# Starting with Termination: A Methodology for Building Distributed Garbage Collection Algorithms

## A submission for Regular Presentation at PODC 2000

Stephen M. Blackburn,[1] Richard L. Hudson,[2] Ron Morrison,[3] J. Eliot B. Moss,[1*]

David S. Munro,[4] and John Zigman[5]

[1]Department of Computer Science, University of Massachusetts,
Amherst, MA 01003−4610, U.S.A.
Email: {steveb, moss}@cs.umass.edu; [*](Author to contact)

[2]Intel Corporation, Microprocessor Research Laboratory,

SC12−303, 2200 Mission College Boulevard, Santa Clara, CA 95052−8119, U.S.A.

Email: rick.hudson@intel.com

[3]School of Computer Science,
University of St Andrews, North Haugh, St Andrews, Fife, KY16 9SS, Scotland
Email: ron@dcs.st−and.ac.uk

[4]Department of Computer Science, University of Adelaide,

Adelaide, S.A. 5005, Australia, Email: dave@cs.adelaide.edu.au

[5]Department of Computer Science, Australian National University,

Canberra, ACT 0200, Australia

Email: john.zigman@cs.anu.edu.au

**Abstract**

We propose an effective methodology in which a distributed garbage collector may be derived from a distributed termination algorithm and a centralized garbage collector in a manner that preserves interesting properties of the original collector, such as completeness. To illustrate our technique we show how two distributed termination algorithms, credit recovery and task balancing, may be suitably described; and then map four centralized garbage collectors—reference counting, mark/scan, a generational scheme, and the Mature Object Space collector (MOS)—onto this description. The advantage of our approach is that, by separating the issues of distribution and collection, we alleviate the difficulty of inventing, understanding, and comparing distributed garbage collection techniques.

## 1 Introduction

The problem of distributed garbage collection (DGC) has proved a fertile ground for research in computer science with many impressive algorithms proposed in the literature (see [JL96] for an overview). A major characteristic of all of these proposals is that it has proved difficult to understand and believe in their correctness, and even more difficult to set them in relation to one another. There are several reasons for this. Firstly, the problem is non−trivial involving the co−ordination of asynchronous tasks where both the network and the sites of the computation may contain information relevant to the co−ordination. Secondly, different model assumptions have been used in the various collectors thereby yielding a large number of subtly different algorithms. Not only does each set of model assumptions seem to have a number of solutions but the number of different models appears to be unbounded. Finally, the algorithms have all been published separately and little attempt has been made to classify them. Here we address this problem of complexity by developing a methodology for deriving DGC schemes starting from a distributed termination algorithm (DTA) and a centralized garbage collector.

Tel and Mattern [TM90] have shown that at least one distributed termination algorithm may be derived from any distributed garbage collector. Here we propose the reverse mapping, which combines any known distributed termination algorithm with a centralized garbage collector to produce a distributed garbage collector. The constraint on such a mapping is that it should preserve the interesting properties of the centralized garbage collector. These interesting properties may be defined in terms of the objects that are

eventually collected. For example, if the centralized garbage collector is not complete, as in reference counting (RC), then the resultant distributed collector will not be complete either [LM86].

The advantage of our approach is the separation of the issues concerned with distribution and the issues concerned with garbage collection. The separation eases understanding and correctness arguments for the resultant DGC algorithms. Building on the vast literature of distributed termination algorithm proofs should simplify correctness arguments of the constructed garbage collector. Furthermore the separation also aids the comparison of distributed garbage collection algorithms since it becomes obvious when two such algorithms share the same distributed termination or garbage collection technique.

Our methodology is as follows:

- Select or derive a distributed termination algorithm that is proven correct. The DTA should be described in terms of *jobs*, *tasks* and a condition we call (*job*) *termination*, see Section 2.2.
- Select and prove safety, and maybe some other properties, of the centralized garbage collector.
- Define an object reclamation mapping,[1] from the centralized garbage collector to the distributed termination algorithm, in terms of jobs, tasks, and job termination, all of which we define in Section 2.
- Prove that job termination is equivalent to the eventual reclamation of objects.

In the following section we present a model of distributed computation, defining jobs, tasks, and job termination. This is followed by a description of two example distributed termination algorithms to give a sense of what they are like in terms of these concepts. The first is a particularly simple one by Mattern [Mat89] called *credit recovery* (CR). The second is *task balancing* (TB) [HMM+98], which we developed as part of our work on garbage collection. Using the above, we show how to perform the mappings from some common garbage collection techniques onto jobs, tasks, and job termination. The techniques chosen to illustrate these mapping are, in ascending order of difficulty: reference counting [LM86, WW87, Bev89], mark−sweep [Ste75, Dij78, Ben84], generational schemes, and the Mature Object Space collector (MOS) [HMM+97]. Finally, for all of these algorithms we show that job termination is equivalent to eventual reclamation of objects.

Our conclusion is that deriving distributed garbage collectors in the above manner separates the issues of collection and distributed termination. In doing so it should be easier to: understand the algorithms; generate correctness proofs; and compare different distributed garbage collectors.

# 2 Distributed Termination

We now proceed with a concise description of the distributed system model that is the setting of our work, and state the distributed termination (DT) problem.

## 2.1 Distributed System Model

Our system model is fairly simple. It consists of a number of *sites*, where computation proceeds asynchronously at each site. Sites communicate (only) by sending *messages* to one another. We assume that sites do not fail, i.e., they do not crash, stop, or act improperly, and that each message sent arrives at some later time, exactly once and uncorrupted. If a given algorithm requires additional properties, such as ordered message delivery, we will state the requirement explicitly.

## 2.2 The Distributed Termination Problem

The distributed termination problem is stated in terms of a *job* consisting of a number of dynamically spawned *tasks*. While a job is distributed, each of its tasks runs at a single site. In particular, the following actions/events define the notion of a distributed job:

[1] Although we define this mapping in terms of the reclamation of objects, certain collectors may operate at a coarser granularity of partitions or clusters of objects. However, the overall effect is the same in that the same objects will be re−claimed in both the centralized and distributed versions of the garbage collector.

- Any site can create a new job *j*. Initially a new job consists of a single task running at the creating site.
- Any running task of a job *j* may *spawn* (create) additional tasks of *j*. A new task may run on the same site as its creating task, or it may run on some other site, i.e., be created to run on a site different from its creator's. One may think of this as sending a task from one site to another. The possibility of such tasks "in flight" between sites contributes substantially to the difficulty of the problem.
- A task may *complete*. This is something the task does on its own ("spontaneously").
- When all of the tasks of job *j* are complete, *j* is said to be *terminated*, written *terminated*(*j*). Note that once *terminated*(*j*) is true it remains so.
- The goal of distributed termination is to determine when *terminated*(*j*) becomes true, for any given job *j*.

Distributed termination is a global property. A local property that is sometimes useful in understanding DT is *idleness*. A site *S* is said to be *idle* for job *j*, written $idle_S(j)$, if all tasks of *j* at *S* are complete. The fact that *idle* can become true and later be falsified (by the arrival of a task created by another site) is part of what makes DT a tricky problem. Note also that it is possible for all sites to be idle yet for there to be tasks in messages sent but not yet received.

### 2.3 Mattern's Credit Recovery DTA

Mattern [Mat87] described a particularly simple DTA called the *credit recovery* (CR) algorithm. In this algorithm, each task carries a certain amount of *credit*. When a site becomes idle, it sends a message to a co−ordinating site informing it of the credit "recovered" by that site, viz., the sum of the credits of the tasks that have completed at the site (since the last time it sent credit to the co−ordinator). When all the credit is recovered, then all the tasks must be done. Here are the details:

- Each task *t* has two associated pieces of information: *job*(*t*), the job of which *t* is a part; and *credit*(*t*), the task's credit, which it turns out will be $2^{-i}$, for *i* a non−negative integer. Each job *j* has an associated piece of information *home*(*j*), the job's home site.
- When we create a new job *j* on site *H*, then *home*(*j*) = *H* and *j* begins with a single new task *t* such that *job*(*t*) = *j*, and *credit*(*t*) = 1. Site *H* also initializes *recovered*(*j*) ← 0. Each site *S* will also maintain $done_S(j)$, assumed to be 0 until it is explicitly set.
- When a task *t* spawns a new task *t'*, then *job(t')* = *job*(*t*), and both *credit*(*t*) and *credit(t')* are set to *c*/2, where *c* = *credit*(*t*) before *t* spawned the new task. That is, each task receives half of the original task's credit, thus preserving the total credit in the system.
- When task *t* completes at site *S*, *S* sets $done_S(j) \leftarrow done_S(j) + credit(t)$.
- If job *j* becomes idle at site *S* ($idle_S(j)$ goes from false to true) then *S* sends to *home*(*j*) a message $recover(done_S(j), j)$ and then sets $done_S(j) \leftarrow 0$.
- When *H*, the home of *j*, receives a message *recover*(*c*, *j*), it sets *recovered*(*j*) ← *recovered*(*j*) + *c*, and if the result is 1 it declares *terminated*(*j*).

It is easy to show that the sum of the credit for any given job is 1 at all times, since spawning tasks preserves the total credit. Thus, once the recovered credit is 1, all tasks must be done and thus the job is terminated. Mattern showed that CR is optimal in terms of the number of messages it sends to detect termination, i.e., one message per each time a site becomes idle. Further, one can trade off the number of messages versus promptness of termination detection by having sites wait a while after becoming idle before sending a CR message, in case they receive additional tasks in the interim.

Credit Recovery is remarkably simple and easy to understand. Its main drawback is that it relies on arbitrary precision binary numbers. In some work (e.g., [Dic91]), partial solutions to this such as wiegthed reference counting have been proposed where have used integer credits, amounting to multiplying Mattern's credits by an agreed upon large integer.

### 2.4 The Task Balancing DTA

In building a distributed garbage collection algorithm called DMOS (Distributed Mature Object Space) [HMM+97], we designed a DTA similar to Mattern's (and some others) which we called our "pointer tracking algorithm". However, that describes an application of it, where in fact one could use any DTA. We have coined the name *task balancing* (TB) for this new DTA.

The TB algorithm maintains counts of (a) the number of tasks of each job sent by each site to each other site, and (b) the number of tasks received by and completed at each site. Specifically, site $S$ maintains for each site $T$ and job $j$, the value $count_S(j, T)$. By convention, we define $count_S(j, S)$ to be the negative of the number of tasks of job $j$ that site $S$ has received (or spawned locally) and that have completed. If a given count has never been set, it is treated as 0. Here are the actions taken on various events:

- When site $H$ (for home) creates a new job $j$, $count_H(j, H) \leftarrow 0$.
- When site $S$ sends a task of job $j$ to site $T$: $count_S(j, T) \leftarrow count_S(j, T) + 1$.
- When site $S$ receives a task of job $j$: no special action.
- When a task of job $j$ completes at site $S$: $count_S(j, S) \leftarrow count_S(j, S) - 1$.
- When job $j$ becomes idle at site $S$: send to $home(j)$ the message $update(j, C)$ where $C = \{<T, count_S(j, T)> \mid count_S(j, T) \neq 0\}$, and then for all sites $T$: $count_S(j, T) \leftarrow 0$. The *update* messages from a given site must be processed in the order they were sent, i.e., we need ordered message delivery for them (but not for messages that convey tasks from site to site).
- When $H$, the home of $j$, receives $update(j, C)$, then for all $<T, n> \in C$ do $count_H(j, T) \leftarrow count_H(j, T) + n$. If, after these updates, $count_H(j, T) = 0$ for all $T$, then declare $terminated(j)$.

The essence of this algorithm is that we track, in a distributed way, the number of tasks *sent* to each site, and (separately) the number of tasks *received and completed* at each site. The home site puts these counts together, and when the numbers of tasks balances at every site, the job has terminated.

We omit the proof here (see [HMM+97] for a brief correctness argument) since it is not relevant to the discussion at hand—indeed, our claim is that one can use *any* DTA. In developing TB we found several variations on it that do *not* work. It is not adequate simply to keep total counts of tasks spawned at and completed at each site. Likewise, it does not work to use counts based on the sites that *send* the tasks as opposed to the sites that *receive* tasks.

## 3 Mapping Between Garbage Collection and Distributed Termination

In our methodology, to build a distributed garbage collector, one takes a non−distributed GC algorithm and develops a correspondence between aspects of the GC and the DT problem. One then uses *any* DTA to solve the DT problem, and thus arrives at a DGC algorithm.

Depending on the characteristics of the GC algorithm, this mapping may be more (or less) complex. There are two considerations in particular that one must handle. First, a distributed algorithm cannot instantaneously and atomically update global shared variables. Thus one must frame the problem to remove shared variables. A particularly helpful way to do this is to *distribute* the state of a shared variable across sites so as to be captured by the DTA. We will give examples of this below.

The second consideration is that in a distributed system, work, either of the program or of the collector, proceeds concurrently and asynchronously at different sites. (Whether one employs concurrency *within* a site is a separate issue.) This is essentially an additional consequence of the state partitioning of the previous paragraph: other sites may change their part of the partitioned state separately and asynchronously from any particular site's part of the state.

### 3.1 Mapping Between a GC and a DTA

To get to details, one establishes a mapping between a GC and a DT algorithm as follows:

- One determines what in the GC constitutes a DT *job* and a DT *task* within a job.

- Based on that, one must determine what actions or events in the GC constitute creation of jobs and tasks and the completion of tasks. Further, one must guarantee that tasks can be spawned only by other tasks, i.e., that they cannot be created spontaneously, e.g., at an idle site.
- Messages that create tasks at remote sites must contain adequate information so that the sending and receiving sites always agree that the message conveys a task. Further, the sending and receiving sites must know the job with which the task is associated and any other information that may be required by the DT algorithm (e.g., the home site of the job).

These requirements establish the DT problem inherent in the mapped GC algorithm. However, we desire further connection between termination and garbage collection:

- Termination of a job should correspond to detecting that a set of objects is unreachable and can be reclaimed. The nature of that set varies according to the GC algorithm.
- Naturally, the DGC should reclaim objects only if they are identified as garbage by the termination of a job. This guarantees that the DGC is *safe*, i.e., that it does not reclaim reachable objects, provided that the original GC is safe.

Given the correspondence, we see that the resulting DGC finds the same possible sets of unreachable objects as the GC, and thus preserves interesting *GC properties*, in addition to preserving safety. For example, since the Mature Object Space (MOS) collection algorithm [HM92] is *complete* (eventually reclaims each garbage object) and (coarse−grained) *incremental*, a distributed version, DMOS, built by our methodology also exhibits completeness and coarse−grained incrementality.

# 4 Example Mappings

We now present four mappings of GC algorithms to DGC algorithms as examples of applying our methodology. We work from simpler to more complex, covering reference counting, (global) mark−sweep (which is essentially the same as copying), generational GC, and Mature Object Space, in that order. For general and thorough background on GC algorithms, and a large number of examples of DGC algorithms as well, we refer the reader to Jones and Lins [JL96].

In all our mappings each object will remain at the site that created it, but the user program may pass references from site to site freely.

## 4.1 Reference Counting

In Reference Counting (RC) one keeps track of the number of references to each object. If and when the number of references to a given object drops to 0, we know it is unreachable and can be reclaimed. RC is clearly safe; it is not complete in that it cannot reclaim cycles of garbage (there are many extensions and variants that attempt to solve that shortcoming). RC is clearly fine−grained incremental: it can recover individual objects as soon as it detects that they are garbage.

One way, perhaps the most obvious one, to map RC to a distributed system is to send reference count increment and decrement messages to the site holding an object. This can lead to complex distributed algorithms because of the asynchrony of message passing. We claim that it is much simpler and clearer to map RC directly onto DT, by considering the termination condition that is the essence of RC: *an object is reclaimable when there are no more references to it anywhere in the system*. This maps straightforwardly to DT, as follows:

- Each object corresponds to a job, and each reference to the object constitutes a task of the job.
- Therefore, creating an object creates a job, with the original reference to the new object being the first task. Any time we copy a reference, we spawn a new task.
- Likewise, when we send a reference in a message to another site, we create a task at the remote site.
- Thus, in order to spawn a new task (copy a reference), we must have a “running task” (an existing reference), so the mapping to DT is sound. This reveals the assumption that references cannot be fabricated by the application.

- Discarding a reference corresponds to task completion.
- Job termination is equivalent to absence of any references to the target object, implying the object may be reclaimed (safety).

What we called the "Pointer Tracking Algorithm" in the original DMOS algorithm [HMM+97] constitutes this mapping of RC, using the TB termination algorithm. We claim that TB is a good algorithm for RC in that it does not send messages for each reference creation or deletion, but only upon idleness (no running tasks, i.e., no references to a particular object) at a particular site.

### 4.2 Global Mark−Sweep

Non−distributed Mark−Sweep (MS) collection proceeds as follows. Every object has an associated *mark*, which can have the values *marked* or *unmarked*, i.e., a single bit suffices. Initially all the objects are *unmarked*. Then one marks each object referred to by a root reference. One then propagates marks: if a marked object refers to an unmarked one, then one marks the unmarked object. One continues until every object referred to by either a root or a reference in a marked object is marked. Unmarked objects are unreachable and reclaimed. Global mark−sweep is safe and complete, but not incremental.

Note that MS and copying GC are abstractly identical. One need only interpret *marked* as "copied and forwarded", and *unmarked* as "not (yet) copied". Put another way, they will find the same reachable and unreachable objects sets and reclaim the same objects. They do differ in performance of reclaiming the unreachable objects, with MS processing each unreachable object and copying reclaiming all the space in a single operation. We chose MS to map primarily because it is easier to describe.

The basic thrust of our mapping in this case is that each invocation of GC is a job, and handling the marking of an object is a task. However, unlike the non−distributed "stop−the−world" MS collector, a distributed MS collector will interleave user program work and MS work, which raises additional issues we need to resolve:

- *New objects*: If the user program creates an object, how should we treat it? Our solution is for new objects to start *marked*. Thus they will not be reclaimed in the current GC invocation, but may be in a future one.
- *Accessing objects*: As the user program runs, it loads heap references from existing objects into registers and the program stack, and may store them into global variables too. How should we treat those? Our solution is to guarantee that roots always refer to marked objects. Therefore, if the mutator loads a reference to an unmarked object, it marks the object and (logically at least) spawns a mark task for the object. (See Jones and Lins [JL96] for a variety of approaches to concurrent MS, and Pirinen [Pir98] for a taxonomy.) Seeing a reference to an unmarked object, given that we must have loaded it from a marked object, implies that there is a "running task" for the marked object. Hence this task spawning is justified. Notice how the DT notions of tasks and rules for spawning them clarify what we need to show!
- *Updating objects*: Similarly, as the user program runs, it may update existing objects' fields to refer to other objects. Is this a problem? Given our approach to accessing objects, the objects involved in an update must all be marked, so we create only *marked*→*marked* edges, and thus we do not need to spawn any tasks.

To complete the mapping, we observe that a job begins by marking from the roots at every site, i.e., it spawns a "root mark" task for each site. A root mark task spawns an "object mark" task for each root reference. An object mark task for a local object checks the object's mark bit. If the bit is *marked*, the task completes. If the bit is *unmarked*, the task marks the objects, spawns object mark tasks for each object referred to by the newly marked object, and then completes. An object mark task for a remote reference spawns a remote object mark task and completes.

From this description it should be clear that job termination implies completion of marking, and the invariants involved guarantee that termination implies that all objects referred to by roots are marked, and all objects referred to by marked objects are marked. Thus, unmarked objects are unreachable, and safely reclaimable.

It is interesting to review the mapping in the light of tri−color marking schemes (see Wilson [Wil92] or Jones and Lins [JL96] for details). A *black* object corresponds to a completed task, a *gray* object to one (or perhaps more) running tasks, and a *white* object is simply not yet processed (unmarked). Our mapping maintains the invariant of no *black→white* edges, and job termination is equivalent to there being no *gray* objects.

What is striking about the MS mapping is that while remaining simple, it is rather different from the RC mapping. A point we make is that the right mapping makes constructing the DGC easy, but the mappings may involve creativity and are not necessarily produced by a "turn the crank" procedure.

#### 4.2.1 Hughes' DGC Algorithm

Hughes [Hughes85] designed a DGC consisting of an extension to MS plus a DTA. Given that he was clear about the separation of the GC algorithm and the DTA (unusual for papers on DGC), it is easy to consider replacing the DTA he used with another one. In fact, since the DTA he used requires a message from every site, TB would appear to offer better performance.

Hughes' extension to MS is to allow *multiple collections at once*, each identified by a "timestamp", such that later collections have later timestamps. As opposed to a single mark bit associated with each object, he associates a timestamp with each object: if the timestamp of an object is *t* or greater, then the object was reachable during the collection associated with *t*.

Here is a mapping for Hughes' algorithm:

- Each GC is a job.
- Each reference from an object with timestamp *t2* to an object with an older timestamp *t1* (i.e., $t1 < t2$) is a task for every GC with timestamp *t* such that $t1 < t \leq t2$.

This mapping is sound because increasing an object's timestamp from *t1* to *t2*, which spawns tasks for jobs with timestamps greater than *t1* and less than or equal to *t2*, occurs only when there are running tasks for those jobs at the object from which we are propagating the timestamp.

### 4.3 Generational Mark−Sweep

A generational collector partitions objects into two generations, called *old* and *young*. (This is readily generalized to more than two generations.) It can perform two kinds of collections, *full*, which processes all objects just like MS, and *scavenge*, which separates the young objects into reachable and unreachable assuming that all old objects are reachable. To avoid scanning the old objects, generational collectors usually include a *write−barrier* mechanism, to detect and record all references from old to young objects. The set of such references is called the *remembered set*.

A generational MS scavenge is essentially a limited MS collection, and proceeds as follows. Initially all young objects are unmarked, and all old objects are treated as marked. One begins by marking all young objects referred to by roots or from the remembered set. One then propagates marks as in MS, but only from young objects to young objects. Once this propagation finishes, the unmarked young objects are unreachable and may be reclaimed.

As with global MS, we need to take care concerning newly created objects and mutator accesses and updates, but the same solutions that worked in the previous section work here.

The mapping is as for MS, with the reclaimed object set being the unmarked young objects. What is interesting here is that we derive a new DGC quite simply, one we do not believe has been described before.

### 4.4 Mature Object Space

Mature Object Space (MOS) [HM92] generalizes generational collection. It consists of a number of *trains*, which partition the objects. (The train terminology is historical and has to do with a metaphor used to describe the algorithm. Also, MOS typically includes one or more young generations as well, but that is inessential here.) Trains are (totally) ordered, by (abstract) age, so we speak of *younger* and *older* trains. Trains further partition their objects into *cars*, where each car contains a bounded volume of objects. We

speak of objects being *associated* with cars, and of collection as *reassociating* them. The cars of a train are ordered, so we speak of younger and older cars of a train. There are always at least two trains.

Each collection processes a single car, the oldest car *C* of the oldest train *T*, using these rules, applied in order:

- If the remembered set of *T* is empty (meaning there are no references from outside of *T* into *T*), reclaim *T* in its entirety. We call this *train reclamation*.
- For each object in *C* that is root reachable, reassociate (move) the object to a younger train.
- For each object in *C* reachable from a younger train, reassociate (move) the object to that younger train.
- For each object in *C* reachable from *T* but outside *C*, reassociate (move) the object to the youngest car of *T*, possibly a new car.
- Remaining objects of *C* are unreachable and may be reclaimed immediately.

To guarantee progress, each car should also remember at least one of its objects reachable from outside the train. It should remember from where these objects were reached (i.e., a root or a specific train), and it should treat these objects as reachable from that source. See Seligman and Grarup [SG95] for further details.

This algorithm is safe (reclaims only unreachable objects), coarse−grained incremental (processes only one car at a time), and complete (eventually reclaims each garbage object). But how do we distribute it? Here are considerations that arise and how we handle them:

First, we will assume that *trains may be distributed* but *cars are <u>not</u> distributed*. The idea is for each car collection to be a local operation requiring no immediate co−ordination with other sites. But in a distributed system, it may not be easy to determine which train is the oldest. Likewise, it might be difficult to determine which car is the oldest, and in any case we would prefer to allow collection to proceed with any car, and with multiple cars concurrently at different sites.

To reclaim a train, we need to know that there are no references into the train from outside of it. This general principle works regardless of whether the train is the oldest. What are the relevant special properties of the oldest train, which we may need to carry over to the distributed setting? One is that we do not allocate objects into the oldest train. Another is that collection will not promote objects into it. Here is how we address each of these properties:

- To control allocation into trains, we introduce the notion of a train being *open for allocation* (or promotion of root−reachable objects) or being *closed* to it. Once closed, a train may not be re−opened. We require that there always be at least one open train younger than any closed train, into which one may promote root−reachable objects. Obviously we reclaim only closed trains.
- To relate promotion and train reclamation, we defer reclamation of a train until there is no possibility that an object will be promoted into the train.

Another property of the original algorithm that we will preserve is that objects never reassociate from younger to older trains.

Putting all of this together leads to the follow updated rules for collecting car *C* of closed train *T*:

- If the remembered set of *T* is empty, and there are no references from *T* to older trains (implying no objects can be reassociated into *T*), reclaim *T* in its entirety.
- For each object in *C* that is root reachable, reassociate (move) the object to a younger open train.
- For each object in *C* reachable from a younger train, reassociate (move) the object to that younger train.
- For each object in *C* reachable from *T* but outside *C*, reassociate (move) the object to another car of *T*.
- Remaining objects of *C* are unreachable and may be reclaimed immediately.

We now consider the mapping of this updated MOS algorithm to DT. A job corresponds to a train. A task corresponds to a reference that might keep the train reachable—either a reference from any other train into *T*, or a reference from *T* to any older train. We map the original task of the job onto the openness of the train: that task completes when the train is closed.

In developing our first distributed MOS algorithms (DMOS [HMM+97]) we did not use the GC+DTA methodology proposed here, and at first overlooked the issue of promotion into *T*. We ended up designing a complex and buggy DGC. With the right mapping, though, we obtained this much simpler algorithm that we more readily see is correct.

To keep track of tasks, we track cross−site references and the trains involved. Creating a reference from train *T* to train *U* creates a task for train *U* (always), and a task for train *T* if *T* is younger than *U*. As references are copied, stored, overwritten, and sent in messages, and as objects are promoted from train to train, we create suitable tasks, or tasks complete.

When the job for train *T* terminates, we can reclaim all cars of *T* (at all sites).

Here is a brief argument that the resulting algorithm is complete. The completeness of MOS hinges on (a) not promoting objects to trains younger than ones from which they can be reached (considering roots to be "infinitely young"), (b) the oldest train is eventually collected, and (c) one cannot create an unbounded number of trains of age between that of two other trains. It is easy to guarantee (c) by using integer numbers plus bounded−size site identifiers to number the trains. We obtain (b) through the idea of a closed train (no new objects can go into the oldest closed train) and that the oldest train cannot have objects reassociated into it from outside, plus requiring that each car eventually be collected, which will force progress in evacuating or reclaiming the oldest closed train. Thus the changes we made to the original MOS rules do not damage the facts on which the completeness of MOS relies, so the resulting algorithm is still complete.

## 5 Related Work

Tel and Mattern [TM90] explored the relationship between distributed garbage collection and distributed termination in some detail. They observed that at least one distributed termination algorithm could be derived from any distributed garbage collector. Hughes [Hug85] also noted this connection, and described his distributed garbage collection algorithm in terms of a concurrent mark−sweep algorithm and a distributed termination algorithm [Ran83].

To our knowledge few, if any, of the DGC algorithms published since Hughes (Jones and Lins [JL96] list at least twenty algorithms) have been described in similar terms. We have, however, analyzed two of the most recent additions to the literature, DMOS [HMM+97] and Le Fessant's algorithm [FPS98], in terms of their composition from GC and DTA algorithms.

It was in the process of grappling with subtle bugs in the original DMOS algorithm [HMM+97] that we realized that it embodied two distinct DT problems, each of which was being solved through a different DTA. By viewing the DMOS algorithm abstractly in terms of the MOS GC algorithm and two DT problems, the description of the algorithm was greatly simplified and our understanding of its properties greatly enhanced. Moreover, the understandable (and correct) task balancing DTA could then be applied to *both* DT problems.

Le Fessant, et al. [FPS98] describe a distributed mark and sweep algorithm inspired by Hughes' algorithm. Incrementality is enhanced by the use of intra−node partitions and instead of using Rana's algorithm for distributed termination, they use a new, more general DTA, that does not depend on a global clock and instantaneous communication. As with our original DMOS algorithm, a failure by the authors to strongly separate the GC and DTA facets of this algorithm makes a clear understanding of its properties (including correctness) somewhat elusive for the reader.

Schiper and Sandoz [SS94] suggest that DTA and DGC are not equivalent. They establish two properties: *stable* and *strongly stable* and show that termination is a strongly stable property and that garbage detection is merely stable. In effect they are demonstrating that, in an observation of the system states, detection that an object is garbage is only established if the observation is a consistent cut, whereas termination can be established without the cut being consistent. This should not be a surprise since given a cycle of garbage, all of it must be established as garbage before any of the individual objects can be declared as such. So while Schiper and Sandoz are correct in their statement of the non−equivalence of DTA and DGC, it does not means that one cannot be derived from the other. Put another way, what they showed is that

instantaneous and local detection of unreachability is not possible, whereas we are concerned with mapping actual GC algorithms, which are either incomplete (do not detect all garbage) or else require more message passing to detect garbage in at least some cases.

## 6 Conclusions

We have presented a methodology for deriving distributed garbage collection algorithms from a distributed termination algorithm and a centralized garbage collector. The aim is to alleviate the difficulty of inventing, understanding, and comparing distributed garbage collection techniques. We claim that separating the issues of distribution and collection eases the understanding and correctness arguments for the resultant DGC algorithms. To back up our claim we have shown how the proposed methodology may be used to derive a number of distributed garbage collectors. We now invite others to follow this style in presenting their DGC algorithms.

While we state that any DTA may be used in the derivation of a DGC, not all such mappings may be useful or efficient. Tel and Mattern, in discussing the possibility of such a mapping, point out that probe−based solutions to DTA, i.e., ones that visit all sites in the system, are unlikely to be as efficient as ones that restrict their activity to the tasks involved in the distributed computation. Furthermore, they point out that a DTA where one task in a job plays a special role, such as the home task in CR and TB, need not be a drawback unless the special role becomes a bottleneck.

This all leads to a new and better DGC implementation methodology, in which one starts with simple DT and GC algorithms, and then incrementally replaces one or the other as performance or other considerations warrant. Given the implementation complexity of systems supporting DGC, our simplification and this incremental implementation strategy should have much impact on the fielding of this technology in real systems.

There remains another significant challenge in this area of work: simplifying and factoring fault tolerance aspects of distributed object management. Given the range of models of failure, this remains difficult and elusive.

## 7 Acknowledgements

The TB algorithm is used in the DMOS collector [HMM+97]. The work was supported by NSF grants IRI−9632284 and INT−9600216 and by EPSRC grant GR/M74931.

## 8 References

[Ben84] M. Ben−Ari, “Algorithms for On−the−fly Garbage Collection”. ACM Transactions on Programming Languages and Systems, 6 (1984), pp. 333−344.

[Bev89] D. I. Bevan, “An Efficient Reference Counting Solution to the Distributed Garbage Collection Problem”. Parallel Computing, 9 (1989), pp. 179−192.

[Dic91] P. Dickman, “Distributed object managament in a non−small graph of autonomous networks with few failures”. Ph.D. thesis, University of Cambridge, United Kingdom, 1991.

[Dij78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, & E. F. M. Steffens “On−the−fly Garbage Collection: An Exercise in Cooperation”. Communications of the ACM 21 (1978), pp. 966−975.

[Dij80] E. W. Dijkstra “Termination Detection for Diffusing Computations. Information Processing Letters 11−1 (1980), pp. 1−4.

[Dij78] E. W. Dijkstra “Derivation of a Termination Detection Algorithm for Distributed Computations. Information Processing Letters 16 (1983), pp 217−219.

[FPS98] F. Le Fessant, I. Piumarta, & M. Shapiro, “An implementation of complete, asynchronous, distributed garbage collection”, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’98) (Montreal, Canada), ACM SIGPLAN Notices 33 (1998).

[HM92] R. L. Hudson & J. E. B. Moss, “Incremental Collection of Mature Objects”. International Workshop on Memory Management (Y. Bakkers & J. Cohen, eds.) LNCS 637, Springer−Verlag (1992), pp. 388−403.

[HMM+97] R. L. Hudson, R. Morrison, J. E. B. Moss, & D. S. Munro, “Garbage Collecting the World: One Car at a Time”. Proc. of the 1997 ACM Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA’97), ACM SIGPLAN Notices 32, 10 (1997).

[HMM+98] R. L. Hudson, R. Morrison, J. E. B. Moss, & D. S. Munro, “Where Have all the Pointers Gone”. Proceedings of the 1998 Australasian Computer Science Conference.

[Hug85] R. J. M. Hughes, “A distributed garbage collection algorithm”. Proc. of the 1985 Conf. on Func. Prog. and Computer Architecture, LNCS 201, Springer−Verlag (1985), pp. 256−272.

[JL96] R. Jones & R. Lins, **Garbage Collection: Algorithms for Automatic Dynamic Memory Management**, Wiley, 1996.

[LM86] C−W. Lermen & D. Maurer, “A Protocol for Distributed Reference Counting”. ACM Conference on Lisp and Functional Programming, Cambridge, Mass (1986), pp. 343−354.

[Mat89] F. Mattern, “Global Quiescence Detection Based on Credit Distribution and Recovery”. Information Processing Letters, 30 (1989), pp. 195−200.

[Pir98] P. P. Pirinen, “Barrier Techniques for Incremental Tracing”. Proceedings of the International Symposium on Memory Management (1998), pp. 20−25.

[Ran83] S. P. Rana, “A Distributed Solution to the Distributed Termination Problem”. Information Processing Letters, 17 (July1983), pp. 43−47.

[SG95] J. Seligmann & S. Grarup, “Incremental Mature Garbage Collection using the Train Algorithm”. Proc. of 1995 Eur. Conf. on Object−Oriented Prog. (ECOOP ’95), O. Nierstras, ed., LNCS, Springer−Verlag (1995).

[SS94] A. Schniper & A. Sandoz, “Strong Stable Properties of Distributed Systems”. Distributed Computing 8 (1994), pp. 93−103.

[Ste75] G. L. Steele, “Multiprocessing Compacting Garbage Collection”. CACM 18 (1975), pp. 495−508.

[TM90] G. Tel & F. Mattern, “The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes”. ACM Transactions on Programming Languages and Systems, 15, 1 (1993).

[WW87] P. Watson & I. Watson, “An Efficient Garbage Collection Scheme for Parallel Computer Architectures”. Proc. of Parallel Architectures and Languages, Europe, J. W. De Bakker, A. J. Nijman, P. C. Treleaven, eds., LNCS 259, Spinger−Verlag (1987), pp. 432−443.

[Wil92] P. R. Wilson, "Uniprocessor Garbage Collection Techniques". Proc. of the Int’l Workshop on Memory Management, Y. Bekkers & J. Cohen, eds., LNCS 637, Springer−Verlag (1992).