Global Garbage Detection in a Distributed Environment: An Implementation for Raven

by

Gordon Devlin

B.Sc. (Computer Science), The University of British Columbia, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
September 1995

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date OCT 16, 1995

ABSTRACT

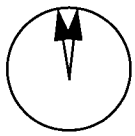
Working in a distributed environment offers challenges surpassing those of operating in a self-contained world. While references and objects can travel beyond known boundaries, it is still possible to maintain an adequate picture of the global situation: allowing identification and reclamation of unreachable objects while preventing the collection of reachable objects which are not locally referenced. This work describes the implementation of a global garbage detection scheme for Raven – an object-oriented distributed system developed at the University of British Columbia.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

My admiration and thanks to Donald Acton, one of the original authors of Raven, whose analytic and descriptive abilities are quite inspiring and more than once got me out of a dark spot. I would also like to thank my supervisor, Norm Hutchinson, who came through for me a number of times.

Finally, a nod to those individuals who made my time at UBC much more than an academic experience — Veronica, Carol, Roland and Marcelo. And to my office-mates, Kurt and Scott, you both rate an 

Whether you think you can or you can't — you're right!
— Ford

CHAPTER 1 Introduction

Garbage. It is a topic of recent popular concern, conjuring up images of massive landfills resulting from our previous attitudes of disposability and replaceability. Until now we have been blind to or afforded little energy to the necessity of conservation and reclamation of our valuable resources.

Not so in the computer world. Most denizens thereof will recall introductory lessons in the basic space/time trade-off. It is an issue brought to our attention early on. And even though technological advances have provided us with vast amounts of memory and improved the speed at which we calculate, we cannot escape the following basic truth:

If an operating system were to continue allocating chunks of memory without any sort of replenishment, the resource would quickly be totally consumed. Replenishment in this case consists of reclaiming garbage — those memory chunks no longer accessible by any means — and putting it back into the store of available memory.

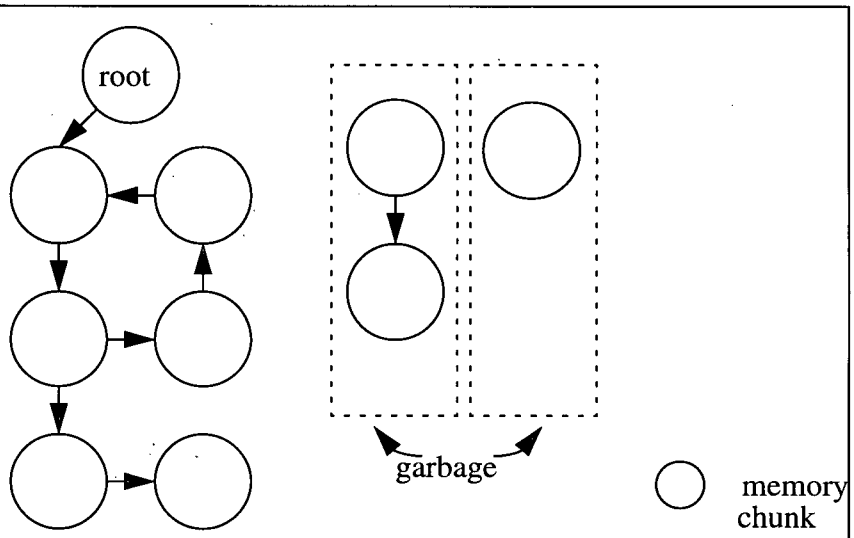


Figure 1. Graph of Allocated Memory

Figure 1 depicts the definition of garbage using a graph. The nodes represent the individual chunks of allocated memory, the arrows represent references or pointers contained within a chunk that provide links to other areas of memory. Starting at the root and following all links from nodes visited would then lead you through a traversal of the reachable portion of the graph. Any node not visited is considered garbage and can be made available for reallocation.

One well-known allocate/reclaim mechanism is the cooperating *malloc()* and *free()* system calls, which requires the programmer to determine when a particular memory location is no longer necessary and to then make the appropriate call to inform the system that it may be reused. To free the programmer from this tedious (and error-prone) task, the system itself may undertake to determine which memory locations are no longer reachable and to return them to the free memory pool. List processing environments (e.g. Lisp, Scheme, and their derivatives) employ this method.

The act of identifying inaccessible memory and reclaiming it for reuse is known as garbage collection. Today there is not the level of research interest that there once was in garbage collection for the local case (i.e. for a single machine), as there are a variety of known methods which effectively address the issue. A number of variations have been devised, but all are derived from one of two main schemes. These are reference counting and tracing algorithms. With reference

counting, there is an exact record kept of the number of references existing to a memory location (e.g. each time a reference is copied, the count is augmented, with the count being decremented with each drop of a reference). When the count reaches zero, there are no viable references to that location and the memory may be re-allocated. Tracing or traversal algorithms can be further subdivided into the copying and mark-and-sweep approaches. A basic copying algorithm would have the available memory store subdivided into an *active* and a *copy* portion. During garbage collection, each memory chunk visited in the *active* section would be copied to the *copy* area directly upon being encountered. At the end of the tracing (and copying) process, the *active* area is devoid of reachable memory. All that remains to complete the process is for the names to be interchanged. The process can be repeated once garbage collection is triggered in the new *active* area. With mark-and-sweep, each node reached during a traversal would be designated live (i.e. marked). Following a total traversal, any unmarked memory chunks would then be returned to the memory pool during the sweep phase.

A popular methodology in the past few years is to have data represented as objects: an object being an encapsulation of data and some procedures (or methods) which can be invoked to access or act on that data. Now, objects need not exist solely within an application; they may outlive the application, or an operating system may also be object-oriented, having the items it deals with (e.g. files, processes) represented as objects also. Objects will also usually not exist in isolation, but rather be linked to other objects. Links in this case would be references (object ids), contained within an object's data, to other objects. To carry on the operating system example, a process might well have a reference to a working file. Going back to the garbage collection discussion (Figure 1, with nodes now representing objects), a mark-and-sweep algorithm could then start out with a root (group of objects), each object referenced within the root being marked and subsequently acting as part of the root itself — thus all accessible objects would then be marked, and any unmarked objects could safely be collected (e.g. unreferenced files could be reclaimed).

While the move from considering memory chunks to dealing with objects had very little impact on the problem, when we leave the local case and move the discussion into a distributed context, there are many implications. Some terminology is in order at this point. A *site* is a single host or node within the network (or could even be an individual process on a host, thus enabling multiple

sites per host). An object is said to be *resident* or *local* to a site if the data portion of that object is present at that site. If a reference exists to an object not resident at that site, the object is said to be *remote* (conversely, the object containing the reference can also be considered to be remote from the point of view of the referenced object). Furthermore, at each site with a reference to a remote object, there will exist a *stub* or *proxy* for that object — invoking a method on this stub will result in the arguments being marshalled and execution of a remote procedure call so that the object itself may execute the call, with results (if any) being returned to the calling object.

Garbage collection thus can also be considered on a global (system-wide) scale. In tracing methods, no longer can the local root be considered in isolation; for in the distributed case, any proposed solution must allow for the possibility of remote references to local objects. In fact, off-site references might not be at any site, but rather could be in transit at the time garbage collection is being conducted. There is also little control of or knowledge about what happens at remote sites — e.g. how many times a reference is copied or subsequently passed on to other sites. Further complicating the task is that objects themselves may migrate from site to site (possibly either at the discretion of the system or in response to a direction from an user application), which could potentially create indirect references (i.e. a reference leading to an object which is itself a stub "standing in" for the migrated object). In addition, the familiar complexities of operating in a distributed context (e.g. failures, message loss, communications downtime) must be accounted for in any solution. The focus of this thesis is garbage collection in such a distributed context, or *distributed garbage collection.*

1.1 Desirable Qualities

Global garbage collection schemes typically consist of a local component (which performs the actual collection of all local data, and is often pre-existing) which acts cooperatively with a global component (which is responsible for the management of all intersite references). Depending on the algorithm, these two components will have varying degrees of interaction and synchronization. There are several desirable characteristics of a good distributed garbage collector:

- a **comprehensive** collector guarantees that all and only garbage objects are

detected and reclaimed. Thus, no garbage objects should escape detection (i.e. be incorrectly considered live); nor, more seriously, should any live objects be collected (i.e. be incorrectly considered garbage). The former situation could arise, for example, in a conservative algorithm when a potential reference is considered sufficient to maintain an object's liveness, but the potential reference is itself non-viable, resulting in that object being overlooked. The latter situation (premature collection) could occur in a reference-based system when the count for an object erroneously gets reduced to zero, rendering the object eligible for collection. A notable sub-group of (often) non-detected garbage is distributed cycles (see Figure 6). These occur when a cycle exists which, were it to be located on one node, would be collected as garbage, yet because it exists across site boundaries, the local collectors involved cannot detect by themselves that it is, in fact, garbage.

- a **concurrent** collector would operate in parallel with active processes. Thus, there would not be any significantly long interruptions to any processes suspended during any or all phases of the garbage collection process.

- a **robust** collector would be tolerant to various failures within the system, including communication and host failure. Local collection should be unaffected by any network or host failure (i.e. local collection may proceed even if one or more hosts are unreachable). A finer point on this matter would include in the definition of robustness the notion of whether or not the global mechanism could either proceed at a time when at least one node was unreachable or conclude at a point even though not all nodes were available.

- turning to more quantitative measures, an **efficient** collector introduces relatively little overhead compared to the amount of memory collected. Overhead measures often considered are the increase in network traffic, the amount of memory necessary to keep track of the intersite relationships, and the extra processing time incurred due to the presence of garbage collection. **Expedience** is a measure of the collector's ability to return garbage to the pool at a rate sufficient to allow the alloca-

tor to meet the demand for new requests.

1.2 Goals

The prime goal of this thesis is to formulate a working solution in order to provide global garbage collection for Raven — an object-oriented distributed system. A secondary goal is to attempt this with a non-language-specific algorithm to illustrate that garbage detection need not be inextricably tied to the target language.

To this end, comprehensiveness and robustness are focused on as significant qualities to be achieved. Performance issues will be attended to only when not in conflict with either of the stated goals.

1.3 Organization

The remainder of this thesis is organized as follows:

Chapter 2 presents the chosen algorithm – outlining the pertinent aspects and noting any facets which were not employed.

Chapter 3 focuses on describing the intended system (Raven) and how various aspects of it affect the algorithm.

Chapter 4 details the implementation issues, highlighting the data structures employed and the integration process.

Chapter 5 concludes the work indicating the current status of Raven's global garbage detection scheme, evaluates the success of the endeavor and points out how some of Raven's future enhancements might impact on the scheme. The chapter concludes with descriptions of other approches to distributed garbage collection, and compares the adopted approach with these.

A mind once stretched by a new idea never regains its original dimension.

— Oliver Wendell Holmes

CHAPTER 2 Algorithm

Operating in a distributed environment, any garbage collection algorithm must pay attention to the possibility of network and/or machine failures, message loss and duplication, as well as messages being in transit. Presented here is a high level picture of a conservative algorithm designed for distributed garbage detection — described in full in [SHAPIRO90]. The algorithm is tolerant of failures, and time information accompanying inter-site communication addresses the mentioned message issues. The implementation does vary from that listed in [SHAPIRO90]; notable exceptions along with rationale being found in the latter part of this chapter.

2.1 Data Structures

Each site will require the following data structures (see Figure 2):

- a monotonically increasing clock (a simple counter is sufficient, incrementing on message send) which need not be synchronized with the clock at any other site. Each message sent from a site will include a timestamp equal to the current clock value
- an Object Directory Table (ODT) listing those local objects that are referenced at remote sites (along with the site and the local time that the reference was sent)
- an External Reference Table (ERT) consisting of stubs representing those objects it has reference to which are resident at remote sites
- a list of the most recent timestamps it has received from each site that it communicates with – which is updated upon receipt of any message

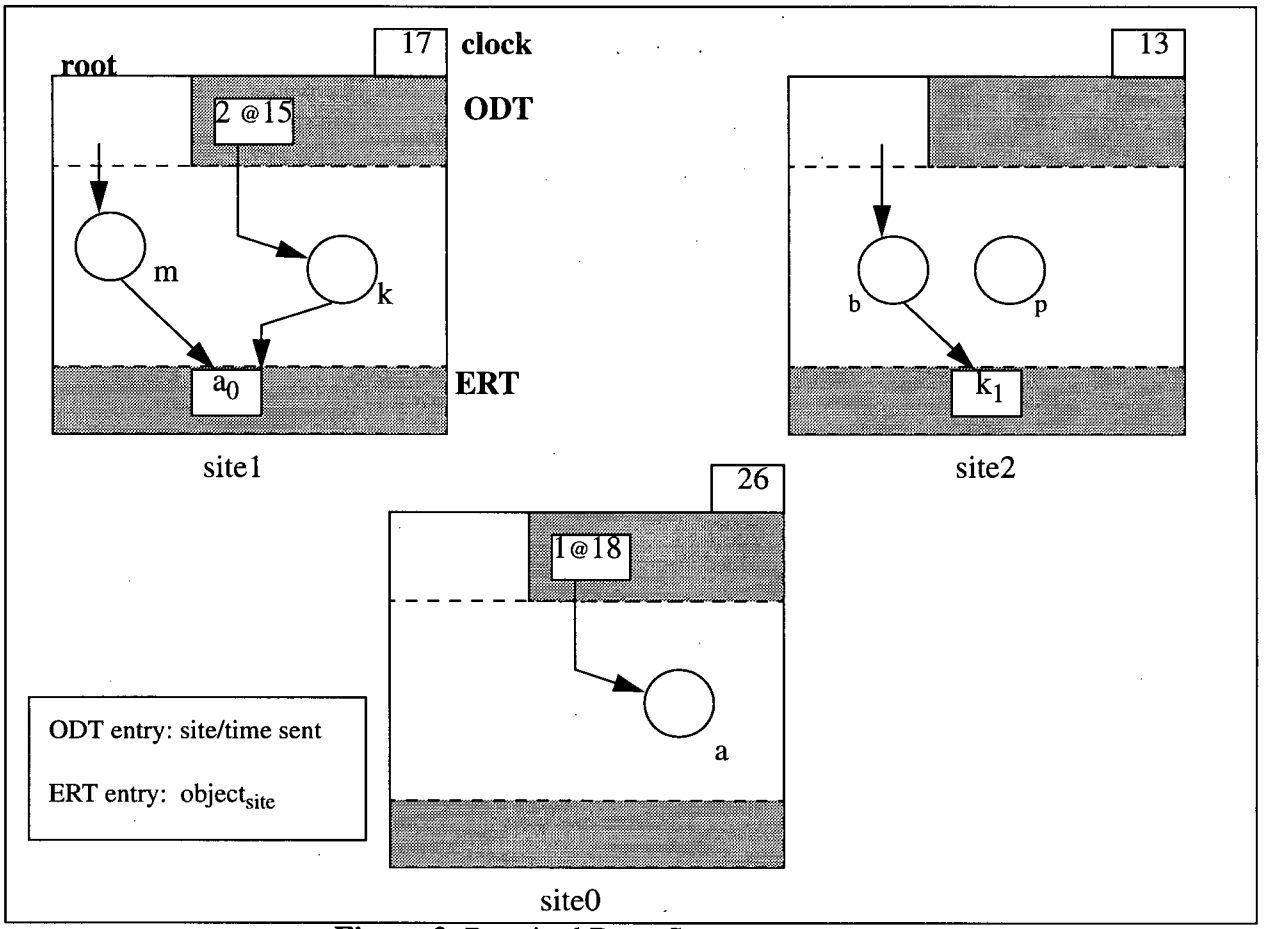


Figure 2. Required Data Structures

2.2 Sending References

Whenever a reference to a local object is sent offsite (e.g. an invocation on a remote (stub) object with a local object as an argument), it must be recorded that the destination site has that reference. So, **before** the message containing the reference is sent, an entry of the reference, destination site, and time is made to the ODT. Of course, if there is already a record of that site having the reference, no new entry need be made, rather just an appropriate update of the time is made. No record is kept as to the specific number of times a reference is sent to any one site; the crucial aspect is recording the existence of that reference for each remote site that it was sent to — for the foundation of the algorithm is that the existence of one reference at any remote site is sufficient to render the object viable (as far as the local collector is concerned).

Correspondingly, upon receipt of a reference to an object not resident at a site, a stub object is created and entered into the ERT along with a record of the originating site (see Figure 3).

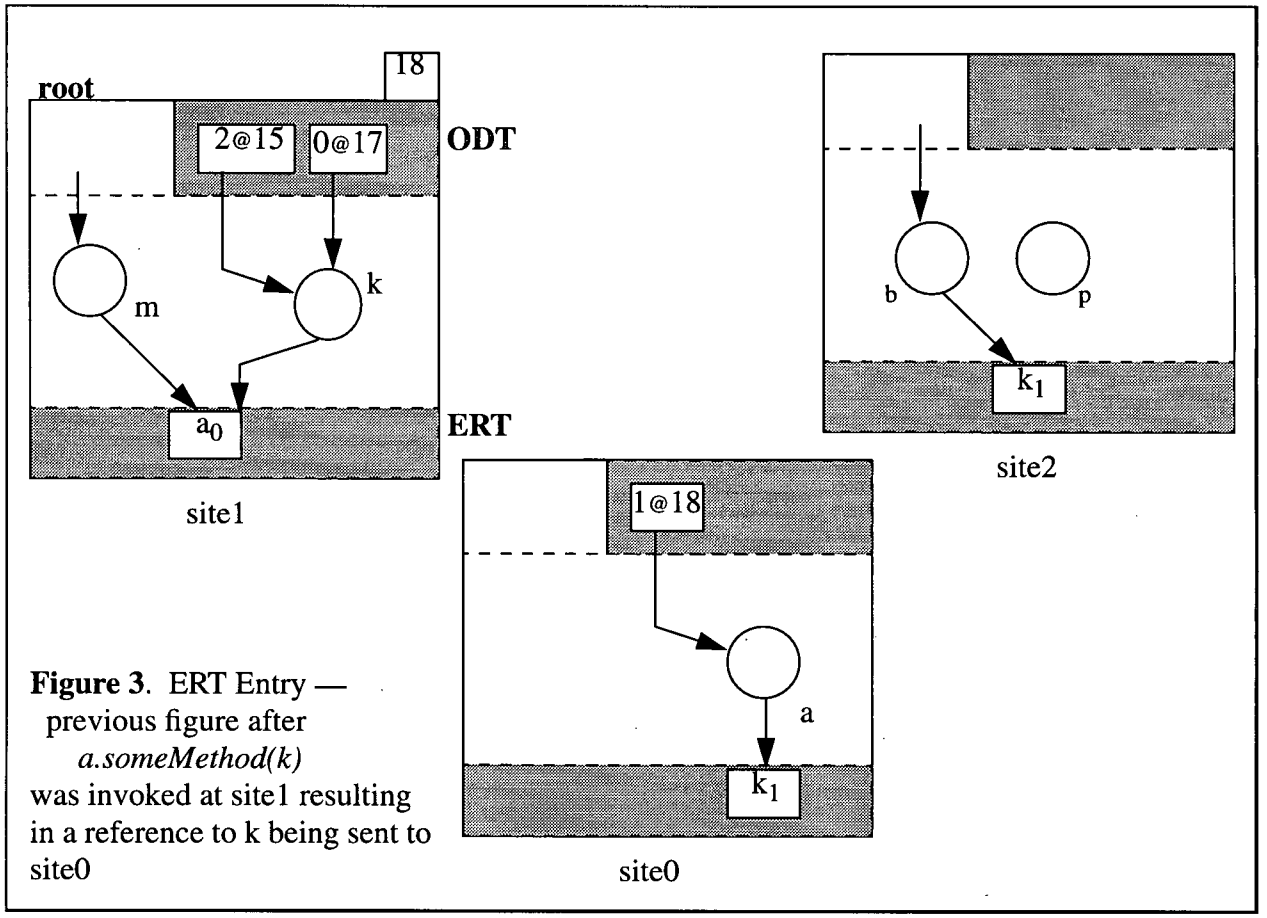


Figure 3. ERT Entry — previous figure after *a.someMethod(k)* was invoked at site1 resulting in a reference to k being sent to site0

2.3 Local Garbage Collector

The only demand on the local collector is that it now extend its notion of the local root to include the local ODT. This way, any object(s) which might be considered garbage when tracing the original root will now be recognized as reachable (i.e. if there is an appropriate entry in the ODT). Consider the situation in Figure 3: if local collection were conducted at site1 without any consideration of the ODT as part of the local root, only objects referenced from the root would survive the collection; thus **k** would be collected, even though it is globally reachable, being referenced from objects at multiple remote sites. Now, the reference at the other site may no longer be in existence, but local collection will be correct in either case — if the reference **is** still present at the remote site, then the object must not be collected; while an object for which the remote reference no longer exists will be (conservatively) kept around longer than required.

2.4 Global Detection Protocol

As has just been seen, the local collector exhibits conservative behavior - the result of which is that objects for which there was ever a remote reference will not be collected as long as the record in the ODT persists. The key to ODT entry removal lies in the action of the local collector of the site to which the reference was sent. If the reference no longer exists at the remote site (e.g. it was overwritten or the object containing the reference itself was collected as garbage) then during the subsequent local collection the entry in the ERT will be discovered to be stale and a removal message will be sent to the originating site before removing the entry. Upon receipt of the removal message, the corresponding ODT entry will be removed. Then, if there were no other entries referencing this object in the ODT (nor from the root), when next the local collector is run the object will be recognized as garbage and collected.

2.4.1 Removal Message

A removal message is sent when it is detected that an ERT entry is no longer reachable (see Figure 4 - as **p** at site1 is not accessible, the stub $\mathbf{q}_0$ is also not reachable). Such a message indicates to the receiver that the remote site is no longer referencing the indicated local object, allowing the removal of the corresponding ODT entry.

In addition to indicating the once-referenced object, the message includes the highest timestamp (hts_0) received by the originating site from the destination site (e.g. in Figure 4, the last message received at site1 from site0 was timestamped 150).

Upon receipt of the removal message, the highest timestamp seen remotely is compared with the timestamp included in the corresponding ODT entry (ODT_{ts}). If they are equal (e.g. $ODT_{ts} = hts_0 = 79$), then the message containing that reference was the last message received at the remote site. Similarly, if the ODT entry timestamp is lower ($ODT_{ts} = 79$, $hts_0 = 150$) , then there were messages subsequent to the one containing the reference. However, if the ODT entry timestamp is greater than the highest timestamp seen remotely ($ODT_{ts} = 79$, $hts_0 = 72$), then the message containing the reference was in transit at the time the removal message was sent (otherwise hts_0 would be at least 79). In this last case, *where the highest timestamp seen remotely is strictly less*

than the timestamp recorded with the ODT entry, the removal message is ignored. This is done for correctness: to preserve a local record of the potential reference at the remote site and thus prevent the possible premature reclamation of the object referenced.

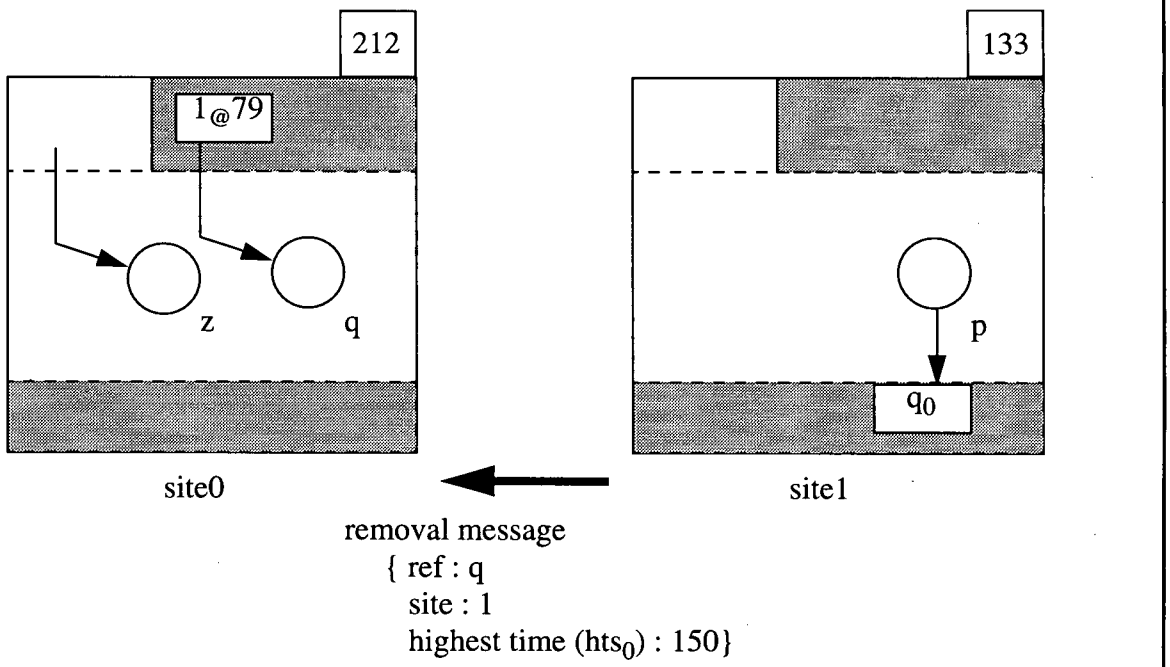


Figure 4. Removal Message

2.4.2 Solution to Messages in Transit

Removal messages are the mechanism to enable cleaning out unnecessary ODT entries. With the possibility of having references in transit and the subsequent possibility of removal messages being ignored, we return to the situation where some entries in the ODT might be stale. In order to ensure timely removal in this case, there is a currentERT message which is sent infrequently (yet periodically) that informs the receiver of those objects to which the sender has a reference.

site1 -> site0 : currentERT(ERT_0 , highest $timestamp_1$, timestamp)

Included in the message would be those entries in ERT_1 that refer to site0, the highest timestamp

received at site1 from site0, and the current site1 local clock value.

In processing the currentERT message, any entries in the receiver's ODT for which there is no corresponding entry in the message may safely be removed (with the exception, of course, of those ODT entries with a timestamp later than the highest timestamp reported).

2.5 Disconnection

If communication with a particular site is impeded (e.g. due to network partitioning), the action of the global detection scheme is affected but its correctness is not compromised. During the down-time, there will be no messages between the disconnected site(s). Thus, the ODT of any site attempting to send references to the affected site(s) will continue to (correctly) contain a superset of the objects remotely referenced. At the disconnected site(s), no ODT entries will be removed (removal and currentERT messages will not be received) so collection there will likewise remain correct. Pairwise cooperation among sites in each partition will proceed normally.

2.6 Failures

The algorithm allows for two outcomes after a site has failed:

1. no recovery is possible — the site's global mechanism structures (e.g. ODT and ERT) are lost and all objects are forever unreachable as all objects are volatile
2. recovery is possible — global detection structures persist and any objects which were volatile are lost but those which are persistent are still reachable

As persistence in Raven is possible, only situation 2 is addressed. With the ODT remaining intact across incarnations, all objects persisting which were reachable from other sites will still have their entries in the ODT, thereby maintaining correctness. Any unused ERT entries (i.e. resulting from references to offsite objects being lost in the crash) will be collected in the normal course of the algorithm. The only possible impact to the global scheme is when there is an ODT entry for a volatile object. Upon recovery, then, there exist offsite references to an object which did not survive the failure. The solution is to extend to ODT referenced objects the persistent nature of the global mechanism structures — i.e. any object which is referenced offsite will be guaranteed to be persistent (and, hence will survive the failure).

2.7 Notable Exceptions

The algorithm described in [SHAPIRO90] is implemented as is with a few exceptions. The exceptions do not affect the correctness of the algorithm; rather they are merely aspects which were unnecessary as they address issues which are not pertinent to the current version of Raven.

2.7.1 Migrating an Object

When migrating an object (e.g. Figure 5a – migrating **w** from site1 to site2), it is important to allow for the possibility of a local reference to it at the site it is leaving - calling for an entry to the ERT (Figure 5b - entry $\mathbf{w}_2$ made to ERT_1). Furthermore, any object referenced by the migrating object must be protected from premature collection - requiring ODT entries prior to the object leaving (e.g Figure 5b - ODT_1 entry for object **j** allowing for reference from site2).

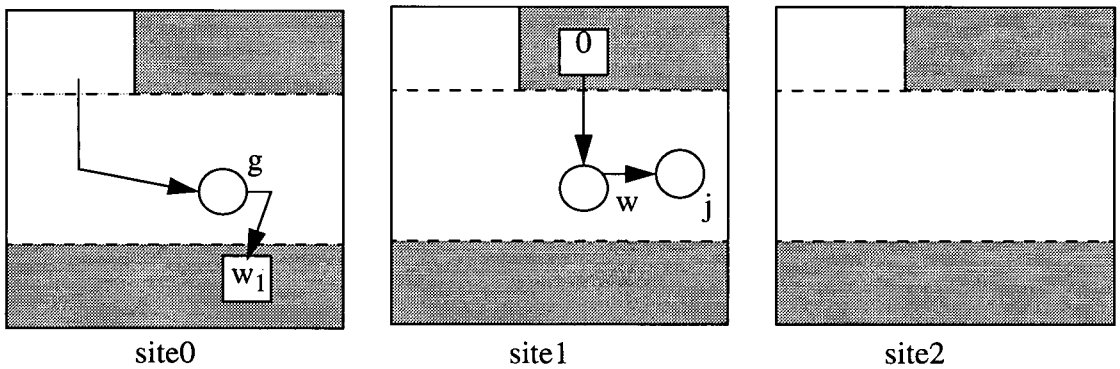


Figure 5a. Migrating an Object

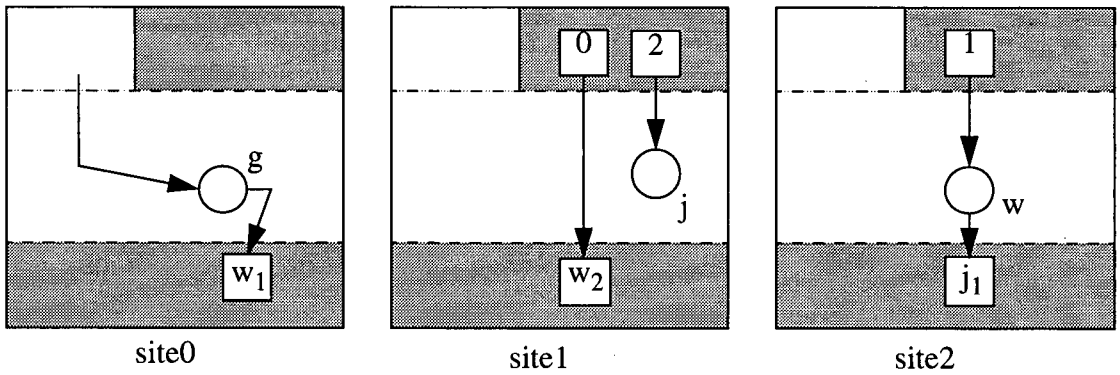


Figure 5b. w is now at site2

However, Raven's migration capability is currently limited to copies of objects - this situation arises, for example, when there is a remote invocation of a method for which at least one of the

declared parameters has been tagged with the **copy** keyword. Contrary to the migration issues described above, any references to (local and offsite) objects contained in copies need no special attention as copy semantics ensure a deep-copy of the object. Any references are not merely duplicated — rather the object referenced is itself recursively copied (and migrated). To reiterate, with Raven's current scope of migration, any object created via the usual means will be resident at its creation site for the duration of its life.

2.7.2 Cycle Elimination

Of interest to the global detection scheme are cycles of objects that persist unnecessarily (i.e. cycles of garbage). A simple case is depicted in Figure 6 below where object **z** has a (remote) reference to object **q**, and **q** contains a reference to object **z** — and there is no local reference to either. If both objects were at the same site, the next local collection would recognize them as unreachable and reclaim them. Because they are at different sites, the entries in their respective ODTs will prevent them from being collected.

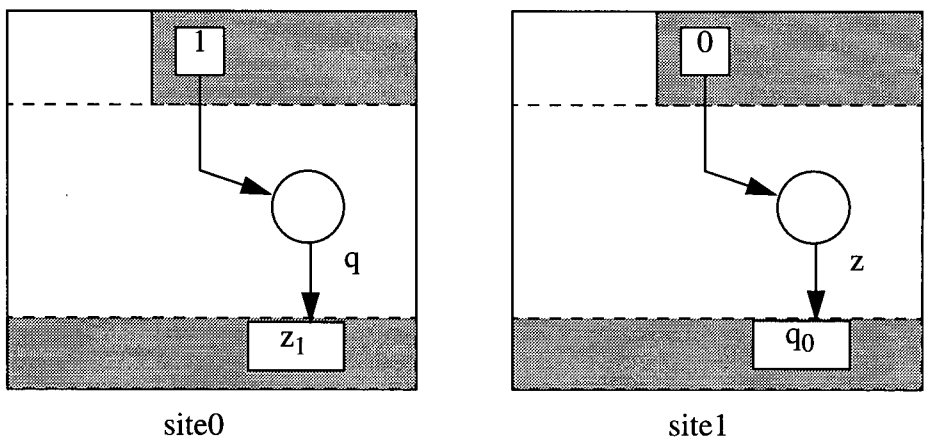


Figure 6. Inter-site cycle of garbage

[SHAPIRO90] proposes to have the local collector mark objects with different colors to differ between those objects accessible locally and those objects which are reachable solely via the ODT. An object that is not referenced locally could then be migrated (according to some total ordering policy) to a site currently referencing it - thereby eventually locating at one site all objects comprising a cycle (enabling reclamation at the next local collection). Although provi-

sion has been made to identify and color objects referenced solely through the ODT (sec. 4.6.2), lacking the necessary migration capability prevents cycle elimination from being carried out.

2.7.3 Indirection Elimination

No consideration need be given to indirection with the current version of Raven. An indirection is encountered in the algorithm when a reference exists to an offsite object, but that object is merely a stub, itself referencing an offsite object. This could arise due to a (stub) reference being passed to a third site or when an offsite reference exists to a local object which subsequently migrates (leaving a stub object in its place).

As mentioned in section 2.7.1, Raven does not have the object migration capability that would lead to indirections being created (e.g. Figure 5b - **w**, now being at site2, is indirectly referenced via site1 from site0). This will be one of the issues to be dealt with, however, when object migration is introduced.

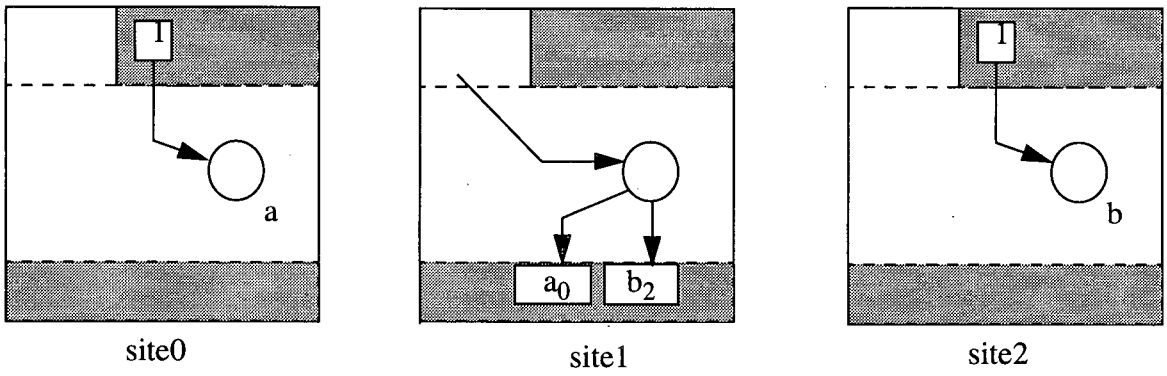


Figure 7a - prior to *a.someMethod(b)* invocation at site1

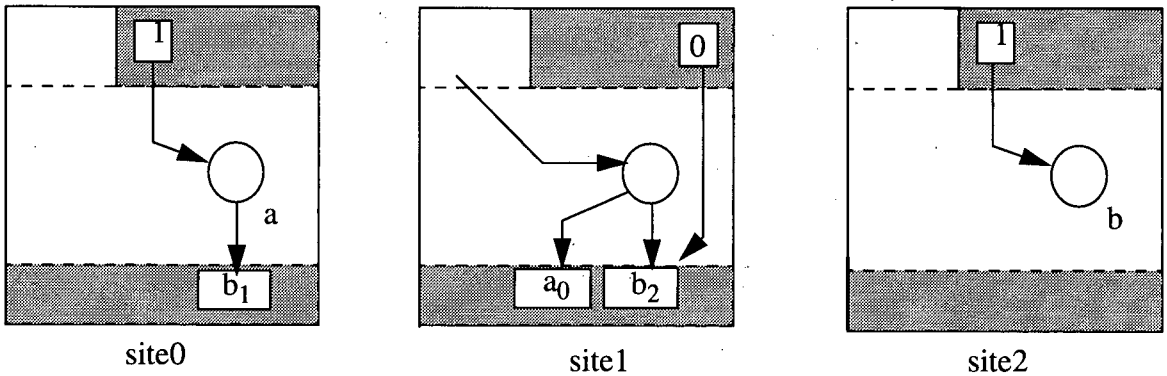


Figure 7b - Indirect Reference to b at site2 created at site0

Likewise, Raven does not allow for indirection creation in the manner of Figure 7. When a reference is sent offsite, the actual information sent [ACTON94] is a globally unique id that is sufficient to generate a stub referencing the true object, not the (stub) object at the sending site (compare Figure 7 to Figure 8). This does have special implications to the implementation which requires notification of the home site (sec. 4.5.4) . [SHAPIRO90] does suggest a number of ways in which indirections can be eliminated — such as piggybacking location information in intersite messages or through the use of an object finder.

We shape our buildings, and forever afterwards our buildings shape us.

— Winston Churchill

CHAPTER 3 Target Environment

3.1 Raven System

Raven is an ongoing project at the University of British Columbia [ACTON92]. A primary goal of Raven is to provide an environment suited to the investigation of issues encountered in the development of object-oriented distributed and parallel systems. Comprising the Raven system are: a compiler, class library, runtime system, and an underlying threads system. Work to date includes research into configuration management of distributed systems [COATTA94], language and operating system support for parallel processing [ACTON94], and development of a comprehensive property scheme for objects in the runtime system [FINKELSTEIN94].

The sections following outline the aspects or subsystems of Raven that either impacted the design of or interact with the global garbage detection scheme.

3.2 Proxies

Whenever there is a local reference to an object resident at another site, that object is represented locally as a proxy. Of prime interest, then, to global garbage detection are the mechanisms in Raven that permit object references to cross site boundaries (Section 3.2.1) and the subsystems involved in processing the references leading to proxy creation (Section 3.2.2).

3.2.1 Communication Protocols

Raven's intended use is for object-oriented (OO) distributed processing; the expectation, then, is that the active (and acted upon) entities in the system will be objects. Now, the manner in which one object 'acts' on another is by invoking some member method (with appropriate arguments). In a distributed context, this requires some method of communicating to remote sites the necessary information. Currently there is one established communication paradigm — remote invocations. Provision has been made, however, allowing the addition of new protocols.
A major decision (see Chapter 4) to be made was whether to write the global detector

1. as an object, making use of existing communication protocols, or
2. as additional code in the existing runtime system, requiring the establishment of a new control-message oriented communication protocol.

3.2.2 Paths to Proxy Creation in Raven

There are three circumstances under which a proxy is generated. The first two occur during remote invocations, when a reference is received at a site and the object represented is not resident at the receiving site (and there is not already a proxy present for that object at the receiving site). The distinction between the two situations is whether or not the received reference represents an object resident at the calling site. For example, in Figure 8, at site1 when *procZ()* is invoked on object **c** with object **a** as an argument, the reference passed to site0 refers to an object present at the calling site; whereas, when the same method on object **c** is invoked at site2 with object **b** as the argument, the reference received at site0 refers to an object not resident at the calling site; it is, in fact, at site1.

The information representing an object in a remote request is sufficient to generate a proxy at the receiving site without any reference to information concerning the calling site [ACTON94]. Indeed, whether or not the object denoted by the reference is resident at the calling site, the proxy generated at the receiving site references the object at its home site and there is no possibility of the situation where a newly generated proxy would be referencing another proxy (i.e. no indirection is possible due to the sending of a reference in a remote invocation). For example, continuing with Figure 8, following the invocation site0 directly references site1 for both objects **a** and **b** while there is no reference to site2 at all.

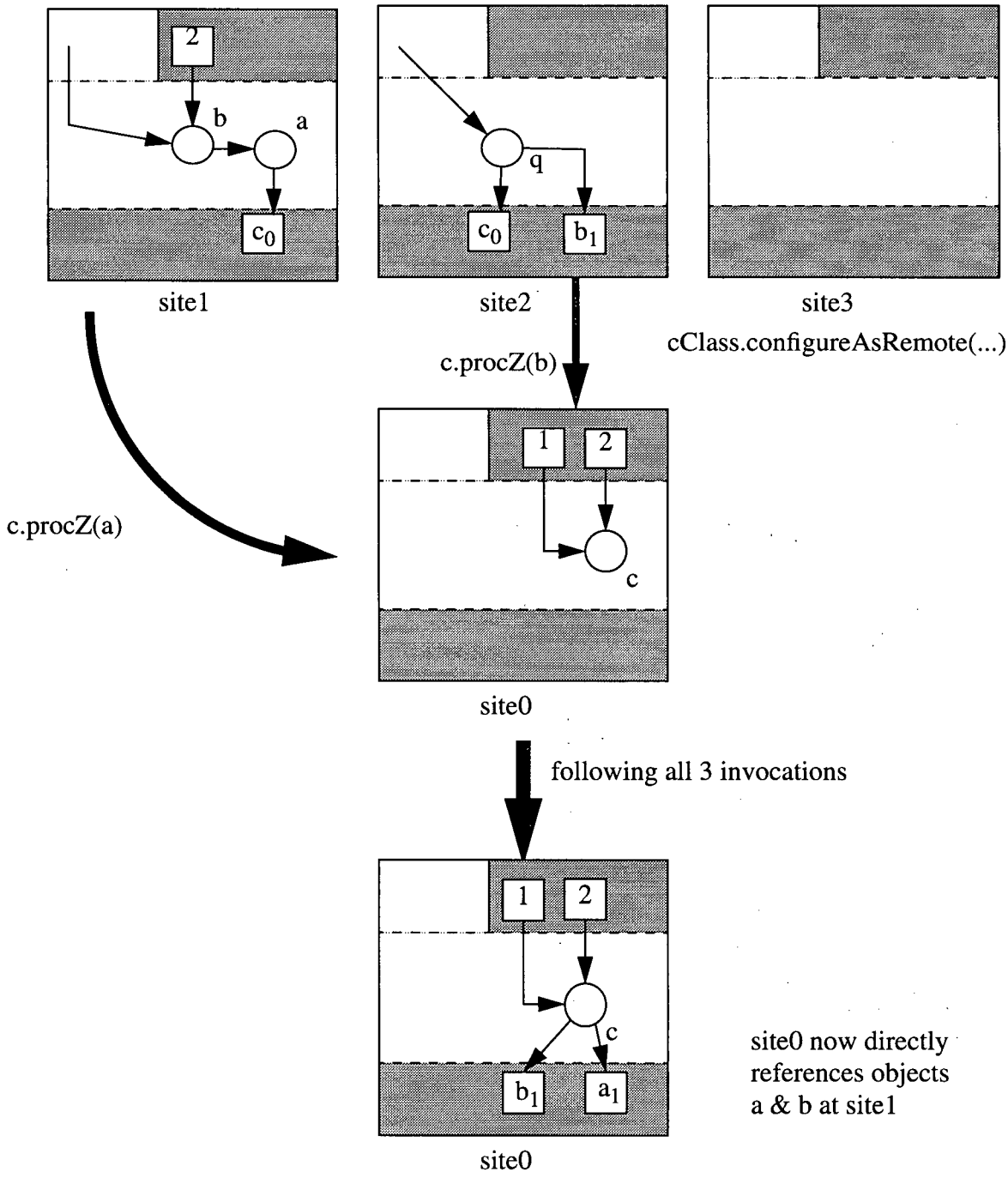


Figure 8. Paths to proxy creation

The third path to proxy generation is through the use of *configureAsRemote()*. Each class in Raven is represented as a metaObject in the runtime system - it is this object that one invokes to obtain a new instance of that class. Every class metaObject will also respond to the method *configureAsRemote()*. This allows a reference to be generated at one site to an object at another site without any direct communication. The arguments to *configureAsRemote()* are the host address and port of the remote site, and the well-known (integer) object ID of the object to be referenced (Raven does limit the per-site number of well-known IDs to one). Furthermore, the object to be so referenced must also have its method *becomeWorldVisible()* invoked (all Raven objects will respond to this method) before any remote invocation could be successful.

The implication of the latter two of these three circumstances is that there may exist in some part of the system references to objects resident at a particular site that that site will most definitely be unaware of. To maintain correctness, as it is not acceptable for those referenced objects to be collected while there is a viable reference in the system, these situations require notification of the resident site - details can be found in section 4.5.4.

3.3 Local Collector

The existing memory management/local collector used in Raven was written by H.-J. Boehm and A. Devers as outlined in [BOEHM88] with some modifications [ACTON94]. Its two most important features, with respect to the design of the global detection scheme, are it is language independent and does its collection via mark and sweep.

As far as language independence was concerned, it was very clear from the start that that should remain the case. How much working knowledge the global detector should have of the local collector's structure and activity was still in question however.

Collection being done by mark and sweep has several implications:

- references stored within the global detector data structures would need to be 'hidden'. This enables identifying memory marked due to true local reference (as opposed to being marked due to references from within the global detector alone)
- those objects 'indirectly' referenced solely from the ODT must remain present through local collections, thus they need to be marked in order to survive the impending sweep

- a mechanism would need to be engineered whereby the global detector is made aware of proxies that are reclaimed due to not being referenced locally - this is an issue since the local collector is unaware of what a proxy is.

3.4 Properties

Raven offers several runtime services - properties - that may be used in the definition of a class or assigned on a per-instance basis [FINKELSTEIN94]. They are :

- Immutable — no modification is possible to the instance data
- Recoverable — at any time during an invocation, instance data can be restored to what it was at the time of the invocation
- Immobile — no migration is possible
- Replicated — object may be duplicated at different sites at the discretion of the system
- Durable — object has a copy in stable storage; guaranteed that updates to instance data are written to disk before control is returned to invoking object
- Persistent — object has a copy in stable storage; no guarantee as to when any updates are written to disk
- Controlled — only one thread may access the object at one time

They may be assigned individually or multiply and, due to their orthogonality, will behave as expected regardless of the combination employed. The next sections outline how properties affect collection.

3.4.1 Impact of Properties on Detection Scheme

For correct operation of the global detection scheme, several of the required data structures would need to be implemented with certain attributes — persisting across incarnations (durable) and limiting access to one thread (controlled). That these properties are readily available for Raven objects would supported any argument for the implementation of the scheme as an object.

Raven's properties might also impact on the algorithm itself. The concern here is whether or not the property would possibly create a situation where an object's status would be compromised in the eyes of the algorithm, i.e. an object being considered garbage when it should not be or (possibly less severe) an object not being considered garbage when it should be. The former is the

more serious as there is no correction possible by the algorithm once the object is reclaimed; whereas, for an object conservatively kept around, it would likely at some future point be detected as garbage and reclaimed.

The following properties will have no impact on the course of the global detection scheme: Immutable, Immobile, Controlled and Replicated — any action involving objects with these characteristics will be correctly dealt with by the algorithm as described. The next two sections will address how the remaining properties affect the global detection mechanism.

3.4.2 Persistent and Durable Properties

Persistent and Durable are very closely related and present similar issues; in fact, since Persistent presents the weaker guarantee, it will require more careful handling by the algorithm. In the following discussion, only Persistent is mentioned since the issues introduced by that property are a superset of those presented by Durable.

The first concern is the timely detection/removal of garbage that is in stable store. The algorithm by [SHAPIRO90] does not concern itself with persistence issues, so special measures must be taken to accommodate stable storage cleanup. Such measures will ensure that when an object that has been stored to disk is no longer referenced within the system, it can be removed from storage.

The second issue is that, for a short period of time, the Persistent mechanism will allow the following scenario: the version of an object in stable storage may not reflect what is currently in volatile space. Imagine an invocation is made on a persistent object; once the invocation returns, there is written to disk a copy of the current instance data, but that copy might not be immediately written, thus the version in stable store does not exactly reflect the current version of the object. The problem here being: if a failure occurs prior to the new instance data being written out, the object upon recovery would be different than when the failure occurred (i.e. possibly referencing different objects). Thus, any referenced objects in the recovered version would need to have been protected from collection while the stable store version referenced them. As an example, in Figure 9, prior to the invocation, object **g** referenced objects **d** and **e**, and, with stable store having been updated, the stored version accurately reflects the current version. The post-invocation situation, however, illustrates that **d** will be collected. But, until stable store is updated, if failure occurs, **g**

upon recovery would be referencing **d**. It is imperative, therefore, that the algorithm account for object references that are present in stable storage by ensuring the referenced objects are not prematurely reclaimed, even though they are not currently referenced (see Section 4.7 for details).

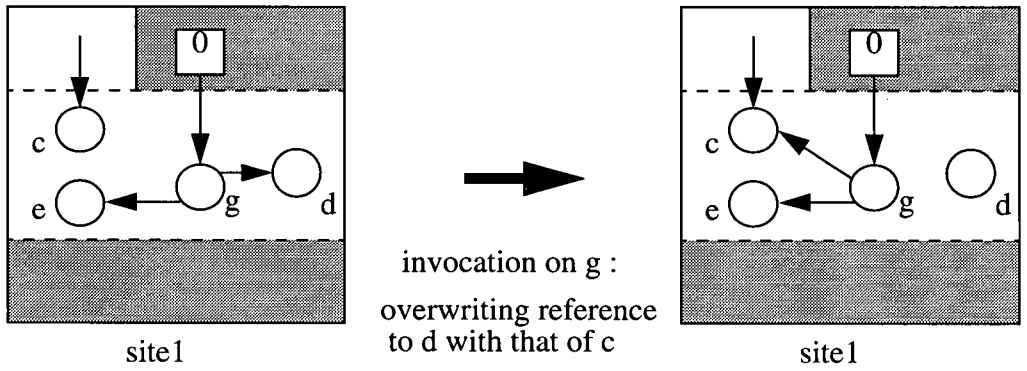


Figure 9. Modifying a Persistent Object

3.4.3 Recoverable Property

With the Recoverable property, the possibility is introduced that modifications to objects made during an invocation can be rolled back to what they were at the beginning of the invocation. If the invocation chain to be recovered also involves any affected data structures of the global mechanism, no special action is warranted.

Circumstances might occur, however, where the recoverable nature of the invocation does not extend to the global mechanism data structures. During the course of a recoverable invocation, though, the data structures employed by the property scheme will maintain adequate references to all objects involved, so local garbage collection will proceed correctly in all cases. Likewise, no special consideration need be given to effects due to the Persistent/Durable properties with regards to recoverable operations; this is because no work is done by the properties system to account for stable storage until after the invocation is complete (i.e. there is no possibility of recovering to the initial state).

A remote recoverable invocation, however, would involve an entry to the local ODT (and to the remote ERT) and an object at the remote site presumably modified in some way to include the reference of interest. If the invocation were to be rolled back at some point, that object would be returned to a state where it did not reference the stub object in the ERT. But the detection algorithm accounts for such a situation in that unreferenced ERT entries are detected during the next local garbage collection and appropriate measures taken to remove the entry and send a removal message to deal with the corresponding ODT entry - thus we are in the normal case.

Now, the object that was rolled back may have distributed the reference in some way that was unaffected by the roll back (e.g. via an invocation on a non-recoverable object) resulting in a viable reference to the stub ERT object. The only incorrect action in this case would be to actively remove the ERT entry. Correctness, then, will be ensured if no action is taken whatsoever to account for remote recoverable invocations thereby allowing the algorithm to simply proceed naturally.

Ho! Ha-Ha! Guard! Turn! Parry! Dodge! Spin! Ha! Thrust!
— Daffy Duck

CHAPTER 4 Implementation Details

In addition to the simplification afforded by using existing Raven mechanisms (remote invocation communication protocol, and several properties - section 3.4.1), the decision was made to honor Raven's object-oriented focus and to implement the global detection scheme as a Raven object. Introducing distributed garbage collection into Raven involved the definition of two new classes, and some modification to the existing code. Figure 10 shows the definition of the GlobalGD class. This chapter will summarize the rationale of that definition as well as any definitions and modifications (to Raven) that were made in support of the GlobalGD class.

```
class GlobalGD {
    behav addSentReference( ... );
    behav addReceivedReference( ... );
    behav updateRemoteTime( ... );
    behav timeStampRequest( ... );
    behav midStream( ... );
}
```

Public interface

```
class GlobalGD {
    ODT : Array[List[ObjRef]];
    ERT :  Array[List[ObjRef]];
    timeStamp : Int;
    highestTimes : Array[Int];
    remoteGDs : List[GlobalGD]
    remoteSites : SymbolTable[Int];

    behav removeReference( ... );
    behav requestReference( ... );
    behav registerRequest( ... );
    behav updateODT( ... );
}
```

Private interface

Figure 10 Definition of Class GlobalGD

4.1 Raven Details

The Raven programming language is object-oriented; its syntax (excluding language support for various features, e.g. parallel processing) borrowing heavily from the C programming language [ACTON94]. While applications are intended to be written in Raven, little of the underlying runtime system is itself written in Raven — rather, it is written in C.

An object reference in Raven is a pointer to a capability structure containing all the necessary class, invocation, lock, property, etc. information required by the runtime system. Also present is a pointer to the actual instance data of the object in question. There is (and currently will only ever be) one capability structure per Raven object per node, thus all objects with a reference to the object in question point to the same structure.

4.2 Implementation Language

The Raven language was used exclusively in the definition and extensively in the implementation of the GlobalGD class. As the runtime system is written mostly in C, in many cases it was either necessary or convenient to implement parts of member procedures in C.

The main uses of C in the implementation include :

- knowledge that Raven object references are really pointers to capability structures was frequently made use of in order to gain access to the current values of various fields (primarily location information)
- access to global variables (e.g. dont_gc), environment variables and UNIX files
- access to runtime routines (e.g. *invoke(), eliminateProxy()*) and to routines written to supplement the memory management/local garbage collector code (*is_it_marked()*)

4.3 General Structure of the GlobalGD Class

Within the GlobalGD class, the required Object Directory Table (ODT) and External Reference Table (ERT) are both implemented as arrays - their component element being lists of type ObjRef. Both Array and List are established parameterized types in the Raven Class Library. The class ObjRef - see Figure 11 - was defined to hold the information describing a reference of interest to the global collection scheme (either an offsite reference to a local object or a local reference

```
class ObjRef {

    objRef     : cap;      // the reference
    hid        : Int;      // remote host address
    lid        : Int;      // remote port
    timeStamp  : Int;      // when ref was last sent
    marked     : Int;      // did local collector mark it

}
```

Figure 11. Definition of Class ObjRef

to an offsite object).

Any one list in the ERT will denote all the local references to objects at one (and only one) particular Raven site. Likewise, the list at any location of the ODT would reflect all of the references to local objects currently thought to exist at one remote site. Furthermore, given a particular index, the lists depicting the current local state of the intersite relationship (with one remote site) could be located using that index in both tables.

The list remoteGDs contains references to GlobalGD objects at remote sites; highestTimes contains the highest timestamps received from those sites. Extending the relationship of the lists at the same index into the ERT and ODT to include all four data structures, the entries at the corresponding index into remoteGDs and highestTimes would contain the appropriate reference/data for the same remote site. Tying these four data structures together is the symbolTable remoteSites – the key to any entry is the string composed from appending the host address and local port (which in combination uniquely define any Raven site); the value stored with that key being the table/list index appropriate for that site.

This structure is convenient as there are different lookup methods required depending on the data available — during a remote invocation, most often the information available would be limited to the originating site's address and port (or sometimes the address and port of a third site), while during an invocation on the local GlobalGD itself, one of the arguments is often a reference to the GlobalGD of the requesting site.

4.4 System Initialization

There is one instance of GlobalGD per Raven environment and it can be accessed (codewise) by the name *globalGD*. During system start-up, these are the steps in the initialization of the local GlobalGD instance:

- create data structures
- create universal site proxy
- access (remote) registry and obtain references to the GlobalGDs of the other currently co-operating Raven environments (section 4.4.1)
- in turn, send each of these sites a reference to self in a registration message to be followed by entering this site into local data structures

To ensure correctness, the initialization of the GlobalGD must take place before the actual program begins executing (i.e. in Raven, prior to invocation of the *start()* routine of the class Main). Placing the call to the GlobalGD constructor before the call to *start()* in the system setup routine resulted in the generation of a fault: for completion of initialization, several remote calls must be made requiring calls to the network communication portion of Raven — part of which attempts to record the value of the current thread. At this point in the setup, however, there is no notion of a current thread, so failure follows. Explicitly creating a thread and 'attaching' it to the GlobalGD constructor succeeds in itself, but no threads start until the (higher level) initialization routine completes (after also creating a thread for the *start()* routine). Once the constructor thread goes remote, it is suspended (awaiting response to complete initialization) allowing the *start()* routine to become active thus breaking the condition necessary for correctness.

To coerce the system to perform as required, the same thread, then, must be used for both the GlobalGD constructor and the program *start()* routine (or at least the latter cannot be allowed to run until the former completes). This was simply solved by having the system *init()* procedure start a thread for an intermediate procedure which invokes both the constructor and subsequently the program *start()* routine — effecting the required timing.

4.4.1 Location of the Registry & Identification of Peers

For the initialization of the local GlobalGD to complete, it must determine the other sites that it will be in contact with. This is accomplished via a central registry service — a separate program,

written in Raven, that consists simply of a list of the GlobalGDs within a system. During initialization, each GlobalGD instance adds itself to the end of this list and then iteratively requests each element of the list[1]. Upon receipt of each of these references from the registry, but before adding the new reference/site to its own various lists, a *registerRequest()* invocation is made on that reference. This notifies the remote GlobalGD that there is an additional Raven environment in the system. The notified GlobalGD would then add the new site to its own lists and record the existence of a reference to itself at that new site.

As the central registry is itself a Raven program, it will go through the same initialization steps as any other program – i.e. GlobalGD initialization including an attempt to contact the central registry. Obviously, for the registry program itself this would be incorrect behavior. An intermediate step in the initialization process prevents this. Prior to going offsite to contact the registry, an environment variable RV_UNIVERSAL is checked for: if defined, it signifies that this is the registry and GlobalGD initialization proceeds on that basis — actually, going offsite being avoided, the remaining activity consists of the host IP address and local port of the registry program being displayed. Subsequent Raven programs intended to be part of the same system would then access this information (via the environment variables RV_UNIVHOST and RV_UNIVPORT) to determine the location of the registry service. Given that location information and the well-known id of the registry list, each initializing GlobalGD would then use *configureAsRemote()* to generate a proxy for the registry. This allows the addition of self, and access to the other currently operating GlobalGDs.

One note on initialization: to effect all the steps correctly, an *isInitialized* flag is initially set to FALSE – member methods return immediately while this condition holds thus preventing any premature access of member tables/lists in response to the sending of local or arrival of remote references while initialization proceeds.

4.5 Interaction with the Runtime System

The decision to make the global detector a true object allowed the use of the existing remote invocation communication protocol. The aspects of the global detection algorithm that conse-

1. During program/class development, it may be desirable to execute/test code that is not distributed. By appropriately defining the environment variable RV_SOLO, GlobalGD initialization omits accessing the central registry since that program need not be executing.

quently needed to be addressed by the runtime system are:

- including a timestamp in each message sent offsite
- passing to the global detector the timestamp for each message received
- notifying the global detector of each local reference sent offsite
- notifying the global detector of each offsite reference received

4.5.1 Timestamps

An integer field was added to the data structure transmitted in a remote invocation to allow the timestamp to be included. While the message is being prepared for network transmission, an invocation of the GlobalGD method *timeStampRequest()* is made — the return value is the time to stamp the message with. Similarly, after a message is received and the contents are being processed, the runtime system will invoke the *updateRemoteTime()* method of the globalGD so that the appropriate entry in the array highestTimes can be updated to the received timestamp.

4.5.2 Intersite Reference Transfer

When a reference is sent offsite (e.g. as an argument in a remote invocation) what is sent is a copy of a global id (gid) structure which uniquely identifies that object. This gid structure is normally stored as part of the capability structure.

4.5.3 Sending a Reference

During assembly of a remote request, each time that the runtime system detects that a reference to a local object is being sent offsite, it invokes the GlobalGD method *addSentReference()* allowing an appropriate entry to be made to the ODT.

4.5.4 Receiving a Reference

After receiving a remote request, any contained global ids need to be resolved to local objects. The three possibilities for what the gid represents are:

1. a local object (e.g. the target object to be invoked)
2. an object resident at the calling site
3. an object at a third site

In cases 2 and 3, the global id will be resolved to a proxy object. The remote request handler initiates resolution and invokes *addReceivedReference()* upon successful return.

During resolution, the runtime system contacts the GIDManager [FINKELSTEIN94], which among other things maintains a data structure mapping global ids to currently existing proxy objects. If there is no proxy for that global id, a new proxy is created and appropriately entered to the data structure (Note: similar to object references in the GlobalGD data structures, references to proxies and objects within the GIDManager must also be encrypted to avoid these references resulting in an object being marked). This proxy creation occurs irrespective of whether the reference is to the calling site or to a third site. It is, in fact, only when the globalGD is notified of a received reference that it can check whether it references the calling or a third site. In the latter case, for correctness' sake, the local globalGD must inform the remote global detector of its reference to that site. The globalGD does this by invoking the *requestReference()* method of the globalGD of the site referenced. Once that remote invocation returns, the entry can be made to the ERT and control returned to the runtime message handling system allowing the original remote request to proceed.

4.6 Interaction with the Local Collector

Although the work of the local collector and distributed garbage detector are co-operative, they are disjoint. To maintain the integrity of the local collector, it was desirable that it have minimal interaction with any higher level code and to have no reference to or knowledge of the way in which the memory being allocated was used. If the boundary was to stay language independent, there could be no mention of proxies, capabilities, invocations on GlobalGDs, etc.

Just as the local collector was to remain free of outside reference, so the global detector was to have no part in the action of the local collector. So, in order to facilitate marking the GlobalGD class member lists and arrays, the GlobalGD is simply considered to be part of the local root allowing the local collector to mark it in the normal course of marking memory — i.e. no special consideration or treatment was engineered in the local collector to bypass marking of the global detector data structures. Recalling the issues related to the local collector listed in section 3.3 :

- any references stored within the GlobalGD are encrypted
- those objects 'indirectly' referenced solely from the ODT, in order to remain present through local collections, are marked in order to survive the impending sweep
- proxies that are not referenced locally are eliminated from the local proxy table follow-

ing local collection, allowing notification of the once-referenced site of the dropped reference

The latter two were addressed by temporarily exiting the local collector after the mark phase. This provides an opportunity for the globalGD to initiate end-of-mark-phase activities (see following section).

In addition, it was necessary to add only one procedure to the local collector code. Its action is to determine whether a particular memory location (e.g. the location being indirectly referenced by either the ODT or ERT) is marked.

4.6.1 Between Mark and Sweep Phases

Once the marking phase of the local collection is complete, control is passed temporarily to the GlobalGD. It is at this time that updates are made to the entire ODT and ERT. Recall that any references to local objects in the ODT and ERT are encrypted. This is done so that when the local collector marks the GlobalGD (and hence the ODT and ERT), local objects referenced therein are not marked; this being done to allow differentiating between marks due to true local reference and marks due to reference from the GlobalGD structures. As far as the ODT is concerned, this distinction is important in the detection of global cycles. As seen in Chapter 2, cycle elimination cannot proceed due to the current lack of migration capability, but preliminary steps are already in place.

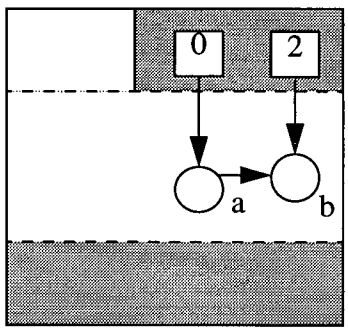
[SHAPIRO90] suggests that the local collector directly mark objects in a different color if they are referenced from the ODT. This was rejected for two reasons:

1. the local collector was to remain free of reference to Raven and GlobalGD constructs
2. the local collector marks an area of memory using only 1 bit — thus one color. Furthermore, that one bit is not part of the memory being marked, but rather is part of an array in a complex control structure used in the allocation of numerous identically sized memory chunks. To reduce complexity, coloring should be conducted outside of the local collector.

4.6.2 Coloring the ODT and ERT

In between the mark and sweep phases of the local collector, objects 'referenced' in the ODT

are checked to see if they already are marked – indicating they are being directly referenced locally. Any that are already marked have their ODT entry tagged with GD_GREEN. If an object 'referenced' by the ODT is **not** yet marked, then it is referenced solely through the ODT (or additionally via another object, itself accessible solely through the ODT). These are tagged GD_AMBER and will later be marked (via an existing procedure in the local collector). Care must be taken to inspect *all* of the ODT entries before marking *any* of them to avoid the scenario in Figure 12 resulting in an object being marked (via the ODT) and subsequently being tagged GD_GREEN instead of the correct GD_AMBER.



during ODT inspection, if object a is marked prior to b being inspected, then b will be incorrectly seen to be marked

Figure 12. Premature Marking of ODT Entries

Also, before any of the ODT entries are marked, all of the ERT entries are inspected. If the objects represented are already marked, their entries are tagged GD_GREEN; if not, they are tagged GD_RED. Unlike ODT entries, no explicit marking will be done of unmarked objects 'referenced' by ERT entries until after being inspected one more time. This different treatment is necessary, since if, at the end of the garbage collection process, the objects referenced are still not marked, the entries will be removed as they reflect a reference to a remote site which is no longer accessible locally (i.e. there is no local reference to the stub (proxy) representing the remote object).

Once the ODT marking is complete, the ERT entries are again inspected and those that were tagged GD_RED but that are now marked (as a result of marking the ODT entries) are upgraded to GD_AMBER. It is at this time that those ERT objects that are not yet marked are now marked in order that the proxies survive the sweep. Control is then returned to the local collector for the sweep phase.

4.6.3 After the Sweep Phase

When the sweep phase is done and local collection is complete, the following cleanup measure is taken:

for any ERT entries (proxies) tagged GD_RED, indicating no local reference (directly from the root or indirectly via the references in the ODT), their corresponding entry in the proxy table is removed, a removal message is sent to the once-referenced site, and the ERT entry is deleted to finalize the ERT cleanup.

At the moment, without the possibility of object migration, there is no work done with the ODT after local collection is complete.

4.7 Raven Properties - Persistent and Durable

In response to the demands placed on the global detection scheme, two data structures were added to the definition of the GlobalGD class:

- stableStore : a list of objects that are currently in stable storage. Also included in this structure is a list of all the objects referenced by the stored version
- toBeStored : a list of those objects which have had an invocation performed on them and which have storage pending

These are the steps to be taken during storage (modified from [FINKELSTEIN94]):

- a storable object is directed to encode its instance data by its StorageManager
- during this encoding, note is taken of those storable objects referenced and at the end of encoding this information is passed to the GlobalGD: the reference to the stored object along with the list of referenced objects is then placed in *toBeStored*
- the object returns its encoded version to its StorageManager
- when the StorageManager effects the object's storage, the GlobalGD is notified and the record for the appropriate object is moved to *stableStore*, thus overwriting the record for the previously stored version.

Recall that the goals surrounding stable store (section 3.4.2) are :

1. cleaning out unreferenced objects

2. preventing premature removal of those objects not currently referenced in volatile space, but that are referenced from a stored object; thus ensuring the presence of referenced objects in the post-failure environment.

How, then, are these tasks to be accomplished? During the phase of local collection when control is passed to the GlobalGD for post-mark-phase activities, additional work is carried out. After the ODT entries are marked, the entries in the data structure *stableStore* are inspected for the first time. Any objects referenced therein that are not currently marked are tagged GD_RED; currently marked objects are tagged GD_GREEN and the objects in the associated list (depicting objects referenced from the stored version of the object) are in turn marked. Once all of the elements of *stableStore* have been inspected, the following step is repeatedly carried out until there is no further reduction in the number of entries tagged GD_RED :

- make a pass through the list, and inspect each entry tagged GD_RED: if it is now marked, tag it GD_GREEN and mark the associated list of objects referenced from stable store

What this process accomplishes is essentially extending the marking phase of the local collector into stable store, which is task #2 listed above. It also serves to identify (for task #1) those currently unreferenced objects that have previously been stored (which are those that remain tagged GD_RED at the end of the process). All that remains, then, is to remove these objects from stable store — which can be accomplished via the established mechanism employed by the property scheme to remove objects from stable storage when they are no longer deemed storable (i.e. when the inherited storable property is no longer effective, the delete routine is called for the TDBMManager [FINKELSTEIN94]).

4.8 Raven Properties - Recoverable

No special measures are necessary to address any troublesome issues arising due to recoverable invocations. The one concern was that a remote ERT entry might be erroneously rolled back while there was a viable reference at that site. That the entry to the GlobalGD ERT is instigated by the runtime system outside of the recoverable invocation chain precludes that entry from being rolled back. This situation reduces, in fact, to the normal case where the ERT entry will not be removed until there is no local reference to it.

4.9 A Lesson in Remote Invocations in Raven

updateODT() is the name of a GlobalGD class routine. It is the implemented version of the currentERT message (section 2.4.2) which the global detector at, say, site1 uses to inform the detector at site2 of the references existing from site1 to site2 (i.e. that portion of ERT_1 pertaining to site2).

What follows is a recounting of the development of the *updateODT()* routine to its current version. This is intended to illustrate a little of the current implementation experience in Raven.

One site initiates the updating of a remote site ODT by invoking the *updateODT()* routine on the proxy for the GlobalGD for that site (recall that the structure remoteGDs contains all such references to remote instances of global detectors). Originally, one of the arguments to the procedure was a direct reference to the ERT list of the invoking site (so the elements of the list were of type ObjRef). The algorithm further called for an iteration through the ODT list (of the invoked site) comparing the ObjRef items therein with each of those contained in the argument list – thereby doing n remote invocations (fetches) for each of m local ODT items (as opposed to doing m local operations for each of n remote invocations). The ensuing flurry of network messages involved dereferences, remote invocations, as well as anticipated and unanticipated comparisons. To cut down on the amount of network traffic an attempt was made to employ the keyword **copy** and thus send a (deep-)copy of the argument list to the target site. As the ERT list contains ObjRef objects (part of which is the 'encrypted' proxy reference), a direct copy of that structure was not feasible. Regardless of the base type of argument employed (List[cap], Array[cap], etc.), invariably a fault was generated; highlighting that no provision had been made for the correct network preparation of a proxy — in Raven, the class of a 'proxy' object is actually the class of the object it represents, so the code would attempt to interpret the various <garbage> fields of the 'proxy' instance data as valid object references and try to encode those also. This was amended to avoid tracing spurious fields and a remote message was sent and received satisfactorily with the object being successfully reconstructed. However, the invocation would then be initiated, but as the **copy** keyword in the function header does not apply solely to remote invocations, that generated a *deep-copy()* call for that argument. Now, part of that argument was a reference to the local GlobalGD — with all its composite lists, ObjRefs, encrypted references and the like — which only generated more faults when its copying was attempted. At this point, rather than modifying copy

semantics for remote invocations possibly introducing unwanted side effects, copying was abandoned and the entire procedure was rewritten so that the argument was of type List[cap] (thus a single reference was sent to the destination site), allowing the invoked site to simply invoke the *size()* method on the list, and then extract that many elements from the list (while iterating on the local ODT list) — resulting in n+1 intersite messages (that is the number one-way, thus 2n+2 total messages) for a significant reduction in network traffic.

He who stops being better, stops being good.
— Oliver Cromwell

CHAPTER 5 Now and Then

5.1 Achievements

A comprehensive global garbage detection scheme has been successfully integrated into the Raven system. The secondary goal of having the algorithm be non-Raven-specific was also met.

Comprehensiveness can be shown by testing various facets with manufactured scenarios designed to highlight weaknesses. Testing was first conducted to guarantee that the (encrypted) references in the global data structures would effect the retention of otherwise non-locally referenced objects.

Testing was also carried out to determine that simple reference passing guaranteed proper recording of the departure and arrival of the reference. A more complicated scenario was devised for the correct handling of third-party references. Other situations were designed to elicit a variety of outcomes including the over-writing of unique remote references (thereby creating garbage at the home site), removal messages being accepted and rejected, and update messages accounting for garbage persisting as a result of conservative behavior – all of which gave positive results.

Verification consisted largely of snapshots of the global mechanism's data structures; any local issues (e.g. leakage) were presumed to be the responsibility of the local collector and outside the scope of this thesis.

Although the algorithm provides for robustness in volatile spaces, this implementation was directed towards the persistent case as Raven is intended to have such a capability. Though communication failure does not hinder any local progress nor interaction between any connected sites, the present implementation's behavior in the face of site failure is not the ideal. For the case where a failure removes the last viable reference to a remote object, that object becomes garbage but there is no mechanism implemented to provide for its collection, as the ODT entry will persist. It is presumed that the process will be restarted, however, and once stable storage is added the ODT and ERT will be intact upon recovery and the normal situation returns.

Performance measures and concurrency were not the focus of the work, but do warrant comment. When the local mark-and-sweep collector is running, there is no other action in the system, as all processes are suspended. Thus, there is no concurrency. As there was no optimization of the global-local components interaction after the local mark phase, the delay to process resumption could possibly be reduced should it ever be considered an issue. Also, as there is no concurrency, expedience (reclamation keeping pace with allocation) is rendered a non-issue. Very few background messages are generated, thus most of the operating cost of the global mechanism can be attributed to the invocation costs involved in the maintenance of the data structures.

5.2 Evaluation of the Algorithm

There are two ways to evaluate the algorithm:

1. as an example of a distributed garbage collection scheme, in which case the characteristics listed in the Introduction (chapter 1) can be used as a yardstick
2. the suitability of the algorithm for the adopted use. Discussion here would involve demands placed on the language by the algorithm, and similarly, demands made on the algorithm not provided for in the original version.

1. [SHAPIRO90] presents a solution of conceptual elegance and simplicity. The algorithm presented is, nonetheless, robust and provides mechanisms for recovery for both the volatile and persistent cases. References and objects in transit are dealt with appropriately. Local and

global collection may proceed while there is a site or network failure (i.e. unaffected sites can pair-wise proceed globally). There is some conservative behavior, but backup measures account for and deal with any garbage which may persist as a result. Cycle elimination across site boundaries requires (atomic) object mobility. The authors do recognize that although general mobility is possible, some objects may have their mobility restricted, thereby rendering their two color marking scheme ineffective.

2. The algorithm adapted well to Raven; a complete implementation being prevented (see section 2.7 for specifics) due to current limits in Raven, namely the lack of migratory ability and persistence. Lacking migration capability prevents the collection of intersite cycles. Related to this, one demand the original algorithm makes on the target language is the responsibility placed on the local collector to use two different colors during the marking process to distinguish locally referenced objects from those solely referenced from remote sites. This might require a major reworking of the local collector in the target language (modifications to the algorithm avoided this in Raven). As mentioned previously, the algorithm also requires that object migration be atomic.

   The main modification to the algorithm forced by the target environment was the attention required to ensure correctness when dealing with proxy creations (section 3.2). References passed from site to site need not point back to the originating site but will rather point to the home site of the object. No additional mechanism or message types were necessary, merely additional steps added. Although persistence is allowed for, there are no details given to the qualities thereof. Special consideration had to be given to Raven persistent objects with respect to global collection (section 3.4.2) due to the temporary inconsistency possible between the object and its stored version.

5.3 Future Enhancements

As features are added to the Raven system, the developers will have to assess how existing constructs will be impacted by the additions. The following sections will provide some insight into several of the foreseeable enhancements that will affect the global detection scheme as implemented.

5.3.1 Migration

By far, the most influential change to Raven will be the introduction of migratory ability for objects. Care must be taken when the object leaves the originating site that allowance is made for possible reference(s) to it — calling for a proxy and ERT entry. Also, any local objects to which the object has reference must have entries placed in the ODT. At the destination site, allowance must be made for possible reference from the originating site — calling for an ODT entry. Any (now remote) objects to which the object has reference must also have entries placed in the ERT.

Cycle elimination will be able to be carried out with consequence to the post-mark and post-sweep activities of the global detector - namely migrating objects to sites that reference them.

Consideration will also have to be given to the possibility of introducing indirections due to migration in the manner of Figure 5. Possible indirections could either be searched for and resolved during the actual migration process or they could be dealt with as they are detected post-migration. As there are a variety of possible confounding circumstances, including a reference to the migrating object being in transit at the same time the object itself is migrating, the latter seems to be the more attractive choice. [Shapiro90] suggests eliminating indirections by piggy-backing location hints or through the use of an independent object-locator.

5.3.2 Object Encoding for Stable Store

As mentioned in section 4.7, two of Raven's properties, persistence and durability, require special action by objects that are encoding their instance data — namely, that they inform the globalGD of those objects which they currently reference. An appropriate encoding routine is provided to all objects in Raven, through the Object class. However, for complex objects that require special handling, that encoding routine may be overridden by a method appropriate for that class. This places the responsibility of fulfilling the requirements of the GlobalGD mechanism on the developer of that class — failing to do so will leave the global scheme in an inconsitent state.

5.3.3 Other Properties

Provision has been made in the property scheme to allow for four user-defined properties. As they have no limitations placed on their scope or implementations whatsoever, it is impossible to predict how they might interact with or affect the global detection scheme. To paraphrase a cautionary cliche : Let the user beware!

5.4 Related Work

[JUUL93] describes the implementation of global garbage collection for Emerald. The solution, although somewhat intricate, achieves the desirable qualities listed in the Introduction (and more).

It is comprehensive and achieves cycle collection with no extra effort or mechanism. There are two collection processes (one local and one global) that work simultaneously and with minimal interaction and synchronization. The local collector sweeps garbage from a previous global tracing. The global mark phase proceeds via a generalization of the Emerald faulting mechanism, whereby any protected (has been *shaded*) object invoked will generate a collection fault, the object is then marked and any objects it references are in turn protected (*shaded*) and put into a set of to-be-marked objects. There is one set each for resident objects and non-resident objects. When a protected non-resident object is invoked, a message is sent to the remote site for the object to be marked. After being marked, objects are removed from the set they are in. When the sets are empty, that site will attempt to determine whether a global termination point has been reached by sending *status* messages to the other nodes. The algorithm is concurrent in that there is no perceptible halting to any processes (to allow for any aspect of collection) as all action will not exceed the normal timeslice afforded to any process. This is facilitated by the incremental nature of the mark phase allowed by the faulting mechanism.

Robustness is exhibited in that local collection can proceed if the global mechanism is stalled or if there is a partial failure. The global mechanism is able to terminate at a time when not all nodes are available (it does require a minimal level of pair-wise availability however). Note that any node may terminate the mark-phase — there is no central mechanism required. The author's only

self-criticism is that the algorithm does not guarantee expedience. This lack of guarantee is partly attributable to the global synchronization algorithm's potential delay in the face of the zipper problem - Figure 13 - where site boundaries are repeatedly crossed in the global mark phase.

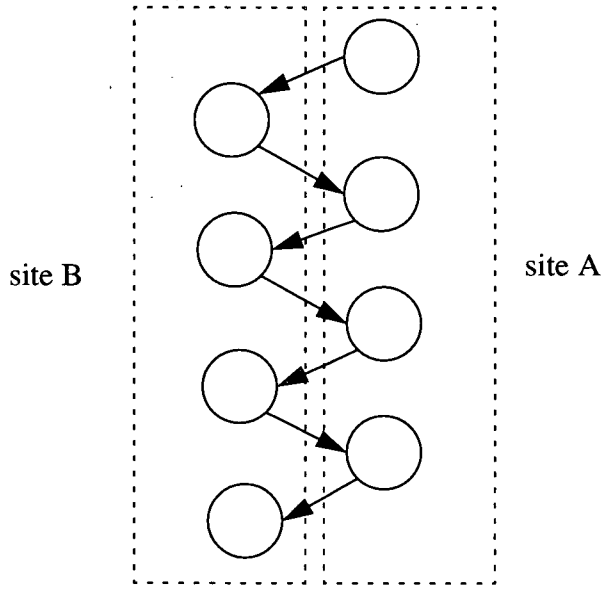


Figure 13. The zipper problem

Liskov and Ladin present a distributed collection algorithm [LISKOV86] based on their highly available centralized service (a logically centralized collection of replicated servers (*replicas*) presenting a consistent view to its clients). The primary requirement of the service is that the property of interest once achieved be stable — garbage collection does satisfy this criteria since once an object becomes garbage it remains garbage. The individual nodes may collect by any scheme they choose; the distributed work proceeds by tracing. Following local collection, nodes provide one of the replicas with information identifying which objects are reachable from its root, which references might be in transit, and which objects are reachable via its *inlist* (a list of objects the owner has made available to other nodes) yet not accessible from its root. The latter information is conveyed to the service using object pairs which describe the graph present on the node (i.e. similar to Figure 1, describing the reachable objects from a subset of the inlist). The service would then undertake to update its picture of which objects are globally reachable and those that are inaccessible. The replica then would pass along to the other replicas its current state using a *multipart timestamp* to guarantee proper sequencing. Nodes learn about which local objects are

now collectible by querying one of the replicas. The algorithm presumes stable storage (for inlist, and recording in-transit references) and that nodes will recover with objects intact. Migration of objects would require an extension to the algorithm, as do intersite cycles — an additional cycle detection algorithm must be run at one of the replicas. A point in favor of the proposal is that the global work is removed from the nodes, thus increasing concurrency with other processes. Other benefits are that there are few messages required (efficient), and that the replication provides robustness. A drawback is that the local collector must be extended to construct the information lists required by the central service. Also, given the potential size of these lists and the number of objects involved on a global scale, the updating process at a replica might involve a considerable amount of time or inter-replica communication, thus questions are raised about the timeliness of the collection process considering that once a node presents current information to the service, it is likely to soon after make a query as to the status of its objects.

[BIRRELL93] presents the solution to the garbage collection problem for the Modula-3 network objects system, and a proof of its correctness. It is based on a reference counting system and has aspects similar to the algorithm employed in this thesis [SHAPIRO90]. One area that differs is in the ability to collect distributed cycles of garbage; as reference count systems are unable to collect cycles, and the network objects system does not allow for object migration, there would be no possibility of cycle collection as in the approach proposed in [SHAPIRO90].

Each site maintains an object table with strong or weak references to an object – a strong reference is sufficient to prevent collection, a weak reference is not. If an object is a *surrogate* (to a remote object), it has a weak reference from the object table. A *concrete* object will have a strong reference from the object table. For each concrete object, a list, *dirtySet*, is kept with process ids of those processes with a reference to the object. The first time that a process receives a reference to a remote object, it notifies the *owner* (home site) of the object with a message to add it to the dirtySet, and, upon successful return of the message send, creates a surrogate. When the surrogate is no longer accessible, a *clean* message is sent and the process id would be removed from the dirtySet. When the dirtySet becomes empty, the object would be eligible for collection (i.e. if no local reference to it existed). Communication failures and race conditions are addressed via distinguished values in the object table, sequence numbers, and special messages. Regardless of

these measures, there is a requirement, upon failure of a dirty message, that the home site of the concrete object retain (until one or the other process dies) the sequence number of the *strong* clean message sent to counteract any possible effect the failure might have had. Owners monitor the liveness of processes identified in dirtySets allowing reclamation of garbage when processes which hold references to concrete objects fail. This, unfortunately, also allows the collection of reachable objects in the face of extended communication failure.

The main motivation of the generational reference count scheme [GOLDBERG89] is the reduction in the number of messages required. As the trade-off is an increase in the amount of space required to manage the reference count, it is intended for systems where the cost of communication is relatively high. Each reference has associated with it an integer *generation* and a *count* of the number of times that specific reference has been copied. Each time a reference is copied, the generation of the copy is set to be one higher than that of the reference copied. The count of the copy is initially set to zero, and the count of the copied is incremented by one. When the first reference is created (i.e. at object creation time), that reference is given a generation of zero and a count of zero. When a reference is discarded, a message containing the generation number and the count is sent to the original object. The object maintains a *ledger* associating the generations with the total number of copies made for that generation; the ledger is initialized to

$$\text{generation}[\,0\,] := 1$$
$$\text{generation}[\,i\,] := 0 \qquad \text{for all } i > 0\,.$$

When a discarded message is received, the ledger's count of copies for that generation is decremented by one, and the count for the subsequent generation is incremented by the count in the message (recall that this value indicates the number of copies made from the discarded reference). When the count for each generation in the ledger reaches zero (some may go negative for part of the time), the object can be reclaimed. The work provides a proof, and limits are discussed (e.g. overflow of a field in a reference or ledger is possible) with direct comparisons to the weighted reference count method. There is no attention given to issues of network or process failure (robustness). On the other hand, with the low communication volume and lack of interruption to processing time (no tracing required), concurrency rates quite favorably.

As with all decisions, the final choice of which scheme to implement is dependent on the parameters of the problem. Given the wide-range of approaches presented in the literature (some of which have actually been implemented), there would likely be a solution available that would not require extensive additions or modifications, while, at the same time, not compromising the target system's characteristics or goals. A comparison of the surveyed works to the solution implemented in this thesis is illustrative.

Given the dissimilarity between the network object model [BIRRELL93] and Raven (e.g. Raven is intended to have migration), the general weaknesses of the reference counting scheme (e.g. the inability to collect cycles), and the consequences of some of the design choices made in [BIRRELL93] (i.e. reachable objects can be collected in the face of extended communication failures, and the possible necessity of retaining sequence numbers indefinitely) would argue against adoption of that scheme for Raven.

Generational reference counting suffers from the same inability to collect cycles seen in other reference counting schemes. Although concurrency promises to be very good, difficulties with comprehensiveness when processes fail would need to be overcome in a more fully realized implementation.

[LISKOV86] presented a solution requiring major modifications to the local collector, in a system that did not allow for object migration. Furthermore, suggestions of improved performance and low message overhead might well not be realized when the system is scaled.

The solution implemented for Emerald [JUUL93], although intricate, scores very well in the measure against an ideal distributed garbage collection mechanism, and appears to be worth the effort. Unfortunately, the central mechanism, incremental marking enabled by faulting on invocation of a protected object, is infeasible in Raven as there is no way to protect individual objects.

Of the schemes looked at, the implemented algorithm [SHAPIRO90] provides the best overall compliance to the problem specifications - provision of comprehensive and robust distributed garbage collection *for Raven*. The suitability derives largely from its simplicity and the lack of intru-

sion into the system (i.e. the ability to be implemented with minimal demands on and modification to the target system), while making provision for key elements already present or anticipated in the system (e.g. migration, persistence).

REFERENCES

[ACTON92]

Donald Acton, Terry Coatta, and Gerald Neufeld. *The Raven System.* Technical Report TR92-15, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, 1992.

[ACTON94]

Donald Acton. *Unified Language and Operating System Support for Parallel Processing*. Ph.D. Thesis PHD094-ACTO, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, April 1994.

[BOEHM88]

Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807-820, September 1988.

[BIRRELL93]

Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. *Distributed Garbage Collection for Network Objects*. Technical Report no. 116, Systems Research Center, Digital Equipment Corporation, Palo Alto, California, December 1993.

[COATTA94]

Terry Coatta. *Configuration Management in a Distributed Environment.* Ph.D. Thesis PHD094-COAT, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, April 1994.

[FINKELSTEIN94]

David Finkelstein. *Object Properties: A Mechanism for Providing Runtime Services to Objects in a Distributed System*. M.Sc. Thesis MSC094-FINK, Department of Computer Science, University of British Columbia, Vancouver, British Columbia, October 1994.

[GOLDBERG89]

Benjamin Goldberg. Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* pages 313-321, Portland, Oregon, June 1989.

[JUUL93]

Niels Christian Juul. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald.* DIKU-rapport 93/1, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, February 1993.

[LISKOV86]
Barbara Liskov and Rivka Ladin. Highly Available Distributed Services and Fault-tolerant Distributed Garbage Collection. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 29-39, Calgary, Alberta, Canada, August 1986.

[SHAPIRO90]
Marc Shapiro, David Plainfosse, and Olivier Gruber. *A Garbage Detection Protocol for a Realistic Distributed Object-Support System.* Rapport de Recherche INRIA 1320, INRIA-Rocquencourt, Paris, France, November 1990.