



Evaluation of different Distributed Garbage Collection Algorithms in the Mozart Language

by
Zacharias El Banna

Master's Thesis in Computer Science
at the Swedish Institute of Computer Science,
Royal Institute of Technology, year 2001-2002
Supervisor was Per Brand <perbrand@sics.se>
Academic advisor at SICS was Erik Klintskog <erik@sics.se>
Examiner was Seif Haridi <seif@sics.se>

Abstract

This Master's thesis presents an architecture, developed for the Mozart language, which allows programmers to implement their own specialized distributed garbage collection algorithms, optimizing memory management for distributed applications. This architecture offers support for failure handling, in terms of message delivery among nodes of a distributed system, reducing the effort needed for implementing fault tolerance for the distributed garbage collection algorithms. The architecture has been implemented into the Mozart language. Further, a few of the common distributed garbage collection algorithms found in distributed systems have been adapted to the implemented architecture, for verification and evaluation of their performance. From those algorithms, three types have proven themselves to be viable options in different types of distributed behavior.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my project and thesis supervisor Erik Klinskog for the patience, guidance, help and being my greatest source of information during this project. I also wish to thank my examiner, Seif Haridi, and my supervisor Per Brand, for providing time and helpful comments during my time at SICS.

Thanks also go to all the other people at SICS that I have been in contact with during my Master's project, for the warm and inspiring atmosphere they all have contributed to.

Table of Contents

1	Introduction.....	11
1.1	Scope.....	11
1.2	Limitations	11
2	Background	13
2.1	Single-Address-Space Garbage Collection.....	13
2.1.1	Direct algorithms – Reference counting	14
2.1.2	Indirect algorithms - Tracing	15
2.2	Distributed Garbage Collection	16
2.2.1	Resource overhead	17
2.2.2	Protocols	17
2.3	Distributed Algorithms	18
2.3.1	Reference Counting	18
2.3.2	Indirect Reference Counting.....	19
2.3.3	Fractional Weighted Reference Counting.....	20
2.3.4	Reference Listing.....	21
2.3.5	Time Lease.....	22
2.4	The Mozart Language.....	23
2.4.1	Replication patterns	23
2.5	The Distributed Sub System	24
2.5.1	DSites – The Site pointer	25
2.5.2	Distribution nature of entities	26
2.5.3	Reference consistency.....	26
3	Methods.....	28
3.1	Identified Mutator Operations.....	28
3.2	Component Based Model.....	30
3.2.1	DGC algorithm configuration.....	30
3.2.2	Set differences.....	31
3.3	Serialization of DGC Instances.....	31
3.4	Two-Layer Messages	32
3.5	Failure Handling	33
3.6	Design of a General Interface for DGC Algorithms.....	35
3.6.1	Reference consistency level.....	35
3.6.2	DGC algorithm level.....	39
3.6.3	A typical message-passing scenario.....	41
3.7	Instrumenting DGC algorithms and Instances.....	42
3.7.1	The DGC Configuration unit	42
3.7.2	Interface to the application layer.....	43
3.7.3	Interface to the Distributed Sub System	44
3.8	Implemented DGC Algorithms.....	44
3.8.1	Reference Counting	45
3.8.2	Indirect Reference Counting.....	47
3.8.3	Fractional Weighted Reference Counting.....	48
3.8.4	Reference Listing.....	49

3.8.4.1	Reference Listing Version 1	50
3.8.4.2	Reference Listing Version 2	51
3.8.5	Time Lease.....	52
3.9	Performance and Benchmarking.....	54
3.9.1	Benchmarking system performance changes.....	54
3.9.2	Comparing DGC algorithms	55
4	Results	57
4.1	Accomplished project tasks	57
4.2	Benchmarking System Performance Changes	57
4.3	Comparing Implemented DGC Algorithms.....	58
4.3.1	Mutator operations	59
4.3.2	Token-Ring tests	60
4.3.3	Synchs tests.....	60
5	Conclusions.....	63
5.1	Future Work.....	63
6	Bibliography	65
7	Appendix A - List of Definitions and Acronyms.....	67
8	Appendix B - Failure Protocol.....	68
9	Appendix C - Mutator Operations	69
10	Appendix D - Test Setup.....	75

Table of Figures

Figure 1.	A small yet illustrative computation graph.....	14
Figure 2.	Memory distribution after a tracing collection.	15
Figure 3.	A small distributed computation graph.....	16
Figure 4.	Manager and proxy.	18
Figure 5.	RC. Avoiding race scenario.	19
Figure 6.	IRC. Parent-child relationship.	20
Figure 7.	WRC operations.....	20
Figure 8.	Time Lease update-protocol.	23
Figure 9.	Distributed Sub System layers.....	24
Figure 10.	The DSite representation.....	25
Figure 11.	A remote reference in Mozart.....	27
Figure 12.	Distributed mutator operations.	29
Figure 13.	DGC messages.	32
Figure 14.	Proposed DGC algorithm message container.	33
Figure 15.	Distributed entity with two DGC algorithms.....	36
Figure 16.	An example message scenario.	41
Figure 17.	User time differences for application operations.....	58
Figure 18.	Messages per operation.....	59
Figure 19.	Messages for token ring.	60
Figure 20.	Synchronization test.....	61
Figure 21.	RC operations depicted.....	69
Figure 22.	Indirect Reference Counting operations depicted.....	70
Figure 23.	Fractional Weighted Reference Counting operations.....	71
Figure 24.	Reference Listing Version 1 operations.....	72
Figure 25.	Reference Listing Version 2 operations.....	73
Figure 26.	Time Lease operations.	74

Table of Tables

Table 1.	Summary of distributed mutator operations.	30
Table 2.	RR instance general methods.....	32
Table 3.	DGC failure messages.....	34
Table 4.	DGC failure messages and their specification.	34
Table 5.	Complete Reference Consistency message table.....	34
Table 6.	Home Reference configuration methods.	36
Table 7.	Home Reference mutator operation methods.	37
Table 8.	Home Reference special methods.	37
Table 9.	Remote Reference configuration methods.....	38
Table 10.	Remote Reference mutator operation methods.....	38
Table 11.	Remote Reference special methods.	39
Table 12.	Reference consistency protocol, operation-to-method mapping.	39
Table 13.	Home DGCI methods.	40
Table 14.	Remote DGCI methods.....	40
Table 15.	DGC configuration methods for the application layer.....	43
Table 16.	Command line functions for DGC configuration.	44
Table 17.	Reference Consistency: protocols methods for DGCI's.....	44
Table 18.	Implemented DGC algorithms.....	45
Table 19.	Reference Counting attributes and messages.....	46
Table 20.	Reference Counting: The operations.	46
Table 21.	Indirect Reference Counting attributes and messages	47
Table 22.	Indirect Reference Counting: The operations.	48
Table 23.	FWRC Weight handler.	48
Table 24.	Fractional WRC attributes and messages	49
Table 25.	Fractional WRC: The operations.	49
Table 26.	SiteHandler methods.....	50
Table 27.	Reference Listing Version 1 attributes and messages.....	51
Table 28.	Reference Listing Version 1: The operations.	51
Table 29.	Reference Listing Version 2 attributes and messages.....	52
Table 30.	Reference Listing Version 2: The operations	52
Table 31.	Time Lease attributes and messages.....	53
Table 32.	Time Lease: The operations.....	54
Table 33.	Summary of tests used.	54
Table 34.	Mutator operation tests..	55
Table 35.	Benchmarking old vs. new implementation.....	57
Table 36.	Failure protocol cases.	68

1 Introduction

This Master's Thesis is a part of a Master's project at SICS, Swedish Institute of Technology. It deals with the issue of distributed memory management and especially the creation of a general interface for distributed garbage collection algorithms.

In modern programming languages, memory management is handled either explicitly by the programmer or through a special garbage collection routine. With a garbage collection routine the burden is moved from the programmer to the system. In a distributed environment the need for effective garbage collection and reference management is of even greater importance than for the uniprocess case. This not only comes from the fact that more memory is consumed for each object distributed, together with costly interprocess communication, but failing to handle references in a secure way may result in instability for the whole system.

The Mozart language is an advanced platform for distributed applications, and provides programmers with transparent distribution tools. The implementation parts of the project are made for this language and its environment. The algorithms presented in section 2 are all adapted to the general interface, further they are presented in this thesis with respect to the operations described early in section 3, making it easier to understand the implementation later in that same section. The component based model is a framework for handling distributed garbage collection. A prototype with two algorithms was implemented into the Mozart language before the project.

1.1 Scope

The scope of the project is to analyze, design, and implement an effective general interface for garbage collection algorithms into Mozart, realizing and improving the component based algorithm schema. Thus the interface should allow:

- Support for the programmer to choose between different distributed garbage collection algorithms.
- Support for basic instrumentation of the specific algorithm instances.
- Failure handling for both algorithms and entities.

Further an evaluation of the chosen algorithms is to be conducted with respect to their behavior in terms of used messages for operations. To enable this, tools for benchmarking performance in various cases has to be developed. The focus of this Master's thesis is the modeling of the general interface for algorithms and handling consistency, along with the evaluation.

1.2 Limitations

- Although there are a much greater variety of DGC algorithms than presented in this thesis, the ones covered are those suited to be implemented into Mozart with its particular access structure.

- Any fine-tuning or optimization of DGC algorithms is not considered during development, only the implementation of the basic protocol enabling the DGC algorithms to function in Mozart.
- The interface to instrument algorithms is not required to enable more than basic capabilities at this stage.

2 Background

Garbage collection has been of interest for the programming language community for many years. Many programming languages allow the programmer to allocate and reclaim memory for data, which may be live outside of the lexical scope, so called dynamically allocated data. There are three types of memory allocation: static, stack and heap. Static is typically constants and other memory bound to a location at compile-time. Stack allocated memory is procedure calls and return addresses which are pushed onto the stack and later popped when the procedure returns. The third type, heap allocation is the interesting for garbage collector. This type of allocation is called dynamic; the program allocates and deallocates memory in any order, thus memory may outlive the procedure that allocated it.

Dynamic memory may be managed explicitly by the programmer, but this is often error-prone and may result in dangling pointers or memory leaks.

Another approach is to hand over the responsibility to the run-time system [1][2], and let it automatically reclaim unused data. Traditionally this technique has been found in functional languages but today it is an important part of both imperative and declarative languages.

Garbage collection, which comes in two classes of algorithms – direct and indirect, has always had a reputation for disrupting interactive, as well as real-time, programs and imposing an intolerable overhead on the runtime system with as much as 40 per cent [2] of the execution-time. Modern techniques have now reduced the overhead to the point where garbage collected heaps has become a truly viable option.

With the introduction of distribution in modern programming languages, new distributed garbage collection issues has to be considered.

2.1 Single-Address-Space Garbage Collection

With dynamic data structures, the state of computation can be considered as a many-rooted, directed graph, called the computation graph. The Roots are nodes that provide entry points to the graph, the internal nodes are objects and the edges are references. Objects transitively reachable from the root set, i.e. stack and program variables, are live and all other are garbage which is depicted in Figure 1 where the grey objects are garbage and the white live.

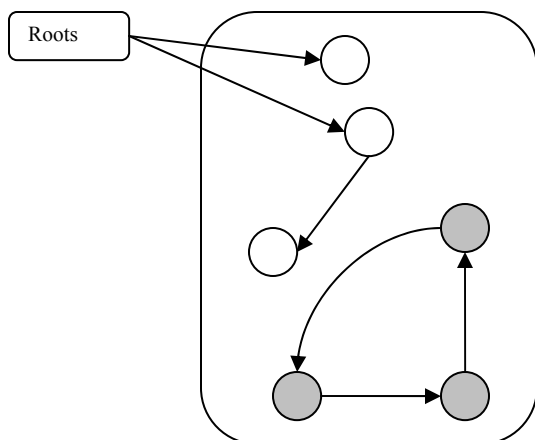


Figure 1. A small yet illustrative computation graph. The objects reachable from the roots are live and the grey objects, forming a circle, are garbage.

The program, or application, that modifies this graph is called the *mutator* and the system component that detects and reclaims garbage is called the *collector* [3]. As the computation progress, addition and removal of roots, nodes and edges modify the graph and render some parts of the graph disconnected from the roots, they will become dead or garbage. An example of this is a file-system where the file table is the root set and all files reachable from this table are live. Deleting all references to a file makes it obsolete.

Most imperative languages make use of explicit reclamation and this leads to two common types of errors: incompleteness (memory leaks) and unsoundness, or the dangling reference problem, i.e. reclamation of accessible cells. A Garbage Collection (GC) algorithm must be correct: both safe in that it only reclaims garbage and live, all garbage will eventually be collected. It can be either direct or indirect in the way it determines the liveness of objects.

Direct methods, which require some kind of record on the object, are known as *counting* methods, storing a count representing the number of pointers referring to that object, reference counting. These records must be kept up-to-date with the mutator activities altering the computation graph.

Indirect, or tracing, methods determine live cells when a Garbage Collector (GC) routine is invoked. This routine traces the roots, hence the name, and when completed, the rest are all garbage cells which are reclaimed. The kind of reclamation depends of the optimization criteria's for the system.

Systems might try to optimize the ratio between mutations and collections or seek to minimize the collector time in any invocation, i.e. interactive or real-time systems.

2.1.1 Direct algorithms – Reference counting

The basic *reference counting* algorithm [4] is the prime example of a direct algorithm. It uses a record, or counter, associated with every object, or entity. Whenever an object is invoked in an operation, the record has to be updated and if the record reaches zero, it is safe to reclaim the space of the object. The

reference counting algorithm obviously imposes a great overhead to the run-time system as it is invoked in every operation, giving a proportional overhead to the number of mutator activities. Another drawback is that objects which are accessible from themselves, thus a part of a circular structure, can not be reclaimed at all. There are solutions to the latter but not described in this thesis.

2.1.2 Indirect algorithms - Tracing

The problem with cycles of garbage is usually overcome by a collector that identifies garbage indirectly. Traversing the computation graph from the roots and identifying all live cells will render all unvisited cells garbage. Thus a cyclic structure not visited is indirectly identified as garbage.

In the simplest form, the *mark-scan* collector [5] delays collection until time of exhausting the memory store. All mutator activities are then stopped. Identification, mark, and reclamation, scan, are two sequential phases where the marking phase traces the graph from the roots and sets a marking bit on all live objects. In the second phase the whole memory store is traversed and cells are either cleared of their marks or reclaimed if unmarked. A problem with the mark-scan collector is that it may trash the memory store if not all objects are of the same size which is depicted in Figure 2. A lot of mark-scan techniques have evolved which solves both the interrupt problem by introducing concurrency and compact the used memory store.

Another tracing algorithm uses another tracing approach. Instead of having two phases, the scanning phase can be eliminated if the cells are relocated as they are identified. The memory store is managed as two or more heaps. When the heap is exhausted, the collector scavenges: simultaneously traversal and copying from one space to another, as shown in Figure 2. When completed, the mutator's current heap is set to the new, copied-to heap. This is known as copy collection [6] and several types of this algorithm have been developed to enhance concurrency and minimize delays introduced by the collector. The two types of tracing algorithms has opposite strengths, the mark-scan algorithm is more efficient if most cells are live, thus only a small portion of the cells are reclaimed at the scanning phase. The copy collection algorithm is intuitively better if most cells are garbage as just a small set of cells is copied to the other heap, resulting in few copy operations. On the other hand the algorithm has serious problems when most cells are garbage as the system might enter a state of massively copying between heaps.

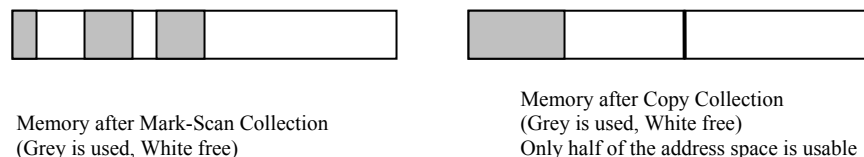


Figure 2. Memory distribution after a tracing collection. As seen the Mark-Scan algorithm might leave memory holes while the Copy Collecting algorithm compacts the space to the price of a smaller address space.

2.2 Distributed Garbage Collection

In this Master's thesis the definition of a distributed system is a collection of autonomous sites, processes that share a communication facility for exchanging messages. With the introduction of distribution, cells are referenced from different sites, which are depicted in Figure 3 showing cells with references from two processes. A reference to a cell on the same site is said to be *local* and a reference to a cell on another site is said to be *remote*.

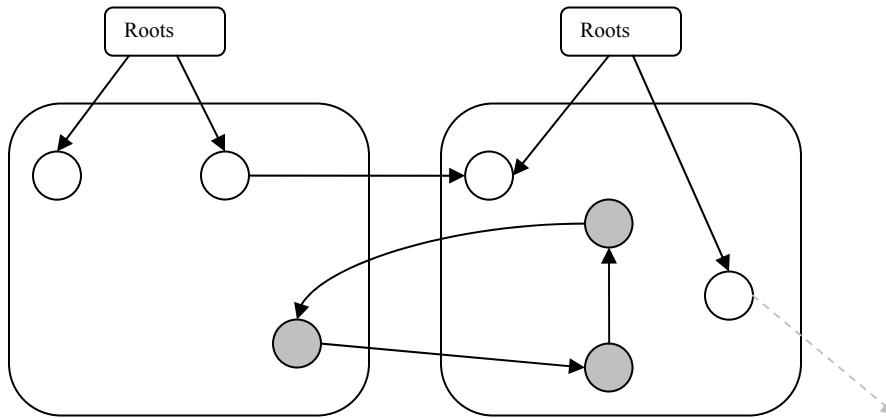


Figure 3. A small distributed computation graph. Two or more processes have a reference to the same object. The grey objects are parts of a distributed cycle.

This introduces some issues with the GC algorithms described in section 2.1. Even though an indirect algorithm may be complete, this approach has several disadvantages when distributed into a large-scale system because it includes some sort of synchronization between its phases. The direct algorithms do not need per-site synchronization, instead access structures and consistency protocols are needed. Some proposals for indirect Distributed Garbage Collection (DGC) designs try to overcome the drawbacks by dividing the system into disjoint spaces (group of nodes). Tracing “locally” within the groups and leave cross-referencing objects to another algorithm or even not reclaim them at all. The direct solutions are much simpler to implement as they are more straightforward. The use of a tracing collector is motivated by its ability to reclaim cycles, something inherently impossible in all counting techniques. However, distributed garbage collection needs to be scalable and that is often achieved at the cost of incompleteness, i.e. reclaiming of only a safe subset of the garbage. Using only tracing DGC algorithms are therefore avoided in favor of performance, and cycles might be left unhandled.

Most systems implement a separate distributed garbage collection routine in cooperation with a local garbage collector. Those solutions are called hybrid garbage collectors and often combine several techniques into the collectors [8][9][10][11].

With the distributed environment, garbage collection becomes a consistency problem since there is no synchronization between local collectors at different sites. If assuming a conservative DGC algorithm, inconsistencies will not be a problem, but the DGC algorithm might be incomplete.

In this thesis the definition of DGC algorithm is an algorithm that can tell whether or not there are references from remote sites to a data structure.

Other issues in a distributed environment is the potential loss of messages, duplication of messages and messages delivered out of order, all which must be considered in a DGC algorithm. However, some of these issues can be avoided if relying on a messaging service offering FIFO message queues and guaranteed delivery.

2.2.1 Resource overhead

With the introduction of distribution new sources of overhead appear. Records for an object must now contain pointers which expands from being just a memory addresses to a whole new expanded access structure which can be, and often is, quite complex. Another issue in a distributed environment is how to find a way to safely identify and distinguish between sites, to express a Globally Unique Id, GUID. Further source of overhead is the communication between processes, or sites, most often is by magnitude more costly than computation, to realize mutator activities.

2.2.2 Protocols

Every distributed system must manage remote references in some way. The common approach is to have an access structure with *managers* and *proxies* handling references for an entity and that structure is similar to a centralized environment with a server and clients. This management, a protocol, is known as the *Reference Consistency Protocol*. As entities are distributed they need an access structure to be able to communicate changes and operations over a network. The realization of this structure is called manager and proxy structures and this is depicted in Figure 4 where an entity at site A has remote references pointing to it. A manager structure, located at the site where the entity is local (site A), called the *Home site* or *Owner site*, handles everything from a distributed perspective. Further it is also a root for the local garbage collector, thus prohibiting reclamation of the entity.

The role of a manager is to be responsible for upholding consistency, executing protocols and both react to, and send, protocol messages from remote references, proxies. It handles acts of mutator activities and detects when objects are not reachable using an algorithm, the DGC algorithm.

The site with the proxy, site B in the figure, is located at the site where the entity is remote. This site is called the *Remote site* or *Borrower site*. A proxy is a structure disguised as the entity but when an operation is invoked on it, it communicates this to the manager. A proxy also reacts to protocol messages, such as forwarding a state, and to local garbage collection operations at its site to know when it is obsolete. The used taxonomy derives from the naming scheme used in Mozart [12] for tables and protocols, and the same scheme is used for simplification throughout the thesis.

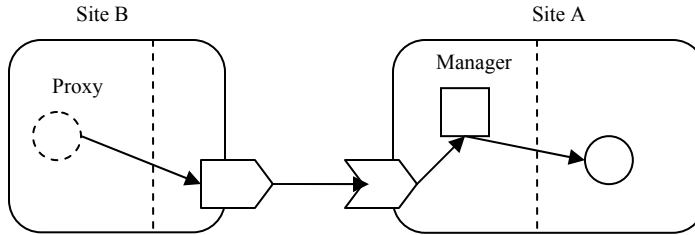


Figure 4. Manager and proxy. The entity located at site A is referenced from site B. The structure to enable operations consists of a Proxy on site B, which when involved in an operation message this to the Manager on the Owner site, through the link built between the sites.

2.3 Distributed Algorithms

There exists a broad range of DGC algorithms today and with the growth of Internet DGC is now receiving its share of commercial attention. For instance, Java RMI [13] and Microsoft's DCOM [14] come with some form of DGC. DGC algorithms of today mainly derive from the two families of GC algorithms: counting and tracing. In addition to these come DGC algorithms using time-outs.

Most practical DGC algorithms of today are hybrids of counting, time-out and sometimes tracing. This section will cover a few of the DGC algorithms but for a more in-depth treatment, there are published surveys [7][15] which covers the area. The chosen DGC algorithms in this section are the distinctive counting algorithms as well as a timed algorithm, all being suitable to the implementation environment. The tracing DGC algorithms are excluded since they are more complex to implement.

Instead of explaining specific hybrids, they can be created, when needed, through the component model later described, allowing for explicit composition of DGC algorithms. The part of the DGC algorithm on the Owner site is referred to as the Home and the part on the Borrower side is referred to as the Remote.

2.3.1 Reference Counting

This scheme has been described for the uniprocess case in section 2.1.1. Each outgoing reference is counted on the Home site and changes are recorded through special increase or decrease messages. In the distributed environment however, there are race conditions, caused by the lack of a global clock, and a solution to the distributed Reference Counting (RC) problem was proposed by Lermen and Maurer [17]. When a reference is passed from an Owner the counter is incremented and when a reference is removed from a Borrower a decrement message is sent which results in decrementing of the counter. However, when another Borrower passes the reference a race condition might occur since one of the Borrowers has to send an increment message to the Owner. If any of the two Borrowers wants to remove the reference they must send a decrement message which is guaranteed to succeed the increment message, otherwise leading to premature reclamation. The Lermen and Maurer solution disallows sending a decrement message before

an increment message have been accepted by the Owner and the protocol implemented in this Master's project uses their algorithm, adapted to Mozart, which works as follow: When a Borrower sends a reference it increments a new counter, located at the Borrower, which represents the dirty-set. Having a dirty-set means that a Borrower has become a temporary local root and thus can not be removed until the condition for the dirty-set has been solved. As long as the dirty-set is non-zero a decrement message can not be sent. When the other Borrower receives the reference it sends a combined increment-and-acknowledge message to the Owner, notifying it of a new Borrower. The Owner increase its counter and sends an acknowledge message to the first Borrower which then decreases its dirty-set counter, the whole scenario is depicted in Figure 5. If the receiving Borrower, however, already has the reference, it can send a simple acknowledge message to the sender as it knows the Owner is aware of its existence.

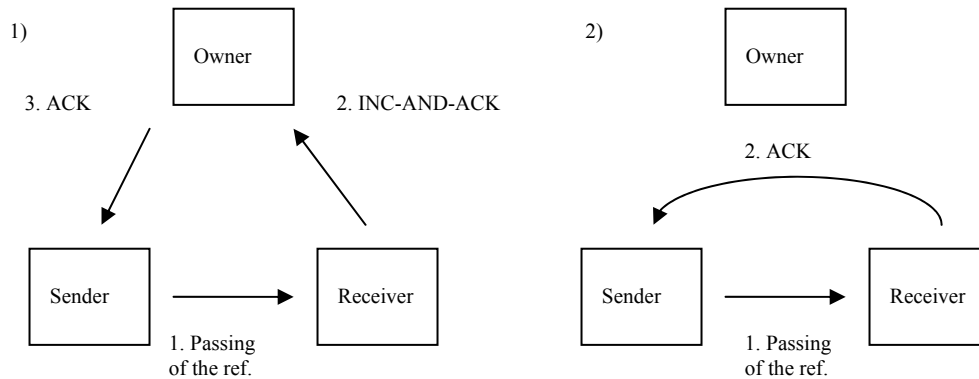


Figure 5. RC. Avoiding race scenario. When a Borrower site passes a reference to another site possible consistency races occur. The above scheme shows the solution to avoid this. If the receiving Borrower site do not have the reference (1) it sends a special message to the Owner to notify it of a new Borrower, the home then sends an acknowledge to the initiating Borrower. In the case where the receiver already has the reference (2) it can safely send the acknowledgement.

2.3.2 Indirect Reference Counting

In Indirect Reference Counting [18] (IRC), every Borrower with reference is a virtual Home when passing the reference to another Borrower, this reduces the dirty-set problem in ordinary RC to a simple two-message protocol. When passing a reference the sender always act as an Owner, using the owner side of the protocol. This leads to a new field for Borrowers in the IRC record: the parent, or sender, pointer. This pointer is set to the first site passing the reference. The counter in RC's Borrower is now a local counter for every outgoing reference.

When the number of children reaches zero the reference might be removed. A special condition in this algorithm is that if a Borrower or Owner sends to a Borrower which already has the reference (and the sender is not the parent), the latter must immediately send a decrement message, otherwise leading to circular inconsistency. The IRC algorithm uses a weak pointer to the real Home for fast access to the entity. The major drawback with this algorithm

lays in the dirty-set which may keep the proxy from being removed if any child points at it. This may create long uncollectible reference chains, as seen in Figure 6.

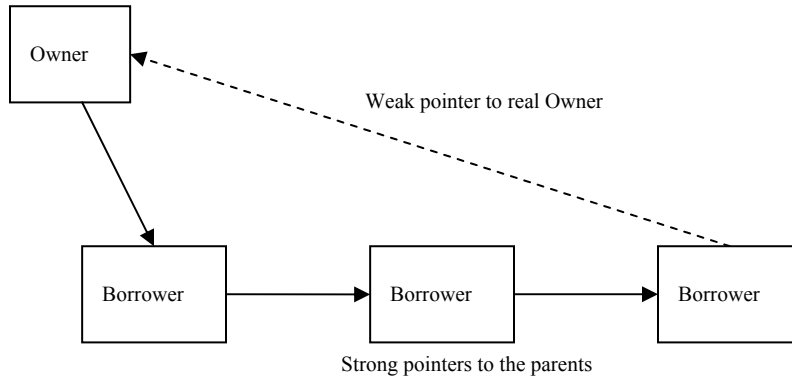


Figure 6. IRC. Parent-child relationship. Every Borrower relies on a parent, thus possibly forming a reference chain.

2.3.3 Fractional Weighted Reference Counting

Fractional Weighted Reference Counting [19] is an extension of ordinary Weighted Reference Counting [20]. Weighted Reference Counting tries to solve race conditions by associating a cardinal, the weight, with each reference. Whenever an Owner or Borrower pass a reference, part of the weight is tagged with it. For instance, if a Home has the weight 128 and pass a reference, this weight might be decreased with 16 to 112. The receiving Remote now has 16 as weight. When that Borrower sends a reference he might split it to 14 and 2, and pass along. When a reference is deleted, all weight is sent back to home. This is illustrated in Figure 7. This scheme obviously avoids races as only the passing of reference is needed, no additional messages. It also preserves an invariant – the total sum of weights is equal (or less in case of failures) to the initial value set by the Owner. A great shortcoming is however introduced: What happens when a reference has run out of weight? The solution to this has been to introduce indirections as IRC, with virtual Homes or secondary weight. This drawback makes the ordinary WRC scheme obsolete and instead the new FWRC scheme is used.

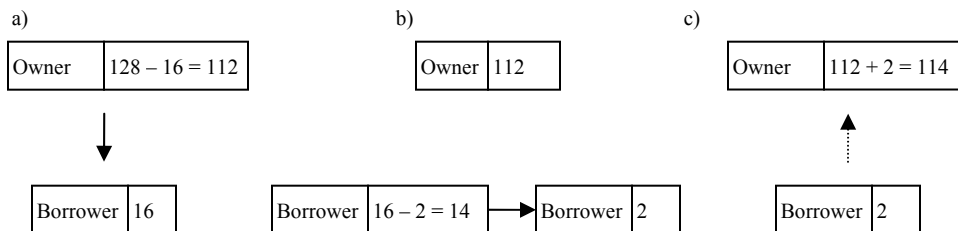


Figure 7. WRC operations. Passing a reference from an Owner (a), or a Borrower (b), means that the local weight is reduced and passed to the receiver along with the reference. When deleting a reference (c) a Borrower sends back its weight.

A solution to the indirection problem comes with Fractional Weighted Reference Counting, developed by Klinskog et al. [19]. FWRC, instead of using a weight which is split as references are passed, has a total sum of one. This sum is then fractioned into partial weights. A weight in FWRC is represented as a pair of an enumerator E and a denominator D , or a power of D , where D is a static integer. The sum of one is equal to D/D^1 , when a ref is passed the Home has $(D-A)/D^1$ weight left and the Remote gets A/D^1 , for any A less than D . When someone has $1/D^1$ left it changes its weight into D/D^2 and might continue as usual. This DGC algorithm has virtually endless of weights and the representation is much smaller than for WRC (typically two 32-bit integers). One can imagine a WRC cell with a huge weight but that leads to much memory allocation for just the weight.

2.3.4 Reference Listing

With distribution, partial failures are introduced in form of message loss and site crashes. The counting DGC algorithms described earlier has no ability to detect and correct failures since their only information about references are at most the number of site, not the distinct sites. Reference Listing (RL) [11][21] is a scheme which overcomes this by using explicit knowledge of referring sites. Implementations [11][21] of this algorithm relies on protocols for message delivery, sequence numbers and trimming messages¹ to uphold consistency. This will not be described in this thesis since the Mozart language implements some of these protocols on other levels. RL implementations can be direct [21] or indirect [11] in that either the Home has all the knowledge of referring sites or the information is kept in an IRC similar way, with every Remote keeping its own reference list. The latter approach introduces serious problems when a Borrower site crashes as the Home has no knowledge of other Remotes behind it. This might be solved with broadcasting messages and timeouts [11]. The direct method proposed by Birell et al. [21] is much more suited to adapt to the implementation environment due to their similar access structures are very similar and use of per-site live message. With this in mind the algorithm can either use a RC similar protocol to avoid races, with increment-and-acknowledge messages, or simply let the sending Borrower be obliged to reliably send an update message to the Owner, containing the new site, before the reference is sent to the receiving site. Thus the Home might in case of a crash of the sending site, safely remove it from the list as the receiving Borrower had not yet received the reference. The increment-and-acknowledge scheme is suitable since the sending Borrower might crash after he has sent the reference, but the receiving site will update the Home the first thing done. Thus when the Home tries to send the last acknowledge message, it will discover that the site has crashed and remove it from the list. If the receiving site crashes before the increment-and-acknowledge messages is sent, the sending site, having the

¹ Messages containing lists of live references.

receiver in its dirty-set, might safely remove it. However, some issues are apparent in these protocols:

- How will the Owner distinguish between a crashed and a temporary communication failure?
- How will it handle detection of crashed sites?

2.3.5 Time Lease

A DGC algorithm using time-outs was first introduced by Hughes [22] in 1985. It relied on assigning time stamps to references. These time stamps were initialized with Lamport's kind of global clock [23]. By using system thresholds and requiring guaranteed message delivery and mutual causal order, the system might distinguish live objects from dead since the time stamps of garbage objects were never increased. These updates were performed by the local GC when it was invoked. It ordered the references to send updates to the Home. After a while the Home can safely remove non-updated objects.

There are hybrid, and non-hybrid, implementations that use the scheme with timeouts, or leases [24]. In this thesis the DGC algorithm is called Time Lease (TL) and a version of the algorithm was implemented to the prototype Component DGC model. Following that prototype, the TL algorithm uses the real wall-time clock representation of time. Instead of propagating the need from Borrowers, two timers are used to keep consistency.

- The timer at the Home identifies the expiration of the lease resulting in reclaiming the space.
- The timer of a Remote is used as an update notifier, when the time on the Owner site is reaching expiration the Remote has to update it. This must be triggered some time before the lease has expired, and the Remote uses a buffer time to handle this. When the update timer expires it sends an update message to the Home and requests more time. The Home adds some time to its expiration timer and message the time (a value) added to the Remote which sets its update notifier to this time minus the buffer. This scenario is depicted in Figure 8.

This scheme allows for fault tolerance since a crashed site will not send an update message, however, as in any pure timeout DGC algorithm, inconsistencies might occur if communication is temporary lost. This must be accounted for with the buffer time.

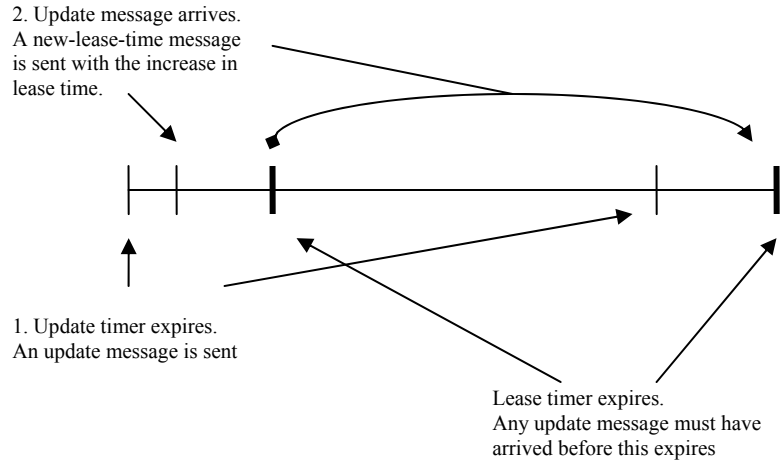


Figure 8. Time Lease update-protocol. When a Borrowers timer signals that an update is needed (1) a message is sent to the Owner. Upon arrival the lease timer is changed to be signaled later and a message with the new time is dispatched to the Borrower. This message contains the length in time to the next lease expiration.

2.4 The Mozart Language

Mozart, an extension to the Oz language which provides for distribution [12], is a multi-paradigm language, it supports imperative as well as declarative programming and it is also object-oriented and supports concurrency through threading. This thesis only contains descriptions of Mozart's basics, for a thorough coverage there is web-based documentation [25], and the Distribution Sub System (DSS) [26], a library module in Mozart.

The Mozart language is mostly developed in the C and C++ languages but some of its packages are written in Oz code, thus the implementations are also realized with combined C++ and Oz code. The Mozart system has a virtual machine which is responsible for parsing the byte code at run-time like java and the JVM [27]. As Mozart offers dynamic typing, type checks are performed during run-time. This is achieved through an *Oz data structure* or *Oz term*, which uses different semantics for different type of entities: lists, records, ports etc. These Semantics are the same when an entity is transformed into a distributed entity and that is achieved through attachment of distribution protocols. Since typing is dynamic using an Oz list in built-in functions might downgrade the system performance.

From the memory management view Mozart offers a tracing local GC which incorporates distributed resources, entities and proxies during the trace. This local collector informs the remote reference-structure if a proxy is garbage and leaves the responsibilities to the DSS.

The important part of Mozart for this thesis is the distribution subsystem and replication of entities.

2.4.1 Replication patterns

In Mozart there are three types of replication patterns for different entity types. Important is whether the entity type is named or not. A named entity

has a GUID whereas an unnamed type is identified through the structure. All stateful entities are named and hence have GUID's when distributed.

There are three types of replication patterns:

- Replicated. Integers and other types not requiring a GUID. These types are copied by structure and rebuilt on the receiving site.
- Replicated uniquely. This type is for entities which should be uniquely replicated, such as large data structures where simple copying would hurt system performance.
- Access structures. Stateful entities which are globalized and have manager and proxies. The GUID assures that only one proxy is built per site.

2.5 The Distribution Sub System

The distribution in Mozart is realized on language entity level, the core engine performs local operations on distributed entities which in turn invokes the Distribution Sub System DSS [28] to carry out the operation. All entities in Mozart have been assigned a distributed behavior and whether they are stateful or not are taken into account. Stateless entities, such as atoms, are merely replicated while stateful entities have to use consistency protocols to keep their state consistent. The DSS has a three-layer design where the top-layer has an interface to the core engine and the lowest layer to the Operating System (OS) as seen in Figure 9 which shows the DSS's relation to the Core engine as well as the layers in the DSS:

- Protocol Layer. Act as the interface to the core engine. This layer is responsible for consistency protocols.
- Communication Layer. Keeps the logical connections to the remote sites and offers message sending functionality. Keeps message queues and monitors the channels state.
- Transportation Layer. Handles the physical communication and marshalling of messages into binary format. Sending and receiving binaries between sites.

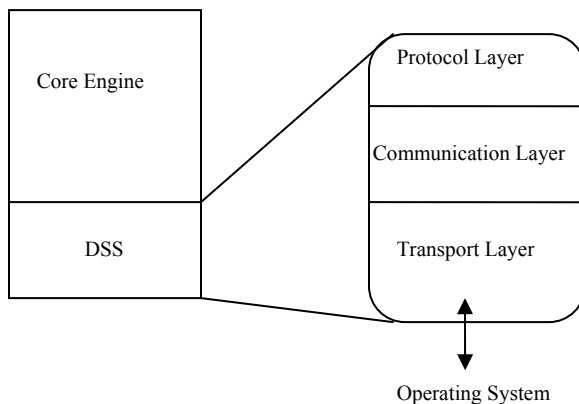


Figure 9. Distribution Sub System layers. The top protocol layer with interface to the Core engine. The next layer handles messages keeps connection information. The bottom transport layer is responsible for the physical connection and communication with the Operating System.

Messages between layers are achieved through message containers which in turn, at the lowest layer allow for serialization of the contained data. The connection between sites is handled through a connection module implemented in Mozart code. As this thesis focus on reference consistency and DGC algorithms the following sections will focus on the Protocol Layer.

The Protocol Layer is responsible for consistency and has five functions.

- To keep track of distributed entities.
- To handle proxies.
- Resolve tasks from the Core engine.
- Keep, and execute, consistency protocols.
- Handle site pointers.

2.5.1 DSites – The Site pointer

References to remote sites are stored as objects called DSites. DSite objects are abstractions of other sites and contain enough information to establish a connection to the remote site and to verify the connection. DSite object does not perform any communication, instead it relies on special Communication objects. A DSite is in one of four states:

- Unconnected, there is no communication need for this site. The DSite can be safely removed (except when DGC algorithms have special needs to retain the object).
- Connected, there are objects which have communication needs.
- Temporary failed. The link to the site is temporary down.
- Permanently failed. The remote peer can never communicate again.

Typically a crashed site.

The uniqueness of the DSite is guaranteed by the pair of a location, address, and a time stamp, which is showed in Figure 10. This scheme assures that even if a new process is started on the same node as a crashed, it can never have the same DSite identifier. Of course a malicious programmer could reset time on the computer but corrections for that is not intended with DSites. To keep DSites up-to-date periodic live messages are sent between the two sites and the state is set thereafter.

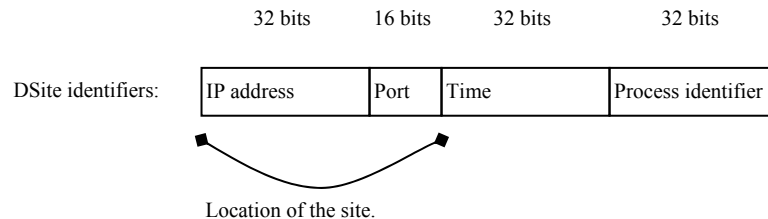


Figure 10. The DSite representation. The two first identifiers are used in the transport layer to enable communication while the two last ensures that the found site is the right.

To simplify and accelerate the lookup of DSites they are hashed over the IP address and port and stored in a DSite table. To avoid memory exhaustion the DSite Table is garbage collected using a mark-scan collector, as all DSites are of the same size the mark-scan technique suits this task very well. When the

system performs a garbage collection all resources which need a particular DSite get the chance to mark it as used. DSites is very important in later chapters.

2.5.2 Distribution nature of entities

The only replication type interesting for the DSS is the access structure described in 2.4.1. This type is divided into three subtypes depending on their kind of state. An entity can for instance have a real state, such as cells with replaceable contents, or be of the single assignment type where assignment means that the entity is transformed into another entity. This is used to implement logic variables. The three types are:

- Stationary State. The manager keeps the state in a centralized server style with proxies sending messages to perform operations. Ports use this protocol.
- Mobile State. A proxy receives the state and can perform operations on the entity as long as not the manager wants the state back. This is used for distributed locks and cells.
- Single Assignment. The manager knows of all proxies and can transform the entity when needed, forwarding the change to all proxies. This protocol is used for logical variables.

2.5.3 Reference consistency

In Mozart are entities either local or global, i.e. distributed. As a global entity can be accessed from other sites it needs an access structure to realize this. Entities of the replication pattern do not need this structure and hence no such is used. In Mozart an entity is *globalized* when it is transferred to another site. Globalize means that that an access structure is built for the entity, a state is assigned and a reference consistency protocol with garbage collection capabilities is assigned. When there are no more references to the entity it is localized, all the structures are removed.

When the entity is globalized a manager structure is created. When a reference is imported to a site a proxy is built there, the proxy act as the entity and can not be distinguished from local entities. Depending on the protocols associated with the type, there might be communications with the manager instance when operations are performed. In Mozart the manager take the place of the entity on the local site, as depicted in Figure 11.

A remote reference uses Owner and Borrow entries to build the link (Figure 11), an Owner entry at the exporting side and a Borrower entry at each importing side. This link is used for message sending. They also act as roots to the local garbage collector to keep the manager and proxies alive.

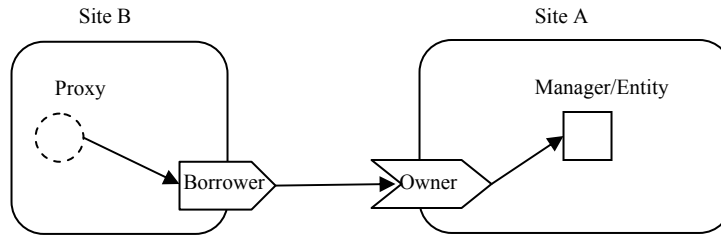


Figure 11. A remote reference in Mozart. The entity is wrapped within a manager structure when transformed into a distributed entity. This manager structure is paired with an Owner object responsible for the link from referring Borrowers. The Borrower, forming a link on the remote side is paired with a proxy-structure responsible for carrying out operations on the reference by sending messages over the link.

An Owner is alive if the DGC algorithm knows of any references and the Borrower will be kept alive if there exist a reference or if the DGC algorithm defines it to be a root, i.e. it has a dirty-set. Both the Owner and the Borrower uses a reference consistency object to handle distributed garbage collection algorithms in the prototype component based schema. There is an invariant saying that there might only be exactly one Borrower entry on a site, to a specific reference. This invariant is upheld by the use of GUID's.

To handle and provide with fast access to Owners and Borrowers they are kept in the Owner Table (OT) and the Borrower Table (BT). The OT associates every manager (entry in the table) with an index value, the OTI. The BT's entries are managed in a similar schema but it combines the DSite, the location of the entity, with the OTI to form a *netaddress*. This netaddress is the GUID for the entity. To send a message to an entity, a Borrower creates a message to the DSite-location and sends it to the Communication Layer for further transportation. At the Owner site the message engine, the communication layer, uses the OTI sent along with the message to look up the right entity.

3 Methods

The chosen model for reference consistency in Mozart is a component based model which incorporates a few of the basic DGC algorithms described in chapter 2.3. How the component based model is defined and implemented is described in this chapter as well as the protocols allowing programmers to both instrument entities, in an application, with specific DGC algorithms, as well as write their own DGC algorithm and add it to the Mozart language.

The single most important task when designing the general interface for DGC algorithms is to identify and clearly distinguish between the distribute mutator operations performed by the system.

3.1 Identified Mutator Operations

In this thesis we define six distributed mutator operations which could be performed on an entity, in a distributed system. These definitions are an extension of the previous three defined by Lermen and Maurer [17]:

1. Creation of a reference. A site where an object is local passes a reference to another site.
2. Duplication of a reference. A site with a reference to an object passes the reference to another site, not being the Owner site. This operation does not involve the Owner.
3. Deletion of a reference. A site holding a reference to an object discards the reference.

The new model accounts for state of the receiving site, and introduces a new operation. These operations are the foundation for designing the interface to the DGC algorithms even though there might not be a distinction between the operations for some of the algorithms. The mutator operations should be kept in mind when reading the rest of the chapter as they are involved in all consistency protocols and they simplify the development and implementation considerably. All the operations are associated with a number for simplification when later involved. The mutator operations are:

1. Sending a reference for an object from the Owner site to another site which does not have the reference, thus becoming a Borrower by creating the access structure and associating the reference with state protocols. This operation is depicted in Figure 12 as number one.
2. Sending a reference for an object from the Owner site to another site which already has the reference, thus no access structure actions need to be taken, only updating the reference consistency protocol. This operation is depicted in Figure 12 as number two.
3. Sending a reference for an object from a Borrower site to another site which does not have the reference. The receiving site will become a Borrower by creating the access structure and associating the reference with state. This operation is depicted in Figure 12 as number three.
4. Sending a reference for an object from a Borrower site to another site which already has the reference yet is not the Owner, thus no access

structure actions need to be taken, only updating the reference consistency protocol. This operation is depicted in Figure 12 as number four.

5. Sending a reference from a Borrower to an Owner. This operation does not contain any useful consistency information and no access structure needs to be created, hence this operation is not accounted for when the algorithm protocols are defined. This operation is depicted in Figure 12 as number five.
6. Deletion of a reference on a Borrower site. This operation is performed when no reference to an entity is found on the Borrower site, resulting in a deletion of the access structure. This operation is depicted in Figure 12 as number six.

These operations are also summarized Table 1 below, along with the number:

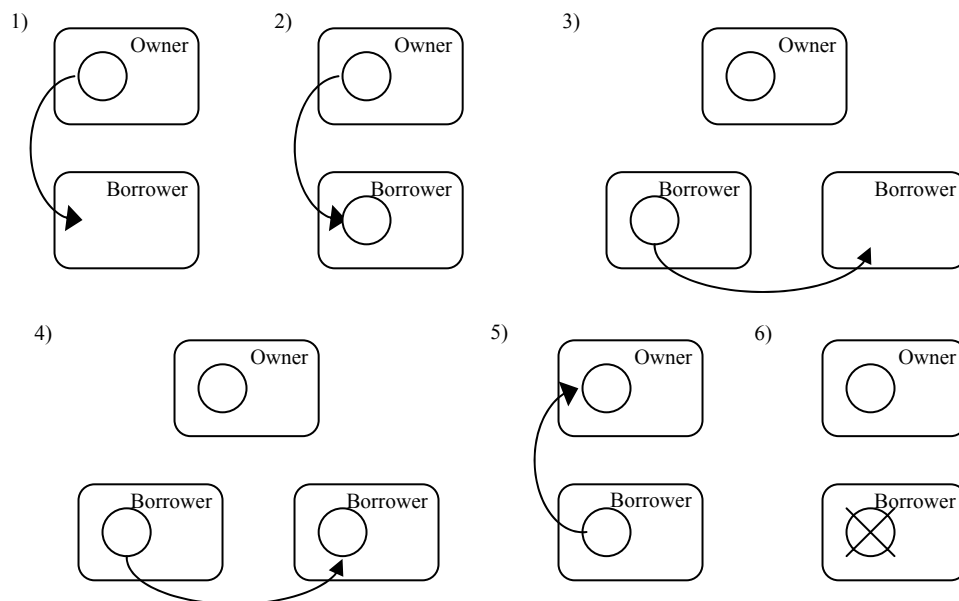


Figure 12. Distributed mutator operations. 1) Passing a reference from Owner to another site which does not have it. 2) Passing a reference from Owner to another site which already has it. 3) Passing a reference from a Borrower to another site which does not have it. 4) Passing a reference from a Borrower to another Borrower already having it. 5) Passing of a reference from a Borrower to the Owner. 6) Deletion of a reference for a Borrower.

The sixth operation, *delete*, performed at the borrower side, is executed when the local GC realize that the entity is no longer needed and notice the DSS about this condition. Whatever action is taken by the DSS is solely depending of the chosen DGC algorithm(s).

It is important to know that neither the reference consistency protocol nor the algorithms at the Owner side might distinguish between, for instance, operation one or two. Some algorithms do have this information but it is only at the Borrower side the information is completely known.

No.	Sender	Receiver	Existence of entity
1	Owner	Borrower	Unique
2	Owner	Borrower	Existing
3	Borrower	Borrower	Unique
4	Borrower	Borrower	Existing
5	<i>Borrower</i>	<i>Owner</i>	<i>Existing</i>

No.	Name
6	Delete

Table 1. Summary of distributed mutator operations.

From now the mutator operations will be referred to by their number.

3.2 Component Based Model

The component based DGC scheme is an approach for the reference consistency problem proposed by the Klintskog et al. [16]. The idea for a component DGC scheme is that applications have different type of distribution behavior, which makes different DGC algorithm and sometimes compositions of DGC algorithms, a more attractive solution than the tedious work of composing DGC algorithms through static merging and using a single hybrid DGC algorithm for all entities.

The composition of DGC algorithms, which works independently of each other, allows the system to benefit from the strength of each DGC algorithm. A reference counting algorithm, for instance, allows for fast detection and reclaiming of space, while not being able to cope with failures (or cycles). Combined with a timed algorithm which has slow detection but handles failure, the system will have fast reclaiming of ordinary garbage and slow detection of garbage from failed processes.

The component model gives the ability to alter set of algorithms in runtime. To uphold consistency there can only be removal of DGC algorithms after globalization, which is obvious since adding another DGC algorithm might lead to prematurely removal. For instance, adding another counting DGC algorithm after passing a reference to other sites leads to a system with only a sub set of the referring sites being counted in the new DGC algorithm. When the subset deletes their references the new DGC algorithm will conclude that it is safe to reclaim the entity.

This altering of the current set can only be initiated at the home, to uphold semantics, although a remote process can order the removal to the home.

3.2.1 DGC algorithm configuration

The component model defines that when an entity is globalized, a Home Reference is constructed for that entity. It is then assigned a set of DGC algorithms, DGC instances or DGCIs. When a reference to the entity is passed to another process, a DGCIs create a Remote Reference instance (RR instance), which formalize information of the type of DGC algorithm as well

as information for creating a DGCI on the other node, explained in section 3.3.

At the point of globalization, when the basic Home Reference structure has been created according to the current settings for DGCI's to be used with entities.

3.2.2 Set differences

When a reference is received for a unique entity, mutator operation one and three in Table 1, the system instantiate all the remote algorithms passed with the entity. If the entity existed, the RR instances are merged into the current DGCI's. However, when the set of DGCI's is larger than the received, the merging method removes the DGCI through the general interface. The removal of a DGCI is not the same as when a delete of the entity is performed since no DGC algorithm action should be executed. The other case is when the received set is larger than the current set, in that case the sender will be notified about the condition for it to remove the DGCI.

3.3 *Serialization of DGC Instances*

When passing a reference to another site, enough information must be sent to the other site for it to be able to create the access structure, identifying the entity type and the set protocol as well as the DGC algorithms associated with the entity. This is called serialization¹, or marshaling, and serialization of DGCI's is implemented through the RR instance objects and every DGC algorithm defines its own type specific RR instance. The RR instance is created when the core engine initiate a passing of an entity. When asking the DGCI's, through the reference consistency protocol, for information of how to create DGCI's on the receiving side, a RR instance is created with:

- Type tag; a unique integer which identify the package with a specific DGC algorithm.
- DGC algorithm specific information; constants, values or DSites

These packages are stored for later sending when the creation is completed. However, sometimes the passing of a reference is revoked, in that case the packages are returned. Methods supporting this must be implemented into the reference consistency protocol.

Before the sending may start the RR instances have to be converted to binary format. This is accomplished by invoking the marshaling method, Table 2, associated with the RR instance, which makes every RR instance save the information to a byte stream.

When the receiving site creates the proxy for an entity, it passes its received byte stream to a DGCI creation method. This method checks the byte stream

¹ Serialization means that state information, and sometimes structure information, is extracted from an object into a format suitable for sending over a communication channel or saving to a stable storage.

for RR instance tags and invokes the RR instances unmarshaling methods, Table 2, and afterwards creates a DGCI from every found RR instance.

All DGC algorithms must implement their own type of RR instance with contain its specific information.

Name	Task
marshal_RR	Marshaling type specific DGCI information to a byte stream.
unmarshal_RR	Unmarshaling of the type specific information from a byte stream.

Table 2. RR instance general methods.

3.4 Two-Layer Messages

For the general interface to be extendable the protocol messages had to be designed so all kind of messages could be sent, independently of the kind of DGC algorithm chosen.

The solution was a two-layer message protocol which reduced the number of specified message types, for DGC algorithms, in the system to just two (Figure 13). Although one message type would have been enough, there is a clear distinction between managers and proxies and that distinction was kept in the message types for practical reasons.

A typical message is defined as the following two levels:

1. For the message engine: Tag for the message (integer); address to reference/entity, a locator to either OT or BT; tag for reference consistency protocol (integer) to point out the DGC algorithm;
2. Second layer: Message in form of a DGC algorithm specific format¹, which allows for undefined sizes. However performance drops would be expected due to the requirement of undefined sizes being met.

To Owners:

M_HOME_DGCI	OT Index	Algorithm type	Message
-------------	----------	----------------	---------

To Borrowers:

M_REMOTE_DGCI	BT Netaddress	Algorithm type	Message
---------------	---------------	----------------	---------

Figure 13. DGC messages. The message formats differs in the composition of the look up address as Owners only needs the index information, site info is obsolete.

In addition to aforementioned definition, information about the sender site is also passed to the reference consistency protocol which in turn passes it to the DGC algorithm. This information, however, is retrieved from the communication layer which reduces transportation costs.

¹ Currently an Oz data structure of the list type. The decision to use the Oz data structure was that no further marshaling routines had to be developed.

The second layer shown in Figure 14 has no strict definition, the proposed protocol is simple:

- ID tag (integer) for the type, ACK, UPDATE etc.
- Container of variable size, for any kind of information such as time stamps, site info or plain integers. Size might be implicitly known from the tag or through the DGC algorithm format, or explicitly stated within the container.

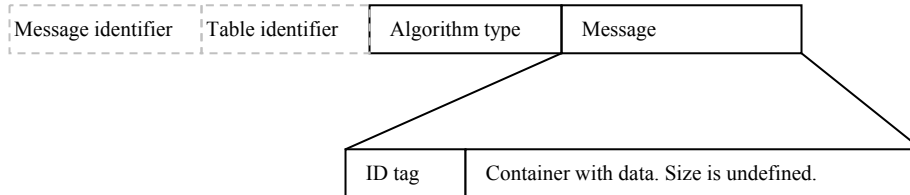


Figure 14. Proposed DGC algorithm message container. A message to either a Home DGCI or Remote DGCI

The second layer protocol is just a proposal, thus optimizations might be performed if few message types are present, leading to a removal of the ID tag if it is considered obsolete.

3.5 Failure Handling

A number of failure events can occur in a distributed system and also because of the component based structure, chosen for Mozart. The fault handling protocol focus on faults for DGC algorithm messages, i.e. non-deliverable DGC algorithm messages, this covers removed DGCI's and prematurely removed entities/proxies (Owner and Borrower entries). Thus failure with crashed sites, consistency or the DGC algorithms actions for failed messages are not handled since that is up to the DGC algorithms themselves, the protocol is a framework informing the system and references about failures occurred. The first issue is to determine which failures could occur within the reference consistency protocol and the above described two-layer message model. Since all messages from the remote consistency protocol follow the same specifications, special DGC algorithm considerations are avoided.

The obvious types of failures that might occur are of two types:

- Entity no longer present; a message to an entity which has been removed. This is not a true error for proxies because of the component based schema where entities might be removed due to one but no other DGC algorithm finding it to be garbage. When entities at managers are prematurely removed this will become a severe failure.
- Algorithm removed; a message to an entity which exists but no longer has this algorithm in its set. This is a less severe error but needs to be address to avoid excessive.

With the above failures, the first error-messages found in Table 3 can be identified:

Failure description	Associated message (tag)
Entity removed at Home	M_HOME_ENTITY_FAILED
Entity removed at Remote	M_REMOTE_ENTITY_REMOVED
Algorithm removed	M_ALGORITHM_REMOVED

Table 3. DGC failure messages.

The specification for these messages, found in Table 4, and their data types follows the same scheme as for the Two-layer message model with tags consisting of integers, sites as DSites and so on.

Sent to	Tag	Lookup info.	Tag	Message
Remote site	M_HOME_ENTITY_FAILED	OTI	-	-
Remote site	M_ALGORITHM_REMOVED	Netaddress	ALG	-
Home/Remote	M_REMOTE_ENTITY_REMOVED	Netaddress	ALG	MSG

Table 4. DGC failure messages and their specification.

Further there must be considerations for the case when the system enters a state where the above messages can not be delivered, i.e. when there are problems with the correct fail and non-fail cases for messages in the system. Those correct message situations can be found in Table 5.

No	At site	Received message type	Action
1	Home	M_HOME_DGCI	Pass message to algorithm
2	Remote	M_REMOTE_DGCI	Pass message to algorithm
3	Remote	M_HOME_ENTITY_FAILED	Fail entity. Invoke failure protocol.
4	Remote	M_ALGORITHM_REMOVED	Remove algorithm from the entity's set
5	Home	M_REMOTE_ENTITY_REMOVED	Pass returned message to algorithm.
6	Remote	M_REMOTE_ENTITY_REMOVED	Pass returned message to algorithm.

Table 5. Complete Reference Consistency message table.

From this the complete picture of erroneous cases evolves. A protocol for handling errors with delivery of failure messages is found in Appendix B, which is complete in the sense that no failures might yield a state of passing error messages back and forth.

3.6 Design of a General Interface for DGC Algorithms

The design for the interface focus on the distinction between managers and proxies, this is followed throughout the whole model. There are two levels:

Reference consistency level:	Home Reference	Remote Reference
DGC algorithm level:	Home DGCI	Remote DGCI

3.6.1 Reference consistency level

In the general interface model, implemented for Mozart, the reference consistency protocol partially works as a front-end for DGC algorithms, hiding them for the Owner/Borrow tables, and partially with handling and managing DGCI's and their communication. The protocol defines the behavior for actions between two sites when passing/deleting a reference as well as how messages are sent between DGCI's. It has two parts: Home and Remote, as shown in Figure 15 where the Home and Remote part each has two DGCI's connected to them. The main purpose is to keep the set of DGCI's and also hold information about the entity's address i.e. the table entry. The tasks performed at the reference consistency level are:

- Change the set current of algorithms associated with the entity.
- Identify when the entity has become garbage, through the DGCI's.
- Perform the distributed mutator operations.
- Implement support for reversal of sending a reference. Sometimes the connection to a site is lost, thus the system is unable to send RR instances. In that case those RR instances have to be revoked.
- Pass algorithm messages.
- Implement simplified message sending capabilities for the algorithms which handle wrapping of identification.
- Gather information from, and about, the algorithms to be presented for the application layer.

If no DGC algorithms are associated with an entity that indicates that the entity is persistent, i.e. there are no DGC algorithms to tell whether or not there are references to the entity. This schema with no DGC algorithm to identify a persistent entity has several advantages: The component model only allows removal of DGCI's to be semantically correct and this implies that either the persistent algorithm is realized as other DGC algorithms with the same interface, thus always being used. The other approach, which is taken in this project, is that persistence is realized as no algorithm, reducing the system effort to handle DGC algorithms while still upholding semantics. The methods to handle persistence is reduced to only check if there are other algorithms in the current set on the both the Home side and the Remote side. However, on the Remote side there is no significant use of the knowledge of persistence, other than the handling of DGCI's that is removed.

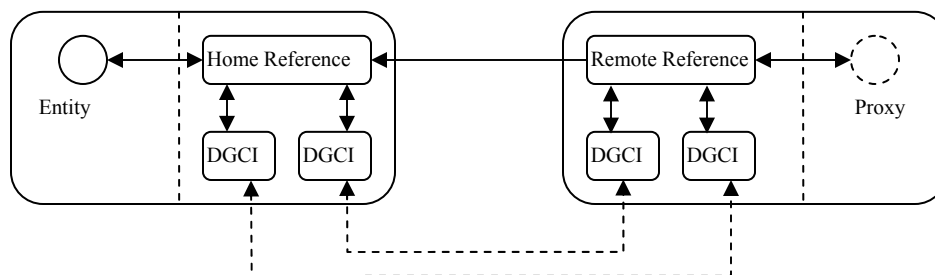


Figure 15. Distributed entity with two DGC algorithms. As seen in the figure, DGCI's might know about corresponding DGCI's at other sites, while at the reference level there is only one-way knowledge.

Both the Home and the Remote part of the reference consistency protocol has the same type of methods offering ability to configure the set of DGC algorithms, perform mutator operations and respond to local garbage collection actions and finally special methods to extract information about the state and methods invoked when local GC is performed allowing for preparations to secure local resources.

Although many of the RR's methods are similar to the HR's there is a significant difference in that the application layer can neither control DGC algorithm settings nor which DGC algorithm to use, this is only done at the home, apart from that the Remote side is active based on the local GC.

The state information kept within the Home Reference is: Set of current algorithms and index to the corresponding owner table entry (OTI). The Home Reference (HR) reacts to messages and timers for localizing an entity. In The tables below OT and BT means the Owner and Borrower tables described in section 2.5.3. The HR's configuration methods, found in Table 6, are:

Name	IF to	Task
canBeReclaimed	OT	Returns a Boolean which tells if any DGCI consider the entity to be garbage.
isPersistent	OT	Returns a Boolean for the persistent algorithm.
makePersistent	OT	Configure the reference to use the persistent algorithm which makes entities persistent at their home. Accomplished by clearing out all DGCI's.
setup	OT	Set up the HR for a new entity. This action asks the DGC Configuration unit for currently activated DGC algorithms and instantiate them. Invoked as a part of the access structure creation.
removeAlgs	OT	Method which is invoked when the runtime system performs a localizing. Removes all DGCI's.
removeAlgorithm	OT	The application layer or a failure message ordered that one of the DGC algorithms were obsolete and therefore the DGCI should be removed from the current set.

Table 6. Home Reference configuration methods.

Table 7 shows the methods involved with the mutator operations:

Name	IF to	Task
getReference	OT	Ask the algorithm for RR instances to pass to another site. Information about the receiving site is passed to the method. Method invokes similar method for each DGCI. This method is invoked when passing a reference. Mutator operation one and two, Table 1.
mergeBackRef	OT	This method returns the RR instance extracted with getReference for the DGCI's to "merge back", i.e. restore their state as if the getReference action which rendered the specific RR instances was not performed.
removeReference	OT	Restore the memory used by the DGC instances to the system and reset look-up information.
passCtlMsg	OT	The message engine has a message to one of the DGCI's.
passFailedMsg	OT	The message engine has a message that failed to deliver because the remote proxy has been removed. See Failure Handling, section 3.5

Table 7. Home Reference mutator operation methods.

Other Home Reference methods are found in Table 8, those methods are used for special non-mutator or configuration tasks:

Name	IF to	Task
extract_info	OT	Invoked when the application layer wants to gather information about the DGCI's in the current set. Returns a textual representation about the state.
makeGCpreps	OT	When the runtime system wants to perform a GC all DGCI's are noticed so they might secure resources such as DSite pointers.
sendRemote	DGCI	Method invoked by a DGCI when it wants to send a message to a Borrower site.

Table 8. Home Reference special methods.

As with the HR, The Remote Reference (RR) contains two kinds of state information, the current algorithm set and the netaddress, the index corresponding to the OTI at the Home along with the DSite. The RR is instantiated when a reference is received, bringing the RR instances. They define the current set when de-serialized, at the creation of the RR. The remote reference works as a reactive protocol to the virtual machine, as opposed to the HR. The protocol is invoked when the system concludes,

through the local collector, that the associated proxy is garbage or when a message to an algorithm is received.

The configuration methods implemented for the RR (found in Table 9) are:

Name	IF to	Task
canBeReclaimed	BT	Sometimes DGCI has a dirty-set which prevent the local garbage collector from removing the reference.
removeAlgs	BT	Works the same way as its counterpart for HR's.
removeAlgorithm	BT	A specific algorithm should no longer be in use. This is either implicitly known, when receiving an existing entity or through the special <i>removal of algorithm</i> -message.

Table 9. Remote Reference configuration methods.

The Remote Reference methods for mutator operations, Table 10, are:

Name	IF to	Task
dropReference	BT	The system has concluded that the reference is garbage and this method is invoked before reclaiming the space to let the DGCI's perform removal operations such as sending messages. This method is invoked when mutator operation six is performed, Table 1.
getReference	BT	Works the same way as its counterpart for HR's.
mergeReference	BT	This method is invoked when a reference is received for an existing entity. It merges RR instances into the current DGCI's and removes obsolete RR instances. This method is invoked in mutator operation two and four, passing of a reference to a Borrower already having the reference (Table 1).
mergeBackRef	BT	Works the same way as its counterpart for HR's.
setUp	BT	Instantiate the RR for a new entity and set up the DGCI's according to the attached RR instances. This is the mutator operations one and three in Table 1.
passCtlMsg	BT	Works the same way as its counterpart for HR's.
passFailedMsg	BT	Works the same way as its counterpart for HR's.

Table 10. Remote Reference mutator operation methods.

Table 11 shows the special Remote Reference methods:

Name	IF. To	Task
extract_info	BT	Works the same way as its counterpart for HR's.
makeGCpreps	BT	Works the same way as its counterpart for HR's.
sendHome	DGCI	Method invoked when a DGCI wants to send a message to a DGCI at the home.
sendRemote	DGCI	Method invoked when a DGCI wants to send a message to a DGCI at another Borrower site.

Table 11. Remote Reference special methods.

The distributed mutator operations are realized through mapping of mutator operation, Table 1, to reference consistency level methods and can be found in Table 12. This mapping does not contain any DGC algorithm specific messages, even though there might be messages as a result of a specific operation.

No	Sending methods	Receiving methods
1	HR invokes getReference method.	RR invokes setUp to set up DGCI's.
2	HR invokes getReference method.	RR invokes mergeReference method.
3	RR invokes getReference method.	RR invokes setUp to set up DGCI's.
4	RR invokes getReference method.	RR invokes mergeReference method.
6	-	RR invokes dropReference.

Table 12. Reference consistency protocol, operation-to-method mapping.

3.6.2 DGC algorithm level

At the algorithm level, the protocol defines a basic interface to the reference consistency level, which allows for mutator operations and message receipt. The basic algorithm state information is the type information, identifying the algorithm with integer tags through a static mapping in a shared namespace, and pointer to the reference consistency protocol for invoking message sending methods and retrieving Owner/Borrower table information. Every algorithm has its own state information and might have special methods invoked by other resources, but all must implement the basic methods. As with the Reference Consistency level there are two parts of an algorithm: Home and Remote.

The Home DGCI is the basis for a DGC instance at the Owner side. It implements methods for mutator operations one and two, Table 1, and message handling along with state retrieval for presentation, which may be requested from the application layer. The basic Home DGCI interface has the following, Table 13, methods against the reference consistency level.

Name	Task
isGarbage	DGCI tells whether it is garbage or not.

getReference	Create a RR instance which contains data to set up the DGCI on the receiving side. Information about the destination is passed to the method, responsible for mutator operations one and two in Table 1.
mergeBackReference	Method invoked when to revoking a pass of a reference. This method is invoked with the RR instances not sent.
getCtlMsg	Message receipt.
getFailedMsg	Message receipt of not delivered messages.
makeGCpreps	Invoked before reclaiming of space. DGC Algorithms might have special arrangements that need to be taken care of.
extract_info	This method is invoked when retrieving the state. Returns a record with textual state information.

Table 13. Home DGCI methods.

The Remote DGCI is the counterpart of the Home DGCI. It defines how the DGC instance at the Borrower side should work, according to the algorithm protocol. It implements methods for mutator operations three, four and six (delete) in Table 1, and message handling along with state retrieval for presentation. As this is a protocol reactive to the Remote Reference it implements methods for deleting the reference as well as determine if a DGC instance is a temporary root (has a dirty-set).

The basic Remote DGCI interface has the following methods (Table 14):

Name	Task
isRoot	Ask the DGCI if it is a root, i.e. if it has a dirty-set.
dropReference	Tells the DGCI that it should invoke its method for deletion of the reference.
getReference	Works the same way as for the Home DGCI except that it is responsible for mutator operations three and four as the sending side, Table 1.
mergeReference	Pass another RR instance to the DGCI. Invoked when already instantiated one DGCI of the type.
mergeBackReference	Works the same way as for the Home DGCI.
getCtlMsg	Message receipt.
getFailedMsg	Message receipt of not delivered messages.
makeGCpreps	Works the same way as for the Home DGCI.
extract_info	Works the same way as for the Home DGCI.

Table 14. Remote DGCI methods.

3.6.3 A typical message-passing scenario

With this schema an example message-passing scenario, shown in Figure 16, would look as follows: When a DGC instance realize it needs to send a message to another DGC instance it first creates the message, in form of an Oz term, which is passed (1) along with destination (i.e. a site) specifics and the type to the reference consistency protocol, according to the *sendHome/sendRemote* method. (2) The reference consistency protocol wraps this message with entity and destination information and (3) leaves a message container to the message engine. (4) On the receiving site, when its message engine has received the message container, the wrapped message is presented (5) along with site information about the sender, for the reference consistency protocol using the *passCtlMsg* method. The reference consistency protocol checks if the algorithm is in the current set, and if so, (6) pass it to the right DGC instance. If the algorithm were not in the current set an error message would have been dispatched to notify the initiating side, see Failure handling 3.5 for complete specification.

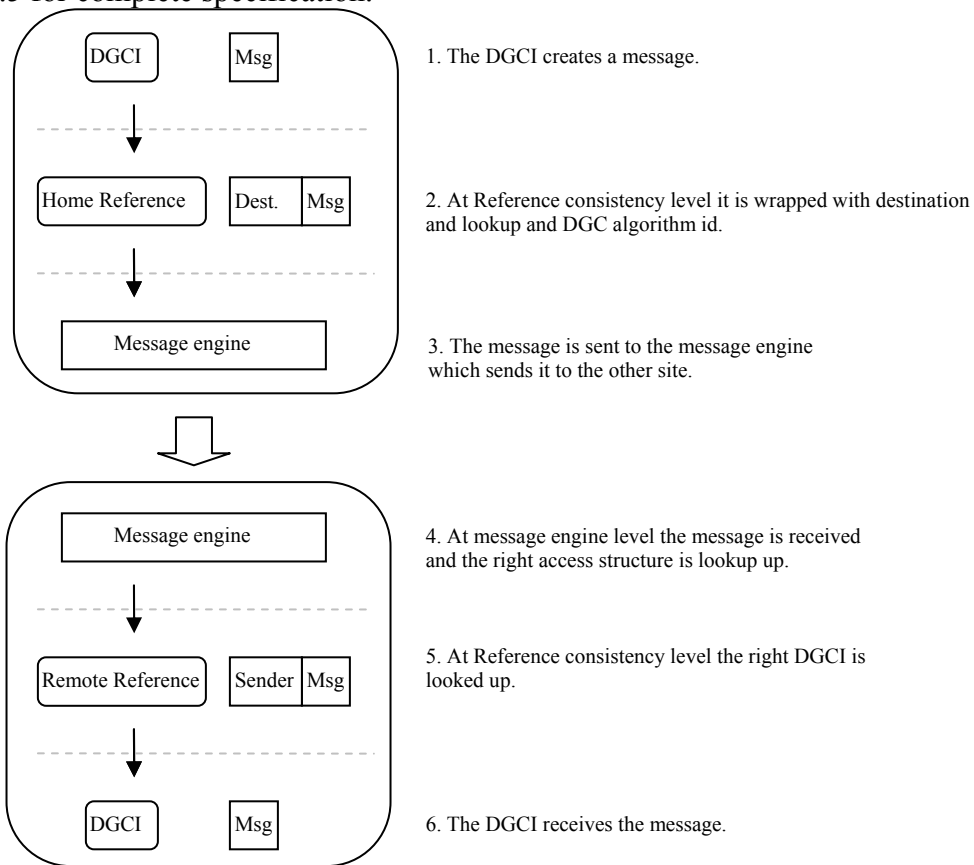


Figure 16. An example message scenario. 1) A DGC instance creates a message. 2) The reference consistency level is presented with the message information to it. 3) The message is sent from the message engine 4) and received on the other side, before presented to the 5) reference consistency level which unwraps and 6) deliver the message.

3.7 Instrumenting DGC algorithms and Instances

A key part of the component model is the instrumentation of DGC algorithms and the DGCI for unique entities. When a programmer is creating an application, he might know more about the behavior of the distribute entities, than the system in run-time. The programmer might develop own DGC algorithms and adapt them to the environment of his special application. An example is that one type of entity might be short-lived and spread mainly to the close proximity of the Owner. Such an entity is not suited to pair with a normal timed DGC algorithm since that kind of DGC algorithm would keep many entities alive for a long time, resulting in poor memory utilization. Rather one would want to pair that kind of entities with a fast reclaiming DGC algorithm which favors less overhead (in terms of added records, lists, etc.) and few messages.

The implemented ability to instrument both DGC algorithms in general and DGCI's comes in two parts:

- An interface to the application layer in form of built-in functions.
- An interface to the DGCI's and core engine.

The interfaces are partly maintained through the centralized DGC Configuration unit and partly from direct operations performed on the Owner and Borrower tables described in section 2.5.3.

3.7.1 The DGC Configuration unit

The Configuration for current properties and current DGC algorithms are stored within the DGC Configuration unit. It offers capabilities of converting between system internal type representation and textual representation for the application layer. When adding new DGC algorithms, programmers need to update this unit with:

- A new DGC algorithm tag and its textual representation, its name.
- The specific settings and their textual representation.
- Updating the marshaling methods for RR instances from section 3.3.

The methods reachable from the application layer associated with the DGC Configuration unit are found in Table 15. All the methods which require arguments are invoked with an Oz term consisting of the textual representation of the DGC algorithm (as stated above) and textual representations of property names and values, or simple Booleans. There is a simple failure protocol which states that any attempt to access not existing DGC algorithms results in canceling of the operation. The programmer is assumed to know what he/she is doing so no extensive error messages are returned except a warning: "*Algorithm not found, getAlgFromOZ_Id*" and the operation causing the error is simply canceled.

If the programmer uses the ordinary interface, described later in 3.7.2, the built-in wrappers handle erroneous calls and avoid system halt.

Name	Task
setAlg	Inform the DGC Configuration unit about the default usage of a particular DGC algorithm. The method is invoked with two arguments: the name and a Boolean.
setAlgProps	Set default properties for a specific DGC algorithm. Invoked with name of the DGC algorithm, name of the property and a value.
getAlgs	Returns an Oz record with DGC algorithms and their usage.
getAlgProps	Returns an Oz record, given a specific DGC algorithm, with properties of the DGC algorithm and their values.

Table 15. DGC configuration methods for the application layer.

3.7.2 Interface to the application layer

The methods found in Table 15 are all reachable from the application layer through built-in functions, which just opens up a pipe to the DGC Configuration unit. Along with these built-ins come other built-ins which operate directly on the Owner and Borrower tables. Those other built-ins allows for removal of DGCI's for a specific entity, as well as retrieving and presenting information about the entities found in the two tables. The latter information is gathered through the *extract_info* methods (section 3.6.1).

To ensure safety however, special functions are implemented to the Mozart virtual machine, which handles failures due to malformed arguments or spelling mistakes. These special functions are found in the DPControl- and DPStatistics-packages, which also generate information about the total number of mutator operations (Table 1). As stated in section 3.1, there is no way to distinguish, on the Owner side, between the types of operation. To get a complete picture one has to gather all sites statistics. With this statistics/knowledge the programmer can tune the application for improved performance with respect to memory utilization and/or reducing the number of messages for entity operations.

The functions available from the command line or in application are shown in Table 16 which also states their return codes:

Name	Task
SetDGCAlg	Interface to setAlg in Table 15. If the DGC algorithm does not exist, the return code is <i>algorithm_not_found</i> else <i>done</i> .
SetDGCAlgProp	Interface to setAlg in Table 15. Method has the same return-codes as above with the addition of <i>property_not_found</i> if the property name does not exist.
GetDGCAlgs	Interface to getAlgs in Table 15.

GetDGCAlgInfo	Interface to getAlgProps in Table 15. If the DGC algorithm does not exist <i>algorithm_not_found</i> is returned, else the Oz record stated for getAlgProps.
GetDGC	Returns the DGC algorithms associated with an entity
SetDGC	Sets the DGC algorithms to use with an entity, according to the lattice structure mentioned in section 3.2.
OperCounter	Returns a record with the number of distinguishable operations performed since the Distribution Sub System started.

Table 16. Command line functions for DGC configuration.

A typical command line argument for changing default DGC algorithm *AlgName* to be used is: {DPControl.setDGCAlg *AlgName* true}
This results in a “done” if the DGC algorithm existed.

3.7.3 Interface to the Distribution Sub System

The interface to the DSS is simple and handles three types of tasks:

- Creating DGCI’s for the Home References at globalization.
- Creating DGCI’s for the Remote References from received binary representations of RR instances.
- Hold current settings for DGC algorithms which are accessed at creation of Home DGCI’s and when some operation is performed.

The two first tasks, creations, are implemented as methods while default values are attributes of the DGC Configuration unit at run-time. The methods are listed in Table 17. There are two RR instance methods as RR instances only needs to be transformed into DGCI’s at the set up of a Remote Reference, they are kept as RR instances if they are to be merged.

Name	Task
getGCAlgorithmsInUse	Creates DGCI’s from DGC algorithms that are stated to be used for the Home References. Those DGCI’s will be the current set.
getRRinstancesFromBuf	Returns the de-serialized RR instances from a byte stream.
getGCAlgorithmsFromRRinstances	Creates DGCI’s from RR instances for the Remote Reference.

Table 17. Reference Consistency: protocols methods for DGCI’s

3.8 Implemented DGC Algorithms

The DGC algorithms which were implemented in this Master’s project are of the reference counting and timed types and are summarized in Table 18. Before the actual implementation were full state diagrams for the mutator operations, see Appendix C, created to ensure consistency all the time during

normal operations. Since the underlying messaging service guarantees delivery and FIFO this was left from the implementation. However, causal ordering between several sites are not upheld by the Distribution Sub System so all mutator operation-cases has to have this in consideration. Further the reliability of the dirty-set, when such were used, must be upheld, otherwise leaving the systems with inconsistencies.

The implemented DGC algorithms were the following:

Abb.	Type	Short description
RC	Reference Counting	Keeps a counter on the Home site with the number of references to the entity.
IRC	Reference Counting	Keeps a counter on each node which passes a reference.
FWRC	Reference Counting	Starts with a weight on the Home which is split and distributed over the nodes when a reference is passed.
RL1	Reference Counting	Same as RC but keeps a list instead of the counter.
RL2	Reference Counting	Same as above but uses a different protocol for keeping consistency.
TL	Timed	Keeps a timer for the <i>lease</i> of an entity. All nodes with a reference to this have to notify the Home in regular intervals.

Table 18. Implemented DGC algorithms.

The chosen DGC algorithms are the algorithms most common in research papers and are considered both useful for testing the general implementation and would suit the needs of the Mozart language.

In the descriptions, the process of creating access structures and deserializing of DGCI's is left out since this section focus on the operations, the messages and the attributes (counters, timers, lists etc.). Home is the DGCI at the manager site and Remote (X) is a DGCI at a proxy site. Attributes are found, and messages are received, at a Home (H) or a Remote (R). All events are listed in the order they occur, matching a mutator operation. An event, for instance:

“X – Home sends a reference and increases the counter” is semantically atomic. After a *send* event the next event is assumed to be a *receive* event.

3.8.1 Reference Counting

The distributed Reference Counting (RC), based on the Lermen and Maurer scheme [17], is described in Section 2.3.1.

The implementation of this DGC algorithm is straightforward, matching the identified mutator operations and considering the causal order problem. Some optimizations were carried out which reduces the number of messages in the cases where the entity already existed on the node. The attributes and messages implemented for RC can be found in Table 19.

Attributes

	Name	Type	Purpose
H	counter	Counter	The number of known references to the entity.
R	unacked	Counter	The number of passed reference for which no ACK message have been received. If this is greater than zero the proxy is a root.
R	decs	Counter	How many decrements to be made at the Home, i.e. how many times the node have received the reference from the Home.

Messages received

	Message tag	Purpose
H	GC_RC_DROP	A referring node has deleted its proxy and returns <i>decs</i> many references.
H	GC_RC_INC_AND_ACK	Notifies the Home that operation three was done. Home increments <i>counter</i> .
R	GC_RC_ACK	The remote node receives an ACK for a passed reference. This decreases the <i>unacked</i> .

Table 19. Reference Counting attributes and messages

The implementation follows the interface proposed in Section 3.6.2 for the Home and Remote parts. A figure with the mutator operations and their events can be found in Appendix C and in Table 20 below:

No	Events
1	1 – Home sends a reference and increases <i>counter</i> . 2 – Remote instantiate the DGCI. <i>decs</i> = 1, <i>unacked</i> = 0.
2	1 – Home sends a reference and increase <i>counter</i> . 2 – Remote merges the DGCI and increase <i>decs</i> .
3	1 – Remote 1 sends a reference and increase <i>unacked</i> . 2 – Remote 2 instantiate the DGCI. <i>decs</i> = 1 <i>unacked</i> = 0. Further it sends a GC_RC_INC_AND_ACK to the Home. 3 – Home increases <i>counter</i> and sends a GC_RC_ACK to Remote 1. 4 – Remote 1 decreases <i>unacked</i> .
4	1 – Remote 1 sends a reference and increase <i>unacked</i> . 2 – Remote 2 merges the DGCI and sends a GC_RC_ACK to Remote 1. 3 – Remote 1 decreases <i>unacked</i> .
6	1 – Remote sends a GC_RC_DROP to Home. 2 – Home decreases <i>counter</i> with <i>decs</i> .

Table 20. Reference Counting: The operations. The events for every mutator operation (No).

This protocol ensures that no reference can prematurely be reclaimed since the use of the dirty-set in operation three and four guarantees that the race condition is avoided.

The RC algorithm implementation was merely a test of the general interface since the latency introduced when duplicating is not tolerable when as there are no advantages over the other DGC algorithms.

3.8.2 Indirect Reference Counting

The Indirect Reference Counting (IRC) scheme, developed by Piquer [18] and described in 2.3.2, is a DGC algorithm similar to RC in terms of implementation. With the new attribute, an indirection pointer, the protocol is very similar for both the Home DGCI and the Remote DGCI. The attributes and messages implemented for IRC can be found in Table 21.

Attributes

	Name	Type	Purpose
H	counter	Counter	The number of known references pointing to this entity.
R	counter	Counter	The number of known references pointing to this reference. When <i>counter</i> is greater than zero, the Remote will be a root.
R	decs	Counter	How many decrements to be made at its home, i.e. how many times the site have received the entity from the Home or the virtual Home.
R	sender	DSite	A pointer to the site where the first reference came from, the parent. This pointer must always be marked as “used” during local garbage collection.

Messages received

	Message tag	Purpose
H	GC_IRC_DROP	A referent site has deleted its proxy and returns <i>decs</i> many references.
H	GC_IRC_DEC	Notifies the Home that it should decrement <i>counter</i> with one since the home passed a reference to a node with a Virtual Home
R	GC_IRC_DROP	Works the same as for Homes.
R	GC_IRC_DEC	Works the same as for Homes.

Table 21. Indirect Reference Counting attributes and messages

The implementation follows the interface proposed in Section 3.6.2 and there is also a figure in Appendix C showing the events. The mutator operations and their events are found in Table 22.

No	Events
1	1 – Home sends a reference and increases <i>counter</i> . 2 – Remote instantiates the DGCI. <i>decs</i> = 1, <i>counter</i> = 0, <i>sender</i> = Home.

2	1 – Home sends a reference and increase <i>counter</i> . 2 – Remote merges the DGCI and, depending of <i>sender</i> , either sends a GC_IRC_DEC or increase <i>decs</i> . 3 – (In case of GC_IRC_DEC) Home decreases <i>counter</i> with one.
3	1 – Remote 1 sends a reference and increase <i>counter</i> . 2 – Remote 2 instantiate the DGCI. <i>decs</i> = 1, <i>counter</i> = 0.
4	1 – Remote 1 sends a reference and increase <i>counter</i> . 2 – Remote 2 merge the DGCI and, depending of <i>sender</i> , either sends a GC_IRC_DEC message or increase <i>decs</i> . 3 – (In case of GC_IRC_DEC) Remote 1 decreases <i>counter</i> with one.
6	1 – Remote sends a GC_RC_DROP to its Home/Virtual Home 2 – Home/Virtual Home decreases <i>counter</i> with <i>decs</i> .

Table 22. Indirect Reference Counting: The operations. The events for every mutator operation (No).

The protocol ensures that no reference can prematurely be reclaimed because of the use of *counter* in both parts act as the dirty-set.

3.8.3 Fractional Weighted Reference Counting

The Fractional Weighted Reference Counting (FWRC) scheme, developed by Klintskog et al. [19], is a DGC algorithm which simplifies the implementation of operations with weights, thus removing the problem of limited weights apparent in ordinary Weighted Reference Counting. Apart from the operations, an object with methods for handling and distributing the weights was implemented which allows for configuration on how to distribute weights, configured from the DGC Configuration unit and its methods. The object, called the *FracHandler*, has the methods described in Table 23. The weights are kept in a pool as enumerator-denominator pairs, both being 31 bit wide in the implementation. This object uses a memory effective way to handle weights, minimizing system memory calls.

Name	Task
insertPair	Inserts a received weight.
getNewRefWeightPair	Extracts/withdraw a weight from the pool, according to the configured distribution behavior.

Table 23. FWRC Weight handler.

Attributes and messages for FWRC can be found in Table 24. Since there is only one message, which returns the weight(s) to the Home, there is no tag for it.

Attributes

	Name	Type	Purpose
H	frac	FracHandler object	Keep the pool of weights.
R	frac	FracHandler object	Keep the pool of weights.

Messages received

Message tag	Purpose
H (None)	A Remote has deleted its proxy and returns a list of weights.

Table 24. Fractional WRC attributes and messages

The implementation follows the interface proposed in Section 3.6.2 and there is a figure in Appendix C showing the events. The mutator operations and their events are found in Table 25.

No	Events
1	1 – Home extracts a weight and sends the reference. 2 – Remote instantiate the DGCI with the weight received.
2	1 – Home extracts a weight and sends the reference. 2 – Remote merges the weight of the DGCI.
3	1 – Remote 1 extracts a weight and sends the reference. 2 – Remote 2 instantiate the DGCI with the weight received.
4	1 – Remote 1 extracts a weight and sends the reference. 2 – Remote 2 merges the weight of the DGCI.
6	1 – Remote sends an untagged message to the Home with its weight list. 2 – Home merges the weights.

Table 25. Fractional WRC: The operations. The events for every mutator operation (No).

The lack of a dirty-set in this DGC algorithm makes it very easy to implement and appealing to use¹ since it can free Remotes immediately and is using few messages.

3.8.4 Reference Listing

With Reference Listing, section 2.3.4, failure handling with respect to references is added to the system. In the project two versions of RL was implemented with different approaches of handling the dirty-set problem. This also results in different patterns when references are spread throughout a system.

The semantics of both versions of RL are very similar to RC except that *counter* and *unacked* are replaced with lists. The lists are handled through an object, the *SiteHandler*, which implements the methods found in Table 26. The *SiteHandler* associate every site with a counter to avoid race problems².

Name	Task
insertDSite	Inserts a site or increase the count of how many times inserted.
removeDSite	Decreases a sites count and when count is zero removes it.

¹ Author's opinion and verified in later tests.

² A race might occur when a site sends delete to home and another passes a ref to the same.

gcPreps	Method invoked when local garbage collections are performed. Marks up all DSites.
extract_info	Return a textual representation of the DSites in the list accompanied by their count.

Table 26. SiteHandler methods

The two main issues with sites Reference Listing are how to detect site crashes and how to distinguish between temporary failed and crashed sites.

The solution to both the issues is that the algorithms rely on the special site-pointer representation in Mozart, DSites, in another part of the Distribution Sub System DSS. This pointer representation allows for both live messages and makes the distinction between temporary or permanently failed sites, see section 2.5.1. With the use of DSites the fault tolerance is reduced to only checking the DSite state when explicit garbage collection is invoked from the run-time system. However, as these DSites are resources within the DSS, they must be protected from the local collector and this is achieved by invoking special preparation methods before collection so that DGC algorithms might secure whatever resource they want, see section 3.6.1.

3.8.4.1 Reference Listing Version 1

The first version, RL1, tries to minimize the impact of the Home when references are passed from one Remote to another. The semantics and the implementation of this DGC algorithm is exactly the same as for RC with the exception of counters replaced with lists as indicated by the name. The attributes and messages implemented for RL1 can be found in Table 27.

Attributes

	Name	Type	Purpose
H	siteList	SiteHandler object	The list of known sites with references to the entity.
R	siteList	SiteHandler object	The list of sites for which no ACK messages have been received. If this list is not empty the proxy is a root.
R	Decs	Counter	How many decrements to be made at the Home, i.e. how many times the node have received a reference.

Messages received

	Message tag	Purpose
H	GC_RLV1_DROP	A referent node has deleted its proxy and returns <i>decs</i> many references.
H	GC_RLV1_INC_AND_ACK	Notifies the Home that operation three was performed. Home inserts the site into its <i>siteList</i> .

- R GC_RLV1_ACK_HOME The remote node receives an ACK message for a passed reference, which comes from the home. It contains a site which is removed from the *siteList*.
- R GC_RLV1_ACK_REMOTE The remote node receives an ACK message from another Remote, see operation four in Table 28, and removes the site from the *siteList*.

Table 27. Reference Listing Version 1 attributes and messages

The implementation follows the interface proposed in Section 3.6.2 and there is a figure in Appendix C showing the events. The mutator operations and their events are found in Table 28.

No	Events
1	1 – Home sends a reference and inserts the destination site into <i>siteList</i> . 2 – Remote instantiate the DGCI. <i>decs</i> = 1.
2	1 – Home sends a reference and inserts the destination site (resulting in a counter increase for that site). 2 – Remote merges the DGCI and increase <i>decs</i> .
3	1 – Remote 1 sends a reference and inserts the destination site. 2 – Remote 2 instantiate the DGCI. <i>decs</i> =1. Further it sends a GC_RLV1_INC_AND_ACK to the Home. 3 – Home inserts the destination site and sends a GC_RLV1_ACK_HOME to Remote 1. 4 – Remote 1 removes the site from its <i>siteList</i> (may be a counter decrease).
4	1 – Remote 1 sends a reference and inserts the destination site. 2 – Remote 2 merges the DGCI and sends a GC_RLV1_ACK_REMOTE to Remote 1. 3 – Remote 1 removes the site from its <i>siteList</i> as above.
6	1 – Remote sends a GC_RLV1_DROP to Home 2 – Home removes the site from <i>siteList</i> .

Table 28. Reference Listing Version 1: The operations. The events for every mutator operation (No).

This protocol assures that if a site has crashed, then when explicit garbage collection is performed by the system, that site will be removed from the list and the entity might be reclaimed if there are no other sites referring to it.

3.8.4.2 Reference Listing Version 2

The second version, RL2, has a different semantic in that the sender of a reference informs the Home instead of the receiver as in RL1. Thus attributes for the Home are the same as for RL1 (Table 27) but Remotes does not receive any messages at all or have any dirty-set, *siteList*. The messages implemented for RL2 can be found in Table 29.

Messages received

	Message tag	Purpose
H	GC_RLV2_DROP	A referent node has deleted its proxy and returns <i>decs</i> many references.
H	GC_RLV2_INC	Notifies the Home that operation three was performed. Home inserts the site into its <i>siteList</i> .
H	GC_RLV2_DROP_UNUSED	This message is for the special case when a reference was prepared to be sent but later cancelled.

Table 29. Reference Listing Version 2 attributes and messages

The implementation follows the interface proposed in Section 3.6.2 and there is a figure in Appendix C showing the events. The mutator operations and their events are found in Table 30.

No	Events
1	1 – Home sends a reference and inserts the destination site into <i>siteList</i> . 2 – Remote instantiate the DGCI. <i>decs</i> = 1.
2	1 – Home sends a reference and inserts the destination site. 2 – Remote merges the DGCI and increase <i>decs</i> .
3	1 – Remote 1 sends both the reference to Remote 2 and a GC_RLV2_INC to Home. 2 – Home inserts the destination site. 3 – Remote 2 instantiate the DGCI. <i>decs</i> =1.
4	1 – Remote 1 sends a reference to Remote 2 and sends a GC_RLV2_INC to Home. 2 – Home inserts the destination site. 3 – Remote 2 merges the DGCI and increase <i>decs</i> .
6	1 – Remote sends a GC_RLV1_DROP to Home 2 – Home removes the site from <i>siteList</i> .

Table 30. Reference Listing Version 2: The operations. The events for every mutator operation (No).

This DGC algorithm is fault tolerant in the same manor as RL1. The difference is that it always messages the Home, thus saving one message in operation three.

3.8.5 Time Lease

The last of the DGC algorithm implemented during the project differs considerably in that this is not a reference counting DGC algorithm. The Time Lease (TL), section 2.3.5, algorithm uses timers to keep reference consistency. This DGC algorithm is not depending on the mutator operations to conclude

whether it is garbage or not, instead this is realized through the timers and the lack of messages for increased lease time.

When the Home is created it will request from the configuration unit the time it will add to its timer. When the timer expires it checks against its *expireDate*, the only attribute apparent in TL (Table 31) to conclude whether it is garbage or not. If not, the timer is reset with the remaining time of *expireDate*, else it will reclaim the space.

There are two situations where *expireDate* at the Home is increased:

- The Remote send a request message for increased lease.
- A reference is passed.

In both situations the lease is extended with a user configurable length, configurable from the DGC Configuration unit.

A Remote gets its *expireDate* through the mutator operations and by sending an explicit update message to the Home. A timer is initialized with the *expireDate*, minus a buffer time, ensuring that it has enough time to send a message for extension of the lease.

Attributes

	Name	Type	Purpose
H	<i>expireDate</i>	time	This time is compared against the system clock. If greater, the timer is reinitialized with the new time, else the Home is reclaimed.
R	<i>expireDate</i>	time	<i>expireDate</i> here works in the same manor as above but when the timer expires a message for increased time is dispatched. The failure protocol will notify the site if the entity has been prematurely reclaimed so no extra handling is required except waiting for answer.

Messages received

	Message tag	Purpose
H	(None)	A Remote requests more time, extend <i>expireDate</i> and dispatch a message containing the increase (value).
R	(None)	Read the returned value and increase the timer with this minus a buffer.

Table 31. Time Lease attributes and messages

The implementation follows the interface proposed in Section 3.6.2 and there is a figure in Appendix C showing the event. The mutator operations and their events are found in Table 32. In all events the timers are set according to *expireDate* although not explicitly stated.

No	Events
1	1 – Home sends a reference and increases <i>expireDate</i> . 2 – Remote instantiate the DGCI. <i>expireDate</i> = increase.

2	1 – Home sends a reference and increases <i>expireDate</i> . 2 – Remote merges the DGCI and increase <i>expireDate</i> = increase.
3	1 – Remote 1 sends a reference and attaches the remaining time. 2 – Remote 2 instantiate the DGCI. <i>expireDate</i> = attached value.
4	1 – Remote 1 sends a reference and attaches the remaining time. 2 – Remote 2 instantiate the DGCI. <i>expireDate</i> = max of attached value and its own remaining time.
6	-

Table 32. Time Lease: The operations. The events for every mutator operation (No).

This DGC algorithm is obviously fault tolerant for crashed sites when at least on site is working correctly. However, messages postponed by the communication handler might lead to prematurely reclaims so it is important to configure the buffer time to be long enough. Despite this, the *expireDate* and buffer time must not be too long since that may result in a system with poor memory utilization.

3.9 Performance and Benchmarking

For benchmarking the new implementation and evaluation of the DGC algorithms three set of tools were used: For the benchmarking part, with the purpose to get a hint of the impact on the system posed by the new protocol and interface, a part of a test-suite found in the Mozart programming package were used [29]. The three tests are given a short description in Table 33:

Name	Purpose
Benchmarking	To verify the impact of the implemented general interface.
Mutators	To verify the behavior of the protocols for the implemented DGC algorithms.
Token-Ring	A Real World test example stressing the ability to merge references.
Synchs	A Real World test example stressing the DSS and the DGC algorithms and their ability to handle many creations and deletions of references.

Table 33. Summary of tests used.

As the evaluation in this Master’s thesis is language-specific, the results might not be valid in other systems with other implementations of access structures, protocols and DGC algorithms. The standard [30] parameter used when comparing DGC algorithms is the number of messages required for a particular operation and that is used in this thesis.

3.9.1 Benchmarking system performance changes

The *Benchmarking* test are, without benchmarking the specific mutator operations, a comparison of the new versus old implementation using three

types of application operations which are common in a distributed environment:

- Exporting an entity.
- Importing an entity.
- Sequential import and export of entities.

These tests are a part of a distributed programming benchmarks suit [29] which tests all levels of the Mozart DSS for performance and is developed and implemented by the Mozart consortium [12]. The selected tests perform each operation 100000 times and they include testing of parallelism. This is realized through threading and all the tests offers the possibility to choose how many threads to be used and the 100000 times per operation is evenly shared among the threads.

3.9.2 Comparing DGC algorithms

For the more important tests regarding this Master’s thesis, two sets of tools were developed. The first set compares the DGC algorithms on a per operation basis, while the second set emulates real world cases. The latter set had to be implemented because of the lack of broad applications with distributed behavior, implemented in Mozart. All tests were conducted with only one DGC algorithm at a time. The tests were designed so that all participants were both connected and then synchronized to start at the same time. Before start, the OperCounter, Table 16, function was used to exclude the number of operations and messages from the connection. All results in the charts are the gathered sum of the total number of messages sent in the whole system for a specific DGC algorithm.

The *Mutators* tests for mutator operations were developed to verify and compare protocols, for the mutator operations in DGC algorithms, discussed in 3.8, and their events, the tests specifications are found in Table 34. The reason to bring in another version of operation four is to verify the nature of IRC, giving it an advantage in cases like 4b. The choice of 40000 entities is empirical from tests with TL, the implementation of timers in Mozart has shown that it is unable to handle more than 40000 timers on a node, thus limiting a test including timers, such as TL, to handle at most 40000 entities.

No	Events
1	Passing 40000 new ports.
2	Passing 40000 existing ports.
3	Setting up Borrower site one with 40000 ports, from an Owner site, and then passing them to Borrower site two.
4a	Setting up both Borrower site one and two with a port from an Owner site and then passing the same port 40000 times.
4b	Setting up Borrower site one with a port from an Owner site and then letting the same Borrower site one pass it to Borrower site two 40000 times.

Table 34. Mutator operation tests. Showing the set up for verifying each mutator operation (No). The 4b operation verifies the nature of IRC.

Operation six, delete, was not simulated and neither was failure handling, both being verified during the implementation phase. Deletion messages, in the cases when being sent, can be low prioritized and piggybacked on other messages, making it more difficult to benchmark an exclusive delete message sent.

The synthetic Real World test were simulating two kinds of events, passing token containing ports to the participants and simulating a synchronization barrier.

- *Token-Ring*: A set of processes are connected in a ring. Each knows only of its successor and predecessor process. The token, containing all the ports, enters the ring from an outside boot-strapping process and it is immediately passed from one process to the next. This procedure is repeated for a chosen number of times. As the token is passed the processes either build a new remote reference structure or merge reference information from all sites in the ring.
- *Synchs*: A synchronization barrier using logical variables. When the variables are distributed, they achieve a distributed synchronization. This test's result might seem questionable but it stresses the distribution sub system with its random behavior of spreading distributed references, thus showing the DGC algorithms ability to receive many instances of the same reference and delete them fast.

4 Results

The reference consistency protocol and the DGC algorithm methods developed as a part of this Master's project allow programmers to write to write their own, or implement an existing DGC algorithm. The protocols are verified through an internal Mozart verification test suit [29], developed by the Mozart consortium [12], with some of the implemented DGC algorithms, which ensure reliability and consistency. Six DGC algorithms have been implemented and their performances have been compared. In addition a benchmarking test using the clients described in section 3.9.1 were used to compare the old prototype of the component schema with the new extended one, to estimate the overhead of the general interface. All tests were run on a 450 MHz Pentium II workstation (See Appendix D).

4.1 Accomplished Project Tasks

The protocols developed are paired with a two-layer message scheme allowing for any type of message and thus simplifying implementation when a programmer wants to add his own DGC algorithm. A variety of DGC algorithms have been implemented which verifies this assumption.

To instrument and fine-tune the algorithms a configuration unit offers, through two interfaces, the ability to change both the default DGC algorithms to be used and later to change the current set of DGC algorithms and to change the behavior of a DGC algorithm in terms of timeouts or other specific settings. Further, a schema for expressing the persistent DGC algorithm has been developed. The component based model for handling DGC algorithms have been implemented, allowing explicit hybrid DGC algorithm composition and decomposition in run-time.

4.2 Benchmarking System Performance Changes

The *Benchmarking* tests were iterated 20 times for each version, using FWRC as the DGC algorithm since it was implemented in both the old and the new version. In every iteration, the three operations were performed two times; first they were configured to use one and then two threads. A resulting average user time in milliseconds was calculated for the 20 runs and the results can be found in Table 35 and in Figure 17 below.

Operation Threads	Old		New		(ms)
	1	2	1	2	
Exporting	13862	9454	15228	10198	
Importing	12552	8110	13273	9149	
ImportExport	15382	10816	17282	11948	

Table 35. Benchmarking old vs. new implementation.

As seen, the new implementation is a little slower, 6 – 12 per cent, than the old prototype and this is probably the consequence of using the Oz data structure as a message carrier (Section 3.4). To determine exactly how much

impact the use of the Oz data structure had were not possible, since this would require too great deal of efforts implementing a new message container and marshaling methods, although this would certainly boost performance. Further, the tests show that imports are cheaper than exports. The reason for this is unknown and would require great in-depth examinations of the DSS which was not conducted during this project.

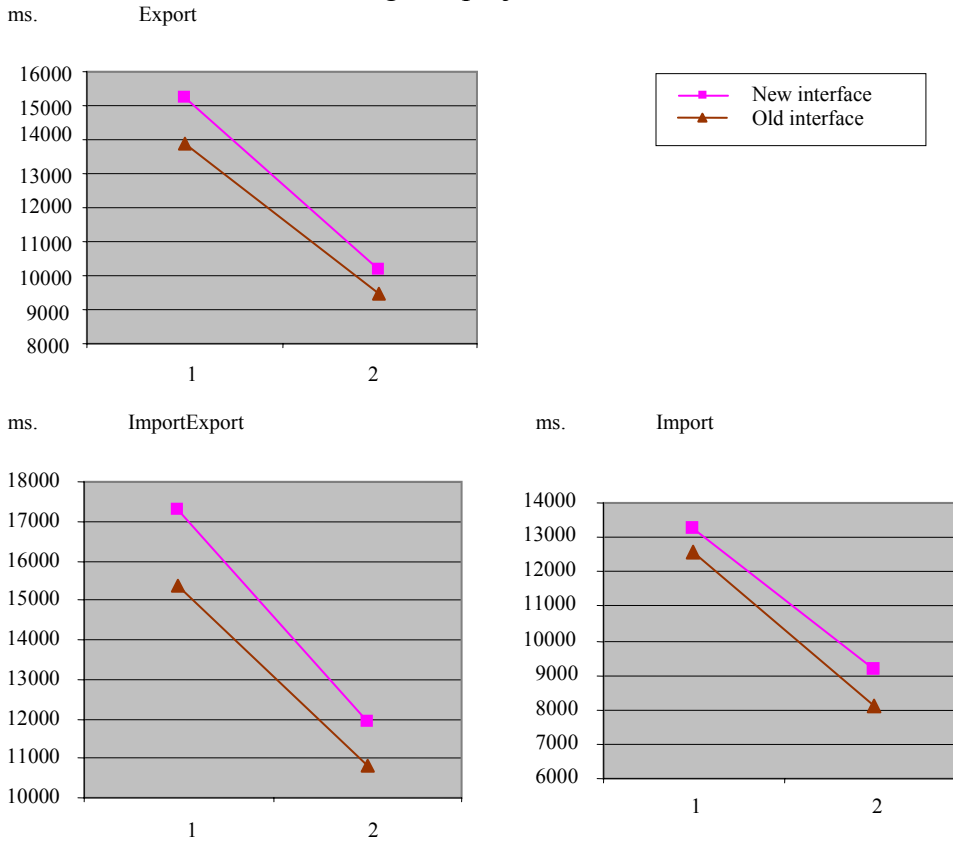


Figure 17. User time differences for application operations. The three application operation benchmarks shows a similar “tax”, introduced with the new general interface and message protocol, for all benchmarks. The figures show the number of messages when using either one or two threads.

4.3 Comparing Implemented DGC Algorithms

The comparison between DGC algorithms uses the number of messages as the factor to measure. This is relevant since the overhead in execution time is very much depending on the number of messages sent and received. A DGC algorithm with less, and shorter, messages involved, results in shorter execution time. In the case of the Reference Listing algorithms the memory consumption by the maintained lists is another matter which must be accounted for in terms of performance, although not dealt with in this thesis. In the following test suits the results are the average of 10 iterations of each test.

4.3.1 Mutator operations

The first tests are operation specific. They compare every DGC algorithm with only the desired DGC algorithm activated with the purpose of verifying the number of messages sent. The tests are described in 3.9.2. The results are shown Figure 18. As seen, operations involving the Owner-site are equal for all DGC algorithms. The difference comes when passing from one Borrower to another. These tests show, along with the theory of these DGC algorithms, that both IRC and RC are obsolete as reference counting DGC algorithms without failure handling. FWRC always uses least messages and deletion of proxies involves only one message for all of them (RC, IRC, FWRC) making RC and IRC obsolete. The fact that IRC may give long reference chains are another reason to this conclusion. In the case of the two Reference Listing algorithms, a conclusion would be harder to draw: in a loosely coupled system it might be of importance to invoke the Owner site as seldom as possible giving an advantage to RL version 1, otherwise version two seems supreme.

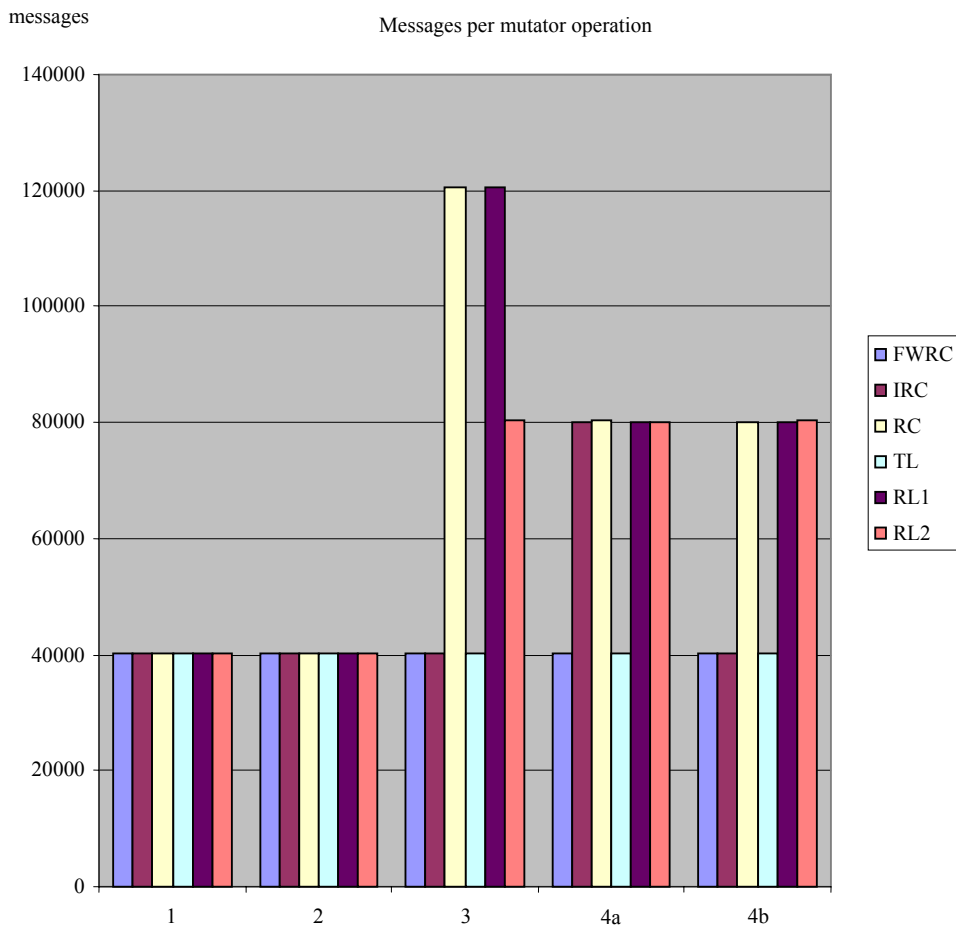


Figure 18. Messages per operation. The number of messages for 40000 iterations of a specific mutator operation. Both FWRC and TL show equal number of messages for all operations having the least number of messages.

4.3.2 Token-Ring tests

The synthetic *Token-Ring* described in 3.9.2 offers a glimpse of the real world performance of the implemented DGC algorithms. These tests show a better approximation of the real behavior of a distributed system and especially the Mozart language. The *Token-Ring* test was configured to use:

- Three clients and 100000 passes of the token.
- Five clients and 100000 passes of the token.
- Five clients and 300000 passes of the token.

These configurations show how varying the parameters of a set up would account for a decrease/increase in the number of messages. The results are presented in Figure 19 below and as seen, varying the number of participants has no impact for a system with TL or FWRC (3 clients - 100000 vs. 5 clients 100000). Changing the number of passes scales perfectly well as expected.

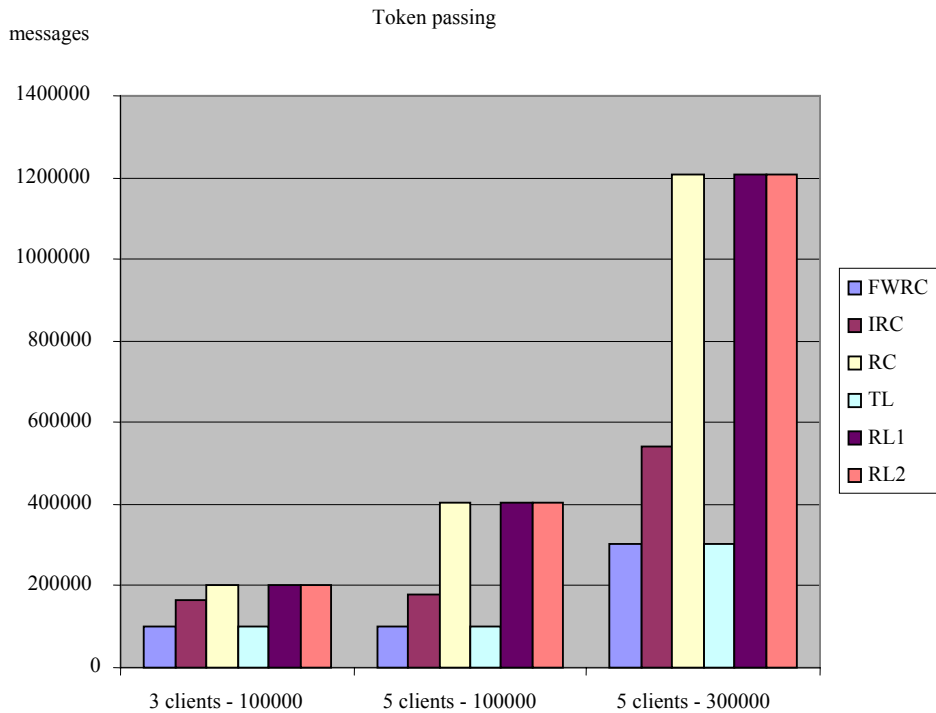


Figure 19. Messages for token passing. This tests stresses merging capabilities and shows that RC, RL1 and RL2 scales with the increase of clients while the other shows no or little impact when merging are stressed.

4.3.3 Synchs tests

The second synthetic test, *Synchs*, shows a situation where clients access new logical variables all the time as the synchronization barriers are completed. The tests are configured as follows:

- Three clients and 1000 synchs.
- Five clients and 1000 synchs.
- Five clients and 10000 synchs.

This test, as concluded when analyzing Figure 20 where the TL algorithm, the only DGC algorithm without deletion messages, shows the least number of messages, introduces the delete operation as the proxies are continuously deleted when a barrier is complete.

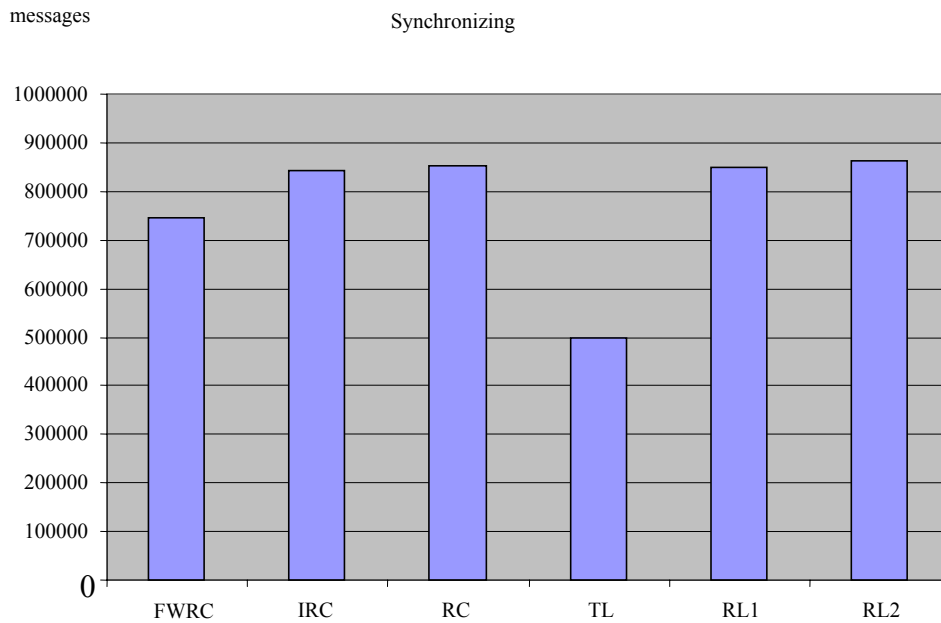
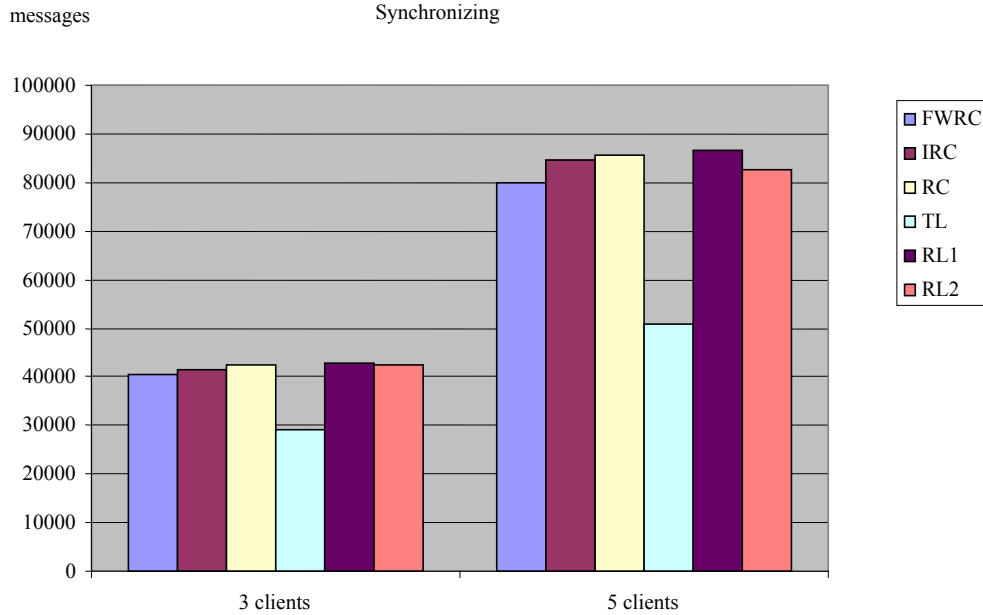


Figure 20. Synchronization test. The two upper are for 1000 synchs and the lower for 10000 synchs with five clients. Varying the both number of clients and the number of synchs shows linear scaling.

A naive conclusion would be that this is an advantage with the timed algorithm which has the least number of messages. However, when considering the Owner table sizes for the clients, the TL algorithm keeps all

the entities in memory for the whole expireDate period (section 3.8.5) which is arbitrary long. This gives an extremely poor utilization but this drawback might be dealt with if clever schemas of selecting the period lengths are chosen.

When not considering the TL algorithm these results show that the variable binding synchronization obviously must contain a lot of either mutator operation one or two since no DGC algorithm is, in terms of messages, clearly better than the others.

5 Conclusions

The protocols and the framework for implementing DGC algorithms presented in this Master's thesis allow programmers to implement their own DGC algorithms, using the interface provided by the reference consistency protocol in junction with the basic DGC algorithm interface. Tools and interface to DGC algorithms, for the instrumentation of instances allows for further tuning of the DGC instances' behavior. Further, the interface allows for explicit composition of a DGC algorithm mix, creating hybrid DGC algorithms.

The overhead added by the new interface and schema is somewhere between 6 and 12 per cent extra execution time compared to the old prototype implementation, this overhead is largely depending on the use of the internal Oz data structure as a message carrier, as opposed to the old prototype using many specialized messages with constant length.

Further, a complete protocol for identification and handling of failure events, such as site crashes or obsolete DGC algorithm messages has been introduced. This protocol is independent of DGC algorithms in use and work primary between the reference consistency layer and the message engine.

Among the implemented DGC algorithms for the Master's project, three (or four distinctive) usable DGC algorithms have emerged as viable alternatives, each with their own strengths and weaknesses:

- The TL algorithm is suitable for entities with fault tolerance needs and which are expected to be long-lived and passed many times, such as a token. It is also very suitable for entities having short-lived references that are constantly imported and then deleted from the referring site.
- The FWRC algorithm is suitable for entities that are short lived or when the system have no expectation of how long they may live, the advantage of this algorithm is that it is fast reclaiming, allowing for better memory utilization than TL. Entities which are close to persistent would have much use of this algorithm since few extra messages are sent when spreading.
- The RL algorithms combines fault tolerance with fast reclaiming to the prize of extra messages, these algorithms main usage is fault tolerance paired with fast reclamation, the major drawback is the memory consumption imposed by the list(s).

5.1 Future Work

As this Master's project concerned the development of an interface for various DGC algorithms and the implementation of a few basic DGC algorithms, some issues evolved during the work. The interface to DGC instances and the configuration unit is very basic and could be enhanced with better tools for configuration, especially for DGC instances. This leads to the second issue, namely the tuning of algorithms. It is certainly feasible to improve the algorithms to use either static or adaptive methods for improved

performance, some examples are: The two Reference Listing algorithms could be merged into one with the ability to change protocol as it is spread throughout the system. The Time Lease algorithm should use another schema of increasing the lease depending of either an adaptive approach which empirically assigning the lease or a static exponentially growing lease approach.

Another area of interest is the implementation of group behavior for entities. Today default DGC algorithms are assigned to every entity independently of the type or purpose. This might be handled through a group notion, assigning an entity to a group consisting of objects with a certain behavior or presumed spreading.

As the project progressed, the need for an extensive test bench appeared. This test bench should allow for examine reclamation of cells in hybrid DGC algorithm compositions and show the impact of tuning algorithms. Further it should be able to give a hint of the significance of mutator operations in applications versus messages and the memory utilization at certain intervals.

6 Bibliography

- [1] Joy, B., Steele, G., Gosling, J. and Bracha G. 2000. *The Java Language Specification, Second Edition*. Addison-Wesley, ISBN 0-201-31008-2.
- [2] Ungar, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 157-167.
- [3] Dijkstra, E. W., Lamport, L., Martin, A. J. And Steffens, E. F. M. 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21, 11, 966-975.
- [4] Collis, G. E. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12, 655-657.
- [5] McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine: Part I. *Commun. ACM* 3, 4, 184-195.
- [6] Fenichel, R. R. and Yochelson, J.C. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11, 611-612.
- [7] Saleh, E. A. and Graem A. R. 1998. Garbage Collecting the Internet: A survey of Distributed Garbage Collection. *ACM Computing Surveys*, 30, 3, Sept 1998.
- [8] Dickman, P. 1991. Distributed Object Management in a Non-Small Graph of Autonomous Networks With Few Failures. *PhD thesis, University of Cambridge, UK*.
- [9] Vestal, S.C. 1987. Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming. *PhD thesis, University of Washington, Seattle, Washington*.
- [10] Lins, R.D. and Jones, R.E. 1991. Cyclic weighted reference counting. *Technical Report 95, University of Kent, UK*.
- [11] Shapiro, M., Dickman, P. and Plainfossé D. 1992. Robust, distributed references and asyclic garbage collection. In *Symp. on Principles of Distributed Computing ACM 1992*.
- [12] The Mozart Programming System. <http://www.mozart-oz.org>. December 2001.
- [13] Wollrath, A., Riggs, R. and Waldo, J. 1996. A distributed object model for the java system. In *Conf. on Object-Oriented Technologies, 1996 Usenix*.
- [14] Sessions, R. 1998. *COM and DCOM: Microsoft's Vision for Distributed Objects*. Wiley. ISBN 0-471-19381-X.
- [15] Plainfossé, D. and Shapiro, M. 1995. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management, UK, 1995*.
- [16] Klinskog, E., Neiderud, A., El Banna, Z., Brand, P. and Haridi, S. 2001. *Component-Based Distributed Garbage Collection*. Un-published, Swedish Institute of Computer Science, October 2001.

- [17] Lermen, C. W. and Maurer, D. 1986. A protocol for distributed reference counting. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming (MIT)*, 343-350.
- [18] Piquer, JM. 1991. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 – Parallell Architectures and Languages Europe, Lecture Notes in Computer Science 505, Springer-Verlag* 150-165.
- [19] Klinskog, E., Neiderud, A., Brand, P. and Haridi, S. 2001. Fractional Weighted Reference Counting. In *EuroPar 2001*.
- [20] Bevan D. I. 1987. Distributed garbage collection using reference counting. In *PARLE '91 – Parallell Architectures and Languages Europe, Lecture Notes in Computer Science 259, Springer-Verlag* 176-187.
- [21] Birell, A., Evers, D., Nelson, G., Owicki, S., and Wobber, E. 1993. Distributed garbage collection for network objects. *TR 116. Digital Equipment Corp. Research Center*.
- [22] Hughes, J. 1985. A distributed garbage collection algorithm, In *Functional Programming Languages and Computer Architecture, LCNS 201, Springer-Verlag, New York*, 256-272.
- [23] Lamport, L. 1978. Time, clocks and the ordering of events in a distributed system. *Commun. ACM* 21, 7, 558-565.
- [24] Austin, C. Pawlan, M. 1999. *Advanced Programming for the Java 2 Platform. Addison-Wesley, ISBN 0-201-71501-5*.
- [25] Mozart Documentation. <http://www.mozart-oz.org/documentation/>. December 2001.
- [26] Mozart Documentation. The Distribution Subsystem. http://www.mozart-oz.org/documentation/ds_white_paper/index.html. January 2002.
- [27] Lindholm, T. Yellin, F. 1999. *The Java Virtual Machine Specification. Addison-Wesley, ISBN 0-201-43294-3*.
- [28] Klinskog, E. 2001. *DS Licentiate. Swedish Institute of Computer Science*. Work in progress September 2001.
- [29] Mozart Download. Mozart: Anonymous CVS. <http://www.mozart-oz.org/download/cvs.html>. February 2002.
- [30] Saleh E. A. 1995. Empirical studies of Distributed Garbage Collection. *PhD thesis, Department of Computer Science, Queen Mary and Wesfield College, University of London December 1995*.

7 Appendix A - List of Definitions and Acronyms

Borrower: A site which has a reference to a specific entity on another site.

DGC algorithm: A garbage collection algorithm used for collection of garbage with reference spanning over multiple sites.

DGC instance: A DGC instance or DGCI is the representation of a DGC algorithm at run-time. While the DGC algorithm defines the protocol and states, a DGCI is the realization of the DGC algorithm having a particular state in the protocol, during run-time.

Dirty-set: Having dirty-set means that a Remote DGCI has entered a state where it can not be removed even if it is no longer needed at the site. This is most often caused by a message to another site not being acknowledged.

DSite: A DSite is a remote site pointer used in the Distributed Sub System of Mozart. DSites are explained in Section 2.5.1.

Entity: Language object or object structure in applications or programs, representing for instance a cell, integer, port.

Home: The Home refers to the part of a DGC algorithm residing on an Owner site.

IF: abbreviation for interface.

Mutator: The mutator is the application performing operations on the memory graph, described in section 2.1, for single processes. However, the notion extends to the distribution environment, where a mutator might perform an operation involving another site.

Owner: Owner refers to the site managing a specific entity. This is for a non-migrating system the same site that created the entity.

Remote: The Remote refers to the part of a DGC algorithm which resides on a Borrower site.

Single-Address-Space Garbage Collection: garbage collection for a process with non-shared memory space.

Site: A site is a process, with location, having communication capabilities and non-shared memory space. References to this memory space from other processes are handled by the communication facilities.

8 Appendix B - Failure Protocol

The full failure protocol shows actions for every possible combination of failure. This protocol guarantees that the system will perform actions resulting in a stable state.

At site	Received Message type	Failure	Action
Home	M_HOME_DGCI	No Entity	Send back (to Remote) M_HOME_ENTITY_FAILED
Home	M_HOME_DGCI	No DGCI	Send back (to Remote) M_ALGORITHM_REMOVED
Remote	M_REMOTE_DGCI	No Entity	Send message back (to Remote) M_REMOTE_ENTITY_REMOVED
Remote	M_REMOTE_DGCI	No DGCI	Do nothing
Remote	M_HOME_ENTITY_FAILED	No Entity	Do nothing
Remote	M_ALGORITHM_REMOVED	No Entity	Do nothing
Remote	M_ALGORITHM_REMOVED	No DGCI	Do nothing
Home	M_REMOTE_ENTITY_REMOVED	No Entity	Do nothing
Home	M_REMOTE_ENTITY_REMOVED	No DGCI	Do nothing
Remote	M_REMOTE_ENTITY_REMOVED	No Entity	Do nothing
Remote	M_REMOTE_ENTITY_REMOVED	No DGCI	Do nothing

Table 36. Failure protocol cases.

9 Appendix C - Mutator Operations

Reference Counting operations depicted:

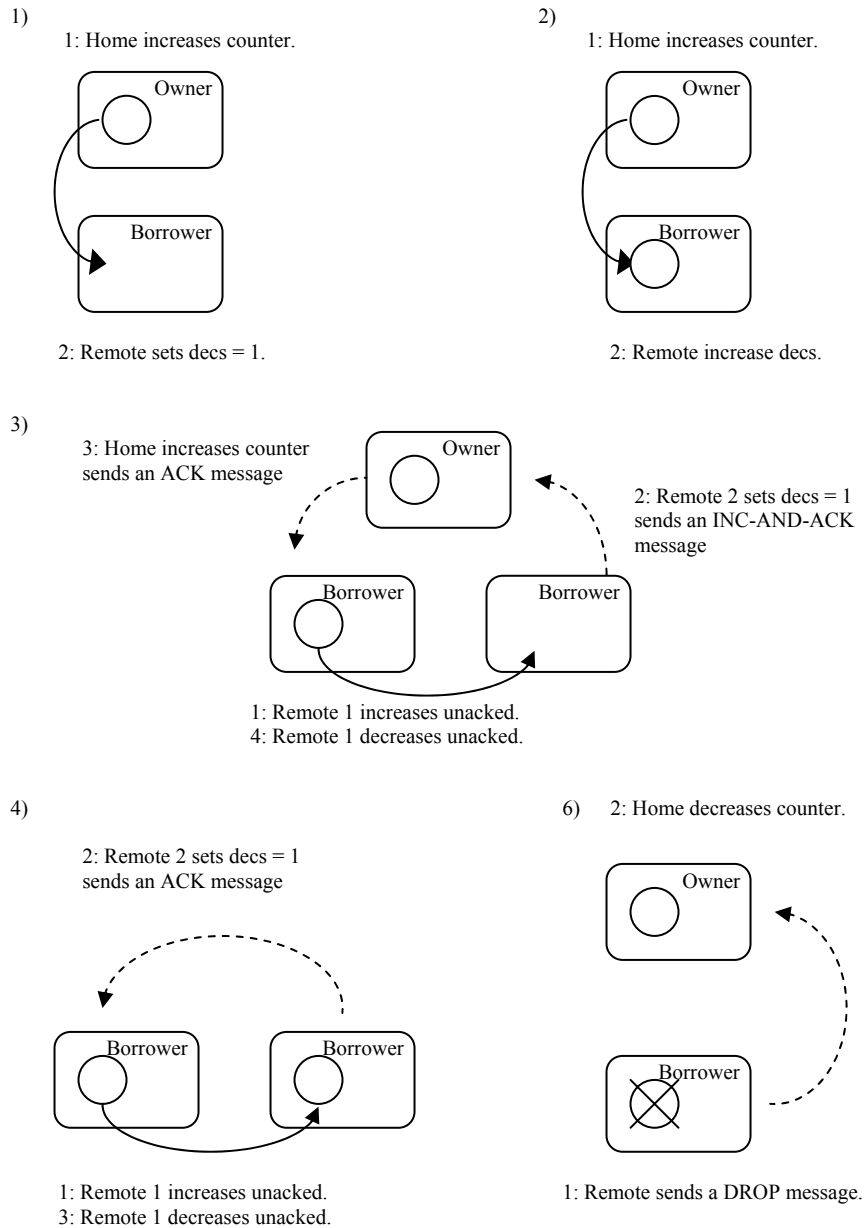


Figure 21. RC operations depicted

Indirect Reference Counting operations depicted:

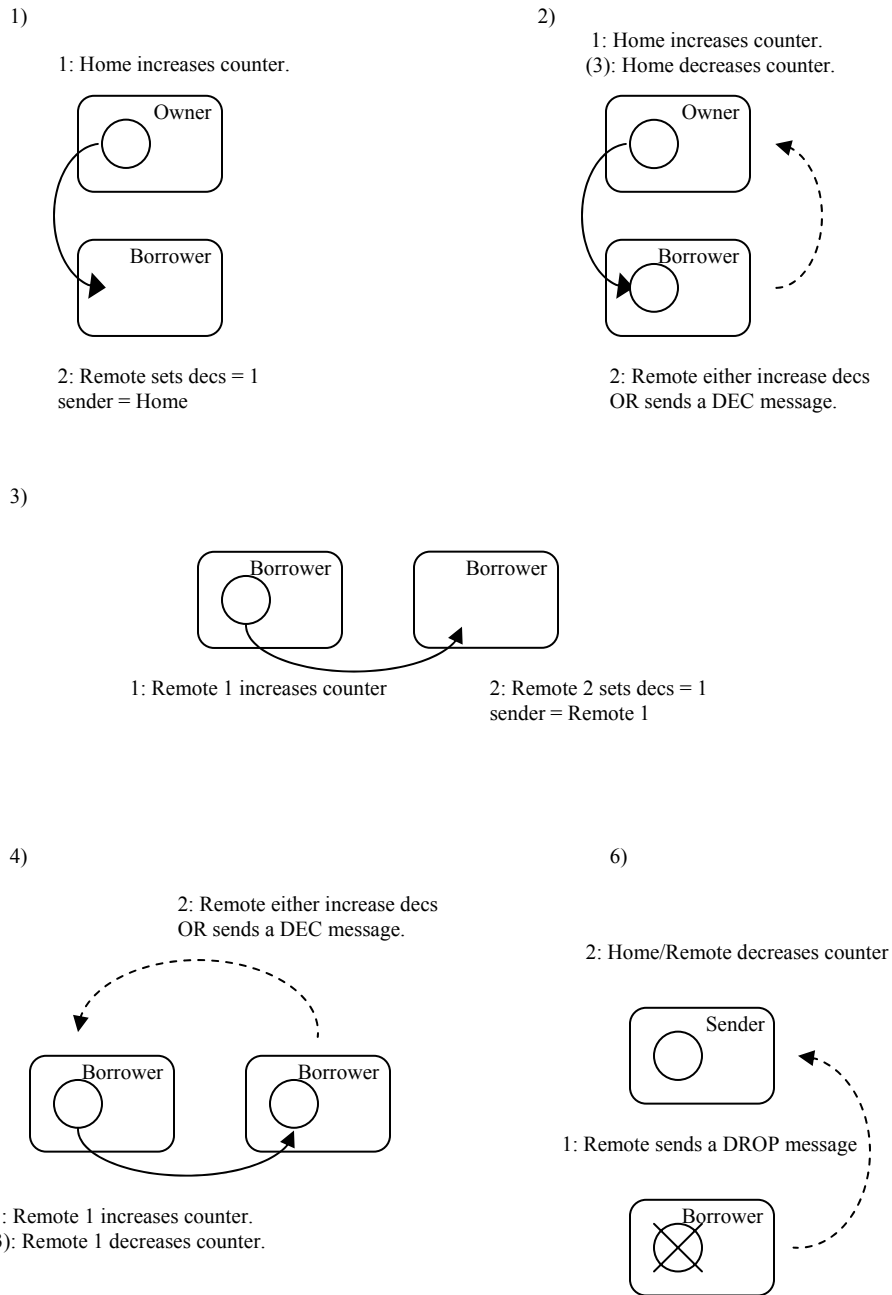
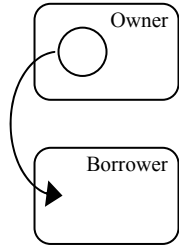


Figure 22. Indirect Reference Counting operations depicted.

Fractional Weighted Reference Counting operations depicted:

1)

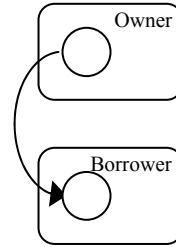
1: Home removes weight



2: Remote inserts weight

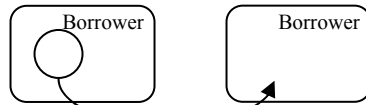
2)

1: Home removes weight



2: Remote inserts weight

3)

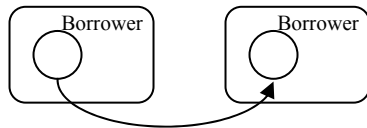


1: Remote 1 removes weight

2: Remote 2 inserts weight

4)

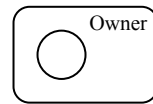
2: Remote 2 inserts weight



1: Remote 1 removes weight

6)

2: Home inserts weight.



1: Remote sends untagged message with weight(s).

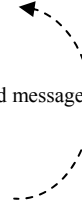
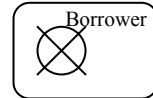


Figure 23. Fractional Weighted Reference Counting operations.

Reference Listing Version 1 operations depicted:

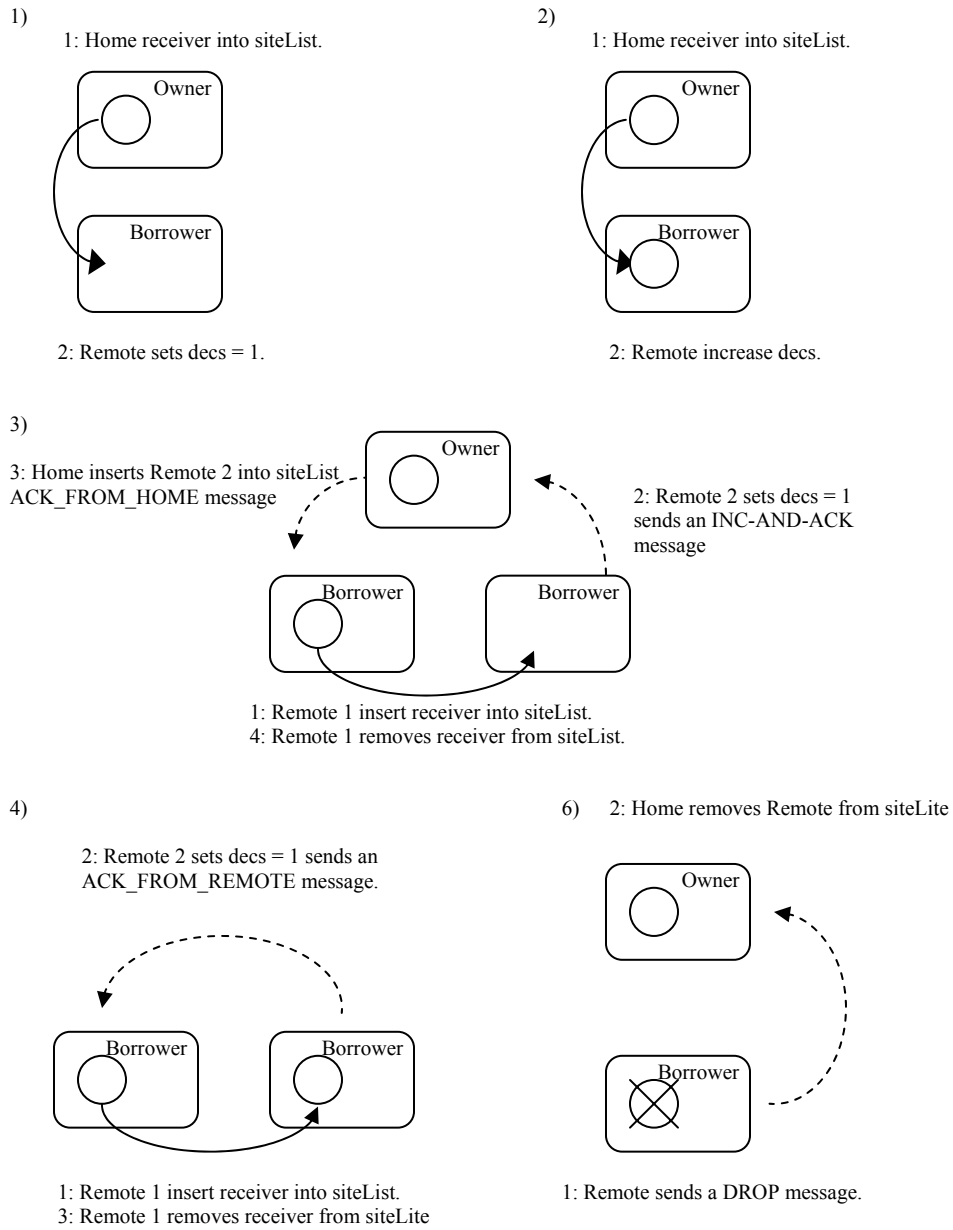


Figure 24. Reference Listing Version 1 operations.

Reference Listing Version 2 operations depicted:

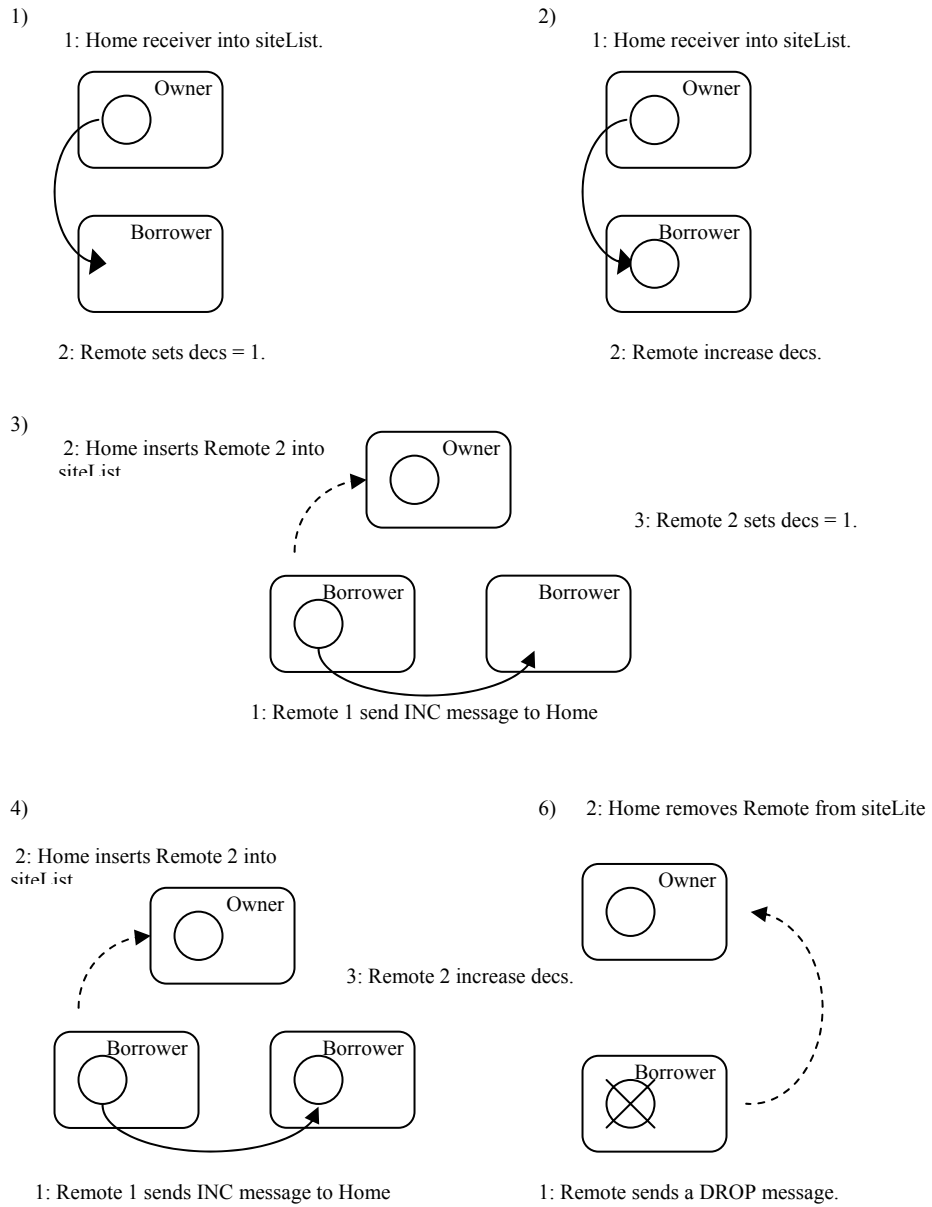
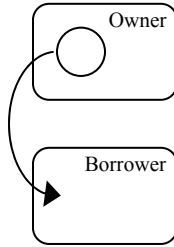


Figure 25. Reference Listing Version 2 operations.

Time Lease operations depicted:

1)

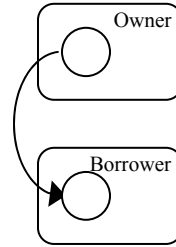
1: Home increases expireDate.



2: Remote sets expireDate = increase

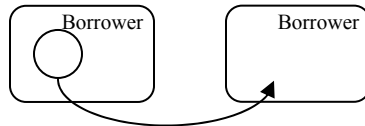
2)

1: Home increases expireDate.



2: Remote sets expireDate = increase

3)

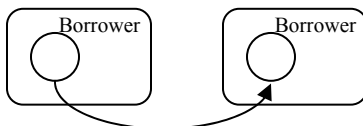


1: Remote sends remaining time.

2: Remote sets expireDate = sent time

4)

2: Remote sets expireDate max of remaining time and received time



1: Remote sends remaining time.

Figure 26. Time Lease operations.

10 Appendix D - Test Setup

The system was configured with the following components:

CPU:	450 MHz Intel Pentium II.
Primary memory:	256Mb.
OS:	Linux, Slackware with kernel version 2.2.19.
Compiler:	Gnu GCC 2.95.3.
Mozart:	Developer version 1.3.0

During the test-runs only the test programs were running.