**Vrije Universiteit Brussel - Belgium**
**Faculty of Sciences**
**In Collaboration with Ecole des Mines de Nantes - France**
**and**
**Universidad de Chile - Chile**
**2001**

# Comparison between different Tree Flattening techniques in an Indirect Reference Listed DGC

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Kristof De Vos

Promotor: Prof. Théo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Prof. José M. Piquer (Universidad de Chile)

# Abstract

Distributed Garbage Collectors can be divided into several groups, according to their way of collecting. this thesis proposes some extensions and optimizations to one of these DGCs, the Indirect Reference Listed DGC. One of the problems with IRL is that the diffusion tree can be badly shaped which makes fault tolerance more difficult. This thesis proposes two algorithms to improve the overall shape of the diffusion tree such that a failure of a node has less impact on the tree.
Orthogonal on these tree flattening techniques, a new way of managing the diffusion tree when performing a migration is proposed. instead of altering the diffusion tree, it will remain the same. This is better when combined with tree flattening techniques.

# Acknowledgments

As with many big projects, this thesis could not be accomplished without the help and support of many other persons. First of all I want to thank my advisor Jo Piquer, who helped my a lot, especially on the details and validation of my algorithms and thoughts. Next Marc Ségura-Devillechaisse helped me many times during the actual thinking and practical problems. Each time I was doubting between different possibilities, I asked his opinion. Same holds for Sofie Goderis, who always had a listening ear when anybody needed that. Of course I also want to thank my friends of the AccessNova-lab where I stayed 4,5 months. And of course I may not forget to thank the family of Victor Hugo, who accompanied me (and Marc) from the first hour we arrived in Santiago until the last hour when we left to come back. Thank you Laura, José Miguel and Sergio.

# Contents

**1 Introduction** **6**

**2 Context** **9**
2.1 Model . . . 9
2.2 DGC Messages . . . 10
2.2.1 Increment Message . . . 11
2.2.2 Decrement Message . . . 11
2.3 Conclusion . . . 11

**3 Garbage Collection** **12**
3.1 Distributed Garbage Collection . . . 13
3.1.1 Reachability . . . 13
3.1.2 Similarities and differences between lgc and dgc . . . 14
3.2 DGC Strategies . . . 16
3.2.1 Distributed Mark And Sweep . . . 16
3.2.2 Garbage Collecting Actors . . . 16
3.2.3 Reference Counting . . . 16
3.3 Conclusion . . . 19

**4 Tree Flattening Techniques** **20**
4.1 Basic Problems . . . 20
4.1.1 Partial failure . . . 20
4.1.2 Migration . . . 21
4.1.3 Zombie-references . . . 22
4.2 Tree Rerooting, by Dr. Luc Moreau . . . 22
4.2.1 Basic Algorithm . . . 22
4.2.2 Detailed Algorithm . . . 23
4.2.3 Remarks . . . 23
4.2.4 Critical View . . . 24
4.3 Diffusion Tree Restructuring, by Dr. Peter Dickman . . . 24
4.3.1 Basic Algorithm . . . 24
4.3.2 Detailed Algorithm . . . 25

4.3.3 Remarks . . . 26
4.3.4 Critical View . . . 26
4.4 Conclusion . . . 26

**5 Proposed Tree Flattening Techniques** **28**
5.1 Introduction . . . 28
5.2 Adding or duplicating nodes . . . 29
5.2.1 Minimize-by-insertion . . . 29
5.2.2 Arguments . . . 31
5.2.3 New invariant: Grandfather . . . 32
5.3 Deleting nodes . . . 33
5.4 Migration . . . 34
5.4.1 Tree is growing . . . 34
5.4.2 Solution . . . 35
5.5 Conclusion . . . 36

**6 Validation** **37**
6.1 Introduction . . . 37
6.2 Simulation Model . . . 37
6.2.1 Adding a new node . . . 37
6.2.2 Migration . . . 40
6.2.3 Statistics . . . 40
6.3 Implementation . . . 42
6.3.1 PicObj . . . 42
6.3.2 Java RMI-clone . . . 43
6.3.3 Original Java RMI . . . 43
6.3.4 RMI-clone . . . 44
6.4 Benchmarks . . . 44
6.4.1 Benchmark Programs . . . 45
6.4.2 Uni-processor Results . . . 47
6.4.3 Multi-processor Results . . . 50
6.5 Conclusion . . . 52

**7 Implementation Details** **53**
7.1 Introduction . . . 53
7.2 General Framework . . . 53
7.2.1 Manager . . . 53
7.2.2 Protector . . . 54
7.2.3 Space . . . 55
7.2.4 WireRep . . . 55
7.2.5 Registry . . . 56
7.2.6 Messages . . . 56
7.2.7 Serialization . . . 56

7.3 PicObj . . . . . 56
7.4 Java RMI-clone . . . . . 58
7.5 Additional Details . . . . . 58
7.5.1 Interleaved Virtual Machine . . . . . 58
7.5.2 Local Garbage Collector of Java . . . . . 59
7.5.3 Configuration . . . . . 59
7.6 Conclusion . . . . . 60

**8 Future Work** **61**
8.1 Benchmark suite . . . . . 61
8.2 Applicable to other DGCs . . . . . 61
8.3 Delaying or not sending DEC-messages . . . . . 61
8.4 Combining LGC and DGC . . . . . 62
8.5 Fault-Tolerance . . . . . 62
8.6 Conclusion . . . . . 62

**9 Conclusion** **63**

# Chapter 1

# Introduction

The application domains in computer science have changed a lot in the last 10–20 years. In the early years, the first computers were used in the army to calculate ballistics, later their domains were broughtened to general calculations (number crunching) to help the work of mathematicians (still in the army or governments). When computers became smaller, cheaper and easier to use, they became available for general purpose.

Since the beginning, the processing capacities of these computers grow larger and larger every day. According to the law of Moore, the speed of the computers (processors) will double each 18 months. Until now, this law still hold.

Many ”revolutions” happened in this field. Not on hardware-level, but on software- and higher levels. The tasks where the computers are used for, have changed many times in the past. In the beginning, huge mainframes are used in companies to do the hard work and thin clients are used to collect the information and instruct the mainframe.
The first revolution happened when computers became smaller and more powerful. Then many computers were placed in different places, where they were needed. Now everybody who needed a computer could have access to it.
Since then, the work is distributed among the different computers, this creates problems if data has to be shared between computers. The second revolution happened when networks were used to connect several computers. This enabled sharing of data and later sharing of processing power.
Currently, this sharing of computer power becomes more and more general. This enables the possibility to combine the processing powers of many computers to do computations on a level that was impossible to be achieved by one computer. On the other hand, it also allows you to specialize certain computers for specific applications (webservers, application servers, database servers...) and make them accessible by all other computers.

Although resources on these computer are growing faster than we read this text, they are still limited. When resources are used on other machines, they must be released in a similar way as on the local machine. One of these resources (subject of this thesis) is memory. Similarly to local management, there are two strategies to do the work:

- free manually
- automatic garbage collection

This thesis will only discuss the second option because at the first option it is completely up to the programmer to do the cleaning.
To do this automatic distributed garbage collection (also called Distributed Garbage Collection or DGC), several algorithms are thought of. One of them is Indirect Reference Listing, which is an extension of reference counting in normal local garbage collectors. Each time an remote pointer to an object is sent to another processor, the original processor adds that processor to the list of processors that contain this pointer. This list prevents the original object to be garbage collected. The algorithm is called indirect because this management does not only happen in the processor where the object is located, but also in other processors that know the object. This distributes the management of keeping track which processors contain references to the object. The whole set of processors containing a reference set for an object is called the diffusion-tree.
The advantage on these indirect algorithms is that they can reduce the number of messages needed for the DGC (see section 3.2.3). The most important disadvantages is the creation of third-party-dependencies: In some cases the diffusion tree contains nodes where the object is not referenced anymore, but the node is retained to keep the tree consistent. In other cases the tree constructed by the computation can be very unbalanced, making it more vulnerable to failures.
Tree flattening algorithms have as their goal to reorganize the tree to improve their global state. This can be by removing third-party-dependencies or generally reshape the tree to be more balanced.

Several authors have proposed different solutions to this problem: Moreau proposes an algorithm that moves all nodes located at deep levels up to the first level. This creates the ideal tree, but on a cost of many messages. Dickman proposes a number of different algorithms that will reshape the tree in different ways. The first is while performing a Remote Procedure Call, it will move the node who invoked the call to become a direct child of the root-processor. The second idea is to choose the best father when a second father for this processor appears. Both algorithms will be explained in depth in chapter 4.
This thesis will propose two different approaches to solve the same problem. The first will delete so-called Zombies, processors that are kept in the tree while they do not contain references to the object. This will shrink the tree in a drastic way, but depending of the computation (whether there are zombies or not). The second algorithm will try to eliminate deep diffusion trees. When the diffusion tree depth goes beyond a certain depth, it will trigger the algorithm and reroot a node to shrink the tree. The idea is the same as Moreau's, but less drastic.

This thesis will start with some general information about the context in which all following chapters will be handled (Chapter 2). Chapter 3 will explain general principles of garbage collection and more specific distributed garbage collection. At the end several common DGC-techniques are explained with their advantages and disadvantages. The next chapter (Chapter 4) explaines the problem of the diffusion trees, and the tree flattening techniques to solve these problems. Both Moreau's and Dickman's ideas will be explained here. In Chapter 5 the new ideas of this thesis are exposed: Minimize-by-Insertion (first tree flattening technique), Zombie Deletion (second technique) and a new migration-technique. In the next chapter (Chapter 6) different arguments are shown to compare Moreau's algorithm with the new algorithms of this thesis. First a model is explained that simulates both algorithms, next two different implementations are explained. Finally different benchmarks are shown to compare the performance of plain IRL, Moreau's algorithm and the two new algorithms. Chapter 7 goes deeper into implementation details of the DGC and both tree flattening algorithms. Chapter 8 gives some other ideas, which are not captured in this thesis, but are interesting for future works. Finally Chapter 9 will give a general conclusion for this thesis. In appendix, several source-code-examples can be found for the RMI-clone.

# Chapter 2

# Context

Many different garbage collecting techniques have been proposed in the last 20–25 years. Some of these algorithms are specially designed for specific environments (e.a. functional languages, uni- or multi- processors, specific languages...).
To be able to reason about these different algorithms, and more specific about distributed garbage collecting algorithms, it is necessary to describe the global model in which they will be used.
The model described below will be used to describe the different algorithms found in literature and the new ideas proposed in this thesis.

## 2.1 Model

The model used throughout this document will be composed of a set of logical different spaces or processors (further denoted as P), with direct, reliable communications between them. Physically, some logical spaces can be located on one single processor, but this does not change the behavior of the spaces. They keep communicating through the same mechanism.
The model consists of a set of objects (further denoted as O). Each object o resides in exactly one space p ε P. This space p is called the owner of o.
To allow accesses to remote objects, ”remote references” are introduced (also called remote pointers). A remote reference is a logical pointer pointing to an object that is owned by another space. A remote reference to an object o is called an o-reference. The complete set of all remote references to one specific object o is denoted RP(o). Local pointers to this object will not be included in RP(o) because these are of no interest for the model.
Because this model allows migration of objects, the owner of an object can change at runtime. RP(o) includes all current existing o-references, including in-transit remote references (these are remote references which are sent to a space, but not yet arrived). The model assumes that these pointers are always accessible.
An additional requirement is that each processor can have at most one o-reference to a certain remote object. This is not a necessary requirement, but it simplifies the implementation without any big additional cost. Usually, remote references go through a local indirection that enforces this unique o-references. As a consequence, a local object containing a (logical) remote reference, actually contains

a (physical) pointer to an o-reference (also called stub).
The model incorporates four different operations with respect to o-references:

- creation
  The owner of an object o, sends an o-reference to another processor p. For example a parameter in a message.
- duplication
  Another processor (other than the owner) sends the o-reference to a third processor.
- deletion
  A processor does not need the o-reference any more and discards it.
- migration
  A processor which already has an o-reference to an object o, will become the new owner of o. This operation will consist of a number of subtasks:
  - real movement of the object itself
  - old owner changes old object into an o-reference
  - new owner changes its o-reference into the real object
  - possibly many duplications by adjusting all internal pointers of the object itself
  - inform all spaces where o-references exist that the object has moved

To keep the model simple and consistent, pointer redirections will not be allowed. An o-reference always points directly to the object, not through an indirection at another space.
Although this model contains only four primitive operations, it has been used by many researchers and is considered to be powerful enough for most existing distributed systems. This model only shows the general idea, but the implementation will be more complex and possibly system or language dependent.
The last requirement is that the communication between processors must be reliable and preserve the order. All messages sent **must** arrive, and in the **same order** as they were sent. [1]

## 2.2 DGC Messages

The distributed garbage collector uses several kinds of messages to inform other nodes of changes in the global state. For example if a new space receives a remote pointer for the first time. Depending on the used algorithm, these messages are used or not used and possibly sent to different spaces.
The most important messages are Increment and Decrement messages:

[1]This model, including the o-reference terminology, is adopted from Lermen and Maurer (1986), Piquer (1991) and Tel and Mattern (1993)

### 2.2.1 Increment Message

An increment message (also called INC-message or dirty call) is sent by a space when it receives an o-reference for the first time. The increment message is sent to the owner space of the o-reference and is used to let the owner know that this space contains a reference to this object. This message is used in Distributed Reference Counting algorithms. Later DGCs have techniques to do not need the sending of these increment messages (e.a. Indirect Reference Counting).

### 2.2.2 Decrement Message

The inverse of the increment messages is the decrement message (also called DEC-message or clean call). It is sent when that space no longer uses that o-reference, and informs another space. In normal reference counting algorithms it will inform the owner space (see section 3.2.3) , but indirect reference counting algorithms will sent it to their father in the diffusion tree (see section 3.2.3).

## 2.3 Conclusion

Now we have a general definition of the model that will be used throughout this thesis. All DGC-algorithms and optimizations will be explained based on this model.
Only two messages are explained because these are the most general used in different DGC-algorithms. There exists lots of other messages, but each one of these is targeted at a specifical DGC, therefore they will be explained together with the DGC.
In the next chapter, a general introduction to Garbage Collection is written, followed by more specific details about DGC. Finally the most common DGC-algorithms are explained.

# Chapter 3

# Garbage Collection

In the beginning of the programming-era, programmers communicated with the machine on a bit-level by switching simple switches. Shortly after, this was improved to a simple input-system that accepted hexadecimal values. The next step was the invention of mnemonics to help the programmer to understand his own program. Nevertheless, until then the programmer was in charge of arranging every detail of his program (e.a. calculating offsets and absolute addresses of instructions).
Later, high-level languages start to appear. They contain compilers that transform user-programs into low-level programs ready to be used by a computer. The biggest advantage of compilers is that this bookkeeping of offsets, variable addresses and other things is done in an automatic way without help from the programmer.
These compilers (which are still used nowadays) can allocate variables in three different ways:

**Static Allocation** A variable is bound to a specific address at compile-time. The mayor problems with static allocation is that the size of the data structure must be known at compile time and no procedure can be recursive because values would be overwritten.

**Stack Allocation** The problem of recursiveness was solved by the invention of stacks. Each time a function was called, its variables were put in a new stack-frame on top of the stack. After finishing the function, that frame was discarded. Although now recursiveness is introduced, it still has some limitations such as that the size of the return value must be known at compile time and no objects can outlive their stack-frame.

**Heap Allocation** To solve this last problem, the heap was invented. Here data structures can be allocated and deallocated at any place in the program. This heap allows the programmer to create and maintain data structures in a more natural and easier way. Now the size of data structures is no longer fixed and it can give programming languages more flexibility (like allowing to return functions as result of a function)

Each of these three allocations manage their memory in a different way.
Static allocation determines the address at compile time and never releases that specific address, it will be kept until the program terminates.

Stack allocation creates a new stack each time a function is called. When this function reaches its end, its stack-frame is popped from the stack and all variables in it are lost.
Heap allocation can have two different managements. All languages need a specific call to claim a new piece of memory. In languages like C or C++ the programmer must explicit inform the management that it does not need this piece of memory anymore. This can have two incorrect consequences: Memory is released too early (other data structures still have a pointer to it) or is never released (memory leak). Both problems are very hard to track down because they do not always generate immediate errors on execution. Alternative languages like Java, Lisp or Smalltalk have an automatic management module that takes care of memory pieces that are no longer useful for the program. This module is the local garbage collector (from now on denoted lgc). An object is no longer useful if it is no longer reachable from another reachable object. Local reachability is defined by a set of roots. When an object is no longer referenced from these roots, it is candidate for local garbage collection (= not reachable). The lgc can now safely free the memory occupied by this object.

## 3.1 Distributed Garbage Collection

The use of networks and more especially of the internet in applications becomes more common every day because it can give a lot of power and freedom to the programmer and user: information can be spread out over multiple machines, calculations can be made on specialized computers... This is called distributed computing [Tan88].
When objects are distributed over several computers, unchanged normal garbage collection can not be applied any more. The problem is that a remote reference can reference an object o in a certain process. Even if this object o is no longer referenced locally, it may not be freed because there exists a remote pointer to this object in another processor.
The purpose of a Distributed Garbage Collector (from now on denoted DGC) is to free these objects only if there are no local **and** no remote references to the object. Local garbage collectors check the local memory and free the memory that is no longer used. A Distributed Garbage Collector usually works the other way: it prevents certain objects (which are remotely referenced) to be garbage collected prematurely. Once it finds out that they are not referenced remotely any more, it gives the responsibility of collecting the object to the local garbage collector. If it is also not locally referenced, it will be collected (eventually).

### 3.1.1 Reachability

To be able to know when an object can be freed or not, we need a slightly more formal way of defining reachability in distributed systems.
Luc Moreau defines global reachability for an object o, if one of the following three conditions is true[Mor01]:

- if o is locally reachable
- if there is a globally reachable object that locally references o

- if there is a globally reachable object that contains an o-reference

### 3.1.2 Similarities and differences between lgc and dgc

Although the goal of both garbage collectors is the same (collecting garbage), their way of solving it is different. If one would apply lgc-techniques unchanged to distributed garbage collection, it would cause errors, undesired delays or even deadlocks [LSB92].
Because it is a garbage collector, it must fulfill these requirements:

**comprehensive** Collect garbage (lifeness)

**correct** Collect only real garbage (safety)

**expedient** Memory should be freed at a speed high enough to fulfill the needs for memory

**efficient** time and space overheads should be minimized

Distribution adds another requirement to this list:

**concurrency** To allow different processors to do things simultaneously

#### local and global objects

To be able to garbage collect an object, the collector has to know whether it is still referenced or not. Most dgcs therefore divide the whole set of objects in two subsets: local and global. An object is global if it is still referenced from another processor, otherwise it is local. During execution an object can switch between these sets. This change of status is handled by the dgc, which decides whether this object will be protected (cannot be freed) or whether the lgc can free it safely.

#### Cycles

In some cases, cyclic structures of garbage can exists. Especially when these cycles are distributed over a number of computers, they can present a problem for a dgc, because each part of the cycle is still remotely referenced (by another part of the cyclic garbage), therefore preventing this part of being collected. These cycles are called **global cycles**.
Most dgcs do not reclaim global cycles because the cost of collecting these is considered higher than the profit of claiming a relatively small number of objects. Other dgcs solve the problem by migrating the parts of the cycles to the same processor, where the lgc can take care of them. A few dgcs can collect them themselves.

#### Reactive vs. Tracing algorithms

An algorithm is called reactive if a change in the state of an object triggers the dgc to free the same and/or other objects. Tracing algorithms are only ran at certain time intervals or if extra memory is needed.

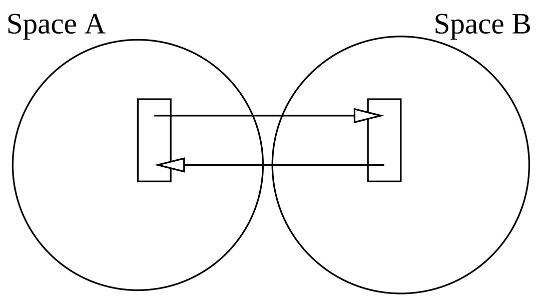

Figure 3.1: Cyclic Garbage

In general, tracing algorithms are more expensive on communication overhead, because they need a lot of node-traversals to decide whether an object is globally reachable or not. On the other hand, they can be more powerful (e.a. by claiming cycles).
A third kind of DGC exists: hybrid. This uses a reactive algorithm to claim most of the garbage, but also, infrequently, starts a tracing algorithm to claim cycles.

### Synchronization

Because local information is not enough to decide whether an object can be reclaimed or not, the DGC has to communicate with other processors to get the necessary information. Because in this distributed environment multiple events can happen at the same time, the DGC must synchronize with other processes at a certain point to decide. The easiest way is the stop-the-world-protocol, where all computation is block until the DGC has decided. For reactive or real-time applications is this unacceptable. More sophisticated algorithms only deny certain operations to happen concurrently with the garbage collection. This allow more concurrency, by blocking less processes.

### Robustness

Distribution adds another aspect to garbage collection. On a single processor, all messages arrive in the same order they were sent, but this is not always the case in networks. Some messages can arrive twice, in another order or not at all.
In each of these cases, the DGC must prevent that non-garbage is collected and that garbage is not kept forever.
Another problem is that a certain processor can stop responding for a certain time or it can crash. The difference between these two states is very hard (if not impossible) to detect by another processor. This makes it much harder for the DGC to know which objects that processor referenced (or still references), and to decide what to do with them.

## 3.2 DGC Strategies

To solve the problem of collecting global garbage, many different techniques are proposed. This section will give a short, general introduction to the three most common used techniques and then will dig further into optimized or extended versions of Distributed Reference Counting.

### 3.2.1 Distributed Mark And Sweep

In this category, two different subdivisions can be made: DGCs which use a distributed version of the local Mark And Sweep, and DGCs that are based on Dijkstra's on-the-fly Mark And Sweep algorithm [DLM$^+$78, DS80].
Both DGCs works by maintaining a mark-tree to which the DGC adds branches. This can be best compared with the marking-phase of a local mark-and-sweep GC. Initially all nodes are colored white. The DGC starts at the root and traverses in parallel all its children. These are colored grey. When a node has no children or they are already visited (= colored grey) or when they all have acknowledged, it is colored black and sends an acknowledge to its father. When the root is colored black, the marking phase is finished and all white nodes can be deleted safely.
disadvantage:

- creates lot of network traffic each time the DGC is ran
- needs to know all nodes (can be a lot)
- difficult to implement synchronization between phases

### 3.2.2 Garbage Collecting Actors

Each actor[Hew77] has by definition its own thread of control, memory associated with it and a communication mechanism (mailboxes). Root actors are considered to be always running and are connected with I/O devices to communicate with the outside world. An actor can be considered garbage if its presence cannot be detected by the outside world. The idea is that garbage actors cannot become non garbage because their mail addresses are not know by another non garbage actor.
Most DGCs for Actors look very similar to Mark and Sweep or Reference Counting.

### 3.2.3 Reference Counting

This algorithm is based on the concept of reference counters as in early lgcs. The main idea is that each remote accessible object contains a counter. This counter represents how many processors are currently referencing this object. When a processor receives a new remote reference, it sends a dirty-call to the owner. This dirty-call (also called decrement message or DEC-message) will increment the counter. When this processor does not need the reference any more, it sends a decrement message to the object to decrement the counter.

When an object is not locally referenced any more **and** the counter equals zero, this object may be collected safely by the lgc.

disadvantage:

- generates lots of network traffic
- no fault tolerance: if a processor containing an o-reference crashes, the object o will never be freed
- if causality is not preserved by the message system, the algorithm is not safe

A lot of optimizations and extensions are added to ”normal” reference counting. The most important are shortly described below:

### Weighted Reference Counting

This is an improvement on the normal Reference Counting to avoid sending dirty-calls. Instead of having a counter, an object has a weight. Initially it contains a certain value X. Each time a reference to this object is transferred to a new processor, this weight is split and divided over the two processors. When the o-reference is no longer needed, it will return its local weight back to the original object.
The idea is that an object can only be collected if there exist no local references **and** the weight is X again (no o-references).
advantage:

- generates less network traffic
- needs no causality in the network

disadvantage:

- problems when the weight can not be divided any more
- no fault tolerance: if a processor containing an o-reference crashes, the object o will never be freed

### Indirect Reference Counting

This new technique is very close to the original reference counting, but now the stubs also contain counters [Piq91]. Each stub contains a reference to its father (from which it received for the first time the o-reference) and a counter.
Now, each time an o-reference is duplicated to another processor, the **local** counter is increased (thereby avoiding network traffic).
The stubs can be cleaned only if there are no local references to it, **and** the counter equals zero. Before destroying itself, it sends a clean call to its father. The object itself can only be collected if there exist no local references **and** the counter is 0 (no o-references).
advantage:

- generates less network traffic (avoiding dirty calls)
- no weight division - problem
- supports object migration
- needs no causality in the network
- simple to implement

disadvantage:

- no fault tolerance: if a processor containing an o-reference crashes, the object o will never be freed
- can create zombie-references: stubs which are not locally referenced, but only exist to preserve the tree (because they have children)

### Indirect Reference Sets

Indirect Reference Sets are closely related to IRC, but instead of counting how many children a node has, it keeps a set of the spaces of its children [PV98]. The size of this set is equal to the reference count in IRC.
Each time the reference is sent to another space, this space is added to the set. It is even added if that space was already in the set. This is necessary to ensure safety (see section 3.1.2).
Similar to IRL, stubs can only be collected if their reference sets are empty (same for the real object).
The biggest advantage is that each stub now knows exactly which spaces are its children. In IRL it only knows how many their are but if one of them fails to send a decrement message, the stub will never be reclaimed. The DGC can now use this information to determine whether its children are still alive or not and do some clean up if it knows that a space became unreachable.
advantage:

- all advantages of IRL
- better for fault tolerance, but not perfect because children of a crashed node can not be recovered

disadvantage:

- can create zombie-references: stubs which are not locally referenced, but only exist to preserve the tree

## 3.3 Conclusion

This chapter explained the basic details of local garbage collection and distributed garbage collection. After explaining all the necessary details of DGC and especially the encountered problems, different DGC-algorithms were explained with their advantages and disadvantages.
The rest of this thesis will focus on the IRL and IRC-algorithms. Next chapter will go more in depth in the problems created by these algorithms and will explain two different solutions proposed by two researchers (Moreau and Dickman).

# Chapter 4

# Tree Flattening Techniques

This thesis will primarily deal with optimisations and extensions to Indirect Reference Listing. Some of these techniques can also be applied to Indirect Reference Counting or maybe even to other DGC-algorithms.
This chapter will first explain the basic problems encountered when using indirect reference listing. In the next sections different diffusion tree flattening techniques are explained.

## 4.1 Basic Problems

Indirect algorithms were invented to reduce communication interaction because no increment messages need to be sent. This new idea works fine, but still has some minor negative points.

### 4.1.1 Partial failure

When a large number of computers are involved in a computation and especially when this computation take a long time, some of these computers can crash or become temporarily unavailable. When using indirect algorithms, the diffusion tree is spread all over the computers involved in the communication. When a computer fails in this system, this diffusion tree can be corrupted.
When space C now fails for some reason, it breaks the diffusion tree in three different subtrees and it is impossible to reattach the lower subtrees to the original tree because no node in the upper tree knows that the nodes of the lower subtree were part of the original tree.
Different techniques can be used to solve this problem the best way possible. Some techniques send redundant information to other nodes, which will reclaim the node when its father has died. Another technique is that when nodes detect that their father is dead, they will ask to be adopted by the root. Many solutions exist, but until now, none is considered as being the best in all cases.
Diffusion tree flattening techniques are used to minimize the number of critical nodes which may cause problems when they crash. This is particularly important in long-lived systems where it is crucial that real garbage is also reclaimed. Otherwise these systems could run out of memory at some point.

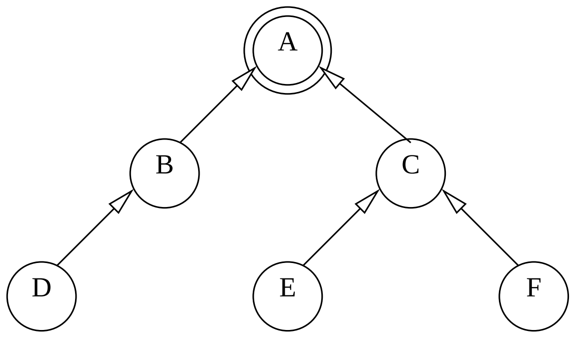

Figure 4.1: Diffusion Tree

### 4.1.2 Migration

Migration is the ability of objects to move from one space to another. Considering the GC, this is an interesting challenge because even when the object moves around, the information needed to collect the object at the right time (not too early, not too late) has to be adapted to the movement of the object. Indirect Reference Counting, as it was first proposed by Piquer, introduces the ability to let objects migrate in a very easy way.

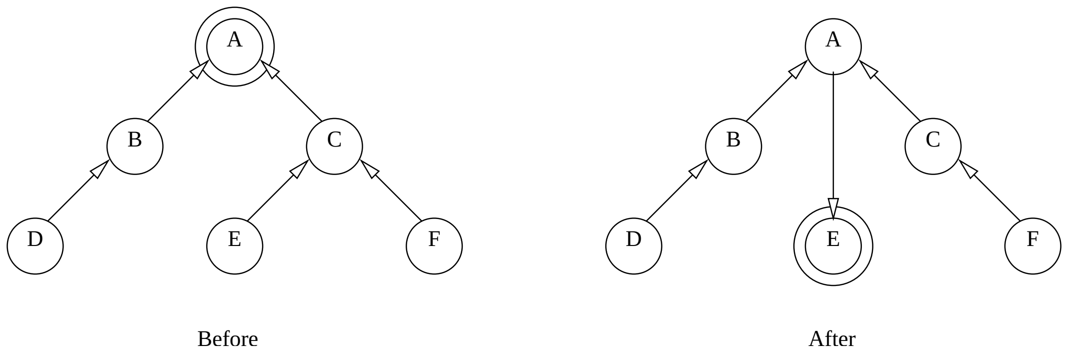

Figure 4.2: Migration

When the object moves from space A to space E, E becomes the root of the new diffusion tree. Now E is the father of the old root and it will send a cleancall to his old father. So with only two messages the tree is restructured to make node E the root of the diffusion tree.
Although this sounds very exciting at first sight, there are some problems with this technique. This migration that changes the tree, can change the tree is a negative way, making it more deep, which is bad for fault tolerance (see previous section). When a computation needs a lot of migrations, this can reshape the tree in a very bad way, annihilating all tree flattening actions.

### 4.1.3 Zombie-references

Another negative point are the so called Zombie-references. These are stubs in spaces which are not locally referenced, but still have children in the diffusion tree. It is unsafe to delete them because they are needed to keep the tree consistent (their children need to stay in the tree).
Zombie-references are not favorable because they create unnecessary dependencies on these spaces. When a space, containing a zombie-reference fails, the tree will become inconsistent (as explained in previous sections). Some tree flattening techniques are targeted on not allowing these zombie-references or cleaning them as soon as they occur.

## 4.2 Tree Rerooting, by Dr. Luc Moreau

The problem with diffusion trees is that they can be very deep, which is bad for fault tolerance (many zombie o-references) and prohibits a fast breakdown of the tree when the last reference has became garbage. Moreau solves this problem by applying an algorithm that puts all nodes in the diffusion tree on the first level. If a node is created at a second or deeper level, it will be moved (rerooted) to become a direct child of the root. Once all these reroots have finished, the tree only contains one level with all the nodes in it.
Originally, Moreau proposed this algorithm to work with IRC. He successfully implemented his algorithm in a distributed scheme and statistics are available in his papers [Mor98, Mor01]. Nevertheless, this algorithm can easily be adapted to work for IRL. The algorithm-section will explain the plain IRC-algorithm, but the reader can easily adopt it to IRL.

### 4.2.1 Basic Algorithm

The algorithm consists of a sequence of two messages to be sent for each rerooting. The first is initiated by the new node C (which is at minimum at level 2), and is sent to the root of the diffusion tree (A). Here the new node requests the root to become his father. If the root accepts this, it will send another message to the original father (B) to decrement his counter.

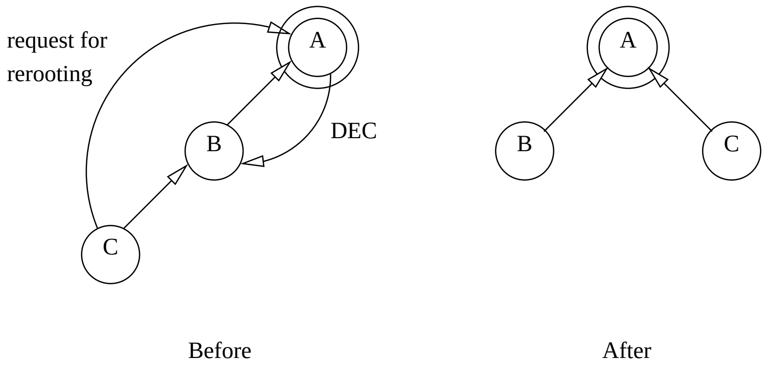

Figure 4.3: Moreau's Algorithm

The key idea that holds the correctness of the algorithm is that the decrement message is always sent after the counter at the root was incremented , this way the original object can never be freed incorrectly.

### 4.2.2 Detailed Algorithm

This section will go deeper in the needed constructions to let the algorithm work correctly.
The algorithm assumes a receive-table containing all the remote pointers known by this process. This table is used to lookup newly received remote pointers to know whether they are already known or are new at this space. This table is not used as a root of the local garbage collector to avoid interference with the definition of global reachability.
To have a clear view on the algorithm, figure 4.3 shows an example. In this case A created an object o and sent its remote pointer to B who forwarded it to C.
Each time a remote pointer is sent, its local counter is incremented. When a remote pointer is received, it is put in the receive table. If it was already there (so it was already known), a clean call is sent to the last sender.
Now, if a remote pointer is received from a process other than the root and it was not already known, tree rerooting is initiated. This is the case of C receiving the pointer of B. Now a INC_DEC - message is sent to the root (A). This message contains also a reference to B which is the current father of this pointer in C.
When the INC_DEC arrives at the root, it increases its counter and creates a second (DEC) message to the original father (which is known by the INC_DEC - message).
At last, the DEC message arrives at the original father (B), which now safely decrements his counter.

### 4.2.3 Remarks

The key idea is to safely propagate reference counts to the root. This is accomplished by only sending a decrement after the counter is incremented at the root.

This algorithm does not change the sum of reference counters in a stable configuration. Although the sum can be slightly higher than it should be, this is only a temporary problem and will be solved by future arrivals of DEC-messages.

Not represented in the tables, but necessary for the algorithm are send-tables. Each time a pointer is sent to another process, the pointer is entered in this table. Because this table is a root for the LGC, it protects the pointer from being garbage collected if there are other processes with this process as father for this pointer. When a counter for a remote pointer reaches zero, the pointer is removed from this send-table. Only if it is also not locally referenced any more, it can be garbage collected (= sending a DEC to its father). This send-table keeps pointers alive, even while they are in transit.

Important is also that the sending of these INC_DEC-messages is asynchronous with the actual computation. This means that they can be delayed or even completely neglected. This way, the algorithm

can be turned into a wide variety of algorithms, going from very aggressive rerooting up to plain IRC (if they are neglected).
Moreau also stresses the potential of parallel execution of the DGC-messages. If both extra messages are processed while that processor is doing nothing else, it hardly means an extra cost to process them.

### 4.2.4 Critical View

Although the algorithm is simple in its basic form, things become more complicated when multiple messages are created in parallel. For example if a remote pointer initiates a rerooting and is deleted at the same time, causing a decrement message to its father. If this is handled incorrectly, it can cause the real object to be freed prematurely. Although this can be solved, it creates more complexity than the original algorithm indicates at first sight.

More important is that each rerooting-action needs a sequence of two messages. In some situations the cost of sending these two messages can be higher than the gain obtained. In this case it is not appropriate to send them. Unfortunately it is very difficult to predict exactly when to send them and when to not send them.

The tree-rerooting algorithm consumes resources on different areas. First of all two messages are sent. This requires bandwidth on the communication channels between the three processes. In case of a simple ethernet local-area network, this means claiming the whole network-segment to send the message, and delaying the sending of other messages. If switches are used, only the link between the two corresponding objects is claimed, but this is an expensive solution.
Secondly, also processing power is consumed. This has an effect on other processes running on the same processor. If these messages can be handled while the processor is idle, nothing is lost, but otherwise it means delaying the other processes.

## 4.3 Diffusion Tree Restructuring, by Dr. Peter Dickman

Dickman uses another approach to flatten the diffusion tree. His ideas are formulated into two different, but coherent algorithms which try to lower the depth of the diffusion tree [Dic00]. His algorithms are primarily based on already present communication between processes.

### 4.3.1 Basic Algorithm

The basic idea behind all these algorithms is that the original father (obtained by the first arrival of the remote pointer at that process), does not have to remain the father. For instance, while performing a remote method invocation on the real object, it can use this communication to reroot itself.
The idea consists of two basic algorithms and two additional extensions to it:

**sub-tree re-rooting during RPC** Each time a process invokes a remote method on the object, it creates a communication. The original object can detect that the originator of the method call is not located as a direct child of the root. Before returning the RPC, the reference counter is incremented to ensure the safety of the algorithm. When the result arrives back at the original process, it can send a DEC-message to its original father.

**Depth-reducing parent selection** If a remote pointer is received for a second time, plain IRC sends a DEC back to that process. This algorithm chooses the best father and sends a DEC to the other father, possibly changing it. It is assumed that each node knows his depth in the diffusion tree. The node with the lower depth is the best father for a node.

## 4.3.2 Detailed Algorithm

#### Sub-tree re-rooting during 'RPC'

The result of this algorithm looks very similar to Moreau's algorithm because it reroots a sub-tree. Although, this rerooting is encapsulated in a RPC-call, causing no extra messages to be generated. The message itself is extended with one bit, which indicates whether the original process asks for a rerooting or not. When an RPC arrives at the root and the bit is set, the local reference counter is incremented. Only after this increment, the RPC is allowed to return the result. Encapsulated in the result, there is a bit telling if the rerooting has failed or not. If it did not fail, the original process will send a DEC to his original father and change its father.

#### Depth-reducing Parent Selection

Each time a pointer is transmitted to another space, the local counter is incremented. If the receiving space already knew this pointer, it can now choose between its two fathers (the original and the new). This decision is made according to their depth in the tree. This idea needs that each node knows his depth in the tree. After switching from father, a DEC is sent to the other father and the local depth is adjusted.

#### Near-leaf Parent Avoidance

An inconvenience happens if a node changes its father by the Parent Selection-algorithm and if the new father was a child of the diffusion tree before. Although completely correct, this causes the diffusion tree to have one leaf less. With respect to failures and fault tolerance, the number of crash-problematic nodes has increased.
To solve this problem, the process that sends a remote pointer, adds a bit telling whether it is a leaf in the tree or not. Depending on this bit, the receiving process can determine whether it will consider the new father as a father or not.

**Simple Weighted Proxies**

In two similar situations, DEC-messages can be eliminated or at least postponed by using a system similar to weighted reference counting.
When a process receives a remote pointer for a second time, it sends a DEC to one of the two fathers (see algorithms above). If the pointer was sent by the same process being its current father, it will not send the DEC immediately, but keep a counter of DECs to that process. In case of a father change or deletion of the proxy, it will send a DEC representing multiple DECs.
The same idea can be used when a process sends multiple RPC-calls to the root and all of them successfully reroot the node. This causes multiple increments at the root and will cause multiple DEC-messages to be sent by the original process. Here they can be combined too by a similar system.

### 4.3.3 Remarks

The extra bit needed in the RPC-call and in the proxy, is used by the proxy to avoid asking for a second rerooting when a previous rerooting failed for some reason. It can be a design-decision or an implementation-decision to re-request rerootings each time, or only the first time.
The sub-tree rerooting uses an RPC-call to transmit the request for rerooting, but it can cause an extra DEC-message if rerooting is done. This extra message could otherwise have occurred in the far future or even never at all.
Because now each proxy contains a depth-indicator, this field can suffer from overflow-problems. To keep the space occupied by this field small, it can be a small integer. The highest value will represent a sticky infinite: all nodes below have the same ”highest” value.

### 4.3.4 Critical View

When a tree is constructed, each node knows its depth in the tree. Unfortunately when time elapses different tree flattening techniques are used to lower the depth of the tree. Each time this action happens, not only the depth of a node changes, but also the depth of all its children. These children are not informed of the depth-change because it would cause too many messages. After some time, and many flattenings, the actual depths of nodes and the depths at which they think they are can differ a lot, causing the second algorithm to behave reversely.
Unlike Moreau’s algorithm, these algorithms are only thought of, no actual implementation exists. When performing different tree flattening algorithms at the same time, one must also consider the problems which are caused by applying them. For instance a DEC can be sent to the old father, while another algorithm is changing the father. Or two algorithms can create a race condition to change the father, causing DECs to be sent to wrong fathers.

## 4.4 Conclusion

Because both network-configuration and the actual distributed computation causes different impacts on the final result of the flattening technique, this is difficult to test. For each different setup, a number of benchmarks can be generated which shows that a certain algorithm is better than another

# Chapter 5

# Proposed Tree Flattening Techniques

As stated in chapter 4, the problem is that when using Indirect Reference Listing in its normal way, the diffusion-tree may become very deep. This can cause different problems:

- zombie-references
- more difficult to manage fault-tolerance

In any case, minimizing the depth of this inverted tree is recommended. At least if the overhead to minimize it, remains within acceptable boundaries.
This chapter will propose some new techniques to minimize the depth of this diffusion tree.

## 5.1 Introduction

As described in chapter 2, there are four different operations which may alter the diffusion-tree:

- creation of a new node (sending a reference from the root to another node)
- duplication of a node (same as creation, but not invoked by the root)
- deletion of a new node
- migration of the real object

The creation and duplication of nodes are handled in a similar way. This will be described in the next section.
Deletion of nodes and migration can cause different kind of actions on the diffusion tree. Each will be explained in a different section.

## 5.2 Adding or duplicating nodes

A new node in the tree can be created in three different ways. The first is by invoking a remote method and sending an object as parameter. Depending on the object, it is sent as a copy, or as a remote reference. The second way is by invoking a remote method that returns a result. This result can be a remote reference. The last way is by calling a lookup on the registry. This will always return a remote reference.

Either way, a remote reference is transferred from one space to another. If this remote reference was not know in the second space, this new space is added to the diffusion tree.

In normal Indirect Reference Listing, the new node is added as a child of the node where it came from. There is no difference whether this was the root or not.

In IRL the new node is added as a child of the node where it came from, but also another mechanism is invoked to minimize the depth of the diffusion-tree. This new mechanism will be called ”Minimize-by-Insertion”, because it is only invoked when a new node is added to (or inserted into) the diffusion-tree.

### 5.2.1 Minimize-by-insertion

The goal of this algorithm is to keep the depth of the diffusion-tree within a limit (unlike in the default algorithm of IRL). Most of the time this limit will be four levels, but for short times, five levels are also allowed. Each time the depth of five is reached, a tree flattening technique is invoked.

New nodes can be added to the tree at four different levels in the tree. Depending on the level, the new node will create a number of additional messages to minimize the tree.

In all cases (= all levels), it will be at the original node (= future parent) where the action starts. The first thing to do is to add the new node to its list of children. The new node will add the first node as its father. This behavior is the standard behavior in IRL-algorithms, and will be the first thing both nodes do at each level.

The father will always add the new node as its child, although it does not know whether this new node was already in the diffusion-tree somewhere or not. If this is the case, the new node will send a decrement message to its second father. This will delete the new node from the reference list of the father and make the tree consistent again.

#### Level 1

This case is where the root is the father of the new node. It is also called ”creation of a new node”, conforming to the list of chapter 2.

Because it is the root which creates the new node, no extra messages will be needed and the new node stays at this level.

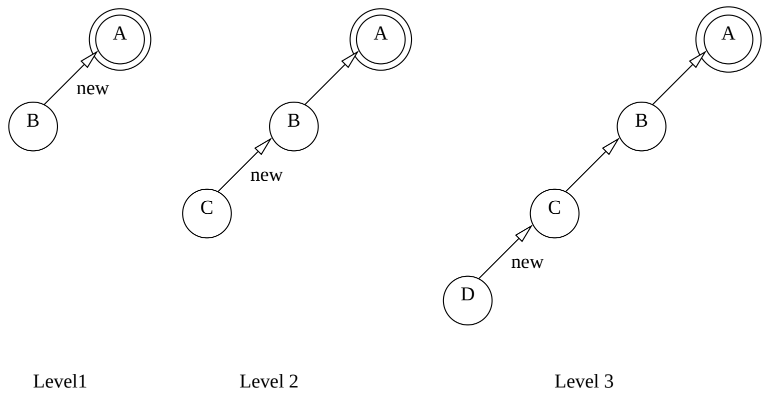

Figure 5.1: Levels 1,2 and 3

**Level 2**

Now the creator of the new node is not the root, but it is a direct child of the root. it knows this, because its father is the root.
In this case, this information is transmitted to the new node, which knows it is at level 3. No extra messages are generated. From level 2 to all next levels, this is also called ”duplication of a new node”.

**Level 3**

Again, the creator is not the root, and it knows that its father is neither. In this case, we need more information to decide that we are on level 3 and not deeper. Therefore extra information is added to a wire-representation: a grandfather. If its grandfather is the root, it means that it is on level 3. Only from level 3, the grandfather makes sense, on higher levels the nodes do not have grandfathers. The information in the wire-representations is updated each time they are transmitted to another node.

Important is that when a wire-representation is sent from a level 3 node, it contains an extra bit to inform the new node that it is on level 4. The same bit will be used at the next level to inform the new node that it is at level 5. The grandfather-information only is insufficient to determine the difference between levels 4 and 5.
Now we know that we are on level 3, no extra messages are generated and the diffusion tree remains the same.

**Level 4**

When a wire-representation leaves a level 4 node, it knows that its grandfather is not the root. This information is represented in the same bit as used in level 3 (but a different value).

When the wire-representation arrives at the new node and it was not already in the tree (this would generate a decrement message back), a tree flattening algorithm is started.

The algorithm consists of a combination of three extra messages, but these can be sent asynchronously with the computation. These extra messages will not create a large overhead because they can be sent in parallel with the computation.

- the new node sends a message to the root to inform it that it has to adopt the father of the new node
- The root will send a message to the father to say that it is its father from now on. When the old father agrees, the root will add the father to his reference list.
- The father will send a cleancall to its original father to keep the tree consistent.

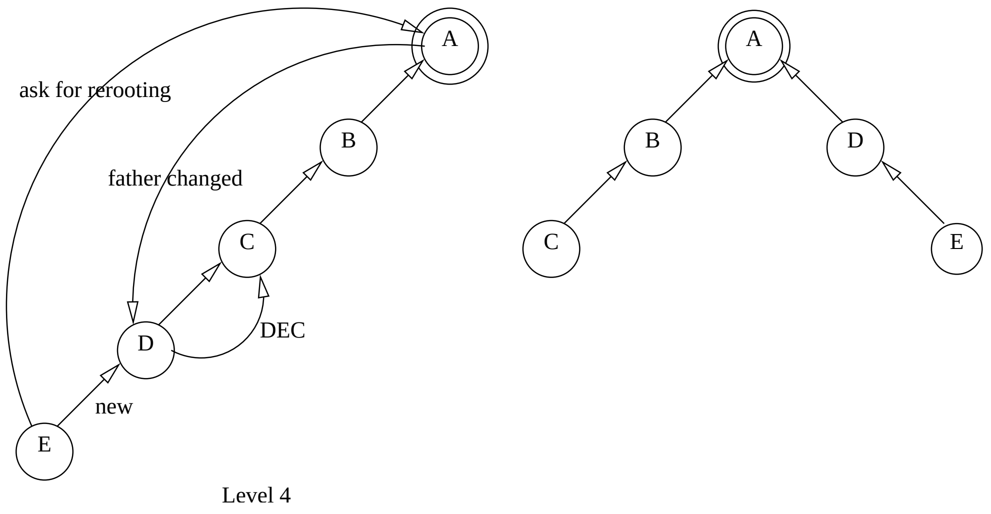

Figure 5.2: Level 4

Thus, the overhead on level 4 equals the sending of 3 messages, but none of them includes a big amount of processing. It is just to keep the tree consistent.

## 5.2.2 Arguments

The idea behind this flattening algorithm is that the depth of the tree can not exceed 4, unless in specific cases. Each time a depth of 4 is reached and a new node is added (creating a fifth level), the tree is flattened.
As in most algorithms, specific case are discovered where the tree can be deeper than 4, but these are only temporal stages.

#### New node on Level 4

When a new node is created on level 4, it will be added as child of this level. This way it is temporary on level 5, but it will we moved up when the message receive the root. This flattening will move the node back to level 3.

#### New node of a new node on Level 4

Because the propagation of these messages that flatten the tree are propagated asynchronous with the actual calculations in the new node, it can happen that this node creates new nodes below himself before it has been rerooted to another (higher) level. The first new node was already on level 5 (but will be moved soon), but the last new node is even on level 6!!
Fortunately, this is also just a matter of time. Because the new node on level 6 also starts a flattening algorithm and will we moved to a higher position in the tree too. The same happens when this last node creates another new node and so on...
after some time, the tree will always have a depth of 4 (or less).

### 5.2.3 New invariant: Grandfather

The correctness of this flattening algorithm now also relies on a new invariant of the diffusion tree: Each node knows its grandfather.
When only creating new nodes, this can be done in a very efficient way. Each time a wire-representation is sent to a new node, the father and grandfather fields are updated. The new node now directly knows its father and grandfather.
Deleting nodes creates more problems, because this can change the grandfather of another node.

#### Level 1 and 4

Both levels do not cause any problems. The first because it can only be deleted if there are no children, meaning no remote references (the tree is completely collapsed). Level 4 normally does not have children, so it is grandfather of noone. In the very exceptional case that a level 6 node is created, this will be only a temporal stage, whereby its grandfather will be changed when its flattening message arrives at the root.

#### Level 2

A level 2 node can have grandchildren, which will have a new grandfather (the root) if this node is deleted. Here two different approaches can be taken. The first is sending a message to each of the grandchildren (through the list of children) to say that they have a new grandfather.
Because this can create a large amount of messages, another possibility is more useful: do not inform them, but let them still think that they are 1 level too deep. This has no catastrophic effect on the diffusion tree, nor on the correct behavior of the algorithm. Although it can create extra (unnecessary, but still correct) tree flattening actions because some nodes think they are deeper than they actually

are. After such a flattening action, all the information (father, grandfather...) in that node is back up to date.

### Level 3

Although a level 3 node normally does not have grandchildren, the grandfather of its children changes. Independent of the algorithm used to deal with deletion of intermediate nodes, no new messages are generated:
If the children of a deleted node are adopted by the father of that node, this new father sends its name and its grandfather to the adopted children. If they are adopted by the root, the grandfather field is erased by adoption.
In neither case, messages are sent.

## 5.3 Deleting nodes

A side-effect of using indirect reference listing is that a node which is no longer locally referenced, may not be deleted if it still has children. The reason for this is that this node protects the root from being freed, while there are still remote references from nodes, below this node.
Example: If the tree contains 4 nodes: 2 children, 1 father and 1 grandfather (root); and the father is no longer locally referenced. An incorrect implementation would delete the father and send a clean-call to its father (root). If the root is no longer referenced, it may be deleted, because he cannot know that anyone else is still referring to him. This wrong behavior should be avoided at any time.
Most algorithms solve this problem by simply not deleting nodes if they are no more locally referenced, but still have children. These kind of nodes are called zombie-references. Once they have received all clean-calls from all their children, they may be deleted. The problem is that this algorithm can create a chain of zombie-references with at the end one node. When this final node is no more used, it will create a lot of clean-calls to clean the chain.

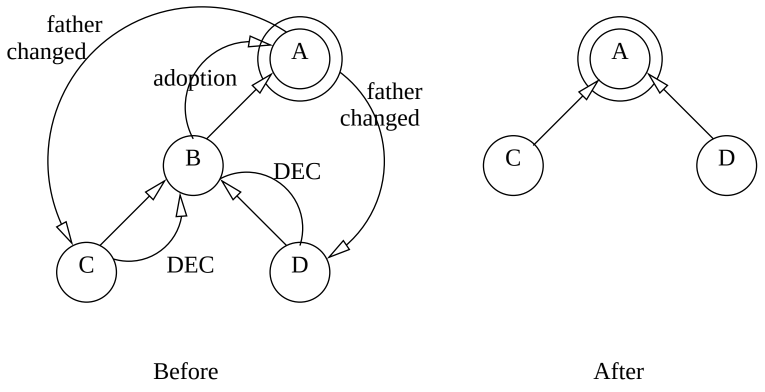

Figure 5.3: Zombie Deletion

The solution proposed in this thesis is to delete these zombie-references, but still prevent the root from freeing itself incorrectly.

Before the node will be deleted, all its children will be adopted by the root. Now all the children are moved in the tree to be children of the root, thereby also flattening the tree. Once the node has no more children, it can be freed without any problem (no more references to it locally nor remotely). A chain of zombie-references cannot be formed because each zombie-reference is immediately removed from the tree.

## 5.4 Migration

Migration is the act of moving the real object from its current node to another node in the tree. This is always a very costly operation (compared to the other operations), because the object has to be moved and the tree has to be reorganized such that it remains consistent.
In classical Indirect Reference Listing (and Counting) this means that once the object is moved, the new node becomes the root of the diffusion-tree. The original root is adopted as child of of the new root.

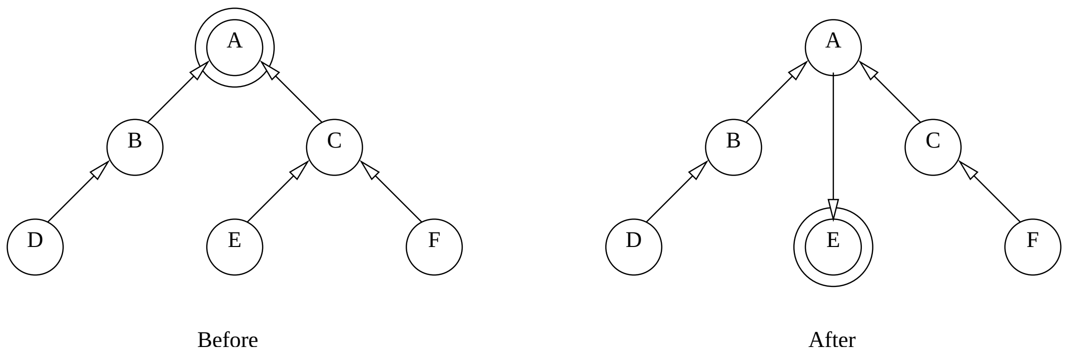

Figure 5.4: Migration

After this movement, all nodes must be informed that the object has changed to another node in the tree, such that they can send remote method calls to the right object.

### 5.4.1 Tree is growing

Unfortunately, this good algorithm does not cooperate very well with the tree flattening approach proposed in this thesis or any other. Each time the object moves to another node, the depth of the tree can be incremented by 1. After 10 migrations, the tree can have a theoretical depth of 14, violating the invariant maximum depth of 4.

### 5.4.2 Solution

Until now all IRL algorithms relied on the fact that the object is located at the root of the diffusion-tree. This prevents the real object to be garbage collected prematurely.
This thesis proposes a radical change to this approach, by not holding the real object in the root, but letting it move around in the tree.
The algorithm works very similar to the original one, but just keeps the tree as it was before (no tree rerooting). This change imposes some new algorithms to prevent the object to be garbage collected premature and to actually reclaim it when it is no longer referenced.

#### Object at the root

When the object stays at the root, the algorithm behaves exactly the same as before. The object can only be reclaimed if no local or remote references are known to the root (= if it is not locally referenced and it has no children).

#### Protection

When the object has moved to another node, the node containing the object must be protected from being garbage collected. If this node has no more local references and it has no children, it could be reclaimed (according to the original algorithm)
But because this node knows that it contains the real object locally, it will prevent itself from being garbage collected.

#### Termination

When the root does not contain the object any more, the root will always have at least one child because the node where the real object is located, can never be reclaimed. This prevents the node with the object to be garbage collected, but it also prevents the root from being freed. If this is not handled, a small tree, containing only the root and the node with the real object, will always remain alive.
Fortunately, this problem can be solved. If the node containing the real object is no longer referenced locally or remotely, it can migrate back to the root. After this migration the last node can be collected and the root too, since we are in the normal IRL-algorithm now (the root contains the object).

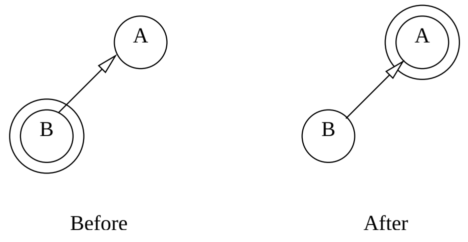

Figure 5.5: Termination

**Improved Termination algorithm**

The algorithm above always causes an object migration, even if there are still other nodes in the tree (other than the root and the node containing the real object). This operation is costly, so avoiding it is preferred.
Avoiding it can be done, by a simple ping-pong-algorithm invoked by the root: When the root is no longer locally referenced and detects that its only child is the node with the real object, it send a message to this node. If this node has no children and is not locally referenced, both nodes can be cleared immediately, because there exist no references to them anywhere in the tree. If it is still referenced (locally or remote), both nodes are kept alive. When this node detects that it is no longer referenced (locally or remote) after a while, it will contact the root. Because the root can have new children by now, it checks again. This ping-pong-protocol keeps going until both nodes are not referenced locally nor remotely (except the root by the second node). Now both nodes can safely be freed.

## 5.5 Conclusion

This chapter described in detail the two new algorithms proposed in this thesis. First Minimize-by-Insertion is explained: each time a new node is added deep in the diffusion-tree (fifth level), its parent will be rerooted and reshape the tree to a maximum of four nodes deep. The second algorithm works on another basis: Zombie Deletion deletes a node as soon as it becomes a zombie (no longer locally referenced but still has children). By deleting these zombies, the tree will be less vulnerable for node-failures and the depth can be reduced.
Orthogonal on these tree flattening techniques another way of managing migrations is proposed. Originally the tree is adapted to make the new space where the object is located, the root. This is bad in combination with tree flattening algorithms because it can increase the depth of the tree. The algorithm proposed here does not change the tree making it easy compatible with tree flattening techniques.
The next chapter will give several different views why new flattening techniques are proposed and whether there is a difference in their execution.

# Chapter 6

# Validation

## 6.1 Introduction

The new optimizations proposed in this thesis and the tree flattening techniques of Moreau are validated in two different ways. First a simulation model is made which simulates the grow of the diffusion-tree in combination with different tree flattening-techniques. The second validation is a proof-by-implementation. An Object-Oriented prototyped language is extended with remote references and a DGC. The same techniques to implement the DGC for the prototype based language were used to construct a java RMI-clone. On top of this RMI-clone different benchmarks are constructed to see the effect of the different tree-flattening algorithms.

## 6.2 Simulation Model

This simulation model is implemented in java and simulates the diffusion tree of a single object that is remotely referenced. All standard operations that can occur on a node are supported by the model:

- create new node as child of this node (creation or duplication)
- delete a node (deletion)
- migrate the object from one node to another (migration)

But also the new operations defined by the new dgc-techniques are supported:

- reroot a node to become a child of root (Moreau's algorithm)
- maintain a maximum depth (Minimize-by-Insertion)

### 6.2.1 Adding a new node

As explained before, adding a new node to the tree can generate a number of messages depending where in the tree it is added and also depending on the flattening-algorithm used.

This model can be used to see the difference in the number of generated messages and the difference in the final diffusion-tree by using different flattening-algorithms. Three different techniques can be chosen:

- Original Indirect Reference Listing
- Luc Moreau's rerooting
- Minimize-by-Insertion

At a small number of nodes in the tree, the result in the diffusion tree is negligible. Only when the tree contains a few hundreds to a thousand nodes, the differences can be clearly seen in figures 6.1, 6.2 and 6.3.

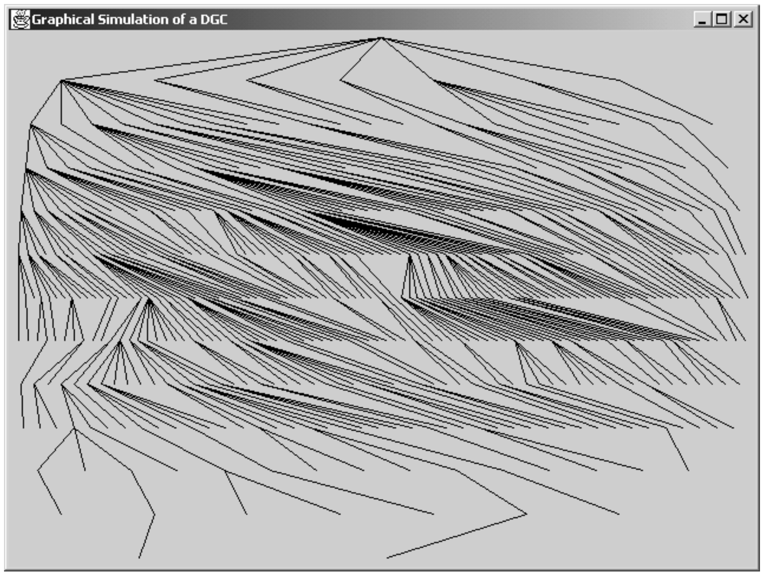

Figure 6.1: 500 nodes without flattening

When nodes are added in a randomly fashion (as in all three examples), normal IRL will create awful diffusion trees. When Luc Moreau's algorithm is applied, the diffusion tree is reshaped in a very drastic way, whereby all nodes become a child of the root. This has several implications:

- it is expensive because two messages are sent each creation of a new node
- it is better for fault tolerance, because no nodes (besides the root) contain children.
- it has less parallelism because all nodes have to communicate with the root (e.a deletion, creation), because it is their father.

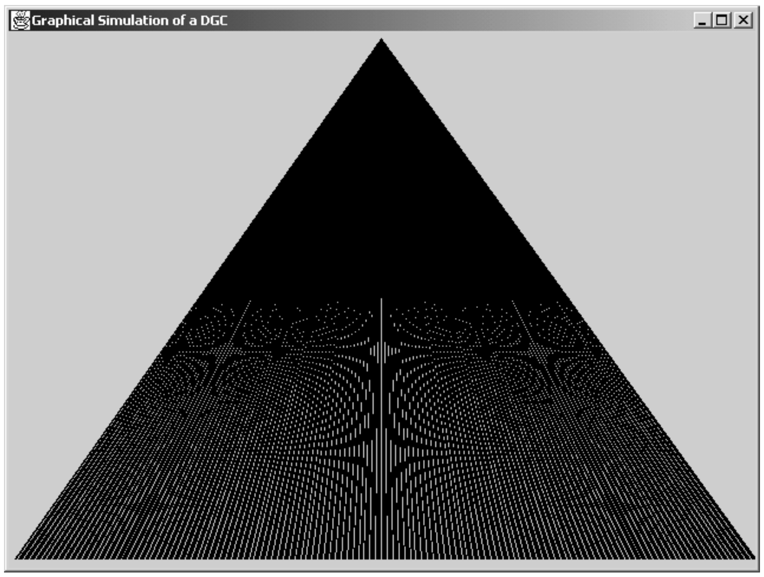

Figure 6.2: 500 nodes with flattening by Moreau

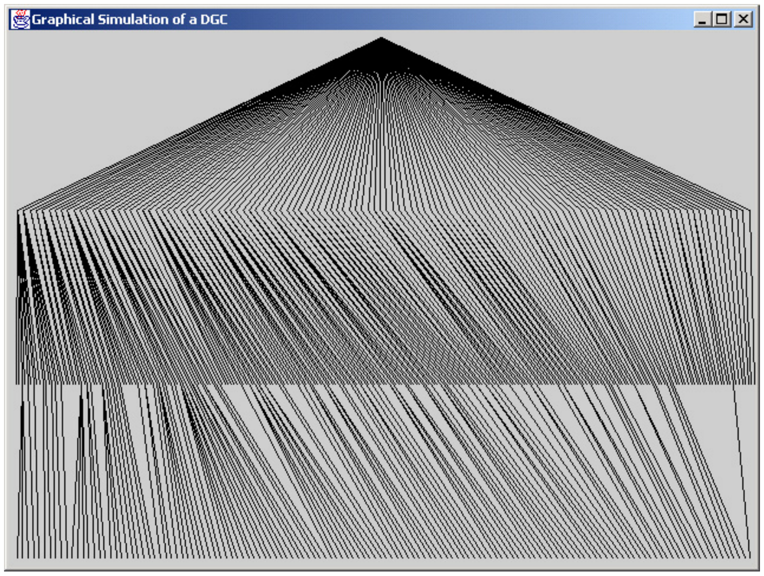

Figure 6.3: 500 nodes with Flattening-by-Insertion

### 6.2.2 Migration

Migration, as performed in normal IRL, can create undesired effects on the diffusion tree. Normally the node where the object is migrating to becomes the new root and the old root becomes a direct child of the new root. This is a very safe way of migrating an object, but in most cases it increases the depth of the tree. This is especially bad in our case, because we apply different algorithms to minimize the depth of the diffusion tree. Most tree flattening techniques do not work well with migrations for this reason, although they can be used when applying migrations as proposed in this thesis.
When the old migration is performed in combination with one of the two tree flattening algorithms, the result is negligible. All the effort to compact the tree is lost by the migration actions.
The application of the new migration algorithm is therefore vital for the effectiveness of **all** tree flattening algorithms. After a migration, the tree remains the same as before. Both flattening algorithms have implicit requirements that the tree is not extended without their knowledge.
Moreau's algorithm places all nodes as direct childs of the root. After a migration, this invariant is no longer satisfied.
Flattening-by-Insertion keeps the tree less compact, but if nodes are added to deeply in the tree, a subtree is rerooted to shrink the tree. The old migration breaks this invariant, by possibly extending the depth of the tree each time the object is migrated.
Flattening-by-Insertion can cope with this invariant-breaker a little better than Moreau's algorithm, because if new nodes are added at a very deep level, their fathers will be rerooted. If these events occur at the right places, the tree will shrink to a normal, acceptable depth. Moreau's algorithm only reroots new nodes, leaving everything else untouched.

### 6.2.3 Statistics

Together with the creation of the visual model, a lot of statistics are gathered. For different setups, the differences in the tree and the number of messages send can be measured.
These different statistics can be used to see the costs of applying tree flattening techniques, but they also measure parameters of the diffusion tree. When different tree flattening techniques are applied to the same diffusion tree (construction), we can compare both cost and benefits of the algorithms.

#### Available statistics

To make a decent comparison, it must be assured that both algorithms are applied on exactly the same diffusion tree. This is accomplished by initiating the Random Number Generator to a certain seed before starting the simulations.
The variety of different statistics gathered about a flattening technique defines the accuracy of the test.

- number of nodes created
- number of nodes deleted
- number of final nodes

- number of clean calls send
- number of flattening messages send
- number of actual flattenings performed
- number of migrations performed
- number of adoptions performed
- number of children of the root
- average depth of the diffusion tree
- highest depth of the diffusion tree

**Comparing tree flattening techniques**

Both tree flattening algorithms are tested in a number of different configurations. Important is how many nodes are added to the tree. By increasing this parameter, the effect on the number of generated flattening messages and actual flattenings can be measured.
Both algorithms are tested with only creation of nodes (no deletions nor migrations):

| nodes | generated messages | actual flattenings | % messages | % flattenings |
|---|---|---|---|---|
| 100 | 188 | 94 | 188 | 94 |
| 200 | 388 | 194 | 194 | 97 |
| 300 | 588 | 294 | 196 | 98 |
| 400 | 788 | 394 | 196 | 98 |
| 500 | 984 | 492 | 197 | 98 |
| 600 | 1188 | 594 | 198 | 99 |
| 700 | 1386 | 693 | 198 | 99 |
| 800 | 1586 | 793 | 198 | 99 |
| 900 | 1782 | 891 | 198 | 99 |
| 1000 | 1986 | 993 | 199 | 99 |

Table 6.1: Moreau’s flattening

The statistics show that for each new node that is not a direct child of root , exactly two new flattening messages are created in Moreau’s algorithm. This results in an average of nearly two messages per new node.
With Flattening-by-Insertion, the number of generated messages depends on where that node is inserted on the tree. Only if it is at the forth level, three messages are created, otherwise none. When nodes are added in a randomly fashion, this creates an average of 0.75 extra messages for each newly added node. This is substantially less that the 2 extra messages noted at Moreau’s algorithm. Moreau’s

| nodes | generated messages | actual flattenings | % messages | % flattenings |
|---|---|---|---|---|
| 100 | 65 | 21 | 65 | 21 |
| 200 | 138 | 46 | 69 | 23 |
| 300 | 209 | 69 | 70 | 23 |
| 400 | 272 | 90 | 68 | 23 |
| 500 | 353 | 117 | 71 | 23 |
| 600 | 432 | 144 | 72 | 24 |
| 700 | 505 | 168 | 72 | 24 |
| 800 | 581 | 193 | 73 | 24 |
| 900 | 647 | 215 | 72 | 24 |
| 1000 | 741 | 247 | 74 | 25 |

Table 6.2: Flattening-by-Insertion

algorithm enforces almost each time a rerooting of the new node, while Flattening-by-Insertion does this only once every four new nodes (in average).

## 6.3 Implementation

Statistics are beautiful to look at, but are only useful if they are relevant. The problem of this simulations is that they only simulate the model that is programmed. Each simulation is carefully designed and does always the same.
Real life programs are usually not that clean. According to the problem they are solving, different diffusion trees are constructed, each time looking different.
Nevertheless how beautiful some statistics may be, some persons will only accept new ideas and theories after they are proved to work correct. This proof can be theoretical, like a mathematical proof of correctness. The other possibility, used here, is a proof-by-implementation.

### 6.3.1 PicObj

Pico is a tiny, academic programming language developed by Theo D'Hondt at the VUB (Vrije Universiteit Brussel). It's primary goal was to teach programming concepts to persons that do not know anything of programming. Most other languages are too complicated for this purpose, bothering the programmer with a lot of syntax and difficult semantics. Pico is kept as simple as possible (it's grammar can fit on one page) to let the programmer think about what he is doing instead of thinking how he should write it syntactically correct.
Pic% (read as a double o), is a Object-Oriented extension of the original pico. Instead of a class-based language (like almost all current OO languages), this Pic% is prototype-based. Objects are represented by dictionaries: mapping of names to other constructs (number, string, dictionary, ...).
PicObj is a distributed extension to Pic% which allows the programmer to export dictionaries to a registry and to look other dictionaries up. Next, methods can be called in these dictionaries. These

methods will not be executed locally, but in the machine where the dictionary is physically located. For the programmer, this is almost completely transparent; in most cases the programmer will not see the difference. Only if native functions are called with a remote dictionary as parameter, this will generate an error, because it expects a local dictionary.

#### Remote Dictionaries

The most important difference between PicObj and the normal Pic% is the possibility of accessing dictionaries on remote machines. Internally this reference is a sort of generic stub. Because Pico is dynamically typed, there are no typing-problems. This generic stub locally acts as a normal dictionary. As with all dictionaries you can ask values of variables or call methods. The local stub knows where the actual dictionary is located and opens a connection to that machine. Then it will marshall all the parameters and send them to the remote machine. There the lookup and/of calculation is done and the result is marshalled and sent back to the original machine.
Marshaling of parameters and return types are done in a very simple way. If the value is a dictionary, it will be transformed into a remote reference. All other values are just copied. Because PicObj is implemented in java, the object marshalling from java is used for all objects except dictionaries. If a remote reference is marshalled, it is always copied. Of course when remote references or dictionaries are sent to or received by other spaces, their reference lists are updated and all normal DGC-activities are triggered.

### 6.3.2 Java RMI-clone

Although the PicObj-implementation is very clean, not many programmers will actually use this. Nowadays, java is standard accompanied with a remote communication package called RMI (Remote Method Invocation). The implementation of standard RMI[Cor] uses a DGC that is based on Birrel's Network Objects[BEN$^+$93]. A reference set is kept in the space where the actual object resides. Each time a remote reference is sent to another space, this space has to communicate and synchronize with the object space. This synchronization can be quite costly.
Unfortunately, the standard RMI is not adaptable in any way. The DGC-algorithm behind RMI cannot be changed or optimized. Therefore a RMI-clone is proposed in this thesis. Exactly the same functionality as RMI is offered, but now with a different DGC-algorithm behind it.

### 6.3.3 Original Java RMI

There exist many papers and websites on how to use RMI, so this section will not be a tutorial, it will only point out the different constucts needed and how they interact with a user-program. To explain this as easy as possible, we will use a small example of a calender-server, where a client uses RMI to ask the current date or time to the server. The sourcecode can be found in Appendix A.

- RMICalenderServerInterface: The methods that can be called on the server must be declared in a seperate interface that inherits from Remote.

- Each method must throw RemoteException
- RMICalenderServer must implement the interface and must extend from UnicastRemoteObject
- The server must be bound in a registry by using: Naming.bind
- The client must lookup the server by using: Naming.lookup
- The server received on the client is casted to it's interface
- After all sourcecode is compiled, the stub and seleton must be generated by calling ”rmic RMICalenderServer”. This creates directly the compiled versions of both stub and skeleton.

### 6.3.4 RMI-clone

The RMI-clone descibed in this paper, only changes some of the requirements mentioned above. Only the changes will be explained:

- RMICalenderServer also implements the same interface, but now extends JavaRemoteObject
- The server is bound in the registry by using: Manager.getLocalRegistry().rebind
- The client looks up the server by using: Manager.getLocalRegistry().lookup
- To create the stubs, the following program has to be run: ”java dgc.Generate RMICalenderServer”. This create the sourcefiles for stub and skeleton. Afterwards these must be compiled.

This means that all current programs written for java RMI can be easily adapted to use this RMI-clone. The changed files can be viewed in Appendix B. The automatic generated stub and skeleton are available in Appendix C.

## 6.4 Benchmarks

As stated in chapter 4, the aim of tree flattening techniques is to make the diffusion tree more compact, and thereby less vulnerable to failures of nodes. Because these flattening algorithms create a number of extra messages on top of the messages needed for computation and normal distributed garbage collection, this extra cost must be kept into reasonable boundaries.
Usually benchmarks are used to statistically ”proof” that a new algorithm works faster than the previous one. Execution times of the new algorithm are then compared with those of the old one. In this case, the same execution times are measured, but the comparison is slightly different.
The primary goal of tree flattening algorithms is not to speed up the computation, nor the distributed garbage collection, but to compact the diffusion tree. Therefore, the benchmarks are used to measure how much extra time is spent on these extra messages. If this time remains within acceptable boundaries, these algorithms can be used in specific cases where fault tolerance is important.

In some cases, tree flattening algorithms will shorten the overall time to collect all the garbage. The reason for this is in general that computations can be done in parallel, while before they would have been blocking each other. Although this is not the primary goal, it is still nice.

## 6.4.1 Benchmark Programs

For many languages, even parallel languages, there exist different many different benchmarks, but until now, no benchmark is especially targeted to the distributed garbage collector. Hopefully, these will become available in the near future, to make an objective comparison between different DGCs.
Luc Moreau proposed two benchmarks, which are especially targeted to his tree flattening algorithm and to another improvement of IRC (accumulating DGC messages and sending them in one time)[Mor01]. Both benchmarks are rewritten into java to be used with the RMI-clone described in this thesis.
Additionally, two other programs are added to the list of benchmarks. The first is a distributed bubblesort, which creates a large number of remote reference sendings. The second is a distributed implementation of the shortest path calculation. These two benchmarks are more real-life programs to check the effect of flattening algorithms on real programs.

**Benchmark 1 : Cycle**

The Cycle benchmark first creates a cycle of master-objects. Each master knows its next object, thereby forming a cycle. The last master starts the benchmark by creating a remote accessible object (CycleObject) and sending the reference to it to its next master. Each master will receive the object and send it to its next until the reference almost made the full circle. The last master (before the master where the object is really located) will create a new CycleObject that references the original CycleObject. This way the original can not be recycled before the new one has been collected. After preforming some cycles (which is a parameter), the last object is not used anymore and will cause the breakdown of the diffusion trees and recycling of the other CycleObjects.

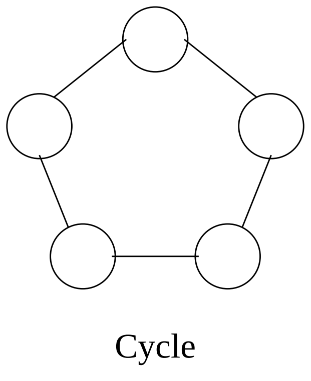

Figure 6.4: Benchmark 1 : Cycle

**Benchmark 2 : Diffuse**

Each master in the diffuse benchmark knows all other master and can send messages to them. Execution starts in the last master who will create a new remote accessible object. He will send this object to a number(this is the first parameter) of other masters. Masters that receive an object will send it to the same number of other masters. These masters are chosen randomly from the list of all masters. The number of levels these objects are resend to other masters is a second parameter of the benchmark.
The idea behind this benchmark is that many masters will receive the same object many times (generating clean calls) and the diffusion tree is not deep or mal-formed (in general).

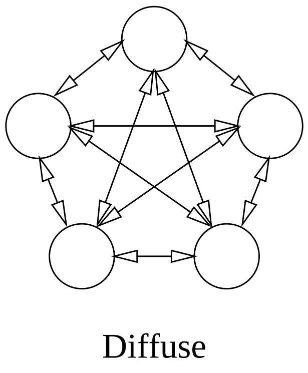

Figure 6.5: Benchmark 2 : Diffuse

**Benchmark 3 : Bubble**

Bubble is a distributed version of the bubblesort. Each master contains a remote accessible object, which represents an integer-value. All masters know their previous and next master and will communicate with them alternatively. Each pass will not move the object, but may swap the different remote pointers in the masters. The benchmark ends when all objects are sorted, displayed and collected.
To create long chains in the diffusion trees, the original objects are not chosen randomly. Each master at index **i** has an object with value **nrofmasters-i**. This way, the objects at the edge of the bubblesort must pass all other masters to get to their final position.

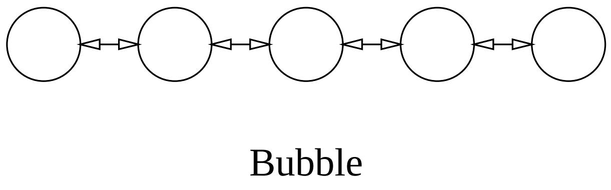

Figure 6.6: Benchmark 3 : Bubble

**Benchmark 4 : Shortest Path**

Here a configuration file is used to define the edges, starting vertex, end vertex and weights of the edges. Enough masters are created, such that each vertex is located at one master and each master knows his neighbours in the graph6.7. Computation starts in the start vertex, where it propagates the current path-length to all his neighbours. These masters will check the path-length they receive with the globally shortest path. This is an object, which is accessible from every master and contains the shortest path length currently found. Each time the path reaches the end vertex and the path length is shorter than the current path length, it will update the global value.
This benchmark was added, not because of some special characteristic or to stress a certain tree flattening algorithm, but to see the impact of these algorithms on a more common distributed application.

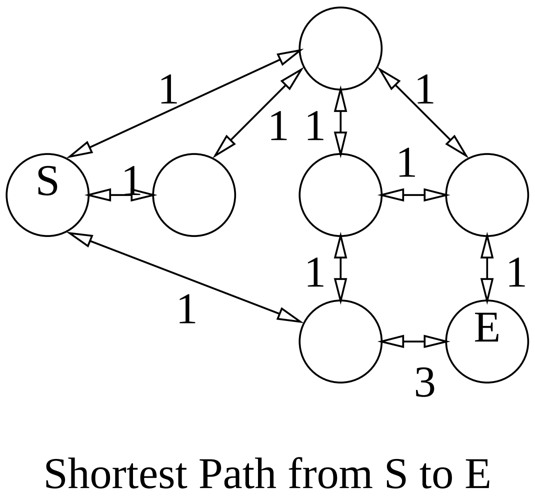

Figure 6.7: Benchmark 4 : Shortest Path from Start to End

### 6.4.2 Uni-processor Results

In this section we will show the results of the different benchmarks, combined with the different tree flattening algorithms. All spaces (processes) are physically located on the same machine. All the tests in this section were performed on a Pentium II 333Mhz with 192Mb ram , running a Windows 2000 operating system.

**Benchmark 1 : Cycle**

The first benchmark is a Cycle with 10 master-objects in the cycle. This cycle will be cycled 5 times.

| | Normal | Moreau | Insertion | Zombie |
|---|---|---|---|---|
| startup | 14 | 22 | 14 | 17 |
| complete | 37 | 120 | 68 | 150 |
| cleanup | 100 | 130 | 95 | 177 |
| calls | 54 | 54 | 54 | 54 |
| registry | 20 | 20 | 20 | 20 |
| clean | 46 | 54 | 61 | 97 |
| extra | - | 114 | 30 | 91 |

- startup : number of seconds from start until all masters are initialized
- complete : number of second from start until computation has finished
- cleanup : number of seconds from start until all objects are collected
- calls : number of remote method calls (computation)
- registry : number of remote method calls (to the registry)
- clean : number of clean calls (DGC)
- extra : number of extra messages (tree flattening algorithms)

**Explanation:**

First important issue is that in all cases, tree flattening algorithms will postpone the completion of the computation. This is explained by the fact that all of them generate a number of extra messages which will consume processing time.
The normal IRL-algorithm needs a long time to clean up his objects, because it created deep and unbalanced diffusion trees. All clean calls are triggered by the previous clean call, therefore taking a long time to cleanup the first object.
All three tree flattening algorithms have as a side-effect a faster cleanup at the end (up to 84% faster). On the other hand they need extra messages to break down these diffusion trees. This breakdown happens during the actual computation, and thus slows down the computation. Both in the cases of Moreau and Zombie-Deletion, this slowdown is higher than the final gain while cleaning up. Therefore their final cleanup-time is higher than in normal IRL. At the Insertion-algorithm the gain is higher, resulting in a faster cleanup-time.
The reason for the slow performance of both Moreau and Zombie deletion on a uniprocessor can be found in the number of messages sent. Both the number of calls and registry stay the same for all algorithms, but especially the number of extra messages is algorithm-dependent. Moreau will generate for almost each new node in the diffusion tree two new messages, resulting in a final count of 114 extra messages. The same happens with Zombie deletion, because all intermediate nodes in the diffusion tree (except the root and the last node) are no longer locally referenced. Each of them generates two extra messages plus an extra clean call.

Insertion however generates a far lower amount of extra messages : only in 15 times two extra messages and an extra clean call are generated. In this case the time spent on executing these messages is less than the gain in the cleanup, making the algorithm faster than IRL.

**Benchmark 2 : Diffuse**

This benchmark will be constructed with five masters and each master will send an object to three other masters. The maximum level of indirections will be three.

| | Normal | Moreau | Insertion | Zombie |
|---|---|---|---|---|
| startup | 8 | 12 | 7 | 7 |
| cleanup | 25 | 31 | 24 | 28 |
| calls | 37 | 37 | 37 | 37 |
| registry | 10 | 10 | 10 | 10 |
| clean | 38 | 38 | 38 | 41 |
| extra | - | 38 | - | 6 |

**Explanation:**

The line ”complete” is missing in this table because it does not make sense in this benchmark. The difference between the ending of the computation and the ending of the cleaning is minimal and is hard to compute.
The most important remark is that Insertion is never triggered in this benchmark. therefore the result is almost identical as Normal IRL. The difference can be statistically explained.
Similar to the previous benchmark, Moreau and Zombie-Deletion create a number of extra messages (38 and 6), but the gain is minimal in this benchmark. Therefore they produce only overhead and slowdown the termination of the benchmark.
Minimize-by-Insertion is not triggered because the computation itself created a diffusion tree which is not too bad. As long as the diffusion tree remains within some boundaries, Minimize-by-Insertion will leave the tree as it is, because it is not worth optimizing the tree (the profit is smaller than the cost).

**Benchmark 3 : Bubble**

This benchmark will be constructed with ten masters, where each master **i** will have an object in his space, representing the value textbfnrofmasters-i.

| | Normal | Moreau | Insertion | Zombie |
|---|---|---|---|---|
| startup | 15 | 30 | 15 | 25 |
| complete | 250 | 500 | 330 | 555 |
| cleanup | 398 | 607 | 430 | 640 |
| calls | 282 | 282 | 282 | 282 |
| registry | 35 | 35 | 35 | 35 |
| clean | 148 | 148 | 168 | 236 |
| extra | - | 230 | 40 | 178 |

**Explanation:**

Here all tree flattening techniques need more time to finish the cleaning up. Insertion needs the less extra time because it generates the least amount of extra messages.
Moreau and Zombie-Deletion both generate a lot of extra messages (230 and 178 respectively) thereby consuming lots of processor time. On a uniprocessor these extra times are directly added to the final end time.

**Benchmark 4 : Shortest Path**

In this benchmark, a graph is constructed with 7 masters (see fig 6.7).

| | Normal | Moreau | Insertion | Zombie |
|---|---|---|---|---|
| startup | 17 | 34 | 18 | 18 |
| complete | 33 | 66 | 34 | 35 |
| cleanup | 57 | 96 | 56 | 57 |
| calls | 55 | 55 | 55 | 55 |
| registry | 50 | 50 | 50 | 50 |
| clean | 63 | 63 | 63 | 63 |
| extra | - | 72 | - | - |

**Explanation:**

This benchmark is an example where remote pointers are looked up by the individual processes (masters). Therefore all the diffusion trees are wide, but not deep (maximum two deep). This characteristic does not trigger the Insertion or Zombie deletion-algorithms because the tree is in a fairly good shape. Therefore both these algorithms have similar times and number of messages as the original IRL.
Moreau however, will always be invoked when the depth of the tree is higher than one, which is the case here. This creates a large number (72) of extra messages which are fairly undesired because they only consume extra processing power, but do not gain anything.

### 6.4.3 Multi-processor Results

Benchmark-results become more interesting when the benchmarks are performed on a network of computers. This way the computation and DGC-activities are shared among these computers. These tests are all performed on a network of 6 UltraSparc-5 stations running Solaris. Five stations will be running the benchmark-masters and one will run the registry.

**Benchmark 1 : Cycle**

The first benchmark is a Cycle with five master-objects in the cycle and it will cycle five times around before starting to break it down again.

| | Normal | Moreau | Insertion | Zombie |
|---|---|---|---|---|
| complete | 2 | 2.5-3 | 3-3.5 | 2.5 |
| cleanup | 4.5 | 3.5 | 4.5 | 3.5 |
| calls | 24 | 24 | 24 | 24 |
| registry | 8 | 8 | 8 | 8 |
| clean | 21 | 21 | 26 | 42 |
| extra | - | 44 | 10 | 36 |

**Explanation:**

The idea of this benchmark is that the completely unbalanced diffusion-tree is reshaped during the computation such that it can be broken down faster afterwards.
The most interesting results of the benchmark is that both Moreau's technique as Zombie deletion create a large number of extra messages, but their total times are faster than the original IRL. The benchmark is designed in such a way that at most only one processor is actually working on the actual computation at a certain time. All the other processors are idle or doing DGC-activities. This leaves a lot of space for parallel execution of these extra messages. Especially because they are directed into the reverse direction of the computation, these processors are probably currently not involved in the computation. Therefore both algorithms achieve a desirable speedup.
Minimize-by-Insertion does not achieve this speedup for different reasons. In this case only five processors are in the cycle, creating a diffusion tree of exactly five levels. This fifth level is the minimum requirement for the insertion-algorithm to start changing the tree. Because this five levels are just on the boundaries, the gain of applying them is not big. If longer chains would be constructed in the diffusion tree, more insertions would occur, but the gain at the end would be bigger and create a speedup. In the current setup, the cost has the same order as the gain, resulting in a similar end-result as IRL. It must be noted however that the final result is not slower than the original IRL, but the diffusion tree has been improved at the same time.

**Benchmark 2 : Diffuse**

The second benchmark is a Diffuse with five master-objects and the remote reference is sent each time to three other processes and sent three levels deep.

| | Normal | Moreau | Insertion | Zombie |
|---|---|---|---|---|
| cleanup | 21-25 | 27-28.5 | 21-25 | 25-26 |
| calls | 36 | 36 | 36 | 36 |
| registry | 10 | 10 | 10 | 10 |
| clean | 35 | 35 | 35 | 40 |
| extra | - | 38 | - | 5 |

**Explanation:**

The idea of this benchmark is that the same reference is sent to processors that already know that reference (after a while). Different tree flattening techniques will create different overheads, but in

all cases their gain is very limited (if any) because the final diffusion tree will maximally contain five nodes (the five masters).
Moreau's algorithm creates the largest number of extra messages and thus creates the most overhead and becomes the slowest. Zombie deletion only creates 5 extra messages and is only slightly slower than IRL. Minimize-by-Insertion does not even generate extra messages because it considers the current diffusion tree not in a bad shape (its depth does not exceed five levels). Therefore its execution times are equivalent with the ones from normal IRL.

## 6.5 Conclusion

The new tree flattening algorithms proposed in this thesis are validated in different ways. First a simulation model has been constructed to simulate the different algorithms. It is shown that Moreau's technique creates a large number of extra messages compared to Minimize-by-Insertion.
The next validation is an implementation of these techniques in two java-programs (PicObj and RMI-clone) using the same DGC-algorithms. On top of the RMI-clone several benchmarks are constructed to show the difference in execution time when applying different algorithms.
The multi-processor-results for the first benchmark show that both Moreau's algorithm and Zombie-Deletion have a speedup. The large number of extra messages are processed in parallel and reshape the diffusion tree in such a way that cleaning is done faster. Minimize-by-Insertion does not create a large number of messages, neither does it modify the diffusion tree a lot. This results in the same execution time as IRL (the gain of the algorithm equals the cost).
The second benchmark shows that Moreau's algorithm and Zombie-Deletion perform slower than standard IRL. The reason is that both create a large number of messages, but do not improve the diffusion tree enough to create a speedup. Minimize-by-Insertion however, ”thinks” that the diffusion tree is not in a too bad shape (not exceeding four levels) and will not create extra messages. Therefore it's execution time is similar to plain IRL.
The next chapter will go more in depth in implementation details and specific problems encountered while implementing IRL and the tree flattening algorithms.

# Chapter 7

# Implementation Details

## 7.1 Introduction

In the last chapter, both implementations for PicObj and a java RMI-clone were introduced to prove the different algorithms proposed in this thesis and to allow some comparison between different tree flattening techniques. This chapter will go deeper in the design issues encountered while implementing both programs. Therefore the chapter will be split into three different parts: first the general framework used by both programs will be explained, then each program in more depth.

## 7.2 General Framework

This section will explain the different subsystems of the core of the Distributed Garbage Collector.

### 7.2.1 Manager

The manager is the very deepest middlepoint of the DGC. All actions pass somewhere through this class. It is responsible for receiving messages, sending cleancalls, managing the object table and many other things.
In each different space there will be exactly one instance of this class. This is enforced by a simple implementation of the Singleton-pattern which uses a caching static method returning the singleton. The first time this singleton is requested, it is created and returned. On creation, it performes a number of initializations like setting variables, reading configuration files and starting a serversocket.
Two threads are created on startup: the first runs the local garbage collector every second (this timeout can be adjusted). At the same time, it can be configured to send local information (like number of generated messages, memoryconsumption ...) to the space containing the registry. The registry will collect all this information and print some statistics to the output on regular times. The second more interesting thread launches a serversocket. This serversocket accepts all kinds of messages from all spaces. Each time a messages arrives, it creates a different thread and executes the message in it. This way, the reception of messages is never blocked. The message itself however can be blocked due to

some synchronization with other messages or local threads. The execution and sending of messages will be explained in a section below.
The manager is also in charge of handling incoming and outgoing objects. Each time a remote accessible object is sent to another space, it will be put in the object table (if it was not already there). The same happens if such an object is received from another space. If not known before, it is put in the object table, otherwise the manager will start a procedure to send a cleancall back to the space where it is received from. Internally this object table does not contain normal pointers to these remote accessible objects, instead it keeps a pointer to a WeakReference that points to the object. This WeakReference is negelected by the local garbage collector, so if an object is only referenced by WeakReferences, it is considered as not referenced and can be collected. This is necessary because the object table is no root of the local garbage collector as stated in chapter 3.

### 7.2.2 Protector

The second important module of the DGC is the Protector. DGCs work, in contradiction to LGCs, by preventing objects from being garbage collected. Basically this is always done by holding extra, artificial references to them by the Protector. The second task of the Protector is to prevent race conditions, by synchronizing different concurrent accesses to the same variable.
References to objects can be hold by five different collections. Four of them use extended version of HashMap and HashSet. These extensions are SynchronizedHashMap and Synchronized HashSet which encapsulates the normal HashMap and HashSet and provides most of the normal operations on it, but all operations are synchronized with each other. The sixth collection is more special: Finalization is constructed to minimize the time waiting for the next available object for finalization.

#### Object Table

As explained before, this table contains references to all remote accessible objects from this space which are referenced from another space and remote accessible objects from other spaces which are referenced from this space. This table is the most widely used table to lookup remote references.

#### Local Objects

The LocalObjects, also called OwnSet in some literature, contains all objects that are owned by this space and are referenced from another space. Because the object table is not a root for the LGC, it could reclaim these objects. To prevent that, they are put in the LocalObjects as long as they are remotely referenced. In normal IRL stubs are also put in this set if they have children. When using Zombie Deletion, this is not the case: when a stub is no longer referenced locally, it will be scheduled for finalization (there the children are sent for adoption to the root).

#### Informs

When a migration is requested, a message is sent to the root to start informing all other spaces in the tree that the object has moved. Each time a node sends this information to a child, it puts this space in the Informs-set. When that space has informed all his children, it will send an acknowledge back. This acknowledge will remove the space from the list and when all acknowledges are received (Informs is empty), it will send his acknowledge to his father.

#### Blocking

When a root of a diffusion-tree receives a request to adopt some children, it will block this object locally. this means that while he is adopting children, the object will never be garbage collected. Each object has an internal blocking-counter. Each request for an adoption will increase this counter. The first adoption will add the object to the Blocking-set. When an adoption is finished, the counter is decremented and when it reaches zero, the object is removed from the Blocking-set.

#### Finalization

Each time an object is detected non-reachable by the LGC, it is put in this Finalization. A special thread (explained at the Manager) is in charge of taking an object from this set and perform all finalization actions (sending cleancall, send children for adoption...). When finished, it asks the next available object to be finalized. This set is designed in such a way that the finalization-thread is sleeping as long as there are no objects to be finalized to improve overall performance.

### 7.2.3 Space

Each logical virtual machine will be identified by a Space. This space consists of the combination of the IP of that machine and a portnumber. On this port the socketserver of the manager will listen to incoming messages. By combining both IP and portnumber, a maximum flexibility is obtained which allows multiple spaces on a single machine. It is however impossible to create multiple spaces inside one java virtual machine because some parts of the system depend on static variables and static methods which would interfere with each other. The same effect however can be obtained by using the Interleaved Virtual Machine, which loads the same classes by different classloaders and creating virtual spaces inside one java VM (see 7.5.1).

### 7.2.4 WireRep

When transferring a remote accessible object or a stub to another space, a wire- representation is sent instead of the complete object or stub. This has several benefits. This wire-representation only contains the basic information needed by the other process to communicate with the object. In standard IRL, it only contains the space of the root (where the real object is) and the space where it received the object from (its father). More complex implementations will add fields to it (e.a. to provide flattenings or migration).

### 7.2.5 Registry

The registry is a vital part of a distributed environment, creating a bridge between different spaces that do not know each other. Instead of directly asking a space for a specific object, it is more general that that space sends the object to a registry and any other space can now request that object. The functionality delivered by the registry is exactly the same as any other registry, following the Registry-interface of java. Internally on that process, the registry is an ordinary program, conform to the specifications of this distributed environment. It consists of a manager, a protector and all other necessary constructs, needed by any space. In addition to that, it also reacts on specific messages to perform the registry- operations like bind or unbind. The processing of these messages, and also the processing of the reception and sending of objects happens identical as in any other space.

### 7.2.6 Messages

All processes in the distributed system are able to send a number of different messages. The most obvious is to invoke a method on a remote object, but also a lot of other messages will be sent to keep the DGC consistent and running. These messages are designed in a hierarchy, whereby all messages somewhere inherit from Message. This class contains all knowledge to connect to the remote space, send itself and get the result back. The real computation and extra fields will be specified by subclasses. This makes the sending and reception of messages very easy and consistent.

### 7.2.7 Serialization

While communicating between remote spaces, objects need to be serialized. Standard java serialization is used for all of the objects, however sometimes extra calculations are performed. All arguments passed as arguments or return value of a remote method invocation must be serializable. Also WireRep and Space are serializable because they are invisibly sent. Remote accessible objects and stubs however are never serialized. Instead, their wire-representations are serialized. This is sufficient because these wire-representations contain all the relevant information for the other space.

## 7.3 PicObj

As explained in section 6.3.1, the prototype based language is extended with remote references to dictionaries. Therefore the language grammar had to be extended with two extra constructs.
Firstly, an abstract class GrammarRemote is added which deals with everything concerning remote accessible objects:

- contains the reference list
- contains a wire representation
- knows how to write itself to a stream (Serialization)

- can send its children for adoption to the root
- puts itself in the finalization-queue

The most important subclass of GrammarRemote for PicObj is GrammarDictionary. In the original Pic%, this class was on the same level as all other grammar constructs, but now dictionaries are remotely accessible and need to inherit the extra information from GrammarRemote.

Because now PicObj becomes multi-threaded, it is unsufficient to have two global dictionaries in static fields. To solve this problem, the two dictionaries can be kept in two different places:

- in static fields for the original thread started by the commandline
- in the thread-object that receives a remote call

The lookup of theses dictionaries will depend on the type of the current thread.

To allow migrations in PicObj, the original object must block the arrival of new messages until it has arrived on its final space. This migration happens in different stages:

- dictionary in original space start blocking new messages
- dictionary is copied to new space
- original dictionary will redirect all messages to new space
- in the local space, all references to the original space are replaced by remote references to the copy
- in the new space, all remote references to the original dictionary are replaced by local pointers to the local copy
- original dictionary start sending updates to all his children to inform them that the object has been moved
- when all updates have finished, the object is discarded

The second subclass of GrammarRemote is GrammarRemoteReference. This class represents a remote pointer to a remote accessible object (a dictionary in PicObj). This class is extended with two important methods, to react on migration request and remote method invocations issued by the language. Because PicObj is not typed, no different stubs and skeletons are needed for each different type. The skeleton behavior is incorporated in the GrammarDictionary and the stub behavior in the GrammarRemoteReference.
To allow a programmer in PicObj to use remote dictionaries, several extra natives have been added:

- registry natives: bind, rebind, unbind, lookup, list
- to trigger the local garbage collector: gc
- to migrate a remote dictionary to this space: migrate

## 7.4 Java RMI-clone

This RMI-clone has been constructed after the PicObj implementation was finished. Originally the idea was to only implement the DGC for PicObj. This delivers a clean and easy language, but not a large amount of persons know this language and it is more difficult to create the benchmarks in it (extra primitives needed like threads and time).

The original framework used by PicObj was adapted to be used for Java RMI with a minimum of changes. Firstly a general class is constructed which will be used as superclass of all remote accessible objects in java: JavaRemoteObject. This class inherits from GrammarRemote (see 7.3) and inherits all methods and fields needed for the DGC. In addition one extra method is added to get the skeleton for this remote accessible object.

Because java is a typed language, each class has to have it's own skeletons and stubs. These are automatically generated by the "Generator". This class will read a class through reflection and create the stub and skeleton for it. The stubs will inherit from JavaRemoteObjectStub (which inherits from GrammarRemoteReference) and contains one important method: **call**. This method sends the method-number and the arguments to the right space where the object is located. The stub itself consists of the same methods as the original class, but with different bodies. Each body will collect all parameters in an array and call the method "call". The result is then returned if necessary. Primitive values are wrapped and unwrapped at the skeleton (inverse for the return value). The skeleton inherits from JavaRemoteObjectSkel and contains a pointer to the real object and one important method: dispatch. This dispatch will unwrap the parameters and call the right method on the real object.

However the framework supports migration, this is not possible in java. In PicObj this was possible because there we had full control over the language. When an object migrates, all pointers to the original object must be changed into pointers to a remote reference. This is possible because PicObj is build on top of java and all references can be looked up (although it is costly).
In java however, this approach is not possible because java is not build on top of another language and we don't have access to these internal pointers. Therefore we can never know which objects still reference a certain object, so these pointers can't be changed.
A possible solutions is to construct the remote accessible objects in such a way that they can act as local objects (the normal case), but can change into remote references if the object migrates. However this blurs the conceptual difference between the normal object and the remote reference.

## 7.5 Additional Details

### 7.5.1 Interleaved Virtual Machine

To ease the debugging and testing of this distributed program, the Interleaved Virtual Machine was used. This software, developed by Marc Ségura-Devillechaise, allows the user to start different programs in different namespaces. This simulates a number of different java virtual machines inside one real VM. It was used primarily because normal threads were not sufficient. Vital information was kept in static variables which would have been shared by these threads. In this Interleaved Virtual Machine, the different VMs are created by loading classes in different classloaders, creating complete different namespaces.

### 7.5.2 Local Garbage Collector of Java

Java has a standard LGC incorporated deep in the language, more precisely in the Virtual Machine. The java VM-specification says that the complete implementation (also the algorithms used) to do the garbage collection are free for the implementer of the VM. This means that each different VM from a different company or even from the same company but different version, can have different local garbage collectors.
The DGC relies on this local GC to clean certain crucial objects, which will trigger some DGC-activities (sending clean calls or adoptions). Ideally, an LGC will clean all local objects that are not used anymore. Normally, most garbage collectors will recuperate almost every unused object, but in many cases there will be some leaks too. These leaks are very difficult to detect, and are not extremely important in normal applications. If 99.99% of the garbage is collected, everybody will be happy and nobody will see that one or a few objects are not collected.
In the DGC, things become more complicated. What happens if one of the objects that are not collected, should trigger the DGC and send a clean call to his father. Now this call is never sent and the father and the rest of the diffusion tree (up to the root) will never be freed.
One example is when a subclass of Thread is constructed which has extra instance variables that can hold remote accessible objects. The constructor will initialize these fields and the values will be unchanged until the thread ends. After ending the thread, one could think that the thread-object (containing the remote accessible objects) would be collected. This is not the case in SUN's VM and the objects were never be collected.

### 7.5.3 Configuration

To configure the DGC and especially the different benchmarks, several configuration files are used.

#### Config.txt

This is by far the most important one. It specifies whether the DGC will work for PicObj or for java. This is necessary to specify because there are some small changes in the DGC (e.a. java RMI needs to load the correct stubs, but PicObj needs to load a generic stub).
Next, this file also indicates which tree flattening algorithm will be used (if any). Possible choices are Moreau's algorithm, Flattening-by-Insertion or Zombie Deletion.

#### Processors.txt

This file lists a number of urls. These urls will be used to bind the different RemoteExecutes into the registry of standard RMI. This is used for easy testing: On different physical machines, a RemoteExecute is started. This object will bind himself (with one of these names from the file) into the (real java) RMI-registry. The last program executed is a ControlCenter which takes as arguments benchmark arguments. This ControlCenter will communicate with all the different spaces and they start performing the benchmark. After finishing, the ControlCenter will stop, but all RemoteExecutes will keep alive for future tests. This way multiple tests can be started from one command line. The RemoteExecutes and the ControlCenter will communicate through standard RMI, to keep a clear seperation between the different DGCs and to do not influence the benchmarks.

#### Shortestpath.txt

This file contains the graph and the start and end point for the Shortest Path Benchmark. The first line contains the start point, a tabulation and the endpoint. Points are represented by integers. The next lines will represent edges: startpoint of the edge, a tabulation, the endpoint, a tabulation and the weight (Integer).

## 7.6 Conclusion

This chapter went deeper into specific details of the implementation. These details have been split up into three distinct parts. The first dels with details of the general framework; the part of the real DGC. The second and third part are more specific about the PicObj- and java RMI-clone-implementations. Here the details of these implementations have been exposed together with the link with the framework.

# Chapter 8

# Future Work

## 8.1 Benchmark suite

Distributed Garbage Collection is a very specific domain which is ideally not noticed by the programmer. All algorithms have in common that they want to minimize the produced overhead. Especially in runtime-systems, it is unacceptable that a process must wait for some time because the DGC is doing some cleaning. Although IRL behaves generally better than most other DGCs (e.a. tracing algorithms) because it is interleaved with the mutation, it still creates some overhead. Further research is necessary to minimize this overhead to an absolute minimum.
To be able to more reliably evaluate the overhead of a DGC, it is necessary to construct a number of benchmarks. Each benchmark should stress one specific point of a DGC, e.a. reactiveness. It is impossible to determine the overhead of a DGC only by doing one or a few benchmark. Only when all different sides of the DGC are stressed, one can make a more reliable conclusion.

## 8.2 Applicable to other DGCs

Many of the ideas stated in this paper could be applied to other DGC-algorithms too. Especially the Zombie-Deletion, Flattening-by-Insertion and keeping the tree intact while migrating could easily be incorporated in a Indirect Reference Counting DGC. For example, keeping the tree intact while migrating could solve some problems in IRC when combined with Dickman's second algorithm. That algorithm chooses its parent according to its depth in the tree. By not changing the tree, the depths keep the same, thereby improving the correctness of the choosing of the father.

## 8.3 Delaying or not sending DEC-messages

In some cases, a certain reference is sent many times from Space A to Space B. If this reference was already known by that space, it has to send back a DEC-message. These DEC-messages can be delayed. In case of a IRL, just one DEC-message must be sent. In case of a IRC, a multi-DEC

can be sent. This multi-DEC decrements the counter not by 1, but by many according to how many DEC-messages it simulates.
Another promising alternative is the delaying of DEC-messages with zombie-deletion. No DEC-messages are sent if that pointer was already known in this process. Instead the new father is added to his list of fathers. When the pointer is no longer locally referenced, it moves all its children (if any) to the root and send DEC-messages to all its fathers. This has the same benefit as the algorithm above to reduce the number of DEC-messages.

## 8.4 Combining LGC and DGC

The future for Distributed Garbage Collectors is by tightly coupling local and distributed collectors. This can significantly reduce the time spent in the local garbage collector. If a stub is only reachable by one object that is only reachable by one other space, it can take some time before both are cleared. First a DEC-message is sent to this space to decrement the counter of the object. If the counter is zero, it must wait until the system schedules a new LGC before it can be claimed. When both LGC and DGC are better coupled, this delay could be minimized.

## 8.5 Fault-Tolerance

Fault-Tolerance is important for long-living systems. In these systems it is important that all garbage is collected even if some parts of the system fail or are temporarily unavailable, otherwise some subsystems may run out of memory. Tree flattening techniques are one step towards a more reliable diffusion-trees, but this is not enough. When a node fails, its father can determine this, but it can not know how many and which children this node had before it failed. An interesting path is distributing fault-tolerance-information together with the communication needed by the tree flattening algorithms. This way the extra communication needed for fault tolerance is minimized.

## 8.6 Conclusion

The new ideas proposed in this thesis are just a piece of a larger amount of general optimizations for Indirect Reference Listed DGCs. It is important to be able to create an independent benchmark suite to evaluate and compare different DGC-algorithms and their optimizations in an independent way.
Important to explore is also the applicability of these new algorithms with different DGC-algorithms, most likely IRC-based algorithms.
Finally these tree flattening algorithms reshape the diffusion tree to make it less vulnerable to node-failures. In this optic, a combination with fault-tolerance-techniques should be explored.

# Chapter 9

# Conclusion

The invention of Automatic Garbage Collection divided the programmers into two different groups of people. The first group believes that automatic collection will always stay slower than manual freeing. In their eyes, the programmer has enough knowledge to decide when to free a certain piece of memory and this way will create the least overhead. Especially in reactive systems and real-time systems it is unacceptable to delay the actual computation too long to do it in an automatic way. The second group believes that programmers make too much mistakes while freeing pieces of memory. In certain cases, it is even not possible to decide always correctly. More especially in distributed applications, it becomes more and more difficult to decide whether a piece of memory is still needed or not.

LGCs and DGCs solve these problems by managing the memory in an automatic way. In all cases however, the overhead must be reduced to a minimum. LGCs are optimized during the last decades to free memory so fast that it is difficult to detect when a LGC is working or not. For DGCs, the impact on the computation will be higher because inter-processor-communications are needed to decide whether a certain piece of memory is still needed. Also here, several optimizations and new techniques have been proposed to minimize this overhead (especially the needed communications).

IRL is one of these optimized algorithms that only sends one kind of messages when a remote pointer is no longer used in a space (a DEC-message) . Although one message is probably the lowest one can go, there are some negative points about this technique: zombie-references and unbalanced deep diffusion trees. Different researchers proposed different techniques to reshape these trees.

Moreau reroots all nodes from deep in the tree to the first level, this creates an ideal tree, with the least third-party-dependencies, but on a cost of a lot of messages (two for each node). This thesis proposed two other different techniques to solve the same problem. The first will be triggered when a new node is added on a fifth level (deep in the tree). The father of this new node will now be rerooted to become a child of the root. This lowers the depth of the tree with one or two levels and re-places (possibly) many nodes at once. Moreau always reroots each node by itself. The second algorithm will delete zombie-references (nodes that are not locally referenced but kept to keep the tree consistent). The zombie-reference will ask the root to adopt its children and then frees itself. This minimizes the third-party-dependencies needed to keep the tree consistent, and as side-effect lowers the depth of the tree can be lowered.

These four techniques (plain IRL, Moreau's, Minimize-by-Insertion and Zombie-Deletion) have been implemented in a java-framework and two instantiations of this framework were constructed: PicObj and java-RMI clone.

To make some kind of comparison between these four techniques, several benchmarks have been implemented, using the java-RMI clone. The first conclusion that should be made is that the results of all these benchmarks are highly dependent of the used benchmark. The benchmarks discussed in this thesis are not designed to highlight specific parts of certain algorithms, but they are constructed to give an idea of the performance of all algorithms in different (every day) applications.

The Multi-Processor results for the first benchmark (Cycle) show that Minimize-by-Insertion creates a small number of extra messages, but there is no speedup (cost = gain). Both other algorithms (Moreau's and Zombie-Deletion) create a lot of extra messages (respectively 44 and 36) but these are executed in parallel with the computation and do not cause many extra delays for the real computation. The result is that the diffusion tree is in a much better shape when he is going to be broken down. This breakdown now performs faster, resulting in an overall speedup.

The Multi-Processor results for the second benchmark (Diffuse) show how the different algorithms behave when references are sent to the same spaces many times. The diffusion tree itself is never in a very bad shape, therefore Minimize-by-Insertion did not do anything (and therefore has the same time). Moreau's and Zombie-Deletion however keep trying to optimize the diffusion tree and create extra messages. Moreau's algorithm creates 38 extra messages without optimizing the tree in such a way that the breakdown of the diffusion tree happens much faster. The result is a slower performance than plain IRL. Zombie-Deletion suffers the same problem but less bad (only 5 extra messages), such that its performance is between plain IRL and Moreau's.

In general tree-flattening-techniques are appropriate if the extra overhead they create is minimized. They reshape the diffusion tree into a better tree, but it is unacceptable to let the application wait for this too long. However, the extra overhead created by the different techniques depends not only of the used technique, but more of the application and the generated diffusion tree. There does not exist one specific algorithm that is the best, but for each algorithm a best benchmark can be generated and for each application a best tree-flattening-algorithm can be chosen. However in general the tree algorithms are equally acceptable. Minimize-by-Insertion tries to minimize the number of extra messages needed to do the job. In combination with Zombie-Deletion it might be the best option in general.

# Bibliography

[BEN$^+$93] A. Birrel, D. Evers, G. Nelson, S. Owicki, and E. Wobber. Distributed garbage collection for network objects. Technical report, Digital, Systems Research Center, Palo Alto, 1993.

[Bev87] D.I. Bevan. Distributed garbage collection using reference counting. *Parallel Architectures and Languages Europe*, 1987. Lecture Notes in Computer Science, Springer Verlag, volume 259, pp. 176 – 187.

[Cor] Sun Microsystems Computer Corporation. Java remote method invocation specification. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[Dic00] P. Dickman. Diffusion tree restructuring for indirect reference counting. *International Symposium on Memory Management 2000, p 167-177*, 2000.

[DLM$^+$78] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM 21(11):965-975*, 1978.

[DS80] E.W. Dijkstr and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters, vol 11, N 1*, 1980.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence, 8(3) p.323-364*, 1977.

[Jon] R. Jones. The garbage collection page. http://www.cs.ukc.ac.uk/people/staff/rej/gc.html.

[Jon96] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[LSB92] T. Le Sergent and B. Barthomieu. Incremental mutli-threaded garbage collection on virtually shared memory architectures. *International Workshop on Memory Management p.179-199*, 1992.

[Mor98] L. Moreau. A distributed garbage collector with diffusion tree reorganization and object mobility. *Proceedings of the third International Conference of Functional Programming 1998, p 204-215*, 1998.

[Mor01] L. Moreau. Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *to be published*, 2001.

[Piq91] J.M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. *Parallel Architeectures and Languages Europe 91*, 1991. Lecture Notes in Computer Science, Springer Verlag, volume 505, pp. 610 – 619.

[PS95] D. Plainfossé and M Shapiro. A survey of distributed garbage collection techniques. *International Workshop on Memory Management, Kinross, Scotland (UK)*, 1995.

[PV98] J.M. Piquer and I. Visconti. Indirect reference listing: A robust distributed gc. *EuroPar'98, Southampton, UK*, 1998. Lecture Notes in Computer Science, N. 1470, pp. 610 – 619.

[Tan88] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, second edition, 1988.

# Appendix A: RMI-CalenderServer

```
/**
 *  CalenderServerInterface by Kristof De Vos 2001
 */
import java.rmi.*;
import java.util.*;

interface CalenderServerInterface extends Remote
{
  Date getDate() throws RemoteException;
  long getTime() throws RemoteException;
}

/**
 *  RMICalenderClient by Kristof De Vos 2001
 */
import java.util.*;
import java.net.*;
import java.io.*;
import java.rmi.*;

public class RMICalenderClient implements Calender
{
  protected CalenderServerInterface server = null;

  public RMICalenderClient()
  {
    try {
      server = (CalenderServerInterface)Naming.lookup("CalenderServer");
    } catch (RemoteException ex)
    {
      System.out.println(ex);
      System.out.println("couldn’t lookup server");
    }
  }

  public Date getDate()
  {
    try {
      if (server==null) return null;
      return server.getDate();
    } catch (RemoteException ex) {System.out.println(ex);}
    return null;
  }

  public long getTime()
  {
    try {
      if (server==null) return 0;
      return server.getTime();
    } catch (RemoteException ex) {System.out.println(ex);}
    return 0;
  }

  public static void main(String[] args)
  {
    RMICalenderClient cl = new RMICalenderClient();
    System.out.println(cl.getDate());
    System.out.println(cl.getTime());
  }
}


/**
 *  RMICalenderServer by Kristof De Vos 2001
 */
import java.util.*;
import java.net.*;
import java.io.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

class RMICalenderServer extends UnicastRemoteObject
      implements CalenderServerInterface
{
  public RMICalenderServer() throws RemoteException
  {
    try {
      Naming.rebind("CalenderServer",this);
    } catch (RemoteException ex) {System.out.println(ex); }
    System.out.println("RMI-Server up and running");
  }
  public Date getDate(Date test) throws RemoteException
  {
    return new Date();
  }

  public long getTime(long test) throws RemoteException
  {
    return new Date().getTime();
  }
}
```

# Appendix B: Clone-CalenderServer

```
/**
 *  CloneCalenderClient by Kristof De Vos 2001
 */
import java.util.*;
import java.net.*;
import java.io.*;
import java.rmi.*;
import dgc.*;

public class CloneCalenderClient implements Calender
{
  protected CalenderServerInterface server = null;

  public RMICalenderClient()
  {
    try {
      server = (CalenderServerInterface)
            Manager.getLocalRegistry().lookup("CalenderServer");
    } catch (Exception ex)
    {
      System.out.println(ex);
      System.out.println("couldn’t lookup server");
    }
  }

  public Date getDate()
  {
    try {
      if (server==null) return null;
      return server.getDate();
    } catch (RemoteException ex) {System.out.println(ex);}
    return null;
  }

  public long getTime()
  {
    try {
      if (server==null) return 0;
      return server.getTime();
    } catch (RemoteException ex) {System.out.println(ex);}
    return 0;
  }

  public static void main(String[] args)
  {
    CloneCalenderClient cl = new CloneCalenderClient();
    System.out.println(cl.getDate());
    System.out.println(cl.getTime());
  }
}


/**
 *  CloneCalenderServer by Kristof De Vos 2001
 */
import java.util.*;
import java.net.*;
import java.io.*;
import java.rmi.*;
import dgc.*;

class CloneCalenderServer extends JavaRemoteObject
      implements CalenderServerInterface
{
  public RMICalenderServer() throws RemoteException
  {
    try {
      Manager.getLocalRegistry().rebind("CalenderServer",this);
    } catch (RemoteException ex) {System.out.println(ex); }
    System.out.println("RMI-Server up and running");
  }

  public Date getDate(Date test) throws RemoteException
  {
    return new Date();
  }

  public long getTime(long test) throws RemoteException
  {
    return new Date().getTime();
  }
}
```

# Appendix C: Clone-CalenderServerStub and -Skel

```
// stub automatically generated

import dgc.*;
import dgc.messages.*;
import java.rmi.*;

public class CalenderServerStub extends JavaRemoteObjectStub
             implements CalenderServerInterface
{
  static final int method_getDate_0 = 0;
  static final int method_getTime_1 = 1;
  public java.util.Date getDate(java.util.Date arg0) throws RemoteException
  {
    Object[] args = new Object[1];
    args[0] = arg0;
    Object o = MessageFactory.getLocalMessageFactory().sendRemoteJavaCall(
      Manager.selfSpace(),getWireRep().getOwnerSpace(),this,method_getDate_0,args);
    return (java.util.Date)o;
  }
  public long getTime(long arg0) throws RemoteException
  {
    Object[] args = new Object[1];
    args[0] = new Long(arg0);
    Object o = MessageFactory.getLocalMessageFactory().sendRemoteJavaCall(
      Manager.selfSpace(),getWireRep().getOwnerSpace(),this,method_getTime_1,args);
    return ((Long)o).longValue();
  }
}

// skel automatically generated

import dgc.*;

public class CloneCalenderServerSkel extends JavaRemoteObjectSkel
{
  static final int method_getDate_0 = 0;
  static final int method_getTime_1 = 1;

  public Object dispatch(int message, Object[] arguments)
```

```
                          throws java.rmi.RemoteException
  {
    switch(message)
    {
      case(method_getDate_0) :
      {
        Object o = null;
        java.util.Date arg0 = (java.util.Date)arguments[0];
        o = ((CloneCalenderServer)obj).getDate(arg0);
        return o;
      }
      case(method_getTime_1) :
      {
        Object o = null;
        long arg0 = ((Long)arguments[0]).longValue();
        o = new Long(((CloneCalenderServer)obj).getTime(arg0));
        return o;
      }
    }
    return null;
  }
}
```