Implementing A Cyclic Distributed Garbage Collector for a Heterogeneous System with Space Failures

Supervisor: Susan Eisenbach
Sanjay Bhudia,
sab1@doc.ic.ac.uk
MEng 4, Software Engineering
Department of Computing

Imperial College of Science, Technology and Medicine
University of London.
Garbage collection is a term for automatic memory management. Manual management of the heap is not an issue as software development today has progressed beyond single developer projects to large scale efforts, often involving hundreds of developers. Uni-processor schemes exist and are becoming widely used, such as in Java.

In the world of distributed systems development, automatic memory management techniques are still rare. This is where it is needed most though, because applications can span hundreds of processing nodes, and remote object references make it impossible to manage the heap manually.

This thesis describes the present distributed garbage collection techniques. The design and implementation of a distributed object-oriented system referred to as Stub-Scion Pair Chains is presented. This system is closely coupled with an acyclic distributed garbage collector. The design and implementation of a cyclic collector as an extension is also given.
I would like to thank Susan Eisenbach for accepting to supervise my project. Her help spanned from giving direction and focus in the research phase, through to the design choices taken and discussing implementation issues. Her reviews of the drafts of this document helped. She has taken the time and interest in supervising this project, even if most of the times I have appeared at her door unannounced, especially toward the end of the project. I am also grateful for her setting up an external supervisor, who is one of her ex-PhD students, Dr. Matthias Radestock.

I thank Matthias Radestock for taking the time to discuss the initial directions on the project and listening to my ideas. Explaining the actor paradigm in general and Rosette / TECCware in particular helped me to understand the requirements better. Thanks for reading the outsourcing document and giving feedback on that. Also, giving tips and pointers via email at times when the project seemed intractable helped a lot.

Sanjay Bhudia,
Friday, 18 June 1999.
Table Of Contents

ABSTRACT .................................................................................................................................. III
ACKNOWLEDGMENTS ................................................................................................................ V
TABLE OF CONTENTS ................................................................................................................ I
TABLE OF FIGURES .................................................................................................................. IX
TABLE OF EQUATIONS .............................................................................................................. XI
TABLE OF TABLES .................................................................................................................. XI
TABLE OF LISTINGS ................................................................................................................. XII

CHAPTER 1

INTRODUCTION .................................................................................................................... 2
1.1 OVERVIEW ...................................................................................................................... 2
1.2 CONTRIBUTION ............................................................................................................... 3
1.3 STRUCTURE OF THIS THESIS ....................................................................................... 4

CHAPTER 2

BACKGROUND ....................................................................................................................... 6
2.1 HISTORY OF MEMORY ALLOCATION .......................................................................... 6
  2.1.1 Static Allocation ...................................................................................................... 6
  2.1.2 Stack Allocation .................................................................................................... 7
  2.1.3 Heap Allocation ................................................................................................... 7
  2.1.4 Problems of Explicit Heap Allocation ................................................................ 8
    Garbage .......................................................................................................................... 9
    Dangling References ................................................................................................. 9
    Ownership of Memory .............................................................................................. 10
  2.1.5 A Case for Garbage Collection ............................................................................. 11
CHAPTER 3 DISTRIBUTED SYSTEMS ..............................................14
3.1 INTRODUCTION.............................................................................................................. 14
3.2 HISTORY OF DISTRIBUTED PROCESS INTERACTION MECHANISMS ..................... 16
3.3 DISTRIBUTED OBJECT SYSTEMS.................................................................................. 17
   3.3.1 Distributed Shared Memory .................................................................................. 17
   3.3.2 Remote References Systems ................................................................................ 17
3.4 NETWORK OBJECTS PARADIGM.................................................................................... 18
3.5 REMOTE REFERENCE MODEL....................................................................................... 18
   3.5.1 Spaces.................................................................................................................... 19
   3.5.2 Stubs and Scions.................................................................................................... 19
3.6 OPERATIONS ON REFERENCES .................................................................................. 20
   3.6.1 Reference Creation ............................................................................................... 20
   3.6.2 Reference Duplication .......................................................................................... 21
   3.6.3 Reference Deletion ............................................................................................... 23
   3.6.4 Differences Amongst Systems ............................................................................... 23
3.7 THE ACTOR PARADIGM............................................................................................... 24

CHAPTER 4 GARBAGE COLLECTION ..............................................................................26
4.1 NOTATION AND DEFINITIONS .................................................................................... 26
   4.1.1 Mutator, Allocator and Collector.......................................................................... 26
   4.1.2 Liveness.................................................................................................................. 26
   4.1.3 Two Phase Abstraction......................................................................................... 27
   4.1.4 Fragmentation....................................................................................................... 27
4.2 CLASSICAL ALGORITHMS ......................................................................................... 28
   4.2.1 Reference Counting ............................................................................................... 28
      Problem with Cycles....................................................................................................... 29
      Efficiency................................................................................................................... 30
   4.2.2 The Mark-Sweep Collector..................................................................................... 30
      Strengths..................................................................................................................... 31
      Weaknesses................................................................................................................. 31
   4.2.3 Mark-Compact Collectors..................................................................................... 32
   4.2.4 Copy Collectors..................................................................................................... 32
      Strengths..................................................................................................................... 33
      Weaknesses................................................................................................................. 33
   4.2.5 Generational Techniques....................................................................................... 33
CHAPTER 5

PROJECT REQUIREMENTS .................................................................36

5.1 REQUIREMENTS FROM OUTLINE ..................................................36
5.2 GENERAL REQUIREMENTS OF A DGC ........................................37
5.3 REQUIREMENTS ANALYSIS ..........................................................37
  5.3.1 Requirement Conflicts ...........................................................38
    Comprehensiveness and Integration with Local Collectors ..............38
    Fault Tolerant and using Reliable Communication Media ..............38
    Interoperable with CORBA .........................................................38

CHAPTER 6

DESIGN SPACE FOR DISTRIBUTED GC ........................................42

6.1 GARBAGE DETECTION AS A CONSISTENCY PROBLEM ..............42
6.2 DESIGN SPACE FOR DISTRIBUTED GC ........................................43
  6.2.1 Concurrency and Types of Objects ........................................45
    Passive Objects ...........................................................................45
    Threads in Java ...........................................................................45
    Active Objects ...........................................................................45
    Actors .......................................................................................46
  6.2.2 Communication Semantics ....................................................46
    Types of Communication Networks ............................................46
    Broadcast Channel Networks ....................................................46
    Point-to-Point Networks .............................................................47
    Modelling on the Most Primitive Types of Networks ....................47
    Synchronous Networks ..............................................................47
    Asynchronous Networks .............................................................48
    Impossibility Results .................................................................48
    Impossibility of Consensus in Asynchronous Systems ..................48
    Impossibility of Atomic Commitment under Network Partitions .......48
  6.2.3 Space Failure Semantics .........................................................48
  6.2.4 Reference Consistency Protocol .............................................50
    Distributed Cycles .................................................................51
    Dealing with Cycles .................................................................51
  6.2.5 Nature of Spaces and Addressing within Spaces ....................52
    Object Hierarchy .................................................................52
  6.2.6 Cross-Space Reference Schemes ..........................................53
    Globally Unique Identifiers .....................................................53
    Forwarders ...............................................................................54
CHAPTER 7

DISTRIBUTED GARBAGE COLLECTION ALGORITHMS

7.1 REFERENCE COUNTING APPROACHES ................................................................. 56

7.1.1 Distributed Reference Counting ................................................................... 56
    Race Conditions ................................................................................................... 56
    Acknowledge Messages ....................................................................................... 57
    Fault Tolerance Issues ....................................................................................... 58
    Efficiency Issues .................................................................................................. 58

7.1.2 Weighted Reference Counting .................................................................... 58
    Weight Exhaustion ............................................................................................... 60
    Fault Tolerance Issues ....................................................................................... 60

7.1.3 Optimised Weighted Reference Counting .................................................... 61
    Reference Duplication ......................................................................................... 62
    Scalability and Fault Tolerance Issues ................................................................. 63

7.1.4 Indirect Reference Counting ........................................................................ 61
    Reference Duplication ......................................................................................... 62

7.2 REFERENCE LISTING APPROACHES ................................................................ 63

7.2.1 Stub-Scion Pair Chains ................................................................................ 64

7.2.2 Garbage Collection for Network Objects ..................................................... 64
    Data Structures .................................................................................................... 64
    Duplicating a Reference ...................................................................................... 65
    Avoiding Race Conditions .................................................................................. 67

7.3 HYBRID APPROACHES ....................................................................................... 68

7.3.1 Complementary Tracing .............................................................................. 68

7.3.2 Object Migration .......................................................................................... 69

7.3.3 Trial Deletion ................................................................................................ 70

7.3.4 Local Tracing ................................................................................................ 71

7.3.5 Summary of Hybrid Approaches .................................................................. 71

7.4 DISTRIBUTED TRACING APPROACHES .......................................................... 71

7.4.1 Distributed Tracing Problem ......................................................................... 71

7.4.2 Hughes’ Algorithm ...................................................................................... 74
    Space Failure Issues ............................................................................................ 74

7.5 GARBAGE COLLECTING ACTORS .................................................................... 77

7.5.1 Halstead’s Algorithm .................................................................................... 77

7.5.2 Washabaugh’s Algorithm .............................................................................. 77
CHAPTER 9

IMPLEMENTATION ........................................................................................................... 108

9.1 OTHER IMPLEMENTATIONS....................................................................................... 108
  ObjectSpace Voyager..................................................................................................... 108
  Java RMI...................................................................................................................... 109
  OmniORB 2................................................................................................................... 109

9.2 WHY RE-INVENT THE WHEEL? ................................................................................. 109
  ORBacus..................................................................................................................... 109
  JaORB.......................................................................................................................... 110
  Implementing SSP Chains.............................................................................................. 110

9.2.2 Choice of Java for Implementation........................................................................ 110
  Existence of a local garbage collector........................................................................ 110
  Java's Local Collector................................................................................................... 111
  A Collector for C++...................................................................................................... 111
  Run Time Type Identification...................................................................................... 111
  Reflection..................................................................................................................... 111

9.3 SYSTEM OVERVIEW.................................................................................................. 112

9.4 SYSTEM CLASSES...................................................................................................... 112
  9.4.1 Spaces................................................................................................................ 112
  9.4.2 Locations............................................................................................................. 113
  9.4.3 Locator................................................................................................................. 114
  9.4.4 Stub...................................................................................................................... 115
  9.4.5 Scion.................................................................................................................... 117

9.4.6 Space Manager...................................................................................................... 117
  Space Identification....................................................................................................... 117
  Service Scions............................................................................................................... 117
  Stub / Scion Management............................................................................................. 117

9.4.7 Threshold Table..................................................................................................... 118
  Caching Classes............................................................................................................ 118

9.5 TRANSPORT PROTOCOL.......................................................................................... 119
  9.5.1 EndPoint.............................................................................................................. 119
  9.5.2 ServerEndPoint................................................................................................... 120

9.5.3 SpaceDaemon........................................................................................................ 121

9.5.4 MessageDaemon................................................................................................... 122

9.5.5 MessageDispatch................................................................................................. 123

9.6 PRESENTATION PROTOCOL....................................................................................... 124
  9.6.1 Message.............................................................................................................. 124
  Marshalling Details....................................................................................................... 125
    Strings........................................................................................................................ 126
    SpaceNames.............................................................................................................. 126
    Locations................................................................................................................... 126
    Locators..................................................................................................................... 126
    Objects....................................................................................................................... 126
  Marshalling the Local References................................................................................... 127

9.7 stub-SCION GENERATOR .......................................................................................... 128
  Import Class Statements............................................................................................... 129
  Class Declaration......................................................................................................... 129
  Writing Stub Methods................................................................................................. 129
  Writing the Scion......................................................................................................... 130
  Issues with the Code Generator.................................................................................. 131

9.8 INVOCATION PROTOCOL.......................................................................................... 131
  9.8.1 Message Formats................................................................................................. 131
    Standard Message Header......................................................................................... 131
    Call and Forward Message....................................................................................... 131
    Reply Message......................................................................................................... 132
    Exception Message................................................................................................... 132
    Live Message............................................................................................................ 132

Sanjay Bhudia
9.8.2 Remote Invocation ................................................................. 133
9.9 COLLECTOR PROTOCOL ......................................................... 135
  9.9.1 Sending live messages ......................................................... 135
  9.9.2 Receiving Live Messages ...................................................... 137
9.10 CYCLIC EXTENSION PROTOCOL ........................................ 138
  9.10.1 Local Trace Sub-Protocol .................................................. 138
    The MarkSet .................................................................................. 139
    MarkDaemon .................................................................................. 140
    Updating Threshold due to Collected Stubs .................................. 140
    Other Components ......................................................................... 141
9.11 REGISTRY AND NAMING SERVICE ....................................... 141
  9.11.1 Bootstrapping ..................................................................... 141
  9.11.2 Service Scions ................................................................. 141
    Creating Service Scions ............................................................. 142
    Getting References to Server Scions ........................................... 143
  9.11.3 Registry Service and Naming Client .................................. 144
    Naming Client .............................................................................. 144
      Binding an Object ..................................................................... 144
      Obtaining a Remote Reference ............................................... 145
      Communication Between Client and Server ............................ 145
    Registry Server .......................................................................... 146
    Possible Improvements ............................................................. 147
CHAPTER 10

SYSTEM ANALYSIS

10.1 Quantitative Tests ................................................................. 150
10.1.1 Types of Tests .................................................................. 150
10.1.2 Testing Environment........................................................... 150
  Operating System ..................................................................... 150
  Virtual Machine ........................................................................ 151
  Hardware .................................................................................. 151
10.1.3 Timing Resolution................................................................. 151
10.1.4 Remote Method Invocation .................................................. 152
10.1.5 Marshalling Overhead for Primitives................................... 153
10.1.6 Overheads of Marshalling References ................................. 153
10.1.7 Overhead of Processing LIVE Messages ......................... 154
10.1.8 Qualitative Analysis of RMI with SSP Chains .................. 156

CHAPTER 11

CONCLUSIONS

11.1 Goals Achieved ................................................................. 158
11.2 SSP Chains in Java Implementation ...................................... 159
11.3 Distributed Garbage Collection.............................................. 159
  11.3.1 Contributions of this Implementation .............................. 160
11.4 Distributed Collectors Survey ............................................... 161
11.5 State of the Implementation .................................................. 161
11.6 Lessons Learnt .................................................................... 162
11.7 Future Work ....................................................................... 163

GLOSSARY .................................................................................. 164

REFERENCES ............................................................................. 166
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Linked list</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Linked list from 169 showing a space leak and a dangling pointer</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Two stacks which have pointers for their elements. One of the elements from each points to a common object in memory.</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Various communication protocols used for remote process interaction</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>A reference R from object A in space X to an object B in space Y.</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Remote reference creation</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Remote Reference Duplication</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Remote Reference Deletion</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Reference counting an its inability to detect cycles in structures. This is a linked list</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Independent Dimensions in the Design Space of a Distributed Garbage Collector</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>x and y form a simple distributed cycle spanning two spaces</td>
<td>51</td>
</tr>
<tr>
<td>7.1</td>
<td>A decrement/increment race condition</td>
<td>57</td>
</tr>
<tr>
<td>7.2</td>
<td>Reference creation with partial weight of 32</td>
<td>59</td>
</tr>
<tr>
<td>7.3</td>
<td>Remote reference duplication. Stub in A initially had a partial weight of 32</td>
<td>59</td>
</tr>
<tr>
<td>7.4</td>
<td>Reference deletion. The total weight has been reduced by 16</td>
<td>60</td>
</tr>
<tr>
<td>7.5</td>
<td>Reference Duplication for the first time in the Indirect Reference Counting Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>7.6</td>
<td>Duplication of a remote reference in the ‘Network Objects’ system</td>
<td>66</td>
</tr>
<tr>
<td>7.7</td>
<td>Migration of elements of a distributed cycle, illustrating Bishop’s Migration Technique</td>
<td>69</td>
</tr>
<tr>
<td>7.8</td>
<td>Bishop’s Migration Technique does not handle indirection well</td>
<td>70</td>
</tr>
<tr>
<td>7.9</td>
<td>An Inconsistent DGC Snapshot leads to Incorrect Collection</td>
<td>73</td>
</tr>
<tr>
<td>7.10</td>
<td>Example of Hughes’ Time-stamping Technique</td>
<td>76</td>
</tr>
<tr>
<td>8.1</td>
<td>SSP Chain extension by exporting references</td>
<td>85</td>
</tr>
<tr>
<td>8.2</td>
<td>Migration of an object causes an SSP chain referring to it to extend at the head end.</td>
<td>86</td>
</tr>
<tr>
<td>8.3</td>
<td>Sub-Protocols of SSP and their Relationships</td>
<td>88</td>
</tr>
<tr>
<td>8.4</td>
<td>Sub-Protocols and Their Interaction via the Shared Data</td>
<td>89</td>
</tr>
<tr>
<td>8.5</td>
<td>Invoking a method via an indirect chain without short-cutting results in a new chain</td>
<td>92</td>
</tr>
<tr>
<td>8.6</td>
<td>Short-cutting the strong SSP chain on method invocation</td>
<td>94</td>
</tr>
<tr>
<td>8.7</td>
<td>Method invocation on an indirect weak reference causes a location exception (same key as Figure 8.6)</td>
<td>95</td>
</tr>
<tr>
<td>9.1</td>
<td>UML Class Diagram: ssp.system.SpaceName</td>
<td>113</td>
</tr>
<tr>
<td>9.2</td>
<td>UML Class Diagram: ssp.system.Location</td>
<td>114</td>
</tr>
</tbody>
</table>
Figure 9.3 UML Class Diagram: ssp.system.Locator, showing containment of two instances of Location

Figure 9.4 UML Class Diagram: ssp.system.Stub showing a subclass TestImpl_Stub

Figure 9.5 UML Class Diagram: ssp.system.Scion, showing a sub class Test2Impl_Scion and the Class cache used by the factory functions

Figure 9.6 All the subclasses to java.util.Hashtable. These are singleton system data structures

Figure 9.7 UML Class Diagram: ssp.system.EndPoint encapsulates a TCP Socket

Figure 9.8 UML Class Diagram: ssp.system.ServerEndPoint

Figure 9.9 UML Sequence Diagram: shows receipt of first message from new space

Figure 9.10 UML Sequence Diagram: Receipt of Message by Message Daemon Thread

Figure 9.11 Delivery of Message by MessageDispatch thread. SSP.transportReceive handles different message types in different manners

Figure 9.12 Message construction if received from an input stream

Figure 9.13 UML Class Diagram: ssp.system.Message, showing containment of other elements

Figure 9.14 Marshalling an instance of java.lang.String

Figure 9.15 Marshalling an instance of ssp.system.SpaceName

Figure 9.16 Pseudo-code for marshalling Local References in a method call message

Figure 9.17 Pseudo-code for converting a local reference into an instance of Locator, i.e. remote reference

Figure 9.18 A stub method for a given method in the class. The original method has the same signature

Figure 9.19 Remote Method Invocation by a Client of a Stub. Output streams omitted and message sent

Figure 9.20 Receipt of a CALL message, local method invocation and returning results to sender

Figure 9.21 The SSP.sendLiveMessage() method call sending a live message to all connected spaces

Figure 9.22 The Mark Set Classes which store references to all variables in current scope

Figure 9.23 SSP.localService(): does installation of a Service Scion for a server object

Figure 9.24 UML Class Diagram: ssp.Naming, the registry client class available to each space

Figure 9.25 Classes that compose the Registry Server

Figure 10.1 Remote Method Invocation without any arguments

Figure 10.2 Marshalling overheads for Primitives

Figure 10.3 Overhead of Marshalling References

Figure 10.5 Overhead of Processing LIVE messages between two spaces

Figure 10.6 Overheads for sending LIVE messages between 2-8 spaces with 40 objects in each space
Table of Equations

Equation 4.1 Definition of Liveness.................................................................27
Equation 7.1 Weighted Reference Counting Invariant.................................59
Equation 7.2 Optimised Weighted Reference Counting Invariant................61

Table of Tables

Table 9.1 The Standard Message Header Format...........................................131
Table 9.2 Format of the Call and Forward Message Types............................131
Table 9.3 The Reply Message Format..............................................................132
Table 9.4 The Exception Message Format.......................................................132
Table 9.5 The LIVE Message Format.............................................................132
Table 9.6 Format of the Registry bind message.............................................145
Table 9.7 Format of the Registry lookup message........................................145
Table of Listings

Listing 8.1 Detection Server. The message is treated only if garbage collection date is the latest date received from the space................................................................. 103
Listing 8.2 Local Garbage Collection................................................................. 104
Listing 8.3 Pseudo-code for the STUBDATES handler............................... 105
Listing 8.4 Pseudo-code for the ACK message handler............................. 105
Listing 8.5 Pseudo-code for the THRESHOLD message handler............. 106
Listing 9.1 Distributed Cleanup Protocol thread....................................... 135
Listing 9.2 SSP.sendLiveMessage(): sends a LIVE message to every connected space ................................................................. 136
Listing 9.3 SSP.receiveLiveMessage(): receives a LIVE message from another space.................. 138
Listing 9.4 SSP.localService() and the delegate method Scion.forRemoteObject()................................. 143
The project detailed in this thesis is part of the requirements of the Master of Engineering in Software Engineering at the Department of Computing, Imperial College. This document is the final document submitted, following an earlier document, the "Project Outsourcing Report", which was submitted in February, 1999.

1.1 Overview

The project is concerned with the design and implementation of a distributed garbage collector. Garbage collection is a term used to refer to automatic memory management. This is in contrast to explicit (manual) heap management, which gives the responsibility of de-allocating memory occupied by objects to the application programmer. As modern software development involves large, complex artefacts of software, often developed by a large team of developers, no one person knows the implementation details of the entire system. It is thus very difficult for any developer to manage memory manually. Manual allocation and de-allocation is prone to errors that cause difficult-to-debug problems. This issue is made more prominent in distributed systems where there are many of threads of control, many processes interacting and often components developed by many different parties. Explicit management is out of the question in such situations.

If there is such a strong case for automatic memory management, why do we not see it integrated into systems today? There are several reasons. First, designing and implementing distributed collectors is difficult because of problems in efficiently obtaining a consistent snapshot of the overall system. Current implementations make compromises in the types of garbage collected, or impose a significant overhead on resources to be used. Second, most people practising system software development were introduced to computers at a time when memory and CPU resources were very scarce and thus minimum resource utilisation at the expense of development times was the order of the day. Systems have changed a great deal since then, but the developers' efforts still seem to be focused on those issues. Both memory and CPU power is very cheap today. Thus automatic management of the heap, which ensures less development times due to a reduced number of memory problems, is desirable.

Another issue in developing distributed collectors is that most of the requirements are difficult to meet in one design. It is desirable to have a collector that imposes minimum overhead on the system. Also, for large systems, it is desirable to have a scalable collector that does not consume resources as fast as system size increases. In today's world of heterogeneous systems, a collector that works with different nodes which have objects and processes compiled in different languages is also essential. The research conducted has not found an implementation that addresses all the issues in one system.
1.2 Contribution

This thesis presents the design and implementation of a distributed garbage collector and a distributed object-oriented system. The distributed object-oriented system provides standard services expected of such systems, namely: location transparency, argument marshalling and registry services. Its design is closely coupled with that of the cyclic garbage collector, due to [Shapiro et al., 1992]. The design and implementation of an acyclic collector, due to [Le Fessant et al., 1997] is also presented as an add-on to the cyclic collector.

To my knowledge, the cyclic collector has not been implemented for a common object-oriented language before this attempt, and the algorithm presented in [Le Fessant, 1997] is in very abstract terms, thus the mapping to an implementation is non-trivial. The paper assumes language features such as pointer redirection by the referred object, which do not exist in common languages such as Java and C++.

Several design issues not documented in [Shapiro et al., 1992] also had to be considered when implementing the distributed system and the cyclic collector. Specifically issues regarding the implementation of strong-chain short-cutting (section 8.8.2) and of weak chain-short-cutting (section 8.8.3) required some original design thought. These were issues such as efficient forwarding by eliminating re-marshalling at every node and additional fields in stubs and scions necessary for forwarding messages when redirecting. Another issue worth mention is the need for weak pointers in the tables that constitute the shared data structures between the object management and distributed clean up protocol (section 9.4.6).

Performance comparisons with Java's RMI show that the implementation is efficient, and in some measures exceeds the performance of RMI, though this is due to design decisions rather than implementation optimisations (Chapter 10). A feature comparison between RMI and the implementation documented here shows that the Stub-Scion-Pair Chains is more efficient in reference duplication as it requires only one message instead of RMI's three messages, it has a collector which is tolerant to process failure while RMI does not have such a collector. Also, with the extension, the key difference is that the RMI collector cannot collect distributed cycles of garbage.

A comprehensive survey of the current garbage collection algorithms is also another contribution that has been achieved (Chapter 7).

Chapter 6 describes the design choices available in the design of a distributed collector as points in a multi-dimensional space. This is a unique way in which algorithms can be plotted as points in this space and helps identify the factors that affect the design choices, such as network characteristics and the need for scalability.

To summarize, the key differences between the collector algorithm implemented in this project and those in other systems are that reference duplication is cheap, the algorithm is scalable because no global information exchange is required. The algorithm is tolerant to message and process failures. The key distinguishing feature of the acyclic collector is that no termination protocol is required as it allows safe inconsistencies by making conservative estimates and allowing floating cycles.
1.3 Structure of this Thesis

The next chapter gives an overview of memory allocation and its history. The problems with explicit heap management are presented. Garbage collection is presented as a viable alternative to the difficult-to-debug problems caused by programmer directed heap management.

Next, a brief introduction to distributed systems is given. The history of mechanisms of inter-process interactions is given. The Network Objects model is discussed, with details on operations on remote references. I discuss this model because it presents the mechanisms which are present in all distributed object oriented systems today, including OMG's CORBA, Java's RMI and Microsoft's DCOM.

The fourth chapter gives the classical garbage collection algorithms. Although the algorithms are used in single processor collection, current distributed collector designs are based on these algorithms. Most of the distributed garbage collection literature I have read assumes knowledge of the common uni-processor algorithms, hence the need for this chapter.

The fifth chapter presents the requirements given in the outline and adds some that should be met by all distributed collectors. Conflicts in requirements are discussed in this section.

The sixth chapter gives the design space of distributed collector algorithms. The various assumptions that have to be made regarding network and space failures, the scale of the system and so on are presented in this section. Also, details on the impact of such decisions is given.

Having set the context by detailing the requirements of the project and the possible design choices that can be made, the seventh chapter presents the current work in distributed garbage collection algorithms. I present each algorithm and then attempt to give its shortcoming. Also, I mention which requirements of the project it may address and which ones it may fail to satisfy.

The survey of collector techniques in the sixth chapter leads on to a choice made of which algorithm has actually been implemented. The design details of this algorithm are given, together with the assumptions that the design makes on other parts that it will integrate with, such as the network, the local garbage collectors, and the applications which will use it.

Having given the design details, this chapter gives details about the implementation of the system in Java. Class diagrams are given in UML 1.1 and some of the pseudo-code of the main protocols is also given. The various components of the system detailed here include the Trasport protocol, remote method invocation protocol, the presentation protocol and the collector protocol. Also, a source code generator tool and a naming registry have been implemented to support the system development and use.

Performance tests to various components of the system, including to the distributed collector, are given, and an attempt is made at making inferences from the results.

Finally, I conclude by giving details of the goals that the project had at outset, and the deliverables that met those goals. Also, I give an account of the lessons learnt during the course of this project.
Although I have made an attempt to be as concise as possible, being comprehensive conflicts with that goal and so the thesis maybe wordy in places. I apologise for that, and hope that you do enjoy reading my thesis.
In this section, I will attempt to give a context to my project. I initially give an overview of the evolution of memory allocation techniques in programming languages. Then I present some problems with explicit memory allocation and show that there is a strong case for Garbage Collection. I give an overview of the classical garbage collection algorithms.

### 2.1 History of Memory Allocation

Programming languages have evolved by giving the programmer increasing freedom in the amount of memory accessed and the patterns of access. Each newer generation of languages have provided greater support for abstraction and the automation of actions that were previously manual or explicit.

In the very early days, all communication between programmer and machine was on a bit-by-bit basis, using switches. With the introduction of simple I/O devices, hexadecimal values were exchanged. Next came mnemonic codes, followed by assembly languages. This lifted the burden of finding absolute address values from the programmer. The exacting art of assembly programming become less important for developers with the introduction of compilers. A compiler must allocate resources for a target machine to represent the data objects manipulated by the user's view of the program.

The amount of memory accessible to developers has increased rapidly, coupled with lowered costs of memory. But its exploitation by developers has been slow. One of the main reasons for this conservatism is that constraints on memory management are built into the fundamental semantics of popular computer languages. Thus, the adoption of a new generation of languages is required.

Another reason is the difference in the concept of memory used by software developer and the concept used by hardware engineers. There is a big difference between the dynamic random-access memory chip (DRAM) provided by hardware engineers and the dynamic object-based graph structure desired by the software engineer [Baker ed., 1995]. This gulf must be bridged by memory management software.

The next sections give an overview of the three ways in which memory storage can be allocated.

#### 2.1.1 Static Allocation

This is the simplest storage allocation policy. All names for variables in the user's program are bound to storage locations (addresses in memory) at compile time.
time. These bindings do not change at run-time. The local variables of a procedure are bound to the same memory locations at every activation of the procedure. The original implementation of Fortran, the current Fortran77 and the parallel language Occam have this policy. Static allocation has these limitations:

- The size of each data structure must be known at compile time.
- No procedure can be recursive since all its activations share the same memory locations in which to store values bound to variables.
- Data structures cannot be created dynamically.

Despite the above limitations, there are important benefits that make it suitable for some specialist areas. These are:

- Speed increase because there is no need to create data structures during program execution. These include structures needed by other languages such as stack frames. Also, since the location of all data is known by the compiler, there is no de-referencing code and storage locations can be accessed directly.
- Safety guarantee: A program cannot fail by running out of space at run-time because its memory requirements are known in advance [Baker ed., 1995].

Because of this, simulations where parallelism is required usually have Fortran77 for implementation.

### 2.1.2 Stack Allocation

The first block-structured languages overcame the constraints of static allocation by allocating storage on a stack. An activation record or frame is pushed onto the system stack as each procedure is called, and popped when the procedure returns. The stack organisation has the following implications:

- Different invocations of activations of a procedure do not share the same storage location or binding for local variables. This makes recursive calls possible. Also, when more than one process is executing, (each with its own stack) the processes can call a procedure at the same time. Recursion is possible, which greatly increases the language’s expressive capabilities.
- The size of local data structures such as arrays may depend on a parameter passed to the procedure.
- A called activation cannot outlive its caller.
- Only an object whose size is known at compile time can be returned as the result of a procedure.

Examples of such languages are Algol-58 and Atlas Autocode [Jones and Lins, 1996].

### 2.1.3 Heap Allocation

Data structures may be allocated and de-allocated in any order and not restricted to the last-in, first-out discipline of a stack. This means that data structures may outlive the procedure that created them. Heap allocation has the following implications:
Software developers usually have to express real world problems by creating abstractions. Most of the data structures that result are naturally hierarchical, the most common examples being trees and lists. Heap allocation allows the concrete implementation of such abstractions to be recursive.

Objects whose size and structure was not known at compile time can be returned as results to procedures.

Many programming languages allow a procedure to be returned as the result of another procedure. This is used in higher order functional languages (such as Miranda and Haskell).

Because the size of data structures is not known at compile-time but can vary dynamically, exceeding system memory limits is one of the most common sources of program failure.

Allocation of storage space in the heap can be explicit or managed. Languages such as C and Pascal allow programmers to allocate storage on both the stack and the heap. The programmer has to explicitly manage the heap. C++ [Stroustrup, 1991] is one recent language that has explicit storage allocation.

### 2.1.4 Problems of Explicit Heap Allocation

Most imperative languages have traditionally placed the responsibility for the management of the heap with the programmer. For example, in C, the programmer can request for a portion of memory by asking the runtime system for memory using the system call `malloc()`. The programmer then has the responsibility of returning the memory for re-use to the run-time system by using `free()`.

```c
struct cell{
    int element;
    cell* next;
}

cell* insert(int item, cell* list){
    cell* temp = malloc(sizeof(cell));
    temp.element = item;
    temp.next = list;
    return temp;
}

int main(){
    cell* myList;
    myList = insert(1, insert(2, insert(3, NULL)));
}
```

Figure 2.1 Linked list

The example in 3 is presented in pseudo-C, but may refer to any language that has explicit heap management. A list element is declared (as a structure called `cell`). The `insert` procedure returns an instance of `cell` populated by data given in its arguments. The second argument is a pointer to a list, thus the returned pointer will be to a list in which the newly created cell is pre-pended to the given list. The `main` function recursively builds a list of 3 elements, as shown in Figure 2.2 (a).
Garbage

If memory is allocated dynamically, it may become unreachable. This happens when there are no references to a particular chunk of memory. When pieces of memory are not reachable, but are not free either, they are called garbage. The following line, when added to the `main` function in 3 after the list is created, will cause a space leak. This happens when garbage cannot be re-used.

```
MyList.next := NULL;
```

This causes only the first cell in the list to be reachable. The cells that contain the values 2 and 3 are not reachable. The memory occupied by them will not be re-used by the program. This is seen in Figure 2.2 (b).

```
MyList.next := NULL;
```

![Figure 2.2 Linked list from 173 showing a space leak and a dangling pointer](image)

Dangling References

The previous section showed how a chunk of memory could be unreachable. Memory can also be de-allocated while there are references to it. Looking again at 3, if we add a line after the list is created which says:

```
free(MyList.next);
```

This is a system call that returns the memory occupied by the second cell to the heap manager. Now, the `next` field of the first cell refers to a piece of memory which has been de-allocated. The heap manager may re-use this piece of memory for another structure or it may not use it for some time. There is no way for the program to know how and when the piece of memory is going to be re-used. The heap manager may allocate heap space to other processes. In such cases, programs with this error exhibit non-deterministic behaviour. This is because for some executions, referencing the dangling pointer may result in the original data being retrieved (as it has not been re-allocated), but this may not happen on other executions. Such errors are development nightmares because they are very difficult to trace or debug.
Ownership of Memory

Object orientation makes explicit heap management more difficult. This is because the primary reason for using the object model is to allow for the controlled sharing of information among objects (via encapsulation). But this sharing however, makes the idea of "ownership" of objects difficult to resolve. Each object will not have enough information about the entire reference graph to determine whether an object that it refers to can be de-allocated after it has no need for it. This is possible because an object can have more than one reference to it, but each of the referents do not know about the others.

For example, if a developer writes code that allocates memory to store account details of a customer in a system. The developer also writes code that populates this memory structure and writes it to file. There exists code which allows other threads of control to obtain a reference to this memory location. Now, does the developer write code to de-allocate at the end of the file write? If he does so, then there may be other parts of the system that will refer to that piece of memory.

Another example is seen in Figure 2.3. from [Boehm and Chase, 1992]. This may occur when a general implementation of a stack is required, such as one which has pointers to a base class and thus can be populated with instances of sub-classes (java.util.Vector is an example, which has elements as references to java.lang.Object). With explicit heap management, code in Stack A has to check for other references to Data Object before deallocating the object or never de-allocate it.

Checking for references by other objects can be done. One way is for every client of the object (a client of an object is an object that holds a reference to that object), to inform the owner, which then keeps a count. When the count reaches zero, the object's memory may be de-allocated. This is essentially a reference counting collector. The argument is that that kind of development overhead is too much and in practise is never done consistently.

Also, the need for checking for references violates one of the principles of object-oriented development. This is that there should be minimal communication between objects and the definition of clean interfaces between objects. This allows the classes to be re-used in different contexts and configurations. Now, if

---

Sanjay Bhudia
they are to be re-used in different contexts, then the class code should not have to worry about de-allocation in the different contexts, otherwise the re-usability is compromised.

2.1.5 A Case for Garbage Collection

As seen in the Section (2.1.4) above, dynamically allocated memory in complex programs is hard to manage correctly with explicit allocation and de-allocation.

The errors that explicit heap management introduces are difficult to debug because they are difficult to replay. On a modern workstation or PC, the main memory at any given time may contain more than one executing process. Processes are not loaded in the same memory addresses each time. Also, some languages ask an operating system to manage the heap for them (for example, the heap management in C is done by the underlying OS). A dangling reference in such cases may not appear as a bug because the memory allocator might not allocate the space to another structure on behalf of another process immediately. Thus the programs exhibit non-deterministic behaviour. This was mentioned above with reference to Figure 2.3, showing the stack.

A considerable proportion of development time may be spent tracking down such memory management bugs. [Rovner, 1995] estimates that forty percent of the time developing a certain system (at Xerox PARC) was spent on memory management.

A further indication of the problem of heap management is the range of debugging tools available to ensure correct use of the heap. The best known example of such a tool, which is used to debug space leaks, is Purify [Purify, 1992]

As mentioned in section 2.1.3 above, languages that allow programmers to use the heap for data structures can be placed in two categories with regards to memory management. The language implementation either places the burden of the heap management on the programmer, or manages it automatically. Examples of the former are C [Kernighan and Ritchie, 1978], C++ [Stroustrup, 1991], Modula-2 and Pascal.

The first language to use a garbage collector was Lisp [Jones and Lins, 1996]. In fact, it has been cited that "one of LISP's most lasting contributions is it's garbage collection" [Sammet 1969]. Traditionally, garbage-collecting techniques appeared in functional and logic languages, but gradually, more imperative object-oriented languages are using GC in their implementations. Most modern object-oriented languages such as Smalltalk, Eiffel and Java have garbage collectors. C++ is the one recent language that remains committed to explicit heap management. Even for this language there exists an efficient garbage collector [Boehm and Weiser, 1988] that is implemented by overloading the allocation and de-allocation methods (new and delete), and thus is not part of the language semantics.

The next chapter introduces distributed systems and distributed object-oriented systems. Classical Garbage collection algorithms are presented in Chapter 4.
I give a brief overview of distributed systems and very briefly mention some of the paradigms and models that exist for distributed object-oriented systems. I give an introduction to the Network Objects paradigm. The terminology used in my presentation of this model is rather specific to a later system by Shapiro et al., 1992, but the idea is one on which most of today's distributed systems are based on. Another reason is that the design allows heterogeneous distributed systems to be built as well. This is in accordance with one of the project requirements, which is for the garbage collector to work with heterogeneous nodes which comply with the CORBA standard.

I also introduce the Actor paradigm. This is a suggested extension for the project is to integrate the implementation with a language based on the paradigm.

3.1 Introduction

In recent years, the dramatic fall in the cost of computer hardware combined with advances in network communications have lead to the increase of networked computers. Interconnected computer systems permit interaction, co-operation and sharing of information and facilities.

But what exactly are distributed systems? [Sloman and Kramer, 1987] have the following definition:

"A distributed processing system is one in which several autonomous processors and data stores supporting processes and/or databases interact in order to co-operate and achieve a common goal. The processes co-ordinate their activities and exchange information ... over a communications network."

[Enslow, 1978] has a more demanding definition which requires that the control, hardware and database have the required degree of decentralisation in order to be classified as distributed.

Distributed systems can be considered to be a special case of a computer network, one in which a higher-level of communication protocol gives a high degree of transparency. [Tanenbaum, 1988] states:

"The key distinction is that in a distributed system, the existence of multiple autonomous computers is transparent (i.e. not visible) to the user. He can type a command to run a program, and it runs. It is up to the operating system to select the best processor, find and transport all the input files to that processor, and put the results in the appropriate place."

Technological changes favour low-performance, low cost processors rather than single high-performance processors. Also, the growth in computing has led to the
demand for more sophisticated facilities. Some of the desired properties of distributed systems are:

- Distributed systems lead to modularity and simpler software because each component must provide a well-defined service. Verification and service provision rules can be specified at the component interaction point.

- The system can be extensible, flexible and scalable. Increase in service demand can be met by providing more service elements to serve the demand.

- Availability is increased because the failure of one node does not result in the failure of the entire system. Similarly, a crash does not bring down the entire system. Replicated service provision can mean that there is more than one node capable of providing the service.

The architecture and technology to develop distributed software applications has been in place for a number of years. There exist a number of distributed functional languages, and a great number of implementations that allow distributed system development over languages that do not support distribution in their semantics.

There are thousands of currently deployed distributed applications, which are critical to the running of various commercial organisations, from entire banking systems, load balancing in power grids, international stock trade to military information systems. If one stops to think for a moment how much we rely on distributed systems, then one can appreciate that the development of technology which makes these systems more robust is absolutely essential.

Most of the currently deployed systems use distribution models which have no semantics of remote references and ownership of objects. This is because they are based on procedural code and thus not object-oriented in nature. As the distributed systems get larger and more complicated to implement and maintain, the advantages offered by object-oriented development will be apparent and considered essential. The object paradigm then exposes the remote references and object lifecycle problems which distributed garbage collectors address. This project focuses on distributed systems supporting object orientation and distributed garbage Explicit Distributed Memory Management?

As discussed in section 2.1.4, object-orientation warrants sharing of objects by other objects through references. Also, each object does not know enough about the reference graph to be able to decide whether it should explicitly de-allocate storage for the object it was referring to. There may or may not be other objects referring to that object. This case is even further stressed in distributed applications, in which components interact by cleanly defined interfaces and well known service nodes. The modularity of software design offered by distributed systems would severely be compromised if the objects were supposed to explicitly manage memory.

Thus, almost all distributed systems have automatic memory management. Most systems are built on top of local nodes which have local collectors.

Another barrier to explicit management would be the very concurrent nature of a distributed system. Since each node has one or more processes, there are very many mutators constantly changing the reference graph of objects in memory. For an object to be able to explicitly collect another object’s memory, it would have to know that the particular object was not going to be referred to by all the processes in the system.
3.2 History of Distributed Process Interaction Mechanisms

In this section, I introduce the idea of distributed object-oriented programming and where some of the concepts originated from.

The distributed systems and applications that are replacing currently deployed applications are all based on CORBA, RMI or Microsoft’s DCOM. All these protocols are examples of Network Objects, but non address the issue of remote reference management and object lifecycles fully.

Currently in use for most interprocess communication in current distributed OS’s and deployed applications. It is used in conjunction with languages that allow extensive heap usage, but not suited to that use.

UDP is also used as a bootstrap protocol in existing distributed OS’ and for session-less interactions amongst processes. TCP is a very commonly used session based protocol, currently used in HTTP, file services etc.

Popular since the 1970’s. Can be seen in a number of existing distributed operating systems as a lightweight bootstrap protocol. Also used by higher level protocols such as TCP / UDP

LANs and other proprietary network protocols were used in early systems when de facto standards such as TCP/IP were not around. Seen in legacy systems and different a different solution from every vendor.

Figure 3.1 Various communication protocols used for remote process interaction

Figure 3.1 shows various communication protocols used for remote process interaction. Around 20 years ago, most software development companies were also large hardware vendors (IBM, DEC, HP) and thus the solutions they provided usually ran on their own custom networks. Later, the use of IP was made, but in some situations TCP was considered heavy and so the application developers had to implement their own message mechanisms and semantics. This resulted in the message passing mechanisms between processes being developed in an ad-hoc manner and subsequent development efforts leading to incompatible systems.

Networking protocols, such as TCP/IP provide a way for process nodes to communicate in an OS independent way. But programming over network sockets is not very easy. This is because different data types will have different binary representations on different platforms, even for the same languages. The problem of system components interacting when they are written in different languages with different primitive types is even bigger.

Remote Procedure Calls (RPC) [Birrell and Nelson, 1984] are an abstraction over network socket programming. They extend the semantics of local procedure calls over networks, hiding the marshalling and de-marshalling (serialising parameters to binary over-the-wire compatible representations) from the application programmer. As mentioned earlier, the vast majority of distributed systems in use today have been implemented over some form of RPC.
mechanism. This mechanism allows remote procedure calls to have the same semantics as local procedure calls by the use of stubs that marshal method invocations and server-stubs that handle the messages, but have no concept of remote references and objects. One may be lead to think that they do not suffer from distributed memory management problems. In fact, extensive use of heap allocated structures in remote calls is ill-suited to RPC as it may lead to referencing de-allocated space, or serialisation of large data structures, both of which are undesirable.

3.3 Distributed Object Systems

Distributed Object-oriented systems fall in two broad categories, which I introduce in this section briefly.

3.3.1 Distributed Shared Memory

These are usually found in systems where the nodes of the system are tightly coupled. They have communication channels which have very high bandwidth, low latency and guaranteed response times. These channels are usually implemented in hardware and integrated with the system so that processors have instructions to access them. These systems allow two or more processes to share part of their address space. If the address sharing is not the same for the owning processor than another, then it is referred to as Non Uniform Memory Architecture (NUMA). If all processors can access the memory in the same manner, then the architecture is referred to as Unified Memory Architecture (UMA).

Objects can be accessed directly using normal memory operations, and any writes or updates to an object residing on a shared part of the memory will have updates reflected in all images of the memory. The system is responsible for providing the appropriate consistency and coherency protocols to ensure the correctness of the shared memory data. Note that these protocols are in addition to the synchronisation required when accessing any shared resource.

Such systems do not necessarily have to involve objects as memory elements, but enforcing the abstraction allows additional mechanisms such as persistence and garbage collection.

I shall not consider such systems further in this report as the project concentrates on loosely coupled distributed systems with no shared memory and interaction only occurs via message passing.

3.3.2 Remote References Systems

When people talk about distributed systems, they often mean systems which fall into this category. In such systems, a process keeps a pointer to another object that exists in a different address space (a process or node). Methods invoked on the object are sent space in which the object exists. They are interpreted and a method call invoked onto the object before sending the result back to the object. Access to object state must always be made via methods because the object may not be local. In languages which allow the state of the object to be accessed directly, this requirement is imposed as an addition to the normal language semantics. Examples of such object oriented languages are Java and C++.
In contrast to shared memory systems, such systems only ever have one copy of the object in the entire system. The shared memory problems of consistency and coherence of an object copy therefore do not arise.

A remote reference can take several forms. Most often it will be in the form of a surrogate object. These details are discussed in more detail below.

### 3.4 Network Objects Paradigm

The Network Objects paradigm [Birrell et al., 1994], is the concept on which most of today's distributed object-oriented systems are based. It is based on the remote procedure call mechanism (RPC), but supports distributed object semantics. The following functionality was introduced by the Network Objects paradigm:

- **Transparent remote invocation.** Interaction between distributed objects is provided by surrogate objects. An object can make a method invocation on a remote object transparently [Almes et al., 1985]. This means that the call can be invoked in a similar manner as a local call. In reality, it calls a surrogate object, which in-turn performs the method call on the remote object on behalf of the local object. In the example implementation discussed by [Birrell et al., 1994], the surrogate managed the marshalling and de-marshalling of parameters. This is similar to Java's RMI and the CORBA ORB.

- **Marshalling between different platforms.** [Birrell et al. 1994] mentions work on conversion of the binary representations of data-types on different implementations of Modula-3. This seems to me to be the very early work on interoperable marshalling is the key mechanism in CORBA.

- **Passing remote objects by reference.** When a method invocation has a reference to an object, the reference will not be valid because the address it refers to is not accessible to the object receiving the invocation. This is solved by the surrogate objects mentioned above. Specifically, a new surrogate object of the message receiving or server-stub kind, is created for each argument of the method that is a reference. On receipt of the message carrying the method invocation, the opposite stub, (again a server-stub) will check to see the existence of client-stubs to match those created on the invoking space. If some or all of them don’t exist, then they will be created.

- **Third party transfers.** An object A running on process P1 refers to an object B running on P2. When A invokes a method on an object C running on P3, it may pass the reference to B without B being aware of it, or A having to invoke a separate method from a local invocation.

These properties introduced in [Birrell et al., 1994] are today found in many systems. [Shapiro et al., 1992] and [Plainfossé and Shapiro, 1994] discuss a very similar idea for distributed object references, called stub-scion pairs. I will use the terminology from their work to present concepts in distributed references in the next section.

### 3.5 Remote Reference Model

In this section, an abstract distributed references model is presented to help understand operations on remote references. The terminology is derived from [Shapiro et al., 1992]. Note that the same operations on references occur in any
3.5.1 Spaces

A distributed system is partitioned into disjoint spaces. This is used to abstract the idea away from any implementation. At the lowest level, a space may be the words in memory with the scope of names being addresses. At the next level up, each processor and its memory is a space, with the scope is the local network, and at the top level, each local network is a space and the scope is the internet.

Spaces can only interact with each other by message passing. Two different spaces do not share the same address space. Thus the model is used for distributed systems that are loosely coupled, like a network of workstations.

The spaces use message passing over potentially unreliable communication channels. This may cause messages to fail: loss, duplication, delay or out-of-order delivery. Although the outline of the project mentions that reliable communication may be assumed, (see Section 5.1, requirement R 8), reliable communication transports such as TCP/IP may fail when nodes in the network fail. Failure of some messages should not lead to total failure of the distributed system.

Spaces may fail due to software or hardware failure. Again, failure of one node in a network should not mean the failure of the total system, if it can be helped. The model considers fail-stop semantics: failed spaces do not send messages.

Spaces may also disconnect due to network failures or reboots etc. A disconnected space cannot modify the distributed reference graph, therefore a disconnection is safe. A disconnected space may reconnect, or terminate.

Each space is a process that has its own memory, its own local root set, its own local garbage collector.

3.5.2 Stubs and Scions

Distributed computation is effected by sending messages that invoke methods or procedures on remote objects. These remote method invocations, have the same components as a local method invocation or procedure call. They specify:

- A distinguished object that is to perform the call, or in object-oriented terminology, an object that will receive the method invocation.
- Zero or more arguments of arbitrary types. These types may be primitive types or references to object types.
- An optional result of arbitrary type.

The reference type mentioned above is of interest in this section. The remote method invocations may result in objects exchanging references either as
arguments or results. Consequently, an object involved in several remote
invocations may be referenced from a number of remote spaces. Such remote
references are created when the reference to an object crosses the space
boundaries.

![Figure 3.2 A reference R from object A in space X to an object B in space Y.](image)

As can be seen from Figure 3.2, a remote reference consists of a local pointer to a
stub, and a corresponding scion with a local pointer to the referenced object. The
stub contains mechanisms to remotely locate the referenced object (B) from its
space (Y). The object referred to is a public object.

The term scion is not used by all distributed systems literature. RMI and
CORBA literature use the term skeleton, [Birrell et al., 1994] use surrogate object
and some other systems use server stub or server-side stub. They all refer to the
same object which accepts remote marshalled messages from the stub and
performs a local method invocation on behalf of a remote client.

The space that contains a public object is called its owner. Other spaces whose
objects refer to the public object are the clients. The roles of client and owner are
specific to the public object. An owner of one object may be a client of another.

### 3.6 Operations on References

This section describes the different operations on references and how they may
propagate to various spaces in the system.

#### 3.6.1 Reference Creation

An owner space sends a message to a receiver space that contains a reference to
an object. This reference may be an argument if the message is a remote method
invocation. It may also specify the object to invoke the method on. If it is a
message conveying the result of a method invocation, then it may be a result
reference.

Upon receiving the message, the receiver space becomes a client of the remote
public object. If the object was local, it becomes public. This may happen if this
was the first remote reference to that object. A reference creation may also
involve an already public object.

Figure 3.3 illustrates creation of a reference to a local object v, owned by space B.
When sending a message that contains the reference, space B creates a local
scion b to prevent v from being reclaimed. Upon receipt of the message, space A
installs a stub v, initialised with the locator found in the message.
3.6.2 Reference Duplication

A sender space, which is a client of a remote public object, sends a message to a receiver space. The message contains a reference to the remote public object (owned by a third space). The receiver space becomes a new client of the remote public object when it has processed the message. Unlike reference creation (section 3.6.1), the owner space is not involved in the reference duplication and not aware of it.

Figure 3.4 shows an example of reference duplication. The reference of a public object \( v \), owned by space \( B \) is duplicated between spaces \( A \) (existing client) and \( C \) (new client). The receiver space \( C \) installs a stub \( v_c \) with the locator to scion \( b \) found in the mutator message.
(i) object $x$ in space $A$ duplicates reference $\nu$ to space $C$.

(i) C receives reference $\nu$ and installs stub $v_C$

**Figure 3.4 Remote Reference Duplication**
3.6.3 Reference Deletion

Locally, a client space deletes a reference it owns. This is usually done when a local object which holds the remote reference via a stub, is deleted. Because the local object pointed to the stub, the stub is also collected. This role of the distributed garbage collector: to occasionally inform the owner of the space which public objects are no longer being remotely referenced.

![Diagram showing reference deletion](image)

(i) $C$ deletes $v_C$. This causes the finaliser in $v_C$ to run, which causes a delete message to be sent to $B$

(ii) $B$ reclaims the scion $b$ on receipt of the control message. The local garbage collector in $B$ reclaims the object $v$.

Figure 3.5 Remote Reference Deletion

Figure 3.5 illustrates the deletion of the reference in $C$ to public object $v$ located on space $B$. First, space $C$ deletes its local stub $v_c$ and afterwards sends a delete control message to the owner space $B$. Upon processing that delete message, $B$ reclaims the garbage scion $b$. When space $B$ triggers a local garbage collection and collects object $v$.

3.6.4 Differences Amongst Systems

The reference model introduced in the previous section does not correspond to any actual implementation. It is an abstraction to attempt to highlight just the operations on remote references without introducing any particulars which bind it to a particular implementation.
One difference found in systems is the cross-space referencing scheme. A system may use globally unique identifiers for objects. An example of such a system is CORBA. On the other hand, a system may use forwarding chains, which are essentially like chains of pointers. This is detailed in 6.2.6.

Another notable difference is the number of scions that an object may have. An object maybe referred to by more than one client in more than one space. Some systems have many client stubs sharing one scion, while others may have a scion for each stub. An example of the latter is [Birrell et al., 1994] “Network Objects” while the former case can be seen in [Shapiro et al., 1992] “SSP Chains”.

3.7 The Actor Paradigm

In this section, I briefly introduce the Actor paradigm. This is because on of the project proposals is to integrate the resulting implementation of a distributed garbage collector to TECCware [TECCware], which is an actor based language adapted from Rosette. Rosette was developed by [RWI], part of the [MCC] Consortium.

The actor paradigm was introduced by Carl Hewitt [Hewitt, 1977], and later elaborated by Gul Agha, in his PhD thesis [Agha, 1986].

The concept of an actor has the following notions:

- a process (thread of control),
- memory, consisting of encapsulated variables, and
- communication, achieved by message passing.

All actors are currently active objects, there are no passive entities. Each actor has a mail queue with a unique identifier. Actors interact by mailing messages to other actors.

Root actors are those that are always running, and can interact with the external world via I/O devices etc. An actor A is an acquaintance of an actor B if A’s mail queue address is known to B.

An actor is garbage if it is neither active nor can become active hereafter, or if it cannot send information to, or receive information from a root [Kafura et al., 1990].

Computation in a system of actors is carried out in response to communications sent to the system. These communications are in the form of tasks. As the computation proceeds, the system evolves by creating new tasks and actors. All tasks that have already been processed, and all actors that are no longer "useful", may be removed or garbage collected.

In simplified terms, the unprocessed tasks in a system of actors are the driving force behind computation in the system. A task is a 3-tuple of:

- A tag which distinguishes it from all other tasks in the system.
- A target which is the mail address to which the communication is to be delivered (the mail address of another actor); and,
- A communication which contains information for the target actor.

The communication is a tuple of values, which may be mail addresses of actors, primitive types (integers, strings etc.). The tuples may also be other actors.
Because an actor is in essence an active object, it has thread of control. This makes the normal garbage detection algorithms inappropriate for collecting actors. This is because an actor may be disconnected from the reference graph, which makes it garbage in the normal sense, but because it has a thread of control, it may reconnect to the graph by sending a message to a live object.

There have been various garbage collection algorithms proposed for the actor paradigm, and I cover some of them in section 7.5.1, although the implementation does not address issues concerning definition of garbage in the presence of such behavioural objects.
I attempt to give an overview of the classical garbage collection algorithms. First, I give some definitions that are used when describing the algorithms, especially the notion of liveness. I then present the algorithms, without details about efficient implementations. This is because the purpose of the presentation is not to give details of the algorithm themselves, but how they relate to distributed garbage collection techniques discussed later. This is because all the distributed garbage collection designs are based on work from the uni-processor algorithms.

4.1 Notation and Definitions

4.1.1 Mutator, Allocator and Collector

The Allocator receives requests from the application program and manages the allocation of free cells of the heap memory. The application program is used in a wide context here, and may refer to any process which requires space on the heap. This includes the kernel, file managers, network daemons etc. As mentioned earlier, the heap memory can be viewed as nodes in a graph which are connected to each other according to the references (pointers) they have on each other. The application program is then termed mutator since it changes or mutates the connectivity of the graph of the heap data structures. This term was used in [Dijkstra et al., 1978]

4.1.2 Liveness

At any time, the set of values that a program can manipulate directly are those held in the processor registers, those on the program stack that are in the current stack frame (these include the local variables), and those held in the global variables.

The above mentioned locations can either be data values or references to structures in the heap (pointers). The locations which hold references to structures in the heap form the root set of the computation.

Automatic heap management demands that the programmer follow certain rules. If the heap management is part of the language design, then the semantics of the language will dictate such behaviour. The rule is that:

Dynamically allocated data should only be accessible to the user program through the roots, or by following chains of pointers from these roots. The program should not access random locations in its own address space (in languages like C, this is possible, but in Java, it is not). The following is a more formal definition of liveness:
An object\(^2\) on the heap is live if its address is held in a root, or there is a pointer to it held in another live heap object.

Define \(\rightarrow\) as 'points-to': for any node or root \(M\) and any heap node \(N\), \(M \rightarrow N\) if and only if \(M\) holds a reference to \(N\). The set of live nodes in the heap is the transitive referential closure of the set of roots under this relation, i.e. the least set live where

\[
\text{live} = \{ N \in \text{Nodes} \mid (\exists r \in \text{Roots} \land r \rightarrow N) \lor (\exists M \in \text{live} \land M \rightarrow N) \}
\]

Equation 4.1 Definition of Liveness

The above is defined in [Jones and Lins, 1996]. Note that the above is a conservative estimate of the actual set of live cells. Data flow analysis by an optimising compiler would reveal some of the elements in the above defined set to be dead. Examples include a local variable after its last use in a procedure, or an obsolete pointer left in a register (to avoid instructions to clear it).

Techniques of determining the above set can be direct or indirect, and they are examined in the following sections alongside the algorithms that use them.

### 4.1.3 Two Phase Abstraction

Garbage collection automatically reclaims heap memory occupied by data objects that the program will never access again. The basic function of a garbage collector, when abstracted, consists of the following two parts:

1. Distinguishing the live objects from the garbage in some way, or garbage detection.
2. Reclaiming the garbage objects' storage, so that the running program can re-use it.

In practice, these two phases may be functionally or temporally interleaved. The reclamation technique is usually strongly dependent on the detection technique (presented in [Wilson et al., 1992]). The reason that I have presented it is because it will help identify the differences between the various algorithms discussed below.

### 4.1.4 Fragmentation

Fragmentation occurs in memory for all heap based memory systems, whether managed automatically or not. It does not concern some designs of distributed collectors which interact with local collectors, but there are other approaches which concern themselves with placement of objects on local heaps.

When a program requests for a piece of the heap memory, the allocator has to decide which part to give to the program. An allocator has to keep track of all the free cells in memory. When designing allocators, the goals are minimising wasted space and minimising time spent managing memory.

An allocator cannot control the number or size of live blocks – they are entirely up to the program requesting for the memory. A conventional allocator cannot compact memory. This is because it cannot move blocks around to make free

---

\(^2\) Again, the definition of object here refers to an individually allocated piece of data in the heap. It will be referred to as a node or an object.
cells contiguous. It must respond immediately to a request for space, and once it has decided which block to allocate, it must regard that block not movable until the application chooses to free it.

The problem that the allocator must address is that a program may free blocks of memory in any order. This will create “holes” of free space among live objects. These holes may be too small to satisfy future allocation requests. This problem is known as fragmentation. There is no reliable algorithm for ensuring efficient memory usage in the general case – and there will be none [Wilson et al., 1992]. It has been proven that for any allocation algorithm, there will always be the possibility that some application will use the memory in such a fashion that defeats its allocation strategy and result in severe fragmentation [Ullman, 1972].

Another severe problem that fragmentation brings is of locality of reference. Since the allocator cannot move objects that are live, it will allocate new objects in the holes left by garbage objects. This will result in objects with very different lifetimes to be interspersed in memory [Wilson et al., 1992]. Thus the "working set" of the application does not reside in the same memory area and is located in fragments all over the heap. Because the working set (newer objects) will be referenced very frequently, the usual benefits of caching gained by referring to a working set are lost. This becomes very severe in virtual memory systems, where disk space is used as a swap for main memory. Access times to disk are very slow and paging to disk is expensive and should be minimised. As the application has new objects spread over the entire heap, including possibly swapped pages, it will cause more paging.

But despite this bleak view, programs in the real world usually don't grind to a halt because allocators exploit regularities in program behaviour. This is a very difficult design factor, and one which is poorly understood even after 35 years of allocator research [Wilson et al., 1992].

4.2 Classical Algorithms

I discuss the three classical garbage collection algorithms: reference counting, mark-sweep and copy-collection. A variation on mark-sweep is also presented. Generation techniques, which can be combined with the classical algorithms to improve performance, are presented.

The reason for discussing the uni-processor algorithms is that all the distributed algorithms are based on the reference counting or mark-sweep algorithms.

4.2.1 Reference Counting

Reference counting, first presented in [Collins, 1960] is a direct method of finding the live set of nodes. Each object has an associated count of the references (pointers) to it. Each time a reference to the object is created, e.g. when a pointer is copied from one place to another by assignment, the object’s count is incremented. When a reference to an object is deleted, the count is decremented. When the reference count reaches zero, the memory occupied by an object may be reclaimed. This is because the zero count indicates that no pointers to the object exist and the program can never reach this object.

In a simple implementation of this algorithm, the reference count is usually stored in a header for every object. This header may store type information etc. This information is not visible at the language level.
When an object is reclaimed, its pointer fields are examined. One possible way to examine this information is to refer to compiler output information stored in an object's class (which may be an object in some systems). Any pointers found in the object are followed. The reference counts of these objects are also decremented. This is because pointers from a garbage object don't count towards determining the liveness of the object. If these objects in-turn have a zero reference count, then this repeats.

Thus reclaiming one object may result in the transitive decrementing of reference counts and reclaiming many other objects.

Comparing the reference count to the two-phase abstraction presented in section 4.1.3, the reference adjustment and checking of reference counts implements the first phase, and the reclamation phase occurs when the reference count reaches zero. This is an example of the two phases closely interleaved with each other. They are also interleaved with the execution of the mutator (program). This is because they may occur when a pointer is created or destroyed.

The advantage of reference counting is its incremental nature of its operation. It is interleaved with the mutator's own execution. It can be made completely incremental and real time. This means performing at most a small and bounded amount of GC work per unit of program execution.

The garbage collection phase is non real-time. This is because reclamation can result in more than one object being collected. Thus this can be deferred, by keeping a list of freed objects to collect at a later time. For example, for a real time application, a constraint can be that for every two-millisecond period, do not spend more than one millisecond performing collection.

The major drawback of the reference counting scheme is its inability to collect cycles. Figure 4.1 shows that although the algorithm's invariant is maintained (the reference count) on all objects, cells R, S and T are now unreachable and yet have not been collected. In real-world applications, structures with pointer cycles occur as linked-lists and trees that have leaf nodes point back to branches.

Fortunately, the other schemes detailed below can handle cycles easily. As a result, there have been several proposals combining reference counting with tracing schemes [Knuth, 1973], [Wise, 1979], [Deutsch and Bobrow, 1976] and others.

**Problem with Cycles**

The major drawback of the reference counting scheme is its inability to collect cycles. Figure 4.1 shows that although the algorithm's invariant is maintained (the reference count) on all objects, cells R, S and T are now unreachable and yet have not been collected. In real-world applications, structures with pointer cycles occur as linked-lists and trees that have leaf nodes point back to branches.

Fortunately, the other schemes detailed below can handle cycles easily. As a result, there have been several proposals combining reference counting with tracing schemes [Knuth, 1973], [Wise, 1979], [Deutsch and Bobrow, 1976] and others.
Efficiency

The problem with reference counting is that its cost is generally proportional to the amount of work done by the mutator, with a fairly large constant of proportionality.

One cost is that when a pointer is created or destroyed, the object to which the pointer pointed has to have its reference count adjusted. If a variable’s value is changed from one pointer to another, two object’s counts must be adjusted—one object’s reference count must be increased, and another must be decreased and checked to see if its reached zero.

The above cost can cause a great deal of overhead. This is because most stack variables are short-lived. For example, when an object is passed as an argument to a procedure (by reference), the counter is increased, but most procedures at the leaves of the call graphs are short lived [Wilson et al., 1992] and will return soon, decrementing the counter.

By treating local variables as a special case and excluding them from the reference count, some of this overhead is reduced. This is referred to as deferred reference counting [Deutsch and Bobrow, 1976]. The invariant must still be maintained though, and this scheme manages that by updating counts for stack variables less frequently.

Fragmentation

Because the reference counting algorithm does not move any objects that are live, its allocator has to allocate new heap memory where there is adequate space. This results in fragmentation (discussed in section 4.1.4). As mentioned, fragmentation is not unique to garbage collectors, it occurs in explicitly managed heaps as well. There are other GC algorithms that address the issue and are discussed later.

4.2.2 The Mark-Sweep Collector

The first algorithm for automatic heap management was a mark-sweep or mark-scan method, presented in [McCarthy, 1960] for the LISP language. Mark-sweep collectors, also referred to as tracing collectors, are named for the two phases that implement the abstract phases mentioned in section 4.1.3:

1. Garbage Detection: This is done by tracing, starting at the root set and actually traversing the graph of pointer relationships by examining the pointer fields in objects. This is usually depth-first or breadth-first and there have been various variations proposed. The objects that are reached, are marked in some way, either by setting a mark bit within the objects, or recording them in a bitmap of the objects.

2. Garbage Reclamation: Once all the live objects have been made distinguishable from the garbage, memory is swept. It is exhaustively examined (all cells) to find all the unmarked objects and reclaim them. All the marked objects have their mark bits reset, in readiness for the next phase.

Note that in this scheme, objects are not reclaimed immediately that they become garbage (references to them don’t exist), as they are in the reference counting algorithm (section 4.2.1 above). They remain undetected and un-
reclaimed till all available storage is exhausted. The collector is invoked when the allocator cannot satisfy a request from the mutator. The mutator’s execution is stopped while the garbage collector marking and sweeping. If the collector manages to reclaim space, then the mutator’s request is finally satisfied, and normal computation resumes. If on the other hand, the collector cannot find any space, then the mutator has to abort.

**Strengths**

The strengths of the mark-sweep algorithm have lead to its adoption in some systems. The functional language Miranda [Turner, 1985] is a notable example. The strengths are:

1. Cyclic data structures are handled naturally. This is because all objects are not examined from the point of view of other objects referring to them, but from the root set. If an island in the graph occurs, it will have no references directly or indirectly from the root set and thus will not be marked.

2. There is no overhead in pointer manipulation. As discussed in section 4.2.1 above, reference counting placed a high overhead whenever a pointer was created, copied or destroyed. In the mark-sweep, because the algorithm does not update its view of liveness every time the mutator mutates the graph of references, this overhead does not occur.

**Weaknesses**

The mark-sweep algorithm does suffer from a number of weaknesses, some of which are common with other GC algorithms:

1. **Fragmentation.** When used to handle objects of varying sizes (as in most object-oriented systems), because the collector does not move any objects that are live, the garbage objects create holes of free space. The allocator then has to find a hole big enough to satisfy the request of the mutator. This can be alleviated somewhat by keeping references to hole sizes and merging holes, but is difficult [Wilson et al., 1992].

2. **Stop/Start nature.** Because the algorithm does not update its view of liveness and reclaim till the free space in the heap is exhausted, there will be a substantial pause in mutator activity when the collector is working. For example, in the early 1980’s, a study found that as memory sizes grew faster than processing speeds, some large Lisp programs were spending between 25—40% of their time marking and sweeping, and users were waiting 5 seconds in a minute [Foderaro and Fateman, 1981]. For highly interactive systems such as window managers and video games, as well as real time applications and safety critical systems, this collector would not be practical. If on the other hand, response time is not an important consideration, mark-sweep does offer better performance than incremental algorithms like reference counting [Jones and Lins, 1996].

3. **Cost of Collection.** The cost of collection is proportional to the size of the entire heap. This includes both the live and garbage objects. All objects (live or garbage) must be checked to see if they are live in the marking phase. Again, all objects must be examined to see if they are garbage in the sweep phase. This limits any possible improvements on efficiency [Wilson et al., 1992].
4. **Thrashing.** The collector needs some head room in the heap to remain efficient. This is because the interval between invocations to the collector is dependent on the amount of free space recovered on each call (fragmentation makes it worse). When an application needs more space on the heap and there is none, the collector will be invoked. If the collector is not able to free adequate space, it will be called again very soon. This leads to the collector thrashing i.e. running, but not freeing much space. The mutator’s CPU share will decrease drastically [Jones and Lins, 1996].

4.2.3 **Mark-Compact Collectors**

This is a variation on the mark-sweep collector [Cohen and Nicolau, 1983]. As with the mark phase of the mark-sweep (section 4.2.2 above), a marking phase traverses and marks all reachable objects. Then the objects are compacted, moving most of the live objects until all of the live objects are contiguous. This leaves the free heap as one contiguous space. The compaction is often done by a linear can through memory, finding live objects and sliding them down to be adjacent to the previous object.

While the locality of reference that results is advantageous, the collection phase has a requirement of many passes over the heap (similar to mark-sweep). One pass is the marking phase, another to compute the new locations of live objects, a third to update the pointer’s to refer to the moved object’s new locations, and a fourth to move them.

Thus in practice, these algorithms are very slow. They are compared in [Knuth, 1973].

4.2.4 **Copy Collectors**

This algorithm is also referred to as a two-space-copying algorithm or as a scavenger [Ungar, 1984].

The first copying algorithm was due to Marvin Minsky for the Lisp language [Minsky, 1963]. Although because of the expensive and limited memory at the time, the live objects were not copied to memory, but to file and restored.

Copy collectors divide the heap equally into two semi-spaces, one of which contains current data, and the other obsolete data. Copying garbage collectors, like mark-compact (section 4.2.3 above) but not mark-sweep, do not really “collect” garbage. They just move the live objects and leave the garbage.

The most common kind of copy collector, which uses semi-spaces as described above is due to [Fenichel and Yochelson, 1969].

In normal use, the allocator uses only one of the two semi-spaces. It allocates linearly upward through the current semi-space and demanded by the mutator. The copy collector is invoked when the mutator’s request for more heap space cannot be satisfied by the current semi-space.

The copy-collector starts by copying all the live data from the current semi-space (from-space) to another semi-space (to-space). Once the copying completes, the two semi-spaces are flipped, with the to-space become the current semi-space and normal execution resumes. The collector works as long as the live objects and current mutator request fits in the to-space.
Issues regarding sharing are solved by having forward references, but are not discussed as a distributed collector will not be based on copy collection if it interacts with local collectors.

**Strengths**

1. The main benefit of copying collectors is their compacting property. Fragmentation is eliminated because the live objects are copied next to one another.
2. Allocation costs are extremely low. The allocator just needs to perform a simple pointer comparison to detect if it has run out of space when the mutator has a request.
3. Acquisition of memory is by simply incrementing the free space pointer. This is because the current semi-space is allocated linearly, in a stack like manner.
4. Asymptotic complexity of copying collection is less than of mark-sweep because it is proportional to the size of live objects rather than the heap size (as is the case for mark-sweep). Further, if the majority of objects do not survive the collection and become garbage, only a small portion of the heap must be copied. This is true for many functional and object oriented programs [Jones and Lins, 1996].

**Weaknesses**

1. The main drawback is the requirement of heap space that is twice the application's heap requirement. Although virtual memory systems help by having semi-spaces larger than the main memory, the locality of reference means that caching benefits will be lost, coupled with high paging penalties (discussed in section 4.1.4 above).
2. Copying live objects may potentially be expensive, but is offset by the fact that most objects will be garbage in some applications (see Strengths 4 above).

**4.2.5 Generational Techniques**

Tracing collectors such as mark-sweep (section 4.2.2) and the copy-collector (section 4.2.4) impose a high performance penalty on the applications. This is because all active data must be marked or copied. Tracing requires that every object (live or garbage) be visited. Copy-collection, requires (over requirement of the tracing) that the whole heap be touched every two collection cycles, even though the application uses half the heap (semi-space). This very poor locality of reference leads to an excessive number of cache misses and paging to swap space.

Tracing algorithms also spend considerable tracing long lived objects. A mark-sweep collector repeatedly marks long-lived objects, and a copy collector repeatedly copies long-lived objects over to the to-space. From the point-of-view of the GC, that is not being very productive.

A study has found that over 98 percent of storage reclaimable at one garbage collection had been allocated after the previous collection. The lifetimes of the objects were shorter than the time between to GC cycles [Foderaro and Fateman,
Garbage Collection

Modern languages such as ML often allocate short-lived objects representing intermediate expressions, or even control structures (such as environment frames), on the heap [Jones and Lins, 1996]. [Ungar, 1984] cites that a lot of research has been conducted to support the weak generational hypothesis that “most objects die young”. Generational garbage collection techniques are based on this evidence. They make GC more efficient and less disruptive by focusing on collecting the young objects.

The strategy is to segregate objects by age into two or more regions of the heap called generations. Different generations can then be collected at different frequencies. The youngest generation is collected most frequently and the oldest with the least frequency, or possibly not collected at all. The number of generations used varies with implementations. Standard ML of New Jersey (SML/NJ) [Appel, 1989] uses two whereas Tektronix 4406 Smalltalk uses seven [Caudill and Wirfs-Brock, 1986].

Generational techniques can be used together with mark-sweep or copy-collection. Issues concerning generational collectors are:

1. **Promotion.** All objects initially are initialized in the youngest generation space. They get promoted as they survive more collection cycles. The promotional rules can be tweaked for optimal performance.

2. **Inter-generational pointers.** Older generations refer to younger generations. When a particular generation is being collected, the references from older generations must be considered as part of the root set.

Some distributed algorithms are based on generational techniques. Others rely on local collectors which are generational.
CHAPTER 5

Project Requirements

Having given context to the problem of Distributed Garbage Collection in the first three chapters, I attempt to give the scope of my project. I start by giving details of the various requirements and show that some of them are in conflict. I attempt to prioritise the requirements in terms of the most benefit to memory allocation efficiency.

The first section gives the requirements as mentioned in the outline of the project. These requirements are discussed and potential conflicts and problems mentioned.

In the next section, the requirements that have to be met by distributed garbage collectors in general are given.

5.1 Requirements from Outline

This section presents the requirements of the project as given in the outline by the supervisor. I give the requirements and also mention potential issues and problems that may be faced when trying to build an implementation to meet these requirements.

The requirements as given in the outline are:

R 1. The ability to collect all types of garbage. This includes distributed cycles of garbage, i.e. objects on different nodes or processes referring to each other without being referred to from anywhere else (not being referenced from the root set). This is described as the requirement of being Comprehensive in [Jones and Lins, 1996].

R 2. Preservation of Resources. The collector needs to be timed such that memory is not exhausted. Unless the memory required by all non-garbage objects (live) exceeds the resources provided by the system.

R 3. Scalability. The algorithm must operate efficiently in a system with a large number of distributed nodes and/or objects.

R 4. Fault-tolerance. The failure of nodes should not affect the rest of the system. The algorithm should cope with both temporary communications failure (in which case the node should be able to rejoin), and total node or communication failure.

R 5. Dynamic Configuration. Nodes should be able to join and leave dynamically, i.e. the set of nodes covered by the garbage collection algorithm changes at run-time. Note that it also must be able to join two distributed systems that each performed distributed garbage collection independently from each other.

R 6. Interoperability. The implementation of the model should operate in a heterogeneous system based on the CORBA standard.
R 7. **Integration.** The algorithm should integrate with local garbage collectors i.e. its implementation should not require changes to them.

R 8. **Assume reliable communication media** which also preserves the order of messages, e.g. TCP/IP, and hence no provisions need to be made for the loss or out-of-order delivery of messages.

Note that the last requirement (R 8) is an anti-requirement because it reduces the number of issues that the design has to address.

### 5.2 General Requirements of a DGC

This section presents some requirements which must be met by a distributed garbage collector which were not addressed in the outline.

R 9. **Synchronisation.** Since liveness is a global property [Wilson, 1994] i.e. each object cannot know whether the object it does not need to refer to any more is garbage because other objects may be referring to it. This is made worse in the distributed case where any number of nodes may refer to an object.

In order for a decision to be made as to when global garbage collection should be performed, processors in all the spaces (the whole network) need to come to a consensus. This is equivalent to the pauses in uni-processor mark-sweep algorithms.

Thus the garbage collector must aim to reduce this restriction to the minimum.

R 10. **Concurrency.** Distributed systems are truly concurrent in their computation. All nodes have one or more processes which changes the connectivity of the reference graph simultaneously in an autonomous way.

R 11. **Correct.** The garbage collector must be correct i.e. it should collect objects which are garbage only. This may seem a trivial requirement, but it is not. This is because the model of computation may be fundamentally different from the uni-processor case. This is especially the case when dealing with active objects and actors (section 3.7). In the uni-processor case, an object is considered garbage if it is detached from the connectivity graph. A disconnected active object or actor with a process may be able to ‘reconnect’ and attach itself to the graph of references. The garbage collector should handle this if it is to handle the collection of actors.

R 12. **Expedient.** The garbage collector should have a rate of recycling or collecting memory which is sufficient to meet new allocation requests. Meeting this requirement is dependent on the local collectors. This is because fragmentation in a local heap may mean that the memory reclaimed by collecting a garbage object may not be able to be re-used.

R 13. **Efficient.** The space and time overheads imposed by the garbage collector should be within acceptable limits.

### 5.3 Requirements Analysis

Most of the papers and research in this area has revealed that designing distributed garbage collectors is difficult. Also, the focus of this project is to have a working implementation as one of its major deliverables. Most of the papers in
this area have designs which have not been implemented. This suggests that implementing the designs is a difficult problem. Moreover, most of these designs compromise one or more requirements.

The requirements presented in the last section are a complete set of requirements for an ideal distributed garbage collector. In reality, some of the requirements will have to be compromised to have a workable and efficient implementation (which in itself is a requirement). This leads to requirement conflicts. I will attempt to highlight some of these conflicts and issues in this section.

5.3.1 Requirement Conflicts

In this section, some of the possible conflicts of the requirements are presented.

Comprehensiveness and Integration with Local Collectors

Requirements state that the implementation of the global collector is to be comprehensive (R 1) and also integrate with local collectors of the nodes. Because the collector also has to be interoperable (R 6), the local nodes may be any CORBA IDL compliant language. Also, because the local nodes may have different run-times, the global collector cannot and should not assume that the local collectors are complete and comprehensive, i.e. if they do not collect local cyclic garbage, then the global collector will not be complete.

Fault Tolerant and using Reliable Communication Media

The outline states that the implementation should assume that the communication medium used for message passing between the nodes is reliable. It will ensure correct and in-order delivery of messages. Messages will not be lost and duplicate instances of messages will not be delivered. TCP/IP is given as an example communication medium (R 8).

The requirement also states that the algorithm should function even in the event a communication failure resulting in one or more of the nodes failing (R 4).

If a space is temporarily disconnected but manages to reconnect, either by restoring a broken connection or rebooting, then the open connections to the space previous to the failure must be re-established. The outcome of messages in transit when the crash occurred is not known. This means that otherwise reliable transport protocols such as TCP/IP can become lossy [Shapiro et al., 1994].

Interoperable with CORBA.

The CORBA specification [OMG, 1996] currently does not address the issue of distributed garbage collection.

If the global collector is to interact with local collectors in nodes, it has to be able to interact with the remote reference mechanism in some manner. This is because the global collector relies on the local collectors to inform it when some object has been collected. This mechanism has been designed for [Birrell et al., 1994] and [Shapiro et al., 1992]. But because the ORB specification in CORBA does not address this, there is no requirement for ORBs to have an interface by
which to inform interested parties (such as the global collector) of remote reference creation, duplication and deletion.

The Object Management Group current has a Request for Proposal that calls for the partners in industry that contribute the CORBA standard to address the issue of garbage collection. The CORBA standard states that ORBs should persist objects if they are not used and the ORB is running out of resources. The standard, however does not specify when the objects should be persisted. The request [OMG RFP, 1998]. The request outlines the use of distributed garbage collection techniques to identify objects which have no references to them at a certain time. These objects may then be persisted to stable storage and their memory de-allocated. This is to enable long running applications to have reduced memory requirements automatically without having explicit de-allocation efforts. But the problem of managing the persisted storage still remains. Another issue with CORBA is that the safety property of a collector may be violated because of the ability to have persistent references which can be written to file. This makes such references "invisible" to the collector. Using such a reference after the referred object has been collected violates the safety of the system.

This means that the design of a collector that interacts with a CORBA compliant ORB has ill defined semantics if the ORB is fully compliant as objects have long lifetimes in CORBA. If full compliance is not required, and persistent references are not of concern, then a possible approach is to use open source ORB (of which there are some available) The ORB can then be modified to notify the global collector of reference creation, duplication and deletion.

Also, most of the development efforts of systems use CORBA for the interoperability it provides with legacy systems. This is by far the biggest advantage that majority of the CORBA user community sees today. Some of the services are rarely used.

Also, because CORBA does not specify how objects are persisted when an ORB does not need to activate them, different vendors of ORBs have implemented the need in different ways [Siegel, 1996]. This means that the mechanisms for interacting with the ORB are already in place.
Before presenting the current work in distributed garbage collector algorithms, it would be helpful to examine the various design choices available when making design decisions. The identification of various dimensions of the design allows a better understanding of the trade-offs involved when making one choice over another. This helps adapt a certain algorithm better to its requirements.

The theoretical aspects, such as safety and liveness need to be considered, as well as practical issues such as ease of implementation, efficiency and fault tolerance.

The semantics of live and unreachable objects maybe different in different languages. The amount of concurrency in a distributed system is high and thus has to be taken into account. Determining reachability is not a local decision and propagating information to all nodes in the system has to be done in a timely manner, in the presence of many mutator processes quickly out-dating the information.

A uni-processor garbage collector’s requirements are to be correct (only collect garbage objects), comprehensive (ability to collect all garbage), expedient (rate of collection and de-allocating memory should at least be as high as rate of allocation by mutator processes). To be feasible for implementation, a collector should also be efficient (its space and time overheads should be within acceptable limits). Distributed garbage collectors have requirements above the ones that are to be met by uni-processor collectors.

Essentially, garbage collection algorithms come in two varieties. Counting algorithms, which count the number of references to an object. Counting is incomplete (hence fails to meet the completeness requirement), because it cannot detect and collect cycles of garbage. Tracing algorithms traverse the reference graph of objects from the root set of a process to its children to discover live objects. Tracing is complete, but very difficult to distribute in a large scale system because it is a global algorithm. Another great difficulty is that it has phases which require synchronisation.

The idea of suiting a particular design to its requirements is key in a search space such as DGC where there is no single ‘best’ algorithm. Design choices depend on the system properties such as scale and use.

### 6.1 Garbage Detection as a Consistency Problem

All the distributed collector designs that I have come across assume the existence of local collectors. They also assume co-operation with the local collectors (although to varying degrees, from the minimal to modifying local collectors). The local collectors in each node or process are not synchronised with...
each other, therefore inconsistencies may arise between the views of local collectors.

For example, consider a system with asynchronous message passing or partially synchronous message passing where message delivery times vary. Space 'A' owns object 'x' and sends a remote reference to space 'B', making object 'x' public. Space 'A' then deletes its own local reference to 'x'. If space 'B' performs a local collection before receiving the remote reference and space 'A' collects after deleting the local reference. If both of the spaces send the partial graphs to a global detector after their local collections, then the global, inconsistent view will be that object 'x' is unreachable from any space, hence garbage. This is incorrect because there is a remote reference in transit between two nodes. Another example of this is pictorially depicted in Figure 7.9 on page 73.

A consistent snapshot of the entire system is the obvious solution. Early DGC algorithms did exactly that: try to obtain a global snapshot by forcing all mutator activity in the entire system to stop for a consistent snapshot. The course in Distributed Algorithms [Kramer and Karamanolis, 1999] makes it very clear that distributed consensus is computationally expensive.

But, a consistent snapshot of the reference graph of the entire system is an overkill. This is because a conservative estimate of the reachable objects is safe. If there are objects which are considered live, but in actual fact are garbage, it will not violate the safety property of the system. This is because reachability is a stable property (at least in passive object systems): once an object becomes unreachable, it stays unreachable.

Inconsistencies are safe as long as they are on the conservative side. Implementation issues as well as efficiency constraints favour safe, conservative and/or incomplete collectors over strongly-consistent ones because they are computationally very expensive. Thus the distributed garbage collection problem is easier than the general consistency problem, but demands more efficiency and negligible resource consumption.

[Shapiro et al., 1994] term the management of remote references as a specialised instance of the consistency problem. They name it “Reference Consistency Protocol”, composed of two sub-protocols. A mutator level protocol is responsible for recording when an object becomes public, i.e. remotely reachable. This occurs as the mutator assigns remote references. Detecting when a public object becomes unreachable is a global level protocol, termed a distributed collector.

6.2 Design Space for Distributed GC

In this section, the various design issues are presented as independent dimensions. [Shapiro et al., 1994] discuss the design of distributed collectors from their experience in a similar manner, but do not address the issue of the amount of concurrency within a system and also its relation to active objects.

Distributed garbage collection comprises of three inter-related tasks:

- Tracking remote references, i.e. those that are inter-space references. This consists of registering references that traverse space boundaries. Most systems use the scheme of an outgoing indirection object (a stub in SSP terms) at the source space side, i.e. the site that accepts the remote reference. This stores the location to an incoming indirection, a scion in SSP.
terms, on the target site, i.e. the site that is being remotely referenced, which is also the owner space.

- Detecting remotely unreachable objects. This is a problem of keeping the set of scions consistent with their matching stubs (explained as a consistency issue above). The number of mutator processes, i.e. the amount of concurrency in the system and the numbers and types of failures, both to messages and spaces, make the problem more complex.

- Reclaiming unreachable objects and the resources they consume. This is usually done by the local collectors. Thus distributed collectors have a different role: they do not keep track of heap addresses and returning them to the heap’s free set. That task is delegated to local collectors. The distributed detector interfaces with the local collectors by making the set of scions part of the local node’s set of root references.

The next sections discuss each of the dimensions shown in fig in more detail.

Figure 6.1 Independent Dimensions in the Design Space of a Distributed Garbage Collector
6.2.1 Concurrency and Types of Objects

Certain single processor algorithms enforced a system in which there could only be one mutator modifying the reference graph. This was only common in the earlier algorithms, some of which kept graph information separate from the actual system. This requirement cannot be enforced in a distributed system because it is an essential aspect of a distributed system to have many threads of control.

Thus distributed collector algorithms have to cope with more than one mutator thread, often in the hundreds and for large scale systems and ones with lightweight threads and processes, potentially in the thousands.

The garbage collection algorithm will depend on the semantics of the language whose objects the garbage collector is going to collect. The computational thread associated with objects is one issue to consider. In most systems, the thread of control is external to the objects in the system, while a few systems have objects with their own thread of control.

Passive Objects

Passive objects may have state but the computational thread of control is external to the instance of the object. Classes of the objects thus define the structure of the state and the operations on the elements of that state.

Once all references to a passive object have been destroyed, i.e. the passive object has disconnected from the reference graph, it is reclaimable. The de-allocator may reclaim the resources and reuse them. Most of the current work in the distributed garbage collection field has focused on collectors for such systems. This is because most of the popular object-oriented languages have object semantics that are of this type.

This project concentrates on the more popular object-oriented systems which have passive objects and a separate thread of control external to them.

Threads in Java

In Java, classes which sub-class the system class java.lang.Thread can be thought of as active objects, but because they are actually a mechanism to create threads, it is safe to assume that objects are passive. This is because all threads objects belong to a thread pool, which may be the main thread pool or a new thread pool. Thus, ultimately, all thread objects are rooted. Also, each thread has a separate stack. Sub-classing threads is permissible, but without invoking the super-class constructor, it is not possible to obtain a new thread of control. Thus an application can never make a thread unreachable.

For the reasons cited above, I am assuming that Java objects are passive and the garbage collection algorithms are designed with such assumptions.

Active Objects

Some systems have semantics that make objects active. This may be on a selective basis, or for every object in the system. Active objects control their own computational thread.
Their management is more complex than that of passive objects. This is because a garbage collection algorithm will have to examine both the reachability of the object and its internal state. This is necessary because the object may have a reference to another object whose method execution can take itself as an argument and thus reconnect to the reference graph.

**Actors**

Some objects have behaviour that is more autonomous than that of active objects. Such objects may come to life by sending a message to a live object. The Actor Model, due to [Hewitt, 1977], is described in more detail in Section 3.7.

**6.2.2 Communication Semantics**

The design of the algorithm depends on the assumptions made about the characteristics of the underlying network. It can be seen from Figure 6.1 that the points along the communication semantics axis range from the instantaneous and reliable to asynchronous and failure-prone.

If we assume an instantaneous and reliable communication layer, then the distributed detecting phase of the algorithm is simple because it is easy to obtain a consistent global snapshot in such a system. Such strong assumptions can be made of a communications layer in a shared-memory multiprocessor, but not for loosely coupled, message passing networks of workstations. Also, even on reliable communication protocols such as TCP, the failure of a space can cause the protocol to become lossy. For example, an open TCP port has to be re-established after a crash and the outcome of the messages being sent in the TCP and IP layers when the space was crashing is unknown.

Causally ordered protocols [Birman et al., 1991], that are often used as a layer under higher abstraction primitives, are essential in making the design of the distributed detection easier because they make maintaining consistency easier. But as seen in the Distributed Algorithms course [Kramer and Karamanolis, 1999] they are not scalable in the implementations given as they rely on sending a set or sequence of messages in the lower layers of the protocol, which the layers above then chose to deliver, delay or drop.

An understanding of the characteristics of the underlying network is essential. This is because there are certain tasks which are impossible under certain types of networks, and are proven to be impossible. The classification of networks networks is given below:

**Types of Communication Networks**

The type of network governs how processes communicate and thus determines the progress of distributed computation, which is effected by communication between processes.

**Broadcast Channel Networks**

A broadcast channel network has a single shared channel which all processes use to communicate. In such a network, a process can broadcast a message to all other processes. But accessing the shared channel in a mutually exclusive manner is an issue in such networks.

Sanjay Bhudia
Examples of such networks are all the very commonly used networks in today's distributed systems: Ethernet, Token Ring, Token Bus and FDDI.

**Point-to-Point Networks**

In a point-to-point network, processes communicate over links that connect pairs of processes. This type of network can be modelled as a directed graph with nodes representing processes and edges representing communication links between processes.

It helps to define the communication primitives precisely. The following definition, found in a lot of distributed systems literature, is from [Hadzilacos and Toueg, 1994]:

Consider a link between a process 'p' and a process 'q'.

If 'p' invokes send with a message 'm' as a parameter, then p sends m to q; when the process returns from the invocation, then p completes the sending of m to q. If 'q' invokes receive with m as a parameter, then on return of execution of the primitive, we say that q receives m.

The following assumptions hold (in the absence of failures, which are discussed in the next section):

1. if 'p' sends a message 'm' to 'q', then 'q' eventually receives 'm'.
2. Every process executes an infinite sequence of steps. This is usual when modelling distributed systems. Each step is an execution of an operation which may be local (writing to memory, reading from memory, arithmetic operations etc.) or sending / receiving a message.

**Modelling on the Most Primitive Types of Networks**

If an algorithm can work on a network which gives the most limited primitives, then it will work for networks which provide more than those limited services. Thus distributed algorithms are always modelled with the minimum of assumptions from the underlying network. The point-to-point network is the most basic type, as seen above, and most networks offer richer services.

The next section describes the synchrony characteristics of networks.

**Synchronous Networks**

A point-to-point network is synchronous if it has the following properties, (in addition to the ones described above).

3. There is a known upper bound on the time to execute a step in a process. In modern computers, the execution time of a single local step can be considered insignificant.
4. Every process has a local clock with a known bounded rate of drift with respect to real time. This is also a valid assumption in a modern network of workstations, which has local clocks accurate to sub-second values.
5. There is a known upper bound on message delay. This time is the time to send (place on outgoing buffer), transport, and receive (place on incoming buffer) over any link.
The above properties are absolutely necessary if a distributed application is to use timeouts to detect crash failures. If any of the three properties mentioned above is violated, and a process 'p' times out on a message expected from a process 'q', 'p' cannot conclude that q has crashed. This is because the message delay could have been longer than expected, the clock used by 'p' to measure the timeout could have been running too fast, or 'q' could be executing steps slower than expected.

**Asynchronous Networks**

A network is asynchronous if there are no timing assumptions on the maximum message delay, clock-drift, or the execution time of a step. Algorithms based on this model are easier to adapt to new environments such as network types because they expect the worst from the network. Also, shared medium networks, which are the most common type (e.g. Ethernet etc) exhibit asynchronous behaviour in times of unexpected traffic. Thus, such algorithms can cope better in such situations.

**Impossibility Results**

In constructing a suitable design of a distributed collector, one should be aware of the impossibility results in distributed systems in general. This helps avoid attempting to design a phase of the algorithm that tries to do something that has been proven to be impossible. These results were covered in detail in the Distributed Algorithms course [Kramer and Karamanolis, 1999] I give a brief overview of two impossibilities:

**Impossibility of Consensus in Asynchronous Systems**

[Fischer, Lynch and Paterson, 1985] present the well known proof that consensus between all nodes in an asynchronous system in which process failures can occur (even a single process failure) is impossible. This is due to the combination of asynchrony, i.e. messages do not have an upper-bound on delivery time and the presence of process failures. A process whose vote is crucial to the decision may fail or its vote message may be delayed infinitely.

**Impossibility of Atomic Commitment under Network Partitions**

This is problem, often referred to as “the general’s paradox”, is technically known as Non-Blocking Atomic Commitment. It has been proven by [Gray, 1978] that nodes in a system cannot reach atomic commitment if the links in the communication network fail in such a manner as to cause a partition. This is because messages cannot be exchanged between the two partitions and nodes in one partitions have no way of knowing the decision of nodes in the other partition. Note that the problem exists even if the system is synchronous.

**6.2.3 Space Failure Semantics**

In the study of distributed systems, the classification of failures is important. This is because distributed algorithms are often modelled in ways that any part of the system that does not meet its performance to specifications is said to have failed. For example, if a communications layer guarantees delivery of a message to any part of the system within a given time of dispatch, t. If a message is
delivered at t+k after dispatch, then a higher level protocol will chose to ignore
the message. This may seem odd at first, but because an algorithm may be
relying on the timely delivery of information, and the late receipt of that
information introduces inconsistencies, it is better to model the transport layer
as having failed.

Classification of failure types helps tackle them better. They were studied briefly
in the course by [Kramer and Karamanolis, 1999], and a short section covering
failure types is found in [Hadzilacos and Toueg, 1994]. Following is a list of some
types of process failures:

- A process commits a clock failure if it violates the property of having an
  upper bound on the clock drift (property 4 above), i.e. its clock drift exceeds
  the upper bound.
- A process commits a performance failure if it takes more time to complete a
  local step than the specified maximum (property 3 above).
- A link commits a performance failure if it fails to transport some message
  within the specified upper bound (property 5 above).
- A process commits a crash failure if it fails to make progress for an
  arbitrarily long time. These are usually modelled by introducing a ‘crash
  step’ in the set of executable instructions, then a crash failure is defined to
  be executing a crash step in a schedule of execution. Crash failures were
  introduced in [Lamport and Fischer, 1982]
- A process commits a send-omission failure on a message ‘m’ if it completes
  the sending of m, but m is not inserted into its outgoing message buffer. This
  implies that the processes transport layer has failed. These types of failures
  were first considered in [Hadzilacos, 1984]
- Similarly, a receive-omission failure on a message ‘m’ is when ‘m’ is inserted
  into the receive buffer, but the process does not receive ‘m’.
- A link ‘l’ from a process ‘p’ to a process ‘q’ commits an omission failure if on
  a message ‘m’, if ‘m’ is inserted on the outgoing buffer of ‘p’, but ‘l’ does not
  transport it to ‘q’s incoming buffer.
- An arbitrary, Byzantine or malicious failure is committed if a process
  executes a sequence of steps that deviates arbitrarily from the sequence
  prescribed by its associated automation (from the code). These failures make
  processes exhibit any behaviour what-so-ever. Such failures are examined in
  [Pease, Shostak and Lamport, 1980].

The above failures are grouped into two categories that include both process and
link failures:

- omission failures include crash, send-omission, and receive-omission failures
  of processes, as well as link omission failures. The reasons for such
  classification are presented in [Perry and Toueg, 1986].
- timing failures consist of omission, clock and performance failures.

Note that the above are overlapping sets. Benign failures is synonymous to
omission failures in asynchronous networks and to timing failures in
synchronous systems. A system with benign failures does not commit arbitrary
failures, thus the sequence of steps executed by every process, whether correct or
faulty, is always consistent with the automation associated with that process (its
sequence of instructions). This means that a process will never change its state arbitrarily, or send a message that it was not supposed to. These kinds of failures are most common in practice.

A more restricted type of process failure, termed fail-stop, refers to a process that fails by crashing, but all correct processes are informed of the crash. Also, they have access to any information written by the crashed process in its stable storage before it crashed. This was introduced in [Schlichting and Schneider, 1983] and is possible to implement in a system using a “watch-dog timer”, which is a mechanism in hardware, or maybe in operating system software that counts down a timer at regular intervals. The software process that it watches is responsible for resetting the counter value to some non-zero number before it reaches zero. If the counter reaches zero, then the watch-dog process forces the process to fail and a message is sent to all other processes that the process has failed.

6.2.4 Reference Consistency Protocol

This dimension refers to how the system creates the surrogate objects (the stubs and scions) and how they are maintained consistent, i.e. the invariants maintained, such as every scion should have at least one stub etc. Also, it is concerned with how the mutators create scions and how the partial local collectors, which are not synchronised with each other, propagate reachability information across space boundaries.

This dimension is related with the communication semantics dimension. This is because maintaining consistency is not an issue in an instantaneous, reliable message network as propagating information can happen instantaneously, so no site has out-of-date information. Also, causal protocols in networks make the reference consistency protocol easier to design.

The two extreme points on this axis are the counting algorithms on the one end and the tracing algorithms on the other end. Most of the recent published algorithms in the distributed field are hybrid, i.e. they combine a counting algorithm with a lower frequency tracing algorithm, and so fall half way in the axis. Counting algorithms can either be fault-tolerant or non-fault-tolerant. For example, the reference listing algorithm and the SSP chains algorithm fall under the fault tolerant variety.

Tracing is feasible only in a small scale system because of the requirement of synchronisation between phases. Variants in the tracing collectors include:

- global tracing, which traverses the entire reference graph across all nodes in the system, and is consequently very expensive on time,
- centralised tracing, where a single node accumulates information on the reference graph and performs a trace on the information rather than the actual objects,
- tracing in groups, where the entire universe of nodes is divided hierarchically and tracing occurs within sub-groups to avoid bringing the whole system to a halt,
- and asynchronous global tracing, which imposes the least overhead on global synchronisation because local traces occur independently of each other and allow safe over-estimates.
For a large scale system, [Shapiro et al., 1994] recommend a fault-tolerant counting algorithm.

The way the algorithms deal with distributed cycles is also an issue. This is discussed in the following sub-section.

**Distributed Cycles**

Just as cycles of references to objects occur in uni-processor heaps (seen in section 4.2.1 and Figure 4.1), they also do in distributed systems. Also, the detection and collection of such cycles is a very difficult problem.

Figure 6.2 shows a simple cycle spanning more than one space. Public objects $x$ and $y$ have remote references to each other, but the root sets of both spaces do not have local pointers to either of them.

Thus the acyclic techniques based on reference counting will only work if garbage in the systems is acyclic, or if cycles are rare enough to be neglected. Section 2.1.4 shows that cycles occur in simple data structures such as doubly linked lists. They also occur frequently in functional language implementations, where recursion is commonly used. Cycles may also occur in imperative language systems, if recursion is implemented such that each recursive step returns an object on the heap.

Neglecting cycles because of the difficulty and computational expense in detecting and collecting them may be fine if they don’t occur frequently. But in certain systems, such as embedded systems that are part of control systems, the resources available are scarce and the system may be expected to be live for years (such as the Voyager interplanetary mission). Other systems where this can occur are long-lived server systems, where the cumulative space leaks can cause crashes due to insufficient free memory.

**Dealing with Cycles**

The distributed collector algorithms presented deal with cycles in one of four ways:

- The protocol does not deal with cycles.
- The protocol deals with cycles with span a sub-graph of the nodes, and that sub-graph is a logical space unit that the algorithm can deal with.
- The protocol forces objects that are members of a suspected cycle to migrate to a single node. If the cycle eventually confines itself to one node, then a local collector will eventually collect it, if the local collector is capable of dealing with cycles.
The protocol can detect and collect distributed cycles.

The fourth is the most desirable feature of a collector design, and no surprise is the most costly in terms of resource overheads in most current designs.

6.2.5 Nature of Spaces and Addressing within Spaces

This dimension is concerned with the relation of logical spaces with physical sites or processes. The simplest model is where a space is a process on a single site, with no memory sharing. In this case, a reference within a space is just a memory pointer, and all inter-space references are explicit messages because of no memory sharing.

A space can be composed of a small set of cooperating processes, possibly on different sites. This is the model used by many multiprocessor collectors. As long as the set uses a single addressing scheme, and assuming no independent failures of one of the processes and reliable communication, this is very similar to the classical model. However, the costs are different because of message-passing between the processes, of synchronization of the trace phases over the set of processes, and possibly of the mapping of addresses to per-process addresses. We know of no design where a space is a replicated set of processes. This would probably not pose any particular problem, as long as the processes in the group remain consistent.

Spaces may also be hierarchically nested. At the lowest level, a space could be a process with no memory sharing (the classical case, described above), at the next level up, a set of processes make a group, and so on. Nesting enables to better control the different trade-offs. For example, tracing at different rates at each level of the hierarchy is a possibility. Similarly, different design decisions for each of these dimensions could be used at different levels of the hierarchy.

The most appropriate design for a large-scale system is probably hierarchical. At a small scale, spaces might be distributed shared memory regions; at a higher level of the hierarchy, disjoint domains communicating with causal messages; at yet a higher level, disjoint domains using unreliable messages.

Object Hierarchy

An issue related to addressing within spaces is the discrimination of local objects from global ones, or in a hierarchical system, discrimination between objects with visibility at different levels.

It has been mentioned several times in this thesis that distributed object-oriented systems strive to make the semantics of local objects and remote objects the same to the application programmer. Having said that, inter-processor communication is still far less efficient than local operations. Thus a remote procedure call is slower by several orders of magnitude compared to a local procedure call.

So discriminating local objects from remote objects helps make a distributed system more efficient. Most implementations do treat local and remote invocations differently at lower layers of their protocols, and hide the differences to the application programmer who only sees the presented layer.

Many distributed systems change the status of an object from being a local one to a global one on export of the first remote reference to that object. This is
because the distributed support system has to manage the global object, but if it is entirely local, i.e. no remote references exist to it, then it need not consider that object. The normal language support system will usually take care of local object collection etc. This allows the design of distributed systems which rely on local services such as local garbage collectors.

When considering very large scale distributed systems that may have thousands of nodes and maybe hundreds of thousands of objects, it becomes necessary to structure the entire universe of all spaces into hierarchical sub-spaces, which may in-turn be divided into sub-spaces and so on. This aids object management because objects are only considered global at one space level if a reference to them crosses a space boundary. At a lower level, when the space is decomposed to sub-spaces, the same object may have many remote references from different sub-spaces. The design of scalable object-oriented systems is in such a manner to avoid the object management services such as allocators, reference tracking and binding mechanisms to be overwhelmed by the number of objects.

### 6.2.6 Cross-Space Reference Schemes

This dimension describes the reference mechanism across spaces. A key requirement is that all remote references must be visible to the collector. The difference between inter-space references is globally unique identifiers and forwarders. This determines whether a global search in reference repositories needs to be initiated when locating or resolving a reference.

**Globally Unique Identifiers**

Many well-known distributed systems, for instance OSF-DCE [Leser, 1992], base their references on global, universally unique identifiers (UUIDs). Identifiers in OSF-DCE can be arbitrarily manipulated by applications.

In some systems, the references can be persisted or freely stored in files, such as in CORBA [OMG, 1996]. This makes references become invisible for the time that they are persisted. If a garbage collection algorithm traverses the reference graph to determine reachability, it may mark objects as collectable as they would appear not to have references. Thus this violates the safety property of garbage collection.

Furthermore, global identifiers do not scale well. This is because of two reasons. First, any system will have the reference size defined. This will mean that at one point, when the system is large enough, the size will be a limit to the number of reference in the system. The second reason for the poor scalability of global identifiers is is when locating a target object given a reference. In the general case, a reliable distributed search is initiated when an object is to be located that matches a reference.

For example, in CORBA, a reference has a vendor identifier. This causes a search for an ORB (Object Request Broker) that is an implementation by that vendor. In a large system, there may be multiple nodes which are running an instance of an ORB from that vendor. This results in all the instances doing a local lookup to see if they have object implementations that match the reference.

Finally, [Black, 1993] identifies that the semantics of global identifiers are ill-defined.
Forwarders

This is a mechanism of locating an object that is pointed to by a remote reference by using point-to-point chains of references rather than a global identifier. This mechanism was introduced by [Fowler, 1986]. A remote reference is represented by a chain of stub-scion pairs (or any structures equivalent to an entry item and an exit item per space). The reference starts as a single pair when an object’s reference is first exported. On further duplication of this reference, the chain grows. Migration of the object also extends the chain in the other end.

This scheme of forwarder chains is used in systems with a distributed collector because they help maintain the invariants. Forwarding is more efficient than global identification in the general case, because a chain locates its target without any search. An object is located by traversing the chain till a local pointer is reached. The rules for forwarding ensure that references are visible to the collector [25]. However, forwarding poses the problem of inefficient method invocation and reliance on third party spaces for access. These problems are detailed further in Section 7.2.3. Both of these problems are addressed by the use of chain short-cutting mechanisms. An example of such a protocol is [Shapiro et al., 1992], which is discussed in detail in Chapter 8.

Some systems supplement the chains with global identifiers to ease short-cutting and equality tests [Juul and Jul, 1992], or to detect safety errors [Dickman, 1991]. Although they do not have the hiding problem of systems that expose their UUIDs (the hiding problem: need for a global search when locating a referent object), they do have the same scaling and semantic problems.
Chapter 7

Distributed Garbage Collection Algorithms

The specification, design and implementation of distributed garbage collection is a difficult problem. The designs presented in most, if not all of the papers have compromised on one requirement. Even then, most of the designs have not been implemented and thus their feasibility as practical implementations have not been tested.

As the project involves considerable research into possible ways of meeting some of the requirements, I give an overview of the many algorithms that I have looked into and attempt to give their shortfalls with regard to the requirements. Also covering many of the algorithms in existence helps express the rationale of my choice of algorithm for implementation.

I first describe reference counting based approaches, inspired by uni-processor reference counting algorithms (Section 4.2.1). Because reference counting schemes are acyclic, I then introduce hybrid schemes, which are reference counting based for most of the times and trace occasionally. I then give trace based techniques, which again are similar to uni-processor trace based techniques described in Section 4.2.2.

7.1 Reference Counting Approaches

In this section, I present the distributed garbage collection algorithms which are based on reference counting and give the shortcomings of each. Reference counting approaches are acyclic and so are do not fully meet the requirements of the project. On the other hand, most of the implementations of distributed collectors are referenced based in one way or another because they only impose a fixed penalty per distributed operation.

7.1.1 Distributed Reference Counting

Compared to the uni-processor reference counting algorithm (section 4.2.1), the distributed case has to handle new issues because the universe (space of all objects) is partitioned to separate spaces which communicate by message passing. These problems vary according to the transport layer that the distributed system assumes.

The naïve implementation of reference counting in distributed systems keeps a count with each public object. The count is updated on each reference creation, duplication or deletion by sending a control message to the owner space.

This increases the communication overhead significantly. Also, if the messages are delivered out of causal order, go missing or are duplicated, then the reference counting invariant is not maintained.
Race Conditions

When messages are delivered out of order, race conditions occur. The following is a decrement/increment race:

When a sender space duplicates a reference by sending a mutator message to a receiver space and immediately deletes its own remote reference to the object (see section 3.6.2 for reference duplication). For example, in Figure 7.1, an object \( v \) owned by space B has a remote reference by an object \( u \) in space A. This object sends a mutator control message to a receiver space C and duplicates the reference. When space C receives the message, it sends a control message to space B to increment the reference count. Also, space A deletes the remote reference, which causes a decrement control message to be sent. These two messages are in a potential race condition.

If the decrement message reaches the owner space B, then the reference count held at the scion will reach 0, and the object will prematurely be collected.

![Figure 7.1 A decrement/increment race condition](image)

Similarly, an increment/decrement race condition occurs when space ‘A’ sends a reference to space ‘C’ and thus also sends an increment control message to owner space ‘B’. The receiver space ‘C’ does not accept the message for some reason and thus sends a decrement control message to owner space ‘B’ to maintain the invariant. If the decrement message reaches before the increment message, then the object is collected prematurely.

Acknowledge Messages

This is a solution first proposed by [Lermen and Maurer, 1986] to address the race condition problem discussed above. Each increment message is acknowledged before sending a remote reference. For example, in Figure 7.1, space A sends the reference to space C, the method call is blocked by not replying. The increment message is sent by receiver space C to owner space ‘B’. Upon receipt of the message, the owner space increments the counter and sends an acknowledge message to space ‘C’. Space ‘C’ then installs the remote reference @\( v \) and returns the call of the initial method message from space ‘A’.

Space ‘A’,
upon returning from the control message sending call, sends a decrement message, waits for an acknowledge message and deletes the reference after the message is acknowledged.

Fault Tolerance Issues

Despite the acknowledge mechanism discussed above, this extension of reference counting is still not resilient to message failures because the increment and decrement messages are not idempotent. When a message gets lost in the system, the invariant is not preserved. As the algorithm is not resilient to message failures, it does not satisfy requirement R 4.

Efficiency Issues

The race conditions described above are solved by acknowledge messages, but at a great expense. The cost of each remote reference is three messages. This is the significant overhead of increased network traffic. Because reference duplications are common in distributed object-oriented systems, this is also expensive in terms of time taken for duplication because the reference exporting space (space ‘A’ in the example of Figure 7.1) is blocked till the increment is sent and the acknowledge is received.

7.1.2 Weighted Reference Counting

[Watson and Watson, 1987] proposed the use of weighted reference counting as an alternative to naïve reference counting. This technique eliminates increment messages and therefore the potential race conditions. Also, it is more efficient in terms of the number of messages compared to the acknowledge messages mechanism.

Each remote reference has two weights associated with it: a partial weight and a total weight. A stub contains a partial weight and the corresponding scion contains the partial weight and the total weight. The total weight does not change when a remote reference is created or duplicated, but decreased when a remote reference is deleted.

When a new remote reference is created (see section 3.6.1), an entry item (scion) is created and a total weight is initially assigned. This weight should have an even value greater than zero. The initial partial weight to be assigned to this scion is half the value of the total weight.

Each time a new reference is created (see section 3.6.1) to the same object, the scion’s partial weight is halved. A new reference may be created on a remote method invocation, where the caller is an argument.

When a reference is duplicated by a client space (see section 3.6.2), half the partial weight contained in the stub (exit item) is used as an initial value of the new partial weight. When the receiving space gets the reference in the mutator message, a new exit item is installed with the partial weight found in the message.

When a reference is deleted, (see section 3.6.3), the partial weight in the remote stub is sent in a control message to the owner space. This weight is decremented from the total weight contained in the entry item.
For any public object $v$, the invariant given in Equation 7.1 is held by the weight management mechanism detailed above from [Plainfossé and Shapiro, 1994]. When the total weight and partial weight are equal, the object has no remote references and has become local again.

$$\text{total \_ weight}_v \geq \sum \text{partial \_ weight}_v$$

Equation 7.1 Weighted Reference Counting Invariant

Figure 7.2 shows the creation of a reference. The total weight is initialised to 64, and the partial weight is 32. After the reference is initialised, the invariant will hold: $32 + 32 = 64$.

Figure 7.3 shows reference duplication. Before the duplication, the stub in space A had a partial weight of 32. The invariant was held ($32 + 32 = 64$). After the duplication, the partial weight is 16, and the new reference has a partial weight of 16. The invariant still holds ($16 + 16 + 32 = 64$).

Figure 7.4 shows reference deletion. Before the stub in space C is deleted, the scion in the owner space has a total weight of 64, after deletion, that is reduced by the value of the partial weight of the deleted stub C. The invariant holds after deletion: $16 + 32 = 48$. 

Weight Exhaustion

The main drawback of the scheme above is that for an initial total weight of $2^k$, the reference can only be duplicated $k$ times before it falls to $2^0$ and cannot be divided by two anymore. Thus if a practical implementation used an integer, which is 4 bytes in most systems, then after 32 duplications, the weight is exhausted.

A solution proposed by [Watson and Watson, 1987] is the creation of an indirect entry. The client which sends a mutator message causing a reference duplication will create an indirect entry when its stub contains a weight that is not divisible. The indirect entry has a weight initialised to it and the reference can be duplicated by halving this weight.

This allows arbitrary number of duplications to be made. But this scheme suffers from efficiency problems because an object that invokes a method via an indirect message will have the message travel two or more hops. In an extreme case, this can be a chain of indirect pointers. This is clearly unacceptable because it would offer very poor performance. The performance is examined in [Rudalics, 1990].

Fault Tolerance Issues

In general weighted reference counting is more efficient than the naïve reference counting (section 7.1.1) because reference duplication can occur without sending a control message to the owner of a public object. Also, because potential race conditions are eliminated, there is no need to send acknowledge messages.

However, weighted reference is not resilient to message duplication or failures. Because the messages are not idempotent, duplication will violate the weight management mechanisms.

A duplicate message causes the total weight to be lower than the sum of the partial weights. For example, in Figure 7.4, if the control message (-16) is duplicated, then the total weight will be reduced to 32 instead of 48. When
references start getting deleted, then at one point, the total weight will reach zero and the object will prematurely be collected.

Message loss will violate the invariant (Equation 7.1). If a message is lost, then the total weight associated with a reference becomes greater than the sum of the partial weights. For example, if in Figure 7.2, the message carrying the reference is lost, then the sum of partial weights in the system is 32, while the total weight is 64. When references started getting deleted, then the weight reduction messages sent to the owner will eventually reduce the total weight by the sum of all partial weights, which is 32. But because the first partial weight message was lost, then the total weight left will still be 32 and the object will never be collected.

Message losses and duplication breaks the liveness property of the algorithm.

7.1.3 Optimised Weighted Reference Counting

This scheme, proposed by [Dickman, 1992] improves on weighted reference counting in two ways: resilience to message loss, and indirected entry items for the weight exhaustion problem. The invariant presented for weighted reference counting (Equation 7.2) is weakened to an inequality:

\[
\text{total} \_ \text{weight}_v \geq \sum \text{partial} \_ \text{weight}_v
\]

Equation 7.2 Optimised Weighted Reference Counting Invariant.

The loss of a message or out-of-order delivery does not violate this weakened invariant. Message duplication still remains a problem though, because it would make the sum of the partial weights would be greater or equal to the total weight.

The other problem that this algorithm tackles is the use of indirected entries. When the weight is exhausted, and partial weights cannot be split, then new stubs are created with a special 'null weight'. This ensures that the total weight is always greater than the sum of the partial weights. But then this object will never be reclaimed at all. Thus the approach only works in systems where weight exhaustion will be infrequent.

7.1.4 Indirect Reference Counting

The solution to weight exhaustion that [Dickman, 1992] proposed was only feasible in systems where weight exhaustion is infrequent. An original solution to this problem is presented by [Piquer, 1991].

The key difference in his idea is to use two scion locators instead of one in the stub. One of the locators is the strong locator and the other one is the weak locator. In case there is an indirect chain, the strong locator of a stub points to the matching pair scion. The weak locator points to a better location in the chain and thus shortcuts the strong locator. The weak locator is always accurate if the objects are not mobile: it always points to the scion which is in the space in which the object is located.
When a client of the object performs a method invocation, the weak locator is used, and thus the method always gets invoked with the message travelling one hop.

The strong locator is used for distributed garbage collection. It prevents the target object from being reclaimed by keeping the chain of stubs and scions from being collected.

Reference Duplication

Like the algorithms for Weighted Reference Counting (Section 7.1.2) and Optimised Weighted Reference Counting (Section 7.1.3), remote references can be exported or duplicated without informing the object owner space. This avoids the race conditions that are associated with Distributed Reference Counting (Section 7.1.1).

Figure 7.5 (a) shows a public object $v$ in space ‘B’ with a remote reference held by object $x$ in space ‘A’. Object $x$ is sending a reference of object $v$ to object $u$ in space ‘C’. Figure 7.5 (b) shows the result of the message receipt. Upon first duplication of a remote reference (in this case, a reference to object $v$), a new scion is created with a counter initialised to one (seen in Figure 7.5 as scion 1a). Further reference duplication increments the counter associated with the stub. The stub is connected to the corresponding scion which refers to the target object (stub $v_A$ connected to scion 1b). The mutator message contains the weak location found in the stub and the remote reference. When space ‘C’ receives the message, the stub is initialised with the strong and weak locators from the message.

Figure 7.5 Reference Duplication for the first time in the Indirect Reference Counting Algorithm

An undesirable effect is that the indirect chain created in such a manner may loopback to the owner space. This can be avoided by assigning a unique identifier (UID) to each public object and copying that onto the stubs and scions. When a space first receives a message for reference duplication, it checks the UID to see if it already contains a reference to this object.
Scalability and Fault Tolerance Issues

The necessity for a UID for every public object is a challenge when the system has hundreds of nodes. Another issue is that there is a lot of garbage in the system due to indirect chains. This is addressed by [Goldberg, 1989], but at a high memory cost by having the scion have one counter per client space rather than one counter as above. This allows faster short-cutting and hence faster reclamation of the chain components.

The algorithm is not resilient to message failures, the algorithm’s liveness property is violated on message loss. This is similar to all the reference counting algorithms presented above. The safety property is violated on message duplication.

7.2 Reference Listing Approaches

Reference listing approaches differ from reference counting techniques in the way the entry items (scions) are managed. Instead of a single entry item (scion) for all clients, containing a counter, a space allocates a list of separate entry items, one for each client space that has a remote reference to that object.

The DEC Network Objects system by [Birrell et al., 1994] is an example of reference listing where explicit lists of client spaces are maintained.

It is usual to implement the scions as doubly linked lists [Plainfossé and Shapiro, 1994]. Reference creation, duplication and deletion operations are then list inserts and deletes, rather than counter decrements and increments.

The major advantage of reference listing is that the scheme is resilient to message duplication or loss. This is because the messages are idempotent. The same delete message may be sent several times. If a previous message has been received and processed, then a subsequent one is ignored, if not then it is processed.

However, race conditions are introduced. If a reference deletion message reaches the owner space before a reference creation or duplication message, then the object will be prematurely collected (as in other reference counting schemes mentioned above). [Shapiro et al., 1990] proposes a solution to the race condition problem by introducing timestamps in the messages.

Resilience to Space Failures

Approaches based on reference listing give support for mechanisms that make the collectors resilient to space failures.

For each public object, the owner space can compute the set of its currently live client spaces by traversing through the list of references of the clients and sending a message requesting a live message. If any of these spaces are down, which can be determined by a timeout, then the owner has a choice between:

1. Keeping the reference of the crashed space. This is appropriate if the object system has persistent object support and will recover the client object states on recovery.

2. Deleting the references if the failed space will not recover the state of the active objects when it crashed.
The choice taken depends on the system. [Birrell et al., 1994] take the choice of keeping references because their spaces recover and restore client object states. [Shapiro et al., 1992] assume fail-stop semantics and thus take the second option.

### 7.2.2 Stub-Scion Pair Chains

When introducing the remote reference model (section 3.5), I mentioned that I was introducing remote references using [Shapiro et al., 1992]. I mentioned that the introduction was using terminology from their work without introducing any of the differences that their approach has with other remote reference mechanisms. In this section, I introduce the mechanisms that [Shapiro et al., 1992] presented to support garbage collection.

The initial proposal was to support the collection of acyclic garbage only. It was supplemented by [Le Fessant et al., 1998] to collect distributed cycles. More details are given in Chapter 8.

The key differences between the work presented so far and SSP-Chains is the support for migration. Objects may migrate from one node to the other. In this case, the Stub-Scion pair that made the remote reference to the public object possible, will be extended to be a SSP chain with a pair of stub and scion for every node that the object has migrated to. In this way, an object may migrate without having to inform all its clients. They would have their stub-scion pairs extended. This was termed above as inserting and deleting elements from doubly-linked lists. This is because extending a chain is inserting elements to the list (an additional stub and scion) and short-cutting or deleting a reference is deleting elements from the chain (not necessarily from the ends of the chain).

There are two locators held in every stub. A strong locator, which is only used for garbage collection purposes, and a weak locator, which will short-cut a chain for efficiency. Method invocations are performed using the weak locator because of speed reasons. This idea was introduced by [Piquer, 1991] as mentioned in the Indirect Reference Counting section (7.1.4, above).

Message failures are tolerated by a conservative ordering of actions and by idempotent messages. Race conditions are avoided by time-stamping all messages and data structures. Crashes are tolerated by making space failure appear atomic to remote reference exports.

This algorithm is discussed in greater detail in Chapter 8.

### 7.2.3 Garbage Collection for Network Objects

The “Network Objects” system mentioned above, by [Birrell et al., 1994] of DEC has a collector based on reference listing. The reference listing approach, together with this algorithm, is described well in [Birrell et al., 1993]. I will give a brief description of their algorithm here.

#### Data Structures

The data structures maintained in the system are logically similar to the SSP system, but differ in implementation. The Network Objects system uses tables per space rather than the stubs and scions structures.
**Per Space Data Structure.** Each space maintains an object table. For each public object in the space, an entry exists in the table. If the space is the owner of the object, then the entry maps a remote reference representation (which is a unique identifier called a wire representation or wireRep in their terminology) to a local pointer to the object. Otherwise, if the space contains a client object to a remote public object, then the entry maps the wireRep to a pointer to the unique surrogate object to that object (surrogate object here is similar to the stub in SSP terminology).

The object table is used to locate objects referred to by incoming wireReps as results to method invocations or as arguments in method invocations.

**Per Object Data Structure.** Each object stores a dirty set, which contains identifiers for all processes that have surrogate objects (stubs) for this object, i.e. for all objects that are clients to this object. When a dirty set becomes empty, the object’s entry may be removed from the object table.

The dirty set is a conservative estimate of the clients of an object. It may contain elements which refer to spaces which are not clients at that point in time. This is a safe relaxation of the invariant because it only delays collecting of objects, but guarantees eventual collection. Also, while transmitting a remote reference for the object, the owner space itself is a member of the set. This ensures that the object is not collected while its references are in transit.

**Duplicating a Reference**

Unlike the SSP protocol described in Section 7.2.2, reference duplication of a Network Object does not cause an indirect chain because short-cutting is done immediately. This is done by informing the owner of the object whose reference is being duplicated. Note that short-cutting indirect chains is lazily done in SSP, and the owner is never informed of reference duplication.

The immediate short-cutting has an advantage over deferring it to the first method invocation (as in SSP). The first advantage is that a client does not depend on a third party (or space) to gain access to an object. For example an owner space ‘A’ of object O has a client space ‘B’ having a remote reference. Space ‘B’ then exports that reference to space ‘C’, thereby duplicating it. In the case of SSP chains above, this would create an indirect chain C-B-A and if space ‘B’ failed, space ‘C’ would not be able to access object O in space ‘A’.
With Network Objects however, this situation does not arise because upon reference duplication, the new client informs the owner. With reference to Figure 7.6, note that the diagram contains stub, scion terminology because it is easier to depict pictorially. In actual fact, the stub refers to a surrogate object. Rooting a stub refers to inserting the space identifier into the object’s dirty set. Creating a stub with NIL initialisation refers to inserting an entry in the spaces ‘object table’ with a NIL pointer, while the owner is being informed. There are no scions in the system, the owner of an object maintains a dirty set per object.

Duplicating a reference, as shown in Figure 7.6 occurs as follows:

i. Object ‘r’ in Space ‘A’ may want to include object ‘s’ in a method invocation or return result. It does so by marshalling the wireRep of object ‘s’.

ii. When the wireRep arrives at space ‘C’, the space looks up its ‘object table’ to see if there is a corresponding entry for the wireRep. If there is an entry, then it may be a surrogate object, in which case it is used. The entry may be to a concrete object, in which case the owner space is being returned a result or an argument that is its local object, so it is used. This is a case of immediate short cutting, where no surrogate is created.
If there is no surrogate or object, then either there is an entry with NIL, in which case another thread is in the process of creating a surrogate, so this thread suspends in waiting. If there is no entry, then an entry is created with NIL, and a dirty call is sent to the owner space. When the owner space receives the call, it adds space 'B'’s identifier to object ‘s’ dirty set. This is shown in Figure 7.6 as a Scion creation, when all that happens in fact is the dirty set update.

iii. The owner space ‘B’ returns the dirty set call, at which the new client space ‘C’ installs a surrogate and updates the object table entry with a pointer to the surrogate.

iv. The new client space ‘C’ then returns the call from space ‘A’.

**Avoiding Race Conditions**

A potential race condition similar to the decrement / increment race condition described in section 7.1.1 exists. This is when a client deletes its reference to the remote object ‘s’. This causes a deletion of the surrogate to ‘s’. The surrogate's finaliser sends a clean message to the owner space ‘B’. The client’s space identifier is removed from the dirty set of object ‘s’. This message may be in a race condition with the message from the new client, in this case space ‘C’, which sends a dirty call. If the clean call reaches before the dirty call, then the dirty set may be empty, in which case the object will be locally collected.

To avoid this, the dirty set is ensured to remain non empty while the reference is being exported. There are two cases to consider:

If the space exporting the reference is the owner (which is not the case in Figure 7.6), then the owner inserts its own space identifier in the dirty set. This is removed on completion of the reference export.

If the space exporting the reference is not the owner, then it has to be an existing client. All clients have a surrogate to the object. Thus making the surrogate reachable (uncollectable), by adding it to the root set of the space will ensure that the dirty set is not empty. (this is shown in Figure 7.6 i)

**Garbage Collection Protocol**

Some of the protocol details have been described above, but will be presented separately from the rest of the Network Objects details in this section.

The object table (mentioned above) was a hashtable with keys being the wireReps of objects and the value entries being pointers to surrogate objects or concrete implementations of objects. If the entry points to a concrete object, then the pointer is of a normal type. Concrete objects are made reachable by the entry in the object table, and so will not be collected even if there are no other local references to that object locally.

If the entry points to a surrogate, then the surrogate is pointed to via a weak pointer. Weak pointers to not keep the referent object from being collected by the garbage collector. The semantics of these weak pointers are similar to those of the Java 2 (formerly Java 2) Reference Objects API’s WeakReference reference object (java.lang.ref.WeakReference).

When a surrogate object that is referred to by a weak pointer is seen to be reclaimable by the local collector, its finaliser method is scheduled to be
executed. The finaliser code sets the object table entry with a distinguished "null weak reference value". It also sends a clean message to the object's owner.

When the 'clean' message is received by the owner, the sending client's space identifier is removed from the object's dirty set. When the dirty set becomes empty, and there are no local references to that object, the object may be collected by the local collector. This is done by removing the entry for that object from the object table. If this was the last pointer to the object, then it will be collected. If there are more local pointers to the object, then the object is still needed locally.

7.3 Hybrid Approaches

The reference counting and reference listing approaches presented in the previous section are inherently acyclic. They cannot collect distributed cycles. Therefore reference counting schemes are only employed in systems where cycles are non-existent or rare. Section 6.2.4, which introduces distributed cycles, presents that cycles do in-fact exist in common data structures such as doubly linked lists. They also occur when recursion is used. Thus hybrid schemes try to minimise the overhead imposed by collector algorithms by using reference counting techniques with a greater frequency than the more costly tracing techniques. This section discusses some of the algorithms that are hybrid.

7.3.1 Complementary Tracing

The key idea here is to combine an acyclic collector with a cyclic one. Usually the cyclic collector is activated at a much lower frequency than the acyclic one [Plainfossé and Shapiro, 1994].

[Dickman, 1991] combines an unspecified cyclic collector with his Optimised Weighted Reference Counting scheme (section 7.1.3, presented in [Dickman, 1992]). The cyclic collector is triggered using a heuristic based on measurements gathered during computation.

[Juul and Jul, 1992] have implemented a distributed cyclic collector for the Emerald system. A global cyclic collector based on distributed tracing marks remotely referenced objects and traces through the complete distributed graph of objects. Local collectors are allowed to run in parallel with the global collector by letting them assume that all entry items (scions) are roots. This technique merges garbage detection with garbage reclamation (see section 4.1.3). The peculiar feature of the distributed collector that they present is that it does not simply discard references and leave the local collector to deal with objects that are no longer needed, but actually does the memory de-allocation itself. This causes two independent collectors to collect garbage.
7.3.2 Object Migration

This scheme, initially proposed by [Bishop, 1977] in his thesis, essentially moves all objects that are members of a distributed cycle to a single space, and leaves the collection to a local cyclic collector. The selection and inspecting of objects suspected to be part of distributed cycles is based on heuristics.

Figure 7.9 shows an example of a distributed cycle which is eventually consolidated to one space. Initially (i), the objects which are elements of the cycle are spread across three spaces ('A', 'B' and 'C'). Step (ii) shows objects 'r' and 's' migrating and short-cutting indirections. Step (iii) shows the cycle entirely local, and the next local collection will reclaim the cycle.

A suspect object is first selected using heuristics. At each step, the subgraph rooted by this suspect object is migrated. This results in unnecessary migration in some cases, e.g. Figure 7.7 (i) and (iii), migration of 'r' would be enough, there was no need to migrate 's' as well.

An object suspected to be part of a distributed cycle is moved to a client space that refers to it. This assumes that the system is such that owners of objects know of all their clients. This is the case in reference listing approaches like Network Objects [Birrell et al., 1994]. Also, the local collector should be able to distinguish between public objects that are referenced locally (locally-rooted) from non-local public objects that are only remotely referenced and have no local references. This is necessary because only non-local public objects can be part of a distributed garbage cycle.
Heuristics help select which non-local public objects are more likely to be part of cycles. For instance, one such heuristic measure would be to select an object that has not been invoked a method on for a long time. Another may be the number of delete messages received to a particular client since the reference has been created and the last invocation of that object. Using multiple heuristics reduces the chance of migrating non-garbage objects, which is expensive.

Figure 7.8 shows a problem in using Bishop's migration technique in a system which has forwarders for remote references. An example of such a system is SSP chains [Shapiro et al., 1992]. Figure 7.8 (i) shows a cycle between two objects 'r' and 's'. After migration, the situation, seen in (ii) is clearly worse. This is because systems such as SSP chains use lazy short-cutting of the chains, (the SSP chains mechanism only shortcut on the first invocation of a method). Short-cutting would require two extra messages. Also, the short-cutting must be initiated by the migration mechanism itself because all objects that are elements of the garbage cycle cannot be referenced from the roots hence no operations can be done on them.

Mechanisms which do short-cut indirections immediately, such as [Birrell et al., 1994] are more suited to Bishop's Migration technique. For example, a reference duplication is immediately short-cut by the new client sending a message to the owner space. This is seen in more detail in Section 7.2.3.

The most serious drawback in using Bishop's technique for detecting cycles for the purposes of my project implementation is requirement R 6 (page 36). This states the implementation should be interoperable. Migrating objects that are implemented in different languages, or even the same language on a different platform is difficult and computationally very expensive. For example, a Java object will have a very different binary representation compared to a C++ object. Also, the methods may use language features only available in one of the systems. For example, a Java method may inspect its fields to find out which class an object is an instance of at runtime. There is no equivalent operation in C++, which maintains no type information at runtime.

7.3.3 Trial Deletion

This was first proposed by [Vestal, 1987]. Heuristics are used to select objects suspected of belonging to a garbage cycle. The object state is backed-up by cloning the object. The object is then deleted and the descendant's reference counts are checked to see if they drop to zero. Thus the algorithm assumes that local collectors are reference based.
Again, the main difficulty in implementing this algorithm is having heuristics that have a high hit rate in selecting the right objects.

### 7.3.4 Local Tracing

[Lins and Jones, 1990] propose a scheme which combines weighted reference counting (section 7.1.2) with mark and sweep in order to collect garbage cycles. The trace algorithm does not traverse the whole reference graph, but starts tracing from an object suspected to be part of a cycle. This is done each time a pointer to a public object, say object ‘s’ is deleted. The sub-graph whose root is ‘s’ is traversed. For, each object in the graph, the reference count is copied, and reduced by one. If it drops to zero, then the object is collected because it was part of a cycle. This is because an object which is part of a cycle is referred to by other objects, but not by an object that is rooted.

This technique suffers from the very high overheads that occur each time a reference is deleted. This is because an object may be part of a dense graph and deleting it will cause This is not acceptable as references get deleted often and this would halt the mutator.

Also, the tracing of a subgraph stops the mutator. If references are deleted with a high frequency, then the resulting wave of sub-graph traces will stop the mutator for most of the time. This problem was addressed later by [Lins, 1990]. He proposes that the sub-graph traces be queued and executed in batches. Even so, the overheads required per reference deletion are too high.

### 7.3.5 Summary of Hybrid Approaches

All three of the above approaches (Migration, Trial Deletion and Local Tracing) have a desirable property of tracing only those spaces which have cycles in them. This is better than Complementary Tracing techniques (section 7.3.1), which require tracing all the spaces within the system.

However, all the techniques rely on heuristics to pick on initial suspect objects. The cost of an incorrect prediction are very high, especially in Bishop’s migration where a migrated object is still referenced from the old space. Also, Bishop’s migration requires chain short-cutting and the local tracing technique requires the local collector to be reference counting.

### 7.4 Distributed Tracing Approaches

Reference counting techniques may be improved to handle fault-tolerance, message failures and efficient, but they still fail to collect cycles. This makes them unsuitable as a candidate algorithm for this project because they do not meet requirement R 1 (section 5.1). Hybrid techniques presented above are inefficient. I now present trace based techniques (based on mark-sweep (section 4.2.2) or copy collectors (section 4.2.4)) and describe some of the problems faced when trying to use tracing for the distributed case.

### 7.4.1 Distributed Tracing Problem

The use of a local collector to do the memory reclamation and a global collector to drop references to public objects when they become unreachable is the standard
approach adopted. The global collector interacts with the local collector by dropping references which causes the local collector to reclaim memory.

Synchronisation is the main problem with distributed tracing. The global mark phase has to be synchronised with the local sweep phases. During the global mark phase, the local collectors send and receive marking messages between the client spaces and owner spaces. The global mark phase is complete when all public objects that are reachable have been marked and there are no messages in transit. After this, when each space has triggered a local sweep independently, the local and public objects that are unreachable are reclaimed.

Figure 7.9 shows the synchronisation problem. When space 'C' is performing the global mark using information from the local spaces, it gets an inconsistent view due to the mutator activity. Its instructions to reclaim public objects which are not referenced are incorrect.
Distributed Tracing Approaches

(i) Space A, holding a local object x, sends a reference @x to B and discards its own local pointer to x.

(ii) Space A performs a local GC and concludes that x is not locally reachable; and sends this information to C, which is doing the global-mark.

(iii) Space B sends the reference @x back to A, then discards its own remote reference to x.

(iv) Space B performs a local GC, and it concludes that x is not locally reachable.

(v) C computes, from an inconsistent view given by spaces A and B, that x is garbage.

Figure 7.9 An Inconsistent DGC Snapshot leads to Incorrect Collection.
Another problem of fault-tolerant distributed tracing is the need to maintain consistency between the stubs and the scions in the presence of message failures, space failures and race conditions.

In reality, strict consistency is never achievable if local GC’s, mutators ad the global collector all operate in parallel with each other. This is because the messages are not instantaneously delivered. The effect of communication networks on the design of a collector are detailed in Section 6.2.2.

A standard solution to this problem is to use strong protocols such as a global barrier. This puts a barrier at the end of all the local mark phases, so that a global sweep can occur [Lang et al., 1992]. An alternative is to allow safe inconsistencies, i.e. those which don’t violate the safety invariants of the algorithm [Plainfossé and Shapiro, 1994]

### 7.4.2 Hughes’ Algorithm

An algorithm based on the distributed mark and sweep, but marking bits are replaced by timestamps is proposed by [Hughes, 1985]. The key idea of the algorithm is that a garbage object’s timestamp remains constant, while a non-garbage object’s timestamp increases monotonically. The most appealing feature of the algorithm is that traces do not need to occur synchronously, although the calculation of a global minimum timestamp (objects with stamps below this can be collected) needs synchronisation.

A global collection cycle propagates an exit item’s (stub) timestamp to its successor entry item.

In order to avoid the barrier synchronisation between the mark and read phases as mentioned above, a timestamp threshold is computed. Scions contain a timestamp initialised with the current time from a global clock.

The local GC traces objects from the local root and the scions. An exit item reachable from the root is marked with the time at which marking started. An exit item that is reachable from an entry item receives the timestamp of the entry item.

Entry items (scions) that carry a timestamp less than the global threshold can be safely collected.

The time when the local GC of a node is triggered is called the GC-time. The local root is labelled with this GC-time. The GC scans the objects reachable from the roots first, then from entry items down to exit items. Entry items are scanned in decreasing time order to avoid multiple scanning of the same object.

At the end of a local GC, up-to-date exit item timestamps are sent to the corresponding owners to increase the entry item timestamps, if they are lower than the exit item's.

When increasing each entry item timestamp, the space records a redo timestamp equal to the highest time propagated. When all timestamping messages have been processed, the owner sends back an acknowledgement message to the sender. The sender space collects acknowledgements before increasing its own local redo to GC-time.

The basic idea is that any entry item lower than a global threshold is garbage. The threshold is equal to the lowest values of all redo. Computation of the threshold is tricky and requires a global termination protocol.
The shortcomings of the algorithm are those of scalability. Determining the global threshold is a termination protocol which requires co-operation from all spaces. This slows down local collection. Also, it is not resilient to failures, since a failed space prevents the increase of the threshold, hence blocking global collection on all spaces. If a space has high mutator activity and its local GC is slow, then it will leave the global threshold stuck to an old value. This is true even if this space does not have remote references to other spaces.

Figure 7.10 illustrates how the algorithm reclaims distributed cycles. The example is a simple case of two objects between two spaces but illustrates the algorithm in the general case. The timestamp of a scion is noted as the scion ID with the stamp as \( (b.\text{timestamp}) \) and similarly \( (yA.\text{timestamp}) \) for a stub.

(i) At clock=40, space B triggers a local GC and propagates a timestamp 40 to entry item \( a \) of space A.

(ii) At clock=50, space B removes the local pointer from \( R_b \) to \( u \). At clock=70, space A processes and acknowledges the timestamp message 40. It initialises a local GC. Then, \( yA.\text{timestamp}=70 \) and \( uA.\text{timestamp}=40 \). It propagates timestamp 40 to entry item \( b' \) of B.

(iii) Space B processes and acknowledges timestamp messages. Thereafter \( b.\text{timestamp}=70 \) and \( b'.\text{timestamp}=40 \). At clock=85, space B triggers a local GC. Entry item \( a \) has a timestamp of 40 is propagated to exit item \( y_b \) and timestamp 40 message is flowed to entry item A.

(iv) For space B, at clock=100, threshold = minimum of (70 and 85). At clock 115, A triggers a local GC, since \( a.\text{timestamp} \leq \text{threshold} \) then entry item \( a \), and object \( v \) are collected.

(v) At clock=100, B triggers a local GC; \( u \) is reclaimed; \( b.\text{timestamp}=115 \).

As seen from the above example, a local collection triggers a global marking. This example is presented in [Plainfossé and Shapiro, 1994]
Distributed Garbage Collection Algorithms

(i) space B triggers a local GC

(ii) space A triggers a local GC, space B deletes local pointer to u.

(iii) Space B triggers a local GC and propagates timestamp 40

(iv) Space A triggers a local GC; threshold increased to 70

(v) Space B triggers a local GC and reclaim u.

**KEY**

- Timestamp message
- Local clock/local redo
- Ack message
- Global threshold

*Figure 7.10 Example of Hughes’ Time-stamping Technique.*
Space Failure Issues

The algorithm above is appealing because it effectively manages to collect distributed cycles, but because all spaces have to participate to determine the threshold, it is not resilient to space failures.

Scalability Issues

Because each local garbage collection in one space triggers computations to determine the global marking in all the spaces, the algorithm is not suitable for a large-scale system.

Improvement on Hughes' Algorithm

[Le Fessant et al., 1998] address the problems mentioned above with Hughes' algorithm. They propose an extension to their Stub-Scion-Pair-Chain collector, which is acyclic [Shapiro et al., 1992].

They use a dedicated space to calculate the threshold. The algorithm can cope with space failures of participating nodes. Also, a local collection does not trigger a global mark, so it is scalable. This algorithm is covered in detail in the 8.10.

7.5 Garbage Collecting Actors

7.5.1 Halstead's Algorithm

[Halstead, 1978] uses the concept of an actor reference tree, a set of processors and connections between processors such that each processor has a reference to the actor. Garbage collection is performed by reducing the tree until it contains a single processor. A local collector is then used on each processor to collect garbage actors. This scheme cannot collect cycles [Jones and Lins, 1996].

7.5.2 Washabaugh's Algorithm


Nelson's marking algorithm for actors assumes that the mutator is halted, and that all actors in the system reside in the same node. It is an example of tri-colour marking, presented in [Wilson et al., 1992].

[Kafura et al., 1990] presents an approach comparable to mark-sweep (section 4.2.2). It marks all root actors black. It then looks for a black actor by performing a depth first search from all active actors.

If a black actor is found, then the originating actor is marked black, together with its acquaintances. The algorithm is repeated. At the termination, (which is not described) the actors which are not black are collected.

[Washabaugh, 1989] shows two major problems with both the above algorithms. The first one is the requirement of the global collector to operate concurrently with the local collectors and mutators, and to synchronise with all local collectors. The second problem is the determination of termination. Termination is complicated because a global collector at one node may finish collecting only to
be awakened by activity in another node. Agreement can be achieved by using a rotating token. The token is sent out and if it ever returns to the owner, signals termination.
In this chapter, I give the rationale for the choice of the algorithm that I have implemented. I give details of the design decisions that meet the requirements of the project. Because the algorithm is closely tied to the distributed object oriented system, I have implemented a distributed object-oriented system as well (see Chapter 9). Design details of the system are presented in this chapter as well.

The initial algorithm presented is acyclic as it is based on distributed reference counting. Subsequently, I present cyclic extension to the algorithm which enables the collection of distributed cycles.

The distributed system uses forwarders ([Fowler, 1986]) rather than globally unique identifiers (see Section 6.2.6 for a discussion on the differences). An application programmer can access remote objects in the same manner as local objects.

The emphasis on the design of the distributed system is more than what should be presented in a thesis focussing on garbage collection, but the reason for this is the effort in implementing the distributed system. The system was entirely necessary because the distributed collector protocol was to be built on top of the distributed system.

8.1 Overview

In this section, I give an overview of the features of the SSP chains system, which is the design of the distributed system and the acyclic distributed collector. The system is due to [Shapiro et al., 1992]. The following are the features and design decisions taken in the system:

- Access to remote objects is via surrogate objects, called stubs and scions. These have been discussed earlier in this thesis. [Bishop, 1977] introduced the first such system in his thesis. An object's method signatures are exposed in its stub. Thus, to the client of the object, i.e. the object holding a reference to the remote object, the functionality of the object appears the same. This is the most commonly taken approach in distributed object-oriented systems. Network Objects [Birrell et al., 1994], Java's RMI, and CORBA [OMG, 1996] all use the same approach.

- Remote references are not identified by globally unique identifiers, but by point-to-point chains of stub-scion pairs. This approach has scalability advantages because locating references does not initiate a global search in all the nodes. This is discussed in section 6.2.6.

- Each outgoing stub contains the location of two incoming scions. The strong location matches the stub with its appropriate scion. The weak location indicates some scion further down the same chain of stubs and scions. This is closer to the target, and the system tries to maintain it so that it points to
the target all the time (it does so in the absence of object migration). This approach was first introduced in 'Indirect Reference Counting' [Piquer, 1991], discussed. He used as a way round weight exhaustion (see Section 7.1.2), but it is used here as a short-cutting mechanism and to make method invocation efficient.

- The rules for constructing, duplication and destroying remote references, i.e. SSP chains are stated precisely, and have been proven correct. Thus they ensure that a set of invariants hold.

- Distributed Garbage collection is supported by inferring on the existence of remote references on a particular object. The design has been constructed with distributed collection in mind. The algorithm used for the acyclic collector is a variant of reference counting. It does not explicitly maintain a counter, but the invariants on the stubs and scions effectively mean that logically, a reference count is kept.

- Most of the distributed techniques covered in Chapter 7 imposed strict consistency requirements. But this design allows safe inconsistencies, i.e. ones which do not violate the invariants. The usual liveness conditions are relaxed, allowing garbage to accumulate in the system (floating garbage).

- Failure and network assumptions (discussed in sections 6.2.3 and 6.2.2) are weak. Thus the design can be used in networks and systems with widely varying characteristics. Here are some of the characteristics of having such weak assumptions:
  - Messages can be lost, delayed, duplicated, or delivered out of order.
  - Processes (spaces in the SSP terminology) may crash in a fail-stop manner (refer to section 6.2.3 for a definition of a fail-stop crash). Also, mentioned in section 6.2.3 is an outline for an external failure detection system to enable processes to fail in this manner.
  - There is no need for clocks to be synchronised across nodes or spaces. This is because the algorithm uses Lamport's Logical clocks [Lamport, 1978].

- Both the distributed system, and the distributed collector algorithm are scalable. This is due to several design features:
  - References can be duplicated by clients, i.e. one client can export the reference to another space my a method invocation (and thus make that space a new client) without having to inform the object owner space. This enables potentially hundreds of clients to exchange reference information which the object owner space is not aware of. Systems in which the object owner is informed of every new client scale poorly. This is because the owner space may be inundated with reference updates. Reference counting systems in which an explicit reference count is maintained are examples of such systems. The Network Objects system [Birrell et al., 1993] also exhibits such scalability issues because of its immediate short-cutting mechanism.
  - Remote references are implemented as forwarders, chains of stubs and scions. This is in contrast to globally unique identifiers. As discussed in section 6.2.6, globally unique identifiers do not scale well because a node may need to initiate a global search in all other nodes. This is especially
true in systems which allow object migration, as caching an object location will not aid subsequent location.

- The collector protocol only sends messages between pairs of spaces. Because there is no need to send information to all the nodes in the system, this scales well. Also, reducing the frequency of the messages sent by the distributed detector protocol does not violate its safety property, it only relaxes its liveness property, thus the amount of floating garbage increases temporarily.
- The messages exchanged by the collector are idempotent, thus the loss or duplication of a message does not violate the safety of the collector.
- There is no synchronisation between processes needed.

8.2 Parts of the System

The protocol does not impose any semantic constraints on the objects that populate the system. An object may contain arbitrary references to other objects, represented as local pointers. An object will have a well defined interface exposing the methods and their signatures that can be invoked on an instance of that object.

The method signatures may have references to other objects, remote or local, as arguments. The execution of some method may return a reference to any remote or local object. This can be assigned to variables in the object that invoked the method call. Such method invocations modify the reference graph of the whole system dynamically.

Sending a reference across a space boundary involves "marshalling" into a network representation of that reference. Marshalling involves transforming a local pointer (which only has meaning to the space that it points to) to a standard external representation, which can be interpreted by any space. This is done by a Presentation Protocol. On receipt of a message containing a marshalled reference, the reference must be "demarshalled" into another local pointer.

Reference chains can only be extended by exporting a reference or by object migration. Marshalling and unmarshalling an already public reference adds a stub-scion pair to the tail end of a reference chain. This is detailed further in section 3.6.2. When a public object migrates, all chains to it will extend by an SSP pair on the head end.

The Invocation Protocol allows the source of a chain to invoke a method on the target object. If the chain is indirect, it is short-cut as a side effect of invocation. There are two cases to consider when invoking methods on indirect chains:

- An exact weak location, and indirections on the strong location. The method invocation is done by sending a message using the weak locator. When the target object finishes method execution, the target space marshalls the result and also includes the location of a direct SSP. This direct SSP would be allocated if it did not exist.

- The weak location is not exact. This can happen if the object has migrated since the last method invocation using the current stub. The target scion does not redirect the method call, but returns an exception message containing the exact weak location of the object. Upon receipt of the exception message, the weak locator of the stub is updated as a method call.
made to that location, which is the same as the first situation, if the object has not migrated again while the exception message was being sent.

As the above cases show, there is no message overhead for short-cuting an object when the reference is indirect due to reference duplication (see Section 8.8). If on the other hand, the short-cutting is due to object-migration, then there is an overhead of two messages above the normal method invocation: the exception message, and the repeated message invocation.

When an object frees a remote reference by not pointing to the stub of the chain, then the SSP chain becomes unreferenced. Unreferenced chains are collected by the garbage collection algorithm. The distributed garbage collector has two sub-protocols, the local GC per node and the distributed Cleanup Protocol. In each space, the local garbage collector, assumed to be correct, but of any variety (see Chapter 4 for an overview on uni-processor collectors), collects locally inaccessible objects, including stubs which are tails of unreferenced chains. Collecting such stubs initiates the collection of the unreferenced chain by collecting the matching scion. This is done by the Cleanup protocol, which propagates reachability information via background messages to neighbouring spaces.

When two separate remote references refer to a single public object, the scion that is at the head of each of the chains does not merge into one structure, although they both point to the same object. This is in contrast to many systems, such as Emerald [Black et al., 1986]. This approach of having one scion for every source space to a single target is essential for fault tolerance.

When two remote references are from the same source space to the same target space, they may take different paths due to different indirections. This can occur because the references may have been propagated through different routes. After method invocation from both chains, when short-cutting occurs, the common tables per space will cause one of them to disappear.

### 8.2.1 Spaces

Spaces were introduced as a way of expressing local nodes in a distributed system in section 3.5.1. In this section, the particulars to spaces which concern the SSP protocol, namely the services that a space must have, are presented.

Each space should be able to be uniquely identified. The SSP protocol assumes a unique identification naming system and does not specify one. In the implementation presented in this thesis, a space is uniquely identified by the host IP address and port number that it is listening to for remote messages.

Each space 'A' has a timestamp generator, stampA(). The timestamp generators need not be synchronised across spaces. They do not have to give the reading of time in the normal sense. The only requirement is to give a monotonically increasing value on every reading. This is the requirement of clocks in [Lamport, 1978]. The stamp value is used to mark every message sent to another space. The stub used to send the message marked and so is the scion that receives the message.

Each space 'A' also maintains an array of timestamp values, thresholdA, received from other spaces. This serves to filter the messages from other spaces. Only messages timestamped with a value greater than the corresponding space value are accepted, the rest are dropped.
After a space comes into existence, it communicates with other spaces. This effects distributed computation. Eventually it terminates. Thereafter, it does not re-appear, i.e. the same space name is never used again.

A space may disconnect due to network overload or some other reason. A disconnected space may reconnect or terminate. This is because disconnection is safe: a disconnected space cannot modify the reference graph.

8.2.2 Objects And References

Spaces contain passive objects. Each object presents a well-defined interface. The interface contains methods with signatures specifying the number and types of arguments and the return types. The objects implement the methods defined in the interface. Note that this is semantically the same as instances of classes implementing defined interfaces in object-oriented programming. But in this case, the objects need not be instances of classes. They may well be procedural code.

An object may hold references to other objects. In this case the object is the source and the objects being referred to are target objects. The object may invoke methods on the target objects through this reference.

A reference may be passed as an argument in a method invocation to another object. It may also be returned as a result of a method invocation. Both of these are ways of propagating references of objects to other objects within the system. References may be local or remote. The occurrence of local objects is assumed to be much more frequent than remote references.

Stubs represent outgoing remote references, scions represent incoming remote references. The creation, management, use and deletion of stubs and scions is entirely done by the SSP protocols. An application programmer using the SSP system need never be aware of their existence.

All application software deals with local pointers, even for remote references. A local reference is a pointer to a local object. A remote reference is a local pointer to the stub of the remote object. The stub exposes the same interface as the remote object, giving the application program the illusion that the object is local.

A stub contains a locator which consists of a strong location and a weak location. Each of these parts identifies a scion. This is done by holding the space name in which the scion resides and the unique name of the scion within that space. The strong part identifies the matching scion to this stub. This is the next scion in the SSP chain. The scion itself either points directly to the target object, or to a stub which is a further pair in the SSP chain. The weak part points to a scion which may directly point to the target object (always the case in systems without object migration) or to a scion closer to the target object in relation to the length of the SSP chain.

For any given scion, there is never more than one stub in a space, even if that space contains many references to the remote object. Similarly, every stub that points to a given space has a corresponding scion in that space, i.e. there is a different scion for every stub. As mentioned before, this is unlike many distributed systems where multiple exit items may refer to a single entry item.

A scion may exist without a stub. This is a safe inconsistency, but it relaxes the liveness property of the distributed collector. It introduces floating garbage in the system. The floating garbage associated with a scion without a corresponding
stub is the scion itself, and all the objects that it refers to, i.e. the sub-graph of referents rooted by the scion.

Distributed garbage collection relies on the invariant that exist only uninterrupted chains of stubs and scions in the system. Any scion that exists without a matching stub may be collected. This may cause the collection of the object that the scion was referring to. Weak stub locators are used for locating target objects and communicating the method invocation from the reference sources. The weakly indicated scion will not be collected as long as it is part of the strong chain.

8.2.3 Reference Duplication

Marshalling and unmarshalling an already public reference adds a stub-scion pair to the tail end of a reference chain.

For example, Figure 8.1 (i) shows an object 'r' in space 'B' which has a remote reference to object 't' in space 'C'. The remote reference is a SSP chain \( t_C \rightarrow c \). Object 'r' also has a remote reference to object 's' in space 'A' via chain \( s_A \rightarrow a \). Now, object 'r' invokes a method on object 's' with an argument that is the reference to object 't'. Thus, it is exporting the reference across a space. This is a situation where an existing client space to the object 't' makes space 'A' a new client space.

After the method invocation, seen in Figure 8.1 (ii), the argument export has resulted in the SSP chain referring to object 't' being extended in the tail end.

![Diagram showing SSP chain extension by exporting references](image)

(i) 'r' refers to 't' using chain: \( t_C \rightarrow c \)

(ii) 'r' invokes method on 's' via stub 's_A' and has 't' as an arg, hence extends chain.

Object 's' now refers to object 't' via chain \( t_B \rightarrow b \rightarrow t_C \rightarrow c \).

8.2.4 Object Migration

A public object may migrate from one space to another. When this happens, all SSP chains referring to that object from other spaces are extended by a stub-scion pair on the head end of the chain. Note that the SSP protocol does not have mechanisms to enable object migration. This is because installing objects and copying state from/into objects is system dependent. If any implementation of the SSP system does support object migration, then the chain mechanisms do
not have to be changed to work. The implementation presented in this thesis (Chapter 9) does not support object migration.

Figure 8.2 shows an object ‘r’ in space ‘A’ referring to an object ‘t’ in space ‘B’. This reference is made via the chain \( t_{B} \rightarrow b \). In Figure 8.2 (ii), the object ‘t’ migrates from space ‘B’ to space ‘C’. This causes the chain to extend to \( t_{B} \rightarrow b \rightarrow t_{C} \rightarrow c \).

A migration protocol which works in concert with SSP chains has been implemented in the Emerald system [Black et al., 1986], documented in [Jul et al., 1988]. It works in the following manner:

- freeze the state of the object,
- serialise state,
- copy it to new space,
- set-up the final SSP chain link,
- commit the migration in the source space by updating all pointers to the object to point to the new stub,
- unfreeze the state.

8.3 SSP Chain Invariants

The SSP protocols described in this chapter have the following invariants:

**Inv 1.** For every remote reference, there is an uninterrupted SSP chain from the primary source (the referent) to the ultimate target (the referred).

This is the garbage collection invariant. Correctness of the garbage collection protocols rely on this invariant. The collectors will collect any stub or scion which violate this invariant.

**Inv 2.** Every stub has a matching scion. A matching scion is one which is identified in the strong locator of the corresponding stub.

The fault-tolerant character of our protocols is based on this invariant.

**Inv 3.** A scion matches at most a single stub.
This invariant relaxes the liveness invariant of a garbage collector. If the system maintained strict consistency all the time, then each scion will be matched by a stub at all times. The Cleanup Protocol occasionally tightens this invariant, thereby collecting garbage which begins retaining due to the relaxing of the liveness property.

**Inv 4.** The weak location of a stub identifies a scion that may also be reached by traversing the strong chain of that stub.

This is the invariant that guarantees that weak locators do point to the target, i.e. not pointing a scion that is not part of the strong chain. Also, because of Invariant 1, the scion pointed to by the weak location will not be collected. Invariants 1 and 4 constitute the safety invariant for communication: the Invocation Protocol relies on the fact that the scion pointed to by the weak location will be present when a method invocation is performed.

### 8.4 Sub-Protocols

#### 8.4.1 Layering of the Protocols

The SSP chain system is composed of four protocols. Three of these protocols are mutator protocols responsible for the exchange of method invocation messages, and one of the protocols is a garbage collector protocol, sub-divided into local and remote components. Figure 8.3 shows the relationships between the protocols and the application which uses them.
As can be seen from the figure, the application sits on top of two object management layers provided by the system. The application interacts with the upper layer of the system. The bottom layer is independent of object semantics, structure or programming language, and is described in [Shapiro et al., 1990]. This bottom layer propagates reachability information gained from the upper layer.

The upper layer is language specific. It consists of the language run-time, and system support to interact with the lower layer. This layer has the local heap management mechanisms such as the allocator and the local garbage collector. The SSP system also exposes the method invocation functions in this layer.

The upper and lower layers interact via the shared tables, which hold the lists of incoming and outgoing references between this space and other spaces.

8.4.2 Services Provided by the Protocols

The four protocols interact by creating, updating and deleting stubs and scions. Figure 8.4 shows the way that the four protocols and the assumed local garbage collector interact.

The following is an overview of the function of each of the protocols:
The Transport Protocol specifies which messages are accepted or discarded.

The Presentation Protocol gives the marshalling and unmarshalling services for references as well as primitive types of the language.

The Invocation Protocol provides services to locate an object from a reference. It also provides short-cutting mechanisms for indirect SSP chains.

The cleanup protocol sends and receives messages which propagate reachability information. From this information, it eliminates scions which have no associated stubs, i.e. chains for which the remote references are no longer needed.

### 8.5 Transport Protocol

As mentioned in section 8.1, when introducing the SSP chains, the assumptions from the underlying network are weak. This allows the design to be portable to networks of very different characteristics (see Section 6.2.2 for communication semantics and distributed algorithm design). The only assumption made is that messages are not corrupted, thus precluding arbitrary failures. Duplicate, lost and delayed message delivery is acceptable, although most applications will demand a better message delivery system.
Communication between mutator processes in different spaces occurs via messages marked with timestamps from the timestamp generator of the space that the messages originate from.

Consider a message sent from space 'A' to space 'B'. On arrival at space 'B', the timestamp of the message is compared with the value held in threshold\(_A[B]\). Only messages with timestamps greater than the threshold value are accepted. This eliminates certain race conditions which are detailed in later sections.

If a message is received from a space which did not send a message prior to that message, then a new entry is made in the vector. Entries cannot be removed from the threshold table even if all stubs to that space are discarded. The only way an entry can be removed is if the space is guaranteed not to interact with other spaces.

Ordering on messages is imposed by using call-response pairs of messages or values from the threshold table. But this ordering is only imposed when out-of-order message processing can violate invariants.

The transport protocol methods to send and receive messages are detailed in Chapter 9, sections 9.5.

### 8.6 Presentation Protocol

This protocol is responsible for the marshalling and unmarshalling of message parameters and method invocations or results of invocations.

A stub / scion generator is used to automatically generate stubs that have an interface that is the same as the object to be made public. The interface has the same methods with the same signatures and return types.

The stubs generated method simply marshalls all the arguments of the method, as well as identifier for the method call that the scion will recognise. The method then sends this to the scion, which is located by using the scion name and space identifier from the stub. After sending the message, the stub does a blocking receive. Upon receipt of the message, the stub unmarshalls the result.

Symmetrically, the scion has a single method which waits for a message and unmarshalls that message. According to the method identifier in the message, the scion method proceeds to unmarshall the arguments. It then invokes a method to the local object using the unmarshalled arguments. It then marshalls the method's result in a reply message, which it sends back to the stub.

More details on the stub, scion and the stub/scion generator are presented in Chapter 9, sections 9.4.4 and 9.4.5.

The presentation protocol provides the service that finds a scion for a local pointer (to an object or a stub). It returns a locator for the given pointer. It looks for a scion from the current instances in that space, and returns a new instance if it cannot find any of the same type. In both cases, they are timestamped with the message being marshalled.

At the method invocation target, or the receiver space, unmarshalling references needs a local pointer given a remote reference. This is a service provided by the Presentation Protocol as well. The service first searches for a matching stub from the stubs instances in this space. If none is found then a new stub is created. If one exists, the timestamp is updated as well as the weak location found in the message.
Since marshalling and unmarshalling of messages is necessary for any message passing distributed object oriented system, the overheads of using SSP are negligible, being the cost of creating stubs and scions. Ensuring uniqueness between a pair of scions has a cost because it requires additional indexing and searching. But this ensures that stub creation is idempotent. Also, deletion of a stub permits the corresponding scion to be discarded without worrying about other stubs keeping a reference to that scion.

The construction of the stubs and scions is conservative. An SSP chain may be constructed without knowing if it will be used for invocation or not. Also, stubs are always created after scions. For example, when exporting a reference to an object for the first time, the scion is created and its locator is marshalled in a message. When the scion reaches the recipient, the unmarshalling of the reference leads to the creation of the stub. If the message is lost on the way, then the scion will be created without a stub. This will cause a local collector to collect it.

Stub and scion code can only allocate objects, while de-allocation is performed by the collector. The local collectors remove unreferenced stubs, while the scions are removed by the Cleanup Protocol.

8.7 Invocation Protocol

This section discusses the design of the mechanism used for method invocation on remote objects. The next section discusses issues when remote objects have migrated or when exporting of references by clients cause indirect chains. The indirect chains cause inefficient communication and a solution is to short-cut strong chains to cope with reference graph updates and reduce floating garbage in the system.

8.7.1 Basic Call-Response Mechanism

An SSP chain is used to invoke a method on a remote object. A call-response protocol is used for all method invocation types, including ones with no results. The initial specifications were to use one-way messages, but this lead to problems with implementation.

A single timestamp is used for the entire marshalling of a message. The invocation protocol loops, sending a call message, until it receives a reply message. This is because a method invocation may not succeed the first time because of a location exception, which is raised if the weak locator is not exact. The need to loop until direct invocation succeeds is in case there is a method call on a highly mobile object which changes location between an exception message and a subsequent call message.

A location exception contains a better weak locator. This is used to update the calling stub's weak locator and the method invocation is retried with this new weak locator.

When a reply message is received, i.e. the remote method invocation has succeeded, then the invocation mechanism exits the loop.

If the reply message contains a locator, then a strong short-cutting has occurred. This necessitates the installation of a new stub.
On the receiver side of the call, if the target object for the method call is not local (this is determined by looking at the location in the message), then the message is passed along the weak chain without unmarshalling. If the target is local, then the message is unmarshalled and a local method invocation made. The results are marshalled and sent back to the calling space.

If the target object is local, but the message is forwarded, then a location exception is sent back to the caller with an up-to-date scion name. This scion will be used in the retry (see above). As a side effect, a strong chain short cut may be made (see Section 8.8.2), and this may create a new scion.

### 8.8 Indirect SSP Chains

A remote reference may be sent as an argument to a method call by one client of an object to a new space, making that new space a client. A number of such method calls create long chains with many pairs of stubs and scions. The liberal use of indirect chains supports passing of references to objects in remote method calls without informing the owner, while also being useful for maintaining the invariants for reachability. But the communication of a method call from source to target can be inefficient through a long chain because it involves exceptions and remarshalling of the message at every hop. Also, the objects in question exhibit poor locality, e.g. method invocation between two remote objects not only involves the two spaces in which they reside, but also the spaces in which their chains traverse.

Another big disadvantage of the use of indirect chains for method invocation is that a method invocation along a chain which itself has a reference to another object causes a new indirect chain for that reference.

![Diagram](image-url)

(i) 's' has an indirect chain to 't' strong: \( t_A \rightarrow b \rightarrow t_B \rightarrow c \rightarrow t_C \rightarrow d \) and weak chain: \( t_A \rightarrow c \rightarrow t_C \rightarrow d \)

(ii) method invocation by 's' on 't' with 't' as an argument (actually remote ref by weak chain above) results in a new chain being created.

Figure 8.5 Invoking a method via an indirect chain without short-cutting results in a new chain
This is seen in Figure 8.5 which shows the result of invoking a method with an argument which is a reference via an indirect chain without shortcutting. Figure 8.5(i) shows an object 's' in space 'A' which holds a remote reference to object 't' in space 'D'. As reference is an indirect chain, the strong chain is given by 
\[ t_{A} \rightarrow b \rightarrow t_{C} \rightarrow d \] 
and the weak chain: 
\[ t_{A} \rightarrow c \rightarrow t_{C} \rightarrow d \].

No object 's' performs a method invocation on object 't'. One of the method arguments is a reference to 't', which is a locator pointing to the stub 't_{A}'. As method invocations occur via the weak chain, the argument will cause a new chain, shown in Figure 8.5 (ii). On executing the method, object 't' may invoke methods on the argument. These will be invoked via the chain 
\[ t_{0} \rightarrow c \rightarrow t_{C} \rightarrow a \rightarrow t_{A} \rightarrow c \rightarrow t_{C} \rightarrow d \], even though the method invocations are really being performed on self! Consider the situation where the method result is the argument given. This will cause an even longer chain. It is easy to see the problems arising from such a mechanism. The next section describes a mechanism known as short-cutting to avoid situations such as the one described.

### 8.8.1 Short-Cutting Indirect Chains

The reasons mentioned above make it necessary to have weak locators which are used for method invocation. In the absence of migration, weak locators always point to the scion nearest to the target object, i.e. the scion in the same space as the scion object. The strong chain is lazily short-cut in a safe manner as a side-effect of method invocation. The obsolete stubs and scions, left after a chain is short-cut, are collected by the local collectors and the Clean-up Protocol (see Section 8.9).

A chain is always short-cut, at the latest, before the harmful long chain situation described above can occur, i.e. before allowing a remote method invocation with arguments to execute.

There are two sub-cases of short-cutting to consider. The first is when the weak locator of the calling stub is exact, i.e. points to the scion nearest the target object. The method call message proceeds directly to the target scion. Upon method execution return, a locator for a scion which is direct is returned, piggy-backed onto the reply message.

The second case is where the disastrous situation described above could occur: an inexact weak locator, i.e. it does not point to the scion residing in the same space as the target object, and the method call has at least one argument which is a reference. In this situation, the scion pointed to by the caller stub's weak locator forwards it to the target scion. The target scion does not execute a method call, but instead returns a location exception. This exception message contains the location of the target scion (which is the scion which sent this message). The caller stub receives the exception message and updates its weak location with the location found in the scion. This case now reduces to the first case mentioned above and can proceed in the same manner.

An intermediate case, in which the calling stub's weak locator is not exact, but the method call does not have any method argument, can be treated as an instance of the second case.

The next sub-sections give further details on both the cases mentioned here.
8.8.2 Short-Cutting Strong Chains

This section gives details on the short-cutting of strong chains when a method invocation is made on an indirect remote reference. Figure 8.6 (i) shows an object 'r' in space 'A' holding a remote reference on an object 's' in space 'C'. This remote reference is an SSP chain with the strong chain given by \( s_A \rightarrow b \rightarrow s_B \rightarrow c \) and the weak chain given by \( s_A \rightarrow c \). This shows that the chain is indirect. Object 'r' invokes a method \( s.f() \) on object 's' via the remote reference. The marshalled method call message is sent to the scion pointed to by the weak location: \{C, c\}. When scion 'c' receives the call message, it identifies that the sender space (space 'A') is not the space containing the scion's matching stub, which is space 'B'. This means that the method call was via an indirect chain. This causes the scion 'c' to create a new scion 'c'''. The scion 'c' invokes a method on object 's' locally. When marshalling the results of the method invocation, it includes the location of the new scion in the message 'c''' . This is seen in Figure 8.6 (ii)

Upon receipt of the message by the calling stub, the location of the new scion is unmarshalled and a matching stub is created. This stub is initialised with locator \{'C, c', C, c''\}. The calling object's pointer to the original stub 's_A' is updated to point to 's_A'''. Other references to 's_A', possibly pointers from scions

Key

- Strong and weak location same
- Strong location
- Weak location
- Local pointer

Figure 8.6 Short-cutting the strong SSP chain on method invocation
part of longer chains, are not affected. This is seen in Figure 8.6 (iii). Updating pointers is difficult to implement neatly, especially in Java.

The Invocation Protocol must be careful to deliver the message to the stub that sent the call, not the new one. The original chain, if not used by other scions, is eventually collected.

8.8.3 Short-Cutting Weak Chains

Figure 8.7 shows the effect of invoking a method on a reference which has an indirect weak locator. Figure 8.7(i) shows object ‘s’ in space ‘A’ has a remote reference to object ‘t’ in space ‘D’. The strong chain is indirect maybe because of the reference to ‘t’ being exported from spaces ‘C’ and ‘B’. The strong chain of the reference is given by \( t_A \rightarrow b \rightarrow t_B \rightarrow c \rightarrow t_C \rightarrow d \) and the weak chain is given by \( t_A \rightarrow c \rightarrow t_C \rightarrow d \). Object ‘s’ invokes a method \( t.f(arg) \), with a reference argument that is not shown in the diagram. The call message is sent to the scion pointed to by the weak location of the stub ‘tA’. Upon receipt of the message, scion ‘c’ knows that the message is for an object which is not local. This

![Diagram](image-url)

(i) ‘s’ has an indirect chain to ‘t’ strong: \( t_A \rightarrow b \rightarrow t_B \rightarrow c \rightarrow t_C \rightarrow d \) and weak chain: \( t_A \rightarrow c \rightarrow t_C \rightarrow d \)

(ii) method invocation on t with a reference argument (not shown), causes a location exception which updates ‘tA’ s weak location to ‘d’.

![Diagram](image-url)

(ii) method invocation retry on the better-weak given by the exception. This results in a new direct SSP chain piggy-backed onto the reply.

Figure 8.7 Method invocation on an indirect weak reference causes a location exception (same key as Figure 8.6)
is because the scion is pointing to a stub and not a local object. Thus the message is forwarded onto the scion pointed to by the stub that the target of scion 'c' points to, which is stub 't_c'. Stub 't_c's weak location target points to scion 'd', hence the message is forwarded onto scion 'd'. Forwarding of a call does not involve unmarshalling the message, but the message type is changed and marked as a forwarded message. Also the timestamp is updated.

Upon receipt of the destination message, scion 'd' notices it to be a forwarded message. Thus it does not invoke a local method call, but it generates a location exception. This exception generates a message which contains the location of the scion sending it, since it is the nearest scion to the target object. When the calling stub receives the location exception, it updates its weak location reference to the location contained in the exception. This is seen in Figure 8.7 (ii).

The calling stub then retries the method call on the new weak location. Assuming that the target object has not migrated between the time it sent the exception message and the time of receipt of the exception message, then this situation reduces to the one discussed in section 8.8.2. This is the case where the weak location of the calling stub is exact, but the strong location leads to an indirect chain. As explained in that section, the side effect of method invocation in such a situation is the creation of a new direct SSP chain.

### 8.9 Collector Protocol

The above sections detailed the sub-protocols which were all mutator protocols. This section details the collector protocol. The relationship between the protocol is seen in Figure 8.3.

The actions of the collector protocol are performed independently from the mutator activity. This is in order to reclaim the heap space that the mutator has requested and used, and no longer requires.

The collector protocol involves two independent activities: local garbage collection and the distributed Cleanup Protocol. The cleanup protocol interacts with the local garbage collector via operations on the stubs and scions. This relationship can be seen in Figure 8.4. The reclamation of unreachable stubs and scions is made more complicated by the possibility of lost, delayed and duplicated messages, and of race conditions.

#### 8.9.1 The Local Garbage Collector

The system assumes the presence of a local reclamation scheme of some sort. The assumptions made of such a de-allocator are those made on any de-allocator, namely that the collector obeys the safety (collect only garbage) and liveness (eventually collect all garbage) properties.

No assumption is made on the comprehensiveness or completeness of the collector. Specifically, the collector need not be able to collect local cycles.

No assumption is made on the algorithm of the collector. The collector need not be tracing or reference counting etc. The acyclic extension to the Cleanup protocol does however require the local collectors to be tracing.

The progress of the local collector governs the progress of the distributed Cleanup Protocol. Thus, if the liveness property of the local collector is relaxed
and it does not require strict consistency, then the global collector will also exhibit these properties.

### 8.9.2 Distributed Cleanup Overview

Remote references are always valid due to the invariant that states that an SSP chain is never broken. A public object can be reclaimed if there are no local references to it as well as no remote references to it. If a public object has no local references, then it is made reachable by the scion that points to it. All scions are rooted by a scion table that the Object Management layer stores.

A client object may no longer need a remote reference. It may do this by deleting the pointer to the stub at the tail of the reference chain, or the client object maybe unreachable and be collected by the local collector.

Thus a local collector may collect objects which are clients to remote objects. The reclamation of local objects destroys their pointers to the stubs that form one end of a reference chain. This makes the stubs collectable as well. Hence the local collector is responsible for the collection of unwanted stubs.

The distributed Cleanup Protocol has a thread running on every node that propagates the reachability information of scions in other spaces by keeping track of the stubs in this space.

The need to propagate this reachability information is because the stubs point to scions across spaces. This location pointers are just identifiers of the spaces in which the scions reside, and a unique scion identifier within that space. Thus if the location identifier is destroyed, there are no language semantics enforcing this information to be propagated to the appropriate scion. This has to be done by the system. One way to propagate this reachability information is to send a delete message in the finaliser method of a stub to the appropriate scion. This causes problems due to the weak network assumptions, and is discussed below.

The distributed Cleanup Protocol thread receives this propagated information from other spaces and acts on it to reclaim unwanted scions. This causes the pointers in the scions pointing either to local target objects, or to further stubs, to be destroyed. Thus it may cause local objects in the target space to be collected as well.

The race conditions that can occur within the Collector protocol are described next.

### 8.9.3 The Create-Create Race

This race condition happens due to the activity of the local garbage collector. The local collector's detector sub-protocol is responsible for the detection of garbage objects. It may find an object that is unreachable, and refers to a stub, or a stub that is unreachable, and thus collect the unreachable stub. This can happen if the remote reference that the stub represents is not needed by all objects in that space anymore.

However, due to the inconsistencies introduced by messages which cannot be delivered instantaneously, a method call message may be in transit, with arguments, one of which is a location for the stub concerned. Thus the method invocation is interested in the remote reference and once the message reaches
the target object, the remote reference pointed to by the argument will become reachable again.

Now if the stub is deleted, and this information propagates to the space which holds the matching scion, then the scion will be deleted as well. Consider a delayed message that was sent out when the reference to the now-deleted stub was valid. If this message is now received at the source space, then the stub will be recreated. This happens normally on reference duplication because all scions are created before their matching stubs. An attempt to use this stub will cause an error in the system because the stub will expect a matching scion in the target space. This is the Create-Create race condition. It can be avoided by the following mechanism.

Consider a stub 'y', in space 'A' with matching scion 'b' in space 'B', i.e. the strong locator of the stub points to a scion in space 'B'. On identification that the stub is unreachable, the finaliser method for the stub increases the value of the threshold array element for space 'B': thresholdA[B]. This will cause the transport protocol to drop messages which are marked earlier than that value from space 'B'. Thus the delayed message which needs to refer to the remote reference will now not be received.

Note that the use of a stronger transport protocol which delivers messages in FIFO order (see [Hadzilacos and Toueg, 1994] for details on message ordering, including FIFO order) would not have a create-create race. Thus the implementation presented in Chapter 9, which uses TCP as the transport layer, does not suffer from such a race condition.

8.9.4 Propagating Reachability Information

The Garbage Collection liveness condition is to eventually collect all garbage. The activity of the mutator increases garbage in the system because it dynamically updates the reference graph, thereby letting go of some references. Formally, this can be stated as relaxing the liveness property. The clean up protocol periodically strengthens the liveness property by collecting some garbage. The liveness property can be in the strong or the weak form. The strong form does not allow garbage in the system at all times. This is difficult to achieve in practice. The weakened form guarantees that garbage will be collected eventually. Thus the strong form forces that every scion has a matching stub, while the weak form tolerates some scions to be without matching stubs.

As mentioned above, the reachability of scions by their matching stubs has to be explicitly sent across spaces. Sending a delete message on stub finalisation is complicated by the following:

- A delete message could be lost. Thus the matching scion will never be collected.
- The application call message which contained the remote reference which created the stub in the first place could be lost. Thus the stub for that scion was never created and hence a delete message for that scion will never be sent out. This happens because stubs are created after their matching scions.

To counter both of the above cases, the system sends messages between pairs of spaces which exchange the reachability information repeatedly. This guards against the first type of message loss.
Also, to ensure that processing the information twice does not cause problems, the messages are made idempotent by exchanging lists of live stubs in the space instead of the recently deleted ones. This message is called a LIVE message.

The distributed cleanup protocol receives this list of live stubs which have matching scion in that space and deletes any scions which do not have matching stubs in the list.

Even this system has a problem due the asynchronous nature of messages, which is detailed below.

### 8.9.5 The Create-Delete Race

There may be a message in transit which makes a scion reachable again. This message may be overtaken by the LIVE message from a neighbouring space.

Thus a scion may only be deleted if it is not reachable and there is no message in transit which may make that scion reachable again. Timestamping messages helps here again as well.

A space 'A' will periodically send a LIVE message to some other space 'B' which has scions matching some of its stubs. This LIVE message contains:

- The set of scions names that match the stubs in space 'A' that have strong locators pointing to space 'B'.
- The threshold value in space 'A's threshold array for space 'B', i.e. thresholdA[B].

The receiving space 'B' processes the live message in the following manner to deduce which scions are unreachable from space 'A'. The following conditions need to be true before a scion is determined to be unreachable:

- If the scion name does not appear in the list obtained from the live message, AND
- If at space 'B': \( \text{scion.stamp} \leq \text{liveMessage.threshold} \)

This ensure that there are no recent messages in transit carrying the location of this scion, which would make it reachable again. Note that a message sent out by a scion will be timestamped with the scion’s stamp value.

Any transport protocol that has non-instantaneous message delivery will exhibit the Create-Delete race, unlike the Create-Create race, which can be eliminated by FIFO ordering and non-duplication of messages.

### 8.9.6 Summary of the Cleanup-Protocol

The mechanisms detailed in the three sections above constitute the clean up protocol, which is not substantial because of the invariants maintained by the SSP system when working with stubs and scions.

The stubs and scion delete operations have been made idempotent by sending lists in the live messages instead of delete instructions. Race conditions have been eliminated by ignoring messages that arrive too late, (for the Create-Create race), and never discarding scions that have recently updated timestamps (for the Create-Delete Race).
Scion timestamps protect scions from being deleted if they have a message in transit which is more recent than the live message. Stub timestamps protect against re-creation of stubs that have their matching scions deleted.

8.10 Cyclic Extension Protocol

8.10.1 Overview

The detector of free distributed cycles is an extension to the SSPC garbage collector. Spaces may elect to choose the acyclic GC without the detector extension. Spaces that choose the extension are termed participating spaces, and the rest are non-participating spaces. Only cycles lying entirely within participatory spaces are detected.

The algorithm is based on date (which is the same as timestamps, but the use of the word distinguishes it from the acyclic terminology) propagation along chains of remote pointers. The useful property of this propagation is that reachable stubs receive increasing dates, whereas unreachable stubs (belonging to a distributed cycle) are eventually marked with constant dates.

This algorithm is an improvement on [Hughes, 1985] algorithm which introduced the idea of propagation of timestamps from roots to differentiate unrooted cycles. But there are key enhancements which make this algorithm much easier to implement. The original algorithm assumed the availability of synchronous global time and a termination protocol for the computation of the time below which stubs are part of cycles.

The termination algorithm, due to [Rana, 1983] is replaced by the computation of the threshold date by a central site.

The need for global time is replaced by Lamport's logical clock [Lamport, 1978]. The development of the changes, by [Le Fessant et al., 1997] was to initially design a synchronous algorithm in which the calculation of the global threshold involved termination. Then the changes to make the algorithm asynchronous were introduced.

A threshold date is computed by a central server. Stubs marked with dates inferior to this threshold are known to have constant dates, and are therefore unreachable. Each participating space sends the minimum local date that it wishes to protect to the central server (stubs with these dates should not be collected). This information is based not only on the dates marked on local stubs, but also on the old dates propagated to outgoing references. The algorithm is has the following properties:

- **asynchronous** as most values are computed conservatively,
- **tolerant to unreliable communications**, as using old values in computations is always safe, since most transmitted values are monotonically increasing,
- and benign to the normal SSPC garbage collector as, non-participating spaces can work with an overlapping cluster of participating spaces, even if they do not take part in cycle detection.

The algorithm assumes that the local collector is a tracing collector. The project implementation, detailed in Chapter 9, was a challenge due to this assumption.
The implementation of the cyclic extension remains a proof-of-concept rather than a usable collector because of the inability to modify Java's local collector.

### 8.10.2 Data Structure Extensions

#### Stubs

Stubs are extended with a timestamp called `stubdate`. This is the time of the most recent trace (possibly on a remote site) during which the stub's chain was found to be rooted. Stubs have a second timestamp, called `olddate`, which is the value of `stubdate` for the previous trace. Thus the stub class needs two additional integer members.

#### Scions

Scions are extended with a timestamp called `sciondate`. This is a copy of the most recently propagated `stubdate` from the scion's matching stub, i.e. the time of the most recent remote trace during which the scion's chain was found to be rooted.

#### CyclicThreshold

Each site has a vector, called `cyclicthreshold`, containing the timestamp of the last `STUBDATES` (see below) message received from each remote space.

#### ProtectedSet

This is a table containing a value per space in the system. The value stored is `ProtectNow`, which is the new dates that a space wishes to protect from being collected locally in the next invocation of the local garbage collector.

### 8.10.3 New Messages

#### Message: STUBDATES

The `stubdates` from a space are propagated to their matching scions in some other space by sending a `STUBDATES` message. `STUBDATES` messages are stamped with the time of the trace that generated them. Each site has a vector, called `cyclicthreshold`, containing the timestamp of the last `STUBDATES` message received from each remote space.

#### Message: THRESHOLD

The `cyclicthreshold` value for a remote space is periodically propagated back to that space by sending it a `THRESHOLD` message. The emission of `THRESHOLD` messages can be delayed by saving the `cyclicthreshold` values for a given time in a set called `CyclicThresholdToSend` until a particular event. This also reduces the number of entries in the `ProtectedSet` table as seen below.
Message: LOCALMIN

Each site can protect outgoing references from remote garbage collection. For this, it computes a time called localmin, which is sent in a LOCALMIN message to a dedicated site, the Detection Server, where the minimum localmin of all spaces is maintained in a variable called globalmin.

LOCALMIN messages are acknowledged by the Detection Server by sending back ACK messages. Finally, to compute localmin, each site maintains a per-space value, called ProtectNow, containing the new dates to be protected at next local garbage collection. These values are saved in a per-space table, called Protected Set, to be re-used and thus protected for some other local garbage collections.

8.10.4 The Algorithm

Global Time

As stated above, the need for global time in [Hughes, 1985] has been replaced by Lamport's logical clocks. The logical clocks simulate global time at each space.

Before sending a message, the value of the clock is increased. The message is then stamped with this value and sent. On a message receipt, the local clock of the receiving space is increased to be greater than the date in the message.

Local propagation

The current date of the Lamport clock is incremented before each local garbage collection and used to mark local roots. The roots are then traversed and all stubs reachable are marked with this date.

Then, the trace starts from each incoming scion, (which is rooted as well). These scions will have a local pointer to a stub if they are intermediate links to stub-scion chains (see Section 8.8).

The stubdate for any visited unmarked stub is increased to the sciondate of the scion from which the trace began. To avoid marking stubs again, the trace from scions starts from the scion which has the lowest sciondate, and the object memory traced from each scion in turn.

Remote propagation

A modified LIVE message, called STUBDATES (see section8.10.3 above), is sent to all participating spaces in the vicinity after a local garbage collection. This message serves to propagate the dates from all stubs to their matching scions. Each scion's sciondate is marked with a date received from its matching stub. These dates will be propagated (locally, from scions to stubs) by the receiving space at next local garbage collection in that space.

As the spaces continue to invoke their local collectors, the tracing will propagate the dates either directly from local roots, or indirectly from scions from which they are reachable. In this manner, a chain of stubs and scions forming an indirect reference will have a stubdate field of its root.
8.10.5 Characterisation of Unrooted Cycles

Local roots are marked with the current date, which is always increasing. Reachable stubs are therefore marked with increasing dates. On the other hand, the dates on stubs included in unreachable cycles evolve in two different phases. In the first phase, the largest date on the cycle is propagated to every stub in the cycle. In the second phase, no new date can reach the cycle from a local root, and therefore the dates on the stubs in the cycle will remain constant forever.

Since unreachable stubs have constant dates, whereas reachable stubs have increasing dates, it is possible to compute an increasing threshold date called globalmin. Reachable stubs and scions are always marked with dates larger than globalmin. On the other hand, globalmin will eventually become greater than the date of the stubs belonging to a given cycle. Scions whose dates are smaller than the current globalmin are not traced during a local garbage collection. Stubs which were only reachable from these scions will therefore be collected. The normal acyclic SSPC garbage collector will then remove their associated scions, and eventually the entire cycle.

8.10.6 Computation of ‘globalmin’

globalmin is computed by a dedicated space (the Detection Server) as the minimum of the localmin values sent to it by each participating space.

Note that globalmin could be computed with a lazy distributed consensus. However, a central server is easier to implement (it can simply be one of the participating spaces), and local networks (where such a collector is most useful) often have a centralized structure.

The central server always computes globalmin from the most recently received value of localmin sent to it from each space. See Listing 8.1.

8.10.7 Computation of localmin’

Receive(space, LOCALMIN, gc_date, localmin) {
    if (gc_date > threshold_date[space]) {
        increase threshold_date[space] to current date;
        localmin[space]:=localmin;
        globalmin:= min(localmin[]);
        Send(space,ACK,gc_date,globalmin);
    }
}

Listing 8.1 Detection Server. The message is treated only if garbage collection date is the latest date received from the space.

localmin is recomputed after each local garbage collection in a given participating space. Listing 8.2 shows details. The Protected Sets and CyclicThresholdToSendSet are FIFO queues with methods: add, remove and atHead()
The notion of a probably-reachable stub is introduced here. A stub is probably-reachable either when it has been used by the mutator for a remote operation (such as an invocation) since the last local garbage collection, or when its stubdate is increased during the local trace.

This notion is neither a lower nor an upper approximation of reachability. A stub might be both reachable and not probably-reachable at the same time; it might also be probably-reachable and not reachable at some other time. However, on any reachable chain of remote references there is at least one probably-reachable stub for each different date on the chain. Therefore, since each space will “protect” the date of its probably-reachable stubs, all dates on the chain will be “protected”.

To detect probably-reachable stubs after the local trace, the previous stubdate of each stub (stored the olddate field), is compared to the newly-propagated stubdate. For each participating space in the immediate vicinity, a date (called ProtectNow) contains the minimum olddate of all stubs which have been detected as probably-reachable since the last local garbage collection.

The value of ProtectNow for each space is saved in a per-space set, called Protected Set, after each garbage collection. ProtectNow is then re-initialized to the current date. The localmin for the space is then computed as the minimum of all ProtectNow values in all the Protected Sets. This new value of localmin is sent to the detection server in a LOCALMIN message.

The next value of globalmin will be smaller than these olddates. All olddates associated with stubs that were detected probably-reachable since some of the latest garbage collections will therefore be protected by the new value of globalmin: stubs and scions marked with those dates will not be collected. globalmin must protect the olddates rather than the stubdates. This is because the scions associated with probably-reachable stubs must be protected against collection, and these scions are marked with the olddate of their matching stub. In fact globalmin not only protects the associated scions,
but also all references that are reachable from probably-reachable stubs and which are marked with the olddates of these stubs.

**Reduction of the Protected Set**

STUBDATES and LOCALMIN messages both contain the date of the local garbage collection during which they were sent. When a STUBDATES message is received, the per-space threshold CyclicThreshold is increased to the GC date contained in the message. The CyclicThreshold for each participating space is saved in the CyclicThresholdToSendSet before each local garbage collection.

<table>
<thead>
<tr>
<th>Receive(space, STUBDATES, gc_date, stub_set, threshold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>increase cyclicthreshold[space] to gc date;</td>
</tr>
<tr>
<td>old_scion_set := space.scions;</td>
</tr>
<tr>
<td>space.scions := {};</td>
</tr>
<tr>
<td>∀ scion ∈ old_scion_set, {</td>
</tr>
<tr>
<td>find(scion.scion_id, stub_set, found, stub_date);</td>
</tr>
<tr>
<td>if (found</td>
</tr>
<tr>
<td>if (scion.scionstamp &lt; threshold) {</td>
</tr>
<tr>
<td>increase scion.scion date to stub date;</td>
</tr>
<tr>
<td>space.scions := space.scions ∪ {scion}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

**Listing 8.3** Pseudo-code for the STUBDATES handler.

<table>
<thead>
<tr>
<th>Receive(server, ACK, gc_date, globalmin)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO_head(cyclicthresholdtosend set, (date, cyclic thresholds to send[]));</td>
</tr>
<tr>
<td>if (date &lt;= gc_date) then {</td>
</tr>
<tr>
<td>repeat {</td>
</tr>
<tr>
<td>FIFO_remove(cyclicthresholdtosend set,</td>
</tr>
<tr>
<td>(date, cyclic thresholds to send[]));</td>
</tr>
<tr>
<td>} until (date == gc_date);</td>
</tr>
<tr>
<td>∀ space ∈ spaces,</td>
</tr>
<tr>
<td>Send(space, THRESHOLD, cyclic_thresholds_to_send[space]);</td>
</tr>
</tbody>
</table>

**Listing 8.4** Pseudo-code for the ACK message handler.

Each LOCALMIN message received by the Detection Server is acknowledged by a ACK message containing the same GC date. When this ACK message is received (Listing 8.4), the CyclicThresholds saved in the CyclicThreshold-ToSend Set for the local garbage collection started at the GC date of the ACK message are sent to their associated space in THRESHOLD messages. Listing 8.4 shows that Old values in the CyclicThresholdToSend Set can be discarded, since they are smaller than those which will be sent in the THRESHOLD messages. Their corresponding ProtectNow values in the Protected Sets will therefore also be removed when the THRESHOLD messages is received.

Older values (for older local garbage collections) in the CyclicThresholdToSend Set are discarded (This is perfectly safe. When a space receives a THRESHOLD message it will perform all of the actions that should have been performed for any previous THRESHOLD messages that were lost).
When a CyclicThreshold date is received in a THRESHOLD message, all older ProtectNow values in the Protected Set associated with the sending space are removed. (See Listing 8.5) These values will no longer participate in the computation of globalmin.

```
Receive(space,THRESHOLD,cyclic threshold) {
    FIFO_head(protected set[space], (protect now, gc date));
    while (gc_date <= cyclic_threshold) {
        FIFO_remove(protected set[space], (protect now, gc date));
        FIFO_head(protected set[space], (protect now, gc date));
    }
}
```

Listing 8.5 Pseudo-code for the THRESHOLD message handler.

The lastest local collections are as follows: The olddate on a probably-reachable stub is protected by a ProtectNow in a Protected Set. It will continue to be protected for a certain time, until several events have occurred. The new stubdate must first be sent to the matching scion in a STUBDATES message. From there it is propagated from by a local trace to any outgoing stubs (new probably-reachable stubs in that space will be detected during this trace). The new localmin for that must then be received and used by the detection server (ensuring that the olddates on the newly detected probably-reachable stubs are protected by next values of globalmin). After this, the ACK message received from the detection server will trigger a THRESHOLD message containing a Cyclicthreshold equal to the GC date of the STUBDATES message (or greater if other STUBDATES messages have been received before the local garbage collection). Only after this THRESHOLD message is received will the ProtectNow be removed from its Protected Set.
CHAPTER 9

Implementation

This chapter describes the SSP chains distributed object oriented system are presented. The software architecture of the implementation is detailed and an outline of some of the mechanisms is given. Implementation details not covered in the previous chapter are given. Problems encountered while implementing the system are also presented. Some of the issues regarding the design of the system were unearthed during implementation.

In the project outsourcing document, my plans regarding the implementation phase of the project were to understand an open source distributed object oriented system, and to implement a distributed garbage collector on that system. Reasons for not taking that path are first given.

The implementation of the SSP chains system in Java is then detailed. Some of the main components of the systems are detailed.

Distributed object-oriented systems rely on remote references to objects in other nodes to effect distributed computation. This is done by invoking methods on remote objects. But initially, where does a local node get the first reference to a remote object from? This requires a well known object with known methods to be present at a well known host at all times. Another solution is to use a registry which holds information about the location of objects. Objects can then register themselves to this registry and allow others to get remote references to them. An implementation of a primitive registry has been accomplished to allow the distributed object-oriented system to boot-strap the first reference.

I then present the acyclic collector as an extension to the system.

A graphical front-end to the Object Management layers was developed in an attempt to show what happens when remote references are created, duplicated and destroyed. Details of this are given as well.

9.1 Other Implementations

ObjectSpace Voyager

This 100% pure Java library can be found at [Voyager]. It allows transparent remote method invocation and remote object creation. It also allows object migration and agent capabilities by allowing executing objects to migrate and resume execution.

It uses a mechanism similar to Network Objects [Birrell et al., 1994]. It manages public objects by attaching a lifetime to each public object. The lifetimes are user defined. The public object is collected at the end of the lifetime. This is not automatic memory management because the programmer still has to specify the lifetime. In practise, most programmers end up specifying a lifetime of forever, which means that the object should never be collected.
It is not possible to extend this library because it is not an open source licence, just a free use licence. It is a CORBA compliant ORB though, although I have not tested this capability.

**Java RMI**

The core Java API (1.1 and Java 2) has a remote method invocation mechanism that is similar to Voyager.

RMI only works for homogeneous Java Distributed Systems. An advantage is that it has a distributed reference counting algorithm and interface support for notification of remote reference creation/deletion. The collector algorithm is based on the work of Network Objects, by [Birrell et al., 1993].

**OmniORB 2**

The use of a CORBA compliant ORB solves the interoperability requirements of the project, but as mentioned before, as Garbage Collection is not specified in CORBA, there is no API or mechanism for interaction and notification. Thus an open source ORB may be adapted to provide remote reference notification to the global collector.

Omni ORB version 2, which is a CORBA version 2 compliant ORB with C++ bindings. It is available with full source under the [GNU] Licence from [ORL].

**9.2 Why Re-Invent the Wheel?**

As mentioned above, my initial plans were to understand an implementation of an open source distributed object-oriented system at source code level and then to implement a distributed collector for that system. I spent over four weeks before Christmas and some time after Christmas trying to understand various systems. I give details below.

**ORBAacus**

This is a very popular Object Request Broker (ORB). It was formerly known as OmniBroker, and is available free for non-commercial use with full source code. It is developed by [OOC]. It has two implementations, one in Java and another in standard C++, which has been ported to most flavours of UNIX, Windows, NT and continues to be ported to new operating systems. There are efforts underway of porting it onto embedded operating systems. It supports the full CORBA 2.0 standard, with an Interface Repository, Naming Service, Dynamic Method Invocation, it offers multiple concurrency models.

My rationale for using such an ORB was to be able to implement a distributed collector for the Java implementation and then implement one for the C++ implementation. This would show the interoperability of the collector.

Understanding approximately 30 thousand lines of code was a challenge. I was also learning about CORBA's design principles by reading the OMG CORBA specification alongside the understanding of the code. The following are some of the issues:

- CORBA does not have specifications for a garbage collector. The OMG have looked into this issue, as documented in [OMG RFP, 1998]. The document is
a request for proposal seeking design ideas from the members of the OMG. Their need to use a garbage collector is entirely different. Because they see a need for objects to have a very long lifetime, the garbage collection algorithm would select which objects to persist onto storage so as to reduce memory requirements of the nodes of the system. The problem of managing the persisted storage is not discussed.

- Understanding code which has been optimised for efficiency is difficult. Also, when trying to understand a system to implement a proof of concept garbage collector, mechanisms for binding, security and dynamic method invocation all get in the way and make it more difficult to understand the code.
- The system does not come with any design documentation and is sparsely commented, though well written.
- I would recommend the use of ORBacus as an ORB for a distributed system, but attempts to modify it are too time consuming. Also, having joined a newsgroup, I found that there were groups which were concentrating on just porting the system and that was taking them months. I quickly realised that I was too ambitious.

JacORB

This is another free CORBA 2.0 ORB implemented in Java. This is a simple system implemented to aid the understanding of the design principles of an ORB. It is developed by Gerald Brose and Reimo Tiedemann, of the Freie Universitat, Berlin, [Brose, 1998]. The reasons for not using this implementation are similar to the ones above, and also that Dr. Matthias Radestock, of TECC, advised me against the use due to the implementation being unstable.

Implementing SSP Chains

After implementing initial test programs in the above systems, I decided against using the systems because mapping a distributed collector algorithm to CORBA was difficult as most of the designs were closely coupled with the distributed systems which ran with them. My choice to implement the SSP chains in Java was because the SSP system is well documented in [Shapiro et al., 1992]. Also the collector algorithm meets most of the requirements of the project (discussed in Chapter 8).

9.2.2 Choice of Java for Implementation

The reasons that I chose to implement the algorithm in Java are somewhat different from the standard reasons for choosing Java to implement projects. The standard reasons are the "write once, run anywhere" promise and the ability to run applets which can be embedded in web pages (the common buzz-word: "Internet Language"). My reasons are briefed below:

Existence of a local garbage collector

As all of the algorithms described in Chapter 7 rely on the presence of a local collector to do the actual heap reclamation, this is an important requirement. C [Kernighan and Ritchie, 1978] and C++[Stroustrup, 1991] have manual memory management. Although the [Boehm-Demers-Weiser] collector for C/C++ is used
in many systems, it is a conservative collector and still not as transparent to use as Java’s collector. More details are given in the local collector section below.

**Java's Local Collector**

Interacting with the garbage collector in JDK 1.1 is not possible. It is only possible to schedule a collection by invoking the `System.gc()` method.

In Java 2 (formerly JDK 1.2) however, limited interaction with the collector is possible by an API known as the Reference Objects API. It is possible to have a varying range of pointer strengths, such as weak references. Also interested listeners can be informed when an object becomes garbage and is ready to be collected.

**A Collector for C++**

The [Boehm-Demers-Weiser] collector is a conservative garbage collector for the C language. It is based on the work presented in [Boehm and Weiser, 1988] and [Boehm and Chase, 1992]. Extensions have been made by [Russo, 1991] for use in C++.

The collector is conservative because it does not receive any co-operation from the compiler or runtime to find out when objects become garbage. Also, in case of pre-compiled libraries, it does not have information on the location of objects in memory. In a language like C++, which has very little runtime information in memory because of very little dynamic type-checking and efficiency, it is difficult because every memory word has to be treated as a potential word.

Having looked at the code, I will find it a considerable challenge to interface a global collector to it. This is because I have very little prior experience in understanding complex C software.

**Run Time Type Identification**

This feature simplifies many issues implementing a distributed object-oriented system, such as finding out whether a pointer is pointing to an object or a stub for that object etc. There is no run time information available for objects in C++. Again, although Stroustrup himself has implemented a run time type system library for C++, documented in [Stroustrup and Lenkov, 1992], Java's run-time type identification is part of the language, not an addition, so is easier to use.

**Reflection**

This feature allows run time inspection of the properties of a class, its instances. The following is a list of some of the features of reflection:

- the inspection of the number of methods,
- the method signatures, argument types and argument order,
- the number, type, names and accessibility of fields,
- ability to create instances,
- ability to invoke methods on instances,
- ability to set values in fields and get values from fields.

The implementation of the stub and scion generator uses reflection heavily. It has been made a lot easier using reflection than if parsing the source code of a class (see section 9.7).
9.3 System Overview

The SSP Chains has been implemented in Java according to my interpretations of the model described in [Shapiro et al., 1992]. A space in the Java implementation refers to a virtual machine. As multiple virtual machines can be running on the same host, it allows more than one space per host.

The layering described in Chapter 8, Figure 8.4 for the design is followed in the implementation. Applications are not aware of the location of objects and thus location transparency is provided by the system. A number of threads run to perform the Object Management functions described in the design.

The model in [Shapiro et al., 1992] describes a monolithic stub which is responsible for marshalling, location of remote references and communication tasks. In the implementation, the communication tasks are implemented by a separate helper class. The location of remote references is implemented by a separate helper class. Also, the stub and scion functionality is divided into abstract super-classes and class specific sub-classes which are generated by the stub/scion generator.

9.4 System Classes

This section details the data structures used in the implementation. The data structures used to identify spaces, for locating scions, for the stub and scions are all detailed in this section.

9.4.1 Spaces

In the Java implementation, a space corresponds to one virtual machine. As more than one virtual machine can easily run on a normal workstation, the space needs to be identified uniquely within the same host name as well. As this implementation uses TCP, a space is uniquely identified by the host name or equivalently, the IP number of the host, and the TCP port number.

There is no reason why the existence of multiple spaces within a virtual machine is a problem. The current implementation is ill-suited to this because the ssp.system.SSP class (see Section 8.2.1), which initialises all the data structures needed for object management is a well known static class. The modification to enable multiple spaces per virtual machine is not difficult though. It just requires making the helper class mentioned non-static and another well known class to manage instances of this class by unique space identifiers.
With regards to the design choice of space naming, discussed in Chapter 6, section 6.2.5, the space naming used here is not at one end of the axis, (see Figure 6.1 for details) which is the classical case. This is because each space does not have only one thread of control associated with it. If the object management service threads, e.g. those used for transport are not included, then the default instantiation refers to that, but creating a new thread changes that.

The `ssp.system.SpaceName` (Figure 9.1) class is used to identify a space uniquely within the system. It consists of the IP name and address stored as a system class `java.net.InetAddress` and the TCP port number.

The class overrides `java.lang.Object.equals(Object)` and `hashCode()` methods to provide testing for equivalence.

The class has numerous constructor methods to encourage setting the identifier variables on initialisation, but it is not immutable, i.e. allows later changes. This is for efficiency reasons when changing a space identifier. For example, when a chain short-cut occurs, creating a new instance of SpaceName to replace the current one just for a change in the location will be more costly in terms of time than changing its state. Having said this, an immutable class would be safer for a large implementation, but because this is not substantially large, it suffices.

### 9.4.2 Locations

An instance of `ssp.system.Location` (Figure 9.2) identifies a scion uniquely within the system. It is used within `ssp.system.Locator` in stubs and in all cases where a scion needs to be identified. Thus it serves to be a remote reference in the system.

**Figure 9.1 UML Class Diagram: ssp.system.SpaceName**

The `ssp.system.SpaceName` (Figure 9.1) class is used to identify a space uniquely within the system. It consists of the IP name and address stored as a system class `java.net.InetAddress` and the TCP port number.

The class overrides `java.lang.Object.equals(Object)` and `hashCode()` methods to provide testing for equivalence.

The class has numerous constructor methods to encourage setting the identifier variables on initialisation, but it is not immutable, i.e. allows later changes. This is for efficiency reasons when changing a space identifier. For example, when a chain short-cut occurs, creating a new instance of SpaceName to replace the current one just for a change in the location will be more costly in terms of time than changing its state. Having said this, an immutable class would be safer for a large implementation, but because this is not substantially large, it suffices.
Its state consists of an instance of SpaceName to uniquely identify the space in which the scion is residing, and an integer to uniquely identify a scion. The model discussed in [Shapiro et al., 1992] refers to this as the scion name. Using an integer suffices to uniquely identify the scion with minimal overheads when performing equality tests.

The class overrides `java.lang.Object.equals(Object)` and `hashCode()` methods to provide testing for equivalence.

Although `Location` is not immutable, there are no methods to access the state of the class. Instead the state is left public. This is because the number of times the states needs to be accessed is very frequent and assessor methods would make the system inefficient.

### 9.4.3 Locator

A stub contains an instance of the class `ssp.system.Locator` (Figure 9.3). This class provides the storage for the remote references (`Location`s) for the strong and weak chains to a stub.

It's state consists of a strong `Location` and a weak `Location`. The class overrides `java.lang.Object.equals(Object)` to provide testing for equivalence. Again the state is left public, but accessor methods are provided.

---

**Figure 9.2 UML Class Diagram: ssp.system.Location**
9.4.4 Stub

An instance of `ssp.system.Stub` (Figure 9.4) holds an instance of a `Locator` as well as a timestamp value. It also stores state to determine whether it is a forwarder, i.e. the target of a scion instead of a normal object.

Methods for the Invocation Protocol are `redirect()`, which short-cuts an indirect chain, and `redirectWeak()`, which short-cuts the weak chain (see Section 8.8.3).

It has factory methods (see the Factory Pattern, [Gamma et al., 1995]) for the installation of instances of sub-class stubs for a remote scion. These methods are `forSvcName()`. Factory methods are those which instantiate other classes on behalf of some requesting client. Every sub-class stub instantiated using these factory methods has its class cached in the `locationCache` instance of the `NameLocationTable`, which is a class cache. Also, because all stub sub-classes get instantiated with the same arguments, the arguments are cached as well.

The Transport Protocol method is `ssp_call()` which is responsible for invoking a remote method call. This method is detailed in section 9.5.

The class overrides `java.lang.Object.equals(Object)` and `hashCode()` methods to provide testing for equivalence.
Figure 9.4 UML Class Diagram: ssp.system.Stub showing a subclass TestImpl Stub

Figure 9.5 UML Class Diagram: ssp.system.Scion, showing a sub class Test2Impl_Scion and the Class cache used by the factory functions.
9.4.5 Scion

An instance of `ssp.system.Scion` stores a unique scion name, which in this implementation is an integer. It has an instance of `SpaceName`, which identifies the space in which its source stub is residing, and a local pointer, which in this implementation is of type `Remote`. It also stores the timestamp of all the messages that it sends out.

An abstract method, `ssp_call()` is defined, thus all sub-classes of Scion, which are the classes generated by the stub-scion generator, have to implement this method.

It provides factory methods for the installation of sub-class scions for a remote object. In the same manner that the stub class caches classes it instantiates, the Scion class holds a static reference to an instance of `NameClassTable` called `scionCache`, which holds references to classes, thereby caching them.

9.4.6 Space Manager

The `ssp.system.SSP` class serves as a helper class for a lot of functionality within the SSP system. It has functions that are part of the Transport protocol and functions which are part of the Invocation protocol. It also serves as a manager for the data structures that need to be stored when managing objects, namely tables of the stubs and scions, the threshold vector and threads for transport functions. The space manager functions are described in this section.

The SSP class is static, i.e. cannot be initialised. This choice has been taken to make the class well known within a space. All objects can refer to the space without having to maintain explicit references to it.

Space Identification

An instance of `SpaceName` uniquely identifies the space of the host. The function `SSP.init(int portNumber)` initialises the space identifier with the host's IP name/address and the given port number, if an earlier method call has not already done so. This sets up the space to listen for incoming messages on that port. (see section 9.5 for more details).

Service Scions

Certain objects can register themselves as services via the method call `SSP.localService()`. This allows local objects to have scions which remote objects create stubs to before invoking method calls.

In the current implementation, sixteen service scions are allowed to register as local services per space. There are no current services that require the use of such registration, but it provides a way for spaces to communicate without bootstrapping via the registry.

Stub / Scion Management

The `SSP` class creates singleton instances of the following for stub and scion management:

`ssp.system.StubTable`
ssp.system.ScionTable
ssp.system.ScionNameTable
ssp.system.ServiceScionTable

Each of these are sub-classes of java.util.Hashtable, which provides a collection class to store and retrieve elements via keys.

The above classes are created using the singleton pattern, i.e. a second instance creation returns a reference to the first and only instance. See [Gamma et al., 1995]

The ScionTable instance stores references to scions in this space. Before a new scion is created on exporting a reference or some other invocation operation, this table is referred to see if there exists an instance of the scion already. The entries in the table are indexed by instances of ssp.system.PtrSpace, which is a pairing of a local pointer and a remote SpaceName. This enables the system to find a stub-scion pair.

The instance of StubTable stores instances of stubs indexed by the Location of the space in which their corresponding strong scion lies. This allows locating scion which match stubs in remote spaces. A method to retrieve all stubs which have a corresponding scion in a specified space is included. This is used to construct LIVE message for garbage collection.

The ScionNameTable allows scions to be accessed by their names, which in this implementation means an integer. This is required by the Invocation Protocol. This accesses a scion by looking at the scion name referred to by the weak location of the stub.

![Diagram of data structures](image)

Figure 9.6 All the subclasses to java.util.Hashtable. These are singleton system data structures

### 9.4.7 Threshold Table

This table is indexed by SpaceNames and stores integer values, which are timestamps below which messages should be dropped, when arriving from those spaces. This is used by the Transport Protocol.

### Caching Classes

In Java, classes are first class objects, i.e. they can be treated as instances. The SSP class pre-loads and caches the commonly used classes. For example, the ssp.system.Stub class is cached so that referring to it is faster. Also, because
all stubs sub-classes have a constructor with three arguments, the arguments are pre-instantiated (two locations and an integer) and cached.

9.5 Transport Protocol

The Soul system, which is a substantial system based on SSP chains, documented in [Plainfossé, 1994] uses multiple transport protocols to gain efficiency, e.g. using TCP for connection oriented messages while UDP for periodical exchanges.

The implementation presented in this thesis only uses TCP for simplicity reasons. This is because the emphasis is on the design and implementation of a distributed garbage collector.

This section details the transport protocol. The SSP class instantiates the necessary objects which manage the transport layer.

9.5.1 EndPoint

The class ssp.system.EndPoint encapsulates a plain socket. Two spaces communicate via two EndPoints, one at each end of the communication link. The state contained in an instance of EndPoint is an instance of java.net.Socket, which is the TCP socket used for communication, instances of the input and output stream from the socket. It also contains an instance of SpaceName to identify the space that it is connected to.

Methods to receive messages on incoming streams and to send to the output stream via the socket are provided. These form the basis of the transport layer.
The class contains a static reference called `connections` to `ssp.system.EndPointTable`, which is a hashtable referenced by `SpaceNames`. Thus it holds all the connections that this space has to other spaces.

### 9.5.2 ServerEndPoint

An instance of `ssp.system.ServerEndPoint` encapsulates a server socket: `java.net.ServerSocket`, which is a TCP server socket. This is used to listen to a published port for client communication. When a client requests communication, an instance of a plain socket is created and returned. The client then talks to the server via plain sockets encapsulated in instances of `EndPoint`.

![UML Class Diagram](image-url)
9.5.3 SpaceDaemon

SSP.init() causes the instantiation of the class ssp.system.SpaceDaemon. This is a sub-class of java.lang.Thread. The class is effectively a multithreaded server which handles messages from spaces wishing to communicate to this space.

The thread spawns a server socket by creating an instance of ServerEndPoint. It then loops forever as follows:

- block until client requests communication. This returns an instance of a plain socket which is encapsulated in an instance of EndPoint.
- Then receive a message and unmarshal its SpaceName,
- Update the table of end points, which is an instance of EndPointTable
- Add new element to the threshold vector,
- Create a new instance of ssp.system.MessageDaemon to handle communications from this space from this point on. Add this instance to the list of message daemons.
- Repeat loop, so go back to listen for a new client.
An instance of `ssp.system.MessageDaemon` is responsible for the receipt of messages from an `EndPoint` (which is a plain socket). As seen from the description of `SpaceDaemon` in section 9.5.3, an instance of `MessageDaemon` gets created when communication between spaces has been established.

The state held in a message daemon is the instance of the `EndPoint` that it is listening on and the `SpaceName` that it is connected to.

The first receipt of a message from a space causes a new instance of `MessageDaemon` to be created. This thread then runs in a loop forever as follows:

- Block on a message receipt on the input stream of the plain socket. The message is received and constructed into an instance of `ssp.system.Message` by a constructor of `Message` with an input stream as an argument.
- Get the timestamp of the message and compare with the threshold value from that space, if the message is timestamped older than the threshold value, then discard the message, else,
- Obtain an instance of `ssp.system.MessageDispatch` and delegate the delivery of the message to this thread. The instance of `MessageDispatch` is obtained from a thread pool manager, `ssp.system.MessageDispatchPool`.

The interaction between classes from the point of receipt of a message by an instance of the `MessageDaemon` class to creation of a `MessageDispatch` thread is shown in Figure 9.10. Note that the `Message(InputStream)` constructor is not shown in full detail in this figure, but is detailed in Figure 9.12, in the case where the constructor has an input stream, i.e. receives data from a socket.
9.5.5 MessageDispatch

An instance of `ssp.system.MessageDispatch` is responsible for the delivery of the received message to the higher transport layers. It is a Java thread. The reason why this task is delegated to a separate thread is to keep the `MessageDaemon` thread ready for message receipt as delivery may possibly take long. This is because the message is unmarshalled on a need basis by the code.

![Figure 9.10 UML Sequence Diagram: Receipt of Message by Message Daemon Thread](image)

![Figure 9.11 Delivery of Message by MessageDispatch thread. SSP.transportReceive handles different message types in different manners.](image)

The message dispatch thread first delivers to the `SSP.transportReceive()` method. This method then delivers to the scion that the message is intended for.
if the message is a method invocation call, or a forward message. In case the
message is a reply message or an exception message, then the message is “given”
to an instance of \texttt{ssp.system.Reply}, which has the sending thread waiting on
its notification.

9.6 Presentation Protocol

This section details the marshalling and unmarshalling of messages. The stub /
scion generator is also detailed in this section. Wire representations of messages,
message formats are presented as well.

9.6.1 Message

This section presents the \texttt{ssp.system.Message} class which contains methods
for marshalling and unmarshalling all primitive message types, as well as
references.

An instance of a message is constructed by the stub sub-class method which are
generated by the stub generator. The method marshals arguments into an
instance of \texttt{Message} specifically for the method invocation that it is performing.
Thus the entire marshalling is specific for the class that the method invocation is
taking place for.

Figure 9.13 shows the class diagram for the \texttt{Message} class. The four
constructors are used for different situations. The constructors with the type
argument are used for constructing outgoing messages so the output buffer
needs to be written to via the marshalling methods. The constructor with the
argument as an input stream on the other hand, is used for receiving messages.

![Class Diagram for Message](image)

Figure 9.12 Message construction if received from an input stream
Marshalling Details

Marshalling and unmarshalling primitives is achieved by writing or reading to/from the message buffer via a `java.io.DataOutputStream` or `java.io.DataInputStream` if reading in.

Figure 9.13 UML Class Diagram: ssp.system.Message, showing containment of other elements
Strings

Strings are marshalled by first marshalling the length of the string as an integer. Then the string is marshalled as a sequence of bytes. This can be seen in Figure 9.14.

SpaceNames

Instances of `ssp.system.SpaceName` are marshalled by getting the byte representation of the IP address, then marshalling the port number as an integer. This can be seen in Figure 9.15.

Locations

Marshalling instances of `ssp.system.Location` is done by marshalling a location’s `SpaceName` first, as shown above, and then marshalling the `scionName` as an integer.

Locators

Marshalling instances of `ssp.system.Locator` is done by marshalling the strong `Location` and then the weak `Location`.

Objects

All public objects in the system have to implement the interface `ssp.system.Remote`, which requires the object to implement methods which reveal if the object wishes to be remotely referenced or passed by value. When an argument to a method is encountered, the object is queried to reveal whether it wants to be passed by reference or value. The following is the manner in which it is marshalled:

1. Query if marshalling by reference, i.e. invoke method on object `Object.passByReference()`, and write a code to the output stream to state that. (current implementation is a zero for passing by reference and a one for passing by value).
2. If passing by reference, then store the position of the output buffer stream in an instance of `ssp.system.PtrPosition`, which is a pairing of the local reference and an integer to record where in the output buffer the reference is to be written. Then write a dummy instance of a `ssp.system.Locator`. This reserves the space needed to marshal the actual remote reference. Also,
marshal the name of the class that the remote reference belongs to as a string. This will be used to install the appropriate scion at the target space.

- If marshalling by value, then write the entire object state out to the stream using the java.io.ObjectOutputStream, which is used for serialisation. All objects which do not subclass the ssp.Remote interface are automatically marshalled by value.

**Marshalling the Local References**

As mentioned in the section above, all arguments that are not of a primitive type are marshalled by reference if they subclass the ssp.Remote interface. The marshalling of such references is deferred until all arguments of the message have been marshalled. Space is reserved for the marshalling of the references.

---

```java
Enumeration enum = enumeration of the pointers
int size = bufferOut.getPosition(); // save position for later restore
PtrPosition ptrPos = null;
while (enum.hasMoreElements()) {
    ptrPos = (PtrPosition) enum.nextElement();
    bufferOut.setPosition(ptrPos.position); // the position where
    // the pointer needed to
    // be written in the buffer
    marshal(ptr2loc(ptrPos.ptr, callingSpaceName, timestamp));
}
bufferOut.setPosition(size); // restore position in buffer
```

---

**Figure 9.16 Pseudo-code for marshalling Local References in a method call message**

When all the fields of the message have been marshalled, then the Message.marshalPointers() method is invoked. This causes the marshalling to happen as shown in Figure 9.16. This takes the set of PtrPosition, and for each one of them, sets the marshalling position in the stream, according to the value set in the stream. Then it calls a function which returns a locator to the local reference. The locator is a remote reference identifying the scion which has a local pointer to the object which is being referred to in the reference. The pseudo-code for the ptr2loc function is given in Figure 9.17.

**Ptr2loc** takes in arguments which are the local pointer to an object, the space for which the remote references have to be returned and the timestamp to mark on a scion if an existing scion is found.

The code checks amongst all the existing scions if there is a scion which points to the local object and refers to the same remote space that is needed by the function. If one is found, then its timestamp is updated and it is returned. If an instance is not found, then a new instance is created by calling on Scion.forRemoteService(), which returns an instance of the scion sub-class that is required.

The opposite is needed when unmarshaling references from call messages. The method SSP.loc2ptr(), takes a locator, the timestamp and a class name for local reference required. It searches through the current references first from the weak location of the locator to find a matching scion which will contain the local pointer. If not found, then the strong location is matched. If both fail, then a new stub is created which points to the remote location given by the locator, hence an indirect chain is created.
9.7 Stub-Scion Generator

This section details the ssp.SSGen static class, which is a command line tool used to generate the stubs and scions for objects which are to be public. The tool works on the compiled class rather than the source. This approach is easier due to Java's Reflection capabilities, allowing the inspection of the classes properties.

There are options for the tool to keep the source of the stub or to delete it, and for the tool to be verbose in its actions.

The fully qualified class name of one or more classes is given. Each class is processed in turn. Checking to see that the classes specified refer to concrete, i.e. non-abstract classes and not interfaces. The next section details the generation of the common parts between the stub and scion for a given class. The common parts are the package declaration, the import statements and the class declaration.
### Import Class Statements

The target class is inspected at the following level to get information on all the referenced classes from that class:

- Get all declared fields. These are the public, protected and package access fields of the class. All non-primitive classes encountered have their names stored in a hash-table, which eliminates duplicates.
- Get all constructors and get each constructors arguments. Again, all non-primitive types are added to the same hash-table.
- Get all methods and get each method’s arguments as well as return types. Store all the non-primitive types to the same hash-table.
- Because some of the arguments or return types will refer to the class itself, that class has to be excluded from the import statements.
- Write out each element of the hashtable in statements like “import <hashtableValue>”.

### Class Declaration

First the access type of the class being inspected is ignored and replaced with “public”, next the class name, stripped off all package naming, is written, together with the appropriate suffix (“_Stub” or “_Scion”). Then, the stub file needs to be written: “extends ssp.system.Stub” and similarly for the scion file. The interface declarations come next: all the interfaces that the class implemented, the stub needs to implement, but not the scion. This is because the scion will not have the same interface as the class as it will not be seen at the application level.

Also note that classes sub-classing base or super-classes do not result in the stubs and scions sub-classing the super-classes of the stubs and scions respectively. This is because using the java.lang.reflect.Class.getDeclaredXX() methods e.g. the getDeclaredFields(), returns all fields declared in the class and all its super-classes.

Specifically, if a class 'C' subclasses a class 'B' which in-turn subclasses a class 'A', but only class 'C' needs remote references, then there is no need to generate stubs and scions for 'A' and 'B'.

### Writing Stub Methods

All the methods of the class are obtained by invoking Class.getDeclaredMethods(), each of them is traversed. If it is a finaliser, then it is ignored.

For each method, first the method signature is written. This is exactly the same as the original method:

- The modifier is written. This is done using Method.getModifiers(). If abstract, then remove the abstract by bitwise manipulation. Write the modifier out (Modifier.toString(thisModifier)).
- Write the return type of the method: Method.getReturnType().getName().
Implementation

- Write the method name out: `Method.getName()`
- Write out all the arguments. They will be returned by `Method.getParameterTypes()` as an array of `Class`. This is fully qualified, and thus it needs to be made unqualified (this is safe because every referred class has been imported).
- Write out the throwing `ssp RemoteException` to handle network problems.
- Write out method body, (an example is seen in Figure 9.18) which creates a new message with type `Message.CALL` and opcode which increases for every method. Opcodes 1 and 2 are reserved for passing by value and passing by reference.
- Each of the arguments is marshalled. Because the `Method.marshal()` method is overloaded with every type of primitive and the special cases, then no further action need be taken to determine marshal types, just a call to the marshaling function with the argument.
- The last line is a call to the `Stub.ssp_call(Message)`. This invokes a call to the method which sends the message. The reply is received and is unmarshalled and the return type cast before returning.

Writing the Scion

```java
public void foo(NonRemote a0) throws RemoteException {
  if (forwarderTarget == null) {
    Message call= new Message(Message.CALL, 2);
    call.marshal(a0);
    Message reply= ssp_call(call);
  }
```

Figure 9.18 A stub method for a given method in the class. The original method has the same signature

The sub-class scion has only got one method which implements an abstract method declaration. The method is called after the message is received by the transport layers. The following code needs to be written for the method:

- The `ssp_call` method receives the message, unmarshals the opcode (the superclasses have unmarshaled previous fields such as message type etc.)
- A new instance of a reply message is created.
- A switch statement, switching on the unmarshalled opcode is written.
- For each method, a case statement with opcodes starting from 2, counting onwards.
- Marshalling the method opcode into the reply message. This is necessary for matching outstanding replies.
- Argument unmarsalling code is written, the argument is stored in a loose type, i.e. Remote for most objects and the interface for the remote object if that is the remote object.
- The reply message is returned at the end of the switch statement.
Issues with the Code Generator

Because the emphasis of the project is not to develop a comprehensive distributed system, but to develop a collector for a distributed system, the code generator is not perfect. The following are some of the issues.

Most commercial code generators of this kind leave all arguments and return types fully qualified if they are inspecting classes. My approach of declaring all referenced classes as imports makes the code more human readable, but it has issues when code written explicitly refers to two different classes with the same name in different packages. For example, if there is a `ssp.test.Hashtable` and the standard `java.util.Hashtable`, then both of them will be written as imports. Attempts to use those classes without fully qualified names will result in erroneous code.

The return types of remote methods are very loosely typed, i.e. to Remote if they are not primitive.

9.8 Invocation Protocol

9.8.1 Message Formats

Standard Message Header

This is used to identify all messages. It contains basic information that is needed, such as package size, which is needed by the transport protocol so that message receipt can be known to be complete.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Data size in bytes</td>
</tr>
<tr>
<td>Integer</td>
<td>Message type: {CALL, REPLY, FORWARD, EXCEPTION, LIVE}</td>
</tr>
<tr>
<td>Integer</td>
<td>Sequence number, for matching call to response</td>
</tr>
<tr>
<td>Integer</td>
<td>Timestamp of sending space</td>
</tr>
<tr>
<td>SpaceName</td>
<td>Sender's space name.</td>
</tr>
</tbody>
</table>

Table 9.1 The Standard Message Header Format

Call and Forward Message

This fields are in addition to the standard message header.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Scion name, name of current target scion</td>
</tr>
<tr>
<td>SpaceName</td>
<td>Space identifier of the original caller.</td>
</tr>
<tr>
<td>Integer</td>
<td>Sequence number, for matching call to response</td>
</tr>
<tr>
<td>Integer</td>
<td>Opcode of the method to invoke, starting from #2</td>
</tr>
<tr>
<td>Bytes[]</td>
<td>Data, all the arguments marshalled as described</td>
</tr>
</tbody>
</table>

Table 9.2 Format of the Call and Forward Message Types
### Reply Message

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td></td>
<td>Sequence number, matching with the call message</td>
</tr>
<tr>
<td>Location</td>
<td></td>
<td>Up-to-date Location of the target, this is for strong chain short-cutting.</td>
</tr>
<tr>
<td>Bytes[]</td>
<td></td>
<td>Data, results marshalled as described</td>
</tr>
</tbody>
</table>

Table 9.3 The Reply Message Format

### Exception Message

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td></td>
<td>Better weak location. Gives location of scion in the same space as the target.</td>
</tr>
</tbody>
</table>

Table 9.4 The Exception Message Format

### Live Message

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td></td>
<td>Sender threshold value</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td>Number of scion names in this message</td>
</tr>
<tr>
<td>Integer[]</td>
<td></td>
<td>Name of each scion for this space</td>
</tr>
</tbody>
</table>

Table 9.5 The LIVE Message Format
9.8.2 Remote Invocation

A remote method call is invoked via a stub. The sub-class stub generated by the stub-scion generator will generate code that instantiates a Message class and marshals the arguments in the message. This is then passed to the super-class ssp_call() method. The code can be followed through in Figure 9.19.

```java
public Message ssp_call(Message callMsg) {
    while (true) {
        int ts = SSP.stamp();
        SpaceName target = locator.weak.spaceName;
        callMsg.calleeScion = locator.weak.scionName;
        Message receiveMsg= null;
        callMsg.marshalPointers(locator.weak.spaceName, ts);
        try {
            receiveMsg = SSP.callAndReceive(target, ts, callMsg);
        } catch (java.io.IOException ioe) {
            System.exit( 1 );
        }
        switch (receiveMsg.type) {
            case Message.EXCEPTION: {
                redirectWeak(receiveMsg.senderSpace, receiveMsg.betterWeak);
                break; // retry
            } //end case EXCEPTION
            case Message.REPLY: {
                if (!locator.weak.spaceName.equals(locator.strong.spaceName)){
                    Locator newLoc= 
                    new Locator(receiveMsg.senderSpace, receiveMsg.calleeScion);
                    redirect(newLoc, receiveMsg.senderTimestamp);
                }
                int receiveOpcode= receiveMsg.unmarshal_int();
                return receiveMsg;
            } //end case REPLY
        } //end switch
    } //end while true
} //end sspCall
```

Listing 9.3 Code for the Invocation Protocol features a loop that retries sends till succeeds
Similarly, the receipt of a CALL message is the other side of this interaction. The scion sub-class for an object receives the call. This is received by an instance of the MessageDaemon thread which blocks on listening to a plain socket. This is shown in Figure 9.10. The MessageDaemon then gets an instance of a thread from the MsgDispatchPool. Reading the rest of the message from the socket stream is done via this thread. Obtaining the thread is shown in Figure 9.11. The message is then constructed using the socket input stream as an argument. The constructor reads off the stream and populates the fields of the message. This is shown in Figure 9.12.

When SSP.transportReceive() receives the message, the action taken is based upon the type of message received. Because we are considering the situation of a CALL message receipt, the scion to which the message was sent is found and the scions Scion.call(Message) method is invoked. The method invoked here depends on whether the scion is pointing to the target locally. If the chain is indirect, then the scion will be pointing to a stub. The simple case where the scion is pointing to a local target is shown in Figure 9.19.
9.9 Collector Protocol

9.9.1 Sending live messages

SSP.init() is invoked on initialisation of a space. The method creates an instance of ssp.GGC, which is a thread subclass. The following is the code that the thread executes.

```java
public void run() {
    while (true) {
        try {
            sleep(10000); // wait for 10 sec before sending a live message
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        SSP.sendLiveMessage();
    }
}
```

Listing 9.1 Distributed Cleanup Protocol thread.

Figure 9.21 shows the interaction between classes when the SSP.sendLiveMessage() method is invoked. The Distributed Cleanup Protocol thread, ggc, invokes this method and thus it runs in the background, independent of mutator activity.

The method call works as follows:

- A static method call to the class ssp.system.EndPoint gets an instance of ssp.system.SpaceNameSet which holds SpaceName identifiers to all the spaces which are connected to this space. This set is calculated on a method call Endpoint.getConnectedSpaceSet().
For each space in the SpaceNameSet, get the set of stubs in this space that have matching scions in that space. This is done by invoking method StubTable.stubsToSpace().

A new LIVE message is instantiated and the threshold value for the space to which the message is being sent, held in this space, i.e. if this is space 'A' and the space to which the message being sent is space 'B', the value is threshold[A][B]. Also, the number of stubs contained in the StubSet returned above is marshalled.

```
public static void sendLiveMessage() {
    SpaceName space;
    SpaceNameSet spaceSet=EndPoint.connectedSpaceSet();
    for (int i= 0; i<spaceSet.size(); i++) {
        space= (SpaceName)spaceSet.elementAt(i);
        StubSet stubs= stubTable.stubsToSpace(space);
        Message liveMsg= new Message(Message.LIVE);
        liveMsg.marshal(threshold.get(space));
        liveMsg.marshal(stubs.size());
        for (int j= 0; j<stubs.size(); j++)
            liveMsg.marshal(((Stub)(stubs.elementAt(j))).locator.strong.scionName);
        transportSend(space, stamp(), liveMsg);
    } //end for all connected spaces
}
```

Listing 9.2 SSP.sendLiveMessage(): sends a LIVE message to every connected space.
For every stub in the stub set, find out the locator's strong location scion name, i.e. the scion name of the matching scion to this stub, and marshal that into the message.

Send this message to the space 'B' and repeat for every space in the connected space set.

The code for the message sending method is shown in Listing 9.2

### 9.9.2 Receiving Live Messages

All messages are received by an instance of the `MessageDaemon` thread. There is one thread for every space that is connected to this space. As seen in Figure 9.10, the receipt of the message causes the creation of a new thread to deliver the message to the appropriate part of the system, which is an instance of `MessageDispatch` thread. This thread executes the `SSP.transportReceive()` method, whose actions are dependant on the type of message received. In the case of a LIVE message, the `SSP.receiveLiveMessage()` is executed.

The receipt of a LIVE message is processed as follows, consider the receiving space to be space 'A' and the space from which the LIVE message arrived to be space 'B':

- The singleton instance of `ssp.system.ScionNameTable` is a hashtable mapping all the scion names to the scions. For each element in this table, i.e. each scion in this space, the scions source space is checked to see if it matches the space from which the LIVE message arrived, i.e. space 'B'. If the source space identifier matches, then this scion is one whose matching stub resides in space 'B'. A reference to the scion is inserted into a `ssp.system.ScionSet` instance.

- The threshold value received from the message, i.e threshold$_b[A]$, is unmarshalled and stored. The number of scion names contained in the message is also unmarshalled and stored.

- For every scion name contained in the LIVE message, unmarshall it. If the ScionSet constructed above contains a scion with the same name, then remove it from the set, i.e. if there is a stub that matches the scion in the set, then remove it. Thus the set will be left with scions which had no matching stubs at the time the LIVE message was sent.

- For each of the remaining scions in the set, if their timestamp is less than the threshold value received from the LIVE message (threshold$_b[A])$, then there are no messages sent out by the scion which may make it reachable (Note that all messages sent out via a scion get a new timestamp, and the scion gets timestamped by the same value as well). Thus that scion is unreachable and thus can be reclaimed.

The code for the LIVE message receive method is in Listing 9.3.
9.1 Cyclic Extension Protocol

This section gives implementation details of the cyclic extension to the SSP acyclic collector presented above. The design of this algorithm, due to [Le Fessant et al., 1997] is given in Section 8.10.

9.10.1 Local Trace Sub-Protocol

This component of the cyclic extension has been the most difficult to approach. This is due to the inability to interact with the local garbage collection algorithm. Also, the design as presented in [Le Fessant et al., 1997] assumes a tracing collector and gives details on modifications to the collector to enable stub marking with timestamps.

The collection algorithm in Java's Virtual Machine is implementation dependent, with the Sun implementation having a generational stop and copy collector. Thus, even access to the Virtual Machine source would not be very helpful as the algorithm detects garbage in a different manner to that expected by the cyclic extension protocol.

The solution to the problem is very expensive in terms of computation. This makes the implementation of the cyclic extension a proof-of-concept rather than an implementation that could be used in distributed systems development. It has two drawbacks: the use of Java Reflection making it inefficient, and imposing programming style restrictions by having to call certain methods on method entry and exit.

```java
public static void receiveLiveMessage(Message liveMsg) {
    ScionNameSet scionNames = new ScionNameSet();
    Enumeration scionNameEnum = scionNameTable.keys();
    int scionName;
    Scion scion;
    while (scionNameEnum.hasMoreElements()) {
        scionName = ((Integer)scionNameEnum.nextElement()).intValue();
        scion = scionNameTable.get(scionName);
        if (scion.sourceSpace.equals(liveMsg.senderSpace))
            scionNames.addElement(scionName);
    }
    int thresh = liveMsg.unmarshal_int();
    int size = liveMsg.unmarshal_int();
    for (int i = 0; i < size; i++) {
        scionName = liveMsg.unmarshal_int();
        scionNames.removeElement(scionName);
    }
    for (int i = 0; i < scionNames.size(); i++) {
        scion = scionNameTable.get(scionName);
        if ((scion.timestamp <= thresh)
            || (scion.sourceSpace.equals(SSP.thisSpace))) {
            deleteScion(scion);
            scion = null;
        }
    }
}
Listing 9.3 SSP.receiveLiveMessage(): receives a LIVE message from another space.
```
As mentioned in the design section (8.10), the algorithm works on the basis that local garbage collector traces propagate timestamps to rooted stubs while unrooted stubs will not have increasing timestamps. Thus a mechanism to trace the root set of the space is needed to traverse the live set and find all reachable stubs.

Method reflection to obtain locally declared variables is not possible. \texttt{java.lang.reflect.Method} only provides reflection capabilities so that the methods of a class may be inspected for their number, names, signatures, modifiers and return types. Thus, the \texttt{static void main()} method, in which it is usual to create instances of components of the system, cannot be reflected. The way round this is to enforce the application programmer to call methods handing references to all the locally declared variables in a method.

### The MarkSet

Each node has a singleton instance of the class \texttt{ssp.system.MarkSet}. As it needs to be well known, it has been implemented as a static class.

This class sub-classes \texttt{java.util.Vector} and contains elements which have references to all variables declared on entry into a method. These elements are instances of \texttt{ssp.system.MethodVariables}, which is also a subclass of \texttt{java.util.Vector}.

Every method for every application class has to do the following:

- Inform the marking sub-protocol that a new method has been entered. Hence this is the first statement. It is a method call \texttt{ssp.system.MarkSet.methodEntered()}
- Declare all local variables at the start of the method. This imposes a restriction similar to some implementations of C. Java does not impose that restriction normally.
- Hand references to the mark-sub protocol via the method \texttt{MarkSet.localVariable(Object)}.
- Inform the marking sub-protocol of the exit of the method. As the return statement needs to be the last, this case has to be handled separately, as it is still in scope when this method call is invoked. The method to be invoked is \texttt{ssp.system.MarkSet.methodExited()}

![Figure 9.22 The Mark Set Classes which store references to all variables in current scope](image)
On a method call to \texttt{methodEntered()}, a new instance of \texttt{CurrentVar} is created, inserted into the singleton instance of \texttt{MethodVars} and a reference held to that. The reference is also returned to the calling method.

On method calls to \texttt{localVariable()} with the local variable concerned and the reference to the vector, the local variable reference is added to the instance of \texttt{CurrentVar} that is specified.

\textbf{MarkDaemon}

This is a sub-class of \texttt{java.lang.Thread}. It is responsible for marking all the stubs with the current time. The following is the functions performed.

- Create an instance of \texttt{ssp.system.DetectGC}, which is a sub-class of \texttt{Object} with a finaliser method that notifies the \texttt{MarkDaemon} thread.
- Set the reference to the above created object to null. This will cause the object to be collected on the next local collection. Its finaliser will wake this thread.
- Sleep till notified.
- On being notified, get the main \texttt{java.lang.ThreadPool} and call the deprecated \texttt{Thread.suspend()} method on every thread. Do the same to all sub groups, which are instances of \texttt{ThreadPool}.
- Start the mark phase. This involves:
  - For each instance of \texttt{CurrentVar}, get an enumeration of the vector.
  - For each element in the enumeration, get the reference and get the class of the reference.
  - Get all declared fields in the class, by \texttt{Class.getDeclaredFields()}.
  - For each of the fields returned in the array, get the instance of the Object referred to by, invoking \texttt{Field.get(Object)} with the object instance as the parameter.
  - Recursively do the same to this objects fields and inspect if each of these fields are an instance of \texttt{ssp.system.Stub}. If so then invoke the \texttt{Stub.setTime(int)} method to set the timestamp to that stub.
  - The recursion does not proceed when primitive types are encountered. Also, only populated fields can be traversed, thus the need for catching \texttt{java.lang.NullPointerExceptions}.

\textbf{Updating Threshold due to Collected Stubs}

The use of \texttt{java.util.WeakHashMap} for a hash map with weak pointers is employed to update timestamps when stubs are collected. After each local garbage collection, the stubs with only weak pointers will be appended onto a queue that the registered weak pointers specified (see Weak Pointers documentation for Java 2). A separate thread polls the queue to see if any new objects have arrived. If a new stub arrives then the threshold value is updated.
Other Components

The implementation of the Detection Server is presented in pseudo-code in sufficient detail in section 8.10.

Similarly, as the implementation for the thread that handles the delivery of the STUBDATES, ACK and THRESHOLD messages is similar to the handler of the LIVE messages, and the pseudo-code is described in section 8.10, it is not detailed in this section.

9.11 Registry and Naming Service

9.11.1 Bootstrapping

In any distributed object-oriented system, remote references to objects are passed by invoking remote method calls on existing remote references with references to objects which may or may not be public already. If a method is invoked on a remote object with an argument that is a local reference to an object that is not public yet, then a new remote reference to that argument object will be created to enable the object receiving the method call to be able to invoke methods on the argument.

Another way that remote references are propagated within the system is by remote method invocations returning results which are remote references.

In my implementation of the Stub-Scion Pairs system, and in most other systems, this is the only way that references are propagated. This poses a problem: on initialisation of a system with a given number of nodes, there is no way to acquire a first reference to a remote object.

The implementation features a registry and naming service which solves the above mentioned problem. The registry sub-system has two key roles:

- Allocating initial transport handles so that nodes can communicate.
- Exporting references to a name server so that they are well known and clients may obtain references to that object by well known names.

There are two separate mechanisms implemented to allow bootstrapping of references. These are discussed below.

9.11.2 Service Scions

Each space initialises on the SSP.init(<port number>) call. Upon initialisation, it allocates a singleton instance of ServiceScionTable and an array to hold references to these scions.

Service Scions are created for server objects. These scions are treated different from other scions because they are prevented from being garbage collected. This is allowed even though there never exists any stub with a strong location referring to the scion. Protecting such scion allows the clients for the server objects to acquire a reference to the server object to invoke methods on it.

This does violate the liveness invariant of the system (Inv 3 in section 8.3), which states that every scion has at most a single matching stub. Note that the
presence of floating garbage is relaxing the liveness condition, but the presence of service scions violates the liveness condition.

For this reason, the mechanism only allows a weak reference to be created to it. This ensures that strong chain short-cutting (section 8.8.2) occurs on the first method invocation. This creates a new strong chain referring to the server object and the stub with the weak reference becomes garbage. This attempts to reduce the time for which the invariant is violated.

Note that a service scion protects a server object from being collected if the local reference is destroyed. This is the normal behaviour of a scion.

Creating Service Scions

Figure 9.23 SSP.localService(): does installation of a Service Scion for a server object.

A server object is created in a space and then a call to SSP.localService() is invoked, which creates an instance of the appropriate scion object and installs it as a service scion. The call delegates the actual instance creation of a scion to Scion.forRemoteObject(). The interaction is seen in Figure 9.23. The two methods are shown in Listing 9.4.
A client space can get a reference to a server space if the client knows the server object’s service name and the location. This scheme is used in many systems, which use URLs to do the same. The current implementation supports the following scheme:

The server object’s location is given by:

```
hostname.domain:portNumber:serviceScionNumber
```

The service name is given by:

```
myPackage.MyImplementation
```

For example: “hex.doc.ic.ac.uk:2030:12” and “testing.TestServer”

The client space requests for a reference by invoking the method `SSP.remoteService(“location”, “serviceName”)` which causes the

```
Listing 9.4 SSP.localService() and the delegate method Scion.forRemoteObject()
```
invocation of `Stub.forSvcName()` which parses the strings to create Location objects and load up the appropriate stub class before dynamically instantiating it. Because the mechanism is very similar to the installation of service scions, with the additional task of setting weak references to the matching service scion, it is not detailed here but can be found by referring to the implementation code.

### 9.11.3 Registry Service and Naming Client

This mechanism is used to register the names and locations of well known objects. It allows objects in other spaces wishing to use the services of certain well known objects to obtain references to those objects.

#### Naming Client

Each space can use the static class `ssp.Naming`, which is the naming client, either to obtain a remote reference via the registry server, or to place an entry on the registry server so that other objects may use this objects services.

```java
Naming
$ serviceCounter : int = 1
$ localPort : int = 0

Naming()
bind(url : String, obj : Remote) : void
rebind(url : String, obj : Remote) : void
unbind(url : String) : void
lookup(url : String) : Remote
list(url : String) : String[]
connect(url : String) : String
```

![Figure 9.24 UML Class Diagram: ssp.Naming, the registry client class available to each space.](image)

As the implementation details of the client are not of concern here, only the method use is presented. Implementation details can be found by referring to the code.

#### Binding an Object

This places an entry in the Registry server which locates a server object. A space which wishes to advertise the services of an object may do so by invoking:

```java
Naming.bind(<URL of the service>, localObjectReference);
```

The URL format is as follows:

```
//host:port/name
```

Where host refers to the host name of the registry server, port refers to the port at which the registry server is listening to and name refers to the name of the object.

The binding request is sent to the registry server in a message whose format is described in Table 9.6. The naming service waits for a positive acknowledgement message to confirm the binding. Failure is indicated by negative acknowledgement, which results in raising of an `ssp.AlreadyBoundException`. 
A space may rebind a service if it needs to by invoking the method:

```java
Naming.rebind(<URL of service>, localObjectReference);
```

This enables the server to overwrite a previous bind. The message format is similar to a bind message (Table 9.6), except with a different opcode.

A space may remove the binding to an object before it destroys the object. This is done by invoking the method:

```java
Naming.unbind(<URL of the service>);
```

This removes the entry from the registry server that referred to this object.

### Obtaining a Remote Reference

A space may enquire about the location of an object by name. This is done by invoking:

```java
(Remote)Scion Naming.lookup(<URL of the service>);
```

The lookup message is sent to the registry server via a message whose format is described in Table 9.7.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Lookup opcode, currently 4</td>
</tr>
<tr>
<td>Integer</td>
<td>Length of the string name of the object</td>
</tr>
<tr>
<td>Bytes[]</td>
<td>String name of the object</td>
</tr>
</tbody>
</table>

**Table 9.7 Format of the Registry lookup message**

### Communication Between Client and Server

The current implementation does not use standard message types described in Section 9.8.1 for the communication of lookups and bindings. It uses its own formats. Also, it uses TCP, which can be replaced with UDP to reduce the resource usage.
The sockets full address is marshalled using the java serialisation streams, which may be replaced by using similar mechanisms to the ones used in method invocation (java.io.ObjectOutputStream).

Registry Server

1) Spawn a server socket listening to the specified socket. Instantiate a new Hashtable to store binding information in.

2) In a loop forever, wait for a client space to initiate communication. This causes the server socket to create a new plain socket for the client to talk on. This creates a new instance of the thread Registry_Daemon, which then services the client request.

The Registry_Daemon thread starts by getting the input and output streams from the socket and creating data streams over that. This enables it to read Java primitive data types straight away. It then reads the message opcode and performs the required action:

- For a bind request, look into the hashtable with the same key. If it exists, then return a negative acknowledge message. Otherwise create a new instance of BindingInfo with the information for the bind and add it to the table.
- For a rebind request, do the same as a bind, but without looking for a previous entry. This overwrites a previous entry.
- For a lookup request, if the lookup in the hashtable succeeds, compose reply message which contains the location of the scion in the system.
- For a list request, return a string named representation of the hashtable.
Possible Improvements

This scheme does not address security issues where malicious spaces could change bindings by performing rebind requests. This is a simple implementation though, by keeping owner information and nonce values, and only allowing rebinds to succeed if the space is the same as the original binder.

The implementation is only meant to allow demonstration and test of the system, and not to be comprehensive.
CHAPTER 10

System Analysis

In this chapter, the performance analysis of my Stub-Scion Pair chains/Java (SSP) is presented. Particularly, the overheads of remote method invocation, marshalling references and the distributed collector protocol are detailed. Also, similar tests conducted on Java's RMI are presented and compared.

10.1 Quantitative Tests

This section gives the quantitative tests that were carried out on the implementation. It gives the basis of the comparisons made and some inferences that were made.

10.1.1 Types of Tests

The following was aimed to be tested out of the various test routines written and conducted:

- **The overhead of remote invocation.** This would reveal the overheads involved in marshalling and unmarshalling messages and copying between different buffers, if any was done.
- **The overhead of marshalling primitive arguments.** This test is aimed at revealing any overheads that might incur when arguments are marshalled.
- **The overhead ofmarshalling references.** This test gives the overheads incurred in creating and managing stubs and scions.
- **The overhead of processing LIVE messages.** This gives the overheads incurred in processing the live messages which are sent by the distributed Cleanup Protocol.

10.1.2 Testing Environment

The testing environment needs to be standardised between the various tests to ensure that the results obtained are valid. The following steps were taken to eliminate any variances that may be caused by factors other than the efficiency of the implementations:

**Operating System**

Although Java Virtual Machines exist for various platforms, they are only similar with respect to functional requirements. The performance of various implementations for different platforms may vary widely. Also, implementation tweaks and changes between versions may change the performance. Another factor is the reliance of the virtual machine on system libraries, which may
provide similar functions across systems but may be different in terms of performance.

The operating system selected for the test was Microsoft Windows NT version 4. Service Pack 3. The reason for using Windows rather than Linux is because the laboratory machines are configured so that a login on Windows gives the user control over all processes running on the system, allowing a user to kill all processes to free the CPU. This is not possible under Linux as remote users may be logged on and running CPU intensive processes.

When running the client and server on different machines, the tests cannot be run on different platforms even though written in Java because of the VM’s reliance on the OS for the transport protocols, and implementations of TCP/IP may be more efficient between operating systems.

Virtual Machine

The Sun Microsystems, Java 2 release, version 1.2.1 for Windows was used as the virtual machine. Concern that the local garbage collector running asynchronously may pollute results because it may collect at different frequencies for different runs lead to the use of JDK 1.1 virtual machine, specifically version 1.1.5. This is because the switch for disabling asynchronous garbage collection is absent from the 1.2.1 version. Pilot tests showed that even with the collector running, the Java 2 virtual machine was faster, thus it was chosen for the final tests.

Hardware

An Intel Pentium II Processor, running at 350 MHz, with 128 MB RAM was used. No other applications were executing in the background while running the tests.

10.1.3 Timing Resolution

The Java system only gives time measurements accurate to 1 millisecond. This is provided by the java.util.Date class. Although sufficient for general use, it is inadequate for timing events which occur in a number of milliseconds.

The other problem is that because the Date class is immutable, the only way to get the current time is to create a new instance of the class. This causes class loading and instantiation overheads. Such overheads may be significant for events closely timed together such as method call times. Class caching has been used but there is no way round the overheads of object allocation.

These problems could be circumvented in two ways. One would be to use a native library interfaced with Java using JNI. The timing start methods and stop methods would then be native. Again, the overheads that JNI itself may have could prove to ruin the result.

The approach taken was to repeat the operation for a hundred times, and only start timing at the first instance and stop after the last. The reading could then be divided and this would provide greater accuracy.

To get a sense of the variation involved, each of the above tests was taken 30 times and the average taken. If the variance of the 30 readings was greater than 5%, the test results were invalidated and that test repeated.
Also, the results were always higher for the first run in the 30 run series. This may have been due to cache misses the first runs increasing the average run time. The first reading was thus excluded from the average.

10.1.4 Remote Method Invocation

This test shows the overheads involved in invoking methods remotely. In the documentation for the Soul system, which is a C++ implementation of Stub-Scion pair chains [Plainfossé, 1994], the results of invoking methods between two spaces on the same host were very different compared to the results of invoking methods between two spaces on separate machines. This is because the transport protocols must be short-cutting buffer copying at some layer on the same host.

Performing similar tests on this implementation surprisingly revealed very little difference between the timings of inter-space communication on remote machines and on the same machine. This is contrary to the result from [Plainfossé, 1994]. This may be explained by the fact that the TCP/IP stack software used in their Sun system may have done buffer-copy short-cutting for the same machine, while the one used in Win NT/J ava does not do that and hence offers no advantages.

The execution times of remote method invocation between two spaces on different servers was lower than on the same machine. This can be explained by the fact that performing a method invocation is not a high amount of computation compared to the amount required to run a virtual machine, with instruction mapping, virtual memory mapping, and threads that are part of the system, as well as SSP object management threads. Thus running two virtual machines compared to one virtual machine per processor is slightly faster, even on a Pentium II 350. This can be seen from Figure 10.1.

The other characteristic of the graph in Figure 10.1 is that the RMI values are higher than the SSP timings. At first it may appear that the SSP implementation is more efficient. But, I am not making such claims because examining the source of RMI, (the parts with source code available, at least) reveals that method calls, class loading and instantiation undergo security checks by an instance of java.rmi.RMISecurityManager. This increases the amount of processing required. Another issue is that the stubs and scions don’t use integers for opcodes, but stringified method signatures. Also, the stubs and skeletons generated in RMI have trees of super-classes if their parent classes
have stubs and scions. This means that the resulting stub code is re-used, and large stubs are not generated for classes in a class hierarchy.

10.1.5 Marshalling Overhead for Primitives

Figure 10.2 shows the execution times of methods with primitive integers as arguments to the invocations. The results are in line with expectations that the overhead is insignificant. This can be appreciated if we consider what happens when a primitive is marshalled. The marshalling is done onto a data output stream. The data output stream is constructed from a lower stream which is a byte buffer stream. This is an allocated array in memory, and writing an integer copies four bytes onto an already allocated data structure. Clearly this is very insignificant compared to a single method local method call which requires arguments to be pushed on the stack etc.

10.1.6 Overheads of Marshalling References

The overheads of marshalling local references in one space to become remote references in others are clearly significant. This is because the local reference may not have an a pair of stub-scion to make it a remote reference and hence a scion needs to be instantiated. This involves the searching for a scion class, loading the class, allocating an instance, updating references in the stub/scion management data structures. This overhead can be seen from Figure 10.3. The overhead of marshalling a single reference is approximately four times that of invoking a remote procedure call.
10.1.7 Overhead of Processing LIVE Messages

This test shows the costs in computation time that have to be paid to benefit from automatic memory management at a distributed level.
As can be seen from the graph in Figure 10.4, the incremental overheads for an increasing number of public objects is measurable, but insignificant. The graph in Figure 10.4 is plotted when messages were exchanged between two spaces.

The graph in Figure 10.5 is the LIVE message sending overhead in a system where each space had 40 public objects with referents randomly assigned from any one of the other spaces. Thus, the number of objects in the system was proportional to the number of spaces. This models a common occurrence in real distributed systems, with respect to the number of objects. But the graph denoting the connectedness of the spaces, i.e. the number of spaces that each space refers to, is fully connected in the test system. This does not occur in distributed systems with a large number of nodes. Public objects exhibit a high locality of reference [Jones and Lins, 1996].

The incremental overhead in this case is quite substantial, i.e. the time taken to send messages increases. The reason for this can be seen from the algorithm presented in Listing 9.2.

![Overheads for Sending LIVE Messages at 40 public Objects Per Space](image)

**Figure 10.5** Overheads for sending LIVE messages between 2-8 spaces with 40 objects in each space

The outer loop in the algorithm is “For each space in the set of Connected spaces”, prepare and send a LIVE message. Each live message has a size determined by the number of public objects that the spaces share. But it can be seen from earlier tests that the size of messages only imposes a very small incremental overhead because marshalling primitives is computationally cheap. Sending a message of any size to another space, however, is not without cost. It is this cost that dominates the time taken in the sending of LIVE messages. If the graph were not fully connected, even with the same number of public objects, the time spent on the send messages would be less.

Receiving a single LIVE message is computationally more expensive than sending one. This can be seen from the graph in Figure 10.4. This is in line with what can be expected looking at the code given in Listing 9.3. It can be seen that the set of scions with matching stubs first has to be constructed, then the LIVE
message processed by demarshalling, which has the same expense as marshalling.

Other scalability considerations in processing LIVE messages, either for sending or for receiving are:

The more scions a space possesses, the more time it will take to generate a LIVE message because in the current implementation, every scion is traversed in looking for the scions which match the space to which the message is being sent. Also, this results in more time taken when generating the set of scions that will be removed when processing the LIVE message. An alternative indexing mechanism to the one used in this implementation can be used in which all scions are indexed by their spaces, these indexes are always consistent by having the references updated on allocation of the scion.

The more connected a space is to other spaces around it, the more time it will take to send LIVE messages because LIVE messages are sent to each space which is connected to this space.

10.1.8 Qualitative Analysis of RMI with SSP Chains

This section gives a qualitative analysis between the design choices made in RMI and SSP chains. RMI is based on the Network Objects system by Digital Equipment Corporation, [Birrell et al., 1993]. The distributed collector is based on the collector implemented for the Network Objects system, described in detail in section 7.2.3.

- A reference export in RMI causes immediate short-cutting. The owner of the object is informed of the new client object. This short-cutting mechanism has an advantage that access to an object does not rely on a third party space through which the references traverse. On the other hand, it has a higher cost of reference duplication. There is a need for two extra messages per transfer of reference. In applications where references may be liberally passed, but not frequently used for method invocation, the short-cutting mechanism in RMI causes overhead which can be avoided. The SSP system uses a lazy short-cutting mechanism. The rationale for short-cutting is that third-party spaces do not have to be relied up to reference objects. Also, the indirections are an inefficient communication means. The lazy policy of short-cutting on first method execution only avoids short cutting before strictly necessary. This allows the references that objects may receive but never invoke, to get away from never being short-cut.

- In SSP Chain reference duplication, the owner space does not need to be informed. This is an advantage to as popular objects may have references to them being exported by hundreds of client objects.

- The reference listing collector implemented in RMI is not fault tolerant because a failed space needs its entry in the list to be removed as it violates the invariants.

- The SSP Chain system supports object migration. This is because the indirection allowed by SSP chains makes moving objects easier.

- The implementation of SSP chains in Java developed for the project is not secure because it does not restrict activity by remote classes. On the other hand, Java RMI has a security manager which restricts access to remote classes.
This chapter concludes the work in this thesis by re-iterating the important issues. Specifically, my contributions are detailed. The experience in implementing the garbage collector and distributed system is also presented. Several implementation issues that can be improved are discussed.

11.1 Goals Achieved

At the outset of the project, the main goals of this project were to have an implementation of a distributed garbage collector on top of an open source distributed object-oriented system. During the research phase of the project, it was found that most of the distributed collector designs presented are tightly coupled with the distributed systems they run on. Also, as I understood distributed object-oriented systems more, especially the open source ones that I examined closely (see Section 9.2), I realised that the modifications to be made to a system to incorporate a garbage collector would be substantial.

In understanding garbage collection algorithms, the ones that were found to meet most of the requirements were the ones that relied on the distributed object-oriented system to behave in a particular way, for example, the Stub-Scion Pair chains requires a set of invariants to be maintained by the entire system. Thus an additional, non-trivial goal was added to the project: to implement a distributed object oriented system as well as a garbage collector that ran on this system.

Also, the survey of garbage collection techniques undertaken required thorough understanding the algorithms. This was because implementing such algorithms takes a substantial amount of time and effort, and thus picking the right one to implement is paramount to the success of the project. Hence a lot of time was spent actually understanding designs of many algorithms, before being confident about the understanding of one to implement it. This is in contrast to many other development efforts, where the requirement and design are better understood as the development progresses.

Thus the major deliverables of the project are:

- The implementation of a distributed object-oriented system that allows location transparency to application programmers. It presents the application programmer with the view that all objects, whether local or remote, can be accessed in the same manner. Associated with this is a stub-scion generator, whose implementation is non-trivial, and a naming and registry service, required for getting initial references in a system.
- The implementation of a garbage collection algorithm that is both scalable and efficient in the use of resources.
• A survey of garbage collection techniques that treats algorithms in a uniform manner so as to be able to compare and contrast each of the algorithms from one viewpoint.

Each of the above is discussed with respect to the requirements set out for it in earlier sections (see Chapter 5 for the requirements of a garbage collector).

11.2 SSP Chains in Java Implementation

The implementation of the Stub-Scion Pair Chains, [Shapiro et al., 1992] in Java has the following properties:

• Provides applications with the view that all objects have the same method of access, via local pointers. Remote objects have surrogate objects, which handle the problem of sending and receiving invocations across spaces.

• Provides support for object migration, as the owner spaces of objects do not have to inform all their clients about new locations. This is done via chain indirection, by growing the chain of remote references at the head end. This feature is not found in many distributed systems today, such as RMI. Some, such as Objectspace’s Voyager do provide such support, but at the cost of building it at a higher level in the system, i.e. on top of RMI in their current version.

• Provides a way for client objects to propagate references to other objects via method invocation and results cheaply. This is because the new clients or reference propagators do not need to inform the owner space of new references.

• Provides a short-cutting mechanism that makes remote method invocation efficient, because without such a mechanism, the method invocation would be too costly in terms of time because an indirect chain of references would have to be traversed. This can be seen in Section 8.8 and Figure 8.5. Also this mechanism is lazy, to avoid a costly message passing scheme on first reference export. This enables references are not used not to be short-cut.

• Provides a source code generator that inspects a class from its object code, and generates the stub and scion code from the signature of the methods of the class. This is detailed in Section 9.7.

• Provides a registry and naming service, which interact to enable references to be obtained by client space upon initialisation of the distributed system. Also, services can be offered via names so that clients may be able to look up and access object services. This is detailed in 9.10.

11.3 Distributed Garbage Collection

The requirements of an implementation of the collector set at the outset of the project, described in the outsourcing document and Chapter 5, have generally been met, in design if not in implementation. They are detailed below:

• Requirement 1, the ability to collect all types of garbage, including distributed cycles, has been met by the extension to the SSP protocol, due to [Le Fessant et al., 1998]. The inability to interface with Java’s local collector has meant that some parts of this collector are inefficient, but it has been implemented as proof-of-concept.
Conclusions

- **Requirement 2**: Preservation of Resources. The collector needs to be timed such that memory is not exhausted. This requirement has been met by the distributed collector's liveness property relying on the local collector's liveness, i.e. progress that the local collector makes results in the collection of stubs. This enables the distributed collector to collect scions, and thus remote references are dropped.

- **Requirement 3**: Scalability. The performance tests and their results, presented in Section 10.1, demonstrate the scalability of the collector algorithm. This is mainly due to the algorithm not having the need to exchange information with all the nodes in the system, and not having to synchronise. This is true even of the cyclic extension, which usually would require synchronisation. The acyclic collector only exchanges information between pairs of connected processes, thus the number of nodes in the system do not increase the amount of work, the connectivity does.

- **Requirement 4**: Fault Tolerance. The algorithm manages message duplication and loss, and process failure in an elegant way. This is done by combining logical timestamping, due to [Lamport, 1978], with idempotent messages, which continually send the same information between pairs of spaces. Also, these messages are not large, only 4 bytes per public object connected between spaces.

- **Requirement 5**: Dynamic Configuration. The algorithm does not require any connectivity or space information at the initialisation of a system. Spaces can join by either receiving a method request, or looking up an object's service from the registry. The collector maintains information about a new space by adding an element to the threshold vector. That is all that is required about a new space's information.

- **Requirement 6**: Interoperability. This requirement has been met by design choice, not by implementation. Because all references are handled in a platform independent manner, rather like CORBA references, method invocations between two different systems is possible. All that is required is a standard mapping between primitive types, such as that supported by CORBA's IDL. Then the marshalling functions in the stubs and scions would support these mappings. Also, the semantics of objects is to have an interface with method signatures. Thus it assumes very little from the objects, allowing objects with widely varying semantics to be supported.

- **Requirement 7**: Integration. The algorithm achieves that by relying on the presence of a local collector. Yet the algorithm is very flexible because it does not rely on any design or implementation aspect of the local collector, thus supporting any local collector scheme, including manual reclamation.

- **Requirement 8**: Assuming Reliable Communication Media. This is one requirement given in the project outline, in which additional design work has had to be done not to meet the requirement. In a sense, it is an anti-requirement. As explained in section 8.5, it assumes very little from the transport layer, thus being able to work with the most primitive networks.

11.3.1 Contributions of this Implementation

To my knowledge, the cyclic collector has not been implemented for a common object-oriented language before this attempt, and the algorithm presented in [Le Fessant, 1997] is in very abstract terms, thus the mapping to an implementation
The paper assumes language features such as pointer redirection by the referred object, which do not exist in common languages such as Java and C++.

Several design issues not documented in [Shapiro et al., 1992] also had to be considered when implementing the distributed system and the cyclic collector. Specifically, issues regarding the implementation of strong-chain short-cutting (section 8.8.2) and of weak chain-short-cutting (section 8.8.3) required some original design thought. These were issues such as efficient forwarding by eliminating re-marshalling at every node and additional fields in stubs and scions necessary for forwarding messages when redirecting. Another issue worth mention is the need for weak pointers in the tables that constitute the shared data structures between the object management and distributed clean up protocol (section 9.4.6).

Performance comparisons with Java's RMI show that the implementation is efficient, and in some measures exceeds the performance of RMI, though this is due to design decisions rather than implementation optimisations (Chapter 10). A feature comparison between RMI and the implementation documented here shows that the Stub-Scion-Pair Chains is more efficient in reference duplication as it requires only one message instead of RMI's three messages, it has a collector which is tolerant to process failure while RMI does not have such a collector. Also, with the extension, the key difference is that the RMI collector cannot collect distributed cycles of garbage.

To summarize, the key differences between the collector algorithm implemented in this project and those in other systems are that reference duplication is cheap, the algorithm is scalable because no global information exchange is required. The algorithm is tolerant to message and process failures. The key distinguishing feature of the acyclic collector is that no termination protocol is required as it allows safe inconsistencies by making conservative estimates and allowing floating cycles.

11.4 Distributed Collectors Survey

In the course of conducting research into distributed collector techniques, their design and implementation, the knowledge and material gained have been presented as a survey. This is presented in Chapter 7. Understanding the algorithms at a level of detail sufficient to compare them to other algorithms and judge them against evaluation criteria, is a time consuming task which requires a great deal of research into work in distributed systems as well as memory management.

11.5 State of the Implementation

The implementation developed for this project is in a fully working condition. The Java package contains examples which test various aspects, e.g. bootstrapping references without a name server, using the registry server and client, many different scenarios of reference export, reference duplication, collection of distributed garbage, and the collection of cycles.

As the emphasis was to develop a collector for a distributed system, there are issues with the distributed object-oriented system which if addressed would improve the efficiency and usability of the system:
Sanjay Bhudia

More use of the Flyweight pattern [Gamma et al., 1995] to implement pools of resources. Currently, message dispatch threads are pooled, but the registry daemon threads are not. If pooled, then threads are only created when the thread pool is empty, thus avoiding the high costs and overheads of thread creation. It must be emphasised however, that the distributed system underlying the garbage collection algorithm was developed simply so that the collector would have a platform on which to be implemented, and not to be an efficient system.

Security issues in the registry server, i.e. allowing only the object owners to issue rebind calls. Currently malicious spaces within the system can place spoof server objects in the system. Again, as this was not of concern in the project, it has not been addressed.

Binding flexibility. Systems such as CORBA allow a great deal of flexibility as to when and how a reference gets bound to an object. This implementation only allows early binding. Looking at binding issues is an area of distributed object-oriented systems which currently enjoys a lot of research and thus is beyond the scope of the project.

The Graphics Server. To be able to demonstrate the progress of the collector and reference exports, the development of a graphical front end which displayed stubs, scions and public objects was started. Due to lack of time, the development effort is not complete yet. This was not a promised deliverable in the project outsourcing document, the decision to develop came from the difficulty that would arise in demonstrating the implementation for assessment. As it has not been completed, the difficulty still stands.

Increased concurrency in certain system components. A LIVE message has to be exchanged per pair of spaces and the delivery of a LIVE message is currently done sequentially, by traversing through the set of connected spaces. If a separate thread delivers each LIVE message, the time between the receipt of the first LIVE message of the same batch by a space and the last receipt will be a minimum. This will reduce inconsistency in the system, rather than speed up the delivery of all the messages.

11.6 Lessons Learnt

This project has allowed me to learn a lot about implementing distributed systems in general and distributed reference mechanisms and collectors in particular. The following are some details:

Sockets programming. Due to the nature of spaces communicating via message passing, sockets programming has been an early lesson. The pattern of having a server socket return a plain socket being encapsulated in a blocking method call which instantiates a new thread or obtains a thread from a pool has been learnt at design principles level to implementing it.

Understanding source code of other developed systems. Because of attempts to understand JaCoRB and ORBacus at a source level, I have learnt more about understanding a system at source code level.

Reasoning about distributed algorithms, especially safety and liveness properties. Also, theoretical issues like failure assumptions, causality and how such concepts tie in with implementation.
An appreciation of the subtle issues that face a developer out to implement a distributed system or algorithm. Also, some lessons continued from the [Kramer and Karamanolis, 1999] course, such as the causal broadcast and multicast protocols of [Birman et al., 1991] and their poor scalability. A better understanding between the theoretical study of failures, as introduced in [Hadzilacos and Toueg, 1994] paper and how they are related to practical failures.

11.7 Future Work

I hope to continue to develop the system so that I may be able to understand it and use it, at least in commercial coding, such as pet projects. Also, I wish to be able to interact better with Java's local collector so as to be able to implement the cyclic extension in a more efficient manner.

The addition of a mapping protocol so that interoperability between C++ and Java would be very useful.
Glossary

**Actor:** an entity with an address and a behaviour.

**Acyclic:** unable to manage cyclic data structures.

**Allocator:** the process or method that satisfies a mutator’s request to have a certain number of words of heap memory for occupancy by a data structure.

**Client Space:** the space that holds a remote reference to an object. This space invokes remote methods onto the owner space that holds the object.

**Collector:** a process responsible for collection of garbage.

**Comprehensive:** property of a collector in which all garbage of a system is collected by the end of one collection cycle.

**Concurrent:** processes are concurrent if they may be executed asynchronously without any pre-defined interleaving.

**Conservative:** a collector which overestimates the number of live objects in the system.

**Copying collector:** a collector that copies all the live data to a fresh region of the heap, see scavenger

**Cycle:** a subset of a linked data structure where any cell may be reached from any other cell in the sub-set by traversing pointers.

**De-allocation:** return of space to the storage manager so that it may be re-used.

**Expedient:** property of a collector to be able to reclaim garbage despite parts of a distributed system being not available.

**Explicit de-allocation:** the programmer has control of when and how much space to return to the system for re-use.

**Finalisation:** a clean-up method invoked on an object before it is garbage collected.

**Fragmentation:** heap is not fully occupied, yet it does not contain a contiguous number of free cells to satisfy an allocation request.

**From-space:** the half of the heap from which a copying collector scavenges objects.

**Garbage:** memory in the heap that is no longer required by the mutator, but has not yet been reclaimed by the memory manager for re-use.

**Generation:** a division of the heap according to the frequency with which it will be collected.

**Head-end** (of an SSP chain): The end of a point-to-point chain of stubs and scions that has a scion pointing to the object being referred to.
Heap: a region of main memory where the allocation and de-allocation of objects follows no particular order.

Incremental: an algorithm which is computed in steps, between which it is suspended.

Live: data that is required by the computation or mutator.

Mutator: the process or processor responsible for executing the user program. The name is derived because it changes or mutates the connectivity of the reference graph.

Owner Space: the space that holds the object's state in memory. Its local collector is responsible for de-allocation of this space if it is in the heap.

Public Object: an object that is visible to objects in other spaces, i.e. remote references exist to this object.

Reclamation: the act of returning garbage to the storage manager for subsequent re-use.

Reference count: a count that stores the number of pointers to an object.

Root Set: a set of storage locations which are always live. They may store values or references to other data structures.

Scavenger: A garbage collector which picks the live objects and leaves the rest. It does not actually collect garbage. An example is a copy collector.

Space: the logical node in a distributed system. The definition of a space may vary from the classical, where a thread of control and an address space that is not shared constitutes a space, to a local network of nodes each with a main memory and a process being a single space.

Space-leak: the situation in which part of the heap memory is neither in use, nor reclaimed by the memory manager.

Strong-chain: The point-to-point chain of stubs and scions which is defined by following the strong location in the stub's locator.

Sweep: a linear scan through the heap memory to reclaim unmarked memory cells.

Tail-end (of an SSP chain): The end of a point-to-point chain of stubs and scions that has a stub. This stub is pointed to by an object's local pointer.

To-space: the half of the heap to which the collector copies live objects to. Used in Copy Collectors.

Tracing: visiting or traversing each live cell by following pointers from one or more roots.

Weak-chain: The point-to-point chain of stubs and scions which is defined by following the weak location in the initial stub's locator.
References


Future Work


Conclusions


[GNU] **GNU General Public License**, Free Software Foundation, Inc., Boston, MA (www.fsf.org) ........................................................................................................... 110


[Kramer and Karamanolis, 1999] **Professor Jeff Kramer and Dr Christos Karamanolis**: "Distributed Algorithms", 2 part MEng 4 course number 4.37, taught at Department of Computing, Imperial College, 1999.  


---

MEng Software Engineering, Imperial College
Conclusions


[MCC] Microelectronics and Computer Technology Corporation: (www.mcc.com), a consortium owned by over 32 corporations, including NASA, NSA, Lucent, HP, DEC, GE, Bellcore etc........................................................... 24


Sanjay Bhudia
Future Work


[Piquer, 1991] José M. Piquer: "Indirect Reference Counting, A Distributed Garbage Collection Algorithm", in Parallel Architectures and Languages Europe, number 365, 366, Lecture Notes in Computer Science, pp 150—165, Eindhoven, the Netherlands, Springer-Verlag, june 1991............... 61, 64, 81


Conclusions


[TECCware] Trans-Enterprise Computer Communications (TECC): "TECCware, Extensions to the Carnot program's agent based language, Rosette". ................................................................. 24


Sanjay Bhudia


