the distributed system model from Chapter 1 and provides a distributed computational environment which serves as an experimental platform for the derivation of distributed garbage collectors.

The contribution here lies in the specification of a simple distributed computation system that can be used as a target environment for distributed collector design. The system is neutral towards cache coherency policy and synchronisation mechanisms. An architecture and computational model for DPBASE was first presented in [BDF+01]

The specific contribution of this author lies in the design and implementation of the instantiation of the DPBASE system described in this chapter.

4 The Task Balancing Distributed Termination

Algorithm

Distributed termination algorithms are useful because they allow for the detection of globally stable states in distributed computations. In [TM93], Tel and Mattern show that a DTA can be derived from any distributed garbage collector. The work of Blackburn et al. (in [BHM+01]) presents a methodology for deriving a distributed collector through the construction of a mapping from a centralised collection scheme to a DTA. Chapter 5 of this thesis examines this in more detail and demonstrates a more general modularisation of distributed collector design that builds on the mapping methodology. This chapter presents one particular DTA called Task Balancing (TB).

The TB DTA was first explicitly described in [BHM+01] although the Pointer Tracking protocol from the DMOS collector described in [HMM+97] represents an implementation of the algorithm. However TB has never been described in a satisfactorily generic form. This chapter examines the fundamental nature of the Task Balancing DTA (showing that [BHM+01] itself presented a particular implementation) and presents a number of implementation issues related to the algorithm. In describing Task Balancing a less process-centric statement of the distributed termination problem is used (originally published in [BHM+01]) than that given in Chapter 2. The Task Balancing DTA is described in terms of this problem statement.

4.1 A Model for Distributed Termination

The distributed termination problem is stated in terms of a *job* consisting of a number of dynamically spawned *tasks*. While a job is distributed, each of its tasks runs at a single site. In particular, the following actions/events define the notion of a distributed job:

- Any site can create a new job j . Initially a new job consists of a single task running at the creating site.
- Any running task of a job j may spawn (create) additional tasks of j . A new task may run on the same site as its creating task, or it may run on some other site, i.e., be created to run on a site different from its creator's. One may think of this as sending a task from one site to another.
- A task may complete spontaneously.
- When all of the tasks of job j are complete and there are no tasks of j in-flight between two sites, j is said to be terminated, written *terminated*(j). Note that once *terminated*(j) is true it remains so. That is, the terminated state is globally stable.

The goal of distributed termination is to determine when *terminated*(j) becomes true, for any given job j . The possibility of having tasks in-flight due to the asynchronous nature of the system contributes significantly to the difficulty of detecting termination.

4.2 Task Balancing

The TB DTA operates by balancing counts of the tasks sent between sites and (separately) the number of tasks received and completed at each site. The algorithm requires that a single site be identified as the home site (or arbiter site) for a given

job. The home site is responsible for detecting termination by bringing together the *sent* and *received/completed* counts for sites that hold (or held) a task of the job.

A central principle in the TB DTA is the distinction between those tasks that are sent between sites and those tasks that are spawned locally at a site. It is shown later that tasks spawned at a site may be balanced locally (at that site). In this regard the description of TB in [BHM+01] differs from that presented here in that Blackburn et al. make no distinction between tasks received from a remote site and those spawned locally. Their approach is by no means incorrect but as we will see, is only one of a number of implementation choices. The intuition behind the algorithm presented here is that if locally spawned tasks can be distinguished from those received from another site, and these local tasks can be balanced locally, then to transmit data concerning locally spawned/completed tasks, having received a task, is redundant.

Progress towards termination detection is made by a remote site S sending, at an appropriate time (described below), to the home node H of job j , an *update* message containing the current *received/completed* and *sent* counts for j at S . More specifically, an update message contains the number of tasks of j *received/completed* at S , and for each site T the number of tasks of j sent from S to T . When the home site H detects that for all sites T the total number of tasks of j sent to T is the same as the number of tasks *received/completed* at T then *terminated(j)* is true.

Since locally spawned tasks are balanced locally the *home* site must not be informed of the completion of a site's final received task until all of that site's locally spawned tasks have completed. Update messages may be sent at any time, at site S and for job j , when either of the following conditions is satisfied.

- All locally spawned tasks of job j at site S have completed.

- There exist, at site S , uncompleted locally spawned tasks of job j and at least one uncompleted received task of j .

An important aspect of the TB DTA is that termination detection for a job j may occur with the minimum possible number of messages; that is one message per site that has received a task of j . Update messages represent the only additional messages imposed on the distributed systems since it is assumed that the sending of tasks between sites is superimposed on existing messages. That is, any message in the system that carries a task from a site S to a site T , contains enough information that the site T can identify the task being received.

Clearly the absolute minimum message complexity for termination detection is achieved if a site S sends an update only on completing the final task it spawns locally or receives from a remote site. This is difficult to achieve in a distributed context (although more likely not possible at all) since the sending of update messages would necessarily be controlled by the evaluation of some global predicate. That is an update for a job j could only be sent if a site knows it will receive no further tasks of j from a remote site. If the absolute minimum message complexity for termination detection is not achievable, what is?

It is useful to define the local state of idleness. A job j is defined as idle at a site S , written $idle_S(j)$, when S contains no running tasks of j . This corresponds to the passive state for processes in the traditional model of distributed termination detection. It is important to note that idleness is not a stable state; while a site can spontaneously move from the non-idle to idle state (for a given job j) the reverse is also possible through the receipt of a task (of j) from a remote site. The minimum achievable message complexity for termination detection of a job j , achievable without global knowledge, is one update message per site per period of activity for j .

A period of activity for j at a site S is defined as the time between S (which is idle for j) receiving a task of j and the point that it becomes idle once more. If each site has only a single period of activity for a job j then only one update message is sent per site and the absolute minimum message complexity is achieved.

4.2.1 Termination Detection at the Home Site

Termination of job j is detected at the home site H by balancing, for each site T , the count of tasks (of j) sent to T against the count of tasks received and completed at T . The home site balances the total number of tasks of job j sent to site T , irrespective of which sites sent the tasks, against the tasks of j received/completed at T .

The intuition behind the correctness of the TB algorithm is that for a job j there is no possible update delivery order that can lead to the home site balancing the *received/completed* count against the *sent* count for j for all sites, other than when j has terminated. This is due to the partial ordering of events in the system, where tasks are recorded on send and updates can only be received at the home site after receipt and completion of the tasks for which they hold counts.

4.2.2 Implementation Choices

Three aspects of the TB algorithm represent areas of choice for a particular implementation. These are:

- When and how the home site processes update messages and detects termination?
- How a site determines the point at which update messages are sent to the home site?
- How each site calculates the task counts (both tasks *sent* and tasks *received/completed*) for update messages?

First, two definitions are made. Each site maintains the value $counts_S(j,T)$ which represents the number of tasks of job j sent from site S to site T . An *update* message for site S and job j is defined as $update(j,C,RC_S(j))$; where

$$C = \{ \langle T, counts_S(j, T) \rangle \mid counts_S(j, T) \neq 0 \} \text{ and}$$

$RC_S(j)$ is the number of tasks of j received/completed at S .

It is the job of the home site H to balance these counts contained in *update* messages and to detect termination⁶.

4.2.2.1 Update Message Content

An implementation of the TB DTA has a choice of two options for the contents of *update* messages. The count values (*sent* and *received/completed*) represent either incremental updates to the values previously sent to the home site or running totals for the sending site.

- If the *update* messages contain incremental updates then the *sent* and *received/completed* values represent the counts for the sending site since an update was last sent.
- If the count values in an update message are running totals then they are monotonically increasing and potentially large. These are called 'standalone updates' since a single update contains all of the count information for a site up to the time at which the update was sent. The last update sent by a site S to the home site H contains all of the information required by H to reason correctly (safely) about the global state of a job j due to the site S .

⁶ It is assumed that the update messages sent from the home site H to itself are instantaneous.

Termination detection is achieved at the home site H for a job j by balancing for each site S the number of tasks *sent* to S against the number of tasks *received/completed* at S . For a job j when the *sent* count (known at H) matches the *received/completed* count (known at H) for each site S then *terminated*(j) is true. To detect termination of a job j the home site must calculate a single value for each site S (including H); which is the task count for j at S known to H and is written $TC_H(j,S)$. *Terminated*(j) is true when $TC_H(j,S)$ equals zero for all sites S . The task count for a site S and job j is calculated by subtracting the total $RC_S(j)$ value for *updates* sent by S from the total $count_S(j, T)$ value for update messages sent by all sites T . The amount of work required to calculate the task count for a site S and job j at any given time is dependant on update message content. The amount of data that must be maintained at the home site and when that data can be processed also varies between the two *update* content schemes.

4.2.2.2 Processing Updates at the Home Site

The choice of *update* content policy impacts on the message delivery requirements of the implementation and on how the home site processes the *update* messages for termination detection.

4.2.2.2.1 Incremental Updates

An implementation using incremental updates must provide a first-in first-out (FIFO) ordering in the channel delivering update messages sent from a site S to the home site H for a job j . This is to prevent false determination of *terminated*(j) for a job j due to update messages overtaking each other.

The *sent* and *received/completed* counts for a job j at all sites T contained in *update* messages are combined into a single task count for each site T at the home site H .

Each *update* message (for a job j) is processed in isolation and the task count values at H for j at each site T are brought up to date. An *update* message is processed on delivery or at some point later (but still in order of delivery) and then discarded.

This incremental processing policy translates into an implementation as follows. In such a scheme the home site H for job j maintains for each site T , a task count $TC_H(j,S)$ which records the number of tasks of j known by H to exist at T . On receipt of an *update* message the home node H of job j adds the value for each site T in the *update* message to its own task count for j at T . The home node H subtracts the *received/completed* value in the *update* message from its task count for the *update* sender. Termination detection for a job j is achieved by examining the task count at H for each site T . When all task counts are zero then *terminated*(j) is true.

Figure 4.1 below illustrates termination detection and update processing at the home site with incremental updates.

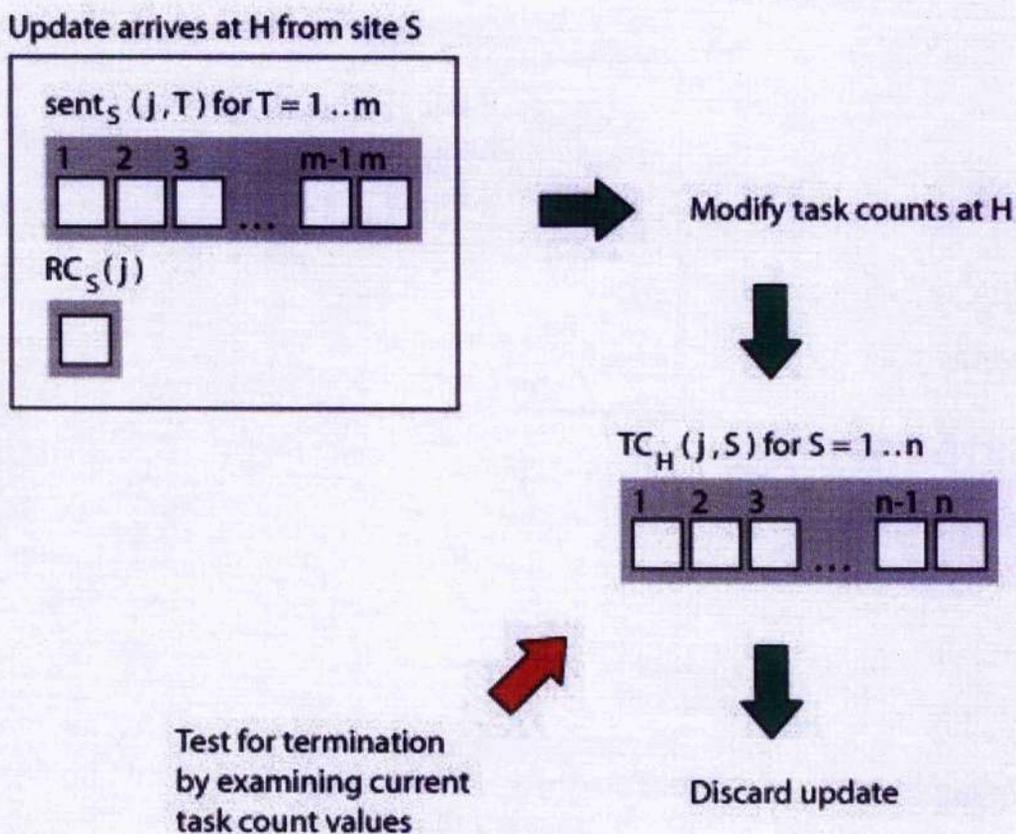


Figure 4.1 - Incremental Updates and Termination Detection

4.2.2.2.2 Standalone Updates

An implementation where *update* messages contain running totals for the *sent* and *received/completed* counts does not require delivery order constraints on the communications channel delivering update messages from a site *T* to the home site *H*. Termination for a job *j* can be correctly determined at *H* at any time by processing any (but only) one *update* message from each site *T*. However only by processing the last update for the job *j* sent by each site *T* will *terminated(j)* ever be determined to be true.

This abstract description translates directly into an implementation. The home site H for a job j maintains a log of all updates for j sent to H . The calculation of $terminated(j)$ involves choosing any update message from the log for each sending site and combining the *sent* and *received/completed* values in each of the updates to generate the task counts for j at all sites S . When task count is zero for all S then $terminated(j)$ is true.

Figure 4.2 below illustrates termination detection at the home site with standalone updates.

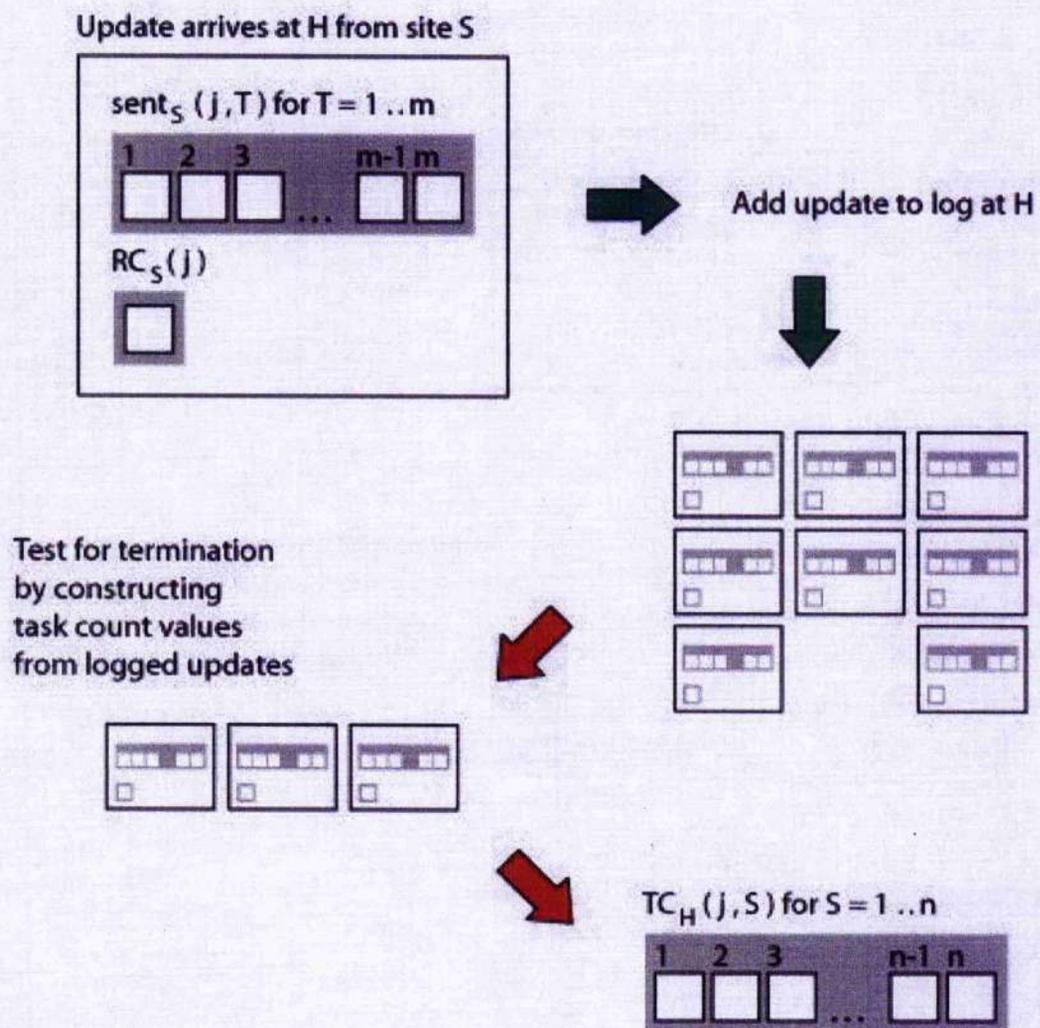


Figure 4.2 - Standalone Updates and Termination Detection

Clearly this scheme can be optimised to reduce the size of the update log. The only set of update messages that can result in $terminated(j)$ being true is that which contains the last update sent from each site T . This means the log can be cleaned periodically to ensure that it contains only one update from each site T and that that update is the last update sent by T . In the most eager case an update from a site S will only be added to the log if it was sent after the log's current update message entry for j at S . No further work is required to implement this optimisation. The count values in standalone *update* messages are monotonically increasing. Given any two updates sent from a site S , the home site H can identify the last sent (from S) by examining the individual counts in the message. The message with a higher value for any count is the last of the two messages sent from the site S .

4.2.2.3 Preparing Update Messages

As the distributed computation proceeds, a site must maintain a record of the number of tasks of each job j sent to each other site. At site S , for each site T the value $counts_S(j, T)$ must be kept current as each task is sent since it is assumed this value cannot be determined at *update* send time by examining the local site.

The same incremental approach is not necessarily required for the calculation of the value $RC_S(j)$; the number of tasks of the job j received and completed at S . There are two options for the calculation of this value. The first is the incremental approach where a site individually tracks each received task of a job j received and counts as these tasks complete. In such a scheme the site knows at any time the number of tasks of j received and completed at S . To achieve this, a site must maintain meta-data for each received task so that when such a task completes it can be identified as a received task and the appropriate RC count incremented.

The second approach is to allow the site to delay the calculation of $RC_S(j)$ until it is needed. The site can delay the calculation of $RC_S(j)$ until the point at which the site S decides to send an *update* to the home site H . The following section describes the role of idleness detection in both deciding when to send an *update* message from a site S and in calculating $RC_S(j)$ at that point.

4.2.2.4 Idleness Detection

Idleness detection is a key factor in the TB implementation presented in this thesis. Idleness is stronger than either of the conditions necessary for an update message to be sent: an update for a job j at site S can be sent when there exists no locally spawned tasks of j at S or when there exists at least one received and uncompleted task. When a job is idle at a site then all tasks of the job received (since the last update was sent) are completed, and there are no locally spawned tasks of the job still running.

In using the local idle state as a trigger for sending update messages, an implementation is spared the task of individually tracking the received tasks of a job and distinguishing them from locally spawned tasks. A secondary benefit of using idleness detection is in optimising the number of messages sent. The goal of the algorithm is (for the home site H) to detect global termination of a job j . This condition is only satisfied when the job is idle at all sites (and there are no messages in transit) and so for site S to send an update message when $idle_S(j)$ is not true, is clearly redundant⁷.

⁷ An argument for the correlation between the idle state and achieving the optimum message complexity for termination detection has already been presented.

There are two ways to detect idleness of a job at a site S . The first involves individually accounting for each task of j locally spawned at S and all tasks of j completed at S and comparing these counts to the number of tasks of j received at S . When the job j is idle at site S the count of all tasks completed at S minus the count of locally spawned tasks equals the number of tasks of j received at S . Two values must be maintained at a site S to allow for idleness detection in this manner. These are $count_S(j, S)$ which is the (locally spawned) task count for job j at S and $received_S(j)$ which is the number of tasks of job j received at S since j was last idle at S . When a task of j is spawned locally at S the value $count_S(j, S)$ is incremented. If a task completes the value is decremented. The job j is idle at site S when the value $count_S(j, S)$ equals the negative of $received_S(j)$. At this point (when job j is idle at site S) the value $received_S(j)$ is equal to $RC_S(j)$ since all of the jobs of j received at S have completed.

The second is to perform a sweep of the entire site S scanning for tasks of j . The details of such a scheme are completely implementation specific but the site must still maintain enough information for the calculation of $RC_S(j)$ when job j becomes idle at S . The site must either maintain the value $received_S(j)$ or $count_S(j, S)$ (unlike above where both are required) and calculate $RC_S(j)$ as described for the first scheme above.

4.2.3 An Example Task Balancing Implementation

Figure 4.3 below demonstrates a Task Balancing implementation with the following properties:

- Update messages are incremental.

- The idle state is used to trigger update sending. While the mechanism used by each site to detect the idle state is not demonstrated in Figure 4.3, the reader should assume a simple task counting mechanism where:

- the home site A uses the task count value for A , $TC_A(j,A)$, to maintain a count of the number of tasks of the job j held locally at site A .
- at any other site N , the value count $count_N(j,N)$ is used to store a count of the number of tasks of the job j held locally at site A .

In either case, the local task count at a site is initialised with the value one on receipt (or creation, in the case of the *home* site) of the first task of j . The task count at a site is incremented on the creation of local tasks of j and on receipt of tasks from remote sites. The task count at a site is decremented when a local task of j completes. When the local task for job j reaches zero at a site, then that site is idle for j .

- Each site N (that is not the *home* site) maintains the value, $received_N(j)$, which records the number of tasks of the job j received at that site since an update was last sent.

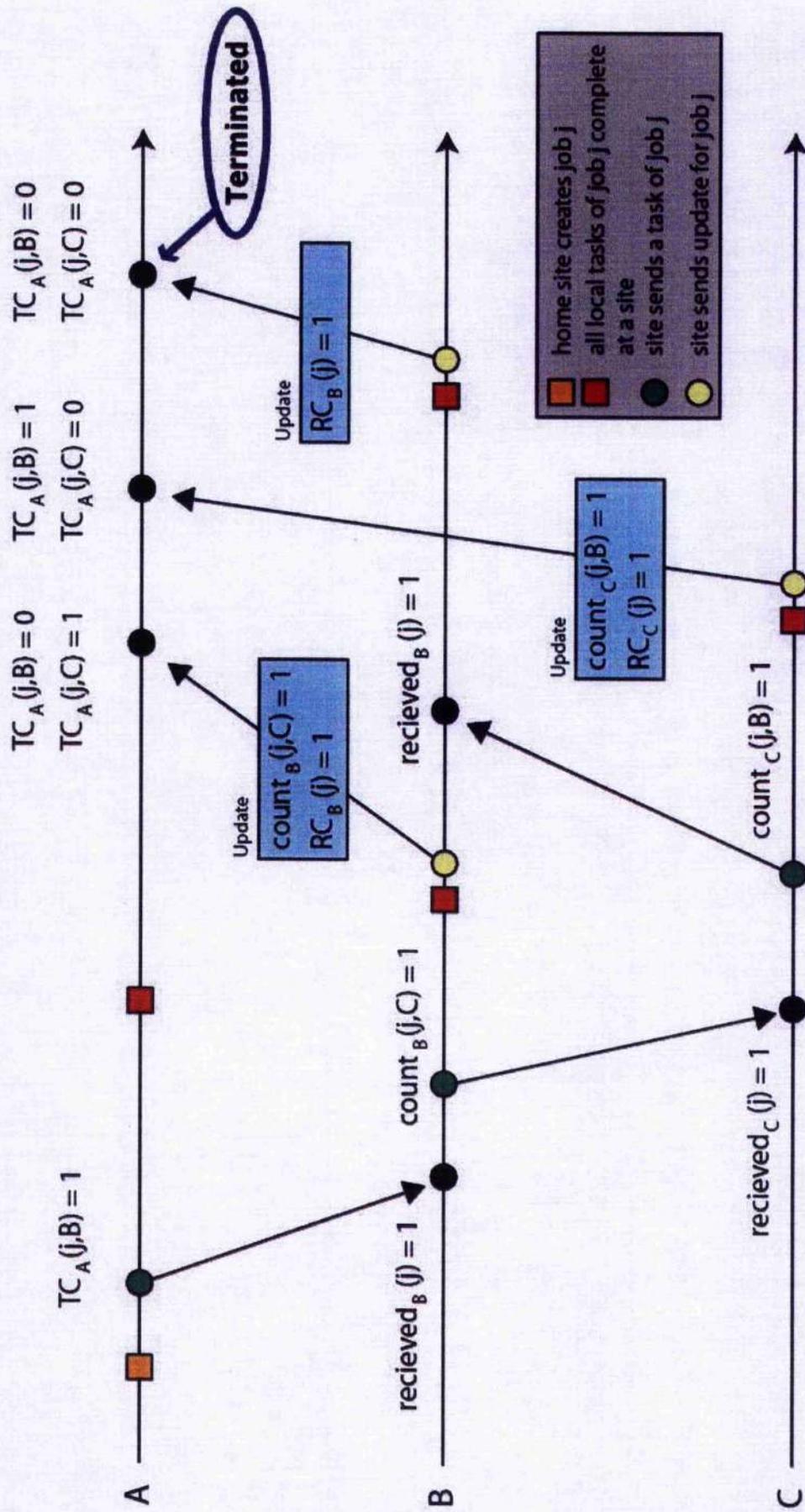


Figure 4.3 - A Task Balancing Implementation

The Task Balancing implementation shown in Figure 4.3 above illustrates termination detection for a job created at site *A* and distributed across two further sites, *B* and *C*. The home site *A* creates the job *j* and sends a task to site *B*. Site *B* then sends a task of *j* to site *C*. *B* then becomes idle for *j* and sends an update to the home site *A*. At the same time, site *C* sends a task of *j* to site *B*. Thus, site *B* changes from the passive to the active state while the update is still in-flight.

On receipt of the update message from *B* at *A*, site *A* is idle for *j* but after processing the update message $TC_A(j,C)=1$ and thus the terminated condition does not hold for *j*.

In the example shown site *C* and then site *B* become idle for *j* and updates are sent to *A*. The timeline shows that the update from *C* arrives first such that $TC_A(j,C)=0$ and $TC_A(j,B)=1$. The update from *B* arrives second and the termination condition holds for *j* since $TC_A(j,C)=0$ and $TC_A(j,B)=0$ and (implicitly) $TC_A(j,A)=0$. Note that if the delivery order of these two update messages is reversed the termination condition still does not hold until both messages have been received (and processed). If the second update form *B* were to arrive before the update form *C* then $TC_A(j,C)=1$ and $TC_A(j,B)=-1$. The termination condition is that $TC_A(j,N)=0$ for all sites *N* and thus *j* has not terminated.

4.3 Why Use TB for DGC Implementation?

In answering this question it is necessary to first present a number of properties of the TB algorithm. These properties are phrased in terms of the standard process-centric terminology (that is, control messages, basic messages and active and passive states) where appropriate to give the reader some intuition as to how they compare to other DTAs, although no explicit comparison is given here.

- The algorithm is asynchronous.

- While termination detection is centralised at the home site for a particular job, this site is not involved when basic messages are sent between sites.
- No message overhead is incurred by the sending or receiving site when a basic message is sent to (received at) a site for the first time, i.e. when a site joins the basic computation. A site takes no further part in distributed termination detection once it is no longer part of the basic computation, and has sent its final update message.
- Termination can be determined at the home site immediately on receipt of the next control message from each site after termination occurs.
- Control messages (updates) for a job are sent directly to the home site for the job and are not passed to any other site.
- The algorithm allows for opportunistic policies controlling when control messages are sent and when they are processed at the receiving (home) site.
- The sending of control messages can be delayed until a site is passive (idle). In such a scheme the message complexity is therefore proportional to the number times each site becomes active, and not directly related to the total number of basic messages. More importantly the algorithm allows a site to avoid sending a control message for a job j when the sending site itself would prevent j from terminating.

Repeating the list of desirable properties for a DGC from Chapter 1: a DGC should be safe, complete, non-disruptive, incremental, non-blocking, independent and scalable. TB is a suitable choice for use in DGC implementation since the algorithm demonstrates each of the properties as follows:

- Safe - A job is not deemed to have terminated until all tasks have completed and there are no tasks in-flight.
- Complete - The algorithm detects the terminated state for any job that does terminate.
- Non-disruptive - Clear bounds can be placed on the computation at a site required for the maintenance of the TB task count structures, on the construction of update messages (at a site that is not the home site) and on testing for termination (at the home site).
- Incremental - There are three main components to the computation involved in the implementation of TB: maintenance of the TB task count structures at each site (including the home site); determining when a site (that is not the home site) should send an update message; termination detection at the home site. The first two components do not require that a site has global knowledge, and the third, (while constructing a globally consistent view of the state of the job; i.e. is the job terminated or not?) requires only information already held at the home site (i.e. updates that have already been received).
- Non-blocking - Update messages are sent asynchronously and the home site does not need to synchronise with any remote site to test for termination.
- Independent - The algorithm is independent in a number of ways. Firstly a site can send a basic message without sending a control message, and more importantly without waiting for a reply to the control message. Secondly, the home site for a job j can safely determine whether or not j has terminated by processing only the updates that it had already received. Thirdly, with purely local information a site can determine when an update should be sent.

- Scalable - Given the above properties, the algorithm is clearly scalable in terms of execution time. However the space requirements scale proportionally to the number of sites to which a site has sent tasks and received tasks from. That is, the space requirements of the algorithm depend on the number of sites that hold tasks of the job at some point, and not on the size of the distributed system as a whole. The algorithm's bandwidth requirements are directly proportional to the size of the count structures maintained at sites.

As will be shown later, the TB algorithm provides additional information that the distributed garbage collector implementations use opportunistically. For instance, in DMOS it is necessary for the home site of a job to be able to calculate the set of sites that have held a task of the job at some point before it became terminated. The task count structures at the TB home site can be used to calculate this set without sending additional messages.

4.4 Summary

The contribution of this chapter is to explain (for the first time) the fundamental principles behind the Task Balancing algorithm independently from any particular implementation. The individual issues relating to the implementation of the algorithm are then presented independently.

Each of the distributed garbage collectors described in this thesis incorporates an implementation of the TB algorithm. These implementations all use the idle state as a trigger for update sending but differ in the particular mechanisms used to achieve idleness detection. Each implementation makes opportunistic use of the TB data structures and site information.

The contribution here lies in the definition of the fundamental structure of the TB algorithm, in the definition of the predicates controlling the sending of update messages and in the identification of the implementation choices for the TB algorithm.

5 Separating Distributed and Local Collection

As discussed in Chapter 1, Blackburn et al. described a structured approach to the design of distributed garbage collectors ([BHM+01]). Their approach centres on the derivation of a distributed collector through the construction of one or mappings from a centralised garbage collector to a distributed termination detection algorithm. This structured approach to DGC derivation is known as the mapping methodology. From [BHM+01],

“The derivation of distributed garbage collectors is structured through the mapping of distributed termination algorithms onto known centralized collection schemes as follows:

- *Select or derive a distributed termination algorithm that is proven correct.*
- *Prove safety, and maybe some other properties, of the centralised garbage collector.*
- *Define an object reclamation mapping, from the centralised garbage collector to the distributed termination algorithm.*
- *Prove that termination is equivalent to the eventual reclamation of objects.*

The methodology starts by making a centralised collector concurrent and then mapping a DTA onto the resultant collector to provide a distributed garbage collection scheme.”

In distributing a centralised garbage collector, state becomes distributed. In using the methodology, the distribution is structured such that globally stable properties of the

distributed shared state are identified so that they may be captured by a DTA mapping.

This chapter describes an extension to the mapping methodology (previously published in [NMM+03]) that minimises the constraints placed on a site. This is achieved by mapping a DTA onto any (non-distributed) garbage collection scheme, to derive a global distributed collector while leaving a site free to implement any local collection scheme. Each mapping is used to define a set of rules that must be obeyed by each participant (site) in the distributed collection scheme. These are the club rules for the distributed collector. Each rule defines as a set of actions that must be carried out at a site corresponding to some event that occurs at that site. An example of an event at a site is the copying of a reference to an object. The corresponding action might be to increment a local count of the number of references to the object.

The club rules define a boundary between the distributed work of the distributed garbage collector (that is necessary to identify distributed garbage) and the purely local work of space reclamation at a site. The participating collectors are free to perform any local actions as long as they preserve the club rules.

The extension to the methodology concentrates on identifying events at a site that correspond to operations on the distributed shared state. The club rules specify the operations on the DTA implementations that correspond to these events and any other actions that are required to maintain the distributed shared state. In this way the club rules define how each site drives the DTA implementation in order that the globally stable properties may be detected. The benefit of such a structured approach to distributed collector implementation is the clear distinction (provided by the club

rules) between providing safety via termination detection (distributed work) and space reclamation (local work).

In general a particular rule can be considered as belonging to one of two sets:

1. The set of rules required to detect globally stable properties of the shared state and to maintain the distributed state. These facilitate the identification of distributed garbage.
2. The set of rules that specify how and when space is reclaimed. These rules specify a local collection scheme for each site, for instance a local mark-sweep or semi-space regime.

Where the club rules for a derived collector consist of both types of rule then the derived collector is said to have homogeneous local collection behaviour. That is, all sites implement the same (homogeneous) local collection mechanisms and it is the club rules that specify when and how space is reclaimed locally at a site. For instance, the club rules might define a local semi-space collection mechanism to be implemented by each site. Thus, traditional distributed collection schemes are classified as exhibiting homogeneous local collection behaviour.

Where the club rules consist of rules only from the first set (i.e. they specify only how to detect the globally stable properties and how to maintain the distributed shared state) then the derived collector is said to have heterogeneous local collection behaviour. That is, sites are free to implement any (heterogeneous) local space reclamation scheme they want. In heterogeneous collector, sites exhibit independent local collection behaviour. Each site is free to carry out local garbage collection (involving no other sites) any time it wants and in any way that it wants. Local collection is thus purely a matter of local policy. One site might implement a local

generational collector while another might implement a local compacting mark-sweep collector.

The work described in this chapter has three goals:

- To identify the boundary between work in the distributed collector that may be carried out purely locally and that which constitutes the DTA implementation for determining globally stable properties of the distributed state. This can be thought of as separating distributed and local collection work in the distributed garbage collector. Of key importance is identifying where this boundary lies and how it can be varied to allow sites of the distributed system more freedom in how they behave.
- To identify a set of rules that define this boundary, implement the DTA mapping, provide sites with an interface to the distributed collector and allow for the independent reclamation of objects at individual sites.
- To allow sites to carry out local collection in any manner they choose (providing they obey the rules) and at any chosen rate.

Three new mappings are presented in this chapter. They constitute the club rules for six distinct distributed collection schemes. For each mapping, two sets of club rules are defined. The first set of club rules define DGC with homogeneous local collection behaviour, whereby the local collection behaviour at a site is dictated by the club rules⁸. The second set of club rules defines a DGC with heterogeneous local collection behaviour. This second set of club rules provides an interface between the distributed garbage collector and the local collector at a site allowing for safe,

⁸ In this case the rules define a traditional distributed garbage collector that dictates how and when local space is reclaimed.

independent local collection using any suitable local collection scheme. The scheme is thus heterogeneous because sites can implement arbitrary local collectors, as long as the club rules are maintained.

The above mappings use implementations of the Task Balancing DTA (Chapter 4) in the ProcessBase distributed cache (Chapter 3) which is itself an instantiation of the system model (Chapter 1).

5.1 Forming the Club Rules

As described in [BHM+01], the mapping between distributed garbage collection (DGC) and distributed termination (DT) is not automatic and depending on the characteristics of the garbage collection (GC) algorithm may be more (or less) complex. There are two considerations in particular that must be handled. The first consideration is that in a distributed system, work proceeds concurrently and asynchronously at different sites. This is essentially an additional consequence of the state partitioning: other sites may change their part of the partitioned state separately and asynchronously from any particular site's part of the state. Secondly, a distributed algorithm cannot instantaneously and atomically update globally shared state. Thus the distribution and the partitioning of the shared state is designed such that globally stable properties may be identified within the distributed shared state. A mapping to a distributed termination algorithm (DTA) is constructed to detect these globally stable properties. This is the DTA mapping process.

The club rules at each site are thus the implementation of the DTA mapping, the distributed collection actions for each site and either the local collection actions (for a homogeneous scheme) or the interface for the local collection mechanism (for a heterogeneous scheme). Using new mappings it is shown how the club rules are constructed for six examples: distributed mark-sweep, distributed generational and

distributed reference counting collectors accommodating both homogeneous and heterogeneous local collectors.

5.2 Distributed Mark-Sweep Collection

A typical mark-sweep scheme [McC60], be it stop-the-world or concurrent, is composed of two phases; a mark phase followed by a sweep. In the mark phase the transitive closure of the graph of objects is traced from a root set marking reachable objects. A sweep of the whole space is then required to identify unreachable (unmarked) objects. Often collectors take the opportunity during the sweep phase to unmark reachable objects and relocate objects to compact the free space.

Marking based collectors traverse the object graph, marking any object encountered, until the point where every object encountered is already marked. That is, when each reference in each marked object refers to a marked object.

In a mark-sweep collector the completion of the mark phase corresponds to the set of references in marked objects that refer to unmarked objects being empty. In a distributed context, the object graph is distributed, and the empty set (of references in marked objects that refer to unmarked objects) represents is a globally stable property of the distributed state. Thus a DTA mapping is required to identify when this set is empty, and hence identify the completion of the mark phase. The club rules consist of an implementation of the DTA, the marking actions and the actions necessary to perform the sweep phase at each site.

In the implementations presented here each site has a distinguished root object from which all locally reachable objects may be found. The root set for a distributed collection is determined by the union of these local roots. Distributed mark-sweep

garbage collection proceeds as follows. Garbage collection starts⁹ by sending a *mark* message to all sites. Each site then traces the local object graph from its root, marking all reachable local objects. If a distributed address (DA) is encountered during the tracing a message is sent, to the site holding the object referenced by the DA, instructing that site to mark all objects reachable from the DA. In this way objects that are referred to remotely are also marked during this phase. The initiating site detects when marking is complete (DTA termination) at which point it sends a *sweep* message to all sites. On receipt of the sweep message the local site in the homogeneous scheme identifies and collects unmarked (unreachable objects) immediately. In the heterogeneous case, the sweep action may delegate the collection of objects to a local collector. In either case, the local sites inform the initiating site when the sweep is complete to allow subsequent distributed collections. This mechanism is similar to the description of a generic distributed mark-sweep from [PS95].

In terms of a mapping to the TB DTA there is one job that corresponds to the distributed marking phase, called the distributed marking job (DMJ). The site that initiates the job is called the *DMJ home* site. A job consists of two types of tasks, the first is called a Root Marking Task (RMT) and the second is called a Distributed Address Marking Task (DAMT). When the *DMJ home* site starts a job, it sends a RMT to every site (including itself). If the RMT at a site encounters a DA, a DAMT is sent to the site holding the object. Similarly if a DAMT at a site encounters a DA,

⁹ To avoid global synchronisation it is assumed, for the moment, that a single predetermined site is charged with the responsibility for starting collection. While the process of deciding which site can start a collection is distributed and possibly non-trivial, it is orthogonal to the actions of the distributed collector.

a DAMT is sent to the site holding the object. Note that the TB sent and received/completed task counts for both types of task may be combined. DAMT's and RMT's both execute in the same way, the only difference is that the start point for tracing in a DAMT is explicit (i.e. any remotely referenced object) while in an RMT it is implicit (i.e. each of the local roots).

Both task types trace the local graph of objects from a given start point. Each object at a site has a distributed mark bit (DMB) associated with it. If an object is traced by a task its DMB is set (to indicate that it is marked) and then it is scanned for references. Both types of task complete when they have fully traced the local graph from their specified start point.

Unlike Hughes' collector [Hug85], instantaneous message passing is not assumed and so the system must safely allow for DAs that are in-flight between sites when marking begins. The following example demonstrates how such in-flight DAs can cause problems. A site S holds the only reference in the system to an object *O* on site R. S sends the DA of *O* to a site T and immediately deletes its own copy. Distributed collection may begin at S and T before the message containing the DA arrives at T and the distributed marking mechanism will incorrectly determine that *O* was unreferenced.

The above problem is solved by having sites record any DA sent to a remote site in a table called the *in-flight* table. Each site is required to send an acknowledgement to the sender site on receipt of any message containing a DA. On receiving the acknowledgement a site can then remove the *in-flight* table entries for the DAs in the original message. All entries in the *in-flight* table are treated as roots for the distributed collection. This is safe but conservative.

5.2.1 Club Rules for Distributed Mark-Sweep

The following describes the club rules that are generic to both the homogeneous and heterogeneous distributed mark-sweep collectors using an incremental-update-style TB implementation. For site S that is not the DMJ *home* site:

- S maintains data structures for recording TB *sent* counts. These counts are maintained as follows¹⁰. Each site S maintains a *sent* count array with an element for each site T recording $sent_s(j,T)$. When a task is sent to site T , $sent_s(j,T)$ is incremented.
- S maintains the value *received* which records the number of tasks of the DMJ received since an update was last sent. This value is incremented on receipt of each RMT or DAMT.
- S maintains an *in-flight* table with an entry per message where all DAs sent in messages to remote sites are recorded.
- On receipt of a message, from a site T , containing DAs, S sends an acknowledgment message back to T . The acknowledgment contains the DAs that were sent to S from T .
- When S receives an acknowledgement it removes the per-message *in-flight* table entries for each DA in the message.
- Implementation of an RMT at S is as follows. Mutator activity is paused at S during the execution of an RMT. An RMT traces the object graph from the distinguished local root at S marking reachable objects using their DMB. For each DA found during the trace, a DAMT is generated and sent to the site as

¹⁰ RMT and DAMT task counts may be combined here.

determined by the site address component of the DA¹¹. A DAMT is also sent for each DA in the *in-flight table*.

- Implementation of a DAMT is as follows. Mutator activity is paused at S during the execution of a DAMT. A DAMT message contains the DA of an object that is remotely referenced. The local object graph is traced from this object marking reachable objects using their DMB. For each DA found during this trace, a DAMT is sent to the remote site (as determined by the site address component of the DA).
- When a DAMT is sent from S to a remote site T the *sent* count for T at S, $sent_S(j,T)$, is incremented.
- An update is sent from the site S to the DMJ home site when S is idle for the DMJ. RMTs and DAMTs are executed one at a time at S, therefore S is idle for the DMJ when there are no received tasks waiting to be executed. At this point the value RC_S for the update is equal to the current *received* value at S. When the update is sent, *received* is set to zero and for each site T, $sent_S(j,T)$ is set to zero.
- On receipt of a *sweep* message the behaviour of a local site is implementation dependent. The actions taken for the homogeneous and heterogeneous collectors are described later. However, all implementations share some common aspects: mutator activity at S is paused while the sweep executes; during the sweep phase all local objects marked by tasks are unmarked; and

¹¹ An obvious optimisation is to ensure that only one DAMT task is sent for each distinct remote DA at a site.

when the local sweep has completed a *sweep acknowledgment* is sent to the *home* site.

- To allow the interleaving of mark task execution and mutator activity at S during distributed collection new objects have their DMB set on creation. This guarantees safety, since new objects are guaranteed to survive at least the distributed collection cycle in which they were created. To avoid the need for global synchronisation on distributed collection start-up all sites must always assume that marking is in progress, and thus always mark new objects.
- Messages containing tasks, updates and termination notification are sent via the inter-site communications channels and as such are subject to site-to-site ordered delivery. Tasks are executed and *update* messages processed in strict order of delivery¹².

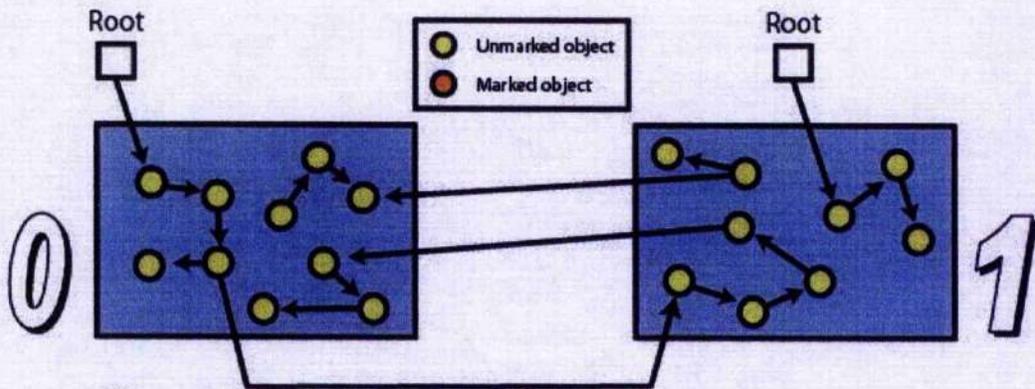
For the DMJ *home* site H the club rules are all of the above and:

- A distributed collection is started by sending an RMT to each site including this site.
- The home site H maintains a task count array with an element for each site T holding the value $count(j,T)$. On receipt of an *update* message from a site S, RC_S is deducted from $count(j,S)$ and for each site T, $sent_S(j,T)$ is added to $count(j,T)$. When $\forall T.count(j,T) = 0$ the termination condition holds and distributed marking has completed.

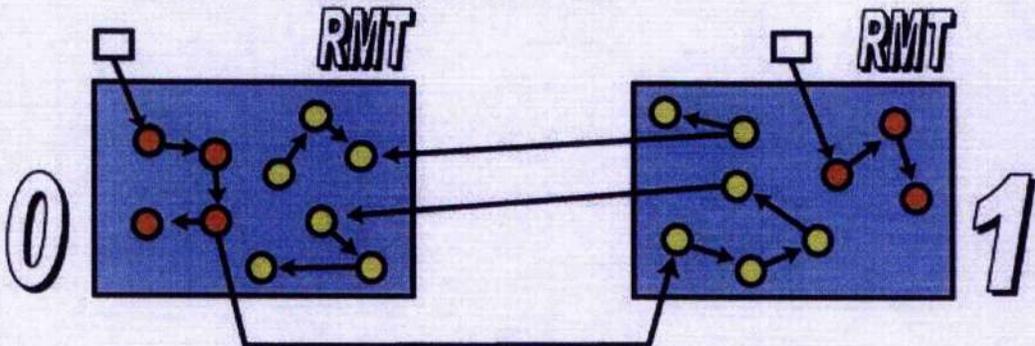
¹² As a consequence, the home site H can ignore the $sent_S(j,H)$ value in an update from a site S, since the tasks to which this count relates have already completed and been balanced.

- The sweep phase is synchronised across all sites by sending a *sweep* message to each site, on DMJ termination detection, and waiting for all sites to reply. Having received a *sweep acknowledgment* message from all sites, the home site is free to start the next distributed collection.

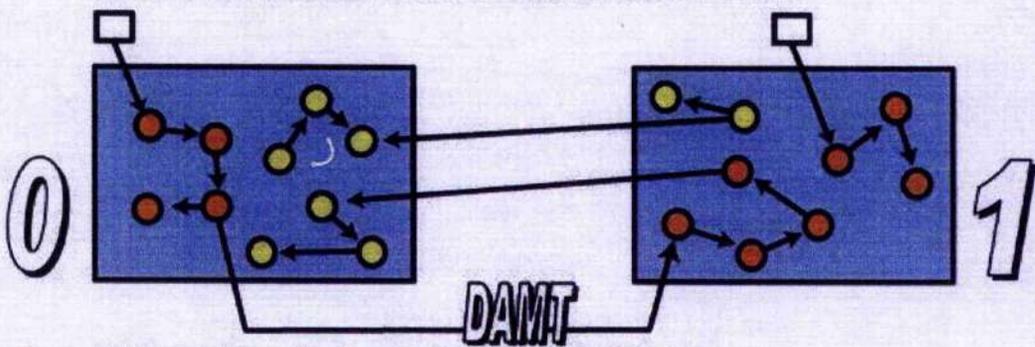
Figure 5.1 below demonstrates the club rules for the mark phase if the distributed mark-sweep collector. The system consists of a rooted object graph distributed over two sites, site 0 and site 1.



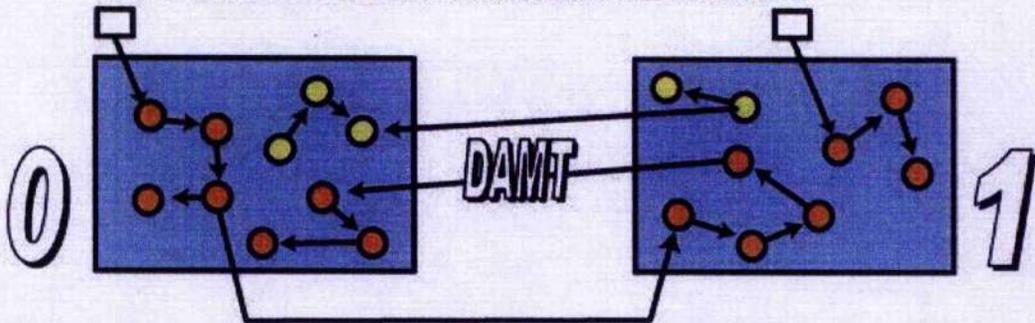
1. Site zero initiates a garbage collection by sending an RMT to site 1 and executing RMT locally.



2. While marking due to execution of RMT at site 0 a remote reference is discovered and a DAMT is sent to site 1.



3. While marking due to execution of DAMT at site 1 a remote reference is discovered and a DAMT is sent to site 0.



4. Marking is now complete and the home site will detect termination of the distributed marking job. A sweep message is then sent to site 1.

Figure 5.1 – Distributed Marking Demonstration

5.2.2 Club Rules for a Homogeneous Mark-Sweep Collector

A homogeneous mark-sweep collector implements the generic club rules as described above plus the sweep phase as follows.

On receipt of a *sweep* message, a site pauses mutator activity and scans the whole of its local cache. Unmarked objects are reclaimed at this point. During the sweep phase the $DA_{\text{sym}} \rightarrow CA$ address translation table entries of unmarked objects are removed. On completion of its local sweep a site sends a *sweep acknowledgement* message to H.

This description deliberately omits discussion of issues such as compaction of the local storage space and free-list maintenance since these are orthogonal to the distributed collector. However during its sweep all objects in the local cache are unmarked in preparation for the next distributed collection cycle. This may be achieved by flipping the meaning of the DMB for a site and thus unmarking all objects simultaneously.

5.2.3 Separating Local and Distributed Collection

The homogeneous distributed mark-sweep scheme restricts the reclamation of objects at a site to the sweep phase following termination of distributed marking. This is clearly unacceptable as many local collections may be required between distributed collections. Separating local and distributed collection allows both the timing of the collections to be independent and also the nature of each of the local collectors and the distributed collector to vary. To this end a site must be provided with a set of local roots that will allow for safe independent local collection. This local root set consists of the distinguished local root plus all local objects referenced by a remote site. The latter part of this root set is called the *distributed root set*. A

safe, but conservative, view of this root set is already implemented at each site in the $DA_{sym} \rightarrow CA$ tables. On DA export an entry is added to the $DA_{sym} \rightarrow CA$ table at a site and safe local collection is possible if each entry is treated as a local root.

The club rules maintain the *distributed root set* at a site by identifying those entries in the $DA_{sym} \rightarrow CA$ table that represent objects still referenced by a remote site. The local collections may take place autonomously from distributed collections by using the site's local root set, thereby separating the implementation of safety from the reclamation of space at a site.

No particular stand is taken on the suitability, desirability or efficiency of independent (and possibly heterogeneous) local collection as compared to the homogeneous scheme. The aim here is only to show how, within the confines of a single DTA to GC mapping, a number of collectors may be implemented. In allowing a site to enact policies of its choosing, regarding how and when collection work is carried out, the potential for performance improvement is not stifled by the distributed collection mechanisms.

5.2.3.1 Club Rules for a Heterogeneous Mark-Sweep Collector

The heterogeneous collection scheme implements the generic mark-sweep club rules as described above. Here the implementation of the local sweep is described along with the additional rules that are necessary for *distributed root set* maintenance and heterogeneous local collection support. For all sites S:

- Each $DA_{sym} \rightarrow CA$ table entry at S has two flags associated with it. The first is the *distributed root* flag (DRF). $DA_{sym} \rightarrow CA$ table entries for which the DRF is set represent roots of reachability at S. On DA export a $DA_{sym} \rightarrow CA$ table entry is created for this DA with its DRF set. The local root set for local

collection at S contains the distinguished root object and each object with a $DA_{sym} \rightarrow CA$ table entry with its DRF set. The second flag is the *DAMT marked* flag. The purpose of this flag is explained in the next rule. The *DAMT marked* flag is set in all new table entries. The two flags are used in conjunction to clean the *distributed root set* entries from the $DA_{sym} \rightarrow CA$ table.

- On receipt of a DAMT for a DA the *DAMT marked* flag is set in the corresponding $DA_{sym} \rightarrow CA$ table entry at S. After setting the *DAMT marked* flag the DAMT executes as specified in the generic rules.
- On termination of a distributed marking phase the $DA_{sym} \rightarrow CA$ table entries that have their *DAMT marked* flag set represent the remotely referenced objects at S. On receipt of a *sweep* message S first pauses mutator activity and then scans its $DA_{sym} \rightarrow CA$ table to reconstruct the *distributed root set* by using the *DAMT marked* flags. This is done by setting the DRF in each entry that has its *DAMT marked* flag set, and clearing the *distributed root* flag for all entries that do not have their *DAMT marked* flag set. During the scan all *DAMT marked* flags are cleared.
- After the $DA_{sym} \rightarrow CA$ table scan the meaning of the DMB is flipped effectively unmarking all local objects and ensuring that they are unmarked before the start of the next distributed mark phase. Here the benefit can be seen with such an approach to unmarking since there is no need to scan the local cache to unmark objects. After its sweep of the distributed root set a site sends a *sweep acknowledgment* to the distributed marking *home* site.

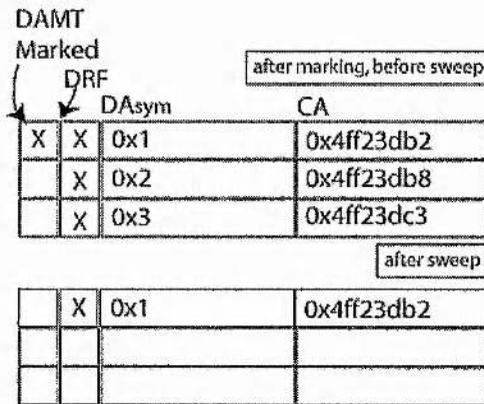


Figure 5.2 - Cleaning DA_{sym} → CA Table Entries

Following the local sweep, each entry in a site's DA_{sym} → CA table with its DRF set constitutes a root of reachability for local collection. To allow for the interleaving of local mutator activity and the execution of marking tasks all new DA_{sym} → CA table entries must have their DRFs set. Figure 5.2 shows the state of a DA_{sym} → CA immediately following a distributed marking phase and the same table after the sweep when only those entries with their *DAMT marked* flag set are maintained. Between distributed collections there may be any number of local collections based on reachability from the *distributed root set*.

5.2.4 Local Collection in the Heterogeneous System

Two local collection mechanisms are now described for sites in the heterogeneous system. In both cases the club rules have a minimal impact on the behaviour of the local collectors. The local collectors are charged with updating both a site's DA_{sym} → CA and DA → CA address translation table entries if objects are moved and with removing entries for reclaimed objects (that have a DA). Recall that the local collectors treat *remote resident* objects as local objects thus ensuring that there is no interference between local collection and the object duplication policy (as

described in Chapter 3). These are identified by having entries in the $DA \rightarrow CA$ table.

5.2.4.1 A Local Semi-Space Copying Collector

As a first example of independent local collection, a non-incremental, semi-space, copying collector implementation based on Cheney's list compaction algorithm (from [Che70]) is described.

The local cache is split into two equally sized areas (semi-spaces). While mutator activity is ongoing all objects reside in one area and any new objects are created in this area; during this phase the other area is unused. The idea of semi-space collection is to trace the object graph copying reachable objects from one space to the other. When an object is copied, the new address of the copied object (a forwarding pointer) is written into the original object. As tracing proceeds each copied object is scanned for references¹³. For references to copied objects the forwarding pointer is used to update the reference, otherwise the object is copied and the reference updated. Typically the free space is compacted as objects are copied, thus yielding a single contiguous area of unallocated space and allowing for a simple allocation mechanism.

The local collector must update the appropriate address translation table for any copied object that has a DA (that is, the $DA_{sym} \rightarrow CA$ table for local objects or the $DA \rightarrow CA$ table for remote resident objects). After all reachable objects have been copied both of the address translation tables are scanned. Each entry that references an object with a forwarding pointer (i.e. a copied object) is updated and those entries

¹³ The existence of a forwarding pointer indicates that an object has been copied.

for objects with no forwarding address, are removed. Mutator activity is then resumed, now using the space to which objects were copied.

When collection begins each object in the local root set is copied. Copied objects are then sequentially scanned for references, resulting in a breadth first traversal of the local object graph.

5.2.4.2 A Local Mark-Sweep Collector

As a second example of independent local collection a stop-the-world, mark-compact local collector is described. This collector is based on the Lockwood Morris algorithm [LM78] for compacting the used space.

A local garbage collection can be performed at any time. Local collection proceeds as follows. Mutator activity at the local site is first stopped and then the object graph is traced from the local root set. Each object has an associated local mark bit (LMB). The LMB is set during the marking phase for each object that is traced.

When marking is complete the heap is scanned and compacted, clearing LMBs and updating local references and address translation table entries (in both the $DA_{sym} \rightarrow CA$ and $DA \rightarrow CA$ tables). During the scan/compact phase the address translation table entries of unmarked objects are removed.

5.2.5 Discussion

The new TB to mark-sweep mapping minimises the number of tasks by only spawning tasks for inter-site references. This contrasts with the DM-S mapping in [BHM+01] where a task is mapped to the marking of an individual object and a task is spawned for each reference. This new mapping also benefits from not having to balance locally spawned tasks since there are none.

Separation of local and distributed collection work enables flexibility in local collector behaviour. Sites are free to implement any local collection scheme, and are constrained only by having to implement the club rules. Importantly, sites can perform as many local collections as determined by local policy, independently of global collection.

5.3 Distributed Generational Collection

A generational collection scheme partitions the address space into two or more parts (generations) and places objects in generations based on their age. All objects are created in the youngest (zeroth) generation. On some threshold of collections (age) an object, if it is not garbage, is promoted from its current generation to the next older generation. The effect of age based promotion is that the zeroth generation acts as a nursery and older objects are found in older generations. Generations are collected in age order, starting at the youngest generation, allowing the collector to reclaim unused space from one generation without having to trace the entire space. The intuition behind the generational approach is that the efficiency of the collector can be improved by collecting the younger generations more frequently since most objects become garbage at a young age [Ung84].

Here a generic stop-the-world generational collector, based on Lieberman and Hewitt's generational collector [LH83] is described. To collect a generation, a data structure known as a remembered set (remset) is used to record references into the generation from objects in other generations. To maintain these remsets two mechanisms are used; a write barrier to catch references from older to younger generations and age ordering of generation collection to discover references from younger to older generations. During the collection of a generation, any reference to an object in another generation will be discovered. Remset entries for such

references are added at the point that the references are discovered and thus, as a consequence of collecting generations in order from younger to older, when we come to collect a generation we can be sure that its remset contains an entry for each reference from a younger generation. The write barrier is required to trap and add remset entries for references from older to younger generations. The intuition here is that the work done by the write barrier is reduced since references from older to younger generations are less common than references from younger to older references.

The transitive closure for a generation G_i , which constitutes the set of live objects in G_i , contains:

- The set of directly referenced objects in G_i which consists of all objects in G_i referenced from a local root or from an entry in G_i 's remset;
- The set of objects in G_i reachable through a path P of references in G_i , where P begins at a reference in an object in the set of directly referenced objects and where each object referenced in P is in G_i .

In the distributed generational collector the address space is partitioned by generations that span sites. A segment is defined as a portion of a generation held on a particular site. To simplify the collector, a fixed number of generations are specified and each site holds a segment of each generation. A segment is a fixed size and represents a contiguous area of storage at a site. Each site maintains a portion of the remset for each generation. Distributed collection is concerned with the identification of garbage within a single generation across all sites of the distributed system. Promotion of an object takes place from one generation to another within a single site thereby avoiding forced migration of objects.

For each intra-site older to younger generation reference update trapped by the write barrier, a remset entry of the form,

<source object, target object>

is added to the remset of the target (younger) generation at the local site.

For each inter-site older to younger generation reference update trapped by the write barrier, a remset entry of the form,

<source generation, DA>

is added to the remset of the younger generation at the local site.

Thus, a site's remset for generation G_i may contain entries for both local and remote objects.

To allow the maintenance of correct remset entries for remote references, a site needs to know the generation of any remote object that it references. Each site maintains a DA generation lookup table that enables the site to determine the generation of each DA it holds. Any time that a DA is sent from a site S to a site T , the generation number of the referenced object is sent to T .

5.3.1 Two DTA Mappings

Since a distributed marking scheme has already been described, a modified version of this is used in the collection of a generation. Distributed collection always starts with generation G_0 and then collects as many older generations as dictated by policy. Collection starts by first pausing mutator activity on all sites. The collection of generation G_i begins with a distributed marking phase to identify all objects in G_i reachable from its remset. Following distributed marking objects are promoted.

As in the distributed mark-sweep collector, the globally stable property of the distributed state that is captured by a DTA mapping is the empty set of references

from marked objects to unmarked objects, although this time within a particular generation. More specifically we are interested only in detecting when this set is empty, as this corresponds to the completion of the distributed marking phase for a generation. The DTA mapping for the distributed mapping of a generation is therefore similar to that of the distributed mark-sweep collector.

A job corresponds to the distributed marking phase for a particular generation, called the *distributed generation marking job* (DGMJ). A DGMJ consists of two types of tasks, the first is called a Root Marking Task (RMT) and the second is called a Distributed Address Marking Task (DAMT). When the DGMJ home site starts a job, it sends a RMT to every site (including itself). Both task types trace the local graph of objects from a given start point, completely within the local segment of the generation being collected. Each object at a site has a distributed mark bit (DMB) associated with it. As an object is traced by a task its DMB is set and then it is scanned for references. Both types of task complete when they have fully traced the local graph within the segment, from their start point. An RMT at site S for generation G_i begins by removing remset entries at S for all references from G_i to all other generations. As the segment of G_i at S is traced by the initial RMT and any received DAMTs, remset entries are added for all references found thereby reconstructing the accurate remset entries for the references from G_i . Such a mechanism means that the write-barrier need only ever add remset entries, thus guaranteeing that an object x in generation G_i , referenced from an older generation G_{i+n} , is maintained when G_i is collected. The remset entry for x in G_i will only be retained if a reference to x is found in G_{i+n} during the collection of that generation.

The DA generation lookup tables held at each site constitute a distributed object to generation mapping. The distributed mapping represents distributed shared state and

when an object is promoted the shared state (the mapping) must be updated. A second DTA mapping is used to capture this state. More specifically the DTA is used to identify the point at which the distributed mapping has been brought up-to-date following object promotion. A job corresponds to the process of updating the DA generation lookup tables at each site, called the *distributed promotion job* (DPJ). For simplicity, let us assume that the *home* site for the DPJ for a particular generation is the same as the *home* site for the DGMJ for that generation.

A DPJ is started after DGMJ termination for a generation and consists of two types of task; a *promotion* task and a *generation update* task. The DPJ starts with the DPJ *home* site sending a *promotion* task to each site. The *promotion* task executes at a site S by sending a *generation update* task to each site that references a promoted object at S , and then completes. A *generation update* task sent from S to T updates the DA generation lookup table at T for the DAs of promoted objects at S , and then completes. The set of referencing sites for an object x being promoted from generation G_i to generation G_{i+1} consists of each site that sent a DAMT for the promoted object during the distributed marking phase for generation G_i .

On DPJ termination, collection of the current generation is complete. The *home* site may then begin collection of the next older generation, or restart mutator activity.

5.3.2 Club Rules for Distributed Generational Collection

The club rules for distributed generational collection provide the mechanisms to identify garbage objects across all the sites of a generation, to maintain the remsets for each generation segment and to update the distributed object to generation mapping on object promotion. The following describes the club rules for distributed generational collection. For a site $S \neq$ DMJ *home* site:

- S maintains data structures for recording TB *sent* counts. These counts are maintained as follows¹⁴. Each site S maintains a *sent* count array with an element for each site T recording $sent_S(j,T)$. When a task is sent to site T , $sent_S(j,T)$ is incremented. Note that the same data structure is used for both the DGMJ and the DPJ for a particular generation.
- S maintains the value *received* which records the number of tasks of the DGMJ received since an update was last sent. This value is incremented on receipt of an RMT and each DAMT, during marking, or a *promotion* task and each *generation update* task during promotion.
- When a DAMT is sent from S to a remote site T the *sent* count for T at S , $sent_S(j,T)$, is incremented.
- When a *generation update* task is sent from S to a remote site T the *sent* count for T at S , $sent_S(j,T)$, is incremented.
- New objects are created in the youngest generation.
- S implements a write barrier that acts on the creation of all (inter- and intra-site) older to younger generation references. When such a reference is written into an object, an entry is added to the remset for the local segment of the referenced object's generation.
- S maintains an *in-flight* table where all DAs sent in messages to remote sites are recorded. On receipt of a message from a site T containing DAs S sends an acknowledgment message back to T . The acknowledgment contains the

¹⁴ RMT and DAMT task counts may be combined here during the distributed generation marking phase, while *promotion* and *generation update* tasks counts may be combined during the promotion phase.

DAs that were sent to S from T . When S receives an acknowledgement it removes the *in-flight* table entries for each DA in the message.

- Implementation of an RMT for generation G_i at S is as follows. Any remset entries at S for references from G_i are removed. The RMT then traces the object graph from each object in G_i 's remset, marking reachable objects using their DMB. Only objects in G_i are traced. For each reference from G_i to $G_{j \neq i}$ (local or DA) found during tracing, an entry is added to G_j 's remset at S . For each DA found for a remote object in G_i (as determined by the DA generation lookup table) a DAMT is sent to the remote site. On RMT completion, a TB update is generated.
- Implementation of a DAMT for generation G_i is as follows. A DAMT message contains the DA of an object O that is remotely referenced. The local object graph is traced from O , marking reachable objects using their DMB. Only objects in G_i are traced. For each reference from G_i to $G_{j \neq i}$ (local or DA) found during tracing, an entry is added to G_j 's remset at S . For each DA found for a remote object in G_i (as determined by the DA generation lookup table) a DAMT is sent to the remote site. On completion of a DAMT a TB update is generated and sent to the home site.
- Messages containing tasks and updates are subject to ordered delivery. Tasks are executed and *update* messages processed in strict order of delivery. As a consequence, the home site H can ignore the $sent_S(j,H)$ value in an update from a site S , since the tasks to which this count relates have already completed and been balanced.
- S decides whether to promote an object O in generation G_i when it first encounters O during the collection of G_i . Each referencing site is informed of

the promotion so that its DA generation lookup table may be updated. S records each site from which a DAMT for a promoted object O is received. This defines the set of remote sites that reference O .

- On receipt of a *promotion* task for generation G_i , S sends a *generation update* task to each site that references a promoted object. A *TB update* message is then sent to the DPJ *home* site. The *TB sent* count data structures and actions used for the DPJ are identical to those used for the DMJ.
- An update is sent from the site S to the DGMJ home site when S is idle for the DGMJ. RMTs and DAMT are executed one at a time at S , therefore S is idle for the DGMJ when there are no received tasks waiting to be executed. At this point the value RC_S for the update is equal to the current *received* value at S . When the update is sent, *received* is set to zero and for each site T , $sent_S(j,T)$ is set to zero.
- An update is sent from the site S to the DPJ home site when S is idle for the DPJ. S is idle for the DPJ when the *promotion* task has been completed and there are no *generation update* tasks that have been received but not yet executed. At this point the value RC_S for the update is equal to the current *received* value at S . When the update is sent, *received* is set to zero and for each site T , $sent_S(j,T)$ is set to zero.
- To complete the garbage collection of G_i at S the local collector must move all objects that it has previously decided to promote from G_i to G_{i+1} and unmark all DMB marked objects in G_i . When an object O in G_i is promoted, remset entries for O in G_i are transferred to G_{i+1} and appropriate remset entries added and updated for any references in O . Note that these actions are completely local to S .

For the DMJ *home* site H the club rules are all of the above and:

- Before collection begins, mutator activity on all sites must be paused.
- Collection of a generation is started by sending an RMT to each site including this site.
- The home site H maintains a task count array with an element for each site T holding the value $count(j,T)$. On receipt of an *update* message from a site S , RC_S is deducted from $count(j,S)$ and for each site T , $sent_S(j,T)$ is added to $count(j,T)$. When $\forall T.count(j,T) = 0$ the *terminated* condition holds and the current job has completed. The same structure is used for both the DGMJ and the DPJ.
- On DGMJ termination detection the DPJ is started by sending a *promotion* task to each site. On DPJ termination, collection of the current generation is complete. At this point H will either restart mutator activity across all sites or start the collection of the next generation.

5.3.3 Club Rules for Homogeneous Distributed Generational Collection

The homogeneous distributed generational collector restricts the reclamation of objects at a site to the promotion phase following distributed marking. Each site implements the generic club rules as defined above and takes the following actions on receipt of the *promotion* task for generation G_i .

Any objects in G_i that are to be promoted are moved to G_{i+1} and all local references and $DA_{sym} \rightarrow CA$ address translation table entries updated accordingly. The local segment of G_i is then compacted using an implementation of the Lockwood-Morris algorithm (from [LM78]) operating only on intra-segment references. Each object on

the local site in a generation G_x ($x \neq i$) that holds a reference to an object in G_i is identified in the remset for G_i . All inter-generation (intra-site) pointers to objects in G_i and any $DA_{sym} \rightarrow CA$ or $DA \rightarrow CA$ address translation table entries (for object with DAs) are updated as the segment is compacted (and objects are moved).

The $DA_{sym} \rightarrow CA$ or $DA \rightarrow CA$ address translation tables for unmarked objects (with DAs) are removed. During the compaction of the local segment of G_i all DMB marked objects are unmarked.

5.3.4 Separating Local and Distributed Collection

To allow for independent local collection an approach is taken similar to that of the distributed mark-sweep scheme described earlier. When a reference (DA) to a local object is first exported from a site, the object is added to a *distributed root set* for that site. The distributed collector removes an object from the *distributed root set* when it determines that no other site holds a reference to the object. If, after the collection of the generation in which an object in the *distributed root set* is held, a site has received no DAMT for that object and there is no remset entry for that object then the object can be removed from the *distributed root set*.

A local collector can work over the whole space at a site using the local root and the *distributed root set* as its roots of reachability. This allows multiple local collections to execute between distributed collections.

5.3.4.1 Club Rules for Heterogeneous Distributed Generational Collection

The heterogeneous collection scheme implements the generic distributed generational club rules as described above. This section describes the actions taken by a site on receipt of the *promotion* task for generation G_i necessary for *distributed*

root set maintenance and heterogeneous local collection support. The rules for maintenance of the distributed root set are almost identical to those for distributed mark-sweep.

For all sites S :

- Each $DA_{sym} \rightarrow CA$ table entry at S has a *distributed root* flag and a *DAMT marked* flag associated with it. The *distributed root* flag is set for all newly exported DAs.
- On receipt of a DAMT for a DA the *DAMT marked* flag is set in the corresponding $DA_{sym} \rightarrow CA$ table entry at S . After setting the *DAMT marked* flag the DAMT executes as specified in the generic rules above.
- On receipt of the *promotion* task for generation G_i the $DA_{sym} \rightarrow CA$ table is scanned and the portion of the distributed root set for G_i at S is reconstructed. The *distributed root* flag is set for each entry, for an object in G_i , that has its *DAMT marked* flag set and for each entry for an object that the remset for G_i at S holds an entry. The *distributed root* flag is cleared for all other $DA_{sym} \rightarrow CA$ table entries for objects in G_i . At this point the *DAMT marked* flag is cleared for all entries for objects in G_i . Having reconstructed the portion of the *distributed root set* for G_i all objects in the local segment must be (DMB) unmarked. To avoid the necessity to sweep the whole segment the meaning of the DMB mark bit for G_i at S is flipped as before. This requires that a site maintains the DMB marked value for the local segment of each generation.

- Following the completion of a distributed collection cycle each entry in a site's $DA_{sym} \rightarrow CA$ table with its *distributed root* flag set constitutes a root of reachability for local collection.

5.3.5 Local Collection in the Heterogeneous System

A local collection mechanism for sites of the heterogeneous distributed generational system is now presented. As in the heterogeneous mark-sweep system the $DA_{sym} \rightarrow CA$ and $DA \rightarrow CA$ address tables for moved objects that have DAs are updated and *remote residents* are treated as local objects. The local collector is aware of the generations at a site and must update the local remsets when objects are moved.

5.3.5.1 A Local Semi-Space Copying Collector

The chosen local collection mechanism for the heterogeneous system is a copying collector that divides each generation segment at a site into semi-spaces. On collection, all reachable objects in each segment are copied from their current location to the free semi-space of that segment, leaving a forwarding pointer to indicate their new location.

Local collection begins by first pausing mutator activity and then copying the distinguished root object and each of the distributed root objects to the free-semi space of their respective generation segments. Each copied object is scanned for references to other objects which are in-turn copied. When copying has completed the $DA_{sym} \rightarrow CA$ and $DA \rightarrow CA$ address tables for the local site are scanned and updated. Table entries for non-copied (garbage) objects with DAs are removed at this point.

5.4 Distributed Reference Counting

A traditional reference counting garbage collector, for example [Col60], associates a reference count variable (initialised to one) with each created object. On reference copy (stack push and pointer field update) the reference count is incremented and on reference deletion (stack pop and pointer field update) the count is decremented. An object can be reclaimed as soon as its reference count reaches zero. Such a collector is incremental by its very nature since it allows for the immediate collection of garbage objects, however it is not complete. The reference counts for objects in isolated (i.e., garbage) cycles will stabilise with non-zero values.

In deriving a distributed reference counting mechanism that allows for heterogeneous local collection behaviour it is necessary to distinguish between local and remote (inter-site) references. That is, a site logically maintains two reference counts for each object. The first count is for local references and is maintained exactly as described above. The second count is a count of remote references to the object which is, by definition, distributed shared state. An object is reclaimed when both the local and remote reference counts are zero.

The remote reference count for an object is captured through a DTA mapping. Sites must be able to detect a remote reference count of zero. Therefore the DTA mapping is as follows:

- A job corresponds to a non-zero remote reference count for an object, called a distributed reference count job (DRCJ). The notation $DRCJ_x$ is used to identify the DRCJ for an object x .
- A task of $DRCJ_x$, corresponds to an inter-site reference to x .

An object's creator site is designated as the DRCJ *home* site for that object. $DRCJ_x$ is created when the first remote reference to x is exported from the home site. When $DRCJ_x$ terminates, the remote reference count for the object x is zero. The club rules are thus an implementation of the DTA for each object at each site that has exported a reference plus the actions necessary to reclaim objects.

The mapping methodology approach ultimately yields an algorithm with properties similar to *weighted reference counting* [WW87]; the difference is that the DRC collector described here distinguishes between distributed and local collection work. Distributed termination detection is applied only to the distributed collection work thus allowing for heterogeneous local collection behaviour.

5.4.1 Club Rules for Distributed Reference Counting

The DTA mapping for reference counting provides a collector framework that uses a TB implementation to identify objects with a zero remote reference count. The following assertions are made:

- A site S sends a TB update for $DRCJ_x$, when S is *idle* for $DRCJ_x$.
- The method by which the idle state is detected is specific to a particular collector. For the description of the generic club rules it is sufficient to assume that a site can detect idle jobs.

The following describes the club rules that are generic to both the homogeneous and heterogeneous distributed reference counting collectors. At a site S that is not the *home* site for $DRCJ_x$:

- S maintains data structures for recording TB *receive* counts as follows: the value $received_S(DRCJ_x)$ records the number of tasks of $DRCJ_x$, received at S .

- S maintains the value $sent_S(DRCJ_x, T)$ which records the number of task of j sent from S to T . When a task is sent to site T , $sent_S(DRCJ_x, T)$ is incremented.
- When the site S detects that $DRCJ_x$ is idle, a TB update is sent to the home site of $DRCJ_x$. On sending an update for $DRCJ_x$, $sent_S(DRCJ_x, T)$ is set to zero for all sites T and $received_S(DRCJ_x)$ is set to zero.

For the *home* site H of $DRCJ_x$ the club rules are all of the above and:

- The home site H maintains a task count array for the job $DRCJ_x$. The count array has an element for each site T holding the value $count_H(DRCJ_x, T)$. On receipt of an *update* message from a site S , RC_S is deducted from $count_H(DRCJ_x, S)$ and for each site T , $sent_S(DRCJ_x, T)$ is added to $count_H(DRCJ_x, T)$. When $\forall T. count_H(DRCJ_x, T) = 0$ then the termination condition is satisfied.

5.4.2 Club Rules for a Homogeneous Reference Counting Collector

In the homogeneous scenario, two separate reference counts are maintained for local objects, one for local references and the other (maintained by the TB implementation) for references from remote sites. An object x must have a zero local reference count *and* a zero remote reference count before it is collected.

A site detects the idle state for each DRCJ, using a local task counting mechanism. That is, a site S records the number of tasks it holds for each DRCJ. When the count is zero, for a $DRCJ_x$, S holds no tasks of $DRCJ_x$ and an update message is sent for $DRCJ_x$. In other words, whenever a site S creates or deletes an inter-site reference to an object (which can be detected through the use of a write barrier for instance), it modifies the equivalent of a local reference count. When this count is zero, S holds

no references to the object. Note that this is not the only means by which a site can detect the idle state for a DRCJ but the decision has been taken to describe this method since it most closely resembles a traditional reference counting scheme.

The homogeneous distributed reference counting collector implements the generic club rules as described above, and the following additional rules.

At a site S that is not the home site of $DRCJ_x$:

- S maintains a *local task count* value for $DRCJ_x$ written $LTC(DRCJ_x)$. When a new task of $DRCJ_x$ is created locally, $LTC(DRCJ_x)$ is incremented. When a task of $DRCJ_x$ is deleted (overwritten), $LTC(DRCJ_x)$ is decremented.
- If $LTC(DRCJ_x)=0$ then $DRCJ_x$ is idle at S .

At the home site of $DRCJ_x$:

- H maintains a *remote reference count* value for the object x , written $RRC(x)$. On creation of $DRCJ_x$, $RRC(x)$ is initialised to one.
- A site maintains a *local reference count* value for each local object x , written $LRC(x)$.
- When a local reference to x is created, $LRC(x)$ is incremented, and when a local reference to x is deleted $LRC(x)$ is decremented. If $LRC(x)$ becomes equal to zero, x is reclaimed at this point if and only if $RRC(x)$ is also zero.
- On termination of $DRCJ_x$, $RRC(x)$ is set to zero. The object x is reclaimed at this point if and only if $LRC(x)$ also equals zero.
- When an object x is reclaimed the local site carries out the appropriate actions for the deletion of each reference in x .

The homogenous collector is not complete since local and inter-site cycles of garbage are not reclaimed.

5.4.3 Separating Local and Distributed Collection

Since the distributed reference counting collection is only concerned with the counts of remote references, the separation of local and distributed collection is almost trivial. The only requirement of the local collector is to identify idle jobs.

To enable safe independent local collection, a similar approach is adopted to that of the distributed mark-sweep scheme described earlier. When a reference (DA) to a local object is first exported from a site, the object is added to a *distributed root set* for that site. Here the distributed reference counting collector will remove an object from the *distributed root set* when it determines that no other site holds a reference to the object, i.e., on termination of the DRCJ associated with that object. As before, local collection can proceed at any time based on reachability from the *local root set* (which contains the distinguished local root and the *distributed root set*).

5.4.4 Club Rules for a Heterogeneous Reference Counting Collector

Two local collection mechanisms for sites in the heterogeneous distributed reference counting system are now described. As in the distributed mark-sweep collector, the club rules have a minimum impact on the behaviour of the local collectors. The local collectors are charged with updating a local site's address translation table entries if objects are moved and with removing entries for reclaimed objects that have DAs. The local collectors treat *remote resident* objects as local objects thus ensuring that there is no interference between local collection and the object duplication policy. These are identified by having entries in the DA \rightarrow CA table.

The heterogeneous collection scheme implements the generic distributed reference counting club rules as described above and the following rules:

- Each $DA_{sym} \rightarrow CA$ table entry has an associated *distributed root* flag (as in the distributed mark sweep collector), or DRF. For an object x , the flag is set to identify that the object is currently in the *distributed root set*, on export of the first remote reference to x .
- On termination of $DRCJ_x$ the DRF flag is cleared in the $DA_{sym} \rightarrow CA$ table entry for x .

5.4.5 A Local Mark Sweep Collector

As a first example of independent local collection a stop-the-world, mark-compact local collector is described. Local collection proceeds as follows. Mutator activity at the local site is first stopped and then the object graph is traced from the local root set. Each object has an associated local mark bit (LMB). The LMB is set during the marking phase for each object that is traced.

The TB data structure that records the sent and received counts for each DRCJ is extended to also include a LMB. During marking, the LMB for a DRCJ is set on discovery of a remote reference to the object corresponding to that job. This can be thought of as marking each DRCJ for which the site currently holds a task.

When marking is complete the heap is scanned and compacted, clearing LMBs and updating local references and address translation table entries (in both the $DA_{sym} \rightarrow CA$ and $DA \rightarrow CA$ tables). The TB data structures are then scanned. Any unmarked DRCJ is idle at this site.

5.4.6 A Local Semi-Space Copying Collector

As a second example of independent local collection a semi-space collector is illustrated. To determine idle DRCJs, the local collector, at the end of a copy phase, records the set of remote references discovered during the copy phase. Before

mutator activity is restarted, this set is compared with the set from the previous copy phase. Any remote reference missing from the latest set corresponds to a DRCJ that is idle at this site.

5.4.7 Discussion

The DRC collector is not complete although the property of completeness is of primary importance in maintaining the automatic memory management abstraction. However the DRC collector is described here since, of the three heterogeneous schemes, the DRC collector best demonstrates the principle of independent local collection. The club rules for this scheme place the least restrictions on the behaviour of a site, requiring only that send and receive counts are maintained as tasks (references) are sent between sites and that update messages are sent at the appropriate times.

The club rules define a contract between the sites of the distributed system and the distributed garbage collector. The contract that exists between a site of the distributed system and the heterogeneous DRC collector is however somewhat different to that which exists for the first five collectors. In both DRC collectors idleness detection is achieved through a set of actions corresponding to local events. In the homogeneous scheme these actions are defined by the club rules; a site maintains a task count for each job and when the count reaches zero the job is idle and an update is sent. However in the heterogeneous DRC collector idleness detection is achieved purely by the actions of the local collectors. In this way DRC collection is driven by the local garbage collectors at each site, although no club rule has been defined to explicitly determine at what intervals local GC is carried out (if it happens at all). However, this problem is not as bad as it first appears. All that is required of a site is that it eventually carries out a local garbage collection.

5.5 Summary

The collectors described here are effectively proofs of concept for the idea that the mapping methodology and the definition of the club rules can yield collectors that allow for independent (heterogeneous) local collection behaviour within a single derived distributed collection scheme. Heterogeneous local behaviour does not necessarily mean different styles of local collector, as demonstrated above, but in general allows for sites of the distributed system to adopt policies that are locally beneficial.

The stop-the-world mechanism used in the collection of a distributed generation is not well suited to a scalable distributed system. The distributed generational collection scheme as presented here represents more of a proof of concept of the DTA to GC mapping derivation technique than a suitable distributed collector. An attempt has been made however to keep the description as close as possible to that of [LH83].

Clearly a distributed generational collector where mutator activity can be interleaved with the collection of a generation is preferable to the approach taken here. The development of such a scheme is seen as further work. The problem of reset maintenance in the face of interleaved generation collection and mutator activity is considered similar to that faced by an implementation of a DMOS collector [HMM+97] in maintaining car and train resets. This is discussed in detail in the following three chapters.

The use of synchronous communication for the transmission of inter-site references (DAs) also reduces the scalability of the collector. Addressing this problem is also seen as further work.

Another area of future work lies in applying the methodology to derive club rules operating over existing local collectors. That is, tailoring the club rules to provide a DGC that allows for independent operation for a pre-defined local collector, as opposed to the work described here where the local collectors are defined to operate through an interface provided by the DGC.

The specific contribution of this author to the work described in this chapter is as follows:

- Development of the three GC to DTA mappings.
- Definition of the six sets of club rules.
- Concrete implementations of the heterogeneous distributed mark-sweep and distributed reference counting collectors, both with local mark-compact collectors.

6 Developing a DTA Mapping for DMOS

This chapter presents a garbage collection to DTA mapping that forms the basis for the derivation of a new implementation of the DMOS distributed garbage collector.

The DMOS collector is discussed in Chapter 2 but it is helpful to reiterate the key features of the algorithm at this point. DMOS demonstrates a combination of desirable properties for a distributed collector. Specifically, DMOS is:

1. **Safe:** it does not collect live (reachable) objects.
2. **Complete:** it reclaims all garbage, including cyclic garbage that spans sites, within a finite number of invocations.
3. **Non-disruptive:** it bounds the amount of collection work, thereby bounding the time and space requirements, for each invocation.
4. **Incremental:** it reclaims space incrementally without global knowledge of reachability.
5. **Local:** it initiates local collections at each site independently of other sites.
6. **Independent:** it is independent of the specific local collection algorithm employed at each site, though it imposes some requirements on the local collectors.
7. **Decentralised:** it uses no algorithms that rely on a single central site for processing or global synchronisation.
8. **Asynchronous:** it communicates via asynchronous messages, and the collector at a site need only synchronise with a site in one particular case; application computation never need wait for such synchronisation.

The DMOS collector therefore has the prerequisites for scalability which are incrementality, locality, decentralisation and asynchrony. DMOS is arguably unique in its combination of all of these attractive properties, however no satisfactory implementation of the collector has yet been published. The aim of this chapter is to employ the mapping methodology to derive a new implementation of DMOS.

The MOS [HM92], PMOS [MMH96] and DMOS [HMM+97] algorithms can be viewed as specific derivations of the Train algorithm that reflect their target environments, namely main-memory, persistence and distributed address spaces. However it is useful, in describing the application of the mapping methodology for deriving an implementation of DMOS, to define a generic form of the Train algorithm, i.e. one that is not targeted at a specific architecture. This is called the UMOS (Unordered MOS) collection algorithm.

In deriving a new implementation of the DMOS collector the approach taken is to first define a distribution of the UMOS collector. The distribution of UMOS is designed such that the shared state that is distributed has some globally stable properties that may be captured by a DTA mapping.

The derivation begins with a definition of the UMOS algorithm that includes safety and completeness arguments. A process of stepwise refinement is then used to define the distributed garbage collector actions for each site. Chapter 7 uses this refinement to describe a new implementation of DMOS using the Task Balancing DTA for the DPBASE system.

6.1 The UMOS Collection Algorithm

The UMOS algorithm is described using the metaphor, due to MOS, of *trains* made up of *cars*. Cars partition the set of all object into disjoint subsets. Thus at any given

time, each object is associated with exactly one car. It is a matter of policy as to how many cars there are in a train, the size of cars, and when new cars are created.

UMOS reclaims space using two interacting collection mechanisms.

The first mechanism collects individual cars. An object is associated with one car at any given time but the object can be (logically) moved to a different car, thus changing the car with which the object is associated. This is known as object re-association. At least one car is collected on each invocation of the collector, by re-associating¹⁵ its potentially reachable objects with other cars in accordance with a set of re-association rules. Any car may be a candidate for collection on any given invocation¹⁶. Once all the potentially reachable objects have been re-associated, the remaining objects within the car are unreachable and the car can be reclaimed immediately.

Cars are grouped into trains to collect cyclic garbage that spans more than one car. Trains can contain an unbounded number of cars thereby providing a collection mechanism for garbage cycles of arbitrary size. It is again a matter of policy as to when new trains are created and how many trains there are, but there must always be at least two. The second collection mechanism reclaims an entire train when all its objects become unreachable from outside the train.

¹⁵ Re-association can be achieved by copying or by address mapping. The mechanism is made non-disruptive by bounding the size of each car in the first case and by bounding the cost of address translation in the second.

¹⁶ This defines the unordered nature of the collector.

Trains are significant in UMOS since they allow objects to be grouped according to their reachability from other objects, and ultimately, using the re-association rules, for garbage to be separated from live data.

The re-association rules make use of an ordering on trains, which can be imagined as being based on the logical time at which the trains are created [Lam78]. Trains are referred to as being younger or older than other trains using the logical ordering. In the UMOS collector any car of any train may be selected for collection but every car is eventually collected. The UMOS algorithm does not define the mechanism by which cars are selected for collection but instead assumes that some policy (external to the UMOS algorithm) is implemented to ensure that each car is eventually the target for collection.

The re-association rules are defined as:

- An object directly reachable from the mutator is re-associated to a car of any *younger* train (possibly creating a new train).
- An object reachable from one or more *younger* trains can be re-associated to a car of (any one of) those trains.
- An object reachable only from another car of the *current* train, or from one or more *older* trains, should be re-associated to some other car (possibly a new one) of the *current* train.

The effect of these rules is to re-associate objects from older to younger trains, but only if they are reachable from roots or from those younger trains. A dead object can re-associate to the youngest train from which it is referenced (from and via other dead objects), but no further. Eventually garbage that spans multiple trains will collapse into a single train.

A data structure is maintained to identify the potentially reachable objects in a car to facilitate re-association. In terms of traditional partitioned garbage collection schemes this data structure is known as the remembered set, or remset, for a car. The data structure records for each reference into a car, a reference to the object x , a reference to the object y that holds the reference to x and the train identifier for the train that holds y . This is illustrated in Figure 6.1 below for an object x in a car $C1$ which is referenced by an object y in a car $C2$.

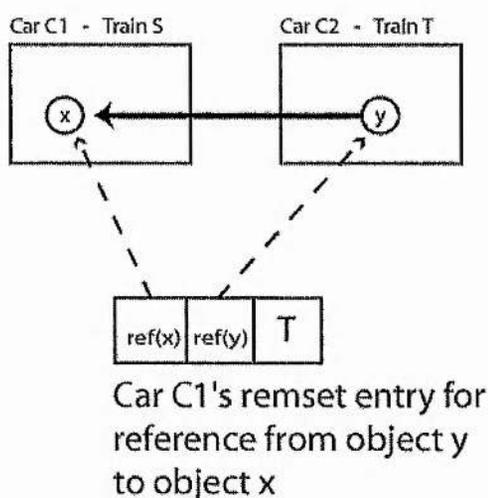


Figure 6.1 - An Example Remset Entry

The remset serves two purposes; to identify the destination train for objects during re-association (for instance in Figure 6.1 train T is identified for the re-association of object x) and to identify the reference fields that must be updated when the object is moved from its current car to its new car (for instance in Figure 6.1 the reference to object x in object y must be updated if x is moved on re-association).

Figure 6.2, Figure 6.3, Figure 6.4 and Figure 6.5 below show the effect of re-association on a garbage cycle that is initially held in three trains T_0 , T_1 and T_2 where T_0 is the youngest train. The example shows how the cycle is collapsed into a single train with the collection of three cars.

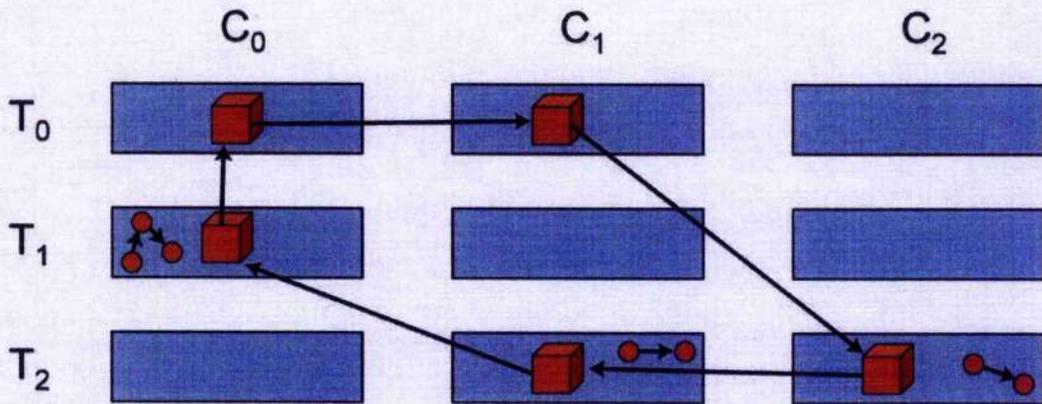


Figure 6.2 - A Garbage Cycle held in Three Trains

Initially T_2C_2 is selected for collection. One of its objects is referenced from train T_0 and therefore the object is re-associated to a car of this train (in this case the object is re-associated to car C_1 of train T_0). This yields the graph shown in Figure 6.3.

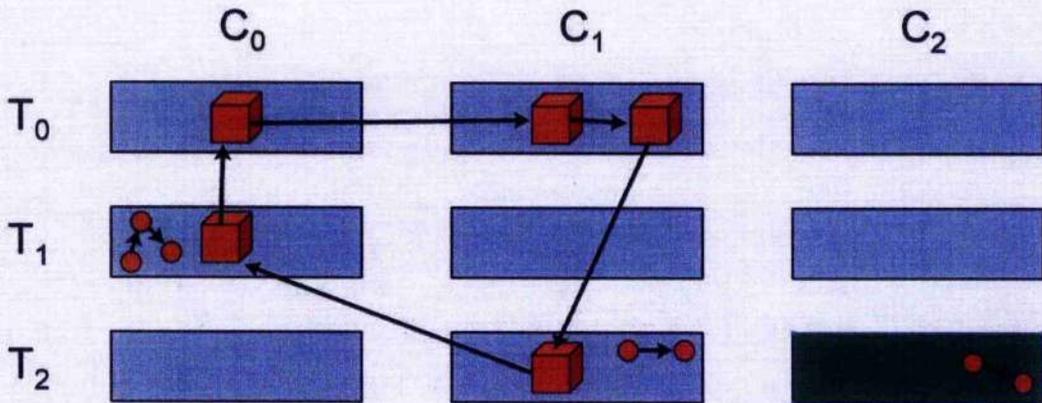


Figure 6.3 - Collection of Car C_2 in Train T_2

Car T_2C_1 is then selected for collection. The object held in this car is referenced from T_0 and in this case it is re-associated into car T_0C_2 (Figure 6.4).

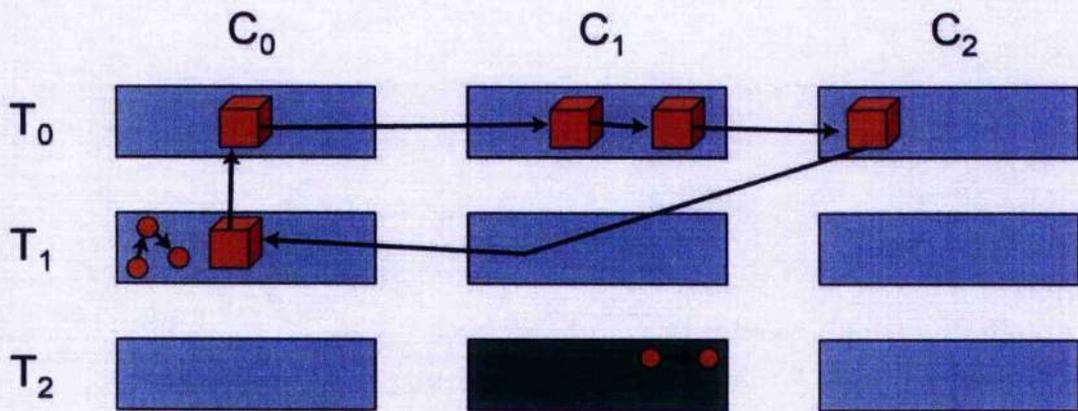


Figure 6.4 - Collection of Car C_1 of Train T_2

Finally car T_1C_0 is selected for collection, whereupon its reachable object is re-associated to T_0C_2 thus collapsing the garbage cycle into a single train (Figure 6.5).

Train T_0 is now isolated and may be reclaimed.

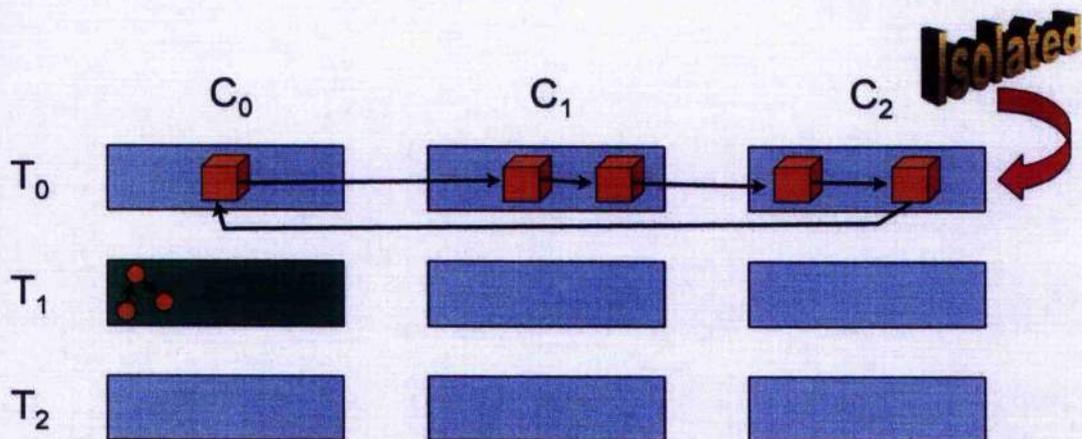


Figure 6.5 - Collection of Car C_0 of train T_1

This example serves to demonstrate the two interacting collection mechanisms whereby the car collection mechanism has collapsed the garbage cycle into a single train and now the train reclamation mechanism can reclaim the unreachable train.

A train is garbage when no further re-association of objects out of the train is possible. That is, when there are no pointers into the train. These trains are known as isolated trains and if isolation is detected a train can be reclaimed. While a particular

sequence of car collection is demonstrated above, any sequence will result in the objects collapsing into train T_0 as long as each car is eventually collected.

The re-association rules as stated are not enough to guarantee the eventual isolation of a train. Figure 6.6 below demonstrates a situation where a particular (unfortunate) interleaving of mutator and car collection activity can lead to a train becoming permanently non-isolated. In Figure 6.6, object y in car b references object x in car a and in turn is referenced by the mutator and object x in car a . On collection of car a , object x is re-associated to a new car c . Now that the collector has finished its current invocation the mutator may then create a reference to x and delete its reference to y leaving the train in the same state as it was before the re-association of car a . This is known as the *zero-progress problem* (first identified in [GS93]) and can be prevented by ensuring that any object known to have been referenced by the mutator, while in a particular car, is re-associated to a younger train, even if the mutator reference no longer exists when the object's car is collected.

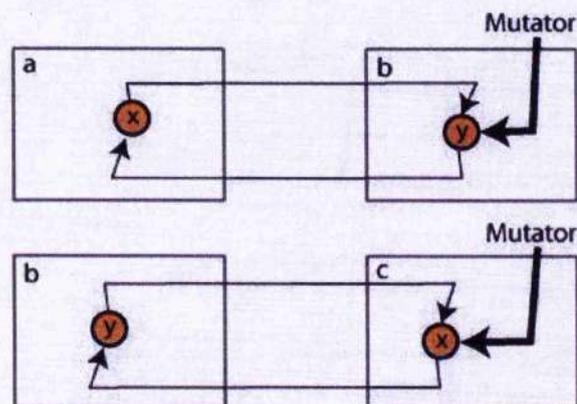


Figure 6.6 - The Zero-Progress Problem

The allocation of new objects can also prevent a train from ever becoming garbage.

To avoid this a rule is introduced that prevents the mutator from allocating new

objects in the oldest train. This ensures that the oldest train will eventually become isolated.

6.1.1 UMOS Safety and Completeness

To show the safety of the UMOS collector it is necessary to show that no live object is ever reclaimed. The set of live objects contains any object that is in the transitive closure computed from the collector's roots of reachability. Therefore if an object x in car C is live, it is reachable through a path of references from one of the roots and as such will be referenced from an entry in C 's remset. Since all objects referenced by the remset are re-associated on car collection, and therefore not collected, no live objects are discarded on collection of a car. The same argument holds for train collection. If a train T contains a live object, the remset for that object's car will, by definition, contain an entry for a reference from outside T . So while a train contains a live object it will not be reclaimed.

To show that UMOS is complete it is necessary to show that every garbage object is eventually collected. Any non-cyclic data structure will be reclaimed through car collection alone since any object that it is unreachable from outside its car will be reclaimed when the car is collected¹⁷. Therefore the completeness argument need only concern the reclamation of cyclic garbage structures that span multiple cars. The re-association rules and the immutable nature of garbage dictate that a garbage object will be re-associated as far as its youngest referent train and no further. This means that a garbage cycle that is completely contained within a set of trains will

¹⁷ The same argument applies to cyclic garbage structures that are completely contained within a single car.

collapse, after some finite number of car collections, into the youngest train in the set.

The completeness of UMOS can now be shown by proving that every train will eventually become isolated and thus reclaimed. This can be proved as follows:

- The oldest train is guaranteed to become isolated since the mutator is not allowed to allocate into it, no object can be re-associated into it and the re-association rules will eventually re-associate any reachable object in it to a younger train.
- Every train will eventually become the oldest train.

The safety and completeness arguments for the UMOS collector are similar to those presented by Grarup and Seligmann in [GS93] for the ordered version of the Train algorithm in the MOS collector.

6.1.2 Concurrency Issues in UMOS

Before looking at the issues concerning the distribution of UMOS it is instructive to examine how the algorithm may be made concurrent. There are two issues that arise.

- The first concerns the atomic updating of the data structures used in maintaining the car and train information. For the moment this will largely be ignored since locking of local data structures is not a problem and in the distributed system only one collection at a time is considered at each site.

- A more subtle problem in UMOS is the realisation that train isolation is not stable. In the non-concurrent version if isolation is detected the train can be reclaimed immediately. Furthermore if the train is not reclaimed immediately then isolation has to be recalculated. The lack of stability can be caused by either the mutator allocating a new object in an isolated train or the collector re-associating an object into an isolated train. This last condition is known as the *unwanted relative* problem and is demonstrated in Figure 6.7 below.

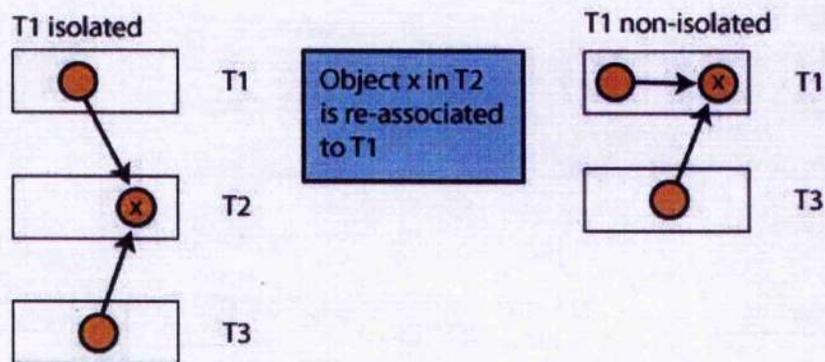


Figure 6.7 - The Unwanted Relative Problem

In Figure 6.7 the object x in train $T2$ is promoted to train $T1$, which in turn causes $T1$ to become non-isolated due to the reference to x from $T3$.

In concurrent UMOS the unwanted relative becomes a serious problem in that one thread may detect isolation while another is busy promoting into the same isolated train. Discussion of a solution to this is delayed until the description of DMOS but at this point it is necessary to emphasise that this is a consequence of concurrency and not of distribution per se.

6.2 Mapping UMOS to Distributed Termination: The DMOS Algorithm

Here the DT mapping methodology from Chapter 1 is applied to the centralised UMOS garbage collection scheme to derive the DMOS algorithm. The steps are,

- Demonstrate safety and completeness for UMOS (Section 6.1.1).
- Make UMOS concurrent (Section 6.1.2).
- Define the distribution of UMOS to yield DMOS and then examine the consequences of distribution on the car and train collection mechanisms.
- Select a particular DTA (deferred until Chapter 7).
- Determine what constitutes DT jobs and tasks in DMOS together with the events that constitute the site actions on jobs and tasks while ensuring that tasks cannot be created spontaneously (Section 6.2.2.1 and Section 6.2.3.2).
- Complete the mapping by showing that termination of a job corresponds to detecting that a set of objects is unreachable and can be reclaimed.

Since the steps in the first two bullets have already been addressed the first issue is to address the distribution of UMOS.

6.2.1 Distributing UMOS

The distribution of UMOS, that is the DMOS distributed garbage collector, is defined as follows. Each *car* resides on a single site, but *trains* may be distributed, i.e., a train may have cars at multiple sites. The idea is that car collection will be local to a site, whereas train collection may be distributed. DMOS proceeds concurrently and asynchronously across sites by ensuring that any global information that may not be entirely up-to-date is at least safe. While collection

proceeds concurrently and asynchronously across sites, only one collection is considered at a time within a single site.

6.2.1.1 Consequences of Distribution on Collection

Recall that the mutator may not allocate into the oldest train and the re-association rules ensure that the collector may not re-associate into the oldest train. Thus the unwanted relative does not apply to the oldest train. Since every train will eventually become the oldest train its collection is assured, if delayed. The consequence of distribution would appear to be that each site must know which train is the oldest in the system. However, it is enough to prevent a site from allocating into the oldest train that it does know about to ensure that no site allocates in the globally oldest train. This ensures that the oldest train in the system eventually becomes isolated.

The above description allows the collection of the oldest train only if the oldest train can be identified. Alternatively any train that becomes isolated could be collected if a globally stable state that corresponds to train isolation can be found. The solution to this is to make train isolation itself a globally stable state. To do this it is necessary to ensure that the mutator does not allocate into, and that the collector does not re-associate into, an isolated train as described in Section 6.1.2 above.

In following the derivation methodology, globally stable properties of the distributed shared state are identified and these properties are captured using a DTA mapping. Clearly the whole of the distributed object graph can be considered as shared state but due to asynchrony it is likely that the calculation of consistent views is liable to be intractable. The trick is to identify subsets of this shared state that ensure consistency of the whole in a tractable manner. More specifically each site maintains a view of the shared state and the DTA is used to construct, or identify, consistent

cuts across the global state by identifying globally stable states. If a site knows that it has a globally consistent view of the shared state then that site can make safe decisions (over which objects are garbage for instance). In the DMOS case the identification of the shared state subsets is determined by the nature of the collector itself.

In the following sections the mapping is described in detail. Here the intuition for the structure of the DMOS collector is given by presenting an overview of the shared state in the distributed DMOS collector and the stable properties that are captured by the DTAs. From the definition of the distribution of DMOS, there are two types of shared state.

A train represents distributed shared state since its cars can be held on multiple sites. The collector reclaims a train when there no longer exists any reference into the train. The consistent subset is identified as the set of inter-train references. There are two parts to this consistent subset: a count of the number of references into a train; and a distributed mapping called the object-to-train map that allows a site to identify the train holding any object which that site references.

- The first part of the consistent subset is the number of references into a train. This may be calculated by tracking all reference manipulations and using a DTA to determine when there are none into a particular train.
- The second part of the consistent subset is to maintain a distributed mapping from objects to trains, called the object-to-train map. This map ensures the integrity of the information required for tracking all reference manipulations. The object-to-train map only changes when an object is re-associated to a younger train, which is known as object promotion. For example, when the re-association rules cause an object to be 'moved' from T_n to T_{n+i} , the map

becomes inconsistent since some sites may believe the object to be in T_n while others believe it to be in T_{n+i} . In this case, a second DTA is used to determine when the map is consistent. More specifically, the DTA is used to determine when the distributed mapping for a particular object has returned to a consistent state following the promotion of that object.

6.2.1.2 Train Numbering

The mechanism of logical train ordering is extended to the distributed context to impose a global ordering on train ages. This is achieved by identifying each train with a pair $n:A$, where the positive integer n indicates the logical *birth date*¹⁸ of the train (i.e., higher numbers are younger), and A is the site that created the train (this is termed the train's *home site*). The number n is unique within the home site, thus $n:A$ is unique within the whole system. It is assumed that sites are also ordered (e.g., by some kind of site numbers), and $n:A < m:B$ iff $n < m$ or $(n = m \text{ and } A < B)$, i.e., lexicographic ordering. The home site H of train $n:A$ is responsible for creating, managing, and cleaning up the train.

Although site H created train $n:A$, any number of sites may contain cars of $n:A$. For clarity in the rest of the thesis a single train number is assumed, knowing that it represents the two part train identifier and logical ordering described here. Therefore, for two trains S and T the inequality $S > T$ shall indicate that the train S is younger than the train T .

¹⁸ The *birth date* need not indicate a date or time but is used only to indicate relative ages of trains.

6.2.2 Train Collection

6.2.2.1 A DTA Mapping for Train Collection

Initially it appears that isolated train detection is not a problem that can be addressed with a mapping to distributed termination detection since the state we are trying to identify is not globally stable. However the subtlety of the proposed solution lies in the mapping that is used, whereby a train T is a job (called an *isolated train job*), written $\text{isolatedTrain}(T)$, and a reference into T is a task of that job. The manipulation of tasks and jobs is governed by the rules of the distributed termination model and thus due to the mapping, neither the mutator nor the garbage collector may place an object into a train unless there already exists a pointer into the train somewhere in the distributed system. That is, a train cannot change from the isolated state to the non-isolated state; train isolation is therefore a globally stable state. However it must be shown that the restriction placed on object allocation and re-association does not affect progress of the distributed collector or compromise its completeness.

Since car collection is a purely local activity the restriction on allocation and re-association is restated as follows. A site cannot place an object in a train T (through allocation or re-association) unless it already holds a task of the isolated train job for T ¹⁹. A *home* site for a train T is defined as being responsible for detecting termination of the isolated train job for T . This is the site that created T . A site that wishes to allocate or re-associate into a train T , that it does not already hold a task of,

¹⁹ That is, there is no spontaneous creation of tasks.

can request a task from the home site of T^{20} . However, such tasks must eventually complete to guarantee progress in the collector. If the only task of a train T held by a site A is a task that was sent from the *home* site of T to A on appeal, then this task must eventually complete; thus allowing T to become isolated. Since these tasks are not mapped to references between objects an additional mechanism (described in the DMOS implementation in Chapter 7) is required to ensure that they eventually complete.

Where the train is not isolated it is always safe for the home site to issue such a task since it is the home site's view of the train that will determine termination. The other possibility is that the train is already isolated and in this case the home site must indicate this to the requesting site. On receiving such an indication the requesting site updates its local view of the references to the object to remove any indication of a reference from the train T . The local site's view is simply out of date and if the object is reachable from a younger train the site will eventually learn of such a reference. In the mean time the object may be safely re-associated to a car of its current train. Progress is guaranteed since (as is shown later) the train reclamation mechanism is complete and is independent from the mechanisms for identifying the set of references into a car.

When the mutator and garbage collector manipulate object references they generate reference events which require appropriate isolated train task actions. While reference events are due to the manipulation of pointers to objects the corresponding

²⁰ This should only be done as a last resort. For instance, if a site knows of more than one train that references an object, and the site holds a task of some of those trains, then the object should be re-associated to one of the trains of which the site already holds a task.

isolated train task actions are for trains holding the referenced objects. In order for a site to know which isolated train job the task actions are associated with, that site must know the train number for all of the objects that it references. This is achieved through the object-to-train map. Two ways are suggested in which this can be implemented: either an object's train number is encoded in the reference to the object or each site maintains a local table that maps objects to trains. Note that here we are concerned only with remote objects since it is assumed that a site holding an object can tell which train that object is in.

Table 6.1 below describes the reference events that can occur at a site *A*. Note that the notation $\langle x, T \rangle$ is used to identify an object *x* that is held in train *T*. These are the only events (due to the mutator) at a site that are of significance to the DTA mapping.

| Description | Reference Event |
|---|---|
| A creates a new object $\langle x, T \rangle$ | Create $\langle x, T \rangle$ |
| A creates a new (copy of a) reference to $\langle x, T \rangle$ | Copy ref($\langle x, T \rangle$) |
| A deletes a reference to $\langle x, T \rangle$ | Delete ref($\langle x, T \rangle$) |
| A sends a reference to $\langle x, T \rangle$ to site <i>B</i> | Send ref($\langle x, T \rangle$), $\rightarrow B$ |
| A receives a reference to $\langle x, T \rangle$ | Receive ref($\langle x, T \rangle$) |

Table 6.1 - Reference Events

In Section 6.3 the site actions, on train isolation tasks that correspond to these reference events, are described.

6.2.3 Car Collection

Collector progress is achieved through the collection of individual cars which involves the re-association of each potentially reachable object to a car of a train that holds a reference to the object. For this, a car needs to know what points to the objects it holds; this is recorded in a remset for the car and in the centralised UMOS scheme the remsets are complete. The remset entries for a car C in the centralised UMOS collector identify the trains that hold references to objects in C (used during object re-association) and identify the objects that hold these references (so that references may be updated if objects are moved during re-association).

In the distributed context, a site holding a car C can only ever construct a local view of the remset for C which can be, due to asynchrony, out-of-date. Referent train information is required to allow object re-association while information relating to individual references to an object is required to allow object references to be updated if objects are moved. It is useful to distinguish between the two types of information encoded in a car's remset since this information can be managed independently in the distributed system:

- To maintain the remset as a root set for car collection only the referent train information is required since a single remset entry for an object x , identifying any train as holding a reference to x , is sufficient to ensure that x is re-associated and not reclaimed during car collection. This information can be out-of-date as long as it is eventually sufficiently accurate to collapse a garbage cycle into a single train.
- Information that allows the identification of the references that must be updated following object re-association can be maintained separately. The work that must be done to update references is dependent on the addressing

mechanism both within a single site and between sites. Re-associating an object does not necessarily require that references are updated, although clearly the object-to-train map must be updated if an object is re-associated to a younger train. With this in mind the mechanism described here is neutral towards addressing mechanisms. Instead of assuming any particular addressing mechanism, an abstraction based on the substitution of one object for another, following object re-association, is used in developing a DTA mapping for car collection.

In summary, to identify the potentially live objects in a car it is sufficient to construct a conservative approximation to the remset for the car that carries referent train information. The rules for a suitable remset for car collection are defined in Section 6.2.3.3 below. At this stage all that it is necessary to say is that the remset acts as the root set for car collection and only objects reachable from the remset are re-associated on car collection.

The re-association of an object x can be considered as moving x from its current car C to a different car C' . If an object's car (and or train) is encoded in the reference to that object then on re-association each reference to the object must be updated with the object's new car (and or train). The process of updating the references to x is the logical equivalent of substituting a reference to a new object x' in car C' for each reference to object x in car C . This is known as *object substitution*.

6.2.3.1 Object Substitution

The subtlety of car collection is that during re-association objects may change trains. This is called *object promotion* and it requires that references are updated (where the object reference encodes train number) or that the distributed object to train map is

updated with a new mapping for the promoted object. Effectively object x in train T written $\langle x, T \rangle$ is substituted with object $\langle x', T' \rangle$ where $T \neq T'$. During the course of the substitution, each isolated train task of T due to a reference to $\langle x, T \rangle$ will be replaced with a task of T' and completed. Note that in a system where an object's car is encoded in its reference then re-association, where there is no promotion, results in a substitution where $T = T'$.

Since substitution is not instantaneous the re-association of $\langle x, T \rangle$ causes a state where sites hold references to both x and x' . The site holding $\langle x, T \rangle$ is required to maintain meta-data relating to the re-association of $\langle x, T \rangle$ so that, for instance, operations on $\langle x, T \rangle$ can be redirected to $\langle x', T' \rangle$. To allow this meta-data to be safely discarded, substitution in DMOS requires a mechanism that can detect when the substitution of $\langle x', T' \rangle$ for $\langle x, T \rangle$ has completed. The solution is based on the assertion that substitution of $\langle x', T' \rangle$ for $\langle x, T \rangle$ has completed when there no longer exists any reference to $\langle x, T \rangle$. On completion of the substitution, $\langle x, T \rangle$ is in the *isolated object* state which is globally stable and as such can be detected through a DT mapping. In this second mapping, an object in a train is a job (called an *isolated object job*), written $\text{isolatedObject}(\langle x, T \rangle)$, and a reference to the object is a task of that job.

In effect isolated object detection is achieved by tracking every reference to an object throughout the object's lifetime²¹. The reference events in Table 6.1 and the object substitution events in Table 6.2 cause site actions on isolated object jobs and tasks, in addition to the isolated train jobs and tasks. Objects can become isolated not only through object substitution but also through mutator activity alone. A

²¹ Recall that references are already being tracked to detect isolated trains.

consequence of tracking references to objects is that the DTA will additionally detect these isolated objects.

Substitution of $\langle x'', T'' \rangle$ for $\langle x', T' \rangle$ may begin before the substitution for $\langle x, T \rangle$ has completed. This does not represent a special case of the algorithm since each substitution effectively operates in isolation from other substitutions, although the substitution for $\langle x'', T'' \rangle$ cannot complete before the substitution for $\langle x', T' \rangle$.

6.2.3.2 *The Substitution Protocol*

The *home* site for an object $\langle x, T \rangle$ is defined to be the site where $\langle x, T \rangle$ is currently resident and is being promoted. This is initially the creator site of $\langle x, T \rangle$. The *home* site is responsible for initiating the substitution process and for determining when it has completed. Note that the term, “*home* site” has now been defined for trains and for objects in trains.

The substitution protocol represents a modified version of the migration protocol from [HMM+98] and is as follows.

- The *home* site sends a *substitution* message, $\text{SUBSTITUTE}(\langle x, T \rangle \rightarrow \langle x', T' \rangle)$, to each site (including itself), that it knows holds a reference to $\langle x, T \rangle$, informing it of the promotion to T' . The implementation in Chapter 7 explains how the set of sites that hold a reference to $\langle x, T \rangle$ is calculated.
- Each site maintains a *substitution table* which records the objects that it is in the process of substituting. Entries in the substitution table are of the form $\langle x, T \rangle \rightarrow \langle x', T' \rangle$. The *home* site adds a table entry for $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ to its local substitution table when the substitution of $\langle x', T' \rangle$ for $\langle x, T \rangle$ begins. A

site that is not the *home* site adds an entry to its local substitution table on receipt of the *substitution* message for $\langle x, T \rangle \rightarrow \langle x', T' \rangle$.

- On receipt of the substitution message for the promotion of $\langle x, T \rangle$ to T' a site finds and updates (at its leisure) each reference to $\langle x, T \rangle$ with a reference to $\langle x', T' \rangle$. This is the process of substitution mentioned in the previous bullet.
- Substitution of $\langle x, T \rangle$ is complete when all sites have replaced all references to $\langle x, T \rangle$ with a reference to $\langle x', T' \rangle$ and there are no references of $\langle x, T \rangle$ in-flight between sites.
- When `isolatedObject($\langle x, T \rangle$)` terminates, the substitution of $\langle x', T' \rangle$ for $\langle x, T \rangle$ has completed and the *home* site sends a *substitution complete* message, `SUBSTITUTE($\langle x, T \rangle \rightarrow \langle x', T' \rangle$, complete)`, to each site that referenced $\langle x, T \rangle$. That is, a *substitution complete* message for $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is sent to each site that the *home* site sent a substitution message.
- On receipt of the *substitution complete* message for $\langle x, T \rangle$ a site removes $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ from its substitution table.

Since substitution is not instantaneous across all sites of the distributed system some sites continue to operate with references to $\langle x, T \rangle$ after substitution has begun at the *home site* for $\langle x, T \rangle$ and $\langle x', T' \rangle$. The substitution table entry at the *home site* represents a reference to the object $\langle x', T' \rangle$ which ensures that the object x' and the train T' are not reclaimed while the system is in an inconsistent state. The reference is deleted when all references to $\langle x, T \rangle$ have been replaced with references to $\langle x', T' \rangle$

Table 6.2 lists the substitution events that occur at a site A due to the substitution protocol. In Section 6.3 the site actions on jobs and tasks associated with these events, are described.

| Description | Substitution Event |
|--|---|
| A re-associates $\langle x, T \rangle$ | Re-associate $\langle x, T \rangle, \langle x', T' \rangle$ |
| A sends a substitution message for $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ to site B | Send substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$), $\rightarrow B$ |
| A receives a substitution message for $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ | Receive substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) |
| A adds $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ to its substitution table | Add substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) |
| A removes $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ from its substitution table | Remove substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) |
| S replaces a reference to $\langle x, T \rangle$ with a reference to $\langle x', T' \rangle$ | Substitute $\langle x', T' \rangle, \langle x, T \rangle$ |

Table 6.2 - Object Substitution Events

6.2.3.3 A Root Set for Car Collection

The collection of a car C of train T (written C_T), at a site S , involves the re-association of each potentially live object x in C_T to a car of a train T' that references x , where $T' \geq T$. The root set for collection of C_T is defined as a structure that holds

references to each of the set of objects in C_T that are referenced from outside C_T . The structure is called the Re-Association List (RAL) and the objects it references are known as *externally referenced* objects. The RAL can be thought of as remset for a car. However, while a remset maintains information relating to the trains that hold references to objects in a car and the location of references that must be updated on object-re-association, the RAL for a car contains information relating only to the trains that reference objects in that car.

The externally referenced objects in a car are referenced either from a local root at S , from a root at any remote site or from another car (which is either local to S or on a remote site). The site S can compute exactly, the set of objects in C_T that are externally referenced locally²², while it can only maintain a conservative approximation to the set of objects in C_T that are referenced from remote sites.

In general terms, the more out-of-date the RAL the longer the delay in garbage identification. Intuitively, a mechanism that maintains an RAL which is as up-to-date as possible has a higher message complexity and is more computationally intensive than a mechanism which provides a more out-of-date RAL. This is seen as a trade-off in the work done to maintain a safe RAL against the delay in garbage identification. A minimal set of rules governing remset maintenance is described here. This is as simple as possible at the expense of potentially delaying garbage identification.

There are four rules relating to the maintenance of the RAL for the car C_T :

²² By examining the root set at S at car collection time and through use of a write barrier (for instance) to trap the creation and deletion of inter-car references at S .

1. The RAL for C_T must contain an entry (identifying the referent train) for each externally referenced object in C_T . This ensures that the RAL represents a safe root set for car collection. Note that the RAL may contain entries for objects that are no longer externally referenced and need only contain a single entry for any externally referenced object. While this may delay collection of this object (and those in its transitive closure) it does not affect completeness of the collector as a whole since the train reclamation mechanism is complete and is independent of the RAL maintenance mechanism. The first RAL entry for an object x in C_T may be added at any point between the creation of the first inter-car or root reference to x and the collection of C_T .
 - It is sufficient for safety to maintain a single RAL entry for each *externally referenced* object even if the entry is out-of-date.
 - Inter-car references, to an object $\langle x, T \rangle$, are created either by the garbage collector on object re-association or by the mutator and can be identified by techniques such as a write barrier. An RAL entry for $\langle x, T \rangle$ is added (if one does not exist) or replaced (if the referent train is younger) on the creation of such an inter-car reference.
 - When a reference to an object $\langle x, T \rangle$ is exported to a remote site S , it is either discarded by S , stored in a train at S or stored in a local root at S . To maintain the RAL as a root set there must be an entry for $\langle x, T \rangle$. Since the train number of the (potential) reference from S is unknown safety can be maintained if the RAL entry for $\langle x, T \rangle$ is recorded with T 's train number.

2. For each object x (that is garbage, in a global sense) in C_T that already has an RAL entry, the RAL must eventually contain an entry for x 's youngest referent train. This guarantees that x is eventually promoted to its youngest referent train.
 - To achieve this, it is sufficient to trap inter-car references on object re-association, since every car is collected eventually. Where such an inter-car reference crosses a site boundary, the remote site may communicate the existence of the reference, to an object x , to the site holding x through an asynchronous message. This is called an *RAL update message* and contains the referent train number and the reference to the object $\langle x, T \rangle$.
 - There is a special case where the reference is from a remote root. Since there is no referent train, the RAL update message indicates that the reference is from such a root.
3. The RAL must identify those objects that are referenced from roots (either local or remote). While the RAL's view of the set of objects in C_T referenced by remote roots may be out-of-date it must eventually be complete. Completeness is required for two reasons. Firstly it ensures that root reachable objects in C_T are eventually identified and therefore promoted to a younger train, thus not preventing the train T from becoming isolated. Secondly it ensures that objects (which have been referenced from a root at some point) eventually stop being promoted after they become garbage and are therefore identified as garbage.
 - A *Root Reference* RAL entry identifies an object as being referenced from a root and identifies the referencing site.

- Root references can be discovered at car collection time by examining the local root set. For each local object referenced by the local root set a *Root Reference* entry is added to the object's RAL. For each non-local object an RAL update message is sent to the object's site. The update contains a *Root Reference* RAL entry for the object. The set *Remote Root Referenced Objects* is defined as containing an entry for each object for which an update has been sent.
 - Each object in the car that has a *Root Reference* RAL entry for this site and that is not currently in the local root set, is promoted to a younger train and its *Root Reference* RAL entry (in its destination car) replaced with an RAL entry for its new train.
 - For each remote object in the *Remote Root Referenced Objects* set, that is not currently in the root set, (the set entry is removed and) an RAL Update message is sent to indicate that the object is no longer root referenced from this site. On receipt of such an update the receiving site replaces the referenced object's *Root Reference* RAL entry for the sending site with a *Root Reference* RAL entry for the object's local site, if one does not already exist, otherwise the *Root Reference* RAL entry for the sending site is removed.
 - The zero-progress problem is avoided with this mechanism because any object that has been referenced by a root is guaranteed to be re-associated to a younger train.
4. If object isolation occurs for an object x , in C_T , while x is not being substituted then all RAL entries for x can be safely discarded. This is an optimisation.

Figure 6.8 below illustrates suitable RAL entries for two cars at a site.

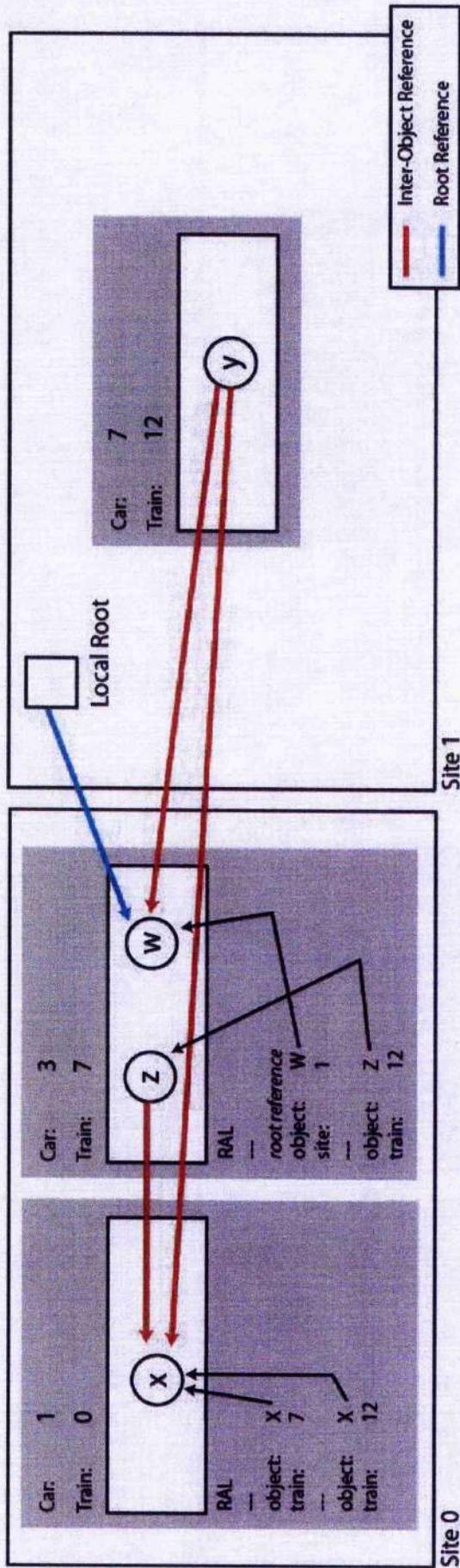


Figure 6.8 - Example RAL Entries

The astute reader will be aware that these rules can lead to a situation where an intra-car cycle contained in a car C_T is not reclaimed when C_T is collected. This arises if any object in the cycle has ever been referenced by a remote site, a local root or by a local car C'_T (when C'_T was collected). The RAL entry for such an object will never be removed since the object is part of a cycle and so will never become isolated. However, RAL entries do not prevent trains from becoming isolated and therefore if an object continues to have an RAL entry after it has become garbage it will still be reclaimed when the train is collected. Object and train isolation detection are based on tracking every reference in the distributed system while RAL maintenance is based on having just enough information to accurately re-associate objects. RAL entries for cars of an isolated train simply represent an out-of-date view of the object graph. To allow for the collection of such intra-car cycles it is necessary for a site to detect when there are no external references to a particular object. In Chapter 7 it is shown how the DTA implementation can detect the absence of inter-site references to any object and how write barrier techniques can be used to detect the absence of local inter-car references. This approach allows us to safely remove the RAL entries for objects at times other than object isolation.

6.2.4 A Summary of the DTA Mappings

The set of references into a train constitutes distributed shared state. A train is isolated when there are no references into it. The isolated train state is globally stable (through design) and a DTA can be used to detect this state.

The DTA mapping is as follows:

- A train is a job, called an isolated train job.

- A task of the isolated train job for a train T , written $\text{isolatedTrain}(T)$, is a reference to an object in T from an object in another train.

When a site manipulates a reference it must modify tasks of the appropriate isolated train job accordingly. To do this, the site must be able to tell which train an object is in from the object's reference. To this end a distributed object-to-train map is defined. This map is also distributed shared state in the DMOS collector. Each site maintains a table containing an entry that maps each reference the site holds to the train holding the object. When an object is promoted this state must be updated, and the substitution protocol is used to achieve this. That is, at the site holding the object an object in the new train is logically substituted for the object in the old train and then each of the referencing sites is informed. On learning of the substitution a site updates its mapping table and replaces each reference to the old object with a reference to the new object. Each site maintains a substitution table that holds an entry for each object being substituted at the site. The table entry for a substituted object constitutes a reference to the object and a reference into that object's train. The substitution table entry is held by each site while substitution is in progress, thus ensuring that the 'new' object (that is being substituted) is not collected while the system is in an inconsistent state.

When substitution is complete, the distributed object to train map will be consistent, all of the tables will be up-to-date, and there will be no references to the substituted object. That is the substituted object is isolated. The isolated object state is globally stable and a DTA is used to detect this state.

The second DTA mapping is as follows:

- An object in a train is a job, called an isolated object job.

- A task of the isolated object job for an object x in a train T , written $\text{isolatedObject}(\langle x, T \rangle)$, is a reference to $\langle x, T \rangle$.

6.3 The Stepwise Refinement of DMOS

The mappings described in Section 6.2 yield two collection mechanisms within DMOS. These are:

- Collection of a whole train which relies on detection of an *isolated train*.
- Collection of a car which relies on detection of an *isolated object*.

Here the site actions on jobs and tasks in DMOS are defined through a process of stepwise refinement. In both cases the termination of a job corresponds to detecting that a set of objects is unreachable and may be reclaimed. In the first case it is an entire train and in the second it is a single object.

The astute reader will be aware of the subtlety in these mappings in that there is now an instantiation of a DTA implementation for every object and every train in the system. This is, of course, not as bad as it first appears in that the system is asynchronous and all messages and detections may be combined and buffered for efficiency.

The (reference manipulation and substitution) events due to mutator and garbage collector activity have already been identified in Table 6.1 and Table 6.2 above. The stepwise refinement identifies the site task actions through the three layers that correspond to these events.

The DMOS collector can be decomposed into three logical layers:

1. Isolated object detection.
2. Car collection.
3. Isolated train detection.

The DMOS algorithm is derived by the stepwise refinement of the above bullets. Layer 1 detects object isolation at the site at which the object is resident (its home). Here the site actions on object isolation jobs and tasks for reference events due to mutator activity are described. Layer 2 is concerned with car collection and defines the actions on isolated object jobs and tasks for substitution events due to object re-association on car collection and refines the reference event actions from Layer 1. Layer 3 is concerned with isolated train detection and refines the substitution actions from Layer 2 and further refines the reference event actions from Layer 1.

6.3.1 Layer 1: Object Isolation

Layer 1 provides an object isolation detection mechanism which detects objects that are not referenced anywhere in the distributed system. The reference manipulation events at a site *A* are mapped onto site actions on isolated object tasks for an object *x* in train *T* in Table 6.3 below. The actions in this table are later refined for the second and third layers. That is, for each event the corresponding actions are expanded to include actions for the car collection and train isolation detection mechanisms.

| Event | Action |
|--|--|
| Create $\langle x, T \rangle$ | A creates a job <code>isolatedObject($\langle x, T \rangle$)</code> |
| Copy <code>ref($\langle x, T \rangle$)</code> | A creates a new task of <code>isolatedObject($\langle x, T \rangle$)</code> |
| Delete <code>ref($\langle x, T \rangle$)</code> | A completes a task of <code>isolatedObject($\langle x, T \rangle$)</code> |
| Send <code>ref($\langle x, T \rangle$), $\rightarrow B$</code> | A sends a task of <code>isolatedObject($\langle x, T \rangle$)</code> to <i>B</i> |
| Receive <code>ref($\langle x, T \rangle$)</code> | A receives a task of <code>isolatedObject($\langle x, T \rangle$)</code> |

Table 6.3 – Object Isolation Action for Reference Events

When the site holding object $\langle x, T \rangle$ detects termination of `isolatedObject($\langle x, T \rangle$)` then $\langle x, T \rangle$ is unreferenced and may be safely collected. While `isolatedObject($\langle x, T \rangle$)` is not terminated, $\langle x, T \rangle$ may potentially be referenced and must be maintained.

6.3.2 Layer 2: Car Reclamation

Layer 2 is concerned with the collection of a car using the re-association rules. The effects of this are two-fold: it frees up a car locally and it eventually traps cyclic garbage in an isolated train.

The site actions on isolated object jobs and tasks due to object substitution events are listed in Table 6.4 below.

| Substitution Event | Action |
|---|---|
| Re-associate $\langle x, T \rangle, \langle x', T' \rangle$ | A creates the job $\text{isolatedObject}(\langle x', T' \rangle)$ |
| Send substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$), $\rightarrow B$ | A sends a task of $\text{isolatedObject}(\langle x', T' \rangle)$ to B |
| Receive substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) | A receives a task of $\text{isolatedObject}(\langle x', T' \rangle)$ |
| Add substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) | A creates a task of $\text{isolatedObject}(\langle x', T' \rangle)$ |
| Remove substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) | A completes a task of $\text{isolatedObject}(\langle x', T' \rangle)$ |
| Substitute $\langle x', T' \rangle$ for $\langle x, T \rangle$ | A creates a task of $\text{isolatedObject}(\langle x', T' \rangle)$ and completes a task of $\text{isolatedObject}(\langle x, T \rangle)$ |

Table 6.4 - Object Isolation Actions for Substitution Events

In Table 6.5 the site actions, on isolated object tasks and jobs for reference events (from Table 6.3), are refined to take account of the object substitution protocol. Note that the events from Table 6.3 that are not listed in Table 6.5 are unchanged in the presence of substitution.

| Reference Event | Action |
|---|---|
| COPY ref($\langle x, T \rangle$) | <p>If $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is in the substitution table then</p> <p style="padding-left: 40px;"><i>A</i> creates a task of isolatedObject($\langle x', T' \rangle$)</p> <p>else</p> <p style="padding-left: 40px;"><i>A</i> creates a task of isolatedObject($\langle x, T \rangle$).</p> |
| SEND ref($\langle x, T \rangle$), $\rightarrow B$ | <p>If $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is in the substitution table then</p> <p style="padding-left: 40px;"><i>A</i> sends a task of isolatedObject($\langle x', T' \rangle$) to site <i>B</i></p> <p>else</p> <p style="padding-left: 40px;"><i>A</i> sends a task of isolatedObject($\langle x, T \rangle$) to site <i>B</i>.</p> |

Table 6.5 - Refinement of Actions on Reference Events from Layer 1

In summary, object substitution for a promoted object $\langle x, T \rangle$, which is being substituted for $\langle x', T' \rangle$, works as follows. The first step is to add a reference to $\langle x', T' \rangle$ to the substitution table at the home site. This is initially the only reference to $\langle x', T' \rangle$ and it will be maintained until substitution is complete, thus ensuring that $\langle x', T' \rangle$ is not reclaimed while the system is in an inconsistent state (with references to $\langle x, T \rangle$ and $\langle x', T' \rangle$). Each site with a reference to $\langle x, T \rangle$ is informed of the re-

association and sent a task of $\text{isolatedObject}(\langle x', T' \rangle)$. These sites may now, at their leisure, find and update any instance of $\text{ref}(\langle x, T \rangle)$ with $\text{ref}(\langle x', T' \rangle)$ thus creating new tasks of $\text{isolatedObject}(\langle x', T' \rangle)$ and completing all of its tasks of $\text{isolatedObject}(\langle x, T \rangle)$. While substitution of $\langle x', T' \rangle$ for $\langle x, T \rangle$ is ongoing each site holds $\text{ref}(\langle x', T' \rangle)$ in its substitution table and hence holds at least one task of $\text{isolatedObject}(\langle x', T' \rangle)$. This is necessary since if a reference to $\langle x, T \rangle$ is received, this site is then in a position to create a reference to $\langle x', T' \rangle$ (and hence a task of $\text{isolatedObject}(\langle x', T' \rangle)$ if it decides to copy the received reference. This guarantees that the substitution will eventually terminate. On termination of $\text{isolatedObject}(\langle x, T \rangle)$ the substitution process is complete and each site is told to remove its substitution table entry.

The mechanism by which the set of sites that hold a reference to $\langle x, T \rangle$ is calculated is implementation dependent. Later it is shown how the DTA implementation used for isolated object detection can provide this information through an asynchronous protocol.

6.3.3 Layer 3: Isolated Train Detection

Layer 3 provides train isolation detection by tracking references into trains. Table 6.6 refines the site actions for substitution events with actions on isolated train tasks and jobs. Note that the assumption made here is that an object x in train T has been promoted to train T' . If the substitution does not involve promotion then $T=T'$ and there are no actions on train tasks.

| Event | Action |
|---|--|
| Re-associate $\langle x, T \rangle, \langle x', T' \rangle$ | <p>A creates <code>isolatedObject($\langle x', T' \rangle$)</code>.</p> <p>For each reference in $\langle x, T \rangle$ to an object P, where P is in train T, A creates a task of <code>isolatedTrain(T)</code>.</p> <p>For each reference in $\langle x, T \rangle$ to an object P, where P is in train T', A completes a task of <code>isolatedTrain(T')</code>.</p> |
| Send substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$), $\rightarrow B$ | <p>A sends a task of <code>isolatedObject($\langle x', T' \rangle$)</code> to B.</p> <p>A sends a task of <code>isolatedTrain(T')</code> to B.</p> |
| Receive substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) | <p>A receives a task of <code>isolatedObject($\langle x', T' \rangle$)</code>.</p> <p>A receives a task of <code>isolatedTrain(T')</code>.</p> |
| Add substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) | <p>A creates a task of <code>isolatedObject($\langle x', T' \rangle$)</code>.</p> <p>A creates a task of <code>isolatedTrain(T')</code>.</p> |
| Remove substitution($\langle x, T \rangle \rightarrow \langle x', T' \rangle$) | <p>A completes a task of <code>isolatedObject($\langle x', T' \rangle$)</code>.</p> <p>A completes a task of <code>isolatedTrain(T')</code>.</p> |
| Substitute $\langle x', T' \rangle$ for $\langle x, T \rangle$ in train T'' | <p>A creates a task of</p> |

| | |
|--|--|
| | <p>isolatedObject($\langle x', T' \rangle$) and completes a task of isolatedObject($\langle x, T \rangle$).</p> <p>If $T'' \neq T'$, A creates a task of isolatedTrain(T')</p> <p>if $T'' \neq T$, A completes a task of isolatedTrain(T).</p> |
|--|--|

Table 6.6 - Train Isolation Actions for Object Substitution

The five reference events for an object x in train T at a site A correspond to actions on tasks of isolated train and isolated object jobs and tasks as shown in the Table 6.7 below.

| Event | Action |
|---|---|
| CREATE $\langle x, T \rangle$ | A creates a new task of isolatedTrain(T) and creates a job isolatedObject($\langle x, T \rangle$). |
| COPY ref($\langle x, T \rangle$) [to train U] | <p>If $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is in the substitution table then</p> <p style="padding-left: 40px;">A creates a new task of isolatedObject($\langle x', T' \rangle$)</p> <p>else</p> <p style="padding-left: 40px;">A creates a task of isolatedObject($\langle x, T \rangle$).</p> <p>If $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is in the substitution</p> |

| | |
|---|---|
| | <p>table then</p> <p>if $T' \neq U$ then A creates a new task of <code>isolatedTrain(T')</code>.</p> <p>else</p> <p>if $T \neq U$ A creates a new task of <code>isolatedTrain(T)</code></p> |
| <p>DELETE ref($\langle x, T \rangle$)</p> <p>[in train U]</p> | <p>A completes a task of <code>isolatedObject($\langle x, T \rangle$)</code>.</p> <p>If $U \neq T$ A completes a task of <code>isolatedTrain(T)</code>.</p> |
| <p>SEND ref($\langle x, T \rangle$), $\rightarrow B$</p> | <p>If $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is in the substitution table then</p> <p>A sends a task of <code>isolatedObject($\langle x', T' \rangle$)</code> to site B</p> <p>else</p> <p>A sends a task of <code>isolatedObject($\langle x, T \rangle$)</code> to site B.</p> <p>If $\langle x, T \rangle \rightarrow \langle x', T' \rangle$ is in the substitution table then</p> <p>A sends a task of <code>isolatedTrain(T')</code> to site B.</p> <p>else</p> |

| | |
|--------------------|---|
| | A sends a task of isolatedTrain(T) to site B. |
| RECEIVE ref(<x,T>) | A receives a task of isolatedObject(<x,T>). A receives a task of isolatedTrain (T). |

Table 6.7 – Reference Events and Isolated Train Actions

When the home site detects termination of isolatedTrain(T) it informs each site that holds cars of T through asynchronous messages. The means by which the home site calculates the set of sites that hold cars of the train T (at the point of termination) is not central to the DMOS algorithm and as such is implementation dependent. One particular mechanism that makes use of the implementation of the Task Balancing DTA is described in Chapter 7. In general, the home site is required to construct the set of sites that have (at some point) held a task of isolatedTrain(T) since, by definition, this set includes the set of sites holding cars of train T when termination is detected.

6.3.3.1 Train Tasks and Remote Dereference Operations

Figure 6.9 below illustrates a potential problem between the DPBASE distributed cache system and the isolated train detection DTA mapping. The problem is illustrated through an example.

In this example site B holds a reference in train T2 to the location object L at site A in train T1. Assume that the object at B holding the reference to L is reachable from a root at B, although this is not shown in the figure. Location L holds a reference to an object x, also in train T1 at site A. If B dereferences L, a reference to <x,T1> is sent

from A to B but the site A holds no tasks of $T1$. In fact this situation can occur in both the remote dereference mechanism and the site to site object faulting mechanism.

The solution for this particular example is to treat the message from B to A , requesting the dereference, as a reference send event for $\langle L, T1 \rangle$. Therefore site A holds a task of $\text{isolatedTrain}(T1)$ when it comes to send the reference to $\langle x, T1 \rangle$ to site B . The isolated train task received at site A completes when the remote dereference request has been serviced, since site A doesn't hold a reference to L . This solution also applies to the site to site fault request mechanism, by treating the send of the fault request message as a send event for a task of the train holding the requested object.

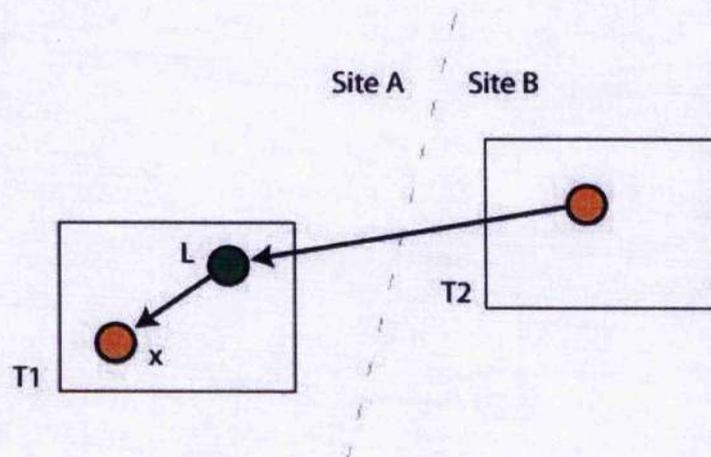


Figure 6.9 - Remote Dereference

6.3.4 A DMOS Garbage Collection Cycle

A DMOS garbage collection cycle at a site S , forces the reclamation of at least one car at S and, as stated in the UMOS rules, each car is eventually the target for collection. A simple mechanism for guaranteeing that each car is eventually collected is to use a round-robin algorithm in selecting a car for collection. For instance cars may be collected in the order that they were created. Although clearly,

for each inter-train garbage cycle there exists one or more particular orderings of car collection that will collapse the cycle into a single train with the minimum of car collections. The order in which cars are collected is ultimately a matter of policy.

Collection of a car C in train T , written C_T , at site S proceeds, for each object x , as follows.

1. Examine the local root set and add any necessary RAL entries and send appropriate RAL Update messages.
2. If the RAL contains one or more entries for x then it is re-associated in accordance with the UMOS rules.
3. The re-association of x from train T to T' corresponds to the substitution of $\langle x, T' \rangle$ for $\langle x, T \rangle$ iff $T \neq T'$. It is assumed that re-association within a train does not require substitution.
4. Transfer C_T 's RAL entries for x to the new car of x . If there is a Root Reference RAL entry (for site S) for x and x is not in the local root set then, if no other RAL entry exists for x , replace this RAL entry with an entry for train T , otherwise discard the Root Reference RAL entry.
5. If $T \neq T'$, for each reference R in x , to an object y in train U .
 - o If $U=T$ then create a task of `isolatedTrain(T)`.
 - o If $U=T'$ then complete a task of `isolatedTrain(T')`.

The target car is reclaimed after all objects referenced by the RAL have been re-associated. Cars of an isolated train can be reclaimed at any time. Each object x , that is still in the car when it is reclaimed, is garbage. For each reference R (in x) to an object y in train U , complete a task of `isolatedObject(\langle x, T \rangle)` and if $U \neq T$ complete a task of `isolatedTrain(U)`. The target car can now be reused immediately.

6.4 Summary

To identify garbage, the DMOS algorithm refines to three situations where global knowledge is required.

- To reclaim a train, a site must be able to detect that there are no references into the train from outside of it.
- To reclaim a car, a site requires an approximation to the set of trains that hold references into the car. Cycles wholly within the car, with no external references, may be collected immediately.
- To reclaim an object, no special action is taken since objects are reclaimed when the cars holding them are reclaimed. However, the re-association rules change the cars and train with which particular objects are associated. In the face of this re-organisation, each site needs to be able to identify consistent views of the global state of a particular trains (for train collection). Identification of these consistent views is achieved by a logical substitution protocol and the detection of the (global) absence of references to individual objects.

In the presence of asynchrony, local information may not be up-to-date and therefore global information difficult to glean. However, the first and third situations correspond to stable properties since once they become true they can never become untrue. Such stable properties may be mapped onto a distributed termination algorithm. The second situation is not so easily dealt with. Car collection, which we need to make progress, is an entirely local operation but is affected by the presence of mutators that may be continually changing the global position. The trick is to

ensure that the local collectors have enough information such that re-association is safe and that eventually the required global information will reach them.

The specific contribution of this author to the work presented here is as follows:

- The development of the distribution of UMOS such that the isolated train state is globally stable.
- The definition of the two DTA mappings.
- The development of the RAL maintenance rules.
- The specification of the three-layered collector architecture.
- The stepwise refinement of the collector actions.

7 Implementing the DMOS DTA Mappings

This chapter describes the implementation of the DTA mappings developed in Chapter 6 for isolated train and isolated object detection. Both mappings are implemented with the Task Balancing (TB) DTA and the implementation is targeted at the distributed ProcessBase system (DPBASE). The implementation breaks down into a number of areas:

- car and train implementation in DPBASE (Section 7.1.1 and Figure 7.1 explain how objects are associated with cars at a site);
- TB implementation for isolated train and isolated object detection. Note that an incremental update TB implementation is used for both;
- idleness detection for isolated train and isolated object jobs;
- implementation of the substitution protocol;

The DTA mapping for DMOS presented in Chapter 6 is suitably generic for any DTA to be used in the implementation of the DMOS collector. Furthermore, there is no requirement that the same DTA be used for isolated object and isolated train detection. However it is necessary to ensure that the DTAs do not interfere with each other. Any implementation of the mappings described in Chapter 6 is required to run a DTA for each object, for isolated object detection, and for each train, for isolated train detection.

To reclaim a train collector must be able to detect when there are no references into the train. To detect an isolated object the collector must detect when there are no references to that object. The DTA mappings effectively define two distributed reference counting mechanisms that detect isolated trains and object. The first of these counts references to individual objects while the second counts references into

trains. This chapter describes the implementation of these reference counters with the Task Balancing DTA.

The DTA mappings define the distributed partitioned state of the distributed collector and the actions taken at each site in order to detect the globally stable properties of this state. In this implementation, write barrier techniques are used to trap object graph mutations and to drive the DTAs. The overall aim in this chapter is to bridge the gap from the abstract mappings developed for isolated object and isolated train detection in Chapter 6 to a concrete implementation using a specific DTA and operating in a particular distributed environment.

7.1 Cars and Trains in Distributed ProcessBase

Before presenting the implementation of the isolated object and isolated train detection DTA mappings a brief résumé of the key aspects of the DPBASE system is given:

- Each site runs a ProcessBase interpreter which operates over a local object cache.
- Objects are created in the local object cache where they are addressed by their local cache address (CA).
- When a reference to an object x is exported from its creating site, x is allocated a distributed address (DA). DAs are two part addresses consisting of a site identifier for the object's *home* site and a symbolic identifier for the object x . Each site maintains an address translation table, called the $DA_{\text{sym}} \rightarrow CA$ table, which maps symbolic identifiers to objects at that site. An object referenced from a site B , where B is not the home site of x , is known at B as a *remote object*.

- A site can request a copy of an object x from the object's *home* site. Such a copy is known as a *remote resident* object at a site holding a copy. Each site maintains a second address translation table called the $DA \rightarrow CA$ table which, for a *remote resident* object x at a site B , maps x 's DA to the CA for the copy of x at B .

7.1.1 Cars

Objects (remote resident and local) are associated with cars through a two-way mapping maintained by the interpreter at a site. The car-to-object mapping identifies the objects that are in a particular car and is implemented by a car data structure which contains the train identifier for the train that the car is in and an expandable array containing the cache address (CA) for each of the objects in the car. The object-to-car mapping for an object x in car c is achieved by storing a pointer in the object x to the car data structure for c .

An object can be re-associated from a car c to a car c' by removing its CA from the address array in c , adding its CA to the address array in c' and updating the car pointer in the object with a reference to c' .

The two-way mapping that associates object with cars is illustrated in Figure 7.1 below.

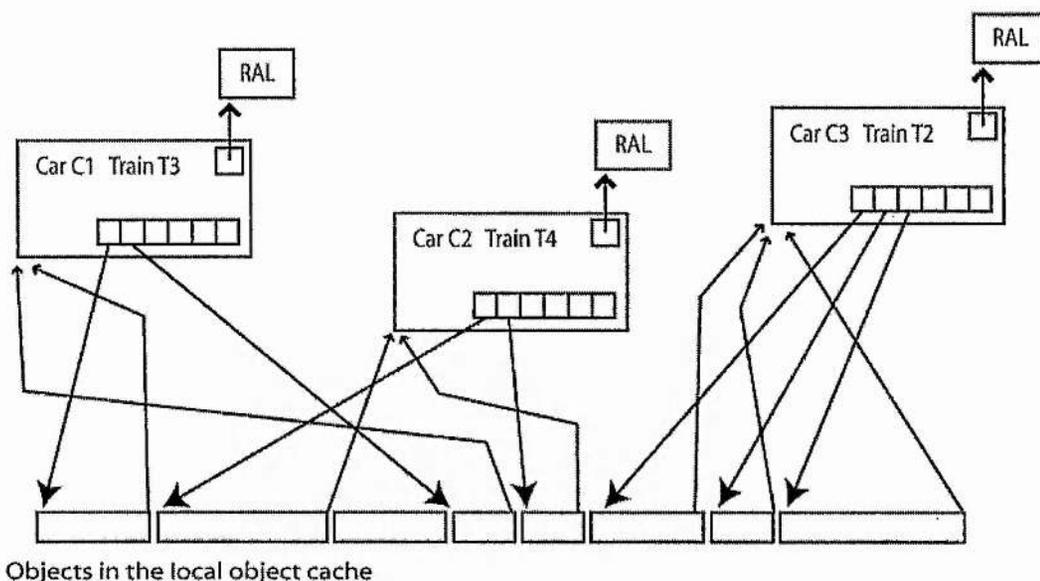


Figure 7.1 - Associating Objects with Cars

Cars are therefore logical sets of objects and an object can be re-associated without having to be moved to a different local cache address. This is a different approach from that adopted by the original MOS collector and its variants (PMOS and previous versions of DMOS) which have always been copying collectors.

The logical association of cars and objects reduces the requirements imposed by the distributed collector on each of the participant sites. The impact of object re-association on the local addressing mechanisms at a site is reduced to the point that when an object x is re-associated from car c to car c' only the address arrays c and c' and the object's car pointer need to be updated. More importantly these changes are restricted to x 's home site.

However if an object is re-associated to a younger train (promoted) then the re-association potentially has a global impact since the distributed object-to-train map must be updated.

The *substitution* protocol is defined using an abstraction whereby an object $\langle x', T' \rangle$ is logically substituted for the object $\langle x, T \rangle$ when $\langle x, T \rangle$ is re-associated. The

intuition behind the protocol is that if every reference to $\langle x, T \rangle$ in the distributed system is replaced (logically at least) with a reference to $\langle x', T' \rangle$ then any distributed state that was made inconsistent (due to the re-association) will eventually be returned to a consistent state.

The re-association of an object within a train does not affect any distributed state (as shown above) and thus substitution is required only when an object is promoted. In effect the *substitution* protocol is used to bring the object-to-train map up-to-date following the promotion of an object. The substitution due to the promotion of an object x from a train T to a train T' is written $\langle x, T \rangle \rightarrow \langle x, T' \rangle$.

7.1.1.1 Interpreter Local Objects

Local interpreter objects of the DPBASE system (for instance the distinguished local root object, the nil view, single character strings, stacks and thread control blocks) are not associated with cars at a site. The root object, nil view and single character strings never become garbage since they are maintained for the entire run-time of the interpreter at a site and therefore to continually re-associate these object to increasingly younger trains is a waste of time.

Since the stack objects are not associated with a car, references on the stack of each thread represent root references for the DMOS collector. The set of stack objects at a site therefore constitutes the local root set at that site for DMOS car collection.

7.1.1.2 Car Collection Overview

A DMOS garbage collection cycle at a site *A* involves the collection of one or more cars at *A*. To collect a car a site needs to know of references into the car. A reference into a car *C* (where *C* is at a site *A*) falls into one of four categories:

1. A root reference at *A*. That is, a reference held on the stack of a thread executing at site *A*.
2. An inter-car reference at *A*.
3. A root reference at a remote site.
4. An inter-site inter-car reference from a site *B*.

References in the first category are discovered at car collection time by examining the local root set. References in any of the other three categories are recorded in the car's Re-Association List (RAL). The RAL maintenance mechanism is described in Section 8.1 below. Therefore the combination of the local root set (described in Section 7.1.1.1 above) and the RAL for a car *C* constitute the root set for the collection of *C*.

On collection of a car *C* each object in *C* referenced by the local root set or reachable from an entry in *C*'s RAL is re-associated to a different car in accordance with the re-association rules. Any object left in the car after re-association is garbage and the reference to such an object is removed from *C*'s reference array. The car structure may now be reused.

7.1.2 Trains

A train is represented by a data structure that holds a train identifier, a pointer to a list of car data structures and isolated train detection data. Train structures are held in one of two train tables at a site called the *local train table* and the *remote train table*.

The *local train table* holds references to the train structures for all locally created trains while the *remote train table* holds references to the train structures for each train T created at a remote site where the local site holds a car of T or holds a reference to an object in T . A *remote train table* entry for a train T is added on receipt of the first reference into T .

The mechanism for removing train table entries depends on the isolated train detection implementation. When an isolated train is detected, each object in the train is garbage and all of its cars may be reclaimed.

The *local train table* and a train data structure (for a train T at a site A) are illustrated in Figure 7.2 below. The *local task count* value $TC_A(T)$ and the TB task counts for $\text{isolatedTrain}(T)$ are explained in Sections 7.2.1.1 and 7.2.1.4 respectively.

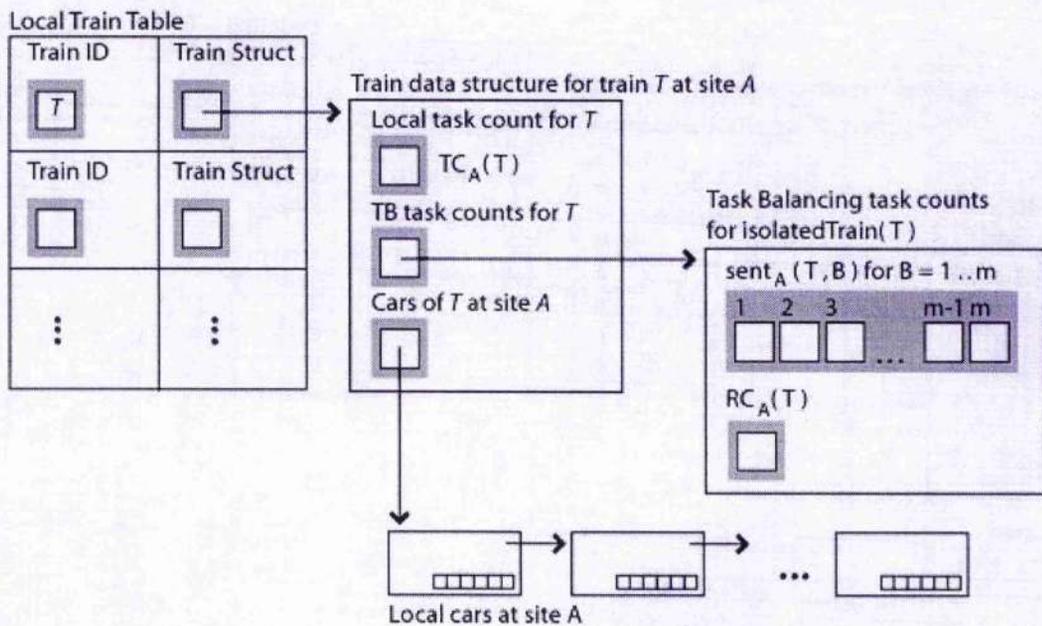


Figure 7.2 - The Local Train Table and a Train Structure

7.2 Isolated Train and Object Detection with Task

Balancing

An instance of the TB DTA runs for each object, for isolated object detection, and for each train, for isolated train detection. Three key points (initially made in Chapter 6) are repeated here;

- The site that creates a train T is defined as the TB *home* site of T for isolated train detection. In this implementation the home site for a train T is fixed, although the substitution protocol can be adapted to allow a train's *home* site to be changed.
- The site holding an object x is defined as the TB *home* site of $\langle x, T \rangle$ for isolated object detection and object substitution. This is initially the creator site for $\langle x, T \rangle$. The substitution protocol allows for object migration but no discussion is presented here. The *home* site for an object $\langle x, T \rangle$ is encoded in its address, as described in Chapter 3.
- The idle state is used to trigger the sending of TB updates for both isolated object and isolated train jobs. A site A is idle for $\text{isolatedObject}(\langle x, T \rangle)$ if A holds no reference to $\langle x, T \rangle$. Similarly the site A is idle for $\text{isolatedTrain}(T)$ if A holds no reference to an object in T .

7.2.1 Isolated Train Detection

When a site creates a new train T a new entry is added to the *local train table* and a corresponding TB job $\text{isolatedTrain}(T)$ is created. Train T may be reclaimed when $\text{isolatedTrain}(T)$ terminates.

7.2.1.1 Idleness Detection for Isolated Train Jobs

A local task counting mechanism is used to determine idleness for isolated train jobs at a site. For each job j (of which a site A holds a task) A maintains a local task count value, $TC_A(j)$. When a task of j is first created at site A , $TC_A(j)$ is initialised to one. On creation of a local task of j , $TC_A(j)$ is incremented and on completion of a task, $TC_A(j)$ is decremented. When $TC_A(j) = 0$ then the job j is idle at A . The notation $TC_A(T)$ will be used as a shorthand for $TC_A(\text{isolatedTrain}(T))$ in the rest of this chapter.

Since a task of $\text{isolatedTrain}(T)$ constitutes a reference into the train T , isolated train task counting is simply reference counting operating over references into trains. Idleness detection for isolated train jobs is implemented through the use of a write barrier that traps the creation of inter-train references on reference field update, and the deletion of inter-train references when a reference field is over-written. This implementation uses a deferred reference counting mechanism to account for reference values that are pushed onto and popped from a thread's pointer stack. This represents an optimisation over the write barrier technique and as such is explained later (see Section 7.2.3 below).

Task count values are also modified when objects are reclaimed on car collection. Effectively each reference in a reclaimed object is deleted.

7.2.1.2 The Object-to-Train Map

In order to detect the creation of inter-train references a site must be able to calculate the train holding any local or remote object from the reference to that object. This is trivial for local objects since the object's car, which identifies the train of which it is part, is referenced from the object itself. To calculate the train number for remote

objects a site maintains the object-to-train mapping table that allows the translation from a DA to the train number for the train holding the referenced object.

When a DA is sent between two sites the train number of the train holding the referenced object is also sent. To store the train numbers for remote objects each site maintains a third address translation table called the DA \rightarrow Count table (Figure 7.4). The table has this name because it is also used to hold object isolation task count data for remote objects.

7.2.1.3 Isolated Train Idleness Task Counts and Object Substitution

The substitution of an object $\langle x, T' \rangle$ for an object $\langle x, T \rangle$ (which corresponds to the promotion of an object x from train T to train T') requires that within each site of the DPBASE system each task of $\text{isolatedTrain}(T)$ due to a reference to $\langle x, T \rangle$ be replaced with a task of $\text{isolatedTrain}(T')$ which corresponds to a reference to $\langle x, T' \rangle$. Note that due to the implementation of cars and trains within sites no references are actually updated during this process.

At a particular site S the substitution of an object $\langle x, T' \rangle$ for $\langle x, T \rangle$ means that a task of $\text{isolatedTrain}(T')$ is created for each reference to $\langle x, T \rangle$ that is in any train except T' , and that a task of $\text{isolatedTrain}(T)$ is completed for each reference to $\langle x, T \rangle$ that is held in any train at S except T . The values $TC_S(T)$ and $TC_S(T')$ at S are updated as follows on the substitution of $\langle x, T' \rangle$ for $\langle x, T \rangle$:

- $TC_S(T') = TC_S(T') +$ the number of references at S to the object x held in any train except T'
- $TC_S(T) = TC_S(T) -$ the number of references at S to the object x held in any train except T

Therefore, for a site S to correctly update its isolated train task count values for trains T and T' , $TC_S(T)$ and $TC_S(T')$, on the substitution of an object $\langle x, T' \rangle$ for $\langle x, T \rangle$, S must calculate three values:

- the number of references to x held at S in the train T ;
- the number of references to x held at S in the train T' ;
- the number of references to x held at S in any other train.

The promotion at a site A of a remote resident object or a local object that has no DA requires that the *train reference count* values at A are modified as described above, however no substitution is required.

For each local and remote resident object x for which a site S holds a reference, S maintains an individual reference count for each train at S that holds a reference to x . These counts are maintained in a *train reference count* data structure which consists of an array of \langle train identifier, reference count \rangle tuples, with one element for each train at S that holds a reference to x . The notation $TRC_x[T]$ is used to denote the reference count value for references from a train T to an object x .

The write barrier which is used to maintain train task count values is extended to update the train reference count values for each object. On the creation of the first reference at a site S from a train T to an object x $TRC_x[T]$ is initialised with the value one. On the creation of each subsequent reference from train T to object x , $TRC_x[T]$ is incremented. On the deletion of a reference for a train T to an object x , $TRC_x[T]$ is decremented. The element recording $TRC_x[T]$ is removed from the *train reference count* data structure for x if $TRC_x[T] = 0$.

The *train reference count* data structure for an object x (local or remote resident) is held in x 's car. The car data structure is illustrated in Figure 7.3 below. This shows a

car C_1 of train T_6 where C_1 holds three objects x , y and z and where the object x is referenced from trains T_6 , T_4 and T_2

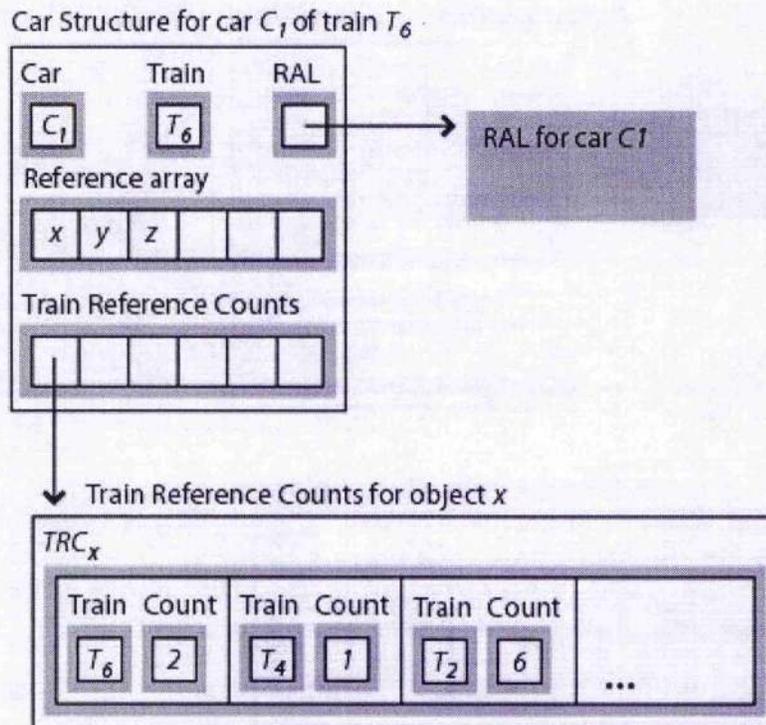


Figure 7.3 - The Car Data Structure

The *train reference count* data structure at a site A for a remote object with DA d is held in the $DA \rightarrow \text{Count}$ table entry for d at A . The $DA \rightarrow \text{Count}$ table is illustrated in Figure 7.4 below which shows the table entry for d at A where d is mapped to train T and is referenced from train's T_5 and T_1 at A . The table entry also holds a TB task count structure for the isolated object job associated with d (see Section 7.2.2.2).

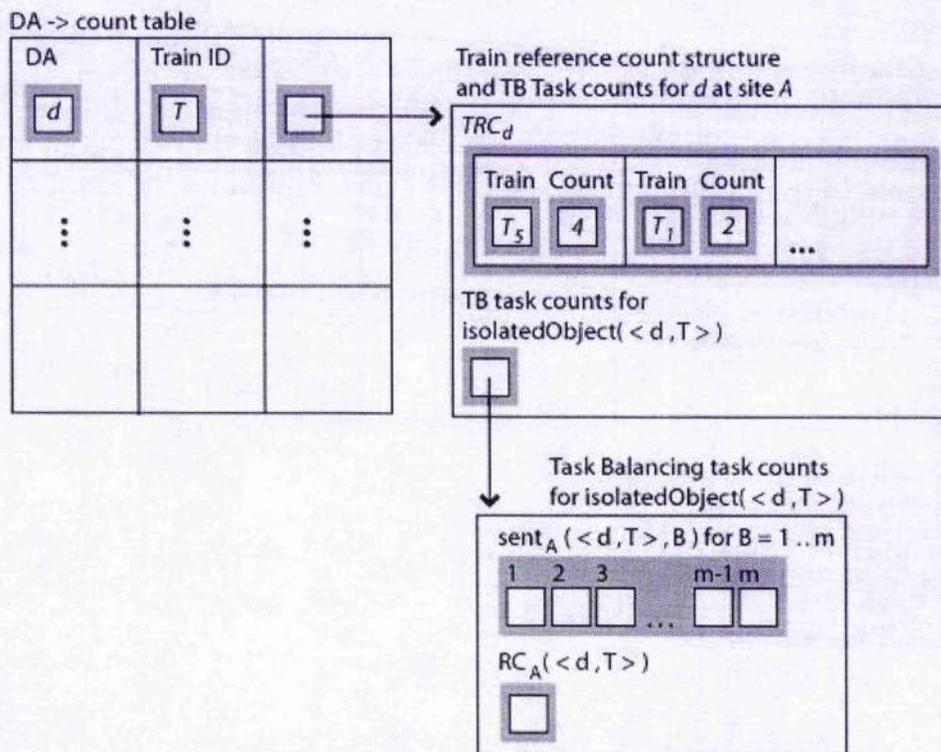


Figure 7.4 - The DA \rightarrow Count Table

7.2.1.4 Task Balancing Task Counts for Isolated Train Detection

Each site maintains a TB task count structure for each train created locally and for each train into which it holds a reference.

The TB task count structure for a train T at site A (where A holds a reference into T and A is not the *home* site for T) holds:

- an array of *sent* count values $count_A(T,B)$ with one element for each site B that A has sent a task of $isolatedTrain(T)$.
- the *received* count for $isolatedTrain(T)$ at site A , $received_A(T)$.
- the current local task count value for $isolatedTrain(T)$ at site A , $TC_A(T)$.

The task count structure for each train T at a site A , where A is not the home site of T , is stored in the *remote train table* at A . A task count structure for the train T is added

to the *remote train table* entry at a site A on receipt of a reference into T at A if no reference into T is currently held at A (i.e. if no task count structure exists). The value $received_A(T)$ is initially one, and is incremented on receipt of each subsequent reference to T .

If at anytime the write barrier detects that $TC_A(T) = 0$ an *update* is sent to the home site of T containing the *received/completed* value, $RC_T = received_A(T)$, for $isolatedTrain(T)$ at site A and each non-zero $count_A(T,B)$ value. The task count structure for $isolatedTrain(T)$ is then discarded at A .

The TB task count structure at the *home* site H of a train T holds:

- an array of *sent* count values $count_H(T, B)$ with one element for each site B for which an update from any site A has been received at H containing the value $count_A(T, B)$.
- the current local task count value for $isolatedTrain(T)$ at H , $TC_H(T)$

Note that no $received_A(T)$ value is maintained for the *home* site. Any tasks of $isolatedTrain(T)$ sent from a site A to the *home* site are guaranteed to arrive before the update message from A containing the corresponding sent count value $count_A(T,H)$ and therefore when the termination condition is detected for $isolatedTrain(T)$, any task sent from A to the *home* site has completed. The *home* site may ignore the value $count_Y(T,H)$ in an update message from a site Y since such tasks are known to have completed at H when $TC_H(T) = 0$.

The task count structure for a train T is used to balance the *sent* and *received/completed* task counts received in *update* messages for $isolatedTrain(T)$ at the *home* site. On receipt of an *update* message for $isolatedTrain(T)$ from a site A ; for each site B $count_A(T,B)$ (from the update message) is added to the *home* site's $count_H(T,B)$ value and RC_T is subtracted from the home site's $count_H(T, A)$ value.

Termination occurs at the *home* site when for all sites C , $count_H(T,A) = 0$ and $TC_H(T) = 0$.

7.2.2 Isolated Object Detection

The primary role of isolated object detection is to determine when the substitution protocol has completed for a promoted object. The substitution protocol is designed to allow for any inter-site and intra-site addressing mechanisms and any implementation of cars and trains within sites. For instance in an implementation where an object's car is encoded in its reference then on re-association each reference to that object in the system must be updated. However, in the DPBASE system no references need to be updated when an object x in train T is re-associated to a different car of T or is promoted to a train T' . Only the distributed object-to-train map must be updated on the re-association of an object and even then only when the object is promoted.

Objects may also become isolated due to mutator activity and the detection of these objects plays a key role in the RAL maintenance mechanism described in Section 8.1.

The distributed object-to-train map only contains entries for objects that are referenced from remote sites. Thus, when a site first exports a reference to a local object x in train T , a corresponding TB job $isolatedObject(\langle x,T \rangle)$ is created. In the terms used in the description of the DTA mapping this is creating an object x in train T . The DTA mapping for isolated object detection defines a task of $isolatedObject(\langle x,T \rangle)$ as a reference to $\langle x,T \rangle$. However the inter- and intra- site addressing mechanisms in the DPBASE system are based on references to objects and not references to objects in trains. Therefore a reference to $\langle x,T \rangle$ at a site A in

the DPBASE system means a reference to the object x at site A where A 's object-to-train mapping maps x to train T .

If `isolatedObject(< x , T >)` terminates then there are no references to $\langle x, T \rangle$, which means one of two things:

- If the home site substituted $\langle x, T' \rangle$ for $\langle x, T \rangle$ then the substitution protocol has completed and the distributed object-to-train map had been updated with the new train for x .
- If no substitution was in progress then there are no references to the object x at any site. Effectively x has become isolated purely through mutator activity.

The reason for this distinction is that the actions taken by the *home* site are different in each case.

A remote resident object x is completely local to the site that holding x and no remote site can hold a reference to x . Thus there is no need for a distributed object-to-train mapping entry for x . The promotion of x has no effect on the distributed shared state and thus no isolated object job is associated with x .

7.2.2.1 Idleness Detection for Isolated Object Jobs

Since a task of `isolatedObject(< x , T >)` corresponds to a reference to $\langle x, T \rangle$ a site A is idle for `isolatedObject(< x , T >)` when it holds no references to $\langle x, T \rangle$. However, as described in Section 7.2.1.3 above, a site already maintains a count of the number of references to an object from each train. No further reference counts need to be maintained to detect the idle state for an isolated object job at a site. A site A is idle for `isolatedObject(< x , T >)` if the *train reference count* structure for $\langle x, T \rangle$ at A is empty. This is detected by the write barrier on removal of the final $TRC_x[T]$ value.

7.2.2.2 Task Balancing Task Counts for Isolated Object Detection

Object promotion causes objects to change trains and since the substitution protocol does not complete instantaneously the distributed object-to-train map becomes inconsistent. At some sites the object-to-train mapping indicates that an object x is in a train T while at other sites the mapping indicates that the object is in train T' . In effect the sites which believe x to be in train T are operating over tasks of `isolatedObject(<x,T>)` while those sites that believe the object to be in train T' are operating over `isolatedObject(<x,T'>)`. In fact the problem is slightly worse than this since there may be multiple substitutions of x ongoing at any time, for instance if x is promoted again before the previous substitution has completed. Although there is only ever one object x , all references to x have the same value and x is only in one train at any time.

For each object x at a site A (where A is the home site for x), A maintains one TB task count structure plus one per substitution of x that is still executing. These TB task count structures are held in the $DA_{sym} \rightarrow CA$ table entry for x at site A as illustrated in Figure 7.5 below.

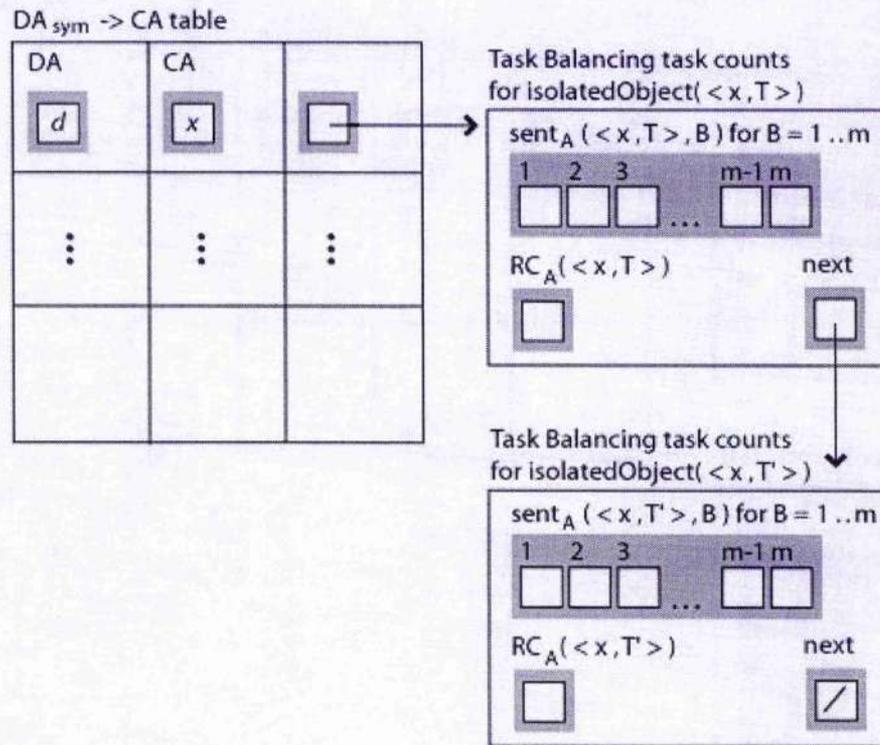


Figure 7.5 - The DA_{sym} → CA Table with TB task Count Structures

For each remote object y referenced by a site A that site maintains one TB task count structure for each isolated object job for y of which A has received a task and not yet sent an update message.

When the DA for an object x is sent between two sites A and B , the site A includes the train number of x , as determined by A 's object-to-train map, in the message. On receipt of such a message, site B determines the appropriate isolated object job for the reference to x from the train of x identified in the message.

The TB task count structure for $\text{isolatedObject}(\langle x, T \rangle)$ at site A (where A holds a reference to $\langle x, T \rangle$ and A is not the *home* site for $\text{isolatedObject}(\langle x, T \rangle)$) holds:

- the train identifier T for the isolated object job.
- an array of $\text{count}_A(\langle x, T \rangle, B)$ values with an element for each site B to which A has sent a task of $\text{isolatedObject}(\langle x, T \rangle)$.

- the *received* count for $\text{isolatedObject}(\langle x, T \rangle)$ at site A , $\text{received}_A(\langle x, T \rangle)$.

At a site A , the isolated object task count structure for a remote object $\langle x, T \rangle$ (with DA d) is stored in the $DA \rightarrow \text{Count}$ table entry for d at x as shown in Figure 7.4.

A task count structure for $\text{isolatedObject}(\langle x, T \rangle)$ is added on receipt of a reference $\langle x, T \rangle$ at A if no reference to $\langle x, T \rangle$ is currently held at A (i.e. if no task count structure exists). Since site A references the object x by its DA, the identifier d is used in-place of x in the isolated object structures at A .

The value $\text{received}_A(\langle d, T \rangle)$ is initially one, and is incremented on receipt of each subsequent reference to T . If at anytime the write barrier detects the idle state an *update* is sent containing the *received/completed* value for $\text{isolatedTrain}(\langle d, T \rangle)$ at site A , $RC_{\langle d, T \rangle} = \text{received}_A(T)$, and each non zero $\text{count}_A(T, B)$ value. The task count structure for $\text{isolatedTrain}(\langle d, T \rangle)$ is then discarded at A .

The TB task count structure at the *home* site H of an object $\langle x, T \rangle$ holds:

- an array of *sent* count values $\text{count}_H(\langle x, T \rangle, B)$ with one element for each site B for which an update from any site A has been received at H containing the value $\text{count}_A(T, B)$.

Again no *received* count is required at the home site as described in Section 7.2.1.4 above.

The task count structure for an object $\langle x, T \rangle$ is used to balance the *sent* and *received/completed* task counts received in *update* messages for $\text{isolatedObject}(\langle x, T \rangle)$ at the *home* site. On receipt of an *update* message for $\text{isolatedObject}(\langle d, T \rangle)$ from a site A ; for each site B $\text{count}_A(\langle d, T \rangle, B)$ (from the update message) is added to the *home* site's $\text{count}_H(\langle x, T \rangle, B)$ value and $RC_{\langle d, T \rangle}$ is

subtracted from the home site's $count_H(\langle x, T \rangle, A)$ value. Termination occurs at the *home* site when for all sites C , $count_H(\langle x, T \rangle, C) = 0$.

7.2.2.3 The Substitution Protocol

Each site maintains a substitution table as described in Chapter 6. The *home* site H of an object x promotes x from its current train T to a younger train T' by removing x from its current car and associating it with a car of T' at H . At this point $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ is added to the substitution table at H . A *substitution* message containing $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ is then sent to each site that holds a reference to x . Note that substitution is not required for remote resident objects since there is no entry in the distributed object-to-train mapping for these objects. The promotion of such an object has only a local impact.

Since substitution table entries represent references to objects it is necessary to extend the *train reference count* mechanism to account for these references. A substitution table reference count entry is defined for the *train reference count* structure. Such an entry is of the form $\langle substitution\ table\ id, count \rangle$ where *substitution table id* is an identifier that is distinguishable from all train identifiers and *count* records the number of substitution table entries for x (recall that there may be more than one).

A site A takes the following actions on receipt of a *substitution* message containing $\langle x, T \rangle \rightarrow \langle x, T' \rangle$:

- Add $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ to the local *substitution table*.
- Increment $received_A(T')$ and $TC_A(T')$, if necessary adding a new *remote train table* entry and allocating a new isolated train task count structure for T' .

Receipt of the *substitution* message constitutes the receipt of a reference to

train T' and the *substitution table* entry constitutes a reference to train T' held at A .

- If the *train reference count* data structure for x does not already contain a substitution table entry, add such an entry and set the count to one, otherwise increment the count by one.
- Update the $DA \rightarrow CA$ table or $DA \rightarrow \text{Count}$ table entry for x with the new train T' . That is, update the local object-to-train mapping for object x at A .
- Allocate an isolated object task count structure for $\langle x, T' \rangle$ to the $DA \rightarrow CA$ table or $DA \rightarrow \text{Count}$ table entry for x .
- If A is not idle for $\text{isolatedObject}(\langle x, T \rangle)$:
 - Set $TC_A(T') = TC_A(T) + \sum TRC_x[z], Z \neq T'$
 - Set $TC_A(T) = TC_A(T) + \sum TRC_x[z], Z \neq T$
 - Send a TB update message for $\text{isolatedObject}(\langle x, T \rangle)$ and discard the isolated object task count structure for $\text{isolatedObject}(\langle x, T \rangle)$.

If A later receives a task of $\text{isolatedObject}(\langle x, T \rangle)$, for instance from a site that does not yet know of the substitution, another TB update is sent from A to the *home* site for x . Such an update message contains only the value $RC_{\langle x, T \rangle} = 1$.

The refined send and copy reference events from Table 6.7 specify that a site should check if an object $\langle x, T \rangle$ is in the substitution table. This is unnecessary in the DMOS implementation since once the object-to-train mapping for a substituted object has been updated for the substitution $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ at a site, that site no longer has any references to $\langle x, T \rangle$. Therefore the site will never find itself in a position of copying or sending a reference to $\langle x, T \rangle$ once the substitution message has been received. The reason for the difference between the implementation and the

definition of the site actions for the reference copy events is that the substitution protocol described in Chapter 6 abstracts over the underlying addressing mechanisms. As such the substitution protocol allows for systems where a site may hold references to both $\langle x, T \rangle$ and $\langle x, T' \rangle$.

The TB algorithm itself presents a solution to the problem of determining the set of sites that must be informed of an object substitution. Initially a substitution message is sent to each site with a non-zero task count (at H) for `isolatedObject($\langle x, T \rangle$)`. As update messages arrive at H , due to sites becoming idle for `isolatedObject($\langle x, T \rangle$)` through substitution, the task count for each site that has been sent a task of `isolatedObject($\langle x, T \rangle$)` will become non-zero. Recall that an update message from a site A contains a count of the number of tasks sent from A to each other site. In this way H learns of each other site that potentially holds a task and sends a substitution message to each new site discovered.

H records each site to which a substitution message has been sent. On termination of `isolatedObject($\langle x, T \rangle$)` if $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ is in the substitution table, H sends a *substitution complete* message containing $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ to each site that was sent a *substitution* message. The substitution table entry for $\langle x, T \rangle$ is then removed at the *home* site.

On receipt of the *substitution complete* message a site removes $\langle x, T \rangle \rightarrow \langle x, T' \rangle$ from its *substitution table*.

7.2.3 An Optimisation for Task Counting

A commonly used optimisation, due to [DB76], in reference counting schemes (such as the local task counting mechanism described in this chapter) is to ignore references loaded onto the stack. When a local task count reaches zero, instead of

declaring the associated TB job idle, the job is added to a *zero count table* (ZCT). On car collection the root set is examined and any job in the ZCT for which a task is not found due to a reference on the stack is deemed idle. Jobs are removed from the ZCT at collection time if they are found to be idle or when their local reference count goes from zero to one. The write barrier therefore only modifies the local task count for a TB job on the creation of references between objects and not when a reference is pushed/popped from the stack.

This optimisation over normal reference counting is known as *deferred reference counting*, was initially proposed for use in the Smalltalk-80 virtual machine [Bad83] and used in the BrouHaHa Smalltalk interpreter [Mir87].

7.3 Summary

The contribution of this chapter is to present:

- The implementation of the train and car partitioning in the DPBASE system.
- The application of the Task Balancing (TB) DTA in the implementation of the two DTA mappings for DMOS.
- The design and implementation of the object substitution protocol.

This chapter bridges the gap from the abstract DTA mappings for isolated object and isolated train detection developed in Chapter 6 to a concrete implementation of these mappings for the DPBASE system. In implementing the DTA mappings a number of mechanisms are described:

- A distributed object-to-train mapping.
- Local idleness detection for isolated train jobs.
- Isolated train identification.
- The site actions for object substitution.

- The site actions that update the object-to-train mapping.
- Local idleness detection for isolated object jobs.
- Isolated object identification.

Local task and reference counting mechanisms are used within each site to detect idle TB jobs. However the local task counting mechanism represents only one particular scheme for detecting the idle state for isolated train and isolated object jobs. For instance a site could test for idleness for a given job by scanning its entire local storage space. If no reference to a particular object or into a particular train is found then the TB job associated with that object or train is idle at this site.

The DMOS collector is a partitioned garbage collector where objects are partitioned across sites by trains and within sites by cars of those trains. The implementation reduces to two reference counting mechanisms. The first of these counts references to individual objects and is used to detect when objects become isolated. The second reference counting mechanism counts references into trains and detects when trains become isolated. When a train becomes isolated all objects in that partition may be reclaimed. The effect of the two DTA mappings is to distribute the reference counting.

8 Implementing DMOS in DPBASE

This chapter presents a full implementation of the DMOS collector based on the isolated object and isolated train detection implementations from Chapter 7. These are combined with implementations of an RAL maintenance scheme and the re-association rules to provide a complete, incremental garbage collector for the DPBASE system. The description of the DMOS collector concludes with safety and completeness arguments for the implementation.

8.1 RAL Maintenance

In Chapter 6 the basic rules underlying RAL maintenance are described. This section describes an implementation of an RAL maintenance scheme that makes use of the site information in the TB isolated object task counts and extends the operation of the write barrier. The RAL rules, as stated, allow entries to be removed only on object isolation. This is clearly safe in terms of the RAL being a root set for car collection and, as has already been shown, the completeness of the DMOS collector is not affected, although collection of objects may be delayed. The aim of the RAL rules is to allow for the safe collection of a car with a minimum message passing overhead at the expense of a more out-of-date remset.

The RAL implementation provides a remset that allows for the collection of an intra-car cycle on the collection of the car holding the cycle rather than waiting for the whole train to be collected. This is done without the addition of further messages by using reference and site information provided by the TB implementation for isolated object detection. The number of RAL entries maintained for each object remains a matter of policy.

The RAL entries for a given car are divided into two sets. RAL entries in the first set are called *local RAL entries* and they represent intra-site references into the car. Entries in the second set are called *remote RAL entries* and they represent inter-site references into the car. The extended RAL maintenance scheme is structured as follows:

- An extension to the write barrier is used to detect when there are no local references to an object from outside its car thus allowing any *local RAL entries* for the object to be removed.
- The TB task counts for remote sites are used to determine when no remote site holds a reference to a particular object in a car thus allowing *remote RAL entries* to be removed.

Inter-car references between objects are detected by the write barrier on reference creation and at car collection time when re-associated objects are scanned for references. The write-barrier ensures that at least one RAL entry exists for each externally referenced object in a car. Further RAL entries are added at car collection time when re-associated objects are scanned for references. Since each car is eventually collected this mechanism is guaranteed to add an RAL entry for the youngest referent train for each object.

8.1.1 RAL Updates and Root Reference RAL Entries

The RAL maintenance scheme from Chapter 6 defines an *RAL update* message that can be sent from a site *A* to a site *B* to indicate a reference to an object at *B* held at *A*. An RAL update message from a site *A* for an object *x* at a site *B* can hold one of three payloads:

1. A root reference RAL entry indicating that a reference to x is held in a local root at A . Such an RAL entry contains a reference to the object x and the site identifier for site A . Each *root reference* RAL entry received from a site A for an object x is added to the set of *remote RAL entries* for x .
2. An indication that the object x is no longer referenced by a local root at A . On receiving an RAL update message at site B from site A indicating that x is no longer referenced by a root at A , each *root reference* RAL entry for site A is removed from the set of *remote RAL entries* for x . A *root reference* RAL entry for site B is then added to the set of *local RAL entries* for x .
3. An inter-car RAL entry indicating a reference in a car at A . Such an RAL entry contains a reference to the object x and the train number for the train T holding the reference to x at A . The actions taken on receipt of an inter-car RAL entry are explained in Section 8.1.2 below.

Each *root reference* RAL entry sent by a site A is added to a log at A . This log implements the *remote root referenced objects* set for the site A as described in Chapter 6. A *root reference* RAL entry for an object x at site B is sent from a site A only if the log does not already contain a corresponding entry. Therefore only one *root reference* RAL entry is sent for a remote object whose reference is found in the local root set at A when the remote reference is found on consecutive car collections. For each *root reference* RAL entry in the log where the specified object y is not referenced by the local root set at A , an RAL update message is sent to the site holding y indicating that y is no longer referenced by a root at A .

8.1.2 Adding RAL entries

To guarantee that the RAL eventually contains an entry for the youngest referent train for an object x , irrespective of any policy governing the number of RAL entries maintained, an RAL entry (either local or remote) for a reference to x from a train U can only be replaced by an RAL entry for a reference from train V if $U < V$ (i.e. V is younger than U).

RAL entries are added to a car as follows:

- When the write barrier detects the creation of a local (intra-site) inter-car reference to an object x from a train U the local RAL entries in the car holding x are scanned and if no entry for x is found, $[x,U]$ is added, if an entry for x already exists then it is a matter of local policy whether an entry is added or not.
- When a local inter-car reference from a train U to an object x is discovered on object re-association a local RAL entry is added to the car holding x if U is younger than any other referencing train in the set of local RAL entries for x . If a local RAL entry with a train number younger than U already exists then it is a matter of local policy whether an entry is added or not.
- If a reference to x (which is in train T) is sent to a remote site then the RAL is scanned and if no remote RAL entry is found, $[x,T]$ is added to the set of remote entries for x .
- Local RAL entries are added at car collection time (for local references) and remote RAL entries are added on receipt of RAL update messages (for remote references).

8.1.3 Removing RAL Entries

RAL entries for an object x are removed when it is no longer referenced from outside its car. Each inter-site reference to x is an external reference. The task balancing DTA allows a site to construct a safe approximation to the set of sites referencing any particular object and hence determine when an object is no longer referenced from a remote site. If the task count structure for $\text{isolatedObject}(\langle x, T \rangle)$, at its home site H , has a zero count for all sites (not including H) then $\langle x, T \rangle$ is not referenced from any remote site and any remote RAL entries are removed.

Local references may be inter- or intra-car. For each object x , that H is the home site for, an *inter-car reference count* value is maintained, written $ICRC_x$, which records the number of references at H to the object x from outside the object's car. The car data structure (Figure 7.3) is extended to record this value for local and remote resident objects respectively. The write barrier is further extended as follows. $ICRC_x$ is initialised to 1 on the creation of the first local inter-car reference to x . When the write barrier traps the creation of an inter-car reference to x , $ICRC_x$ is incremented and when an inter-car reference to x is over written, $ICRC_x$ is decremented. If $ICRC_x = 0$ then H holds no local inter-car reference to x and all local RAL entries are removed.

Two rules are implemented to maintain correct *inter-car reference count* values on object re-association. Say an object x is re-associated from a car C_1 to a car C_2 :

- for each reference to an object x in C_2 , decrement $ICRC_x$;
- for each reference in x (which is in train T) to an object $\langle y, U \rangle$, where $\langle y, U \rangle$ is not in C_2 (note that this is a stronger predicate than $T \neq U$), increment $ICRC_x$.

The deferred reference counting optimisation extends to the inter-car reference counts for objects. When the *ICRC* value for an object reaches zero, the object is added to the ZCT and on car collection the object's RAL entries are only removed if a reference to the object is not found in a local root. As before, ZCT entries for inter-car reference counts are removed if an object is not found in a local root or if its *inter-car reference count* value increases from zero.

8.2 Requesting Train Tasks

The DTA mapping for isolated train detection prevents a site from re-associating an object to a train T if that site does not already hold a task of `isolatedTrain(T)`. However a site may request a task of a particular train by sending a *task request* message (containing the train identifier for the required train) to the train's *home* site. On receipt of a *task request* message for a train T the *home* site sends a *task reply* message to A . If `isolatedTrain(T)` has not already terminated the reply contains a task of `isolatedTrain(T)`, otherwise the reply contains an indication that `isolatedTrain(T)` has terminated. Note that even if the home site does not hold a task of `isolatedTrain(T)` it is safe for it to spontaneously create a task to send to A since the *home* site is responsible for the detection of termination of `isolatedTrain(T)`. If the home site does not hold a task of `isolatedTrain(T)` then this does not necessarily mean that the job has terminated, however, in this position (of holding no tasks) the home site alone may safely create a new task. On sending the reply message the *home* site increments $count_H(T, A)$ in the task count structure for `isolatedTrain(T)`. Only if T is not the oldest train created at H may a task of `isolatedTrain(T)` be sent by the *home* site.

On receipt of the *task reply* message at a site A for a train T the value $received_A(T)$ is incremented by one if the message contains a task. This may require that a new task

count structure for $\text{isolatedTrain}(T)$ is initialised since the task was only requested because the site A was idle for $\text{isolatedTrain}(T)$. A task requested in this way is known as a *requested isolated train task*. As explained in Chapter 6, these tasks must eventually complete so as not to prevent the train T from becoming isolated. This is achieved as follows.

When a site receives a *requested isolated train task* that was requested for an object x an entry is added to a *requested task* table at that site. At this point the value $TC_A(T)$ is incremented by one. The notation $\text{ritt}_A(T,x)$ denotes a *requested isolated train task* of $\text{isolatedTrain}(T)$ held at a site A for the re-association of an object x . An entry in this table indicates the object that the task was requested for and the isolated train job that the task is for.

Following the re-association of an object x each entry for x in the *requested task* table at A is removed. The removal of an entry for object x and train T corresponds to the completion of $\text{ritt}_A(T,x)$ and therefore the value $TC_A(T)$ is decremented by one. If $TC_A(T) = 0$ an update message is sent for $\text{isolatedTrain}(T)$.

If the site A receives a *task reply* message for the object x and train T that indicates that $\text{isolatedTrain}(T)$ has terminated each RAL entry for the object x , indicating a reference from train T is replaced with an RAL entry indicating a reference from x 's current train. Effectively A 's view of the trains that hold references to the object x is out-of-date.

8.3 Collecting Isolated Trains

On detection of termination for $\text{isolatedTrain}(T)$, the *home* site for the train T must inform each site holding a car of the isolated train T so that the cars may be reclaimed at those sites. When a site A becomes idle for $\text{isolatedTrain}(T)$ (where A is

not the home site for $\text{isolatedTrain}(T)$), an update message is constructed for sending to the home site. At this point the site A is in the position of knowing whether or not it holds any cars of the train T , and hence whether or not the home site will have to inform A of the termination of $\text{isolatedTrain}(T)$. This information is passed to the home site in the update message for $\text{isolatedTrain}(T)$. An additional boolean field, $\text{haveCars}_A(T)$, is defined for TB update messages. This field is set to true if the sending site has any cars of the train and false if it does not. If a site A is idle for $\text{isolatedTrain}(T)$ (where A is not the *home* site of T) and holds no cars of T then the *remote train table* entry for T at A is removed after the update for $\text{isolatedTrain}(T)$ has been sent.

The home site for $\text{isolatedTrain}(T)$ maintains a list of sites, $\text{carsList}(T)$, from which it has received an update message (for $\text{isolatedTrain}(T)$) with a true $\text{haveCars}_A(T)$ value. This list is held in the train's *local trains table* entry. On receipt of an update from a site A with a false $\text{haveCars}_A(T)$ value, where $\text{carsList}(T)$ already holds A , site A is removed from the list.

When the *home* site detects termination of $\text{isolatedTrain}(T)$, each site in $\text{carsList}(T)$ is sent an *isolated train* message informing it that the cars of train T that it holds may now be collected. On receipt of an *isolated train message* for a train T , a site reclaims each car of T that it holds and removes its *remote train table* entry for T .

The mechanism used to identify the set of sites that have to be informed on the detection of an isolated train is different from the actions taken by a site to discover the sites to which a *substitution* message must be sent. In both cases the implementation avoids additional message overhead by using the TB control messages that are sent anyway. However, in the case of substitution, the home site can start only with the current set of sites that have a non-zero task count and the TB

algorithm allows H to ‘learn’ of each other site as update messages arrive. In the case of isolated train detection the set of sites must be constructed before termination occurs in order to avoid additional message overhead.

8.4 Car Collection

A DMOS collection cycle involves the collection of one or more cars at a site. The root set for the collection of a car C at site S consists of the combination of C 's RAL and the set of local root references at S . The collection of a car C at a site S is considered in three stages:

- First the local root set is examined.
- Secondly objects in C referenced from C 's RAL and those objects reachable from RAL referenced objects are re-associated to different cars. The re-association rules (from Chapter 6) are as follows:
 - An object directly reachable from the mutator is re-associated to a car of any *younger* train (possibly creating a new train).
 - An object reachable from one or more *younger* trains can be re-associated to a car of (any one of) those trains.
 - An object reachable only from another car of the *current* train, or from one or more *older* trains, should be re-associated to some other car (possibly a new one) of the *current* train.
- Thirdly the car is reclaimed.

8.4.1 Examining the Local Root Set

The first stage of car collection involves the examination of the local root set. In the DPBASE system the local root set at a site S consists of the stack for each thread executing at S . The actions for the collection of a car C are as follows:

- For each reference in the local root set at S to a remote object x , where the RAL update log does not already contain an entry for x , a Root Reference RAL entry is sent to the home site of x . The Root Reference RAL update is then added to the RAL update log at S .
- For each object x for which the RAL update log contains an entry, but where the local root set does not contain a reference to x , an RAL update indicating that x is no longer referenced by a root at S is sent to the home site of x . The RAL update log entry for x is then removed.
- For each reference in the local root set at S to local object x a *Root Reference* RAL entry is added to the car of x , if no such entry already exists. Now the RAL for car C contains a *Root Reference* RAL entry for each object in C which is referenced by the local root set. That is, each object in C which is referenced by the Mutator now has a *Root Reference* RAL entry.

8.4.2 Re-Associating Objects

The second stage of car collection involves the re-association of those objects in the car that are reachable from an entry in the RAL.

For any object x re-associated from a car C of train T to a car D of train U (where $U \geq T$) each RAL entry for x is transferred to x 's new car D . For each local reference in x a *local RAL entry* for a reference from train U is added to the car of the referenced object. For each remote reference in x an RAL update message,

containing a *remote RAL entry* for a reference from train U at site S , is sent to the home site of the referenced object. When x is re-associated, each *requested task* table entry for x at S is removed. If $U \neq T$ and x is a local object that has a DA, $\langle x, U \rangle$ is substituted for $\langle x, T \rangle$ as described in Section 7.2.2.3 above. In summary, $\langle x, T \rangle \rightarrow \langle x, U \rangle$ is added to the substitution table at S and a *substitution* message is sent to each site with a non-zero task count for `isolatedObject($\langle x, T \rangle$)`. S records each site to which a *substitution* message is sent.

The re-association rules are implemented as follows (and in the order listed here) for a car C in train T at a site S :

1. Each object x in C where x is referenced by a *Root Reference RAL entry* is re-associated to a car of a younger train. This can be an already existing train or it may be a new train. Where the object is re-associated to an existing train it may be associated with an existing car of that train or with a new car. In the case where a site holds no tasks of any existing train, a new train is created. If x is not currently referenced by the local root set at S each *local RAL entry* for x in car D is removed and a *local RAL entry* is added to D 's RAL indicating a reference to x from its new train U .
2. Each object x in C where x is referenced by an RAL entry indicating a reference from a younger train U , is re-associated to a (possibly new) car of train U . If site S holds no tasks of train U then x is instead re-associated to a different (and possibly new) car of T and a task request message is sent to the home site of U .
3. Each object x in C where x is referenced by an RAL entry indicating a reference from T , is re-associated to a different (and possibly new) car of T .

While objects are re-associated in accordance with the re-association rules, there is still a large scope for policy choice in this process. For instance the RAL may contain multiple entries for an object thus presenting a choice of destination train. Policy choice appears at a number of points in the DMOS implementation however policy choices cannot be considered in isolation since in a complex system such as the DMOS collector these policies are unlikely to be independent. The policy space is discussed in more detail in Chapter 9.

8.4.3 Reclaiming a Car

Cars are reclaimed at two points:

- During car collection; following the re-association of all reachable objects.
- When an isolated train is detected.

In both cases each object remaining in a reclaimed car is garbage. The following actions are taken at a site S for each object x in a car C on the reclamation of C :

- The reference to object x is removed from C 's reference array and the pointer to the car structure for C is removed from the object x .
- Each *requested task* table entry for x is removed. This corresponds to the completion of a task of the isolated train job for the specified train.
- Each reference in x is deleted, i.e. the actions on isolated object and isolated train tasks corresponding to reference deletion are carried out for each reference in x .
- The address translation table entry for x at S is removed. If x is a local object the $DA_{sym} \rightarrow CA$ table entry for x is removed, otherwise x is a remote resident object and as such the $DA \rightarrow CA$ table entry for x is removed.

The car structure may now be reused however no local storage space is reclaimed in doing so since a car represents a mapping from objects to cars rather than a contiguous area of local storage. Ultimately the particular mechanism by which local cache space is reclaimed is a matter of policy. The simplest mechanism is for a site to maintain a free-list to which each object in a reclaimed is added.

8.5 Safety and Completeness of the DMOS Implementation

As with many algorithms for continuously running systems, correctness of incremental garbage collection algorithms breaks down into two distinct parts. One is safety, in this case that the collector never deletes a reachable object. The other is completeness (or progress), in this case that the collector eventually reclaims every garbage object. Chapter 6 presents arguments for the correctness of the centralised UMOS collector and the arguments for the safety and completeness of the DMOS implementation are fundamentally the same.

However, distribution adds a further dimension to the correctness arguments. In general terms, a centralised collector can exactly compute the transitive closure of the object graph (and hence the set of reachable objects) while a distributed collector can only ever compute a conservative approximation. More specifically, a centralised UMOS collector can compute the remset (and thus the set of reachable objects) for each car exactly. While in the distributed context, the remset for a car (the RAL) can be out of date. In arguing for the correctness of the DMOS implementation it is therefore necessary show that the approximation to the remset represented by the RAL is conservative enough to ensure safety and that it eventually converges to a globally consistent view (for that particular car) that provides completeness.

8.5.1 Safety

Objects are discarded on the collection of a car and on the collection of a train. To show that the DMOS implementation is safe with regards to car collection it is sufficient to show that no object, reachable from outside the car, is discarded when the car is collected. Since the RAL constitutes the root set for car collection we must show that the RAL for a particular car contains entries for at least those objects referenced from outside the car. References held in local roots can be ignored since these references are added (temporarily at least) on car collection. The RAL maintenance mechanism distinguishes between remote and local references since two different mechanisms are used (opportunistically) to detect when entries may be removed. To demonstrate that safety is guaranteed we show that an object's RAL contains a remote entry while the object is referenced from a remote site and a local entry while the object is referenced from another car locally.

- On creation of the first inter-car intra-site reference to an object x , in a car C at a site H , a local RAL entry is added. *Local RAL entries* for x are only removed when the local inter-car reference count, $ICRC_x$ is zero. The write barrier at H is capable of computing exactly the number of references to $\langle x, T \rangle$ from outside of the car C . Thus C 's RAL is guaranteed to contain an entry for $\langle x, T \rangle$ while there exists a reference to x at H from outside C .
- When a reference to x is first exported to a remote site, a *remote RAL entry* is added. *Remote RAL entries* are only removed when the task balancing DTA for isolated object detection indicates that no remote site holds a reference to x . Since the isolated object detection mechanism effectively tracks all references to an object across the entire distributed system, C 's RAL is

guaranteed to hold a *remote RAL entry* for x while x is referenced from a remote site.

Safety of the DMOS collector in the face of train reclamation is demonstrated by showing that no train is reclaimed while it contains a live object. A train is only reclaimed when the task balancing DTA for train isolation detects the absence of any reference into the train. Since the isolated train detection mechanism tracks all references into a train then it is guaranteed to not be reclaimed while it contains a live object. However, while an object is being substituted the system is in an inconsistent state with sites believing they hold references to one train when (due to the substitution) they hold references to a different train. The object substitution protocol prevents a train from becoming isolated while it contains an object that is in the process of being substituted, by adding a reference to the substitution table at the object's home site and only removing the reference when the substitution is complete and the system has returned to a consistent state (as regards the substituted object).

8.5.2 Completeness

Before explaining the completeness arguments for the DMOS implementation the completeness arguments for the centralised UMOS collector are repeated. The approach taken is to then built on the completeness arguments of the centralised collector in arguing the correctness of DMOS.

Recall that each car in the UMOS collector has a complete remset which records every reference into the car. In order to show the completeness of UMOS it is necessary to show that each object is eventually collected. The argument (from Chapter 6) is as follows:

- Any garbage object which is part of a non-cyclic data structure will be reclaimed through car collection alone, since any object in a car C which is not referenced from outside C is reclaimed when C is collected.
- This argument also holds for garbage objects which are part of a cyclic structure which is completely contained within a single car.
- The re-association rules and the immutable nature of garbage dictate that a garbage object will be re-associated as far as its youngest referent train and no further. This means that an inter-car garbage cycle that is completely contained within a particular set of trains will collapse, after some finite number of car collections, into the youngest train in the set.
- The completeness of UMOS can now be shown by proving that every train will eventually become isolated and thus reclaimed. This can be shown as follows:
 - The oldest train is guaranteed to become isolated since the mutator is not allowed to allocate into it, no object can be re-associated into it and the re-association rules will eventually re-associate any reachable object in it to a younger train.
 - Every train will eventually become the oldest train.

The UMOS collector can atomically reclaim a train once it becomes isolated, however this is not true of the DMOS collector since the cars of a train may be held

on multiple sites. The DTA mapping for isolated train detection in DMOS means that train isolation in the distributed context is a globally stable state. Thus once train isolation is detected, the system may reclaim the cars of that train at its leisure. Completeness of the DMOS collector relies on the home site of an isolated train T being able to calculate the set of sites holding cars of T . This ability is derived from the particular DTA used in the implementation, which is the TB DTA. Each site which has at some point had a non-zero task count for `isolatedTrain(T)` may hold a car of T and therefore must be informed when T becomes isolated.

The key difference between the DMOS collector and the UMOS collector (in terms of this completeness argument) is the form of the remset which is maintained for each car. While remsets in UMOS are complete and up to date at all times, remsets in DMOS (RALs) are not complete and can be out-of-date. The UMOS completeness argument can be applied to DMOS if RALs can be shown to become accurate eventually. However objects can be moved between cars during car collection and therefore what must be shown is that for each object x , there eventually exists an accurate set of RAL entries.

This means that the RAL entries have the following properties:

1. If an object x is referenced by the mutator at a site S , x 's RAL entries eventually contain a *Root Reference* RAL entry for that site. This ensures that root referenced objects continue to be promoted to increasingly younger trains.
2. If there exists a *Root Reference* RAL entry from a site S for an object x , and x is not referenced by the mutator at S , then the *Root Reference* RAL entry from S must eventually be removed. This ensures that objects that have been referenced by a root at a site eventually stop being promoted.

3. Any Root Reference RAL entry for an object x in train T must be maintained long enough to ensure that x is promoted to a younger train at least once. This ensures that the *zero-progress problem* (described in Chapter 6) is avoided.
4. The set of RAL entries for an object x must eventually contain an entry for the youngest referent train for x . This ensures that each inter-train garbage cycle eventually collapses into a single train, which is the youngest train that holds an object in the cycle.
5. If there exists no reference to an object x in a car C from outside of C all RAL entries for x must be removed. This ensures that non-cyclic garbage structures and inter-car cycles of garbage are collected through the car collection. Note that this is not strictly necessary since the train reclamation mechanism is capable of collecting such structures.

First it is necessary to assume that each car is collected after some finite amount of time from its creation. In fact this assumption is already made in UMOS with the statement, "every car is eventually collected". The point here is that this assumption does not change in the distributed context, even though the distributed system model places no bounds on the relative speed of computation between individual sites.

The implementation of the RAL maintenance rules provides each of the properties defined above. In collecting a car, the root set is examined, RAL entries for local references are updated and RAL updates are sent for all remote references (property 1). Since each car is eventually collected the reference to an object x from its youngest referent train is eventually discovered and an appropriate RAL entry for x added (property 4). A site logs the RAL updates which it has previously sent and thus can identify remote objects which are now referenced by the mutator and those which have previously been referenced by the mutator (property 2).

RAL entries for an object x are removed when the local site holds no references to x and the isolated object detection mechanism indicates that no remote site holds a reference to x (property 5). However the final Root Reference RAL entry for an object x is removed only following the promotion of x if x is not referenced by the local mutator (property 3).

Thus RALs are eventually sufficiently accurate to allow completeness arguments for UMOS (with its complete resets) to be used in relation to DMOS.

8.6 Summary

This chapter presents a complete implementation of the DMOS collector based on the isolated object and isolated train detection implementations from Chapter 7. In addition to the DTA mapping implementations the following mechanisms complete the implementation of the DMOS collector:

- A Remset (RAL) maintenance scheme for cars that allows for the identification of intra-car garbage cycles at car collection time. This is implemented through an extension to the write barrier and use of distributed reference information provided by the isolated object detection implementation.
- A protocol that identifies the sites holding cars of isolated trains, based on an extension to TB update messages for isolated train detection.
- A car collection scheme that represents an implementation of the DMOS re-association rules.
- A local garbage collector that reclaims space in the local object cache at a site.

The DTA mapping implementations provide distributed reference counting over references to objects and references to trains. Reference counting is known to be incomplete; however there are two aspects of DMOS that make the collector complete. The first is a subtlety of the train algorithm whereby the partitioning of objects is changed through the application of the re-association rules to objects in cars. While garbage is effectively immutable the re-association rules continue to re-arrange the partitioning of objects and guarantee that the reference count for each train eventually stabilises with the value zero. The second is a subtlety of the DTA mapping for isolated object detection where the substitution protocol effectively forces substituted objects to become isolated.

The specific contribution described in this chapter can be summarised as follows:

- The design of an asynchronous RAL maintenance mechanism.
- The specification of the actions at a site to reclaim a car and to reclaim an isolated train.
- The design of a simple local garbage collector.
- Safety and completeness arguments for the DMOS collector implementation.

9 Conclusions

The motivation for the work in this thesis is the implementation of the DMOS distributed garbage collector (DGC). DMOS appears to exhibit a unique combination of attractive characteristics for a DGC but to-date lacks a satisfactory implementation. Before arriving at a suitable implementation it is first necessary to understand the interaction of the two collection mechanisms in DMOS and to examine the role of distributed termination detection within these mechanisms.

Since the original DMOS algorithm is known to contain a bug, the approach taken is to derive a new DMOS algorithm. This is done in two stages:

- First the DGC derivation methodology from [BHM+01] is examined. The methodology provides an outline structure for DGC design through the combination of a centralised collector with a DTA. In this case the aim was to investigate the suitability of the methodology for DGC design.
- Secondly the methodology is applied in deriving a new version of the DMOS collector.

The hypothesis being tested in this thesis is that the derivation methodology can be used to guide the development a suitably understandable implementation of DMOS.

In testing this hypothesis a number of side-tracks have been explored.

9.1 The Mapping Methodology

The derivation methodology described by Blackburn et al. is based on the construction of a *reclamation mapping*, whereby actions of the centralised collector are mapped to actions on jobs and tasks of a DTA. The process is not automatic.

Chapter 5 refines the methodology by concentrating on the state over which the centralised collector and the basic computation operate. The basic computation and the centralised collector are first distributed, thus distributing and consequently partitioning the shared state. The aim of the mapping is now to apply a DTA to identifying globally stable properties of the partitioned distributed state. The actions of the centralised collector and the basic computation are thus mapped to actions on jobs and tasks of a DTA. In general terms a globally stable property of some subset of the distributed partitioned state constitutes a DTA job. The nature of tasks of that job depends on the initial centralised collector and the nature of the state partitioning. Termination of the job corresponds to detection of the globally stable property.

Garbage collectors typically use reachability from a set of known roots in order to calculate an approximation to the set of garbage objects. Unreachability is a globally stable state (in a distributed system) by definition, and it makes sense to use a DTA mapping to detect this state. However this is not a requirement. Of the three mappings in Chapter 5 only the DTA mapping for the two Distributed Reference Counting collectors uses a DTA to directly identify unreachable objects.

Transforming the centralised collector into a concurrent collector (as described in [BHM+01]) is a useful technique in understanding how the shared state is to be distributed and in identifying stable properties of the partitioned state. This technique proved important in the development of the DTA mappings for the DMOS collector.

The mapping methodology ultimately results in the modularisation of the derived distributed collector, where the mapping allows a DTA to be 'plugged-in' in order to identify globally stable states. The extension to the methodology that is described in Chapter 5 is to increase this modularisation by separating:

- the work of the distributed collector that can be carried out with completely local information;
- the work at a site involved in the maintenance of the distributed shared state;
- the work involved in the implementation of the DTA to detect the global stable properties of the distributed shared state.

By modularising the distributed collector a boundary between the distributed and local work can be established. This boundary is defined by a set of *club rules* which describe the actions at site required to implement the distributed collector. By varying the boundary, sites can be given more or less freedom in how they behave.

Participant sites are provided with an interface to the distributed collector, defined by the club rules, which allows a degree of heterogeneity within the system. More specifically objects can be reclaimed independently at a site and at any chosen rate. Sites can carry out local collection in any manner they choose, so long as they obey the club rules.

9.1.1 DTA Mappings and Reference Counting Collectors

While the six collectors described in Chapter 5 serve mainly as proofs of concept (for the capabilities of the mapping methodology) they illustrate an important point concerning the nature of the DTA mappings. In both the Distributed Mark-Sweep (DM-S) and the Distributed Generational Collector (DGC) the club rules describe mechanisms for accounting for references in-flight between sites. This is not true of the Distributed Reference Counting (DRC) Collector.

In the DTA mapping for the DRC collector a job is an object's *distributed reference count* and a task for the job is an inter-site reference to that object. The effect of mapping object references to tasks is that the DTA accounts for any references in-

flight between sites. However in doing this the resultant collector can only ever be a reference counting collector, and such collectors are not complete.

9.2 Task Balancing

The TB algorithm is used in the implementation of each of the collectors described in this thesis. Chapter 4 describes the algorithm in its most generic form and implementations of the algorithm are presented in Chapters 5 and 7.

The mapping methodology incorporates a DTA as a component of the DGC and therefore it is important that choice of DTA does not preclude any of the properties required of the derived DGC. The Task Balancing DTA exhibits a number of properties which suggest its suitability for the implementation of DGCs. As explained in Chapter 4, the algorithm is safe, complete, non-disruptive, incremental, non-blocking and independent and thus satisfies each prerequisites of scalability.

9.3 The DMOS Collector

Use of the mapping methodology has resulted in a new implementation of the DMOS DGC. The development of the DTA mapping for DMOS in Chapter 6 illustrates the construction of the two DTA mappings and the process by which the collector is modularized. The distribution of the UMOS collector is not new since this is the distribution described in [HMM+97]. However the distribution of shared state and its partitioning across sites is new. The DMOS collector breaks down into four parts:

- **Isolated train detection:** This is based on a mapping of trains to jobs and references into trains to tasks. The detection of isolated trains relies on each site knowing the train holding every object which that site references. To this end a distributed object-to-train mapping is defined. On termination of the

job corresponding to a train, that train is isolated and its cars may be reclaimed.

- **Object substitution:** The *substitution protocol* allows objects to be re-associated from one car to another by logically substituting a *new* object in target car for the old object in the original car and updating every reference in the system (to the *old* object) with a reference to the *new* object. If an object is promoted to a younger train on re-association the *substitution protocol* is used to update the distributed object-to-train map for that object. The substitution protocol is guaranteed to complete for any given substitution since each site maintains meta-data relating to the substitution and there are a finite number of sites. The meta-data represents both a reference to the *new* object and (in the case of a promotion) a reference into its *new* train. This is required to ensure that neither the *new* object or the train in which it is held are reclaimed while the system is in an inconsistent state²³. However this requires that the system can detect the completion of the substitution of an object in order to safely discard the meta-data.
- **Isolated object detection:** A second DTA mapping is defined to detect the completion of the *substitution protocol* for an object which has been re-associated. This is based on a mapping of an object x in a train T (written $\langle x, T \rangle$) to a job and references to $\langle x, T \rangle$ to tasks. On termination of the job the corresponding object is no longer referenced anywhere in the distributed system.

²³ With some sites holding references to the *old* object and others to the *new* object.

- **RAL maintenance:** RALs constitute the DMOS equivalent of UMOS's car remsets. Where a UMOS remset provides a complete and up-to-date view of the set of references into a car the RAL provides an incomplete and out of date view of this set. An RAL only identifies the referencing trains for the object in a car. The DMOS implementation uses an asynchronous message passing mechanism for maintaining RAL entries for inter-site references. The subtlety of the RAL mechanism is that while an RAL may be out-of-date and incomplete it is eventually sufficiently accurate for objects to be re-associated correctly. The implementation leverages information from the TB data structures for isolated object detection to provide a more accurate RAL without incurring additional messaging overheads.

The substitution protocol is more generic than that required for the implementation of the DMOS collector in the DPBASE system. The substitution protocol is designed to allow for any underlying addressing mechanism and as such supports systems where each re-association requires the update of every reference (to the re-associated object) in the system. The DPBASE system implements inter-site addressing and the association of objects and cars through a set of local mapping tables and as such does not require the full generality. The only distributed state that must be updated is the distributed object-to-train map. The mechanism for updating this distributed mapping is effectively layered on top of the generic substitution protocol.

The implementation is based on the definition of a set of site actions which correspond to mutator and garbage collector events at a site. Following the development of the DTA mappings, the site actions on DTA jobs and tasks are defined using a process of stepwise refinement through the three layers of the collector.

9.3.1 DMOS: Reference Counting with Trains

Both of the DTA mappings for DMOS map tasks to individual references. The isolated train mapping maps tasks to references into trains while the isolated object mapping maps tasks to references to objects within particular trains. Thus the DMOS collector can be considered as consisting of two independent reference counting mechanisms. The isolated train detection mechanism counts references into trains while the isolated object detection mechanism counts references to individual objects. However this has no effect on the completeness property of the collector. There are two reasons for this:

1. Trains are guaranteed to become isolated.
 - o The re-association rules re-organise the association of objects with cars and trains, guaranteeing that any garbage cycle is eventually collapsed into a single train. The re-association rules can be thought of as a second mutator process which operates over references to trains (instead of objects). In such a scheme an inter-train cycle of objects is transformed onto a cycle of trains. Thus as the re-association rules collapse the garbage cycle into a single train, each train reference (due to an object reference in the cycle) is effectively deleted.
2. Substituted objects are guaranteed to become isolated.
 - o The substitution protocol effectively forces substituted objects to become isolated. Once a site S has been informed of the substitution of object x' for object x , S will create no more references to x and will eventually replace each of its references to x with a reference to x' .

Since there are a finite number of sites, x will eventually become isolated.

9.4 Contribution

The work described in this thesis was undertaken as part of the DMOS project [MKB99] funded under the EPSRC Distributed Information Systems Initiative. DMOS was originally presented (in [HMM+97]) as an algorithm which incorporated the implementation of two DTAs within two interacting collection mechanisms.

From the grant proposal:

“Our aim is to understand, implement, and measure a family of DMOS implementations that vary in the two distributed termination algorithms and the local collector. Such variations may be targeted towards intrinsic properties such as fault tolerance, persistence, efficiency and object migration. This will allow different implementations of DMOS to be tailored to a specific environment such as a high-performance multi-computer or a loosely coupled set of distributed sites.”

Three key deliverables of the original project were:

- a categorisation of current distributed termination algorithms suitable for detecting absence in terms of their intrinsic properties and suitability for mapping to different distributed architectures.
- a categorisation of current distributed termination algorithms suitable for detecting empty train in terms of their intrinsic properties and suitability for mapping to different distributed architectures.
- a study of the interaction of the two termination algorithms.

In summary, the goal was to implement the algorithm as it stood and to experiment with the policy space and the DTA implementations. The purpose of this was to understand the interaction of the two collection mechanisms, the interaction of the independent DTA implementations and the effect of policy choice on the behaviour of the collector.

One of the original goals has been achieved. This thesis describes the implementation of a DMOS collector for the distributed ProcessBase (DPBASE) system which operates over a loosely coupled multi-computer (a Beowulf). The DPBASE system is described in Chapter 3.

However in achieving this goal the research has adopted a different approach from that suggested in the original project proposal. In this thesis importance is placed on understanding the interaction of car and train collecting mechanisms in order to ensure that this interaction is safe. In order to be able to change either of the DTAs that is used in the collector it is first necessary to map out those areas of the collector which require distributed termination detection.

The approach taken was to step back from the algorithm as it stood and to look at the specific roles played by each collection mechanism and within each of these, the role of the DTA. The intuition here was that the garbage-collector-to-DTA mapping methodology from [BHM+01] would provide a structured approach to separating the components of the DMOS collector.

The work described in Chapter 5 aims to explore the utility of the mapping methodology. However, the result was the development of an extension to the methodology. While the development of the extended methodology was ultimately a side-track from the work on understanding DMOS the insight that was gained into

the role of DTAs in DGC design was of primary importance in the development of the DMOS collector.

The extension to the methodology [NMM+03] aims to minimise the constraints placed on a site of the DGC. This is achieved by mapping a DTA onto a (non-distributed) garbage collection scheme, to derive a global distributed collector while leaving a site free to implement any local collection scheme. Each mapping is used to define a set of club rules that must be obeyed by each participant (site) in the distributed collection scheme. The participating collectors are free to perform any local actions as long as they preserve the club rules. The benefit of such a structured approach to distributed collector implementation is the clear distinction between providing safety via termination (distributed work) and space reclamation (local work).

The extension centres on identifying stable properties within the shared state of the centralised collector and designing the distribution and partitioning of this state such that a DTA can be used to identify globally stable properties.

Having shown that the mapping methodology could be used to derive modularised distributed collectors the next step was to apply the methodology to derive an implementation of DMOS. However, rather than develop DMOS around any particular DTA, the operations (of the mutator and the collector) over the shared state are mapped to abstract actions over jobs and tasks of the DTA model. The aim was to develop a new, modularised, version of the DMOS collector and thus:

- identify the distributed information required by the two collection mechanisms.
- describe the distribution and partitioning of the shared state.

- describe the globally stable properties of the state that must be detected for distributed garbage collection.
- describe the events within the mutator and the collector which are used to drive the DTAs to detect the globally stable properties.
- minimise the constraints placed on each site particularly relating to local and distributed addressing mechanism, local storage architecture and local reclamation mechanisms.

That is, rather than start with the algorithm as it was already described and work backwards to discover the exact role of the DTAs, the methodology was used to derive a new version of DMOS from the centralised MOS collector [HM92]. The derivation of DMOS is described in Chapter 6. This breaks down into a number of stages:

- The development of a generic version of the MOS collector is described first. This is the centralised UMOS collector.
- A distribution for UMOS is then developed and the issues relating to concurrency and distribution are examined.
- The development of the train reclamation mechanism is described. This involves the identification of the distributed state that is required for isolated train detection coupled with the development of a DTA mapping to identify the necessary globally stable properties.
- Finally the development of the car collection mechanism is discussed. The intuition is that when an object moves to a new car the distributed state of the system becomes inconsistent. The car reclamation mechanism therefore consists of a DTA mapping that identifies when the system has returned to a

consistent state and a set of rules governing the maintenance of a car's remset. An abstraction, known as *substitution*, is used here to account for objects moving between cars and the DTA mapping identifies when individual objects become isolated.

The result is the identification of distributed information required by the collector and an explanation of the exact role played by the DTAs. Two DTA mappings are constructed thus describing the events due to the collector and the mutator that correspond to actions on DTA jobs and tasks. However the derived version of DMOS does not require any particular DTA. This differs from the original DMOS algorithm where a Task Balancing implementation is embedded within the *Pointer Tracking* protocol and a ring based DTA is used for train isolation detection.

The final contribution of this thesis is to describe a full implementation of the DMOS collector based on the modularised version of the collection algorithm. The Task Balancing (TB) DTA is used for both isolated train and isolated object detection. The implementation makes opportunistic use of site information provided by the TB DTA to achieve a number of goals:

- The calculation of the set of sites that must be informed when a train becomes isolated.
- The calculation of the set of sites that must be informed to seed the *substitution* protocol.
- To identify when RAL entries may be safely removed.

Chapter 7 describes how the implementation ultimately reduces to two distributed reference counting mechanisms. The first of these counts references to individual objects and is used to detect when objects become isolated. The second reference

counting mechanism counts references into trains and detects when trains become isolated.

Notably, one of the goals of the DMOS grant proposal has not been addressed in this thesis. This goal was to implement a family of DMOS collectors which vary in the DTAs they use and in the underlying systems they support. Instead this thesis has described the derivation of a generic, DTA neutral, version of the DMOS collector. The derived generic algorithm is verified by demonstrating its implementation with a single DTA. In this regard the thesis has posed more questions than it has answered. The generic DMOS collector allows for a wider range of policy than the original algorithm by allowing any DTA to be used to detect either of the globally stable properties and by specifying only a minimal set of rules governing the maintenance of remsets.

9.4.1 Summary

The contributions of this thesis can be summarised as follows:

- A new description of the DMOS algorithm that clearly identifies the distribution and partitioning of the shared state (of the mutators and collectors) and the role of distributed termination detection in identifying globally stable properties of this state.
- A modularised implementation of DMOS which demonstrates a clear separation of concerns between the two collection mechanisms and the local and distributed work required for distributed garbage collection.
- A distributed computational environment, the DPBASE system, which provides an experimental platform for experiments in distributed garbage collector design and implementation.

- A generic description of the Task Balancing DTA and an examination of its suitability for use in distributed garbage collector implementation.
- An examination of the utility of the distributed garbage collector mapping methodology and the description of an extension to this methodology. The extension leads to the derivation of modularised garbage collectors which are described through the definition of a set of *club rules*. The club rules force a separation of concerns within the derived distributed collectors that support independent heterogeneous local behaviour.

Finally it is necessary to return to the hypothesis tested in this thesis. By further developing the mapping methodology a modularised implementation of DMOS has been described. The modularisation provides a clear separation of concerns between the mechanism of the collector and yields an understandable distributed garbage collection mechanism. Through the development of the DTA mappings the role of distributed termination detection within DMOS has been clearly explained. In developing the implementation through a process of stepwise refinement the interaction of the two collection mechanisms has been explained and shown to be safe.

9.5 Future Research

9.5.1 A Formal Proof for Task Balancing

One of the benefits provided by the modularisation of derived collectors is that we have the opportunity to incorporate the formal proofs for the DTA's in the correctness arguments for the collectors. The distributed termination problem has been worked on for 30 years or so and there exists an extensive literature of formal correctness proofs.

However no proof for the Task Balancing DTA has yet been published. Collaborative work on a proof for Task Balancing has taken place in parallel with the work described in this thesis and will be published independently.

9.5.2 A Modularized Formal Proof of DMOS

As result of the modularization of the DMOS collector a formal proof of the collector may also be modularized. Effectively the proof of the DTA is independent from the proof of the collector and therefore may be addressed separately.

Further work on this area will involve forming formal proofs for each of the collection mechanisms within DMOS. The correctness and safety arguments from Chapter 7 give an outline of the mechanisms within the collector that need to be formally proved correct.

9.5.3 Policy Evaluation

The policy space of the DMOS collector, as described in [HMM+97], appears to be large but without first defining a concrete implementation this space cannot be defined. Now that a concrete implementation has been defined it is possible to examine and attempt to evaluate the policy space. However the DMOS implementation poses more questions than it answers regarding policy choice.

Policy can be divided into two areas. The first area of policy is specific to the DMOS implementation. For instance, policy relating to the implementation of idleness detection within a site or policy relating to how messages are batched together and when they are sent.

The second area of policy is more generic and covers the management of trains and cars within DMOS, for instance:

- Car collection order.

- Car capacity - the total size of objects that are associated with a car.
- Train size - the number of cars in a train.
- Choosing a car for object re-association.
- Choosing a train for object promotion.
- RAL maintenance policy - how many entries are maintained for each object?

Future research in this area will involve the implementation of a range of policies and an evaluation of their effect of the operation of the collector. In order to evaluate policy choices a number of performance metrics are required. Since the distributed garbage collector is ultimately part of an automatic memory management system the key metric is that of the overhead due to garbage collection. However there are a number of other metrics that contribute to the overall overhead for instance throughput, pause time, message complexity and delay in garbage identification. An analytical model or simulation system can be used to support the evaluation of the policy space for DMOS.

9.6 Finally

DMOS is a complex distributed garbage collector which appears to exhibit a unique combination of properties and presents a seemingly large policy space. This thesis derives a modularised implementation of DMOS using previously published derivation methodology. Use of the derivation methodology has resulted in a clear separation of concerns within the implementation. The local work of the collector is separated from the distributed work and it has been shown how the collector can be deployed within the DPBASE system while placing no restrictions on the local cache architecture or local and distributed addressing mechanisms.

With the implementation fully defined it is now possible to begin to examine the properties and policy space of the collector.

Appendix A

An Annotated Implementation of DMOS

An annotated implementation of the DMOS collector is now presented. This illustrates the actions taken on car collection, for RAL maintenance and for each reference and substitution event. The purpose of this annotated implementation is to bring together the key points of the DMOS algorithm from Chapters 6, 7 and 8. The presentation style is designed to demonstrate the process of stepwise refinement. The code for car collection and for each event is written in such a way as to separate the required behaviour for each of the three layers described above. The shaded areas are labelled to identify the actions for isolated object detection (L1), object substitution and RAL maintenance (L2) and isolated train detection (L3).

A number of simplifications are made:

- The deferred reference counting optimisation is not used.
- The object copy, remote dereference and remote update mechanism of the DPBASE are not implemented. These high level operations decompose into the lower level reference and substitution events.
- None of the areas of policy are described.

The following annotated implementation also makes a number of assumptions:

- There is a total ordering of sites in the system;
- The home site of an object x can be determined from its reference: designated $\text{home}(\langle x, T \rangle)$;
- The home site for an object $\langle x, T \rangle$ can determine the car holding an object from the object's reference, written $\text{car}(\langle x, T \rangle)$;

- The home site of a train T can be determined from its train number: designated home(T);
- Any function with the name XXXMessage(S,..) represents the sending of an asynchronous message to the designated site S. Such messages, of course, can be batched and sent when opportune. Each of the messages defined in Chapters 6, 7 and 8 are summarised in Table A-1 below.

| Message | Description |
|---------------------------------------|--|
| <i>Substitution</i> message | On the substitution of an object x from train T to train T' , a <i>Substitution</i> message is sent to each site holding a reference to x . |
| <i>Substitution Complete</i> message | On completion of the substitution of an object x a <i>Substitution Complete</i> message is sent to each site that was sent a <i>Substitution</i> message. |
| <i>Isolated Train</i> message | The <i>home</i> site of a train T sends an <i>Isolated Train</i> message to each site holding cars of T when $\text{isolatedTrain}(T)$ terminates. |
| <i>Isolated Object Update</i> message | If a site S becomes idle for $\text{isolatedObject}(\langle x, T \rangle)$ an <i>Isolated Object Update</i> message is sent to the <i>home</i> site of x containing the TB <i>sent</i> and <i>received/completed</i> task counts for S . |
| <i>Isolated Train Update</i> message | If a site S becomes idle for $\text{isolatedTrain}(T)$ an <i>Isolated Train Update</i> message is sent to the <i>home</i> site |

| | |
|-----------------------------|---|
| | of <i>T</i> containing the TB <i>sent</i> and <i>received/completed</i> task counts for <i>S</i> . |
| <i>Task Request</i> message | A site can request a task of a job <i>isolatedTrain(T)</i> by sending a <i>Task Request</i> message to the <i>home</i> site of train <i>T</i> . |
| <i>Task Reply</i> message | The home site of a train <i>T</i> responds to a task request message from a site <i>S</i> by sending a <i>Task Reply</i> message to <i>S</i> . |
| <i>RAL Update</i> message | A <i>Root Reference</i> or inter-car RAL entry for an object <i>x</i> is sent from a site <i>S</i> to the <i>home</i> site of <i>x</i> in an <i>RAL Update</i> message. |

Table A-1 - DMOS Messages

DMOS Implementation Pseudo-Code

sendPointer(x,T,S)

{ // A reference to x is exported from *thisSite* to site S. This site believes x is in train T

L1 isolatedObject(< x,T >,sentCount,S,+1) // increment TB sent count for S for

// isolatedObject(< x,T >)

L3 isolatedTrain(T,sentCount,S,+1) // increment TB sent count for S for

// isolatedTrain(T)

if *thisSite* == home(x)

L2 if there is no remote entry for x

addRemoteRAL(car(x),x,T)

}

receivePointer(x,T)

{ // A reference to x is received at *thisSite*

// The message containing the reference to x, indicates that x is in train T

// *thisSite* believes that x is in train V

//set up object-to-train mapping for x if necessary

L3 if there is no object-to-train mapping for x at *thisSite*

addObjectToTrainMapping(x, T)

// Increment received counts for *thisSite* for isolated train and isolated object jobs

L1 isolatedObject(< x,T >,receiveCount,+1)

L3 isolatedTrain(T,receiveCount,+1) // increment TB receive count at *thisSite* for

// isolatedTrain(T)

//if T is older than V

L1 if T<V

updateIsolatedObjectMessage(home(x),x,T) // send TB update for

// isolatedObject(< x,T >)

```

L3 if taskCount for isolatedTrain( T ) == 0
    updateIsolatedTrainMessage( home( T ),T )    // send TB update for isolatedTrain( T )
}

```

```

copyPointer( x,y )

```

```

{ // On thisSite a copy of a reference to x is created locally in an object y in train U
  // thisSite believes that x is in train T

```

```

L1 trainTaskCount( x,U,+1 )    // increment count of references to x from U at thisSite

```

```

L3 if T ≠ U    // this is an inter-train reference

```

```

    isolatedTrain( T,taskCount,+1 )    // increment task count for isolatedTrain( T )

```

```

L2 if thisSite == home( x )

```

```

    if car( x ) ≠ car( y )

```

```

        interCarReferenceCount( x,+1 )

```

```

    if there is no local entry for x in its car's RAL

```

```

        addLocalRAL( car( x ),x,U )

```

```

    else

```

```

        policyDecision( addLocalRAL,car( x ),x,U )

```

```

}

```

deletePointer(x,T,y,C,U)

{ // On *thisSite* a reference to object x in train T is deleted in an object y in car C of train U

if T ≠ U // this is an inter-train reference

L1 trainTaskCount(x,U,-1) // decrement count of references to x from U at *thisSite*

L3 isolatedTrain(T,taskCount,-1) // decrement task count for isolatedTrain(T)

if *thisSite* == home(x)

L2 if train(x) ≠ C

interCarReferenceCount(x,-1)

if interCarReferenceCount(x) == 0

remove all local RAL entries for x from train(x)

L1 if trainTaskCount for x is empty // *thisSite* is idle for isolatedObject(< x,T >)

if *thisSite* ≠ home(x)

updateIsolatedObjectMessage(home(x),x,T) // send TB update for

// isolatedObject(< x,T >)

else

if isolatedObject(< x,T >) is terminated

detectedIsolatedObject(x,T)

L3 if taskCount for isolatedTrain(T) == 0 // *thisSite* is idle for isolatedTrain(T)

if *thisSite* ≠ home(T)

updateIsolatedTrainMessage(home(T),T) // send TB update for isolatedTrain(T)

else

if isolatedTrain(T) is terminated

detectedIsolatedTrain(T)

}

```

receiveSubstitutionMessage( x,T,T' )

{ // A substitution message arrives at thisSite from site home( x )

  // indicating that x has been promoted from T to train T'

  L1 isolatedObject( < x,T' >,receiveCount,+1 ) // increment TB receive count at thisSite for
                                                // isolatedObject( < x,T' > )

  L3 isolatedTrain( T',receiveCount,+1 ) // increment TB receive count for
                                         // isolatedTrain( T' )

  updateObjectToTrainMapping( x, T' )

  L2 addSubstitutionTableEntry ( < x,T >, < x,T' > )

  L1 trainTaskCount( x,SubstitutionTable,+1 ) // add count for reference to x from
                                              // the SubstitutionTable

  L3 isolatedTrain( T,taskCount,+1 ) // increment count of references into T'

  if thisSite is not idle for isolatedObject( < x,T > )

    for each train Y that holds a reference to x at thisSite

      if Y ≠ T'

        isolatedTrain( T',taskCount,+trainTaskCount( x,Y ) )

    for each train Z that holds a reference to x at thisSite

      if Y ≠ T

        isolatedTrain( T,taskCount,-trainTaskCount( x,Z ) )

    if taskCount for isolatedTrain( T ) == 0 // thisSite is idle for isolatedTrain( T )

      if thisSite ≠ home( T )

        updateIsolatedTrainMessage( home( T ),T ) // send TB update for isolatedTrain( T )

      else

        if isolatedTrain(T) is terminated

          detectedIsolatedTrain( T )

  L1 updateIsolatedObjectMessage( home( x ),x,T ) // send TB update for
                                                // isolatedObject( < x,T > )

}

```

```
receiveSubstitutionCompleteMessage( x,T,T' )
```

```
{
```

```
L2 removeSubstitutionTableEntry ( < x,T >, < x,T' > )
```

```
L1 trainTaskCount( x,SubstitutionTable,-1 )           // subtract count for reference to x from  
                                                       // the SubstitutionTable
```

```
if trainTaskCount for x is empty                       // thisSite is idle for  
                                                       // isolatedObject( < x,T' > )
```

```
updateIsolatedObjectMessage( home( x ),x,T' ) // send TB update for  
                                                       // isolatedObject( < x,T' > )
```

```
L3 isolatedTrain( T',taskCount,-1 )                   // decrement count of references into train T'
```

```
if taskCount for isolatedTrain( T' ) == 0            // thisSite is idle for isolatedTrain( T' )  
if thisSite ≠ home( T' )
```

```
updateIsolatedTrainMessage( home( T' ),T' ) // send TB update for isolatedTrain( T' )
```

```
else
```

```
if isolatedTrain(T) is terminated
```

```
detectedIsolatedTrain( T' )
```

```
}
```

```
receiveIsolatedObjectUpdate( x,T )
```

```
{
```

```
processUpdate( u,x,T )                               // add the RC and sent values to the task count
```

```
                                                       // structure for isolatedObject( x,T ) at thisSite
```

```
if isolatedObject( < x,T > ) is terminated
```

```
L1 detectedIsolatedObject( < x,T > )
```

```
else
```

```
if <x,T>→<x',T'> is in the substitution table
```

```
for each site Z with a non-zero task count for isolatedObject( < x,T > )
```

```
if Z has not been sent a substitution message for <x,T>→<x',T'>
```

```
L2 substitutionMessage( Z,x,T,T' )
```

```

    if all remote sites have a zero task count
        remove all remote RAL entries for x from car( x )
    }

receiveIsolatedTrainUpdate( u,T,S )
{ // A TB update message u has been received for isolatedTrain( T ) from site S
L3 if S had cars of T
    addCarsList( T,S )
    processUpdate( u,T ) // add the RC and sent values to the task count
                        // structure for isolatedTrain( T ) at thisSite
    if isolatedTrain(T) is terminated
        detectedIsolatedTrain( T )
}

detectedIsolatedTrain( T )
{ // The home site thisSite for train T has detected termination of isolatedTrain( T )
    // Tell each site that holds cars of train T that T is isolated and its cars can be reclaimed
L3 for each site S in carsList( T )
    isolatedTrainMessage( S,T )
    for each car C of T at thisSite
        reclaimCar( C )
}

```

detectedIsolatedObject(x,T)

{ // The home site *thisSite* for object x has detected termination of isolatedObject(< x,T >)

if <x,T>→<x',T'> is in the substitution table

for each site Z that was sent a substitution message

L2 substitutionCompleteMessage(Z,x,T,T')

L1 trainTaskCount(x,SubstitutionTable,-1) // subtract count for reference to x from

// the SubstitutionTable

if trainTaskCount for x is empty

// *thisSite* is idle for

// isolatedObject(< x,T' >)

if isolatedObject(< x,T' >) is terminated

detectedIsolatedObject(x,T')

L3 isolatedTrain(T',taskCount,-1) // decrement count of references to T'

if taskCount for isolatedTrain(T') == 0

// *thisSite* is idle for isolatedTrain(T')

if *thisSite* ≠ home(T')

updateIsolatedTrainMessage(home(T'),T') // send TB update for

// isolatedTrain(T')

else

if isolatedTrain(T) is terminated

detectedIsolatedTrain(T')

}

receiveIsolatedTrainMessage(T)

{ // The train T is isolated so reclaim each car of T held at *thisSite*

L3 for each car C of train T held at *thisSite*

reclaimCar(C)

removeRemoteTrainTableEntry(T)

}

```

collectCar( C )

{ // Re-associate each object in the car C referenced by C's RAL then reclaim the car
  for each object x in the local root set
    if home( x ) == thisSite
      L2      addLocalRAL( car( x ),x,thisSite,RootReference )
    else
      if x is not in RAL update message log
        RALUpdateMessage( home( x ),x,RootReference )

  for each object x in C which has a Root Reference RAL entry
    let targetTrain = policyDecision( chooseTrain,x )
    re-associateObject( x,C,train( x ),targetTrain )
    if train( x ) ≠ targetTrain
      substitute( x,train( x ),targetTrain )

  for each object x in C which has an RAL entry for a train U ≠ train( x )
    let targetTrain = policyDecision( chooseTrain,x )
    if thisSite holds no tasks of isolatedTrain( targetTrain )
      taskRequestMessage( T,x )
      re-associateObject( x,C,train( x ),train( x ) )
    else
      re-associateObject( x,C,train( x ),targetTrain )
      if train( x ) ≠ targetTrain
        substitute( x,train( x ),targetTrain )

  for each object x in C which only has RAL entries for train( x )
    re-associateObject( x,C,train( x ),train( x ) )

  reclaimCar( C )
}

```

```
substitute( x,T,T' )
```

```
{ // The object x has been promoted from train T to train T'
```

```
L2 addSubstitutionTableEntry ( < x,T >, < x,T' > )
```

```
L1 trainTaskCount( x,SubstitutionTable,+1 )
```

```
// add count for reference to x from
```

```
// the SubstitutionTable
```

```
L3 isolatedTrain( T',taskCount,+1 )
```

```
// increment count of references into T'
```

```
L2 for each site S with a non-zero task count for isolatedObject( < x,T > )
```

```
sendPointer( x,T',S )
```

```
substitutionMessage( S,x,T,T' )
```

```
if thisSite is not idle for isolatedObject( < x,T > )
```

```
for each train Y that holds a reference to x at thisSite
```

```
if Y ≠ T'
```

```
isolatedTrain( T',taskCount,+trainTaskCount( x,Y ) )
```

```
for each train Z that holds a reference to x at thisSite
```

```
if Y ≠ T
```

```
isolatedTrain( T,taskCount,-trainTaskCount( x,Z ) )
```

```
if taskCount for isolatedTrain( T ) == 0
```

```
// thisSite is idle for isolatedTrain( T )
```

```
if thisSite == home( T )
```

```
if isolatedTrain(T) is terminated
```

```
detectedIsolatedTrain( T )
```

```
else
```

```
updateIsolatedTrainMessage( home( T ),T ) // send TB update for isolatedTrain( T )
```

```
L1 if isolatedObject( < x,T > ) is terminated
```

```
detectedIsolatedObject( < x,T > )
```

```
}
```

```
re-associateObject( x,C,U,T )
```

```
{ // Re-associate the object x from car C of train U to a car of train T
```

```
  remove x from C
```

```
  add x to car of T
```

```
L2 move all RAL entries for x from C to car( x )
```

```
  if x is not root referenced at thisSite
```

```
    remove all Root Reference local RAL entries for x from car( x )
```

```
    addLocalRAL( car( x ),x,T )
```

```
  if U ≠ T
```

```
    for each reference to an object y in x
```

```
      copyPointer( y,x )
```

```
      // create a pointer from x in car( x ) to y
```

```
      deletePointer(y,train( y ),x,C,U)
```

```
      // delete the 'old' pointer from x in car C
```

```
    if y is local object
```

```
      // The copyPointer function only adds a local RAL entry if there is not one already.
```

```
      // So...
```

```
      addLocalRAL( car( y ),y,T )
```

```
    else
```

```
      sendRALUpdateMessage( S,y,[y,T] )
```

```
  for each reference to x in car( x )
```

```
    interCarReferenceCount( x,-1 )
```

```
}
```

```
receiveRALUpdateMessage( x,CONTENTS )
```

```
{ // An RAL update message for x has been received from site S
```

```
L2 if CONTENTS == [x,U]
```

```
    addRemoteRAL( car( x ),x,U )           // x is referenced from train U
```

```
else
```

```
    if CONTENTS == [x,S,RootReference]     // x is root referenced at S
```

```
        addRemoteRAL( car( x ),x,S,RootReference )
```

```
    else                                     // x is no longer root referenced at S
```

```
        remove all Root Reference RAL entries for site S and object x from car( x )
```

```
        addRemoteRAL( car( x ),x,train( x ) )
```

```
}
```

```
receiveTaskRequestMessage( T,id )
```

```
{ // A request for a task of isolatedTrain( T ) has been received from a site S
```

```
    // The id parameter is returned to the sending site to allow that site
```

```
    // to match the reply with the request
```

```
L3 if isolatedObject( < x,T > ) is terminated
```

```
    requestedTrainTaskMessage( S,T,"Terminated",id )
```

```
else
```

```
    if T is oldest train created at this site
```

```
        requestedTrainTaskMessage( S,T,"Terminated",id )
```

```
    else
```

```
L3    isolatedTrain( T',taskCount,+1 )     // increment count of references into train T
```

```
        requestedTrainTaskMessage( S,T,"Task of isolatedTrain( T )",id )
```

```
}
```

```

receiveRequestedTrainTask( T,CONTENTS,id )

{ // A Requested Train Task message has been received in response to
  // a Task Request Message sent previously. 'id' corresponds to a
  // Task Request Message that was sent for object x

  if CONTENTS == "Terminated"

    remove all remote RAL entries for x in car( x ) that specify train T

L2  addRemoteRAL( car( x ),x, train( x ) )

    remove all local RAL entries for x in car( x ) that specify train T

    addLocalRAL( car( x ),x, train( x ) )

  else

    addRequestedTaskTableEntry( x,T )

L3  isolatedTrain( T,taskCount,+1 ) // increment task count for isolatedTrain( T )
}

```

```

reclaimCar( C )

{ // remove each object from car C and remove the car from its train

  for each object x in C's reference array

    removeReferenceArrayEntry( x,C ) // drop the object x from car C

    removeAddressTranslationEntry( x ) // remove x from the  $DA_{sym} \rightarrow CA$  or  $DA \rightarrow CA$ 
                                        // address translation table at thisSite

  for each reference in x to an object y in train T

    if  $T \neq \text{train}( C )$ 

      deletePointer( y,train( C ),T ) // delete each pointer (in x) to an object in
                                        // another train

    removeCarFromTrain( C, train( C ) )

}

```

References

- [ACC82] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott, "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, Vol. 17, No. 7, p. 24-31 (1982)
- [AR98] S.E. Abdullahi and G.A. Ringwood, "Garbage Collecting the Internet: A Survey of Distributed Garbage Collection", *ACM Computing Surveys*, Vol. 30, No. 3, p. 330-373 (1998)
- [Aug87] L. Augusteijn, "Garbage Collection in a Distributed Environment", in *Parallel Architectures and Languages Europe (PARLE'87)*, *Lecture Notes in Computer Science*, Vol. 259, p. 75-93 (1987)
- [Bad83] S.B. Baden, "Low-Overhead Storage Reclamation in the Smalltalk Virtual Machine", in *Smalltalk-80: Bits of History, Words of Advice*, p. 331-342, G. Krasner, Editor. Addison-Wesley (1983)
- [BDF+01] W. Brodie-Tyrrell, H. Detmold, K. Falkner, M. Lowry, R. Morrison, D.S. Munro, S.J. Norcross, T. Olds, Z. Tian, and F. Vaughan, "Design of the Distributed ProcessBase Architecture", University of Adelaide (2001)
- [BEN+93] A.D. Birrell, D. Eyers, G. Nelson, S. Owicki, and E. Wobber, "Distributed Garbage Collection for Network Objects", DEC SRC (1993)
- [Bev87] D.I. Bevan, "Distributed Garbage Collection Using Reference Counting", in *Lecture Notes in Computer Science 258, 259*, Y. Bekkers and J. Cohen, Editors. Springer-Verlag (1987)
- [BHM+01] S.M. Blackburn, R.L. Hudson, R. Morrison, J.E.B. Moss, D.S. Munro, and J.N. Zigman, "Starting with Termination: A Methodology for Building Distributed Garbage Collection Algorithms", in *24th Australasian Computer Science Conference (ACSC2001)*, Gold Coast, Queensland, p. 20-28 (2001)
- [BLS+71] P. Branquart, J. Lewi, M. Sintzoff, and P.L. Wodon, "The composition of semantics in Algol 68", *Communications of the ACM*, Vol. 14, No. 11 (1971)

- [BMM80] P.J. Bailey, P. Maritz, and R. Morrison, "The S-algol Abstract Machine", University of St Andrews (1979)
- [BNO+93] A.D. Birrell, G. Nelson, S. Owicki, and E. Wobber, "Network Objects", *Operating System Review*, Vol. 27, No. 5, p. 217-230 (1993)
- [BSS+95] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, and C. Packer, "Beowulf: A Parallel Workstation for Scientific Computation", in *24th International Conference on Parallel Processing*, p. 11-14 (1995)
- [BZ99] S.M. Blackburn and J.N. Zigman, "Ensuring the Safety of Distributed Garbage Collection in DMOS", Australian National University (1999)
- [CDK01] G.F. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems, Concept and Design". 3rd ed, Addison-Wesley (2001)
- [Che70] C.J. Cheney, "A Non-Recursive List Compacting Algorithm", *Communications of the ACM*, Vol. 13, No. 11, p. 677-678 (1970)
- [CL85] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 3, No. 1, p. 63-75 (1985)
- [CM82] K.M. Chandy and J. Misra, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, p. 37-43 (1982)
- [CM86] K.M. Chandy and J. Misra, "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 3, p. 326-343 (1986)
- [CM98] T. Camp and J. Matocha, "A Taxonomy of Distributed Termination Algorithms", *The Journal of Software and Systems*, Vol. 43, p. 207-221 (1998)
- [Col60] G.E. Collins, "A Method for Overlapping and Erasure of Lists", *Communications of the ACM*, Vol. 3, No. 12, p. 655-657 (1960)

- [DAN02] P. Drayton, B. Albahari, and T. Neward, "C# in a Nutshell", O'Reilly, ISBN: 0596001819 (2002)
- [DB76] L.P. Deutsch and D.G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector", *Communications of the ACM*, Vol. 19, No. 9, p. 522-526 (1976)
- [Der90] M.H. Derbyshire, "Mark Scan Garbage Collection on a Distributed Architecture", *Lisp and Symbolic Computation*, Vol. 3, No. 2, p. 135-170 (1990)
- [DFG83] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren, "Derivation of a Termination Detection Algorithm for Distributed Computations", *Information Processing Letters*, Vol. 16, No. 5, p. 217-219 (1983)
- [DFM+03] H. Detmold, K. Falkner, D.S. Munro, T. Olds, R. Morrison, and S.J. Norcross, "An Integrated Approach to Static Safety of Web Applications", in *12th International World Wide Web Conference*, Budapest (2003)
- [Dij87] E.W. Dijkstra, "Schmuel Safra's Version of Termination Detection", University of Texas (1987)
- [DLM+78] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation", *Communications of the ACM*, Vol. 21, No. 11, p. 966-975 (1978)
- [DS80] E.W. Dijkstra and C.S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters*, Vol. 11, No. 1, p. 1-4 (1980)
- [FDH+04] K. Falkner, H. Detmold, D. Howard, D.S. Munro, R. Morrison, and S.J. Norcross, "Unifying Static and Dynamic Approaches to Evolution through the Compliant Systems Architecture", in *Thirty-Seventh Hawaii International Conference on System Sciences (HICSS-37)* (2004)
- [FR82] N. Francez and M. Rodeh, "Achieving Distributed Termination without Freezing", *IEEE Transactions on Software Engineering*, Vol. 8, No. 3, p. 287-292 (1982)

- [Fra80] N. Francez, "Distributed Termination", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, p. 42-55 (1980)
- [Fuc95] M. Fuchs, "Garbage Collection on an Open Network", in *Lecture Notes in Computer Science 986*, p. 251-265, H.G. Baker, Editor. Springer-Verlag (1995)
- [GJS+00] J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification". 2nd ed. Java Series, Addison Wesley, ISBN: 0201310082 (2000)
- [Gol89] B. Goldberg, "Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme", in *ACM SIGPLAN Conference on Programming Languages Design and Implementation* (1989)
- [GR83] A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison Wesley (1983)
- [GS93] S. Grarup and J. Seligmann, "Incremental Mature Garbage Collection", M.Sc. Thesis, *Aarhus University* (1993)
- [HK82] P. Hudak and R.M. Keller, "Garbage Collection and Task Deletion in Distributed Applicative Processing Systems", in *ACM Symposium on LISP and Functional Programming* (1982)
- [HM92] R.L. Hudson and J.E.B. Moss, "Incremental Garbage Collection for Mature Objects", in *Lecture Notes in Computer Science 637*, p. 388-403. Springer-Verlag (1992)
- [HMM+97] R.L. Hudson, R. Morrison, J.E.B. Moss, and D.S. Munro, "Garbage Collecting the World: One Car at a Time", *ACM SIGPLAN Notices*, Vol. 32, No. 10, p. 162-175 (1997)
- [HMM+98] R.L. Hudson, R. Morrison, J.E.B. Moss, and D.S. Munro, "Where have all the pointers gone?" in *Computer Science '98*, p. 107-119, C. McDonald, Editor. Springer (1998)
- [Hoa78] C.A.R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, p. 666-678 (1978)

- [Hug85] R.J.M. Hughes, "A Distributed Garbage Collection Algorithm", in *Lecture Notes in Computer Science 201*, p. 256-272. Springer-Verlag (1985)
- [JL96] R. Jones and R. Lins, "Garbage Collection", John Wiley & Sons (1996)
- [Lam78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, p. 558-565 (1978)
- [LH83] H. Lieberman and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects", *Communications of the ACM*, Vol. 26, No. 6, p. 419-429 (1983)
- [Lin92] R.D. Lins, "Cyclic Reference Counting with Lazy Mark-Scan", *Information Processing Letters*, Vol. 44, p. 215-220 (1992)
- [LJ91] R.D. Lins and R.E. Jones, "Cyclic Weighted Reference Counting", University of Kent, Canterbury (1991)
- [LL86] B. Liskov and R. Ladin, "Highly-Available Distributed Service and Fault-Tolerant Distributed Garbage Collection", in *5th ACM Symposium on Principles of Distributed Computing*, Calgary, Canada, p. 29-39 (1986)
- [LL92] R. Ladin and B. Liskov, "Garbage Collection of a Distributed Heap", in *12th International Conference on Distributed Computing Systems*, Yokohama, Japan, p. 708-715 (1992)
- [LM78] F. Lockwood Morris, "A Time and Space Efficient Garbage Collection Algorithm", *Communications of the ACM*, Vol. 21, No. 8, p. 662-665 (1978)
- [LM86] C.W. Lermen and D. Maurer, "A Protocol for Distributed Reference Counting", in *ACM Conference on LISP and Functional Programming*, Cambridge, Massachusetts, p. 343-350 (1986)
- [LQP92] B. Lang, C. Queinnec, and J.M. Piquer, "Garbage Collecting the World", in *19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, p. 39-50 (1992)

- [LY99] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification". 2nd ed. Java Series, Addison Wesley, ISBN: 0201432943 (1999)
- [Mat87] F. Mattern, "Algorithms for Distributed Termination Detection", *Distributed Computing, Vol. 2, No. 3, p. 161-175* (1987)
- [Mat89] F. Mattern, "Global Quiescence Detection Based on Credit Distribution and Recovery", *Information Processing Letters, Vol. 30, No. 4, p. 195-200* (1989)
- [MBC+89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle, "The Napier88 Reference Manual", Universities of Glasgow and St Andrews (1989)
- [MBG+00] R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro, and B.C. Warboys, "A Compliant Persistent Architecture", *Software - Practice and Experience, Special Issue on Persistent Object Systems, Vol. 30, No. 4, p. 363-386* (2000)
- [MBG+99] R. Morrison, D. Balasubramaniam, M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro, and B.C. Warboys, "ProcessBase Reference Manual (Version 1.0.6)", Universities of St Andrews and Manchester (1999)
- [MBG+99a] R. Morrison, D. Balasubramaniam, M. Greenwood, G.N.C. Kirby, K. Mayes, D.S. Munro, and B.C. Warboys, "ProcessBase Abstract Machine Manual (Version 2.0.6)", Universities of St Andrews and Manchester (1999)
- [MBM+99] D.S. Munro, A.L. Brown, R. Morrison, and J.E.B. Moss, "Incremental Garbage Collection of a Persistent Object Store using PMOS", in *Advances in Persistent Object Systems, p. 78-91*, R. Morrison, M. Jordan, and M.P. Atkinson, Editors. Morgan Kaufmann: San Francisco (1999)
- [McC60] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine", *Communications of the ACM, Vol. 3, No. 4, p. 184-195* (1960)
- [MFL+01] D.S. Munro, K.E. Falkner, M.C. Lowry, and F. Vaughan, "Mosaic: A Non-intrusive Complete Garbage Collector for DSM Systems", in *1st IEEE/ACM International Symposium on Cluster Computing and the Grid, 3rd*

- International Workshop on Software Distributed Shared Memory*, Brisbane, Australia, p. 539-546 (2001)
- [Mir87] E. Miranda, "BrouHaHa- A portable Smalltalk interpreter", in *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, Orlando, Florida, p. 354-365 (1987)
- [Mis83] J. Misra, "Detecting Termination of Distributed Computations Using Markers", in *2nd ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, p. 290-294 (1983)
- [MKB99] R. Morrison, G.N.C. Kirby, and D. Balasubramaniam, "Collecting Distributed Garbage using the DMOS Family of Algorithms", EPSRC Distributed Information Systems Initiative(1999)
- [MKI+95] M. Maeda, H. Konaka, Y. Ishikawa, T. Tomokiyo, A. Hori, and J. Nolte, "On-the-fly Global Garbage Collection Based on Partly Mark-Sweep", in *Lecture Notes in Computer Science 986*, p. 283-296, H.G. Baker, Editor. Springer-Verlag (1995)
- [ML95] U. Maheshwari and B. Liskov, "Collecting Cyclic Distributed Garbage by Controlled Migration", in *14th ACM Symposium on Principles of Distributed Computing*, p. 57-63 (1995)
- [ML97] U. Maheshwari and B. Liskov, "Collecting Distributed Garbage Cycles by Back Tracing", in *16th ACM Symposium on Principles of Distributed Computing*, Santa Barbara, California, p. 239-248 (1997)
- [ML97a] U. Maheshwari and B. Liskov, "Partitioned Garbage Collection of a Large Object Store", in *ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, p. 313-323 (1997)
- [MMH96] J.E.B. Moss, D.S. Munro, and R.L. Hudson, "PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores", in *7th International Workshop on Persistent Object Systems (POS7)*, Cape May, NJ, USA (1996)

- [Mor79] R. Morrison, "S-algol Language Reference Manual", University of St Andrews (1979)
- [MTH89] R. Milner, M. Tofte, and R. Harper, "The Definition of Standard ML", MIT Press, Cambridge, Massachusetts (1989)
- [MWL90] A.D. Martinez, R. Wachenchauser, and R.D. Lins, "Cyclic Reference Counting with Local Mark Scan", *Information Processing Letters*, Vol. 34, p. 31-35 (1990)
- [NFB+01] S.J. Norcross, K. Falkner, D. Balasubramaniam, G. Kirby, R. Morrison, and D.S. Munro, "An Object Cache for the Distributed ProcessBase Interpreter", University of Adelaide (2001)
- [NMM+03] S.J. Norcross, R. Morrison, D.S. Munro, and H. Detmold, "Implementing a Family of Distributed Garbage Collectors", in *26th Australasian Computer Science Conference (ACSC 2003)*, Adelaide, Australia, p. 161-170 (2003)
- [Piq91] J.M. Piquer, "Indirect Reference Counting", in *Parallel Architectures and Languages Europe (PARLE '91)*, p. 150-165 (1991)
- [PS95] D. Plainfossé and M. Shapiro, "A Survey of Distributed Garbage Collection Techniques", in *Lecture Notes in Computer Science 986*, p. 211-249, H.G. Baker, Editor. Springer-Verlag (1995)
- [Ran83] S.P. Rana, "A Distributed Solution to the Distributed Termination Problem", *Information Processing Letters*, Vol. 17, p. 43-46 (1983)
- [RJ96] H. Rodrigues and R. Jones, "A Cyclic Distributed Garbage Collector for Network Objects", in *Lecture Notes in Computer Science 1151*, p. 123-140, Ö. Babaoglu and K. Marzullo, Editors. Springer (1996)
- [Rud86] M. Rudalics, "Distributed Copying Garbage Collection", in *ACM Conference on Lisp and Functional Programming*, p. 364-372 (1986)
- [Rud90] M. Rudalics, "Correctness of Distributed Garbage Collection Algorithms", Johannes Kepler Universität, Linz, Austria (1990)

- [Sch93] M.D. Schroeder, "A State of the Art Distributed System: Computing with BOB", in *Distributed Systems*, p. 1-16, S. Mullender, Editor. Addison-Wesley (1993)
- [SF86] N. Shavit and N. Francez, "A New Approach to Detection of Locally Indicative Stability", in *Lecture Notes in Computer Science 226*, p. 344-358, L. Kott, Editor. Springer-Verlag (1986)
- [Tel94] G. Tel, "Introduction to Distributed Algorithms", Cambridge University Press, ISBN: 0521470692 (1994)
- [TL86] G. Tel and J. Leeuwen, "The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols", University of Utrecht (1986)
- [TM93] G. Tel and F. Mattern, "The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 1, p. 1-35 (1993)
- [Top84] R.W. Topor, "Termination Detection for Distributed Computations", *Information Processing Letters*, Vol. 18, No. 20, p. 33-36 (1984)
- [Ung84] D. Ungar, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm", *ACM SIGPLAN Notices*, Vol. 19, No. 5, p. 157-167 (1984)
- [Wen80] K.S. Weng, "An Abstract Implementation for a Generalised Dataflow Language", MIT (1980)
- [Wil92] P.R. Wilson, "Uniprocessor Garbage Collection Techniques", in *Lecture Notes in Computer Science 637*, p. 1-42. Springer-Verlag (1992)
- [WJN+95] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", in *Lecture Notes in Computer Science 986*, p. 1-116, H.G. Baker, Editor. Springer-Verlag (1995)
- [WW87] P. Watson and I. Watson, "An Efficient Garbage Collection Scheme for Parallel Computer Architecture", in *Lecture Notes in Computer Science 258, 259*, p. 432-443, Y. Bekkers and J. Cohen, Editors. Springer-Verlag (1987)