Comprehensive Distributed Garbage Collection by Tracking Causal Dependencies of Relevant Mutator Events.

Sylvain R.Y. Louboutin* Vinny Cahill†
Distributed Systems Group‡
Department of Computer Science§
Trinity College, Dublin 2, Ireland

Abstract

Comprehensive distributed garbage collection in object-oriented distributed systems has mostly been addressed via distributed versions of graph-tracing algorithms, a legacy of centralised garbage collection techniques. Two features jeopardise the scalability of these approaches: the bottleneck associated with having to reach a global consensus *before any resource can actually be reclaimed, and the overhead of* eager log-keeping. *This paper describes an alternative approach to comprehensive distributed garbage collection that entails computing the vector-time characterising the causal history of some relevant events of the mutator processes computations. Knowing the causal histories of these events makes it possible to identify garbage objects that are not identifiable by means of per-site garbage collection alone. Computing the vector-times necessary to identify garbage is possible without the unbounded space overheads usually associated with dynamically reconstructing vector-times of arbitrary events of distributed computations. Our approach integrates a* lazy log-keeping *mechanism and therefore tackles both of the aforementioned stumbling blocks of distributed garbage collection.*

Keywords: comprehensive global garbage detection, causal dependencies, mutator events, lazy log-keeping, scalability, robustness.

1 Introduction

Automated garbage collection in object-oriented systems is often advertised as a means of obviating the burden and hazard of explicit resource management, i.e., as a lesser evil or expensive convenience, which could nevertheless be avoided altogether under appropriate circumstances. This might be true in the context of a centralised system where each thread of control independently manages its own private object graph, i.e., where the visibility, accessibility, and lifespan of objects does not extend beyond the scope of the thread that created them. However, we contend that automated distributed — or global — garbage collection is necessary and unavoidable in a distributed system featuring persistent and shared objects [13, 14]. In this case, objects may outlive the thread(s) that created them and may be shared by threads that cannot have an up-to-date, consistent, and comprehensive view of the overall object graph spanning a number of disjoint address spaces scattered among autonomous processors.

Traditional global garbage collection algorithms based on the iterative graph tracing approach are comprehensive, i.e., inherently able to detect all garbage, including distributed cycles of garbage, but make it necessary to account for all live objects in the system before garbage objects can be detected and their resources reclaimed. Garbage is detected in at most, but no sooner than, one iteration of the algorithm. In particular, distributed global garbage collection requires that every site in the system eventually participates in every iteration. This drawback, inherent to all graph tracing based global garbage collection algorithms and referred to as the *consensus bottleneck,* as well as the overhead of *eager log-keeping* (see §2.3), jeopardises the scalability of these approaches.

This paper describes a new approach to global garbage collection that entails reconstructing the vector-times that characterize the causal history of some relevant events of the mutator processes computation. These events are those that result in modifications to the inter-site paths in the global object graph. It will be shown that knowing the causal history of these events makes it possible to identify

*E-mail: Sylvain.Louboutin@dsg.cs.tcd.ie
†E-mail: Vinny.Cahill@dsg.cs.tcd.ie
‡URL: http://www.dsg.cs.tcd.ie/
§fax: (+353-1) 6772204

0-8186-7813-5/97 $10.00

garbage objects that are not identifiable by means of per-site garbage collection alone. This algorithm is intrinsically comprehensive, although it is not based on a graph-tracing algorithm, and hence its scalability is not hampered by the aforementioned consensus bottleneck. Its message complexity depends on the number of garbage objects, rather than the number of live objects, as is the case for graph tracing based approaches. Moreover, its underlying *lazy log-keeping mechanism,* does not require additional control messages, even in the case of third party exchanges of references, guaranteeing the robustness of the algorithm. Loss of messages cannot cause erroneous identification of live objects as being garbage, i.e., message loss does not compromise the safety of the algorithm. Instead, loss of messages can only cause residual garbage to remain undetected.

We proceed as follow: §2 lays the background for this paper by discussing how global garbage collection algorithms address the issues of partitioned address spaces and distribution by decoupling local and distributed aspects of garbage collection. The rôle of log-keeping is introduced leading to a short survey of traditional comprehensive global garbage collection algorithms outlining the main flaws of these graph–tracing approaches. §3 introducing our alternative approach, explaining how garbage can be identified from the causal histories of some of the events of the mutator processes computation, and introduces the way in which our algorithm computes the vector–times characterising these causal histories, using an underlying lazy log-keeping mechanism. §4 details how our approach compares with a related algorithm, proposed by Schelvis [16], that relies on an eager log-keeping mechanism, which compromises its scalability and robustness. §5 summarises the contribution of this paper.

2 Background

An *object* is a contiguous portion of address space and a container of references to other objects. Objects are the vertices and references the edges of a directed graph. Some objects known as *roots,* constitute the "entry points" for the application processes that access and modify the object graph. An object that is not *reachable* from any of these roots, i.e., when there is no longer a directed path along the edges of the object graph, from any root to the object, is *garbage.* Garbage objects must be collected in order to reclaim their resources. In a distributed system, this object graph is partitioned over a number of independent address spaces — or *sites* — themselves distributed over a set of autonomous physical hosts. Edges of the object graph may or may not cross site boundaries. In a loosely-coupled distributed system, each site independently manages its own resources. Thus, distribution demands some degree of decoupling between local garbage collection and what we later define as *global garbage detection.*

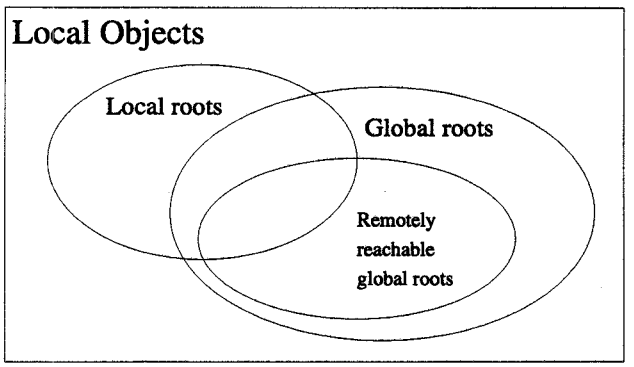

Figure 1: Root set used for local garbage collection.

2.1 Partitioned Address Spaces & Root Sets

An approach inspired by Bishop [3] to scavenging a partitioned address space can be generalised to describe all *decentralised* global garbage collection algorithms, i.e., algorithms that allow autonomous local garbage collectors to proceed concurrently. In this approach, per-site garbage collection is performed locally and independently of any other site. The root set used for local garbage collection 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., local objects that are alleged to be referenced from other (possibly remote) sites.

Once a reference to some object crosses its site boundary, it is not possible to determine locally whether or not it is still reachable from any root. The local garbage collector must therefore conservatively consider it to be a global root. Until proven otherwise, all local objects reachable from this root are considered to be live, as are any objects reachable from local roots. Figure 1 shows the different root sets.

The union of the local root set and global root set is a superset of the *actual root set.* The actual root set of a given site contains only the roots from which all the live objects, and only the live objects of that site, can be reached. The actual root set of a site is the union of the local root set and the set of remotely reachable global roots. Only global garbage detection can determine whether or not a given global root is still remotely referenced.

Taking full advantage of the decentralised nature of a distributed system entails maintaining a conservative approximation of the actual root set for each individual site locally, and progressively ridding it of objects that are no longer remotely referenced. We refer to the process of maintaining this conservative approximation to the actual root set as log-keeping. The process of removing objects that are no longer remotely referenced from the global root set is known as *Global Garbage Detection* (GGD).

2.2 The Global Root Graph

The global roots of each individual site taken together form a distributed graph known as the *global root graph*. A vertex of the object graph that has had at least one incoming edge that crosses its site boundary is a global root and becomes a vertex of the global root graph. Every outgoing path from a global root, which crosses its site boundary (via some vertices of the object graph collocated on the same site), becomes a single edge in the global root graph. A root of the global root graph is a root of the object graph that has such an outgoing path from it that crosses its site boundary.

Figure 2 depicts an object graph in its upper portion and the corresponding global root graph in its lower portion. As every edge of the global root graph crosses some site boundary, these boundaries become implicit and hence need not be represented.

As the distributed object graph evolves, a global root on some site may no longer be remotely reachable. This means that there is no longer a path from any root to this object along the edges of the global root graph. Such an object can be discarded from the set of global roots of its site, narrowing the root set of that site down to a better approximation of its actual root set. GGD can therefore be described as performing garbage collection of the global root graph. A global root discarded by GGD may however remain reachable from some local root, i.e., it is up to local garbage collection to detect and collect actual garbage.

2.3 Log-keeping

Log-keeping is essentially the task that application processes — or mutators — must perform in addition to their own computation, in support of GGD. Log-keeping serves two purposes:

1. It keeps track of objects to which references have crossed their site boundary, i.e., log-keeping identifies global roots.
2. Log-keeping also contributes to maintaining additional information depending on the choice of GGD strategy. For instance the contents of the logs may consist of a "weight" as in "weighted reference counting" schemes [2, 19, 6] or the identity of the recipient of a reference as in "reference listing" schemes [15].

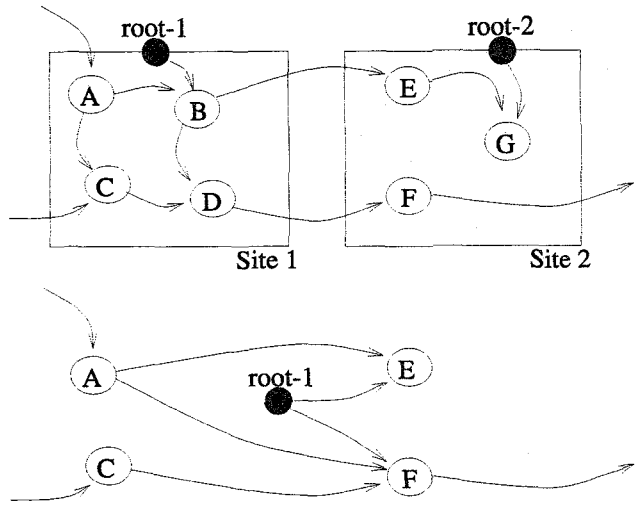

Figure 2: An object graph and its global root graph.

These logs, which may be either centralised or distributed, together constitute a consistent, although not necessarily complete, snapshot of the object graph. The logs are *consistent* if they reflect a consistent cut of the distributed mutator computation [1]. This means that if the occurrence of some event of the mutator computation is recorded in these logs, then all events that causally precede it have also been recorded. Such a snapshot may be built incrementally by the mutator processes as the overall object graph evolves and remains consistent provided that there are no race conditions between control messages necessary for log-keeping. The logs may form an incomplete snapshot because log–keeping alone is not necessarily sufficient to identify garbage.

We distinguish two strategies for log-keeping: eager and lazy. The former may require additional control messages to be sent by the mutator processes, while the latter does not.

When an object reference crosses a site boundary, an eager log-keeping mechanism attempts to immediately update the log maintained for the target object. If this log is collocated with the corresponding object, or maintained by some centralised service, this may involve additional control messages when exchanging references to some third-party remote object. This may in turn lead to race conditions between these control

messages and control messages used to signify the destruction of edges in the global root graph. This race condition could jeopardise the consistency of the logs and ultimately can compromise the safety of GGD. Ensuring the consistency of these logs under such conditions, i.e., when eager log-keeping is used, can be rather costly and compromise the robustness of GGD as explained in §3.

Lazy log–keeping prevents this race condition from occurring by postponing the delivery of control messages used to signify the creation of new edges in the global root graph until they become necessary, as explained in §3.4.

2.4 Pitfalls of Distributed Graph Tracing

Comprehensive GGD has mostly been addressed via distributed versions of graph-tracing [7] based algorithms. Two phases can usually be identified in these GGD algorithms [10, 9, 4]. The first phase involves detecting live objects, while the second phase makes sure that the first one is complete.

These algorithms are better described as *live object detection* rather than genuine *garbage collection* algorithms because garbage is characterised as being everything that is not alive. Live objects may be either detected directly by colouring the object graph *in situ* [10, 9], or indirectly from the logs maintained by the log-keeping mechanism. The contents of these logs may be used to reconstruct consistent representations of the overall object graph that can be traced locally, either by a conceptually centralised service [11], or by each site that is participating in GGD [4].

Once it has been determined that all live objects have been accounted for, or that enough information has been collected to reconstruct a consistent representation of the object graph, garbage objects can then be safely identified and their resources reclaimed. This termination detection is often performed as a distinct phase [10, 9, 4], although using a conceptually centralised log-keeping service obviates an explicit termination detection phase [11]. Moreover Tel has also shown how these two phases can be superimposed [18, pp.193–226].

To increase concurrency, multiple GGD iterations may overlap and proceed concurrently, e.g., an approach using time-stamps [9] makes it possible to interleave any (bounded) number of iterations. However, all sites in the system are still required to eventually participate in completing any given GGD "iteration" and must reach some kind of consensus.

In summary, two features of graph–tracing based GGD approaches jeopardise their scalability: on one hand the bottleneck associated with having to reach global consensus before any resource can actually be reclaimed, i.e., the "consensus bottleneck." On the other hand, these GGD algorithms must either rely on eager log-keeping in order to benefit from the flexibility and increased parallelism of autonomous per-site garbage collector or must use exhaustive *in situ* global graph tracing.

3 An Alternative to Graph Tracing

Comprehensive approaches are therefore generally believed to be necessarily unscalable [15]. As a consequence, comprehensiveness has often been traded-off for scalability under the assumptions that distributed cycles are relatively rare, and that only graph–tracing algorithms can be intrinsically comprehensive [6, 2, 19]. Instead, in the absence of empirical evidence to the contrary, we contend that distributed cycles of garbage are as likely to occur as local cycles, and that intrinsically comprehensive GGD algorithms can, in fact, be scalable as well.

Our alternative to graph-tracing consists in analyzing the mutator processes *computation*, focusing on those aspects directly relevant to GGD, and subsequently referred to as "log-keeping events." Reconstructing the causal history of a given log-keeping event, using known techniques, e.g., [8], might require that each site or object maintains a rather large local history of events. We show however that it is possible to do so with reasonable space overhead because log–keeping events are not events of an arbitrary computation (see §3.3).

3.1 Log-keeping Events

The execution of the mutator computation can be represented as a space–time diagram where each global root appears as a process exchanging log-keeping control messages with other global roots. An event number j for a process i is denoted $e_{i,j}$.

There are two kinds of log-keeping control messages, indicating respectively the creation or the destruction of an edge in the global root graph. Whenever a new edge is created, or an existing edge is removed, from some global root to another one, a log-keeping control message is *conceptually* sent from the former to the latter. A *log-keeping event* corresponds to receiving a log-keeping control message. There are two kinds of log-keeping events that correspond respectively to the creation or destruction of an edge to the corresponding global root. These events are referred to respectively as *edge creation events* and *edge destruction events*.

Figure 3 represents the evolution of a global root graph used throughout the reminder of this paper to

illustrate our algorithm. Figure 4 shows the corresponding space–time diagram. In this particular example, each object is assumed to be located on its own site, and as a result, the actual object graph is identical to its corresponding global root graph. This graph evolves according to the following scenario: a root object 1 creates an object 2 (event $e_{2,1}$). Object 2 creates object 3 (event $e_{3,1}$) and then object 4 (event $e_{4,1}$). Object 2 subsequently sends object 4 a message containing a reference to object 3, creating an edge from object 4 to object 3 (event $e_{3,2}$). Similarly, object 2 sends object 3 a message containing a reference to object 4 hence creating an edge from object 3 to object 4 (event $e_{4,2}$). Object 2 then sends a reference denoting itself to object 4 creating an edge from object 4 to object 2 (event $e_{2,2}$). The last modification to the global root graph by the mutator process represented in Figure 3 is the destruction of the edge from the root to object 2 (event $e_{2,3}$). Subsequent modifications to this global root graph are due to GGD as explained later on.

The log-keeping mechanism contributes to maintaining a direct dependency vector (DDV) similar to what Fowler & Zwaenepoel [8] describe. Log-keeping events are numbered sequentially, i.e., time-stamped, at each process with a monotonically increasing counter. The DDV of an event is derived from the DDV of its local predecessor by including its own time-stamp, and the time-stamp of its direct remote predecessor.

The value 0 in a dependency vector indicates that no log-keeping message has yet been received from the corresponding global root. On the other hand, $\overline{n}$ represents the time-stamp of the direct remote predecessor of an edge destruction event, i.e., it indicates that the last log-keeping control message received from the corresponding global root was an edge destruction log-keeping control message. Edge creation events and edge destruction events are represented as black dots or white triangles respectively in Figure 4.

3.2 Characterization of Garbage

Unlike the DDV that the log-keeping mechanism contributes to maintaining, the full vector–time of an event takes the transitive closure of causal dependencies into account, and fully characterises the causal history of the corresponding event,

The DDV of event $e_{3,1}$ in Figure 4, denoted $DDV(e_{3,1})$, is $(0,1,1,0)$. It indicates that the events directly (and causally) preceding $e_{3,1}$ are $e_{2,1}$ and $e_{3,1}$ itself. On the other hand, the full vector–time characterising the causal history of event $e_{3,1}$, denoted $V(e_{3,1})$, is $(1,1,1,0)$ (see Figure 5).

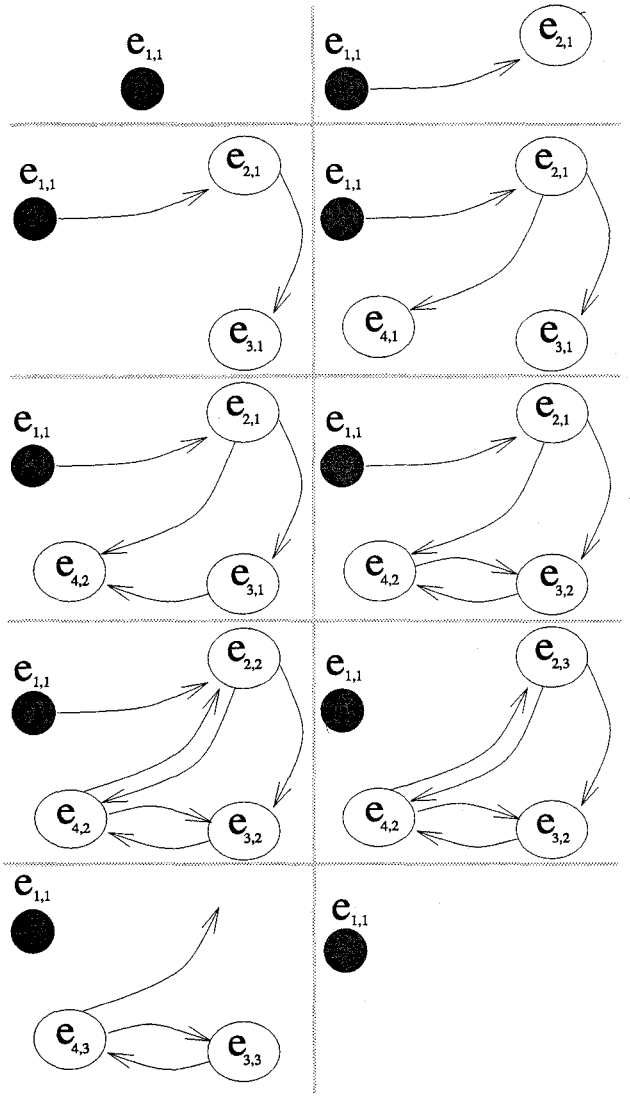

Figure 3: Evolution of a global root graph.

Schwarz & Mattern [17] demonstrate that if two events a and b of a distributed computation are causally related, i.e., $a \prec b$, then $V(a) < V(b)$. This partial order relation between vector–times u and v of dimension m is defined as follow (from [17]):

1. $u \leq v$ iff $u[k] \leq v[k]$ for $k = 1, \cdots, m$
2. $u < v$ iff $u \leq v$ and $u \neq v$

Our approach is based on the idea that it is possible to construct a vector–time that characterises the events responsible for the creation of all the paths to a global root that actually exist when some event occurs, and no other events, and, in turn, makes it possible to identify garbage in the global root graph. For instance, in

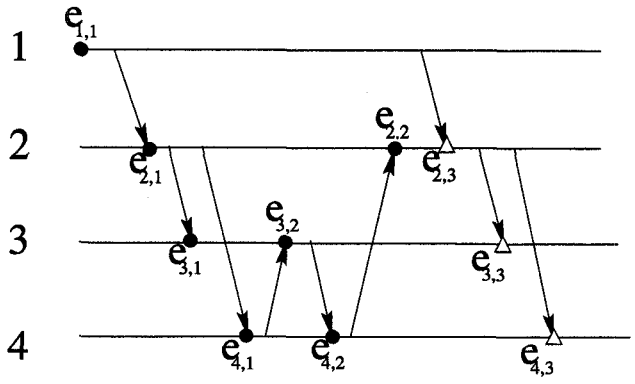

Figure 4: Space–time diagram corresponding to the scenario of Figure 3.

Event	DDV	V
$e_{1,1}$	(1,0,0,0)	(1,0,0,0)
$e_{2,1}$	(1,1,0,0)	(1,1,0,0)
$e_{2,2}$	(1,2,0,2)	(1,2,2,2)
$e_{2,3}$	($\overline{1}$,3,0,2)	($\overline{1}$,3,2,2)
$e_{3,1}$	(0,1,1,0)	(1,1,1,0)
$e_{3,2}$	(0,1,2,1)	(1,1,2,1)
$e_{3,3}$	(0,$\overline{3}$,3,1)	(0,$\overline{3}$,3,1)
$e_{4,1}$	(0,1,0,1)	(1,1,0,1)
$e_{4,2}$	(0,1,2,2)	(1,1,2,2)
$e_{4,3}$	(0,$\overline{3}$,2,3)	(0,$\overline{3}$,2,3)

Figure 5: Dependency vectors of the log–keeping events of Figure 4.

Figure 4, the global root 2 is reachable from global root 4 in the global root graph when event $e_{2,2}$ occurs, because $e_{4,2} \prec e_{2,2}$ in the execution graph. This can be shown by comparing their respective vector–times: $V(e_{4,2}) < V(e_{2,2})$, i.e., $(1,1,2,2) < (1,2,2,2)$. On the other hand when event $e_{2,3}$ occurs, it is possible to determine directly from the vector–time of this event — which is $(\overline{1},3,2,2)$, see Figure 5 — that global root 2 is no longer reachable. The vector–time of event $e_{3,2}$ however does not yet reflect the fact that object 3 has also ceased to be reachable. (When comparing vectors–times, the time-stamp of the direct remote predecessor of an edge destruction event, indicating that there is no longer an edge in the global root graph via the corresponding global root, is treated as if no edge creation event had ever been sent from this global root, i.e., as 0.) Detecting the absence of a live path to this object requires additional edge–destruction control messages to be sent by the GGD algorithm as part of the *finalisation* of those garbage objects already detected.

However, these vector–times cannot be used to characterise the existence or the absence of a path between two global roots in the global root graph if both kinds of log-keeping events are undifferentiated. Whenever a path is created in the global root graph, there is a corresponding path of causally related edge creation events in the execution graph. When an existing path in the global root graph is broken as the result of the destruction of some edges in the object graph, there is a corresponding edge destruction event in the execution graph. Edge–creation events that reflect the creation of paths to the object that receives an edge–destruction control message via the object that sent it, must therefore not be taken into account when computing the full vector–time of the edge–destruction event. The full vector–time must therefore be reconstructed dynamically and cannot be incrementally updated by piggy-packing some kind of vector–time to each message exchanged between mutator processes.

3.3 Algorithm

An algorithm that illustrates quite intuitively how vector–times characterising the causal history of an event can be dynamically reconstructed from partial information gathered locally by each process is the algorithm proposed by Fowler & Zwaenepoel [8]. This algorithm, proceeds by recursively gathering the DDVs of the causal predecessors of the event, and therefore assumes the existence of some backward pointer from each event to its direct remote predecessor [17, 8]. It also requires an unbounded space overhead to cope with keeping the DDVs of every past event as far back as might be needed.

Our algorithm similarly gathers partial information logged locally, but takes advantage of fact that the model introduced in §3.1 does not describe an arbitrary distributed computation, but only the creation and destruction of edges in the global root graph. This provides us with *forward* pointers to all remote causal successors of an event by following the actual edges of the global root graph. Knowing these forward pointers, one may incrementally reconstruct the causal histories of log-keeping events, or rather the dependency vectors characterizing them, by repeatedly circulating increasingly accurate approximations of these dependency vectors, along the paths of the global root graph, until the complete transitive closure, i.e., the full vector–time, has been determined.

The initial approximation of the vector–time of some event $e_{i,j}$ is similar to the DDV mentioned in §3.1. This vector, noted $DV(e_{i,j})$, records the latest

index of each of the predecessors of event $e_{i,j}$, i.e., characterising the event itself, all of its local predecessors, and all of the direct remote predecessors of these events. This vector can be transitively merged with the dependency vectors of its causal predecessors until the full vector–time is obtained. This is the case, when the only difference between the vector–time received from the direct *remote* predecessor of $e_{i,j}$ and $DV(e_{i,j})$ itself, lies in $e_{i,j}$'s own index j. The resulting algorithm shown in Figure 6 can be summarized as follow:

1. Each vertex i in the global root graph maintains a log DV_i. Each entry in this log is a vector containing the best locally held approximation for the dependency vector of the latest known log–keeping event of the corresponding global root.
2. Whenever a global root receives a dependency vector from another global root adjacent to it, the received vector is merged with the corresponding log entry. A new approximation of the vector–time for the latest log–keeping event can then be computed from the updated contents of the log. If this newly computed dependency vector is the actual full vector–time and indicates that the global root is no longer reachable from an actual root, the global root is removed from the global root graph.
3. This new dependency vector is in turn sent along the out–bound edges of the global root graph to each adjacent global root.

The pseudo–code of our algorithm is shown in Figure 6. For the purposes of illustration, an algorithm similar to the algorithm of Fowler & Zwaenepoel [8] is used (procedure ComputeV). This procedure is however applied to a strictly local structure, i.e., recursive invocations do not involve any remote invocation. The test of the predicate $\neg\Delta(\alpha)$ in the recursive procedure ComputeV stops the recursion if the time–stamp of the direct remote predecessor of an edge–destruction event is encountered. This predicate therefore evaluates to true for either a null time–stamp, i.e., $\Delta(0)$ is true, or the time–stamp of the direct remote predecessor of an edge–destruction event, i.e., $\Delta(\overline{n})$ is true. The procedure Receive corresponds to the code executed when a dependency vector v is received by a global root i from an adjacent global root m. The vector m may be the contents of an edge–destruction control message which is the case when $\Delta(v[m])$ is true, this vector may therefore contain the time–stamps of delayed edge–creation events as explained in §3.4.

```
Receive (i: process, v: vector, m: process)
        if v[m] > DV_i[i][m] ∧ Δ(v[m]) then
           DV_i[i][i] + +
           for all k do
               DV_i[i][k] = max(DV_i[i][k], v[k])
           end for
        else
           DV_i[i][m] = max(DV_i[i][m], v[m])
           for all k do
               DV_i[m][k] = max(DV_i[m][k], v[k])
           end for
        end if
        for all k ≠ i do
           V[k] = 0
        end for
        V[i] = DV_i[i][i]
        ComputeV(i)
        if V ≠ DV_i[i] then
           DV_i[i] = V
           for all k ∈ Acquaintances_i do
               send (k, V) -- to remote successor k
           end for
        else if ¬(∃k : ¬Δ(V[k]) ∧ root(V[k])) then
           for all k ∈ Acquaintances_i do
               V = DV_i[k]
               V[i] = overline(DV_i[i][i])
               send (k, V) -- to remote successor k
           end for
           remove -- garbage detected
        end if
end Receive

ComputeV (p: process)
        for all q ≠ p do
            α = DV_i[p][q]
            if α > V[q] ∧ ¬Δ(α) then
               V[q] = α
               ComputeV(q)
            end if
        end for
end ComputeV
```

Figure 6: Global garbage detection algorithm.

3.4 Lazy Log-keeping

The model described in §3.1 assumes some kind of log–keeping mechanism that is somehow able to immediately react to the creation of an edge in the global root graph, even though this edge may have been created

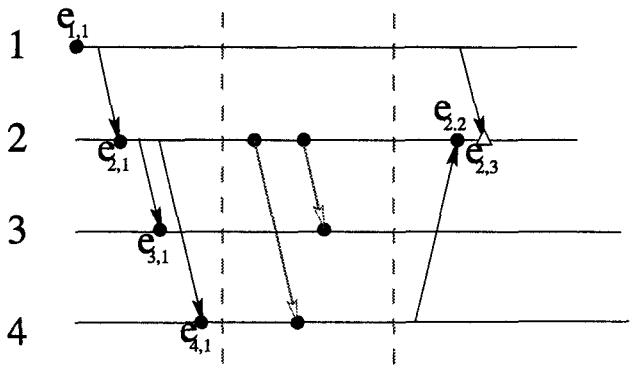

Figure 7: Lazy log-keeping.

by an exchange of messages involving some third party (remote) object, and in such a way that there cannot be any race conditions between log-keeping control messages.

Each vertex i of the global root graph maintains a two-dimensional log of dependency vectors, noted DV_i. Conceptually, as explained in §2.3, whenever an incoming edge (to this vertex) in the global root graph is either created or destroyed, the vector $DV_i[i]$ (one of the entries of the log maintained by i) is updated, with the latest event index of the remote vertex from which the edge is pointing. $DV_i[i]$ would therefore be equivalent to the dependency vector described by Fowler & Zwaenepoel [8].

Such an eager log-keeping mechanism would involve exchanging additional control messages, e.g., when a message containing a reference denoting k is sent from i to j, some control message must somehow be sent to k as well. However, race conditions between log–keeping control messages must be avoided. Although it is possible to implement such an eager log-keeping mechanism (see §4), additional log-keeping control messages compromise the scalability and robustness of the whole GGD. Instead, we adopt a lazy log-keeping approach that avoids the problem altogether [12]. Our lazy log-keeping mechanism updates the logs as follows:

- Whenever an object i sends a copy of its own reference object j, the log DV_i is updated as follows:
  - $DV_i[i][j] = DV_i[i][j] + 1$
  - $DV_i[i][i] = DV_i[i][i] + 1$
- Whenever an object i sends to an object j a copy of a reference denoting an object k, the log DV_i (and not DV_k) is updated as follows:
  - $DV_i[k][j] = DV_i[k][j] + 1$

  On receiving the reference, the recipient, i.e., object j, updates its own log as follows:
  - $DV_j[j][j] = DV_j[j][j] + 1$

In other words, whenever some object i sends a reference across a site boundary, only the logs of the objects involved in the exchange are updated, and not the log of the target (third party) object. Our lazy log-keeping mechanism makes it possible to adequately update the logs without requiring any additional control messages to be sent, hence avoiding race conditions altogether. This is illustrated in Figure 7 where the messages actually carrying the copy of the reference are represented as light grey arrows.

There can be more entries in the log DV_i than the number of inbound edges towards i in the global root graph, i.e., entries logged on behalf of remote third party objects. Such an entry is eventually sent to this third party object as part of the edge–destruction control message, when the edge from i to k is destroyed, i.e., multiple edge–creation control messages can be bundled with an edge–destruction control message in one atomic delivery.

An edge–destruction control message is sent by the local garbage collector when the last reference to a remote object is destroyed locally, i.e., when the proxy for that remote object is collected. An edge–destruction control message sent from an object i to a remote object k essentially contains the contents of the vector $DV_i[k]$ maintained by object i on behalf of object k, where $DV_i[k][i]$ is replaced by $\overline{DV_i[i][i]}$.

3.5 Granularity and Clustering

§3.1 describes our model of log-keeping events using a finer granularity than is actually necessary. Each global root is modeled as a process in the space-time diagram. Actually, one need not distinguish between individual *remote* objects which can be lumped together as one "process." Two distinct objects will always be assigned distinct event indexes, and since the propagation of dependency vectors is actually done along the paths of the global root graph, there is no ambiguity as to the recipients of these vectors. From the point of view of an external observer, collocated objects on some remote "process" are indistinguishable from one another, i.e., two distinct events may either be associated to two distinct objects, or to the same object at two distinct times. Our lazy log-keeping mechanism [12] uses object *clusters* [5] as the granularity of the information it maintains.

DV_2	DV_3	DV_4
$(0,0,0,0)$		
$(\mathbf{1,2,0,1})$	...	...
$(0,0,0,1)$		
$(0,0,1,0)$		
$(\overline{1},0,0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
$(\overline{\mathbf{1}},\mathbf{3},\mathbf{0},\mathbf{1})$	$(0,0,0,0)$	$(0,0,0,0)$
$(0,0,0,1)$	$(\mathbf{0,1,1,0})$	$(0,0,0,0)$
$(0,0,1,0)$	$(0,0,0,0)$	$(\mathbf{0,1,0,1})$
$\gg(\overline{1},3,1,1)$		
$(\overline{1},0,0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
$(\overline{\mathbf{1}},\mathbf{3},\mathbf{1},\mathbf{1})$	$(\overline{1},3,1,1)$	$(\overline{1},3,1,1)$
$(0,0,0,1)$	$(\mathbf{0,1,1,0})$	$(0,0,0,0)$
$(0,0,1,0)$	$(0,0,0,0)$	$(\mathbf{0,1,0,1})$
	$\gg(\overline{1},3,1,1)$	$\gg(\overline{1},3,1,1)$
$(\overline{1},0,0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
$(\overline{\mathbf{1}},\mathbf{3},\mathbf{1},\mathbf{1})$	$(\overline{1},3,1,1)$	$(\overline{1},\overline{3},1,1)$
$(0,0,0,1)$	$(\overline{\mathbf{1}},\mathbf{3},\mathbf{1},\mathbf{1})$	$(\overline{1},3,1,1)$
$(\overline{1},3,1,1)$	$(\overline{1},3,1,1)$	$(\overline{\mathbf{1}},\mathbf{3},\mathbf{1},\mathbf{1})$
$\gg(\overline{1},\overline{3},1,1)$	$\gg(\overline{1},3,\overline{1},1)$	$\gg(\overline{1},3,1,\overline{1})$
remove	**remove**	**remove**

Figure 8: Evolution of the logs of the objects shown in Figure 3 according to the algorithm listed in Figure 6.

3.6 Example

Figure 8 depicts the evolution of the logs of each of the global roots already illustrated in Figure 3 and Figure 7. GGD is only triggered when the edge between 1 (the actual root) and 2 is removed.

Figure 8 is made of three columns, one for each global root, except 1 which never changes because it is an actual root. Reading from the top of each column, each box represents the state of the log of the corresponding global root, one box for every modification of the log, starting just one stage before initiating GGD, i.e., just before DV_2 merges the vector $(\overline{1},0,0,0)$ sent from 1. Therefore, the first row shows the state of the different logs as updated by the lazy log-keeping mechanism described in §3.4. The sign $\gg$ is used to indicate the dependency vector sent to successors.

4 Related Work

Schelvis [16] previously proposed a comprehensive alternative to graph–tracing GGD, that proceeds by analysing the mutator processes computation graph, although the author describes his own algorithm as an incremental graph–tracing algorithm. It is therefore not surprising that this algorithm has often later been either overlooked or mis-identified in the literature [4, 15].

Schelvis's algorithm entails determining for each global root, the potential existence or absence of open paths to that root, by constructing *time–stamp packets* from its local logs. A time–stamp packet is a form of dependency vector characterizing the causal history of some log-keeping events. An eager log-keeping mechanism keeps track of the creation and destruction of edges between any two global roots, as the graph evolves. Whenever an edge in the global root graph is either created or destroyed, packets are repeatedly propagated down the paths that are potentially affected by this modification, until each global root along these paths has determined whether or not it remains potentially reachable from an actual root.

Unlike the approach described in this paper, time–stamp packets characterise the potential existence or absence of paths to a global root via only one of the global roots adjacent to it. Schelvis' approach actually consists in a depth first tracing of the mutator processes computation graph. As a result, it suffers from a worse message complexity than our own algorithm when processing recursive data structures such as double linked lists, or any cyclic structure containing subcycles. For instance, identifying the k elements of a double linked list that becomes disconnected from the object graph as garbage, requires $O(k^3)$ messages using Schelvis' algorithm, while our approach requires only $O(k)$ messages.

5 Conclusion

This paper describes a novel approach to GGD that entails computing the vector–time characterizing the causal history of some relevant events of the mutator computation, i.e., log-keeping events. This algorithm evaluates whether or not an object is garbage directly from knowledge of the mutator computation instead of examining its by–product, i.e., the object graph.

Algorithms similar to the aforementioned Fowler & Zwaenepoel's algorithm [8] require that each process stores the dependency vectors of all its previous events. Our algorithm avoids the space overhead that could be expected from a method dynamically reconstructing causal dependencies, because it is not necessary for the purposes of GGD to compute the vector–time of every log–keeping event of the mutator computation. In other words, this algorithm may not be able to compute $V(e_{i,j})$ for all j, but eventually comes up with $V(e_{i,k})$ where $k > j$ which is sufficient to determine whether the corresponding vertex i has become garbage (because garbage is a stable property).

The major drawbacks of our approach are its unbounded detection latency, and a space overhead greater than that of a graph tracing approach. However, unlike other algorithms that dynamically reconstruct vector–times, this algorithm does not require any form of "back pointer" to (causally) preceding

events, and does not make it necessary to maintain dependency vectors for all events of the distributed computation, which would lead to unbounded space overhead. It takes advantage of the fact that the vector–times of interest characterise the causal history of non-arbitrary events of the mutator processes. These events are those related to the creation or destruction of paths in the object graph.

Contrary to popular belief, comprehensive GGD is therefore not necessarily based on an object graph–tracing approach and intrinsically comprehensive alternatives to traditional graph tracing based GGD are possible. Combined with a lazy-log keeping mechanism, this makes it possible to tackle the two problems that jeopardise the scalability of GGD, namely the overhead of eager log-keeping and what we described as the consensus bottleneck. Additionally, messages exchanged for GGD are idempotent, which contributes to the robustness of our approach as neither loss nor duplication of messages compromise the safety of the algorithm.

References

[1] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Tech. Report UBLCS-93-1, University of Bologna (UBLCS), (Italy), Jan. 93.

[2] D.I. Bevan. Distributed Garbage Collection using Reference Counting. In *PARLE'87*, pp. 176–187, Jun. 87. LNCS, No.258/259.

[3] P. B. Bishop. *Computer systems with a very large address space and garbage collection.* PhD thesis, MIT, (USA), May 77. Tech. Report MIT/LCS/TR-178.

[4] A. Björnerstedt. *Secondary Storage Garbage Collection for Decentralized Object-Based Systems.* PhD thesis, The Royal Institute of Technology and Stockholm University, (Sweden), Jun. 90. Tech. Report 77.

[5] V. Cahill, S. Baker, C. Horn, and G. Starovic. The Amadeus GRT – Generic Support for Distributed Persistent Programming. In *OOPSLA'93*, pp. 144–161, Sep. 93. Tech. Report TCD-CS-93-37.

[6] P. W. Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures.* PhD thesis, Darwin College, Cambridge University (UK), Sep. 91.

[7] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *CACM*, 21(11):966–975, Nov. 78.

[8] J. Fowler and W. Zwaenepoel. Causal Distributed Breakpoints. In *ICDCS'90*, pp. 134–141, May 90.

[9] J. Hughes. A Distributed Garbage Collection Algorithm. In *FPLCA'85*, pp. 256–272, Sep. 85. LNCS, No.201.

[10] N. C. Juul. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System Emerald.* PhD thesis, DIKU, (Denmark), Feb. 93. Tech. Report Nr.93/1.

[11] R. Ladin and B. Liskov. Garbage collection of a distributed heap. In *ICDCS'92*, pp. 708–715, Jun. 92.

[12] S. Louboutin and V. Cahill. A lazy log-keeping mechanism for comprehensive global garbage detection on Amadeus. In *OOIS'95*, pp. 118–132, Dec. 95. Tech. Report TCD-CS-95-13.

[13] S. Louboutin and V. Cahill. On Comprehensive Global Garbage Detection. In *ERSADS'95*, pp. 208–213, Apr. 95. Tech. Report TCD-CS-95-11.

[14] S. Louboutin. *A Reactive Approach to Comprehensive Global Garbage Detection.* PhD thesis, Trinity College, Dublin (Ireland), in preparation.

[15] D. Plainfossé. *Distributed Garbage Collection and Referencing Management in the Soul Object Support System.* PhD thesis, Université Pierre & Marie Curie – Paris VI (France), Jun. 94.

[16] M. Schelvis. Incremental Distribution of Timestamp Packets: A New Approach To Distributed Garbage Collection. In *OOPSLA'89*, pp. 37–48, Oct. 89.

[17] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[18] G. Tel. *Topics in Distributed Algorithms.* Cambridge International Series on Parallel Computation, 1991. ISBN 0-521-40376-6.

[19] P. Watson and I. Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87*, pp. 432–443, Jun. 87. LNCS, No.258/259.