A Survey of Distributed Garbage Collection

Timothy B. Terriberry
Virginia Tech

Abstract

This article presents a selection of current distributed garbage collection techniques, and evaluates them with respect to their completeness, fault-tolerance and efficiency, including promptness, concurrency, and scalability. In particular, techniques based on *reference tracking* are presented, followed by those based on *tracing*. Finally, *hybrid* schemes which extend reference tracking with tracing in order to collect cyclic garbage are examined. The strengths and weaknesses of the existing algorithms are outlined, and directions for further research are suggested.

1 Introduction

Garbage collection has become a common feature modern programming languages [Wei90, CDG+88], because it solves the necessary task of collecting resources which are no longer in use, removing the burden of this complex problem from the programmer. Explicit memory management is a difficult problem, where errors are easy to make, but harder to detect, and much harder to correct. The symptoms of premature deallocation and memory leaks often remain unobserved until execution reaches a point far removed from the original error, or fail to show up repeatedly, or even at all, unless the program is stressed in an unusual way [Wil92]. Tracking these errors down consumes a considerable portion of development time [Rov85]. Conflicting programming styles, concurrent access to the heap, and programming languages with non-deterministic execution orders further exacerbate the problem [PS95, Jon96].

Distributed systems provide additional levels of complexity, as a shared heap can be accessed by many different applications, and may include persistent data which outlives its creating program. All processes must come to an agreement over whether a particular object is garbage, but establishing this in the absence of a consistent, global state is non-trivial. Process failures, network outages, and other system errors further complicate the task of maintaining a robust memory management system. Explicit memory management also defies modularity, since modules must introduce non-local bookkeeping to ensure a module knows when other modules are no longer interested in a particular object [Wil92]. Such bookkeeping is often non-trivial in a distributed system.

Providing an automatic solution to this problem that remains unobtrusive is challenging enough that garbage collection has been actively researched for over 40 years [McC60, Col60]. The problem is well understood in the case of a single stand-alone process, where there are clear tradeoffs between promptness, computational overhead, and concurrency [AR98, Wil92]. However, the conversion of techniques that apply to a uniprocessor environment to a distributed system is not a simple task [PS95]. There is a large collection of algorithms for the distributed case. However, they all make some tradeoffs between completeness, fault-tolerance, and efficiency, including promptness, concurrency, and scalability [Rod98].

1.1 Goals

The goals of a distributed garbage collector should be:

1. **Safety.** The collector should never reclaim objects that are not garbage.

2. **Completeness.** The collector should reclaim all garbage. This is especially important in long-lived distributed systems, where the accumulation of even a small amount of garbage over time can lead to system failures.

3. **Fault-Tolerance.** One of the attractive features of a distributed system is that the failure of one node does not hinder the availability of another. The collector should also continue to operate in the presence of failures due to message delay, loss, re-ordering or duplication, or process failure.

4. **Efficiency.** The efficiency of a system is dependent upon many factors. Some of the important aspects for distributed garbage collectors include:

   (a) **Scalability.** The collector should make minimum use of non-scalable resources such as network bandwidth, CPU time or storage space on each machine.

   (b) **Concurrency.** The collector should not require excessive synchronization with other machines, with the user application, or with a local garbage collector.

   (c) **Promptness.** The collector should reclaim garbage with as little delay as possible. If excessive amounts of garbage accumulate before it can be collected, the system may exhaust its storage space, or page volatile storage to disk, further slowing the system.

A good metric for evaluating collectors on the terms of fault-tolerance and efficiency is that of *locality*. Locality is a measure of the number of machines whose cooperation is required in order to collect garbage. An algorithm is said to have the property of locality if it only requires the cooperation of those machines which contain the garbage [Rod98]. However, locality is not the only consideration in evaluating a collector, as other factors may outweigh its benefits.

This paper will analyze existing algorithms with respect to these goals, with the aim of identifying areas for further development. In Section 2, techniques based on *reference tracking* are examined. Section 3 covers techniques based on *tracing*. Finally, Section 4 examines *hybrid* schemes, which augment reference tracking with some form of tracing to collect cyclic garbage.

1.2 Object Model

A brief overview of the environment used by distributed garbage collection algorithms will help introduce some terminology. A distributed system is divided into one or more separate *machines* or *processes*. Each machine has its own address space, which contains pieces of data called *objects*, for which it is the *host*. Each object may contain *references* to other objects, or be the target of references. The directed graph with objects as vertices and references as edges is called the *object reference graph*. Each machine contains a *mutator*, whose sole purpose, as far as the collector is concerned,

is to modify the object reference graph. This represents the actions of the user's application.

References to other objects on the same machine are called *local* references, while those to objects on different machines are called *remote* references. Objects which can be referenced by another machine are called *public*, while those which can only be referenced by other objects on their host machine are called *local*. Some special objects are identified as *roots*, which are never garbage (i.e.: global variables, the run-time stack for a thread of execution, etc.). Objects which are reachable by traversing references from a root are *live*, while those that are not are *garbage*. Objects which are reachable only by traversing local references from a root are said to be *locally reachable*, and have a *local root*.

Remote references are represented by pairs of special objects, called *entry-items* and *exit-items*. The host of a public object contains a single entry-item for that object, which contains a local reference to it. A machine containing a remote reference to the object will have an exit-item, containing a *locator*. The host machine uses the locator to find the entry-item corresponding to the object, and thus obtains a local reference to it. The remote reference consists of a local reference to the exit-item, which in turn is used to identify an entry-item on the host machine, which contains a local reference to the actual object.

There are four basic operations on remote references. First, references may be created by the machine that hosts an object if it sends a locator to a remote machine, which constructs an exit-item from it. Next, references may be duplicated if one machine sends a locator to another, allowing it to construct its own exit-item. This does not require any interaction with the host of the object, only a machine that already has a locator for that object. Also, references may be traversed by passing their locator as the parameter to a remote procedure call to the host machine. This has the effect of creating a new local root for that object, in the thread that handles the call. Finally, a reference may be deleted by deleting the corresponding exit-item.

2 Reference Counting

Reference tracking and its variants are the most common forms of *direct* identification of garbage. Direct identification of garbage proceeds by detecting which objects are not reachable by traversing references from a live object and marks them for reclamation. All other objects must be live. Reference tracking is attractive in distributed systems for two reasons: its operation is interleaved with that of the mutator, allowing for a high level of concurrency, and its operation does not require any global information, meaning that it has good locality.

Reference tracking is usually considered too expensive for uniprocessor systems, since it adds overhead each time a reference is modified, yielding a cost proportional to the total amount of work done by the system. Its use in distributed systems, however, is quite different, in that the overhead is only applied when distributed references are created or copied. These operations tend to be much rarer, since they involve communications overhead to send the references to different machines.

2.1 Basic Reference Counting

The idea behind reference counting is simple [Col60]. Each object keeps a count of the number of references to it in its entry-item. Creating a new reference increments the count, and when the reference is destroyed, it is decremented again. When the count reaches zero, the object is garbage and can be reclaimed. This procedure is applied recursively, decrementing the counts of objects the reclaimed object referred to, and continuing to reclaim objects whose counts drop to zero.

This operation can cause unbounded delays while large portions of the reference graph are reclaimed after a single reference is destroyed. The solution is to place reclaimed objects in a queue instead of decrementing the count of objects they refer to immediately [Wei63]. When the storage for the objects is about to be reused, the old references in the object are cleared, and their referents' counts decremented. In this way, only one object is ever processed at a time, since the objects it refers to are merely queued for later processing if they are reclaimed. This provides fine-grained concurrency with the mutator, as the amount of work done at each step is at most proportional to the number of references in a single object.

In a distributed system, only remote references need to be tracked. All the objects on one machine can share a single exit-item, so that local operations impose no overhead. Each machine can run its own local garbage collector to handle local references, using any of the uniprocessor techniques. Objects with a positive number of remote references are considered live by the local collector. When the remote reference count reaches zero, the object may be reclaimed as soon as the local collector also determines there are no local references to it.

Whenever a remote reference is duplicated, or deleted, *increment* or *decrement* messages must be sent to the host of the object. However, messages may be lost, delayed, or re-ordered. If an increment message is sent followed by a decrement message, then the decrement message might arrive first, causing the object to be unsafely reclaimed before the increment message arrives. Furthermore, if a decrement message is lost, or an increment message is duplicated, then the object might never be reclaimed. Also, if a process crashes, it will never send decrement messages for any remote references it contained, which again will prevent those objects from ever being reclaimed. Finally, reference counting is not complete even without any failures. If two objects hold references to each other, neither will ever be reclaimed.

2.2 Acknowledgement Messages

There are two types of race conditions between increment and decrement messages that must be avoided, depending on who sends the increment message when a reference is copied from one machine to another. Both race conditions can be avoided by using acknowledgement messages [LM86].

If the receiver of a reference is responsible for sending the increment message, a *decrement/increment* race condition may occur, when a machine sends a remote reference to another machine, and then deletes its own reference (see figure 1). If the sending machine had the last reference to the object, and its decrement message arrives before the receiving machine's increment message, the object might be unsafely reclaimed. To avoid this, when a machine receives a copy of a remote reference, it sends along with the increment message an acknowledgement request. The owner of the object then sends an acknowledgement to the machine that sent the copy of the reference. That machine must queue any decrement messages until it receives this acknowledgement.

If the sender of a reference is responsible for sending the increment message, an *increment/decrement* race condition may occur, when a machine sends a remote reference to another machine that immediately deletes it (see figure 2). If the sending machine had the last reference to the object, and the receiving machine's decrement message arrives before the sending machine's increment message, the object might be unsafely reclaimed. To avoid this, when a machine wants to send a reference to another machine, it first sends an acknowledgement request along with its increment message. The owner of the object then sends an acknowledgment to the sending machine. Once the sending machine has received this acknowledgement, it can send a copy of the reference to the receiving machine.

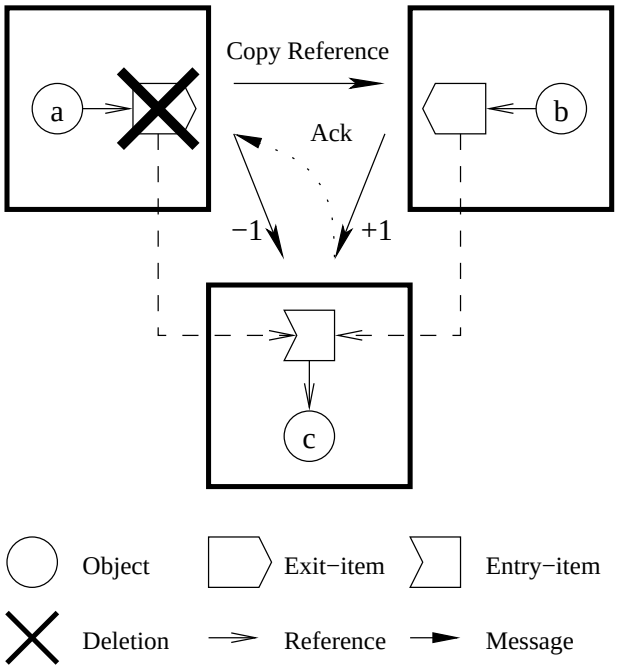

Figure 1: A decrement/increment race condition

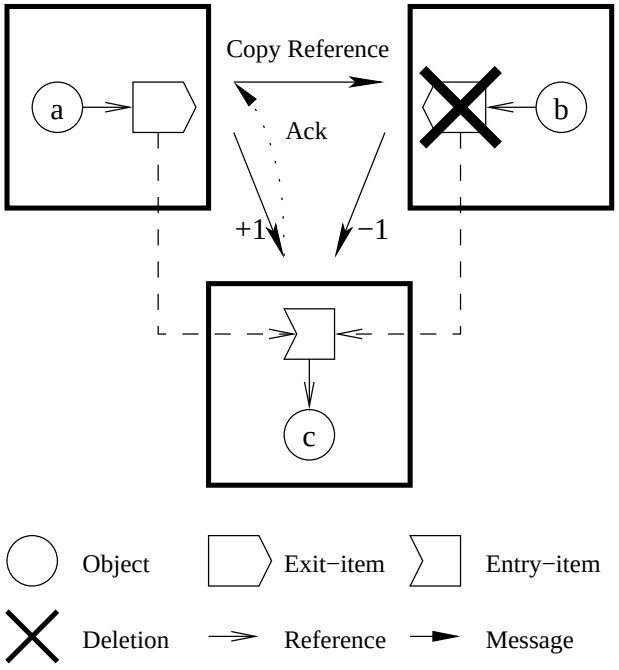

Figure 2: An increment/decrement race condition

Even with these acknowledgement messages, reference counting is still not safe in the face of lost or duplicated messages. Several other reference tracking schemes show better resilience to these message failures.

2.3 Weighted Reference Counting

Weighted reference counting is a variant of reference counting that eliminates the increment message, which improves message efficiency and avoids race conditions [Bev87, WW87]. In addition to associating a count with each object, a weight is associated with each reference. The system keeps as invariant that the sum of the weights associated with all the references to an object is equal to the object's reference count.

When an object is created, its count is initialized to its maximum value, and the first reference to it receives that value as its weight. When a machine wants to copy a reference, it divides the weight of that reference in half, and associates the other half of the weight with the new reference. When a reference is deleted, it returns its weight in the decrement message, which is subtracted from the object's count. When an object's count reaches zero, no references to it may exist, and so the object can be reclaimed.

Although this reduces the number of messages and eliminates race conditions, it requires extra space overhead to store the weight with each reference. In practice, this can be stored as the logarithm of the weight. However, it must be expanded to perform the subtraction once the reference is deleted. Also, the algorithm is still not resilient to message loss or duplication, or process failure.

The most serious drawback, however, is that once a weight reaches some minimum granularity, it cannot be further subdivided. One solution to this is to request additional weight from the owner of the object, but this is equivalent to an increment message. Another is to create an indirection object. The indirection object has its own count, and forwards messages sent to it to the original object. In the worst case, long chains of indirection objects might be required, which would greatly increase the message complexity of a computation.

By weakening the invariant so that an object's count is only greater than or equal to the sum of the weights of all references to it, both message loss and reference duplication can be addressed [Dic92]. When a reference cannot divide its weight any longer, it is discarded, and replaced with a special *null weight* value, which can be copied any number of times. Both this and message loss would cause an object's count to exceed the sum of the weights of its references, which would prevent it from being collected. However, reference counting is not complete in the first place, so this does not hinder a scheme where reference counting is augmented by some other, more expensive technique to achieve completeness, as in the hybrid algorithms described in Section 4. Message duplication, however, could still cause an object to be unsafely reclaimed.

2.4 Indirect Reference Counting

Indirect reference counting improves on weighted reference counting by avoiding indirection objects, though it requires additional memory [Piq91]. In this case, instead of associating a weight with each reference, every reference gets its own counter. When a reference is copied, its local counter is incremented, and the copy stores the location of the reference it was copied from. Then a reference cannot be deleted until its own local counter reaches zero. When this happens, it sends a decrement message to the reference that it was copied from instead of the original object.

Since increments are done locally, there are no race conditions associated with indirect reference counting. However, it is still not resilient to message loss or duplication, or process failure. The same technique can be used to reduce the message overhead of

more reliable forms of reference tracking, though, such as the one presented in the next subsection. The reduced number of messages is offset by slightly larger messages, as each reference now contains two pointers: a *weak* pointer to the actual object, and a *strong* pointer to the sender of the reference. In addition to the space overhead required to store both pointers, each machine may also be required to keep around references which it would have otherwise deleted, because it has sent copies of them which are still in use.

2.5 Reference Listing

In order to deal with message loss and duplication, and process failure, reference listing replaces the standard reference counter with a list of processes that hold a reference to an object [Sha91, SDP92, BNOW94]. This adds a significant space overhead to maintain the list, but provides the robustness that other reference tracking schemes lack. Instead of increment and decrement messages, insert and delete messages containing the identity of the process holding the reference are sent when a reference is copied or deleted, respectively.

Lost messages can be overcome by re-sending failed insert or delete messages [BNOW94], or by periodically sending the entire list of references one machine has to objects on another machine [Sha91, SDP92]. Although distributing lists of references is attractive because it is simple to implement and does not require any acknowledgements to ensure delivery, in practice these messages can be very large, while queued and retransmitted messages and their associated acknowledgements are very small. Furthermore, in a truly unreliable networking environment, where large packets must be broken up into smaller ones for delivery, it becomes increasingly unlikely that even one entire list of references could be transmitted without errors. Duplicate messages can be safely ignored, since a process can only be added to the reference list once, and can only be deleted once. Race conditions must be handled by using acknowledgement messages, or a scheme that avoids insert messages altogether, such as indirect reference counting. Process failure can be handled by removing processes from the list which are known to have failed.

Reference listing does introduce an additional race condition, however. If a machine deletes a reference to an object, and later acquires a new reference to the same object, its delete message might not arrive until after the insert message, in which case the insert message would appear to be a duplicate and be discarded. This can be avoided by time-stamping insert and delete messages, and storing the largest timestamp of an insert message in the reference list. If a delete message is received with a timestamp that is smaller then the one in the reference list, it can safely be ignored.

Reference listing is therefore completely resilient to message loss, duplication, or re-ordering, as well as machine failures. This safety comes at the cost of additional storage overhead for maintaining the reference lists, however this is probably acceptable if only used for public objects. Like all reference tracking techniques, it displays good scalability, concurrency, and promptness. The major drawback is that it cannot collect cyclic garbage, and so is not complete.

3 Tracing

The various types of tracing algorithms are the most common *indirect* garbage collectors. Indirect identification of garbage proceeds by detecting which objects are reachable by traversing references from a live object and marks them as live. All other objects must be garbage. Tracing algorithms often require a good deal of synchronization between machines and with the mutator. Furthermore, the trace must pass through all live objects, and all processes must be notified when it is complete, which destroys the property of locality. However, unlike reference tracking schemes, tracing can collect garbage that belongs to distributed cycles. Tracing algorithms are some of the most successful on uniprocessor machines, but their poor locality and increased complexity make their extension to a distributed environment less appealing.

Each individual machine can still utilize its own local collector to reclaim garbage that is not referenced by other machines. Tracing is then used to identify the remaining garbage objects. As long as each local collector is complete, the global tracer need only consider references between different machines.

3.1 Basic Global Tracing

Tracing algorithms generally operate in two distinct phases. The first phase, or mark phase, traverses all objects reachable from the roots, such as global variables and pointers on the run-time stack. The second phase, or sweep phase, reclaims all objects that were not marked by the first phase, as they must be garbage.

In a simple scheme [MA84], a machine that is running low on storage space may initiate garbage collection by sending a request to some controlling machine, or *coordinator*. The coordinator ensures that all machines suspend their mutators, so that the reference graph cannot be modified during the marking phase. Each machine then marks all its live objects, and sends mark request messages for any remote reference it encounters. Machines alternate between an *active* state, where they are marking objects, and an *idle* state. Mark messages may only be sent by active machines, and receipt of a mark message by an idle machine makes it active again. When all machines are idle, the mark phase is complete. General distributed termination detection algorithms can detect when all machines are idle, though more complicated schemes may use custom methods that are less expensive, or that deal with mutator concurrency better.

After the mark phase is complete, each machine resumes its mutator and begins its sweep phase. Because all the objects reclaimed in the sweep phase are not reachable from live objects, there is no mutator contention for them, so no synchronization needs to be done.

As an initial approach, this method does not seem to be very satisfactory. Because it requires the cooperation of every machine in the system, it is neither fault-tolerant or scalable. In addition, even if only one machine needs to reclaim garbage, all other machines are forced to do so as well. It also does not allow any work to be done while the mark phase is in progress. Synchronization of the mark phase itself requires all idle machines to wait until the last one is finished to resume doing useful work. The next few subsections will cover how some of these problems can be alleviated.

3.2 Marking-tree collector

One of the earliest distributed tracing collectors, the marking-tree collector, is based upon a concurrent version of the mark and sweep algorithm for uniprocessor systems [HK82]. Unlike the naïve approach, identification of garbage and mutator operations execute concurrently.

It is assumed that there is a single root of the entire distributed reference graph, from which marking begins. Each machine maintains a queue of mark tasks and mutator tasks, which it executes in turn. During a mark task, additional mark tasks may be created for objects that are not already marked, forming a tree. When a mark task completes, it spawns an up-tree-task. A mark task is not complete until it has received an up-tree-task for each mark task it spawned. When the root receives all of its up-tree-tasks, the marking phase is complete. This tree requires a large space overhead, proportional to the number of live objects in the system. This can be extended to a computation with more than one root by constructing

a tree from each root, and declaring termination only when every root's mark task is complete.

Objects are marked with one of three colors.

- A *white* object has not yet been visited by the marking phase, and will be considered garbage if it is still white when the mark phase terminates.
- A *gray* object has been visited by the mark phase, and has spawned mark tasks for each reference it contains. Once all of its mark tasks are completed a gray object is colored black.
- A *black* object has either been allocated after the mark phase began, or was previously gray, and has had all its mark tasks completed. Black objects have been visited by the collector and need not be visited again. At the end of the mark phase, all live objects are black.

Mutator tasks and mark tasks compete to modify objects, locking all the objects they intend to modify. Mutator operations must preserve the invariants associated with an object's color, which can introduce significant delays. For example, copying a reference to a white object into a black object requires that the mark task for the white object be executed (not just queued) and spawn its up-tree-task, which could cause large portions of the reference graph on many machines to be traversed.

However, the algorithm does allow some mutator concurrency, though the mark phase must still be synchronized across every machine.

3.3 The Emerald System

The Emerald system is also based upon a three color tracer, but attempts to provide better mutator concurrency and achieve completeness despite temporarily unavailable machines [JJ92]. Mutator concurrency is achieved with an *object-fault* mechanism akin to a virtual memory system's page-fault. Instead of preventing the copying of a white reference into a black object explicitly, this system imposes a read barrier that protects all gray objects. Attempting to traverse a reference to a gray object causes a fault, wherein the collector shades all the objects reachable from it gray and colors the object black. An object is colored black as soon as all its referents are colored gray, instead of waiting for them to be colored black as well. This prevents large portions of the reference graph from being traversed to satisfy a single attempted reference of a gray object.

Since a black object may never have a reference to a white object, and all gray objects are colored black before allowing the mutator to read them, the mutator may never obtain a reference to a white object to copy to a black one. To avoid arbitrary communication delays while waiting for a remote object to be colored gray, each machine also maintains a non-resident gray set. This set contains remote references for which mark messages have been sent, but not acknowledged. They are treated as references to gray objects, though the shading of the actual objects may be postponed. Once the remote site does color them gray, it acknowledges the mark message, and they are removed from the non-resident gray set. Finally, to prevent a black object from traversing one of these references to a remote object that is still white, objects that are remotely invoked are also colored gray before the invocation proceeds.

Because only one level of objects is colored before a gray object colors itself black, the diffusing tree algorithm from the previous subsection cannot be used to detect termination of the mark phase. Instead, termination is detected using a two-phase commit protocol that is resilient to process and message failures, but requires each process to know of every other process in the system, which detracts from scalability.

3.4 Tracing with Time-stamps

Time-stamps can also be used in a fashion similar to the colors of previous examples [Hug85]. Instead of marking an object with a color, it is marked with a time-stamp. The mark phase essentially runs continuously, so that the time-stamps of live objects continuously increase. The time-stamps of garbage, however, eventually stop increasing. A global threshold is then computed, so that objects with a time-stamp below that threshold can be safely reclaimed.

The actual marking operations can be built directly into the local collector, provided it is also some form of tracing collector. Each machine keeps a clock, such as Lamport's logical clock, and records the start time of a local collection, called the *GC-time*. Local roots are then time-stamped with this time, while public objects keep the last time-stamp placed in them, or the time-stamp from when they were created if they have not been marked before. The local collector then propagates these time-stamps throughout the local machine, so that all remote references are marked with the largest time-stamp of any object from which they are reachable. This can be achieved by processing mark requests in descending order of their time-stamps. The local collector is responsible for allowing mutator concurrency, so that standard uniprocessor techniques can be used.

These time-stamps are then forwarded to their corresponding remote objects in *time-stamping* messages, and their current time-stamps are replaced if smaller, similar to the mark messages of a standard tracer. Each machine also maintains a local threshold value, called a *redo* time-stamp. Whenever a time-stamp is replaced and the redo value is larger than the old time-stamp, the redo time-stamp is reduced to the object's old time-stamp. Whenever a site completes a local collection and receives acknowledgement messages for all the time-stamping messages it sent out (and has not received any additional time-stamping messages), it sets its redo value to the GC-time. The global minimum of these redo values is then the global threshold, below which objects may be reclaimed. This propagation of time-stamps can be seen as multiple, concurrent mark phases operating in parallel. As the global threshold increases, it can be assumed that the mark phases corresponding to time-stamps below it have completed.

However, computing this global threshold requires a costly termination detection algorithm, and involves the cooperation of every machine, which means that the algorithm has just as poor locality as other tracers. Furthermore, if even a single process fails, the global threshold can never increase beyond its last reported redo value, which means that collection will eventually stop. Even if a process is merely slow, or rarely triggers a local collection, the threshold can be prevented from increasing far enough for other machines to collect needed garbage, regardless of the fact that the slow machine might not even have any remote references.

3.5 Logically Centralized Tracing

Instead of having each machine participate in the tracing and try to piece together the results from other machines, another approach is to off-load this task to some centralized service. As each machine performs its local collections, it informs the centralized service of the remote references it contains. Periodically, a machine asks the centralized service if any of its public objects that are not locally reachable still have a remote reference to them. If not, the object can be reclaimed.

Garbage cycles that span machines are collected using time-stamps, as in the previous section. Every object that could be remotely reachable is time-stamped with the last time it was accessible from some machine. The centralized service maintains a copy of the time-stamps, and computes the threshold time, avoiding a

costly termination detection algorithm. Unlike the previous section, the time-stamps are used only to collect cyclic garbage.

The centralized service can be replicated to achieve fault-tolerance, however this does not aid scalability. As the number of processes increases, the service must still communicate with all of them in order to collect any garbage. Furthermore, it requires storage proportional to the total number of public objects in the entire system.

4 Hybrid Schemes

Reference tracking systems have good locality and can be made fault-tolerant, and they are prompt and highly concurrent. Their main drawback is that they cannot collect cyclic garbage. Tracing algorithms, on the other hand, are complete, but they are not scalable, and they are not very prompt, since an entire trace must complete before any garbage can be collected.

Some systems trade off completeness for efficiency and fault-tolerance, assuming that distributed cycles are uncommon enough to be ignored. This may be true for short-lived systems, or if there is sufficient memory to allow storage leaks without excessive paging. However, for long-lived distributed systems, even small leaks accumulate over time and are unacceptable. Furthermore, systems commonly create cycles for such things as client-server communication, replication, or mobile computing. Without a cyclic garbage collector, such cycles must be broken manually, and there is no good methodology of doing this [Rod98].

The obvious solution is to try to use reference counting to collect acyclic garbage, and a tracing algorithm to collect just the cyclic garbage. However, this still has the same drawbacks as a stand-alone, tracing collector, only on a smaller scale.

4.1 Tracing in Groups

One algorithm attempts to use groups of processes to alleviate the problems of scalability and fault-tolerance associated with standard tracing algorithms, as mentioned in the previous section [LQP92]. Reference counting is used to collect all acyclic garbage, while a mark-and-sweep collector is used to collect garbage cycles within a group. However, groups are not fixed. Instead, groups can be configured dynamically, and can overlap with other groups. Collections from different groups can run on the same machine in parallel, though this requires more computation by the local collector.

This algorithm gives a rough approximation of locality, since garbage cycles contained in a single group can be collected without the cooperation of machines outside the group. Fault-tolerance is achieved, since even if a machine faults the remaining machines can still form a group and continue the trace. Promptness is also improved, since small groups can collect small cycles without waiting for a trace to complete in all the machines in the system. The main problem with the algorithm is that no clear method of forming groups is presented. A tree-like hierarchy of groups is suggested to ensure that every cycle is covered by some group. However, the smallest group containing a particular cycle may still contain a large number of machines.

4.2 Local Tracing

Local tracing algorithms also combine a reference counting algorithm–weighted reference counting–with a tracing algorithm [JL92, MKI+95]. Unlike previous examples, tracing does not begin from the roots of the reference graph, but from objects that are suspected of being garbage. Suspect objects must be chosen by some heuristic. The proposed heuristic was to trace any object that had just had a reference to it deleted, though this is clearly very expensive. This tracing is similar to an earlier technique, called trial deletion [Ves87].

The tracing proceeds by copying the reference counter of the objects reachable from the suspect object, and then decrementing them as if the suspect object had been deleted. If, at the end of the trace, the suspect's counter has also been reduced to zero, then it was actually garbage, and can be deleted. This algorithm only approximates locality, since some of the traced objects may actually have been live. However, it is more scalable than global tracing, because the entire reference graph does not need to be traversed, but instead only those objects whose counts drop to zero and their immediate neighbors.

The initial algorithm also had the drawback that the local collector was required to be based on reference counting [JL92], though a later proposal eliminated this problem [MKI+95]. The algorithm must also ensure that two different traces do not try to modify an object's count simultaneously. This requires either a global synchronization method that ensures that only one trace is performed at a time, or extra storage to keep a separate count for every trace. Both solutions are likely to be too expensive in practice.

4.3 Partial Tracing

Another algorithm, called partial tracing, resolves some of the issues that remain with tracing in groups and local tracing [Rod98]. It uses reference listing to collect acyclic garbage, and also identifies suspect objects using some heuristic, and performs a trace from there to collect garbage cycles. Instead of using trial deletion to detect if an object belongs to a garbage cycle, a three-phase mark-and-sweep tracer is used.

The first phase marks red all objects that are reachable from the suspect object, but which are not reachable from a local root. These objects are said to belong to the *suspect sub-graph*. The second phase, called the scan phase, colors objects green which are reachable from outside the group of objects marked red by the first phase. These objects can easily be detected by remembering which machines sent an object a mark red message, comparing this to its incoming reference list, and then propagating this information throughout the suspect sub-graph. These objects are the ones conservatively estimated to be live. Marks are propagated from incoming remote references to outgoing remote references by the local garbage collector, which is assumed to be tracing based. The final phase, a sweep phase, reclaims all the objects that were marked red by the first phase, but not marked green by the second phase.

The advantage of this algorithm is that different partial traces can cooperate. If two traces meet in the mark red phase, then they form a group of cooperating traces, neither of which will proceed to the scan phase until both have completed their mark red phases. As additional traces meet, the group grows, so that two traces are in the same group if their suspect sub-graphs intersect. If a trace in the mark red phase meets another in the scan or sweep phases, it can safely assume it has no garbage cycles through that part of the sub-graph. If the two traces contained objects on a common garbage cycle, then the second trace would had to have met the first during its mark red phase, and would not have proceeded to its scan phase.

During the mark red and sweep phases, there is no contention between the mutator and the tracing algorithm, so no synchronization is required. Mutator concurrency is achieved during the scan phase by imposing a write barrier during the first local step on each machine, which detects changes in reachability between incoming remote references and outgoing remote references. After the first local step, as mark green messages propagate between machines a remote barrier marks green any red object which has one of its remote references traversed, along with any outgoing remote references which were reachable from that red object. The remote

barrier is less expensive than the write barrier, however the write barrier is necessary at the beginning of the phase, since red objects could have acquired new local roots, or the connectivity between incoming and outgoing remote references could have changed during the mark red phase.

This method provides a clear mechanism for forming groups of machines (those encountered in the mark red phase) which is defined by the shape of the reference graph. However, it still has only an approximation of locality, since the mark red phase may trace through live objects (namely, those marked green in the scan phase). Completeness can be traded off against locality by limiting the extent of the mark red phase, or by limiting which traces can join a group. This tends to increase promptness, fault-tolerance, and scalability, at the cost of floating garbage, which may accumulate over time. No clear method to tell when these tradeoffs should be made is presented.

Promptness and fault-tolerance are also very dependent upon the heuristic chosen to identify suspects. Promptness is obviously affected, since the heuristic may impose long delays between the time an object becomes garbage and the time it becomes suspect, in order to avoid wasted work. Fault-tolerance is also affected, since if a machine belonging to a group fails, then the current trace will fail. A subsequent trace might succeed, if the failed machine did not contain any objects on a garbage cycle, because the new trace would not include the failed machine in the group. However, the heuristic controls the next time an object participating in a failed trace becomes suspect.

4.4 Train Collection

The train algorithm is an adaptation of another uniprocessor algorithm which is complete, concurrent and scalable [HMMM97]. It partitions the address space into disjoint regions called *cars*, each of which resides on a single machine. Again, a local collector reclaims garbage cycles within a car, but to collect garbage which spans more than one car, cars are grouped into *trains*. Collection proceeds in a train by moving the reachable objects to other trains. The remaining objects in the train are cyclic garbage, and can be reclaimed.

Reference listing is used to keep track of which cars contain references to an object. The same protocol is used to notify an object when a reference to it has been moved from one car or train to another. Cars in a train form a ring, with the creator of the train as the master. In order to join a train, an car communicates with the master, who informs the other cars in the train. In order to leave a train, *leave* messages are propagated around the ring. Any number of leave messages may be propagated simultaneously, but they may not pass each other. In either case, only the machines containing cars in the train are involved.

When there are no references into a train from cars outside the train, the entire train can be collected. This is detected using a distributed termination detection protocol that only involves machines in the train. Special handling is required to account for cars that wish to join or leave the train while this algorithm is running.

The algorithm achieves better locality than some of the previous examples, since only those cars in a single train are required to collect the garbage cycles in it. This implies good locality and better promptness than global tracing algorithms. However, this is still not optimal, since a single train may contain more than one garbage cycle. Fault-tolerance is achieved to the degree that failure of a machine in one train does not hinder the collection of garbage in another train, however the algorithm would need to be extended to handle failures within a single train. Only the reference listing operations need to be synchronized with the mutator, which gives the algorithm good concurrency. The major drawback is the message complexity of moving an object to a new car or train, which requires updating all the old references to the object.

4.5 Timestamp Packet Distribution

A slightly different approach to distributed garbage collection is to attempt to construct causality information between mutator events which modify the reference graph [LC97]. Knowing the causal history of these events allows garbage to be detected that otherwise would require some global state. This is a direct method of identifying garbage, like reference counting, though the propagation of the causal history of an object is akin to tracing with time-stamps.

Causal relations between events can be represented using vector times, which maintain a given machine's view of the time at every process in the system when the event occurred. This information is propagated between machines whenever a message is sent, by updating the time at a remote machine if it is less than that contained in the message. The underlying reference listing scheme circulates these approximations until it can be determined that an object is no longer reachable from a root.

The algorithm has all the benefits of reference tracking, in addition to being complete, yet has none of the drawbacks of tracing. Vector time propagation is only required to pass through a garbage cycle in order to collect it, giving this algorithm perfect locality. However, every object must maintain a set of vector times corresponding to the paths it is reachable from, and each vector time is proportional to the total number of machines in the system. The corresponding space overhead is therefore very large, and hinders scalability. The algorithm is also very complicated, and some issues with mutator concurrency remain unclear.

4.6 Object Migration

Object migration attempts to collect garbage cycles between machines by moving all objects on a cycle to the same machine [Bis77, ML95, GF93]. From there, the local collector can detect the garbage and reclaim it. Again, a heuristic is used to determine when an object is suspected of belonging to a garbage cycle. Suspect objects are then migrated to one of the machines that refer to them.

This mechanism also exhibits perfect locality, since only the machines that originally contained the garbage items are required to migrate them to a single machine. However, it has a number of serious drawbacks, as well. Migrating objects can create additional indirection objects to allow it to traverse its old references, and eliminating these indirections requires additional messages. Also, migration is an expensive operation, which may require the transmission of large amounts of data that is likely to merely be thrown away. This is aggravated by the fact that an object may need to be migrated several times before the cycle it is on is consolidated in a single machine. Migration may also overload a single machine with a large collection of garbage objects before any can be reclaimed. Finally, object migration must be supported in the first place. In a heterogeneous network, different computer architectures may make this extremely difficult in practice.

4.7 Back Tracing

Back tracing is a variation of local tracing that preserves locality. Reference listing is used to collect acyclic garbage, and a heuristic identifies the suspect objects that remain [Fuc95, RRR97, ML97]. However, instead of following the references contained in a suspect object, the tracing algorithm instead traverses back along the references *to* the suspect object. If no roots are found among the objects that can reach the suspect, then all the traced objects are garbage, and can be reclaimed. Faulty machines encountered in the

back trace are assumed to contain live objects, which makes the algorithm safe in the presence of failures. Message identifiers and acknowledgements handle message duplication or loss.

When introduced, back tracing had four main challenges to overcome [Fuc95]:

- A good heuristic is required to detect suspect objects. This same challenge is faced by many of the hybrid algorithms, and is addressed below.
- There is considerable overhead on the local collector in computing the backwards reference information. A later proposal introduces an algorithm which only visits each object once [ML97]. However, the space required for storing full reachability information between incoming and outgoing references is prohibitive. This can be made tolerable by restricting traces to suspect objects, but this only increases the dependence on a good heuristic for identifying suspects. Furthermore, it requires knowing which incoming references come from suspect objects, which requires additional messages. Caching of back information may avoid re-computation for objects that have not changed between traces, however this does not help the space overhead.
- Multiple back tracings must be synchronized with each other. The current literature allows multiple tracings to exist independently, but provides no means for overlapping tracings to share information and avoid repeated work. One author suggests imposing a total ordering on the different tracings, and blocking lower priority ones until higher priority ones have completed [Fuc95]. This alleviates some extra work in the case that the objects being traced are in fact garbage, provided the highest priority trace is the first to encounter an object. If a lower priority trace arrives first, it will have already traversed part of the graph that must now be re-traversed by the higher priority trace. If an object is actually live, then every trace will still traverse it.
- The back tracing must be synchronized with the mutators. Recent work demonstrates that this can be done by adding a barrier to reference creation and remote method invocation, without imposing any additional remote synchronization [ML97].

Back tracing's good locality, fault-tolerance and concurrency make it attractive as a complete solution. However, the wasted work by repeated tracing and the space overhead could prove excessive in real applications. Another problem is that the mutator could indefinitely create new backwards paths while a trace is in progress, which would prevent it from ever terminating. This would destroy promptness, and garbage would accumulate until the system exhausted its storage. However this situation is likely to be rare, and can be avoided by abandoning the trace after traversing a large number of references (wasting the work).

4.8 Heuristics

Many of the hybrid algorithms perform their tracing not from the roots of the reference graph, but from objects they suspect of being garbage. For these algorithms, it is very important that a good heuristic be used to determine when an object is suspect, as wrongly suspecting an object can lead to a good deal of wasted work. The heuristic originally proposed for local tracing was to suspect any object that had a pointer to it deleted. This heuristic was not just limited to when remote references where deleted, but applied to any pointer to any object. If only remote references were considered, modifications to the object graph on a single machine could disconnect a cycle from the live portion of the graph without causing a remote reference to be deleted. This is clearly far too expensive.

One property every heuristic should have is that no suspect object should be reachable from a root on the same local machine. This information can be computed by the local garbage collector, and does not require any message passing to determine. Clearly any object that does not satisfy this condition is not garbage, so it can be added to any heuristic.

The simplest heuristic is to use this as the only condition for suspect objects. However, it is believed that in long-lived distributed systems there are many objects that are not locally reachable, but still live [ML95]. This could lead to an unbounded number of failed traces.

To alleviate this problem, imposing a delay between traces has been suggested [GF93]. This delay could be increased with each failed trace, similar to generational schemes from uniprocessor algorithms, which take advantage of the fact that objects which have survived previous collections are far less likely to be garbage than newly allocated objects. However, the same patterns may not hold on long-lived distributed systems, where it is more expensive to create and use distributed objects. Thus, it may be likely that many objects are not accessed for long periods of time. In this case, such a delay would hurt promptness, since it would be imposed on most of the objects in the system, while still wasting an unbounded, albeit relatively smaller, amount of work on failed traces.

Another technique, called the *distance heuristic*, is stable for unchanging portions of the reference graph [ML95]. That is, so long as the structure of the reference graph does not change, it does not cause any new traces to begin. However, it uses messages in order to determine this information, so it is more costly than previously suggested heuristics.

The idea is to estimate the shortest "distance" of an object from any root, where the distance is measured as the minimum number of remote references that must be traversed to reach the object. Each public object maintains a distance for each remote reference to it. Newly created remote references have a conservative estimate of one. Roots also have an associated distance, which is always zero. The local collector propagates distance estimates from local roots and public objects to remote references. Changes in the distance estimate of a remote reference are sent to the object it refers to, which updates its estimate accordingly. The observation is that for live objects, the distance estimate eventually stabilizes at the actual distance. However, for garbage cycles, the distance estimate continuously increases.

This is similar to tracing with time-stamps, however it does not suffer from some of the drawbacks. Mainly, the failure of a node does not prevent the distance estimates of objects in a garbage cycle not involving that node from continuing to increase. This allows the system to set an arbitrary suspicion threshold distance. Objects with a distance estimate above the threshold are considered suspect. The threshold can be modified dynamically if too many live objects are being traced, or even on a per-object basis. Furthermore, multiple thresholds can provide multiple levels of suspicion. For example, back tracing can use one threshold to allow objects to start a trace, and a smaller threshold to allow the trace to pass through an object, which helps alleviate the presence of multiple simultaneous back tracings on a single cycle, and reduces the required space overhead.

There are still several problems with the distance heuristic, however. One is that it sends extra messages for all remotely reachable objects, regardless of whether they are likely to be garbage or not, e.g. whether or not they are also locally reachable. Another is that every cycle, no matter the size, must propagate at least as many messages as the (conservatively large) threshold value before becoming suspect. This means even a cycle involving two or three objects has to do the same work as a cycle involving hundreds or thousands in order to be collected. The effect is to trade one form of wasted work for another, sacrificing promptness as well. For algorithms which only approximate locality, such as partial tracing,

or which involve a significant amount of work per trace, such as object migration, this trade-off might be acceptable.

5 Conclusion

As has already been noted, reference tracking systems have good locality, are prompt, and highly concurrent. The reference listing variant is also completely fault-tolerant. The only drawback is that they are not complete, since they cannot collect cyclic garbage. Tracing algorithms provide completeness, but very little else. They are not prompt, and exhibit extremely poor locality.

Hybrid schemes offer a compromise between the two, and are probably the best area for further research. There are two important ideas that arise repeatedly in hybrid algorithms. The first is organizing processes into small groups. The second is the idea of applying tracing to suspected garbage objects instead of known live ones.

The first idea is seen in sections 4.1 and 4.3. Small cooperating groups of processes provide better promptness, scalability, and fault-tolerance than a single global tracer. However, the organization of processes into groups is a non-trivial matter. If the groups do not coincide with the distribution of cyclic garbage, then the cooperation of many groups may be required to collect it. The mark phase of [Rod98] gives a method of constructing groups by tracing from suspect objects, but suggests that these may be too large in general. Several factors are mentioned that should be considered when limiting the size of these groups, however no clear heuristics are presented for evaluating when a machine should be added to a group, or when two groups should merge. Developing such heuristics, and comparing them in simulations or on actual systems would clear up some of these issues.

Examples of tracing from suspect objects is found in sections 4.2, 4.3, 4.6 and 4.7. In a pure tracing environment this is not possible, but with reference listing an object has more information about where it is reachable from. This makes it possible to determine if an object is garbage without examining the entire reference graph. Again, this provides better promptness, scalability, and fault-tolerance. Of these methods, back tracing, whose fundamental principle is based on this observation, provides the best locality and communication overhead. However, there are still some unresolved issues with back tracing, such as the cooperation of multiple traces and preventing the tracer from following indefinitely long chains of backwards links. Methods for resolving these two issues would certainly be of interest.

The other obvious requirement for this idea to be successful are good heuristics. Very little is known about the behavior of long-lived distributed systems, which hinders the development of accurate heuristics, and comparisons between them. However, it is fairly certain that the naïve heuristics are too inaccurate, causing many unnecessary traces. But the distance heuristic requires many extra messages, especially for live objects and portions of the reference graph that are changing frequently. Furthermore, its uniform delay, even for small cycles of garbage, hurts its promptness. A good heuristic would operate entirely from local information, and would suspect garbage objects immediately. The best place for it to obtain information would be from failed traces. Since each machine has some knowledge of the outcome of a trace, the heuristic could use this to refine its view of the global state.

Garbage collection is a much harder problem in a distributed environment than with a single address space. The lack of any consistent, global state and the increased cost of communication complicate the task. Although developing solutions which work is fairly straightforward, making them complete, fault-tolerant, and unobtrusive is not. Hybrid algorithms that combine reference counting and tracing techniques seem to be the most promising candidates. However, many of these rely on heuristics and assumptions about the behavior of long-lived distributed systems to achieve efficiency. In order to tell how accurate these assumptions are, performance evaluations on real-world applications are necessary.

References

[AR98] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[Bev87] David I. Bevan. Distributed garbage collection using reference counting. In Jacobus W. de Bakker, L. Nijman, and Philip C. Treleaven, editors, *PARLE'87 Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 176–187, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[Bis77] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

[BNOW94] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for network objects. Technical Report SRC-116, DEC Systems Research Center, Palo Alto, CA, 1994.

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical report, DEC Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[Dic92] Peter Dickman. Optimising weighted reference counts for scalable fault-tolerant distributed object-support systems. Unpublished, 1992.

[Fuc95] Matthew Fuchs. Garbage collection on an open network. In Henry Baker, editor, *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Concurrent Engineering Research Center, West Virginia University, Morgantown, WV, September 1995. Springer-Verlag.

[GF93] Aloke Gupta and W.K. Fuchs. Garbage collection in a distributed object-oriented system. *IEEE Transactions on Knowledge and Data Engineering*, 5(2), April 1993.

[HK82] Paul R. Hudak and R. M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 168–178, Pittsburgh, PA, August 1982. ACM Press.

[HMMM97] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications – Twelth Annual Conference*, volume 32(10) of *ACM SIGPLAN Notices*, Atlanta, GA, October 1997. ACM Press.

[Hug85] R. John M. Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 256—272, Nancy, France, September 1985. Springer-Verlag.

[JJ92] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, Department of Computer Science (DIKU), University of Copenhagen, September 1992. Springer-Verlag.

[JL92] Richard E. Jones and Rafael D. Lins. Cyclic weighted reference counting without delay. Technical Report 28–92, Computing Laboratory, The University of Kent at Canterbury, December 1992.

[Jon96] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996. With a chapter on Distributed garbage collection by R. Lins.

[LC97] Sylvain R.Y. Louboutin and Vinny Cahill. Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In *Proceedings of the 17th International Conference on Distributed Computing Systems*. IEEE Press, 1997.

[LM86] C.-W. Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, pages 343–350, Cambridge, MA, August 1986. ACM Press.

[LQP92] Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In *Conference Record of the Ninteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 39–50. ACM Press, January 1992.

[MA84] Khayri A. Mohamed-Ali. Object-oriented storage management and garbage collection in distributed processing systems. Academic Dissertation, Royal Institute of Technology, Dept. of Computer Systems, Stockholm, Sweden, December 1984.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[MKI+95] Munemori Maeda, Hiroki Konaka, Yutaka Ishikawa, Takashi Tomokiyo, Atsushi Hori, and Jorg Nolte. On-the-fly global garbage collection based on partly mark-sweep. In Henry Baker, editor, *Proceedings of the International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Tsukuba Research Center, Real World Computing Partnership, Tsukuba, Japan, September 1995. Springer-Verlag.

[ML95] Umesh Maheshwari and Barbara Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 239–248, 1995.

[ML97] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by back tracing. In *Proceedings of PODC'97 Principles of Distributed Computing*, 1997.

[Piq91] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts et al., editors, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1991.

[PS95] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), September 1995.

[Rod98] Helena C.C.D. Rodrigues. *Cyclic Distributed Garbge Collection*. PhD thesis, Computing Laboratory, The University of Kent at Canterbury, 1998.

[Rov85] P. Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox PARC, Palo Alto, CA, 1985.

[RRR97] Gustavo Rodriguez-Rivera and Vince Russo. Cyclic distributed garbage collection without global synchronization in CORBA. In Peter Dickman and Paul R. Wilson, editors, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapports de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, November 1992.

[Sha91] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage collection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, Pisa, September 1991.

[Ves87] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, University of Washington, Seattle, WA, 1987.

[Wei63] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[Wei90] P. Weis. The CAML reference manual, version 2.6.1. Technical Report 121, INRIA-Rocquencourt, 1990.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St. Malo, France, September 1992. Springer-Verlag.

[WW87] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In Jacobus W. de Bakker, L. Nijman, and Philip C. Treleaven, editors, *PARLE'87 Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 432–443, Eindhoven, The Netherlands, June 1987. Springer-Verlag.