Lazy, per Cluster Log-Keeping Mechanism for Global Garbage Detection on Amadeus

Sylvain R.Y. Louboutin* Vinny Cahill†

Distributed Systems Group‡,

Department of Computer Science,

Trinity College, Dublin 2, Ireland

Abstract

This document describes a log-keeping mechanism designed to support *Global Garbage Detection* on *Amadeus*. This log-keeping mechanism maintains, on a per *site* basis, a conservative approximation of the *actual root set* for that site. Exchanges of object references across site boundaries are logged on a per *cluster* basis to cope with the dynamic nature of the overall object graph. Clustering also determines the granularity of the information logged. Furthermore, this mechanism proceeds lazily, that is, it does not require either any additional messages to be exchanged (and thus does not cause any race condition), nor trigger any object fault which would not otherwise have occurred. This mechanism makes it possible to implement at a reasonable cost, a higher level *comprehensive*, although scalable, Global Garbage Detection algorithm.

1 Introduction

Log-keeping makes it possible to maintain a conservative approximation of the root set for each individual site locally, thereby allowing local Garbage Collection (GC) to proceed independently on each site. Log-keeping is performed by the mutator and essentially entails keeping track of objects to which references have crossed site boundaries. These objects are locally considered as "alleged roots" (also referred to as global roots).

Global Garbage Detection (GGD) entails eventually ridding the alleged root set of objects which are not actually referenced remotely. It is up to the local GC to proceed with the actual collection of garbage objects. This approach has often been employed in decentralized GC [Pla94b, Sch89] and can be traced back to Bishop [Bis77].

We distinguish two strategies for log-keeping: eager and lazy. The former attempts to update the log-keeping information as soon as possible, at the cost of additional background messages sent by the mutator. When an object reference crosses a site boundary,

*E-mail: Sylvain.Louboutin@dsg.cs.tcd.ie

†E-mail: Vinny.Cahill@dsg.cs.tcd.ie

‡URL: http://www.dsg.cs.tcd.ie/

an eager log-keeping mechanism attempts to update the log-keeping information maintained for the target object on the site where this object is located immediately. The latter attempts to postpone these updates as long as possible. Lazy log-keeping also avoids additional messages, without prejudice to the safety of the GGD.

The need for log-keeping is orthogonal to the choice of GGD strategy. Moreover log-keeping does not dictate the nature of the GGD algorithm *per se*. The choice of log-keeping strategy does not guarantee scalability, nor does it preclude comprehensiveness. However, the choice of strategy used by the GGD to determine which of the global roots are not actually referenced remotely, affects the way the log-keeping is performed, as the nature and amount of information which must be logged may be different.

The information maintained by the log-keeping mechanism constitutes a consistent, although not necessarily accurate, snapshot of the actual object graph, built incrementally as the overall object graph evolves. In Amadeus [CBSH93] this snapshot is maintained as a set of logs, one log per cluster, and contains enough information for the GGD to be comprehensive.

To guarantee the consistency of the logs, race conditions between messages containing references and background messages used for the log-keeping itself must be avoided. Otherwise live objects could erroneously be identified as garbage. This consistency constraint can therefore potentially be both costly (in terms of additional messages for instance) and complex when eager log-keeping is chosen. GGD approaches based on weighted reference counting [Bev87, WW87, Dic91] or reference listing [Pla94b] makes it possible to avoid this form of eager log-keeping but are not intrinsically comprehensive.

This document describes a lazy log-keeping facility aimed at supporting comprehensive GGD on *Amadeus* [CBSH93].

2 System Model

This section presents an abstract view of the underlying system. This is a conceptual description which attempts not to be too specific about actual implementation details although reflecting the Amadeus [CBSH93] model. It focuses on the essential characteristics which are the basis of the design of the log-keeping mechanism.

2.1 Root Sets

A *site* is a contiguous address space. Per-site GC is performed locally and independently of any other site. The root set for local GC consists of some *local roots* – the local root set – i.e., objects arbitrarily designated as roots, plus some *global roots* – the global root set – i.e., objects alleged to be referenced from other (possibly remote) sites including objects which are no longer referenced from other sites, and may have consequently become garbage, but have not yet been identified by the GGD.

The *actual root set* is made of objects, which although not necessarily reachable from a local root, are nevertheless alive; the union of the local root set and global root set is a superset of the actual root set as shown on Figure 1.

To make it possible a loose synchronisation between mutator processes and GGD, the actual root set cannot be efficiently known accurately at all times, a conservative

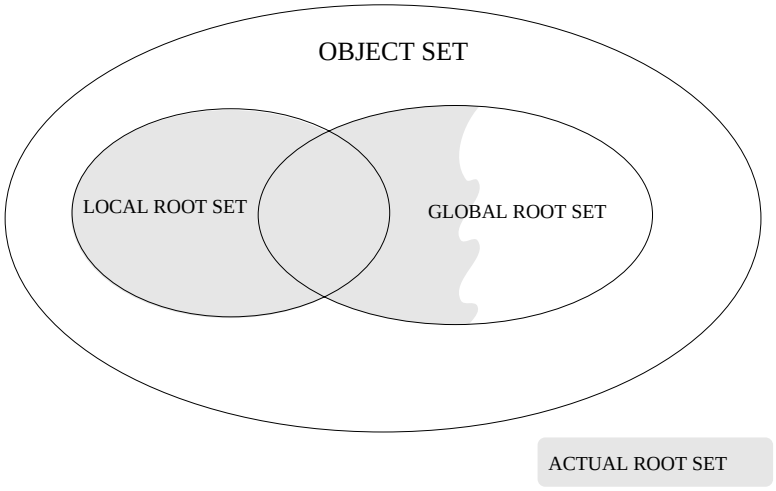

Figure 1: Object set, local root set, global root set and actual root set

approximation is used instead. This conservative approximation is the union of the local root set and the global root set, and is maintained jointly by mutators and the GGD algorithm. A mutator conservatively adds (write only) objects to the global root set as references to them cross site boundaries. It is then up to the GGD to purge the global root set in order to narrow it down to objects actually referenced from other sites. GGD is therefore decoupled from local garbage collection so that each site may actually implement its own garbage collection strategy.

The necessary counterpart of GGD is therefore a mechanism which makes it possible for the mutators to keep track of the exchange of references between sites. As the burden of this log-keeping task belongs to the mutators, overhead must be kept to a minimum. For instance this mechanism should not trigger object faults, nor require remote invocations which would not otherwise have occurred. This must remain true even when a reference is exchanged between third party remote objects. This is why lazy log-keeping is preferable to any eager log-keeping approach.

The invariant which this mechanism must maintain can be expressed as follows: *the union of the local and global root sets is a superset of the actual root set of the local object graph.*

2.2 Objects

An object is a contiguous portion of address space, whether on primary or secondary storage, potentially containing references to other objects.

An object can be designated as being global, i.e., potentially known and invoked from a remote location, and/or persistent, i.e., may potentially outlive the thread of control that created it, as well as the context in which it was created. Conversely, an object can be local and/or volatile.

A persistent object should not hold references to any volatile object, so as to prevent the eventual occurrence of dangling references. All objects transitively referenced by a persistent object should eventually be made persistent.

It would not incur much additional cost to conservatively consider any global object to be persistent, and leave it to the GGD to identify those global objects which are actually not reachable from any persistent (non-garbage) objects.

2.3 Clusters, Contexts and Containers

A context is a transient address space. A cluster is a collection of one or more objects. Clusters of objects are the unit of mapping into contexts. Each context contains a set of clusters which may vary dynamically as clusters are created, mapped into or un-mapped from it. A cluster is mapped into at most one context at a time.

A cluster of persistent objects is stored in some container. A container is a logically or physically contiguous area of secondary storage. There may be zero, one or more containers per node, i.e., physical host. Each container stores a subset of the clusters in the system.

The log-keeping mechanism considers that a cluster is *local* to a context if its OUT_TABLE (see Section 3.2) is accessible in that context. Clusters represented by proxies (see Section 2.5) are not considered to be local. It should be noted that a deactivated but not yet unmapped (see Section 2.4 and Section 2.5) cluster is still considered local.

At context termination, all co-located clusters must be deactivated before any one of them may actually be unmapped. This is necessary to ensure that their respective OUT_TABLEs can be updated appropriately before their contents are committed to secondary storage[1].

Only those clusters which are not mapped in some context, i.e., only *dormant* clusters, are considered to be local to a container.

2.4 References

Objects are the vertices and references the edges of the global object graph. Two forms of references are considered: *canonical references* and *language-specific references.* Canonical references are used in objects stored on secondary storage and are sent to other contexts. Language-specific references are used between objects co-located within the same context.

The process of converting a canonical reference into a language-specific reference is called *swizzling*; the reverse is called *unswizzling.*

The log-keeping mechanism relies on the fact that when an object is activated (sometime after its cluster has been mapped into a context), every reference that it contains is swizzled; conversely, when this object is eventually deactivated (before its context is unmapped from a context), every (swizzled) references that it contains is unswizzled.

Similarly, references are *marshalled* and *unmarshalled* when exchanged between contexts. The former involves unswizzling the reference to its canonical form, so that it can be sent across context boundaries, while the latter involves swizzling the reference back to its language-specific form.

The canonical and language specific forms of a reference may in fact be identical. Swizzling and unswizzling may then be null operations, but it is required that every

[1] This constraint could however be lifted if the local GC could participate in appropriately updating the logs, making it possible to preemptively un-map deactivated clusters.

reference crossing a site boundary be examined in turn[2]. However, references exchanged within a context are not required to be trapped by the log-keeping mechanism. This makes it possible to keep the overhead due to the log-keeping mechanism to a minimum.

The details of the implementations of canonical references, e.g., stubs and offsets to stubs [CBSH93], and of language-specific references, e.g., memory addresses as in C++, are not relevant to our log-keeping mechanism.

2.5 Proxies

When swizzling a reference to an absent object, a *proxy* for the object is created. If the absent object is already mapped into some other context, a *G-proxy* is created; such a proxy has the same interface as the remote object that it represents and acts as its surrogate. The G-proxy handles the marshalling and un-marshalling of the parameters to be sent to or received from the remote object that it represents[3].

If on the other hand the absent object is dormant, i.e., a persistent object stored in some container, a *P-proxy* for its whole cluster is created[4]. When such an absent object is eventually invoked by some thread of control the entire cluster containing this object is mapped into the current context, overlaying its P-proxy[5], and the invoked object is activated.

2.6 Cross Context Invocations

The system (and therefore the log-keeping mechanism) can only be aware of object invocations made across context boundaries since only these invocations require down-calls to the system, for instance to marshal and unmarshal parameters.

When an object reference is exchanged between a proxy and the server object that it represents, the system is able to identify both the server object and the object to which the reference is being exchanged. The system is however not able to identify the client object since interactions between co-located objects, in this case between the client object and the proxy of the server object, are performed independently from the system.

2.7 Mature Objects

When an object is created it is said to be immature. A global object is promoted to being mature when a reference to it is marshalled. A persistent object is promoted when a reference to it is unswizzled or when it is first deactivated. The allocation of a global name, or canonical reference, to an object is postponed until it is promoted. Note that promotion is irreversible as shown in Figure 2. When promoted, an object is assigned to a cluster (which may have to be created).

---

[2]Except for the special case of the references contained in clusters migrated between containers.

[3]The absent object might eventually be made to overlay its proxy if it is later mapped into the same context. The proxy is thus made to occupy the same amount of space as the object that it represents.

[4]We assume the existence of a mechanism which makes it possible to locate an object for which a reference is known anywhere in the system.

[5]Actually load balancing or security considerations may require that a cluster be mapped in some other context.

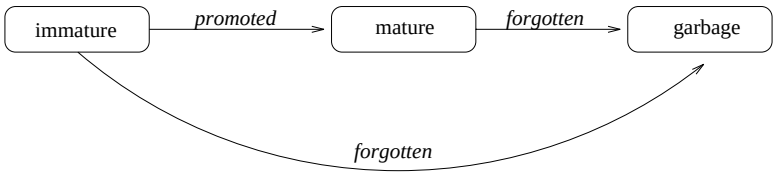

Figure 2: Maturity transitions

3 Log-Keeping

This section describes the design of the log-keeping mechanism including both the data structures and the actual algorithm used for log-keeping. The different situations which this mechanism may have to take into consideration are discussed giving informal indications about its correctness (see Section 3.4).

3.1 Notation and Definitions

This section introduces the notation used in the reminder of this document. This notation is only meant to facilitate the description of the mechanism. For instance no assumptions about the actual naming mechanism used by the underlying system should be drawn from the way in which an object is denoted.

X denotes a cluster (i.e., a name in upper case).

blue denotes an object (i.e., a colour name in lower case).

$X.blue$ denotes object *blue* belonging to cluster X.

$@A$ denotes a site which may be either a context or a container (i.e., an upper case letter preceded by an "@").

$X.blue@A$ denotes object *blue* (which belongs to cluster X) at site $@A$; $@A$ being either a container or a context. Note that any of *blue* or $X.blue$ or $X.blue@A$ can be used interchangeably to refer to the same object.

$\uparrow$ *blue* **or** $\uparrow X.blue$ **or** $\uparrow X.blue@A$ denotes a reference to the object *blue*.

$\{blue, \cdots, Y, \cdots\}_X$ denotes an entry in the OUT_TABLE of cluster X (see Section 3.2) associating object *blue* with cluster Y. The ellipsis $\cdots$ is used to show that the object may also be associated with other clusters by the same entry as there is at most one entry for a given object.

$\{blue, Y\}_{@A}$ denotes an entry in the IN_TABLE of context $@A$ (see Section 3.5) in this case, $@A$ can only refer to a context since there are no per container IN_TABLEs.

$<$ *cluster* $>$ **or** $<$ *object* $>$ is a generic token which denotes any cluster or any object in some set of clusters or objects, e.g., any cluster associated with some entry in a given table.

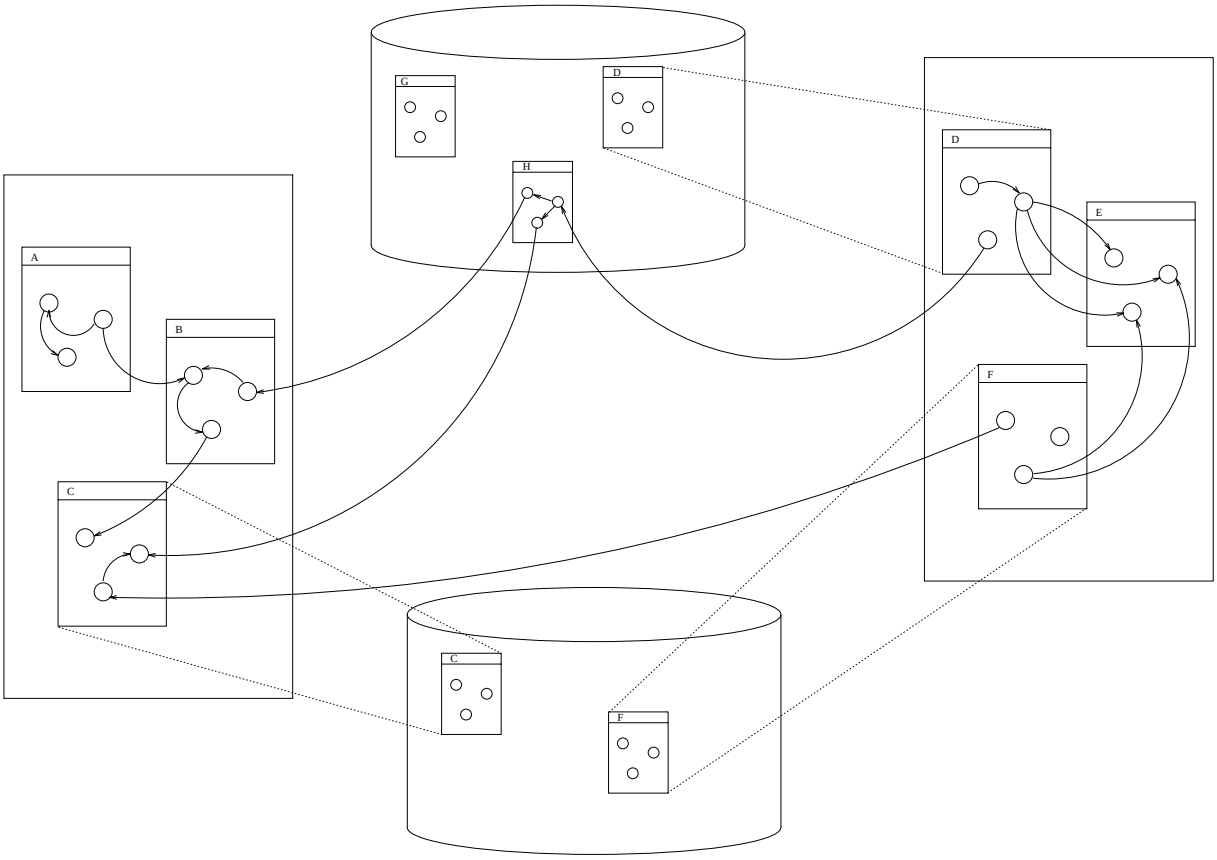

Figure 3: Dual representation of persistent clusters

3.2 Clusters As Log-Keeping Unit

The overall system-wide object graph potentially spans both primary and secondary storage[6] as shown in Figure 3. The information related to the exchange of references between sites should be maintained at the cluster level since the cluster constitutes the "largest common denominator" between both kinds of sites.

Keeping information about exchanges of references among objects at the per context or per container level would be difficult not only because contexts are transient entities, but also because of the very dynamic nature of the global object graph. Objects stored in the same container can be dynamically mapped into different contexts, and objects which were at one time co-located in the same context, can eventually be unmapped into different containers or migrated to different contexts. This information should therefore be more closely associated with individual objects.

Using clusters as the log-keeping unit makes it possible to reduce the overhead of managing the log itself by sharing its space overhead among several objects. It also

[6] A cluster which has already been unmapped once but is currently active in some context technically has two representations co-existing simultaneously (see Figure 3). However, only its primary storage, i.e., active, representation is significant. The local GC of the container where the secondary storage representation remains can safely ignore it.

potentially takes advantage of the locality of reference within clusters as objects are more likely to keep references to other objects belonging to the same cluster (which are not logged), hence minimizing the amount of information the log has to keep. Additionally, as will be explained in Section 3.4, this information can be maintained using a coarser granularity than if it were maintained on a per object basis, which should contribute to drastically reducing the amount of information to be maintained.

Each cluster maintains a table known as its OUT_TABLE. Logically, each entry in this table is indexed by the identifier (ID) of some object and contains a list of cluster IDs (and possibly a timestamp). Such an entry means that this object is "known" by each of these associated clusters. A cluster "knows" an object if it either contains a reference to this object, or has an entry in its OUT_TABLE indexed by the ID of this object. The index objects may or may not belong to the cluster where the table resides as will be explained later in Section 3.4.

3.3 Lazy Log-Keeping

Figure 4 shows an "ideal" situation; this situation is ideal because every cluster's OUT_TABLE shows the complete list of clusters containing references to each of its objects and nothing else.

In this Figure, $@A$ and $@B$ are contexts while $@C$ is a container, although, conceptually the distinction does not matter. For instance, object *pink* in cluster Z mapped in context $@B$, i.e., $Z.pink@B$, is known by $Y.red@B$ and $W.green@C$. Therefore, the OUT_TABLE of cluster Z contains the entry $\{pink, Y, W\}_Z$.

It should be noted that the global root set of context $@B$ consists of the objects *Pink* and *Yellow*. Object *White* which is only associated with a local cluster in the entry $\{White, Y\}_Z$ does not belong to the global root set.

Since the logs are updated as late as possible, i.e., lazily, this mechanism does not cause any object fault or any message transmission which would not have otherwise occurred (so as to not interfere with the execution and behaviour of the application), the situation depicted in Figure 4 is unlikely to occur.

However the log-keeping mechanism maintains the invariant that *every object in the global root set of some site, has an entry in at least one* OUT_TABLE *located at this site.*

Exchanges of references are only trapped when such references cross site boundaries as explained in Section 2.6. Only inter-context exchanges of references can be accounted for and logged directly, i.e., in the OUT_TABLE of the cluster to which the object referenced belongs, as soon as they occur. Moreover only exchanges which do not involve references to third party remote objects, i.e., references to objects located in the site of neither the sender nor the recipient, can be logged directly.

Intra-context but cross-cluster exchanges of references can only be logged later when the clusters involved in these exchanges are eventually deactivated and the references they contain unswizzled (see Section 3.5). Furthermore, as previously noted, exchanges of third party references, cannot be logged directly either, but are nevertheless logged indirectly via the creation of *partial back pointer paths* as explained in Section 3.4. This will nevertheless not affect the correctness of the local garbage collectors, i.e., its safety property.

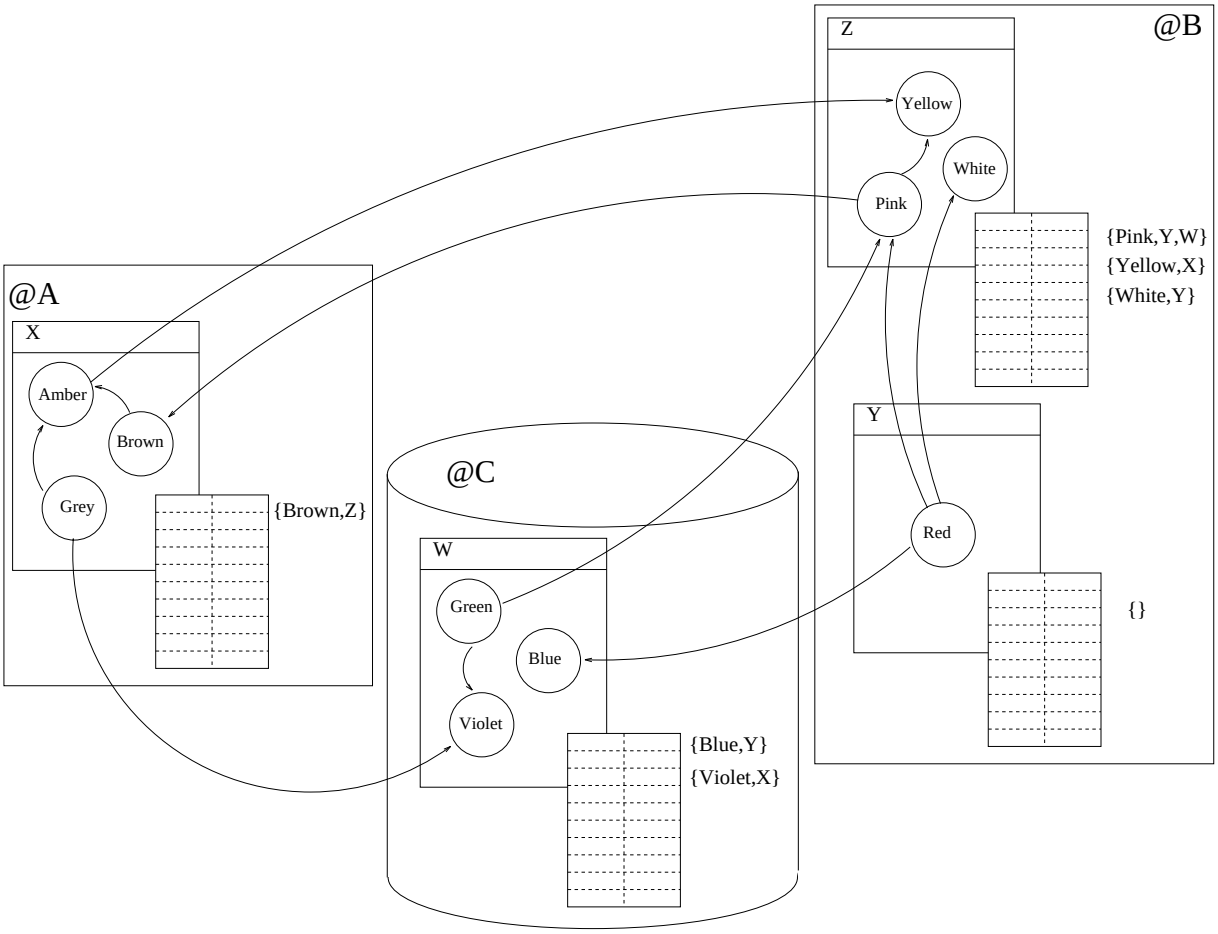

Figure 4: Ideal situation

3.4 Partial Back Pointer Path

The aforementioned invariant can be maintained without necessarily keeping a strictly accurate per cluster log as in Figure 4. It is not necessary for such a log to maintain an exact list of all the clusters holding a reference to some object.

The only requirement is that any dormant object to which a reference is held by some object in a different cluster, has any entry in its cluster's OUT_TABLE, and that any active object, to which reference is held by some object which is not co-located in the same context, has any entry in its cluster's OUT_TABLE.

However, enough information should be kept to make it possible for the GGD to decide whether or not a particular entry is obsolete and can safely be discarded (see Section 4). To do so, it should be possible to eventually gather, for every object, the complete list of clusters which "know" this object. However the list of clusters which know a particular object is not necessarily kept entirely in the OUT_TABLE of the cluster to which the object belongs.

The idea of this log-keeping mechanism is that, rather than trying to eagerly maintain a situation as shown in Figure 4 whereby each per-cluster OUT_TABLE contains the complete

list of clusters which know a given object, a trail of partial back pointers is lazily left behind, along the paths its references have followed during successive exchanges between sites.

In the example shown in Figure 8, an entry in the OUT_TABLE of cluster Z, associates an object *red* with a list of clusters, in this case $\{red, Y\}_Z$. This means that cluster Y contains either a reference $\uparrow red$, or that its OUT_TABLE contains an entry $\{red, \cdots\}_Y$. In other words, an OUT_TABLE entry associates an object with a list of clusters which know this object, where "knowing" may only mean containing an entry indexed by this object in their own OUT_TABLE.

An entry in some OUT_TABLE can be described as a *partial back pointer*. It is a back pointer because it leads to whatever cluster or clusters know the given object. It is a partial back pointer because it does not point to each individual object which holds such a reference but to their clusters.

The complete list of clusters which know a given object can be gathered by tracing these *partial back pointer paths*. This list does not necessarily include some remote active clusters which may contain a reference to the object. However, since such clusters would be co-located into the same context as some other cluster already logged in some OUT_TABLE along these partial back pointer path, the invariant is not broken and an appropriate entry will eventually be logged when their context terminates as explained in Section 3.5.

Locality of reference within clusters should therefore contribute to reducing the number of such paths for a given object and hence the complexity of the resulting tree or graph made of the partial back pointer paths rooted at this object.

3.5 Growth of a Tree of Partial Back Pointer Paths

This sections describes the algorithm used by the log-keeping mechanism, that is, how a graph of partial back pointer paths grows from the initial entry logged when a newly created object is promoted, up to entries pointing to all the clusters which actually contain a reference to this object. It shows how a root entry can always be logged when a newly created object is promoted and how a reference in transit through a site does not fail to leave behind a partial back pointer path.

Missing Link Cluster

The ID of some cluster, chosen at random among the clusters mapped with a client object is piggy-backed with the parameter list of any remote invocation to a server object in case this invocation may return a reference. This cluster[7] will be referred to as the *missing link cluster*.

Per-context IN_TABLE

The per-context IN_TABLE is a structure logically maintained at the context level which is not persistent, i.e., it persists only for as long as the context. This table logically associates each proxy in the context with the local cluster having first imported (either

[7]For instance the cluster associated with the ID of the server in the per context IN_TABLE.

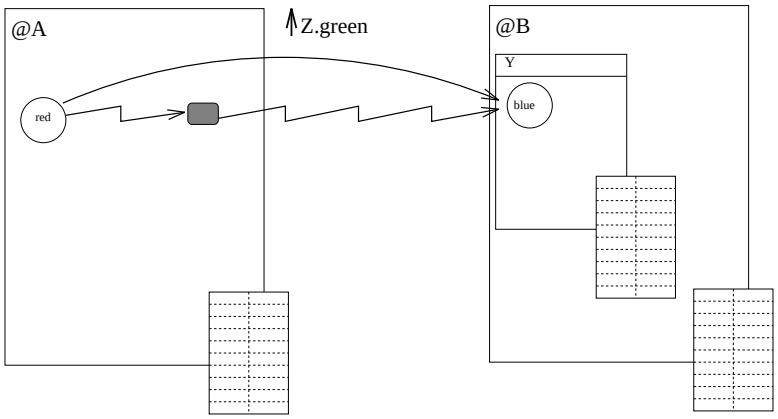

Figure 5: Reference sent as a parameter of a remote invocation

unmarshalled or swizzled) a reference to this remote object into this context, as explained below.

Exporting a Reference

A reference to an object can be exported from some context $@A$ to another context $@B$ in either of the following two ways:

1. As shown in Figure 5, where a client $red@A$ sends a reference $\uparrow Z.green$ to the server $Y.blue@B$[8].

   (a) If $Z.green$ is promoted as the result of its reference being marshalled for the first time (see Section 2.7), $green$ has just been allocated to its cluster $Z@A$. The identity of the remote server $Y.blue@B$ being known (see Section 2.6), the entry $\{green, Y\}_Z$ must be logged[9].

   (b) The entry $\{green, \cdots, Y, \cdots\}_Z$ must also be logged if $Z.green$ was already mature and mapped into context $@A$.

   (c) If $Z.green$ is not mapped in context $@A$, an entry indexed by $green$ necessarily exists in the per context IN_TABLE. For instance if this entry is $\{green, W\}_{@A}$, the entry $\{green, \cdots, Y, \cdots\}_W$ must be logged.

   In any case a link to Y is added to the partial back pointer path (if required).

2. Figure 6 shows a server $X.red@A$ returning a reference $\uparrow Z.green$ to client $blue@B$ as the result of some invocation. In this case, the identity of the remote client is not known (see Section 2.6). The ID of the missing link cluster, for instance $V@B$, must be used instead.

[8] The proxy of the server is represented as a small grey square.

[9] An entry is only logged into some table if it is not already present.

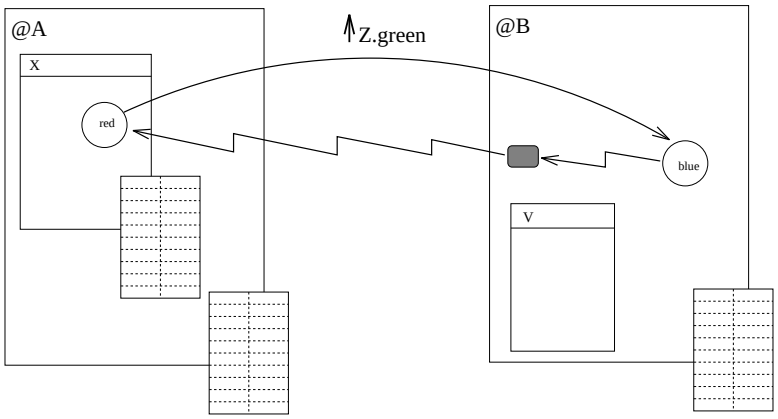

Figure 6: Reference returned as a result of a remote invocation

(a) If *Z.green* is promoted as the result of its reference being marshalled for the first time (see Section 2.7), *green* has just been allocated to its cluster $Z@A$. The root entry $\{green, V\}_Z$ must be logged.

(b) Similarly, if *Z.green* was already mature and mapped into context $@A$, the entry $\{green, \cdots, V, \cdots\}_Z$ must logged.

(c) If *Z.green* is not mapped in context $@A$, an entry indexed by *green* necessarily exists in the per context IN_TABLE. For instance if this entry is $\{green, W\}_{@A}$, the entry $\{green, \cdots, V, \cdots\}_W$ must be logged.

In any case a link to the missing link cluster V is added to the partial back pointer path (if required).

Note: When a global object (as seen in Section 3.5) is promoted, that is, when its reference crosses a site boundary for the first time (marshalled), an entry for this object can always be logged immediately into the OUT_TABLE of its cluster. This entry constitutes the first link, or *root entry*, in a partial back pointer path rooted at this object. From this point, and until the GGD eventually removes this entry, if ever (see Section 4), and as long as this entry is associated to some non-local clusters, local per site GCs will consider this object as a global root.

Furthermore, when a reference to either a local mature object, or a reference to some remote object is exported, the OUT_TABLE of respectively the cluster to which this object belongs, or the cluster known to have initially imported the reference into the context can be updated with an entry pointing to the next link in the partial back point path.

Importing a Reference

A reference to an object can be imported into some context $@B$ from another context $@A$ in either of the two ways previously described. The object to which the reference is imported is already mature.

1. As shown in Figure 5, where the server $Y.blue@B$ receives a reference $\uparrow Z.green$ from the client $red@A$.

   If $Z.green$ is not mapped in $@B$ and there is no entry indexed by $green$ in the IN_TABLE of context $@B$, the entry $\{green, Y\}_{@B}$ must be logged.

   Cluster Y is now known to have initially imported $\uparrow Z.green$ into $@B$.

2. As shown in Figure 6, where the client $blue@B$ receives a reference $\uparrow Z.green$ from a server $X.red@A$ as the result of some invocation. The identity of the client, i.e., of the object who first imports the reference into context $@B$, is not known (see Section 2.6). The ID of the missing link cluster, for instance $V@B$, must be used instead.

   If $Z.green$ is not mapped in $@B$, and there is no entry indexed by $green$ in the IN_TABLE of context $@B$, the entry $\{green, V\}_{@B}$ must be logged.

   Cluster V is now known to have initially imported $\uparrow Z.green$ into $@B$.

Note: When a remote reference is imported for the first time, i.e., as soon as a proxy is created for the global object to which the reference is imported, an entry is logged in the IN_TABLE. The cluster associated with this entry is identical to the cluster associated with the corresponding entry in the OUT_TABLE of the object to which the reference is being imported.

Activating a Cluster

When some cluster X is activated into context $@A$, for each reference to some non-local object which is swizzled but not yet indexed in the IN_TABLE of $@A$, e.g., $\uparrow Z.green$, the entry $\{green, X\}_{@A}$ must be logged.

Additionally, if activating a cluster results in overlapping a proxy, i.e., if an object which was indexed in the IN_TABLE is mapped, the corresponding entry in the IN_TABLE must be removed.

Note: Activating a cluster is equivalent to importing all the references that it contains.

De-activating a Cluster

When some cluster X is deactivated from context $@A$, for each remote reference being unswizzled, the cluster indexed by the referenced object in the IN_TABLE must be updated.

For instance, when $\uparrow Z.green$ is unswizzled, if the entry indexed by $green$ in the IN_TABLE is $\{green, W\}_{@A}$, the entry $\{green, \cdots, X, \cdots\}_W$ must be logged.

If unswizzling $\uparrow Z.green$ actually results in promoting object $Z.green$, which would then necessarily be local, the root entry $\{green, X\}_Z$ must be logged.

Note: When a reference to a persistent object is first unswizzled, i.e., when a persistent object is promoted because its reference is held by a cluster being deactivated, the root entry for this object can be logged directly in the OUT_TABLE of its cluster[10]. Similarly

[10] When a persistent object is promoted by crossing a site boundary, that is, when this object is itself deactivated, no OUT_TABLE needs to be updated.

an entry associating the cluster being deactivated with a local mature object to which a reference is unswizzled can also be logged directly. Entries associating some remote object with the cluster being deactivated are however logged into the OUT_TABLE of the cluster known to have first imported it.

Even though inter-cluster but intra-context exchanges of references cannot be trapped by the log-keeping mechanism, they are eventually logged when clusters are deactivated. Deactivating a cluster is equivalent to exporting all the references that it contains.

References in transit

Whenever a reference to some object transits through a context, it is first logged into the IN_TABLE of this context, associated with the cluster known to have first imported this reference. This cluster can either be the cluster having actually imported it, or a missing link cluster arbitrarily chosen. What matters is that both parties involved in exchanging a reference agree upon which cluster is known to have imported it. In this way, the previous link in the partial back pointer path points to this cluster. And the OUT_TABLE of this cluster may eventually become the next link in the path, should this reference be re-exported to another site. The path of partial back pointers therefore remains unbroken. This is illustrated in the following example:

Figure 7 shows an object *pink@B* which invokes object *Y.blue@A*. *Y.blue@A* returns the reference $\uparrow$*Y.green@A* as the result of this invocation. The identity of the cluster of the client object (in this case *pink*) is not known[11], the missing link cluster *V@B* is used instead so that the entries $\{green, V\}_Y$ and $\{green, V\}_{@B}$ can be logged.

Object *pink@B*[12] later re-exports this reference by invoking object *X.amber@C* and passes the reference $\uparrow$*Y.green@A* as one parameter of this invocation. The cluster of the remote server, i.e., *X*, being known, the entries $\{green, X\}_V$ and $\{green, X\}_{@C}$ are logged.

3.6 Inaccuracies in the Logs

Figure 8 represents a set of clusters and the contents of their respective OUT_TABLEs as could be observed after a few exchanges of references have taken place within the system. Sites boundaries are not represented.

Unlike the "ideal" situation represented in Figure 4, their OUT_TABLEs are not necessarily accurate, although they contain enough information for any local garbage collector to be safe, no matter how these clusters eventually end up being distributed across different sites.

These tables may contain three kind of inaccuracies:

- Obsolete entries such as $\{blue, W\}_Z$ and $\{maroon, Y\}_Z$. The former is obsolete because the reference to object *blue* previously held by some object of cluster *W* does not exist anymore; the latter because object *maroon* does not exist anymore. Both entries would eventually be removed by the GGD.
- Incomplete entries, i.e., which give an incomplete list of the clusters actually containing a reference to some object. This list can nevertheless be reconstructed using

[11] Furthermore, *pink* may not be mature.

[12] It could be any other object mapped in context *@B*.

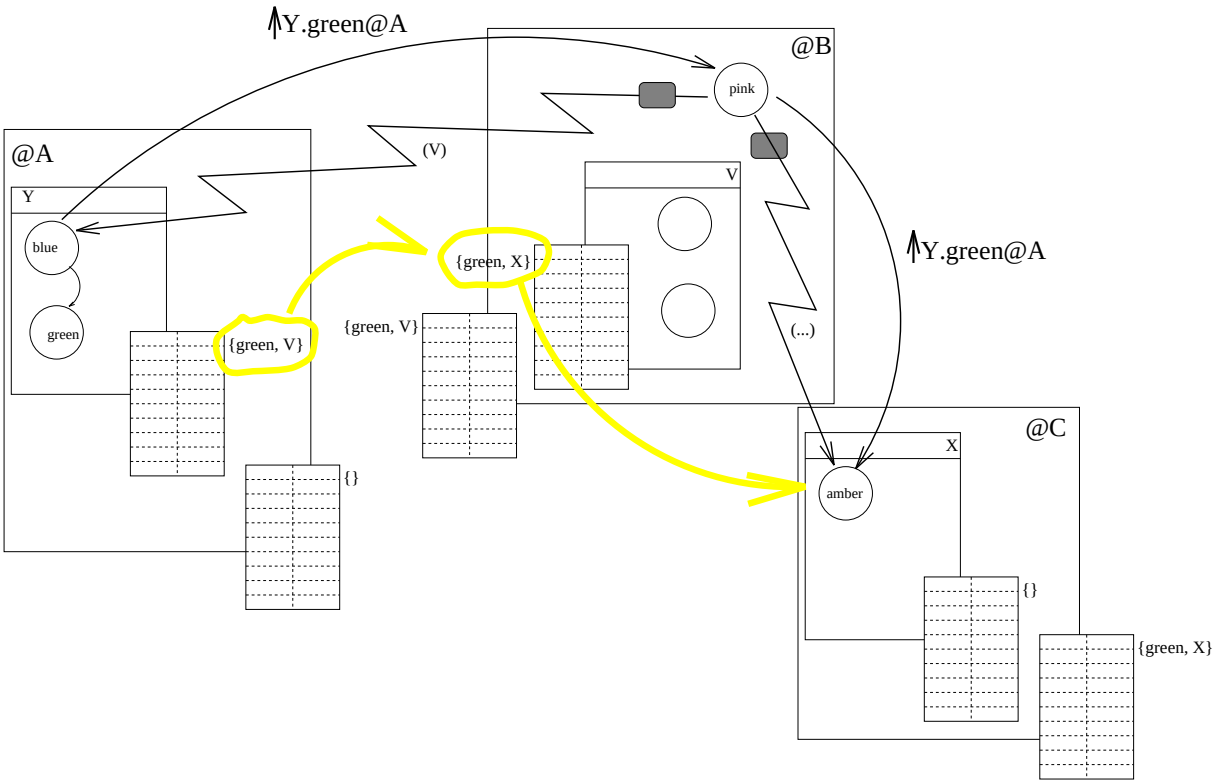

Figure 7: A reference in transit

available information held by other OUT_TABLEs which form the partial back pointer path.

For instance, the entry like $\{red, W, X\}_Z$ would be more accurate than the entry $\{red, Y\}_Z$ that the OUT_TABLE for Z actually contains.

However the partial back pointer path can be traced as follow: entry $\{red, Y\}_Z$, means that object *red* is known to cluster Y. However, no object in cluster Y holds any reference to object *red*, but there is an entry $\{red, X\}_Y$. In turn object $X.grey$ actually holds a reference $\uparrow Z.red$ and there is an entry $\{red, W\}_X$ in the same cluster. In cluster W, object $W.pink$ is the only place where a reference $\uparrow Z.red$ can be found.

- Entries belonging to some partial back pointer path such as $\{red, Y\}_Z$. Although no objects in cluster Y *per se* holds any reference to object $Z.red$, this entry is not obsolete since the OUT_TABLE of cluster Y contains an entry indexed by object *red*.

  Y may either have been used as a missing link cluster, or an object in Y may have once held the reference $\uparrow Z.red$ and forgotten it after having forwarded it to $X.grey$, or else, Y may have been the first cluster to have imported $\uparrow Z.red$ in a context where it was mapped although it was another cluster which re-exported it to $X.grey$. It is not possible at this stage to know which of these possibilities applies, which shows, as stated in Section 3.5, that any cluster is equally valid to serve as the missing link cluster.

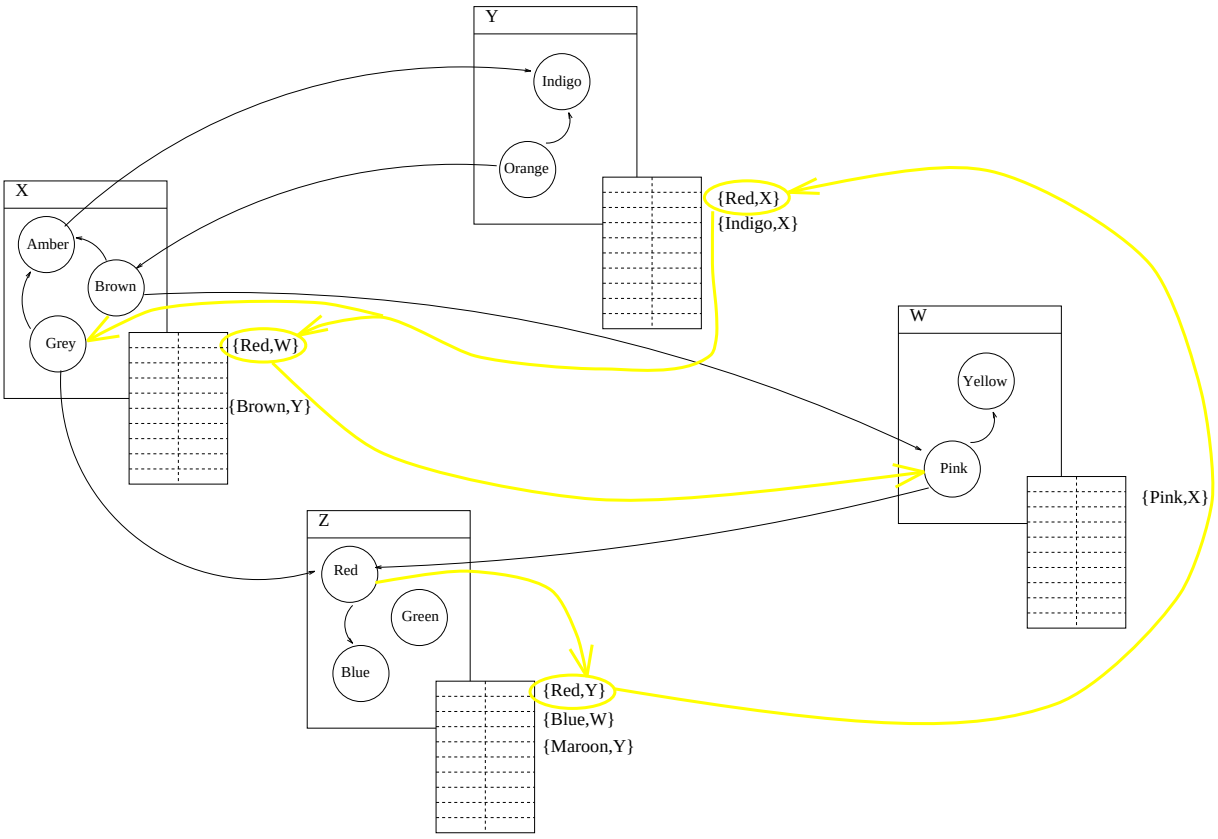

Figure 8: Partial back pointer path

Similarly, the entry $\{red, X\}_Y$ is not obsolete either even though this cluster does not contain any object *red*.

In other words, although these logs may contain inaccuracies, they are nevertheless complete, since the invariant stated in Section 3.3 is not broken.

4 Pruning the Partial Back Pointer Path Trees

The relationship between the lazy log-keeping mechanism described in this document and a GGD utility, can be described by stating the conditions under which the entries in the different tables maintained by this mechanism, i.e., per-cluster OUT_TABLEs and per-context IN_TABLEs, can safely be discarded. This does not impose the use of any particular comprehensive GGD policy, any graph-tracing based approach can be used. See Louboutin and Cahill [LC95] for further details about an adaptation of an algorithm for GGD inspired by that of Schelvis [Sch89, SB88] using this lazy per-cluster log-keeping mechanism.

Garbage Objects

An object becomes garbage when:

- it is not reachable from any local root,
- there is no entry in the log of any local cluster associating the object's ID with the ID of some non-local cluster.

Garbage Clusters

A cluster becomes garbage when:

- it contains only garbage objects,
- and there is no entry in its OUT_TABLE associating some object's ID with the ID of some non-local cluster.

Garbage OUT_TABLE entry

An entry in some per cluster OUT_TABLE indexed by some object *red* can be collected when *red* is no longer known by any of the associated clusters in this entry. An object is known by a cluster if this cluster contains either a reference to this object, or a corresponding entry in its OUT_TABLE. An entry can also be collected whenever the object indexing it has been collected by the GC.

Garbage IN_TABLE entry

An entry in some per context IN_TABLE indexed by some object *blue* can be collected by the local per-context GC[13] if no local object contains any reference to object *blue*.

In other words, the entry in the IN_TABLE indexed by object *blue* can be collected by the local GC as soon as the proxy for *blue* is itself collected by the local GC. The IN_TABLE does not have to actually be implemented as an independent data structure. An additional field in the header of an object proxy (see Section 2.5) containing the ID of the associated cluster would be sufficient. In that way, both proxy and corresponding IN_TABLE entry would therefore be collected simultaneously by the local GC if it becomes locally unreachable, or, if the actual object *blue* eventually overlays its own proxy, the corresponding entry would also be removed appropriately.

5 Related Work

Ferreira and Shapiro [FS94b, FS94a] describe a system based on a Distributed Shared Memory (DSM) model rather than the Remote Procedure Call (RPC)/object-swapping model adopted by Amadeus; it features fine-grain, i.e., smaller than a page, objects, clustered into fixed-size, i.e., made of a fixed number of contiguous pages, and disjoint, "segments." These segments are themselves logically grouped into "bunches." Bunches can be replicated and shared via the underlying weakly consistent DSM system. Objects are identified by their address within a 64 bit system-wide address space encompassing both primary and secondary storage[14]. GC is performed at two levels; a per bunch

[13] Unlike entries in per cluster OUT_TABLE which can only be collected by the GGD mechanism.
[14] Object addressing is a combination of OID and SSP approaches [Pla94a].

comprehensive GC and a "scion cleaner" which is not comprehensive. A heuristic is used to group bunches at one site so that a comprehensive GC can tackle cycles locally.

This approach relates to ours in the sense that both systems use some form of object clustering, and that log-keeping is done on a per cluster basis. However, when an inter-bunch reference is created[15] and the bunch of the target object is not local, a "scion-message" must be sent to the target; this creates an additional overhead (additional message) for mutator processes. Race conditions that these additional log-keeping messages would create are avoided by piggy-packing these messages on the messages used by the underlying consistency protocol. Our system avoids such additional log-keeping messages altogether. It should also be noted that unlike the Ferreira and Shapiro approach, the logs and their contents are not part of the object addressing scheme.

6 Conclusion

Laziness

Our log-keeping mechanism does not attempt to update remote third party logs even in the case of exchanges of third party references, it does not require additional "control" messages, and hence avoids race conditions common to eager log-keeping approaches.

This mechanism is also said to be lazy because it postpones the update of the log as late as possible (e.g., until context termination for inter-cluster, intra-context exchanges of references), and does not trigger object-faults which would have not otherwise occurred.

Log-keeping for comprehensive GGD

The first time a reference to some target object crosses a site boundary, an appropriate entry can always be logged in the target's cluster. This initial partial back-pointer identifies the target as a global root. Our mechanism ensures that when this reference subsequently crosses another site boundary, that there is a co-located cluster, already belonging to the partial back-pointer path, where an appropriate entry can be logged.

Thus it can be seen that the complete list of the clusters which hold a reference to a given object can be gathered by transitively tracing these partial back-pointer paths. It therefore maintains enough information for a comprehensive GGD which would proceed by tracing the graphs of partial back-pointer paths rather than the actual object graph.

Locality of reference within clusters should contribute to reducing the number of such paths for a given object and hence the complexity of the resulting graph of partial back-pointer paths rooted at this object.

Robustness

This mechanism is robust. Logs are updated when a reference is marshalled, unmarshalled, unswizzled or swizzled, and before such action is actually performed. If the actual exchange of reference or the mapping or unmapping of some cluster which triggered these log-keeping operations fails, it would result in some unnecessary log entries which would

[15] Such creation is trapped via a "write-barrier" unlike our RPC based system which uses (un)swizzling operations.

have to be later collected whose only consequence would be potential additional detection latency.

Overhead

The overhead of this mechanism is mostly space overhead, i.e., size of the OUT_TABLEs and possibly clusters containing only garbage objects but which cannot be collected (see Section 4). Since log-keeping operations are performed by trapping (un)swizzling operations it should only add a negligible computing overhead.

However, as a consequence of this approach, a GGD using this log-keeping mechanism generates potentially more inter-site messages than a GGD which traces the actual object graph or uses an eager log-keeping approach. These messages are due to the inaccuracies in the logs explained in Section 3.6. Additional messages would be required when tracing entries belonging to some partial back pointer path, and logged in the OUT_TABLE of some cluster where no objects actually hold any reference to the object indexing this entry. We contend that shifting the overhead from the log-keeping mechanism, i.e., from the mutator processes, to the GGD is in itself beneficial to overall system performance even if it does not decrease the number of messages exchanged globally.

Worst case scenario

If k is the total number of objects in the system, and n the total number of clusters in the system, a worst case scenario may generate an OUT_TABLE of size $k(n-1)$, i.e., a monstrous OUT_TABLE with k entries, each of them associated with $n-1$ cluster IDs. If every OUT_TABLEs in the system grow to this proportion, a potential space overhead of $kn(n-1)sizeof(ID)$ would have to be considered.

However, it is expected that such a scenario is highly unlikely to occur, and that typical cases would be more reasonable due to object clustering (and dynamic re-clustering) and locality of reference within clusters, and among clusters. An effective GGD algorithm would also contribute to continuously keeping the growth of the logs under control. The log-keeping mechanism could also be optimized so as not to log entries corresponding to exchanges of references between clusters which remain co-located after being (un-)mapped.

Residual side effects

The mechanism requires that some information (of the size of a cluster ID) be piggy-backed on the parameter lists of all inter-context object invocations. This overhead is deemed acceptable.

The mechanism as described in this document also prevents the preemptive unmapping of clusters, since all co-located clusters must be deactivated before any one of them may actually be unmapped, as explained in Section 2.3. This constraint can be lifted by involving the local GC in the log-keeping operations. When a cluster is deactivated, the log-keeping mechanism described in this document is able to update accordingly the OUT_TABLEs of all other co-located clusters. If a cluster is pre-emptively un-mapped, i.e., while some other co-located clusters are still active, the log-keeping mechanism as described, cannot update this cluster own OUT_TABLE appropriately, However, a local

graph-tracing GC could update this OUT_TABLE, making it possible to un-map this cluster before the context termination, but not before the completion of the next local GC iteration.

References

[Bev87] D.I. Bevan. Distributed Garbage Collection using Reference Counting. *PARLE (Parallel Architectures and Languages Europe)*, pages 176–187, June 1987. LNCS 259.

[Bis77] Peter B. Bishop. *Computer systems with a very large address space and garbage collection.* PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, USA, May 1977. MIT/LCS/TR-178.

[CBSH93] Vinny Cahill, Seán Baker, Gradimir Starovic, and Chris Horn. Generic runtime support for distributed persistent programming. In Andreas Paepcke, editor, *OOPSLA (Object-Oriented Programming Systems, Languages and Applications) '93 Conference Proceedings*, volume 28, pages 144–161, Washington D.C., USA, 1993. ACM, New York. Also technical report TCD-CS-93-37, Dept. of Computer Science, Trinity College Dublin. ftp://ftp.dsg.cs.tcd.ie/pub/doc/TCD-CS-93-37.ps.gz.

[CTS+92] Vinny Cahill, Paul Taylor, Gradimir Starovic, Brendan Tangney, and Darragh O'Grady. Supporting the Amadeus platform on UNIX. technical report TCD-CS-92-25, Dept. of Computer Science, Trinity College, Dublin, Ireland, July 1992. ftp://ftp.dsg.cs.tcd.ie/pub/doc/TCD-CS-92-25.ps.gz.

[Dic91] Peter William Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures.* PhD thesis, Darwin College, Cambridge University, September 1991.

[FS94a] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California, USA, November 1994.

[FS94b] Paulo Ferreira and Marc Shapiro. Garbage collection of persistent objects in distributed shared memory. In *Proceedings of International Workshop on Persistent Object Systems*, Tarascon, France, September 1994.

[LC95] Sylvain R.Y. Louboutin and Vinny Cahill. On Comprehensive Global Garbage Detection. In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, Alpes d'Huez, France, April 1995. Also technical report TCD-CS-95-11, Dept. of Computer Science, Trinity College Dublin. ftp://ftp.dsg.cs.tcd.ie/pub/doc/TCD-CS-95-11.ps.gz.

[Pla94a] David Plainfossé. Comparaisons entre les OIDs et les chaînes de PSS. Note Technique SOR-131, INRIA–SOR, Paris, France, May 1994.

[Pla94b] David Plainfossé. *Distributed Garbage Collection and Referencing Management in the Soul Object Support System.* PhD thesis, Université Pierre & Marie Curie – Paris VI, Paris, France, June 1994.

[SB88] Marcel Schelvis and Eddy Bledoeg. The implementation of a distributed smalltalk. In *ECOOP'88*, pages 212–232, 1988.

[Sch89] M. Schelvis. Incremental Distribution of Timestamp Packets: A New Approach To Distributed Garbage Collection. In *Proceedings OOPSLA'89*, pages 37–48, New Orleans, October 1989. ACM.

[WW87] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In J.W. de Bakker, A.J. Nijmaan, and P.C. Treleaven, editors, *PARLE (Parallel Architectures and Languages Europe)*, pages 432–443, Eindhoven, The Netherlands, June 1987.