Secondary Storage Garbage Collection for Decentralized Object-Based Systems

Bjoernerstedt, Anders

Secondary Storage Garbage Collection for Decentralized Object-Based Systems

Anders Björnerstedt

Abstract

This paper describes a mechanism for secondary storage garbage collection that may be used to reclaim inaccessible resources in decentralized persistent object based systems.

Schemes for object addressing and object identification are discussed and a proposal is made which handles volatile objects separately from persistent objects. The garbage collection of the space of volatile objects is decoupled from the garbage collection of the space of persistent objects. The first kind of garbage collection can avoid the complexity and overhead of a distributed algorithm by classifying "exported" objects as persistent. The problem of detecting and collecting "distributed garbage" is then deferred to garbage collection of persistent objects.

1 Introduction

Object Oriented programming generates large numbers of dynamically allocated objects. Many of these are used for a limited period of time and then become unreachable. To detect such "garbage" and make it available for reuse is the garbage collection problem. The responsibility of releasing unused resources can be placed either on the system (execution environment) or on the programmer. If an automatic garbage collection scheme is incorporated into the execution environment, then a great deal of programmer effort and error can be avoided. The disadvantage with automatic garbage collection is that it causes either overhead or disruptions in the normal application processing.

Another service which the execution environment can provide is *persistence* [Atkin87]. By this we mean that the execution environment makes it possible for objects to exist for an indefinite period of time. This service also relieves the programmer of a problem, namely how to map the abstract objects with which it is desired to compute, to secondary (non-volatile) storage.

For several technical reasons objects which are persistent (long lived) are managed differently from the volatile (short lived) objects. Persistence is very much technology dependent because persistence places demands on the properties of the memory architecture, such as fault tolerance and non-volatility. The difference in management usually leads to different garbage collection mechanisms, one for volatile memory and one for persistent memory. Garbage collecting the memory of persistent objects is expected to be much more time consuming than garbage collecting the memory of volatile objects. Because the management of persistent memory is more complicated than the management

of volatile memory, the number of persistent objects is much larger than the number of volatile, and persistent memory builds on slower hardware. On the other hand we expect the *need* to garbage collect the persistent memory to be a much rarer occurrence than the need to garbage collect the volatile memory. The reason again being the large size of the persistent memory.

A *decentralized* system [Gray86] allows multiple centers of control. This is useful for several reasons: possibly increased performance due to increased potential for concurrency; increased availability due to opportunity for physical distribution; and, perhaps most importantly, increased autonomy for the "owners" of information. We prefer the term *decentralized* instead of *distributed* because autonomy is important to us. By autonomy we mean that the system should allow different logical locations to have independent control over resources. Each location is associated with an authority that "owns" and controls the local resources. The authority has the power to veto remote requests to use the local resources. Physical distribution does not imply decentralization, while decentralization "weakly" implies physical distribution. A "federated system" would be an acceptable alternative term to "decentralized system." A decentralized system is one special kind of distributed system. The garbage collection mechanism we present is not suitable for all kinds of distributed systems.

Garbage collection for decentralized systems is a difficult problem. Either a distributed algorithm working in parallel with normal operation has to be devised, or the system has to stop normal operation while doing garbage collection. In this paper we propose a solution which is a compromise between the two. We delegate the problem of garbage collecting the global object space to the mechanism for collecting the persistent object space. We assume that garbage collection of volatile objects is managed on a local basis and during normal operation. We do not care which method is used for garbage collecting the space of volatile objects, indeed, different methods could be used at different locations. Persistent objects are not reclaimed during normal processing but may be removed or flushed from primary storage to free space when needed. We say that an object is *exported* if either the object itself or a reference to it is moved from one location to another. If an object is exported, then the system will manage the object as persistent.

Garbage collection of persistent objects, i.e. reclaiming secondary storage, does not have to be done as frequently as for primary storage. We are talking about periods on the order of magnitude of days or weeks, or even longer, instead of seconds or subseconds which may be needed for primary storage garbage collection. Because of this we assume that it is acceptable to either temporarily stop normal operation locally (not globally) for secondary storage garbage collection, or alternatively to temporarily significantly degrade local performance, if the secondary storage garbage collector is run in parallel with normal operation. The different locations of the decentralized system decide when and how often to have this disruption. There is then local autonomy concerning when local persistent garbage should be collected. For global garbage, i.e. persistent objects which have become known to more than one location and then have become garbage, cooperation is required for safe garbage collection. The mechanism we propose puts emphasis on local autonomy yet achieves safe global garbage collection when all locations

cooperate.

The paper is structured as follows. In section 2 we give a more elaborate motivation for our approach. In section 3 we describe a model of the system we envision and point out some important assumptions we are making. In section 4 we discuss the requirements we have on the garbage collection mechanism. In section 5 we present our mechanism.

2 Background and Motivation

The mechanisms for providing garbage collection, persistence and decentralization all have in common that they are very dependent on the addressing and identification mechanisms used by the execution environment. First, we discuss addressing schemes in relation to persistence, then in relation to decentralization. Because our focus is on garbage collection we concentrate on the design issues of addressing schemes which specifically influences garbage collection. We then propose a combined decentralized and persistent identification scheme. We end this section by explaining how the global garbage collection problem can be subdivided into smaller sub-problems.

2.1 Addressing and Persistence

The reasoning in this section borrows from a study of different addressing mechanism suitable for implementing persistence, done by Cockshott [Cocks88].

A very coarse grained level of persistence is to load and dump the whole object space (the image) using a file. Loading is done at the start of a session and dumping at the end or whenever a checkpoint is desired. The execution environment uses primary memory addresses (real or virtual), as provided by the operating system and hardware, for the lowest level of object addressing. During a session all objects reside in virtual memory (or real memory if virtual memory is not supported). There is no distinction between volatile and persistent objects in terms of how they are managed, i.e. the whole space of objects has the same failure mode. But the file on which the image is dumped resides on secondary storage with a failure mode independent from virtual memory. For example in most implementations of Smalltalk-80 [Goldb83] use this approach. This approach is either not very fault tolerant since a crash destroys all changes since the last save of the image, or very expensive if frequent saves of the whole image have to be made. It is mostly suitable for single user systems. We call this the *workspace* approach to persistence.

Another possibility is to have different failure modes for different segments of virtual memory, but still make use of virtual memory as the lowest level of addressing. The primary purpose of virtual memory is still to obtain a large address space, but the virtual memory backing store is also used as the basis for persistence. Since a system crash could corrupt the system image, or leave it in an unknown state, additional information such as a log must also be maintained on secondary storage to aid recovery from crashes. This approach has been explored by Thatte [Thatt86] and by Kolodner, Liskov and Weihl

[Kolod89]. When the system does not encounter any failures then the backing store remains consistent. There is then no need for initial and final loading and dumping of objects as in the workspace approach. If a crash does occur then a special recovery routine is used to restore the backing store to a consistent state which may need to use the log. The underlying operating system and hardware provides the translation between physical RAM and the backing store, but the execution environment has control over when and how this is done. This requires either the execution environment to be indistinguishable from the operating system (e.g. a Lisp machine) or the operating system to allow the implementor of the execution environment some control over the virtual memory backing store, as in Multics [Daley68] or Mach [Jones86]. In the system described in [Kolod89], the heap is divided into the stable heap and the volatile heap. This is done by partitioning the address space for the virtual memory used for the heap into two disjoint segments. The execution environment manages the log and the paging for the stable heap so that the stable heap has a different failure mode from the rest of virtual memory. The management of the log and stable heap uses a transaction model [Gray78], so a finer granularity of persistence is achieved (compared to the workspace model) and multiple users can be supported. We call this approach the *disciplined virtual memory* approach.

A third approach is when the execution environment maintains a separate mapping between a repository on secondary storage and virtual memory. In other words virtual memory is only used for obtaining a large primary store. Persistence is implemented by a separate mapping to secondary storage. A transaction mechanism may also be used in this approach so the consistency of the persistent store is ensured. One difference between this approach and the disciplined virtual memory approach is that normally only a subset of all existing objects will be in virtual memory. In the same way as the operating system uses physical RAM as a cache against the virtual memory backing store, the virtual memory is used by the execution environment as a cache against the persistent store on secondary storage. Examples of this approach are GemStone [Purdy87], PS-algol [Atkin87], Mneme [Moss88], Comandos [COMAN87], Avance [Björn88] and, in general, most of today's database systems. We call this the *multilevel* approach, because several levels of memory are explicitly managed by the execution environment.

Note that no logical difference between these approaches is visible to the user of the system. The whole point with "persistence" is to provide a single level store abstraction to the programmer.

2.2 Addressing and Decentralization

A requirement for decentralization is a very large address space. Such a space could be structured or flat. We will not attempt to characterize the actual size needed for the global address space. We can give a lower bound by saying that todays hardware architecture, dominated by 32 bit addressing, is insufficient.

An excellent exposition of these issues can be found in a paper by Moss [Moss89]. Moss argues for short context dependent addresses (which are in essence structured addresses) and against flat global addresses. The argument is mainly against using a

large global paged virtual memory as the basic approach. His arguments are based on price/performance, complexity, autonomy and flexibility.

Moss also takes the position that a large flat object identifier space has most of the same disadvantages. This we do not agree with. An identifier space differs from an address space in that an identifier is not location dependent. Having large context free identifiers for objects does add a level of indirection, which does add execution cost, but caching can significantly reduce this problem. The major advantages with context free identifiers are the flexibility they give and simplifications in system software. However, if objects are basically stationary, then using a global flat identifier space may not be worth the cost. We assume that objects may move in the decentralized system and that therefore the distinction between identity and address is important. We note that the Comandos architecture [COMAN87] has taken this approach.

We can accept structured identifiers that *hint* at location. For example the identifier may be constructed on the basis of where the object is created. But it should be an identifier and not an address. In other words if the object moves then its identity should not change.

The identifiers used are machine generated surrogates which are guaranteed to be globally unique. Global uniqueness is easy to achieve and does not need to have any practical consequences on local autonomy. For example, one can easily construct a world wide globally unique identifier by combining:

1. The internet address of the physical host, which must be unambiguous in the world wide distributed system.

2. A logical location number, unique relative to the host, to disambiguate between logical locations in the decentralized system.

3. A host local timestamp of sufficient resolution.

Such an identifier would be unique and would also tell where the object was created, which could be used as a good hint of location.

Smalltalk has been used as the implementation basis for decentralized systems [Decou86, Benne87, McCul87, Schel88]. Each location is then a single user workspace. Various forms of forwarding objects (proxys) are used for remote addressing. Except for the system described by McCullough [McCul87], objects generally do not maintain strong (surrogate based) identity between different locations. The reason being, we believe, that all of the systems based on Smalltalk-80 want to avoid major changes to the virtual machine. In addition these systems have the problem of ensuring fault-tolerant consistency only on a per-workspace basis. [1] Transactions would require all workstations to take coordinated dumps of their images each time a commit is made, or a change to the implementation of persistence has to be made.

[1] The system described in [Schel88] does provide global consistency for *replicated* objects using the so called Thomas write rule [Thoma79]. However these objects are special and only a small part of the total object space.

We conclude the discussion on addressing and decentralization by assuming a large and global identifier space as the best solution for a decentralized system where objects move. We also conclude that a transaction based approach is appropriate to ensure consistency in a multiuser system, which a decentralized system is by definition. This means that the simple workspace approach for persistence is not appropriate. Using a transaction based approach also means that we can have multiple users at each location.

2.3 A decentralized and persistent identifier space

An addressing scheme for persistent objects which interacts with a transaction mechanism is complicated to implement. The same is true for an identification scheme for objects in a decentralized system which also interacts with a transaction mechanism. It then makes sense to try to unify the two as far as possible. It also makes sense to find means to avoid using the expensive decentralized and persistent identification mechanism for all objects.

Using location independent identifiers for persistent objects means that there must be at least one level of indirection between identifiers and persistent addresses. The disciplined virtual memory approach could still be used for *local* persistent addressing. In fact an advantage of having the indirection between identifier and address is that several different persistent addressing mechanisms could be used, even at one location in the decentralized system.

A beneficial consequence of this approach is that any persistent object may be exported without complications. Another consequence is that any object which is exported is also persistent. This may in some cases cause efficiency problems. We will return to this problem in section 2.4.

Since the proposed identification mechanism will be expensive to use, alternative and simpler mechanisms must also be provided. Volatile objects could use a short and simple addressing scheme. If necessary, they would be *matured* to persistent objects, and only then be associated with a long identifier. [2] A distinction orthogonal to volatile/persistent is used in the Avance system [Björn88], where a distinction is made between dependent and independent objects. [3] Dependent objects are owned by, and internal to, some independent object and may not be shared between independent objects. Dependent objects always accompany their enclosing independent object, when moved between locations in the decentralized system, and when read or written to secondary storage for persistence. The dependent objects are a simple and obvious way of clustering or statically grouping [Stamo84] objects around an independent object. The expensive identification scheme is only used for the independent objects. Dependent objects are addressed relative to their enclosing independent object. Reclaiming an independent object also means reclaiming all its dependent objects.

[2] This is similar to the *maturing* of objects done in the Smalltak LOOM [Kaehl86]. Although the LOOM does not support persistence with fault tolerance it does provide a large non volatile memory.

[3] Independent objects are called "packets" and dependent objects are called "datatype values" in Avance.

We conclude that a large global identifier space is a feasible and appropriate way to manage both persistent and exported objects. Such an identification scheme can be used together with other forms of identification and addressing, for objects which are more constrained and less demanding in their use.

2.4 Subdividing the garbage collection problem

Garbage Collection methods.

A great many algorithms have been presented for doing garbage collection. Perhaps the simplest method is *reference counting* [Colli60, Cohen81]. Each object has a counter field which is incremented each time a reference to the object is created and decremented when a reference is destroyed. When the count reaches zero the object can be reclaimed. Reference counting has the advantage that it does not require application processing to stop. It has the disadvantage that it does not reclaim cyclic garbage, that it requires space for the counter in each object, and that it has unpredictable short term overhead. Removal of the last reference to a large aggregate causes a cascade of activity.

Another method is *mark and scan* [Knuth68, Cohen81]. In a first phase a traversal is made of the object-reference graph starting at a root object. Each object which is reached is marked. In a second phase the memory is scanned and every unmarked object is reclaimed. The major disadvantage with the mark and scan approach is that application processing must stop during garbage collection. It has the advantage that it has no overhead during normal operation and that it reclaims cyclic garbage. Refinements of mark & scan have been made to allow it to run in parallel with application processing [Dijks78]. Of course it then does have overhead during normal operation.

A *copying collector* [Fenic69, Cohen81] can be seen as a refinement of mark and scan. Instead of marking, objects are copied from a *from-space* to a *to-space*. Instead of scanning the memory for unmarked objects, the roles of from-space and to-space are interchanged. Copying collection compacts allocated memory as a side effect. The basic copying approach suffers from the same problem as mark & scan of requiring application processing to stop while the garbage collector is runing. Refinements of the copying approach have been made to make the copying incremental [Baker78]. Instead of having one long interruption of application processing where the complete graph is copied, several short interruptions are made and only a few objects are copied at each time.

Many Smalltalk systems use a method called *generation scavenging* for garbage collection [Ungar84, Liebe83]. This method, which is a refinement of copying garbage collection, capitalizes on the knowledge that most objects either "die young" or live very long. If an object has survived a sufficient number of cycles of the garbage collection mechanism it is moved to a separate address space called *old-space*. Objects in old-space are ignored by the regular garbage collector. The old-space is garbage collected separately and as infrequently as possible. The reason being that it will generally be very large and therefore garbage collection will take a long time and cause a lot of paging activity. On the other hand, not collecting the old-space when there is much garbage

will also slow down normal processing, because of increased paging due to fragmentation of virtual memory [Ungar88].

It is not strictly necessary that old-space (or at least not all of it) be a paging virtual memory segment. If old-space is a multilevel address space, where objects are swapped in and out of virtual memory, then the problem of the old-space fragmenting a paged virtual memory disappears. Garbage collecting the old-space can then be postponed until we need to reclaim secondary storage. Such a hybrid approach using both paging virtual memory and object swapping has been proposed by Ballard and Shirron [Balla83].

The old-space is similar to a space of persistent objects because both contain "long lived" objects. They are not exactly the same since objects are placed in the old-space not because of fault tolerance reasons but because the primary garbage collector wants not to be bothered with them. If one unifies the old-space with a persistent space then objects should be allowed to be moved to this space either prematurely by the garbage collector, or by other parts of the system (i.e. the transaction manager) for fault tolerance reasons.

Garbage Collection of Secondary Storage

Reference counting is in general not appropriate for secondary storage address spaces. An update of an object changing a reference from pointing to one object to point to another will have to update three objects instead of one [Butle87], because reference counts have to be updated in the objects pointed to. A delete will require all objects referenced by the deleted object to be updated. Although it is true that reference counting is used in the Unix file system [Ritch74], the objects (inodes) are in this case usually large and the frequency of creation and destruction of references low, compared to what one would expect for an object oriented system. Furthermore the graph does not contain cycles and the system is centralized. [4]

Copying collection is suitable for garbage collecting address spaces based on secondary storage [Butle87, Kolod89]. Because it compacts and increases locality of reference, the number of disk accesses can be reduced. This is especially true for paged virtual memory address spaces, but also for segmented and multilevel address spaces, since these also may use clustering techniques. Copying collection does fewer traversals of accessible objects than mark and scan. This also reduces the amount of paging or swapping.

As explained previously, doing parallel mark and scan or incremental copying garbage collection avoids the problem of having to interrupt application processing. But even if application processing can continue in parallel when such an algorithm is used, a significant degradation of performance will be noticed while garbage collection is being done. In a study made by Butler [Butle87] it is indicated that even the best alternative [Baker78] may increase paging by as much as 10 times.

Garbage Collection of Decentralized Systems

[4] More modern versions of Unix have "symbolic links" allowing cycles in the reference structure. But these are second class references and in fact not guaranteed to be valid. Several distributed versions of the Unix file system have also been built [Howar88, Svobo84] but they are decentralized on the granularity of a physical file system, i.e. mounted volume. Cross volume hard links are not allowed.

Most mechanisms designed for *distributed* systems should be applicable also for decentralized systems. However, if a mechanism infringes much on local autonomy, then it is probably not suitable. Using "naive" reference counting in a distributed system is certainly not appropriate, since many purely local operations (copying a reference) will require a remote access to update the counter of the object. Another problem is that remote access cannot always be guaranteed. Although this does not have to affect the safety of the mechanism, it complicates it. One of the advantages of reference counting in a centralized system is that it spreads out the overhead (not necessarily evenly) over application processing. This may turn out to be a disadvantage in a decentralized system. Executing an application at one location may degrade performance at another location and possibly at unpredictable times. This is precisely the type of problem we want to avoid when striving for autonomy. This problem may also be overcome, but it adds complexity. If cyclic garbage is to be reclaimed then we have yet another complication. Reference counting can no longer be seen as having an advantage of being simple in a distributed system.

Using "naive" mark and scan is even less appropriate for decentralized systems since it requires the whole system to stop. Using one of the parallel algorithms for mark and scan is conceivable, but any method depending on a global scan certainly infringes on autonomy. Some way of partitioning the problem into more manageable parts is needed.

Our approach

It seems to be the case, that if a system must maintain a large address (or identifier) space of objects, then the space should be partitioned, multileveled or both, and garbage collection should be done at different times, with different frequencies, and probably with different algorithms for the different partitions or layers of the address space.

We make the distinction between volatile and persistent objects and assume these reside in different address spaces. Volatile objects may become persistent by being moved to the persistent address space, but not vice versa. It is probable but not certain that an object which stays volatile will become garbage sooner than an object which has become persistent. We are not concerned with how the volatile space is garbage collected, but the persistent space could simultaneously be the old-space of a scavenging garbage collector for the volatile space.

We assume that a global decentralized and persistent identifier space, as described in section 2.3, is used for both persistent and remote object referencing. The identifier space should be thought of as pure and flat. At the very least we cannot assume that the identifiers contain any location information guaranteed to be valid. At each location in the decentralized system, identifiers are mapped to a local persistent address. We are not concerned in this paper with the details of this lookup operation. [5] We then take the approach that Moss advocates [Moss89] , of having a segmented and heterogeneous address space for persistent objects, but with the difference that we add a flat identifier layer on top of it.

As we have said, this means that any persistent object can be exported and that all

[5] It is of course crucial that the lookup operation is as fast as possible. Either hashing [Larso88] or indexing [Comer79] could be used.

exported objects also are persistent. Immediately making exported objects persistent increases the rate of generation of persistent garbage. If many exported objects are short lived, then this may cause the problem of increasing the frequency when secondary storage garbage collection is needed. We have two answers to this.

First, if it is the case that secondary storage is quickly exhausted by large amounts of garbage caused by exported objects being made persistent, then it suggests to us that there is something very strange or even wrong with either the application creating the objects, or the system structure. If objects are created, exported and then forgotten in large amounts then it seems better to communicate *values* instead of objects. If we define an object as a mapping from a fixed identity to a changing state, then a value is simply a state, without any particular identity. Values can be copied but not shared since sharing requires identity. Once a value has been communicated from one location to another, the receiver can encapsulate it in a local volatile object, if sharing is really necessary locally. If an application requires large amounts of objects shared across locations and at the same time discards the objects quickly, then the decentralized system may not been structured in the best way.

Second, since we have the additional level of indirection provided by an identifier to address mapping, we can use several different local persistent address spaces. If there are a fair amount of objects which are neither short-lived nor long-lived, then using a multilevel address space for them is probably better than a disciplined virtual memory address space. Having many garbage objects in a multilevel address space does not increase paging. Long-lived and eternal objects should be placed in a disciplined virtual memory partition since these, when they are not fragmented, are more efficient than multilevel spaces.

We admit however that there may be applications which need to create many shared and exported but short-lived objects. The addressing scheme and garbage collection mechanism we propose will not be appropriate for such applications. In the rest of the paper when we speak of garbage collection we will mean garbage collection of the persistent and decentralized identifier space.

Increased garbage on secondary storage is not the only problem resulting from making all exported objects persistent. It also reduces efficiency due to writes to secondary storage which may have been unnecessary. We believe the added overhead is usually worth it. The overhead added because of persistence should be balanced against the overhead and complexity of providing an additional garbage collection mechanism for exported volatile objects.

Our Mechanism is based on the mark and scan approach. Instead of doing a global mark and scan and requiring the whole system to stop, we do local mark and scans at each location. This means that each location has to stop application processing to do local garbage collection, but not that the whole system has to stop at the same time. Objects which have been classified as exported and objects reachable from them are not garbage collected. A second part of the mechanism is devoted to un-classifying objects as exported so that they become local again and available for local garbage collection. The mechanism does have some overhead during normal operation when messages are

sent to or from a node. There is no overhead during purely local processing.

We use mark and scan instead of a copying collector because it is more general, i.e. it is applicable to a heterogeneous address space. Our mechanism could easily be adapted to use a copying collector for the local garbage collection. In section 5.4.3 we sketch how this could be done. We use simple mark and scan instead of a parallel mark and scan because it is simpler and more efficient (but of course interferes with application processing). Simple mark and scan also has the advantage that it can be aborted at any time without any problems. We use a timestamp for marking so a new marking traversal will ignore marks from previous traversals. A parallel mark and scan collector inevitably degrades application processing because it has to cooperate with the marking traversal. An incremental copying collector also degrades performance because forwarding pointers from the old-space to the new-space have to be used until the copying is finished. These problems are greater for secondary storage address spaces because accessing to secondary storage is more costly. In section 5.4.3 we also sketch how a a parallel or incremental collector could fit in our scheme.

3 System model

We first give a short description of our system model and a list of assumptions. Following this there is a section with more detail on the reasons behind the assumptions.

3.1 Summary of assumptions

We model the decentralized system by a set of *nodes*, where each node contains persistent *objects* and persistent *references* to objects. Both objects, and references to objects, may be moved between nodes. Each node has one or more *root objects* which are reachable by definition and which may not be moved. This is quite a general model and many existing systems fit this model [Lisko83, Shriv88, Marqu88, Björn88]. On top of this we make the assumptions listed below. They simplify the system and the garbage collection problem. We believe they are reasonable assumptions. In other words the decentralized system and the garbage collection problem are not made trivial by the assumptions.

- Users may only access objects by using references.
- Only the system may create or copy references.
- All references are globally valid.
- Objects or references do not get lost as a result of failures in communication or of nodes.
- Messages between nodes can be inspected by the system and references distinguished from other data.
- An object resides at one node at a time.

- Objects have a timestamp associated with them indicating when they were last updated.
- If new nodes are added to the system, then all existing nodes must be informed before distributed garbage collection takes place.
- It is always possible to distinguish a reference to a local object from a reference to a remote object, at least by doing a local lookup (dereferencing) of the object.
- All nodes can stop normal processing at *some time* which does not mean that all nodes must be able to do this at the *same time*.
- All objects, which have references to them from other nodes than the one where they are located, are persistent.

3.2 Details of the system model

Nodes are the autonomous locations of a decentralized system. Physical distribution is not essential for decentralization, but we assume that in practice it will be the case. A consequence of this is that inter node messages are assumed to cost much more than messages internal to a node. Autonomy is a relative term. We use this term to emphasize that a major goal of the system architecture is to maximize the possible independence between nodes. But independence is certainly not the only goal. Another major goal is to support a great degree of interoperability between nodes. In other words, it should be as easy as possible to write applications which utilize the resources (objects) at several nodes. The architecture should guarantee that such applications can be executed without causing inconsistency, and also that they will make progress.

Object Based Systems

The interoperability requirement means that the nodes must compromise some of their autonomy to be able to cooperate. Not only do we require that all nodes be able to communicate with each other, we assume that all nodes run a homogeneous software platform called an *object manager*. Its main function is to manage the resources of a node and to cooperate with other nodes. Garbage collection is one of its functions. Although this software layer is homogeneous the hardware need not be. In fact one of the goals of the object manager is to insulate applications from hardware dependencies. The object manager can be compared to an operating system. It makes each node into a virtual machine. The reader can therefore think of nodes as physical machines in a distributed system. It should be remembered however that that nodes are logical entities. We want it to be relatively simple to move a node from one physical machine host to another and this is reflected in our use of a global object identification scheme. In the rest of the paper when we speak of "the system" this will mean either the object manager of one node, or the object managers of all nodes working in cooperation.

We assume an *object based* data model. In other words resources are objects which are instances of abstract data types. To access an object, an application needs a *reference* to that object. References are capabilities in the "operating systems" sense [Levy84].

Thus the application/user is not allowed to use a reference in any arbitrary way. What is important in this paper is that references are the only way applications can access objects and that references are controlled by the object manager(s). In particular the system can distinguish between references and objects, and references may not be forged or copied.

The object manager must be well behaved in the sense that it does not reconnect garbage. Any object which has become garbage will stay garbage until collected by our mechanism. This could be said to be one correctness invariant for the object manager. We make use of this invariant to avoid the need for exact synchronization in the distributed mechanism.

The object manager of a node maintains a secondary storage repository (or "database") for the persistent objects located at that node. These objects generally contain references, some of which refer to other local objects in the repository and some of which refer to objects at other nodes. All the objects of the system and the references between them, form a large directed graph. References to remote objects are then edges comming from an object in one node and entering an object at another node. We assume that each node has at least one local persistent *root* object. A root object is any object which should never be reclaimed by the object manager.

We assume that references to objects are *globally valid*, both in space and time. This means we assume strong identity [Khosh86] for persistent objects and if a reference is communicated from one node to another then the receiving node can use the reference to operate on the object just as if the object resided locally. The object managers in cooperation make it possible to achieve both location transparency (under the autonomy restrictions particular to the nodes) and location visibility when this is desired. Given a reference, the interpretation of that reference (the object it refers to) is independent of location context. Hence references, and objects which may contain references, can be passed between nodes without any semantic problems. We also assume that a reference points to at most one object. A reference although not necessarily the same thing as an identifier [Stein89], can for the purposes of this paper be thought of as simply containing an object identifier.

We assume a transaction based model like that of Argus [Lisko83] or Camelot [Spect87] and that operations on remote objects are done using remote procedure calls [Spect82]. The transaction mechanism and the rpc communication mechanism are both part of the object manager. We assume that the object manager can inspect incoming and outgoing messages and distinguish references from other data. The object manager has two "modes" of operation. During *normal processing* transactions may be processed. During a *quiescent state* there may be no transactions in progress. Although the quiescent state is not necessarily static in the sense that nothing is happening at the node, it is truly a state from the perspective of transaction management. When a node is about to enter the quiescent state, the object manager refuses to accept new transactions and waits until all locally ongoing transactions have either committed or aborted. During the quiescent state, the local repository will be in a consistent state. We can ignore consistency problems which could be caused by communications or node failures. Such problems are assumed to be taken care of by the transaction mechanism. Parts of the

garbage collection processing is done in the quiescent state. Other tasks such, such as reorganization and compaction of the repository may also be done in the quiescent state. All such operations must maintain the consistency of the repository. We assume that all nodes on a regular basis are able to enter the quiescent state. This does not mean that all nodes must be able to do this at the same time.

When an object is created by some transaction it is initially *volatile*. The object becomes persistent if a reference to it is inserted in an already persistent object by some operation and the transaction in which the operation participates commits. We also assume that all objects which are known by more than one node are persistent. This means that if a reference to an object is exported, then the object must become persistent if it was not already.

The existence of both volatile and persistent objects complicates the global object-reference graph. We defined a node being in the quiescent state as there not being any transactions in progress locally. We need in fact to have a stronger definition of the quiescent state than this. During the quiescent state we require also that there exist no local volatile objects referencing persistent objects. Otherwise we could have persistent objects which are not reachable from any persistent root yet reachable from a volatile object. Such an object could be used to reconnect persistent objects to the rooted graph in a transaction after the quiescent state. If persistent threads are allowed then we could have suspended threads which survive through the quiescent period. All such threads must then be regarded as root objects.

The object manager may use *stubs* to represent remote objects. An object stub is not an object. It should be seen more as an object header, invisible to applications, and used by the object manager for bookkeeping purposes. For example it could contain information hinting at the real object's location. There is an important difference between stubs in our system model and proxys in many of the implementations of distributed Smalltalk [Decou86, Benne87, Schel88]. A proxy is reached using a local address and encapsulates the location and address (or identifier) of the remote object. A stub in our model is reached using *the same global identifier* as the real object. We do not require that stubs always exist for all remote objects referenced by a node, because this would cause much overhead. But if and when stubs are used then our garbage collection mechanism will take advantage of their existence and will also perform garbage collection of them. We assume that it is always possible to distinguish remote objects from local, even if there are no stubs for some or all of the remote objects. When the object manager does a lookup using a reference it will either hit a local object, a stub representing a remote object, or it will fault in which case all the object manager knows is that the object does not reside locally. The use of stubs in certain parts of our mechanism should not be seen as overhead caused by the garbage collection mechanism.

An object resides at one node at a time. Objects may move from one node to another as part of a transaction. Replicas of an object could also be allowed as long as the location of the original is well defined and updates are coordinated by that node. Replicas may be seen as just extended stubs.

Each object contains a timestamp indicating when, in terms of local host time, the

object was last updated. This is not a very strong assumption. Most general computer systems today have a system clock accessible to applications. Note also that we only require updates to be time-stamped, not read accesses, and that no synchronization of clocks between nodes is required.

We assume that each node knows of the existence of all other nodes. New nodes may be added dynamically to the system, but our mechanism for detecting distributed garbage will only work properly when all other nodes have been informed of the new nodes existence. [6]

4 Requirements on the garbage collection mechanism

We say that an object **x** is *reachable* from an object **y** if either **y** contains a reference to **x** or if **x** is reachable from another object **z** and **y** contains a reference to **z**. An object is said to be globally reachable if it is reachable from a root object on some node. Objects which are not globally reachable are called *garbage*. We may also say that an object is locally reachable with respect to some node, if the object is reachable from a root object at that node using only the objects and references known at that node. Note that an object which is locally reachable with respect to a node does not have to reside at the node, although some object at the node must have a direct reference to it.

The purpose of the garbage collection mechanism is to detect garbage and to remove it, i.e. reclaim the resources occupied by the garbage. In our case the resource is mainly space on secondary storage. There are two correctness criteria which are required from a garbage collection mechanism [Manci88]. [7] The first is that any object which is globally reachable must not be classified as garbage and collected. This may be called the *safety property*. The second is that any object which is not globally reachable, i.e. garbage, should be identified as garbage within some definite and specified upper bound in terms of time or processing. In our case the upper bound will be "two cycles of the collection mechanism." This may be called the *liveness property*. Among other things this means that cyclic garbage whether distributed or not must be detected.

The garbage collection mechanism we propose should satisfy the two criteria. This does not mean that the object manager taken in a broader perspective has to satisfy them. Both of the properties can be relaxed by the object manager. Relaxing the liveness property means that one is willing to accept that some garbage might never be collected, at least by this mechanism. Relaxing the safety property is only acceptable if object identifiers are not reused, so that at least it is possible to detect a reference to a nonexistent object. Either of these or both allow for reducing the execution cost of the mechanism and for increased autonomy of nodes. We will return to this possibility in

[6] New nodes should not be added to the system during phases 5-7 of the garbage collection mechanism (see section 5.2.2).

[7] Note that the definition of these properties in [Manci88] is in terms of reference counts. Since we are using a mark and scan scheme instead of a reference counting scheme, our definition is in terms of reachability.

section 5.4.

In addition to the two correctness criteria there are some properties which we desire the mechanism to have.

1. It should be possible to detect some garbage on a purely local basis.
2. It must allow the disruptive quiescent state to be entered at different times for different nodes.
3. It should have as little overhead as possible during normal operation.
4. It should require as few messages as possible.
5. It should be possible to turn it off completely for some time and then resume, without violating the two correctness criteria.
6. It should only try to collect garbage residing locally.
7. Objects must be able to move from one node to another.

5 A garbage collection mechanism

We will first give an outline of the mechanism. We will then explain the mechanism in detail under two simplifying assumptions:

1. Our mechanism has not been turned off. In other words requirement 5 has not been exercised.
2. All nodes go into the quiescent state and do garbage collection processing *at the same global real time*. In other words requirement 2 is not enforced.

These are not assumptions we want to maintain. They are only made to simplify the explanation of our mechanism. After this we describe the modification to the mechanism so that we can drop the assumptions.

5.1 An outline of the mechanism

The general idea of the mechanism is as follows. Each node does its own local search for garbage, producing an initial set of candidates for collection consisting of objects which are not locally reachable. Each node also maintains some information making it possible to locally identify a subset of the set of candidates which are globally unreachable. This subset can be reclaimed without consulting other nodes. The remainder of the set consists of objects which are not locally reachable, but which *may* be globally reachable. This first part of the mechanism and the garbage collection made possible by it, is done whenever the administrator of the node finds it convenient, independently of other nodes. This is

one of the desirable properties mentioned earlier (1). It is then possible for a node to do local garbage collection to free resources as often as needed.

The next step is the generation of the *external reachability message* or ERM for short. This message contains two parts:

1. The references to objects at other nodes which are locally reachable on this node.
2. A set of pairs of references: The first being a reference (entry point) to a local object which is **not** locally reachable but may be globally reachable. The second being a reference to an object at another node which is **not** locally reachable, but is reachable from the first object.

The ERM contains all information concerned with which objects outside of the node are referenced from the node. The first part contains references rooted at the node, the second part paths "passing through" the node.

The ERM of a node is made available (broadcast in one way or other) to all other nodes. This means that, after a while a node will have received the ERM of all other nodes. When this is the case, a node has all information it needs to obtain a view of the global object-reference graph, except for one problem. The problem is due to the movement of references during the construction of the ERM's of the other nodes. Since the construction of the ERMs is not synchronized, references (or objects) may move in a way so that they are not reflected in any ERM.

We solve this problem by obtaining information from each node on which references have moved since the ERM of that node was generated. This information we call the moved references message (MRM). The MRM of *each* node must be collected after the ERMs have been received from *all* nodes. This is how synchronization is achieved in the algorithm. We know that all the ERMs were made before any of the MRMs This has the effect that any reference which has moved will appear in some MRM. The MRM is also exchanged between all nodes. Any local object having a reference in the MRM from some node will not be reclaimed. Since garbage does not move it will never appear in an MRM. When both the ERMs and the MRMs have been received from all other nodes then each node has enough information to detect objects which have been globally unreachable since just before the generation of the ERM. Any such objects residing locally are made available for reclamation in the local garbage collection part of the next cycle of the mechanism.

5.2 The mechanism in detail

Having the perspective of one node, let us call the set of all objects which either reside in the local repository or which reside at some other node but have a reference to them in the local repository, for **Known**, (see figure 1). This set in general includes garbage objects.

Our mechanism needs to partition **Known** into three subsets: **Local**, **Export** and **Import**. The **Import** set is the set of objects not located at this node but which this

node has references to. The **Export** set is the set of objects located at this node which could be *directly* referenced from other nodes. In other words at some time, for each object in **Export**, a message containing a reference to that object has been sent to another node. The **Local** set is the set of objects located at this node which cannot be directly referenced from other nodes. The idea of keeping track of which objects are exported and imported we have obtained from Mancini and Shrivastava [Manci88].

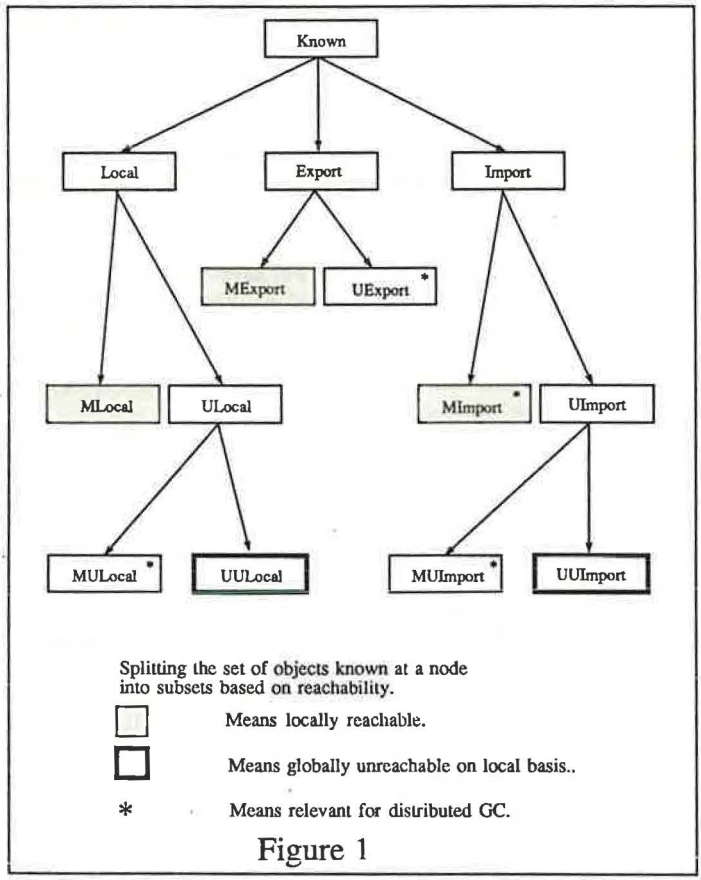

Figure 1

It should be clear to the reader that **Local**, **Export** and **Import** are disjoint. The **Import** set can easily be identified during the first processing phase of our mechanism. If stubs are maintained for normal processing then the set consists of all stubs. Otherwise we can assume that any reference which cannot be resolved locally is a reference to an object located elsewhere, and hence that object is a member of **Import**.

To differentiate between the **Local** and **Export** sets requires additional information. If we simply allow any messages containing references to pass out to other nodes without control, then **Export** becomes identical to all objects residing at the node, and **Local** becomes the empty set. The advantage of allowing this is that the node can operate without any overhead (for garbage collection of persistent objects) during normal operation. There are two disadvantages: First, and most importantly, we can do very little local garbage collection of persistent objects without consulting all other nodes. Only stubs which are not reachable from any other object residing at the node can be removed (the **UUImport** set, explained below). Second, the ERM message, in particular the second part, can become much larger. This is because now all local objects which are not locally reachable must be considered as potentially globally reachable. Still, this solution might be preferable in some systems, where it is desired to keep the overhead during normal operation at a minimum. Our mechanism is also consistent with this possibility, it simply means that **Local** is empty. One can achieve a greater degree of autonomy with respect to garbage collection, at the cost of some overhead during normal operation.

Given that we can distinguish between **Export**, **Import** and **Local**, our mechanism works by further subdividing these sets. First a marking is done from the local persistent roots, splitting the three sets into six sets, (see figure 1). Then a second marking is done using the members of **Export** which were not marked by the first marking (**UExport**) as roots. This second marking splits the other two sets of objects unmarked by the first marking (**ULocal** and **UImport**) into four sets. The identification of these subsets is the basis of our garbage collection mechanism.

5.2.1 The Export-Import list

As in [Manci88] we keep track of which objects have references to them exported or imported. While Mancini and Shrivastava have two lists we only have one list, but information is kept with each entry in the list as to whether it represents an import or export of an object. We will call this list the EI-list. The object managers maintain the EI-list during normal operation.

Every outgoing or incoming message has to be checked for references not already in the list. When such references are found they have to be added to the list. Each EI-list entry consists of three parts. The reference (identifier) itself, which is used as the lookup key; a timestamp showing when the reference was added to the list; and an export/import indicator which indicates whether the reference points to a member of **Export**, **Import** or "not known."

Because we are talking about persistent objects and persistent references, this list also has to be maintained persistently. Furthermore, some operations which previously were pure functions (side effect free) may now have side effects.

The maintenance of the EI-list interacts with the distributed transaction mechanism of the system. Since the maintenance of the EI-list is done by and for the object manager, we do not require full update atomicity. We do not require aborted transactions to backout effects on the EI-lists. If messages which are part of an aborted transaction have

caused some additions to the EI-lists of one or more nodes, then this is not a consistency problem. It just means that we are overly conservative and that some garbage may not be recognized as such as early as was otherwise possible. The garbage collection mechanism regularly purges the EI-list. This will not only clean up the effects of aborted transactions, but more generally remove references which might have been exported but which are in fact not referenced by other nodes any more.

On the other hand we do require that for a transaction to commit, any effects it has had on EI-lists have been made non-volatile. We will return to the issue of how to efficiently implement the EI-list in section 5.4.1.

Since both incoming and outgoing messages are scanned, and since the EI-list is maintained as one list, imported references which are reexported will be found already on the list. The same holds for exported references which are reimported. We want to avoid doing actual object lookups when a reference is to be added to the EI-list during normal operation. This might be needed to determine if the reference was to an imported or exported object. Instead we assume that if a reference is added to the EI-list as part of an outgoing message then it points to an object in **Export** and set the export/import indicator of the EI-list entry to be added to "export." For references added as part of incoming messages we set the indicator to "import." The export/import indicators in the EI-list will be consistent with **Export** and **Import** as long as objects do not move and if the EI-list is continuously maintained, i.e. requirement 5 is not exercised. The last is one of the simplifying assumptions which we remove later (see section 5.3.1). The movement of objects is handled by setting the indicator to "unknown" at both the source and destination nodes. Moving an object from one node to another is an operation which like other application operations must be part of a transaction. Again, this means that an aborted move will not undo the "unknown" setting of the export/import indicator, but a committed move must have made this setting persistent.

Figure 2 shows a system consisting of four nodes and a number of objects distributed over the nodes and referencing each other. Root objects have a reference (arrow) from the node border. Taking node 3 (N3) as the basis we see that **Export** consists of 3, 4 and 7. These objects are referenced from other nodes. **Import** consists of 2 10 12 and 14 since objects at N3 have references to these objects on other nodes. We can assume that these objects are represented by stubs on N3. In the figure, the small circles on the border of a node represent stubs. **Local** will then consist of 5, 6, 8 and 9.

5.2.2 Processing for secondary storage garbage collection.

Besides the maintenance of an EI-list during normal processing, the garbage collection mechanism consists of seven phases. The first three phases need to be run when a node is in the quiescent state. Phases 4-7 could all be run in parallel with normal operation, but only phase 5 needs to be run in parallel. The administrator of a node then has some freedom of choice on whether to minimize the time of interruption of normal operation that the quiescent state causes, or to minimize the overhead of the garbage collection mechanism during normal operation.

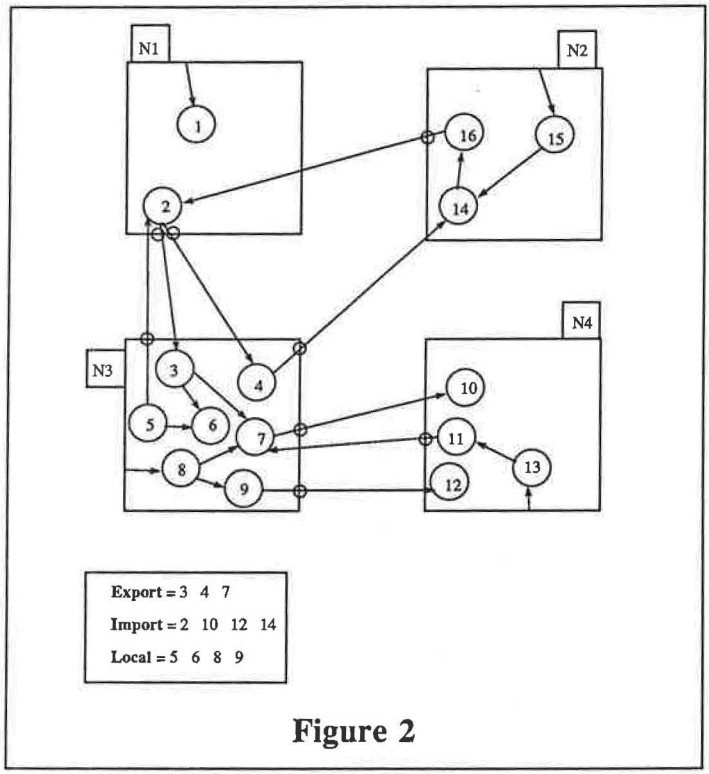

Figure 2

Phase 1: Marking from roots

This phase is executed when the node is in a quiescent state. It consists of the marking phase of the standard mark & scan. We assume that the repository and the EI-list of the object manager is not being modified by anything but the marking traversal. Starting a traversal at the local root object(s), all objects reachable from the roots are marked. The traversal stops at remote references, i.e. only a local traversal is made. We use the update timestamp field of each object for marking and use the time when phase 1 started as the marking. When a remote reference is found, the stub is marked if it exists. In addition, the remote reference is looked up in the EI-list and its timestamp is also marked, (if the export/import indicator indicates "unknown" it is set to "import").

The three sets **Export**, **Import** and **Local** are each split in two (see figure 1). The objects which are marked by this first phase are trivially globally reachable since they are locally reachable. Any garbage located at this node must be a member of one of the three sets of unmarked objects **UExport**, **UImport** and **ULocal**. Figure 3 illustrates the effect of the first phase on our example.

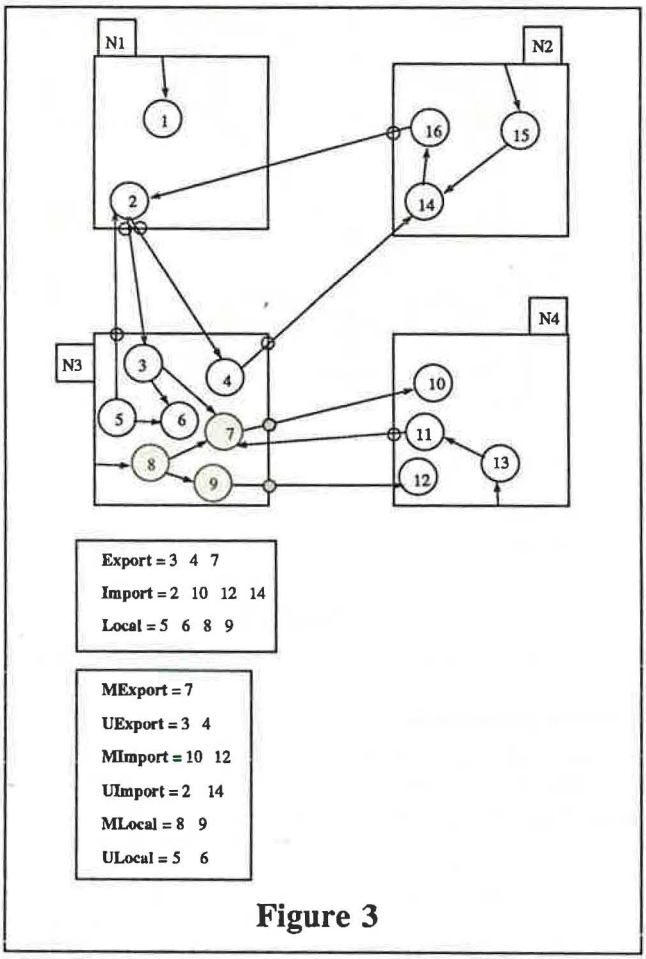
N1
N2
N3
N4
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Export = 3 4 7
Import = 2 10 12 14
Local = 5 6 8 9
MExport = 7
UExport = 3 4
MImport = 10 12
UImport = 2 14
MLocal = 8 9
ULocal = 5 6

Figure 3

Phase 2: Marking from members of UExport

The second phase is also run during the quiescent state of the node. The EI-list is scanned at the same time as it is updated. Each reference will be either unmarked (have an old timestamp), be marked by phase 1, or be marked by this phase (phase 2). Only remote (imported) references in the EI-list are marked by either phase 1 or 2. Different actions are taken for each EI-list entry depending on its contents.

For entries found with the import/export indicator set to "unknown" an object lookup is made to determine the proper setting. Processing then continues according to whether the setting is "export" or "import."

An exported reference is looked up to see if the object in the repository was marked or not in phase 1. If the exported object is marked then it is a member of **MExport** and the EI-list entry is skipped, i.e. processing continues with the next EI-list entry. If the exported object was not marked then the object is a member of **UExport** and it is used (immediately before continuing the scan of the EI-list) as the root for a marking traversal. This marking traversal starts by marking the exported object, and not the reference in the EI-list.

Imported references found during the scan of the EI-list are not looked up in the repository. Imported references marked in phase 1 correspond to **MImport** and are appended to a list which we will call ERM-1. This list will contain all references to remote objects rooted at this node. Imported references which are unmarked correspond to **UImport** and are skipped. Imported references marked by this phase correspond to the **MUImport** subset of **UImport** and are also skipped.

The marking traversals in this phase start from the members of **UExport** as they are found during the scan of the EI-list. Each such traversal uses a new timestamp which we assume is different and later than both the mark of phase 1, and marks obtained for traversals for previously found members of **UExport**.

The marking traversals in phase 2, like the traversal in phase 1, stops at remote references and at objects already marked in phase 1. It does **not** stop at objects marked by previous traversals of phase 2. This is why each traversal in phase 2 must obtain a new timestamp, otherwise we would risk that the traversal did not terminate. When remote references are found during a traversal in the repository, their entry in the EI-list is looked up and marked if it was not already marked in phase 1 or 2. Since only remote references in the EI-list which were not marked by phase 1 are marked by phase 2, they correspond to the **MUImport** subset of **UImport** (see fig 1).

This second marking divides **ULocal** into **MULocal** and **UULocal** (in the repository), and **UImport** into **MUImport** and **UUImport** (in the EI-list). Figure 4 illustrates the result of the second phase. Marking in this case started from 3 and 4.

During the traversals of phase 2, whenever a member of **MUImport** is reached, i.e. a remote reference in the EI-list which is not marked by phase 1, then a pair of references is added to a new list which we will call ERM-2. The pair consists of a reference to the member of **UExport** which the current traversal started with, and a reference corresponding to the member of **MUImport** which has been reached. The

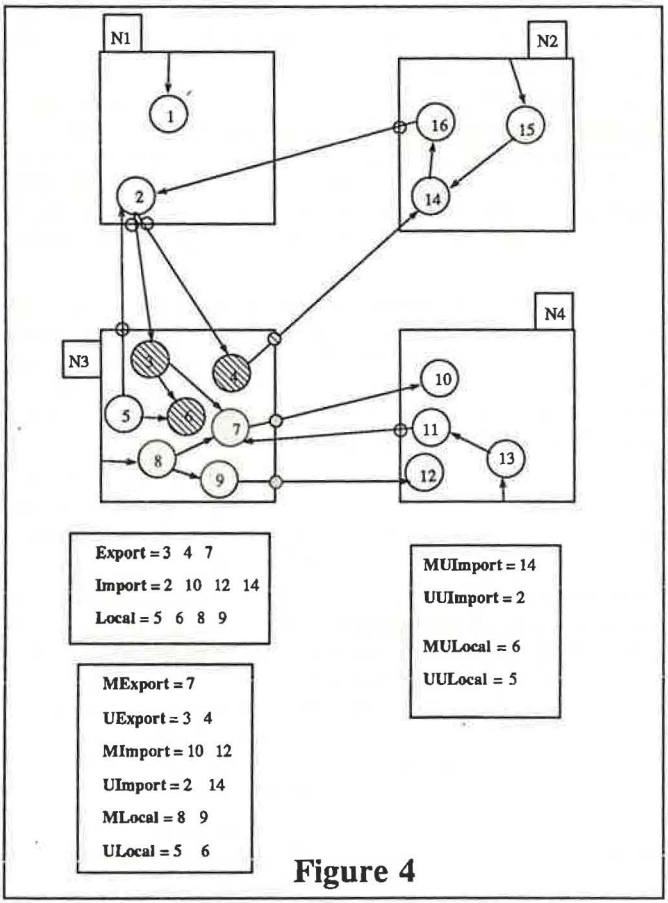
N1
N2
N3
N4
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Export = 3 4 7
Import = 2 10 12 14
Local = 5 6 8 9
MExport = 7
UExport = 3 4
MImport = 10 12
UImport = 2 14
MLocal = 8 9
ULocal = 5 6
MUImport = 14
UUImport = 2
MULocal = 6
UULocal = 5

Figure 4

ERM-2 list will consist of all "paths" through the node which are not rooted at the node. Both the ERM-1 and ERM-2 lists are saved to secondary storage once phase 2 is completed.

When the scan of the EI-list has completed then:

> All entries with an export/import indication of "unknown" have been corrected to either "export" or "import."
>
> All "import" entries marked by phase 1 can be found in ERM-1
>
> All "import" entries marked by phase 2 can be found as the second component in at least one entry-exit pair appended to ERM-2.
>
> No "export" entries were marked by either phase 1 or 2.

It should be obvious that the members of **UULocal** are garbage. They are obviously not locally reachable since they were not marked in the first phase. They are not reachable from any other node either and hence not globally reachable, since anything reachable from another node has to be reachable from an exported object, and should then have been marked in the second phase. **UUImport** can also be used to reclaim resources, namely stubs and EI-list entries. Members of **UUImport** are remote objects which currently cannot be reached by or through this node and hence all information related to them can be removed from the node. However this has to be done (in the next phase) before resumption of normal processing. The reason is because the members of **UUImport** do not reside at this node and so they might actually be reachable from other nodes and could become reachable again at this node once normal processing starts. A removal of stubs and references in the EI-list corresponding to **UUImport** could therefore conflict with operations during normal processing.

Phase 3: Collecting UUImport references from the EI-list

The third phase is also run in the quiescent state. The EI-list is scanned a second time to remove references to **UUImport**. Any reference with the export-import indicator set to "import" and with a timestamp from before phase 1 can be deleted.

For each reference to **UUImport** deleted from the EI-list, the corresponding stub (if there is one) in the repository is either deleted now or made into a "tombstone" for delete in phase 4. [8] In the example (figure 4) the EI-list entry and stub for the object 2 in **UUImport** can be collected. It is possible to skip this phase without violating the liveness requirement. This phase does not remove garbage. It only removes remote references which are not currently used locally. This will reduce the size of the EI-list. References in the EI-list to garbage will be removed in phase 7.

Phase 4: Collecting UULocal

This phase can be described as the scan phase of the standard mark and scan algorithm. An exhaustive search of the repository is made looking for objects having a timestamp

[8] If the scan in phase 4 is done during the quiescent state then stubs need not be marked as tombstones.

from before the mark of phase 1. Members of **UULocal** (and possibly tombstone stubs in **UUImport**) have an old update stamp associated with the object in the repository.

If the node goes into normal operation after phase 3 has completed, but before phase 4 (this phase) has completed, then we have to be careful not to regard new or moved objects as garbage. Objects created, updated or moved to the node after the quiescent state will have a timestamp after that of phase 2 and so will not be reclaimed.

We have thus identified some garbage on a purely local basis, without consulting other nodes. This phase could be run either directly after the other three during the quiescent state of the node, or in parallel with normal operation. Since timestamps are used as the marking "colors," phase 3 could even be run in parallel with phases 1 and 2 of later cycles of the mechanism. [9]

The reason that phase 4 can be run in parallel with normal operation is the axiom that says: "garbage stays garbage." Anything which has become globally unreachable cannot be reconnected if the object manager is working correctly.

In the example (figure 4) the object 5 in **UULocal** can be collected.

Phases 1 to 4 could be cycled several times without the following phases as need arises to reclaim resources.

The following phases are all concerned with detecting garbage on a global basis and purging the EI-list so that the garbage becomes locally identifiable and collected in phase 4.

Phase 5: Creating and exchanging the ERM

The ERM consists of two parts ERM-1 and ERM-2, both generated in phase 2. Phase 5 consists of the exchange of the ERMs between all nodes. This node's ERM should be communicated to all other nodes and the ERMs of all other nodes should be received by this node, sooner or later. This phase could start when the node is in the quiescent state, but will normally need to continue during normal operation of the node. The reason being that it may take a long time before the ERMs from all other nodes can be obtained.

Because our mechanism is cyclic there is a need to be able to identify which cycle a particular ERM belongs to. We therefore assume that an ERM generation number is maintained at all nodes and that this number is incremented and added to the ERM when it is generated. Only phase 5 increments the generation number. Several cycles of phases 1-4 are possible in the same ERM generation. Once a node has communicated its own ERM of a certain generation to any other node then it has committed that generation. If it later sends an ERM with the same generation number to any other node, then it must be the same ERM.

The ERMs received from the other nodes will usually contain some references to objects located at other nodes and unknown to this node. The receiving node does

[9] Running phase 3 in parallel with phase 3 of the next cycle of the mechanism is pointless and could cause consistency problems depending on the storage organization. Phase 3 of a later cycle will reclaim any garbage which should have been reclaimed by phase 3 of an earlier cycle.

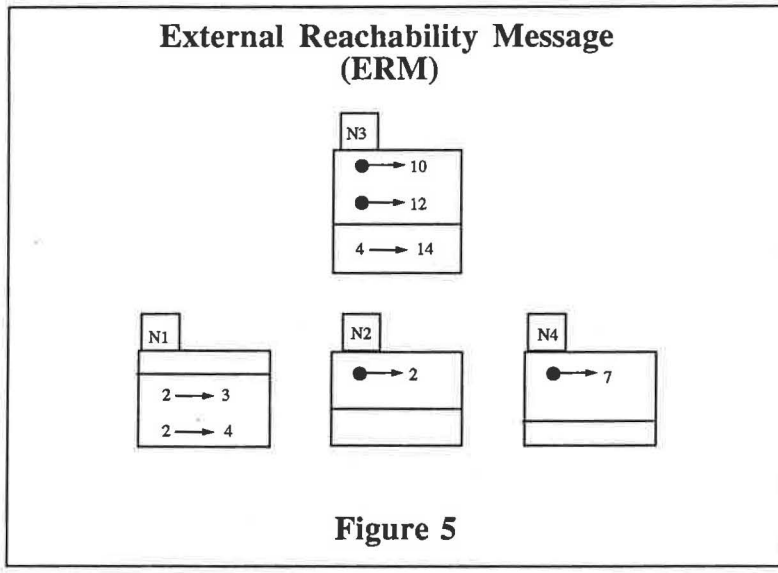

Figure 5

not attempt to produce entries in the EI-list (or stubs) for new references found in the ERMs. The object manager recognizes the ERMs as being special and not part of normal processing. Although it would not cause inconsistency if new references imported with ERMs are added to the local EI-list, it could cause the EI-list to be unnecessarily large.

Phase 6: Traversing the global graph

Once a node has received all the ERMs of the latest generation from all other nodes it has sufficient information to obtain a view of the global graph. We remind of the temporary simplifying assumption that all nodes entered the quiescent state and did phases 1-3 at the same time. Synchronizing the generation of the ERMs in this way has the effect that all the ERMs taken together reflect a consistent global state of the system.

Figure 5 shows the ERMs produced at the four nodes in the example. When the node N3 has received the ERMs from N1, N2 and N4 it can reconstruct a view of the global object-reference graph. It is not a complete view of the global graph since only members of **Export** and **Import** are visible. Figure 6 illustrates how the global graph appears when constructed from the ERMs.

A third marking traversal is now made. This traversal does not actually traverse objects in the repository. Only the ERMs, or a graph structure constructed from them, is used.

The traversal starts with the members of the ERM-1 parts of all ERMs as roots and continues using the paths found in the ERM-2 parts. The traversal stops when marked entries are found or when no more paths with a matching entry reference are found in

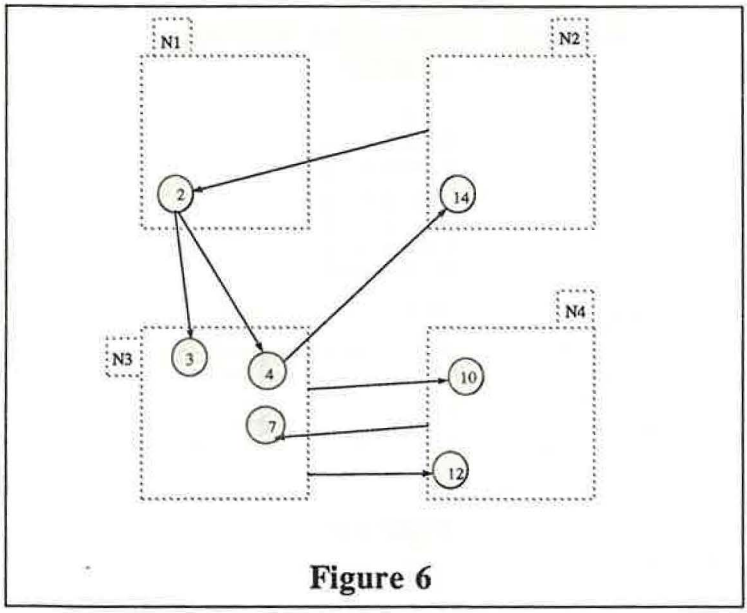

Figure 6

the ERM-2s.

Looking at figure 5 and 6. The traversal starts with objects 2, 7, 10 and 12 since these are the references present in the ERM-1s. The traversal from 2 will mark the entries 2,3,4 and 14. Traversals starting with 7,10 and 12 will only mark these. In the example, no global garbage is detected.

If we go back to figure 4 and assume that there was no reference from object 15 to 14 at node N2, then we see that objects 14, 16, 2, 3, 6 and 4 now become garbage. This would not have affected phases 1-4. In phase 5 the altered ERM received from node N2 is shown in the top of figure 7. In other words, instead of a rooted reference to 2 in the ERM-1, we get a path 14 -> 2 in ERM-2. Figure 7 also shows the global view given by the ERMs.

Phase 7: Removing references to garbage from the EI-list

This phase is similar to phase 3 but can be run concurrently with normal processing because we are only removing garbage Any reference which according to phase 6 is garbage is looked up in the EI-list and removed. If the reference is remote then a lookup is made to see if it has a stub. If there is a stub it is made into a tombstone for removal in phase 4 of the next cycle.

In the modified example shown in figure 7, the members of **Export** 3 and 4 at N3 can have their entries cleared from the EI-list. This effectively moves them from **Export**

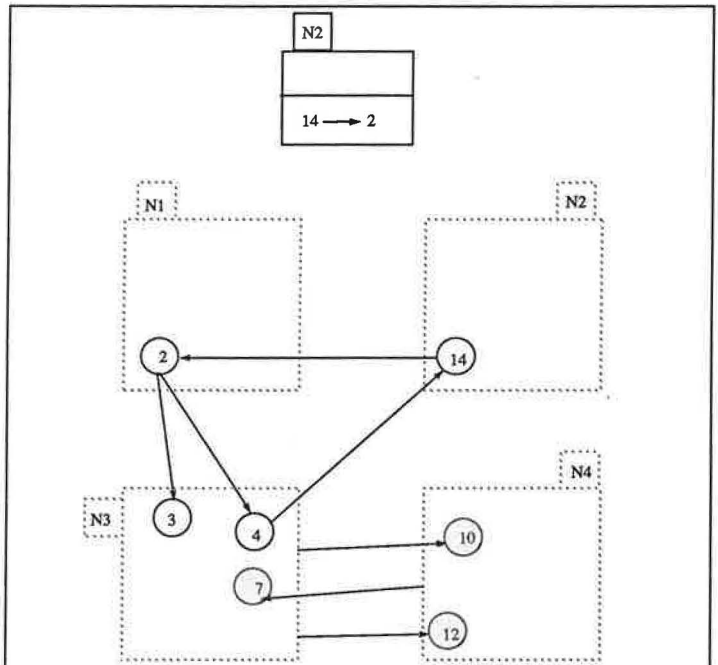

Figure 7

to **Local** and they will be collected as members of **UULocal** in the next cycle, (in fact the local object 6 will also be collected). The member of **Import** 14 is also removed from the EI-list. If phase 3 was skipped then the entry and stub for 2 will be removed now instead.

5.3 Dropping the simplifying assumptions

5.3.1 Interrupting the maintenance of the EI-list

The maintenance of the EI-list is the only major overhead of our mechanism during normal operation. Overhead is only incurred when messages containing references are being exchanged between nodes. It might be desired to turn of the maintenance of the EI-list for some time to have greater performance. If the EI-list is not maintained while messages are sent and received from a node, then references may be exported and imported without being recorded in the EI-list. If no other information is kept then this means that

we must regard all objects residing at the node as members of **Export**. The EI-list must also be regenerated from the repository. If the IE-list maintenance is turned off then it is advisable that at least stubs are maintained for all remote references. Otherwise the regeneration of the EI-list becomes prohibitively expensive, if the repository is large. So, even if our mechanism does not depend on the maintenance of stubs as part of normal processing, if requirement 5 is to be exercised then stubs are needed.

Phase 0: Regenerating the EI-list

This phase can be run in parallel with normal operation. This phase is only necessary if the maintenance of the EI-list has been turned of for a period of time when messages which may have contained references have been exchanged. We assume that the maintenance of the EI-list has been turned on again. Any references added to the EI-list as a consequence of message traffic, after maintenance has been turned on but before phase 0 has completed will get the export/import marking set to "unknown." Such entries will be corrected in phase 2. Phase 0 must be completed before phase 1 can start.

A complete (read-only) scan is made of all objects in the repository. All objects found are looked up in the EI-list using the object identifier. If an entry for a reference to the object does not exist then an entry is made to the EI-list and marked as "export." If it is certain that stubs have been added for all references which have been imported while the EI-list was not maintained, then lookups in the EI-list are also made for all stubs found in the repository. If an entry for a reference to the stub does not exist then an entry is made to the EI-list and marked as "import." If stubs have not been created for all new references imported while the EI-list was not maintained, then *every reference of every object* has to be checked to see if it is a remote or local reference, and if remote looked up in the EI-list and possibly added to it.

5.3.2 Dropping the synchronized start assumption

The synchronized start assumption ensures that the ERMs reflect a consistent static snapshot of the system. If we drop this assumption, in other words if the nodes enter the quiescent state and execute phase 1-3 at different times, then we need to cope with the problem of references "moving" between nodes. By a reference moving from one node to another we really mean that the reference is copied to a node where the object was previously not imported, and deleted at a node where it previously was imported. We note that moving objects is not a problem, as was explained for phase 4. Any moved object will have a new timestamp at the receiving node and so will not be collected by phase 4. Of course an object which moves may move references, but this then reduces to the moving reference problem.

The following small example illustrates the problem. Assume that some object **x** is reachable from some node **A** and not reachable from another node **B**. The object **x** could reside at any node in the system including **A** itself. Some other node **B** enters the quiescent state and generates its ERM. After **B** resumes normal operation, a transaction is executed which makes **x** reachable from **B** but unreachable from **A**. Now **A** enters the quiescent state and generates its ERM. We now have a situation where **x** is reachable

but this is not reflected in either of the ERMs. The ERM of **B** does not reflect it because it was generated before the reference to **x** was received. The ERM of **A** does not reflect it because it was generated after the reference to **x** was removed. If we do garbage collection based on only these ERMs then **x** may be removed despite the fact that it is reachable, violating the safety property.

Our solution requires a slight modification of how the EI-list is maintained. In section 4.2.1 it was said that only references not found already on the list were added to the list. We now modify the behavior in the following way: During phase 4-5, i.e. after the quiescent state and until the end of phase 5, *outgoing* messages not only add new references to the EI-list, but any reference already on the list with a timestamp before phase 1 will get the timestamp updated to current time.

The effect of this is that we keep track of which references have passed out from the node since normal processing resumed. All nodes do this until the ERM has been received from all other nodes. This ensures that any reference moved after any of the ERMs was generated must be recorded in some MRM, because it must have been part of an outgoing message at some node. We split phase 5 in two and change phases 6 and 7. Phase 5a is the same as phase 5 previously, (exchanging the ERM). Phase 5b is as follows:

Phase 5b: exchanging the MRM

Once the ERMs of all other nodes have been received by a node (phase 5a finished at the node) it can start phase 5b. A scan of the EI-list is done (in parallel with normal operation) and any reference found with a timestamp after phase 2 is added to a moved references message (MRM). [10] After the MRM has been generated it is broadcast to other nodes in the same way as the ERM. Once the MRMs of all other nodes have been received phase 5b is finished.

Changes to phase 6

In phase 6 we make two changes. First the MRMs are considered as additions to the ERM-1s. This means that any reference which has moved since the generation of the ERMs is considered as rooted. This is a conservative approach and guarantees that we do not violate the safety property. Liveness is guaranteed by the fact that garbage does not move, since no thread can reach it, and hence cannot have references to it end up in the MRMs.

The second change to phase 6 is that when we do the traversal, whenever a reference is traversed it is looked up in the EI-list and marked. We may in fact use the EI-list to represent the nodes and the ERM-2 messages to represent the edges. References present in the ERMs but not in the EI-list may be placed in a temporary "extra" EI-list with the same structure as the EI-list. If this temporary list is small enough it could be kept in virtual memory only and discarded after phase 6.

We assume that despite that normal processing is being done in parallel with the marking traversal, that we can obtain a unique timestamp. This is needed to distinguish

[10] Since phase 2 used several timestamps, "after phase 2" means after the last timestamp used in phase 2.

a marked entry in the EI-list from an entry which has been recently inserted or updated. Any entry with a timestamp different from the unique timestamp of phase 6 is regarded as unmarked. This means that some new or updated entries may get their timestamp set back in time by the marking traversal. There is no risk of a race condition between the marking traversal and updates to the EI-list due to normal processing because updates by normal processing only add new entries. The updating of existing entries, for the purpose of the MRM is only done until the end of phase 5a (and in fact only updated entries with a timestamp older than phase 1).

Changes to phase 7

Instead of just removing references to garbage from the EI-list we scan the EI-list and remove all references which have either a timestamp from phase 2, [11] or a timestamp from before phase 1. The first are imported objects not marked after phase 2 and are definitely references to garbage, the second are exported objects who have not been marked at all, and includes both references to garbage and references to non-garbage. We leave entries with a timestamp from phase-1 (which are imported references and were added to ERM-1) and references added or updated after phase 2. This will not only remove references to garbage from the EI-list but also references which have been exported (before phase 1) but are not used by other nodes any more. This reduces the size of the EI-list and of future ERMs.

5.4 Optimizations and Refinements

5.4.1 The EI-list

The EI-list must maintain information on all imported or exported objects of a node. Each list entry consists of a reference, a timestamp and an export/import indicator. It must do this with the same degree of fault tolerance as the basic transaction mechanism. During normal processing the object manager observes incoming and outgoing messages. Any reference present in a message and not already on the EI-list must be added to it. Obviously this operation must be as fast as possible. During garbage collection processing the EI-list is used to lookup entries, update entries, delete entries, and for exhaustive linear processing.

The best data structures and algorithms for solving this problem depends on what kinds of messages applications are sending between nodes. We here only sketch one possible design where we assume that the **Export** and **Import** sets form a "substantial part" of **Known** and that the EI-list undergoes a "fair amount" of change. We propose to integrate the EI-list with the object lookup mechanism and augment this with logging.

The object lookup mechanism is used by the object manager to map the large identifiers of persistent objects to a slot in an "object table," (this is similar to some Smalltalk implementations which have a level of indirection where "oops" point to slots in a table containing the physical memory address of the object). A hashing technique is appro-

[11] Since phase 2 used several timestamps, "a timestamp from phase 2" means a timestamp after that of phase 1 but not after the last timestamp of phase 2.

priate here because the number of entries in the table will be much smaller than the size of the identifier space. The hash function maps identifiers to slot addresses. Whether or not the entire object table is maintained in primary storage or some virtual memory technique is used we leave open. Each table entry contains the long identifier (to check for collisions) and the address of the object in some persistent address space. We propose that EI-list entries are stored in the object table. The disadvantage is that this will make all table entries larger (but still of fixed size), and if we have very few exported and imported objects this solution is not cost effective. But it is a simple solution, we have one lookup mechanism instead of two.

The log is used if the object manager has trouble in keeping pace with message traffic, i.e., if keeping the EI-list up to date risks being the bottleneck in the handling of messages. This could be the case if very many or very large messages (in the sense of having many references) have to be processed. If very many references have to be checked against the EI-list at one time, the object manager can simply dump the potentially new EI-list entries to the log. The log could be processed in parallel by a low priority thread. If the object table is small enough to be kept in primary storage then the log is also needed to provide recoverability. The log is also well suited for keeping track of recently exported references, which is done after the quiescent state until the end of phase 5a for the purpose of the MRM. If the transaction manager maintains a log then the log for the EI-list could be integrated with it.

Since we are assuming that communication is done using remote procedure calls (rpc) consisting of paired messages of request and reply there are four "points" at which the EI-list may be updated. A node acting as a client for an rpc may discover new exported references in the outgoing request message and new imported references in the incoming reply message. A node acting as a server for an rpc may discover new imported references in the incoming request and new exported references in the outgoing reply. What we need to ensure is that when the transaction in which the rpc participated is to commit, that all updates to EI-lists have been securely written to non-volatile storage. If the EI-list has been altered at a node acting as a server, then either the update of the EI-list to non-volatile storage has to complete before the reply message is sent, or the message is sent in parallel with the write to non-volatile storage and an additional message containing an acknowledgement is sent later.

The acknowledgement message may be sent asynchronously since it is only of concern to the object managers and not part of the control flow of any application. In other words each reply message of an rpc may contain additional system information indicating that an export-import list was updated and that further information should be expected concerning whether the update of the EI-list succeeded. This acknowledgement may be piggy-backed on some other application related message or sent in a separate message. When a transaction is to commit the local transaction managers at client nodes must check if there are still acknowledgement messages which have not been received from nodes which have been servers to the transaction. If there is any acknowledgement which has not been received then the transaction manager must either wait, request a resend of the acknowledgement, or abort the transaction. [12] At the latest then, acknowledgements

[12]Since the EI-list update acknowledgements may be sent asynchronously in a separate message it may

concerning updated EI-lists become part of the distributed transaction commit messages.

5.4.2 The ERMs and MRMs

One problem concerning ERMs and MRMs is how they should be exchanged. Since there is generally no "hurry," i.e. most garbage will probably be purely local and collected in phase 4, it makes sense to take a lazy approach. If a node **N1** has not received the ERM of the latest generation from some other node **N2**, then the next time **N1** is sending some message to **N2** as part of normal processing, a request for the ERM is appended. In the same way, if a node **N2** knows that some other node **N1** has requested the ERM, then the next time **N2** is sending some message to **N1** as part of normal processing, the ERM is appended. Timeouts could be used to avoid too long delays. The request or reply would then be forced out in its own message.

Another problem is the size of the ERMs. In particular the ERM-1 part could be very large since it contains all outgoing references rooted at the node. A simple solution to this is to send the difference with respect to the previous message instead of the whole set of references each time.

Phases 5-7 of the mechanism force some synchronization on the global system. All nodes are required to "be in step" with the ERM generation number. The node which is slowest to supply its ERM and MRM will set the pace for the whole system. Furthermore the added overhead during phase 4 and 5a of updating the EI-list for recently exported references, can cause problems. If one node is slow to respond it keeps all other nodes in phase 5a longer. It is an advantage if all nodes can agree to enter the next generation so that it is not postponed by some nodes indefinitely. There is then the question of who initiates a new generation.

Any node which has received all ERMs and MRMs of a certain generation, from every other node, is in principle free to enter phase 5 of the next generation whenever it is ready. Conversely, before a node has received all the ERMs and MRMs it may not enter phase 5 of the next generation. Phases 1-4, as has been explained, can be cycled any number of times at a node, under the current generation. The problem is that once a node has finished phase 3 and is about to resume normal operation it has to make a choice. Either it assumes that it will do phases 1-4 again without entering phase 5, in which case it does not have to do the extra processing for the purposes of the MRM (see section 5.3.2). Or it assumes that a "go" decision has been reached for the next ERM generation and it then needs to do the extra processing. A voting algorithm could be used to reach agreement between the nodes to enter the next generation [Thoma79]. Once a decision has been reached to enter the next generation by a majority of the nodes, all nodes should comply by doing all phases of the mechanism the next chance they have.

It is possible for a node to skip a certain ERM generation. It does this by sending an ERM where the ERM-1 and ERM-2 components are empty. The MRM is built by

sometimes happen that the acknowledgement is received by a node before the rpc reply message. This does not matter since when the transaction reaches the point of commit there cannot be any application messages belonging to that transaction still in transit.

inserting *all* imported references found in the EI-list. This ensures safety but not liveness.

5.4.3 Using a copying and incremental collector

In section 5.2.2 it was stated that phase 4 (the reclaiming of garbage) can be run in parallel with normal operation. This is because logically the reclaiming of garbage cannot conflict with any application processing. However, this fact does not mean that phase 4 *should* be run in parallel with normal operation. It will usually be the case that resources are shared on a lower level than abstract objects. Because of this, short term synchronization will usually be required between the application activities and the activity of reclaiming resources. The performance of application processing will then inevitably be degraded. If we have interrupted normal processing to do phase 1-3 anyway, it might be more reasonable to complete phase 4 also during the quiescent state and be rid of its processing cost.

Furthermore, it is often advantageous to do reorganization and compaction of secondary storage at the same time as removing garbage. If phases 1-4 are all run in the quiescent state then we can use a copying collector instead of mark and scan. The marking in phases 1 and 2 will then instead do copying and phase 4 disappears entirely. There are however some complications.

Since we can have more than one persistent address space below the identifier space, the copying may have to work with several from-spaces and to-spaces simultaneously. It might even be the case that some spaces use marking while others use copying.

Furthermore, since phase 2 did not stop the marking traversal if objects are found marked by earlier traversals of phase 2 itself, it should not stop the traversal if doing a copying traversal either, although it *must* stop the copying of the current branch. Remember that phase 2 has one traversal for every member of **UExport** found in the EI-list. Finding an object marked by phase 2 corresponds to finding an object already copied by phase 2. Traversals in phase 2 then do both copying and marking. Marking is still needed if an object already copied by phase 2 is traversed, to ensure termination of the current traversal. The traversals of phase 2 should only stop: at objects copied in phase 1; at objects copied or marked by the current traversal in phase 2; at remote references or stubs.

Using an incremental copying collector [Baker78] in phase 1 and 2 is also possible. We can then allow transactions to be processed in parallel with the local garbage collection. This is relatively straight forward if we only allow transactions which are purely local to the node. Instead of a *quiescent state* as defined in section 3.2 we have an *isolated state*. This means that the EI-list is not modified by anything but the garbage collection mechanism in phases 1-3.

The next step of refinement is to allow multinode transactions in parallel with local garbage collection, but not allowing objects to move to or from a node in phase 2. Transactions which try to move an object during phase 2 will be delayed or aborted. Phase 3, which is not essential for safety or liveness, would be skipped.

Finally it may be possible to allow unrestricted transaction processing in parallel with

all phases of our mechanism. This would require further complication of our mechanism, making heavier use of the timestamps in the EI-list. We are currently studying this possibility.

There is a tradeoff here between either accepting a shorter but disruptive quiescent state with a simpler mechanism, or a longer non-disruptive period of reduced overall performance with a more complicated mechanism.

5.4.4 Problems of scale

A major disadvantage with our mechanism as presented is that it does not scale well. In particular there are two problems. The first is the number of messages. Our mechanism generates fewer messages (but larger) than other proposed solutions to the distributed garbage collection problem. But if there are many nodes then the number of messages may become too large (O(n**2) where n is the number of nodes). A consequence of the number of messages being large is that the secondary storage space required at each node, for storing the messages, may be large. The second problem has already been mentioned. The slowest node in the system to produce the ERM and MRM sets the pace for all other nodes in doing global garbage collection.

The reason that all nodes have to be involved in the exchange of the ERMs and MRMs is because no information is maintained on where a reference has been exported to, or imported from. Imported references are copied and reexported freely, without informing other nodes. Our mechanism only takes advantage of locality of reference by distinguishing between local objects and objects elsewhere. In reality we expect that locality of reference will manifest itself also in larger contexts than a node. We expect that a node will normally have most of its interaction with a few other nodes and only occasionally interact with the larger world. It is also probable that this reasoning applies to several levels, e.g. department, company, city, region, country, world.

One idea is to group nodes into cooperating units. Each member of the group would agree not to reexport any reference, to a node outside the group, without permission from the node where the referenced object resides. A node may be a member of more than one group and maintains EI-lists for each such group. A prerequisite for one node to permit another node to reexport a reference to one of its objects is that both nodes are members of the group to which the object is reexported. Both nodes add an entry in the EI-list for the group to which an object becomes exported.

Phase 1 would be the same as before, except that remote references must be looked up in all EI-lists. Phases 2 and 3 must be run for each EI-list. Phase 4 is unchanged. Phases 5-7 may be run separately for each EI-list. The different EI-lists would have different generation numbers permitting different cycle periods for different groups.

It would also be possible to organize the groups hierarchically. This would be similar to the Unix file system security domains *user*, *group*, and *other* [Ritch74]. Although our node groups are intended for defining locality of reference and not access control, the two problems are related and it is possible that node groups could be used for both purposes at the same time. When a reference is exported from one node to another, the two nodes

would each insert an entry in the EI-list corresponding to the smallest node group both are members of (unless the reference already exists in a larger group at the exporting node).

Different node groups could be bound by different "contracts" concerning how to reach agreement on when to enter the next ERM generation, or concerning the conditions when reference may be exported outside the group.

The problem of scale can also be attacked by relaxing the safety and liveness requirements. One could have a *global* node group which all nodes are members of by definition (corresponding to the security domain *other* in Unix). No EI-list would be maintained for this group. References exported to this group would be marked as "unreliable." When the object manager scans incoming our outgoing messages and finds such a reference it ignores it. In other words no lookup or addition to the log is made for any EI-list. Safety would not be guaranteed for such references.

6 Comparison with other work

The Commandos architecture [Marqu88, COMAN87] uses aging instead of garbage collection. Objects which have not been accessed for a certain period of time are successively moved to "lower levels" in a memory hierarchy. This is similar to swapping except that objects reaching the lowest level never return. This approach is not safe in a formal sense since reachable objects may be removed. Nevertheless it may be acceptable. The approach has the advantage that it scales well. Aging and garbage collection are not mutually exclusive. In fact garbage collection can offload the aging mechanism by removing large quantities of "young" garbage, and the aging mechanism can offload the garbage collection mechanism by relieving it of the responsibility of collecting certain domains, like the global node group discussed in the previous section.

Mancini and Shrivastava argue for using a reference counting approach [Manci88]. Their mechanism is primarily intended for garbage collection of volatile objects. Although they do explain how to extend the mechanism to also handle garbage collection of persistent objects, their mechanism does not reclaim distributed cyclic garbage. This is more important for persistent objects than it is for volatile objects. Because some node in the cycle will sooner or later be taken off-line (either by a crash or by a controlled shutdown), and this will break the volatile cycle. Furthermore, as presented, their mechanism can not handle objects which move from one node to another. Nevertheless their paper has been the chief influence on our design. We believe their mechanism is more suited for systems with relatively few internode references to volatile objects, which do not move, while our mechanism is probably more suitable for systems with relatively many internode references to persistent objects, and objects which frequently move from one node to another. We believe persistence will tend to increase both the number of internode references and the need to move objects.

Kolodner, Liskov and Weihl describe a garbage collector and recovery system for a persistent heap [Kolod89]. Their garbage collection algorithm is based on a copying collector. It may be run while transactions are in progress, but suspends all transactions

during the garbage collection. They assume a disciplined virtual memory address space as the basis for persistence. They claim that their algorithm, with minor modifications, could be used in a distributed system. It is not clear however whether they assume a global flat virtual address space and if their distributed solution requires all transactions in the entire distributed system to be suspended at the same time. If this is the case, then their solution is not acceptable in a decentralized system.

Moss outlines how garbage collection could be done in his system based on location dependent addresses, which has many similarities with the mechanism we propose [Moss89]. Because we use location independent identifiers our mechanism has to keep track of some locality information. This is not necessary if the identifiers themselves contained locality information as proposed by Moss. As already explained, the disadvantage with location dependent identifiers is that either are objects not allowed to move, or we do not maintain strong identity for objects.

In [Schel88] a scheme is outlined, although not fully explained, which does dynamic incremental distributed garbage collection for a distributed Smalltalk system. This is a multiple workspace system and does not have global identifiers. Their system does not make any explicit distinction between volatile and persistent objects and does not seem to address the problem of inconsistencies when some nodes crash. [13] They base their algorithm on the work of Ali [Ali84] which is an incremental copying collector for distributed systems, but which does not reclaim distributed cyclic structures. This problem is overcome in [Schel88] by an "AccessPath" mechanism, the details of which are not explained in the paper, nor delegated by reference to a technical report.

The distributed Smalltalk described in [Benne87] is similar to the previous one (multiple workspace model, no global identifiers). Distributed garbage collection is however done by global mark and scan in parallel with normal activity. This must be quite expensive and goes against autonomy.

7 Conclusions

We have proposed a global persistent and decentralized identification scheme and provided a garbage collection mechanism for a system employing such a scheme. Using a global identification scheme allows references and objects to move freely in the system.

Our garbage collection mechanism allows a considerable degree of node autonomy. It is symmetrically decentralized and does not require synchronized clocks. A node doing purely local processing does not need have *any* overhead from the secondary storage garbage collection mechanism when all phases are finished. This is because overhead for the garbage collection mechanism during normal processing only consists of the maintenance of the EI-list, which is only relevant when messages are sent to and from other nodes.

Each node decides locally when and how often to perform local garbage collection. We suggested some form of voting protocol so that at least consensus could be established

[13] see footnote 1.

on when a global garbage collection is desired. Liveness is guaranteed if all nodes do garbage collection with a nonzero frequency and are willing to cooperate when a decision has been reached to do a global garbage collection.

The mechanism has the weakness of not scaling well. We introduced the notion of node groups and node group hierarchies as a means of coping with a large system. It seems likely that a node will have most of its interaction with a few closely related nodes. The garbage collection mechanism will then generate fewer but larger messages for a small group closely cooperating nodes, and relatively many but small messages for a larger group of nodes where exchanges are less common.

References

[Ali84] K.A-H.M. Ali, "Object-Oriented Storage Management and Garbage Collection in distributed Processing Systems," TRITA-CS-8406, Stockholm, Sweden, Dec. 1984.

[Atkin87] M.P. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, vol. 19, no. 2, pp. 106-190, June 1987.

[Baker78] H.G. Baker, "List-processing in real time on a serial computer," *Communications of the ACM*, vol. 21, no. 4, pp. 280-294, April 1978.

[Balla83] S. Ballard and S. Shirron, "The Design and Implementation of VAX/Smalltalk-80," in *Smalltalk-80 Bits of History, Words of Advice*, ed. G. Krasner, pp. 127-150, Addison Wesley, Reading, Massachusetts, 1983.

[Benne87] J.K. Bennett, "The design and implementation of distributed smalltalk," *Object-Oriented Programming Systems, Languages and Applications*, pp. 318-330, Orlando, Florida, Oct. 1987.

[Björn88] A. Björnerstedt and S. Britts, "AVANCE: An Object Management System," *Object-Oriented Programming Systems, Languages and Applications*, San Diego, CA, 1988.

[Butle87] M.H. Butler, "Storage reclamation in object-oriented database systems," *ACM SIGMOD Proceedings*, pp. 410-425, San Francisco, May 1987.

[COMAN87] "COMANDOS: Object-Oriented Architecture," D2-T2.1-870904, Esprit project 834, 1987.

[Cocks88] W.P. Cockshott, "Addressing mechanisms and persistent programming," in *Data Types and Persistence*, ed. M.P. Atkinson, P. Buneman, R. Morrison, pp. 235-252, Springer-Verlag, 1988. Based on proceedings from a workshop held in Appin in August 1985

[Cohen81] J. Cohen, "Garbage Collection of Linked Data Structures," *ACM Computing Surveys*, vol. 13, no. 3, pp. 341-367, Sept. 1981.

[Colli60] G.E. Collins, "A method for overlapping and erasure of lists," *Communications of the ACM*, vol. 3, no. 12, pp. 655-657, Dec. 1960.

[Comer79] D. Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121-137, June 1979.

[Daley68] R.C. Daley and J.B. Dennis, "Virtual Memory, Processes, and Sharing in Multics," *Communications of the ACM*, vol. 11, no. 5, pp. 306-312, May 1968.

[Decou86] D. Decouchant, "Design of a distributed object manager for the Smalltalk-80 system," *Object-Oriented Programming Systems, Languages and Applications*, pp. 444-452, Portland, Oregon, 1986.

[Dijks78] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-Fly Garbage Collection: An Exercise in Cooperation," *Communications of the ACM*, vol. 21, no. 11, pp. 966-975, Nov. 1978.

[Fenic69] R.R. Fenichel and J.C. Yochelson, "A LISP Garbage-Collector for Virtual-Memory Systems," *Communications of the ACM*, vol. 12, no. 11, pp. 611-612, Nov. 1969.

[Goldb83] A. Goldberg and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison Wesley, 1983.

[Gray78] J.N. Gray, "Notes on database operating systems," in *Operating Systems*, ed. R. Bayer, R.M. Graham, and G. Seegmuller, pp. 393-481, Springer-Verlag, Berlin, 1978.

[Gray86] J.N. Gray, "An Approach to Decentralized Computer Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 6, pp. 684-692, 1986.

[Howar88] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51-81, Feb. 1988.

[Jones86] M.B. Jones and R.F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *Object-Oriented Programming Systems, Languages and Applications*, pp. 67-77, Portland, Oregon, Sept 1986.

[Kaehl86] T. Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language," *Object-Oriented Programming Systems, Languages and Applications*, pp. 87-106, Portland, Oregon, Sept. 1986.

[Khosh86] S.N. Khoshafian and G.P. Copeland, "Object Identity," *Object-Oriented Programming Systems, Languages and Applications*, pp. 406-416, Portland, Oregon, Sept. 1986.

[Knuth68] D.E. Knuth, *The art of computer programming - Fundamental Algorithms*, Addison Wesley, 1968. vol. 1

[Kolod89] E. Kolodner, B. Liskov, and W. Weihl, "Atomic Garbage Collection: Managing a Stable Heap," *Proceedings of the ACM-SIGMOD Conference*, pp. 15-25, Portland, Oregon, 1989. SIGMOD RECORD Vol. 18, Nr. 2, June 1989

[Larso88] P.A. Larson, "Dynamic Hash Tables," *Communications of the ACM*, vol. 31, no. 4, pp. 446-457, April 1988.

[Levy84] H.M. Levy, *Capability - Based Computer Systems*, Digital Press, 1984. ISBN 0-932376-22-3

[Liebe83] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, vol. 26, no. 2, pp. 419-429, June 1983.

[Lisko83] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 381-404, July 1983.

[Manci88] L.V. Mancini and S.K. Shrivastava, "Fault-tolerant reference counting for garbage collection in distributed systems," No. 260, University of Newcastle upon Tyne, June 1988.

[Marqu88] J.A. Marques, R. Balter, V. Cahill, P. Guedes, N. Harris, C. Horn, S. Krakowiak, A. Kramer, J. Slattery, and G. Vandome, "Implementing the Comandos Architecture," *ESPRIT'88: Proceedings of the 5th Annual ESPRIT Conference*, pp. 1140-1157, Brussels, Nov. 1988.

[McCul87] P.L. McCullough, "Transparent Forwarding: First Steps," *Object-Oriented Programming Systems, Languages and Applications*, pp. 331-341, Orlando, Florida, Oct. 1987.

[Moss88] J.E.B. Moss and S. Sinofsky, "Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface," *2nd International Workshop on Object-Oriented Database Systems*, pp. 298-316, Sept. 1988. Lecture Notes in Computer Science: 334 "Advances in Object- Oriented Database Systems"

[Moss89] J.E.B. Moss, "Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach," *Second International Workshop on Database Programming Languages*, pp. 269-285, Glenden Beach, Oregon, June 1989.

[Purdy87] A. Purdy, B. Schuchardt, and D. Maier, "Integrating an Object Server with Other Worlds," *ACM Transactions on Office Information Systems*, vol. 5, no. 1, pp. 27-47, Jan. 1987.

[Ritch74] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, vol. 17, no. 7, pp. 365-375, July 1974.

[Schel88] M. Schelvis and E. Bledoeg, "The Implementation of a Distributed Smalltalk," *ECOOP'88 European Conference on Object-Oriented Programming*, pp. 212-232, Oslo, Norway, Aug. 1988.

[Shriv88] S.K. Shrivastava, G.D. Dixon, F. Hedayati, G.D. Parrington, and S.M. Wheater, "A Technical overview of Arjuna: a system for reliable distributed computing," No. 262, University of Newcastle upon Tyne, July 1988.

[Spect82] A.Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM*, vol. 25, no. 4, pp. 246-260, April 1982.

[Spect87] A.Z. Spector, D. Thompson, R.F. Pausch, J.L. Eppinger, D. Duchamp, R. Draves, D.S. Daniels, and J.J. Bloch, "Camelot: A distributed transaction facility for Mach and the Internet - An interim report.," CMU-CS-87-129, Dept. of Comp. Sci. Carnegie Mellon University, Pittsburgh, PA 15213, June 1987.

[Stamo84] J.W. Stamos, "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 155-180, May 1984.

[Stein89] J. Stein, T.L. Anderson, and D. Maier, "Mistaking Identity," *Second International Workshop on Database Programming Languages*, pp. 287-291, Glenden Beach, Oregon, June 1989.

[Svobo84] L. Svobodova, "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys*, vol. 16, no. 4, pp. 353-398, Dec 1984.

[Thatt86] S.M. Thatte, "Persistent Memory: A Storage Architecture for Object-Oriented Database Systems," *Proceedings of the International Workshop on Object-Oriented Database Management Systems*, pp. 148-159, Pacific Grove, CA, 1986.

[Thoma79] R.H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180-209, June 1979.

[Ungar84] D. Ungar, "Generation Scavenging: A non-disruptive high performance storage reclamation algorithm," *ACM Software Engineering Notes*, vol. 9, no. 3, pp. 157-167, May 1984. Proceedings ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments

[Ungar88] D. Ungar and F. Jackson, "Tenuring Policies for Generation-Based Storage Reclamation," *Object-Oriented Programming Systems, Languages and Applications*, pp. 1-17, San Diego, California, 1988.