Practical distributed garbage collection for networks with asynchronous clocks and message delay

Doug Kwan

Dept. of Computer Science
University of Hong Kong
Hong Kong

Francis Chin

Dept. of Computer Science
University of Hong Kong
Hong Kong

Abstract

Distributed garbage collection over a message passage network is discussed in this paper. Traditionally, this can be done by reference counting, which is fast but cannot reclaim cyclic structures or by graph traversal, e.g. mark-and-sweep or time stamping, which is capable of reclaiming cyclic structures but is slow. We propose a combined scheme which is fast in reclaiming acyclic garbage and guaranteed to reclaim cyclic garbage. Our scheme does not rely on synchronized clocks nor zero message delay and is thus practical.

1 Introduction

Garbage collection [2, 10] is a very useful memory management technique. In recent years, much research effort has been put into the area of distributed and parallel garbage collection. In this paper, the problem of distributed garbage collection over a message-passing network is discussed.

Many algorithms for distributed garbage collection have been proposed and studied. One important class is based on *reference counting*[8, 9][1]. Unlike their counterparts in uniprocessor environments, these algorithms have to address the problem of synchronisation, or live objects may be deleted prematurely because their reference counts may be reduced to zero temporarily owing to race conditions. When the reference count of an object reaches zero, that object can be reclaimed immediately and the reference counts of all other objects pointed to by this object can also be reduced subsequently. Therefore the latency of reclamation is low. However, reference counting cannot detect cyclic garbage[8, 9].

[1]In the reference counting scheme, each object has an associated count on the number of pointers (references) at it.

Another class of distributed garbage collection algorithms uses graph traversal to detect live, i.e. reachable objects[5, 7]. The simplest one is a distributed version of the *mark-and-sweep* algorithm[2, 10]. The local graph traversal in each processor is done by local garbage collectors conservatively. Local graph traversal is conservative because all objects reachable from local roots or from non-local objects through interprocessor pointers are considered live even though objects referenced through inter-processor pointers may be garbage. Global liveness information is propagated through inter-processor pointers on top of local garbage collection. If an inter-processor pointer is encountered during a local collection, liveness information is sent to the processor containing the object. Garbage is detected when system has stablized and no global liveness information needs to be propagated.

There are several problems with this kind of algorithms. First, we have to detect stability with respect to global liveness information propagation. No object can be reclaimed unless all liveness information has been propagated. Second, we may have to wait for a long time before the system stablizes. For acyclic garbage, reference counting should perform better under such circumstances.

A way to reduce latency of reclamation is to overlap a number of distributed garbage collection processes. Liveness information of several processes is propagated by a single local collection. The can be done by *pipelining* or *temporal overlapping* [5, 7], i.e. a number of distributed garbage collection processes are carried out simultaneously. Liveness information of a distributed garbage collection instance overrides the liveness information of all unfinished ones started earlier. With this approach, liveness of an object is associated with a *time stamp* [5, 7] corresponding to the latest distributed garbage collection and it is assumed that the object is live by all unfinished instances of distributed garbage collection started earlier. Latency of

0-8186-6555-6/94 $04.00 © 1994 IEEE

reclamation can be high as information may have to be propagated through the whole network.

We propose here a combined algorithm using time stamps and reference counters in order to have the virtues of these two approaches. Acyclic garbage can be reclaimed as soon as possible through reference counting while cyclic garbage is reclaimed by means of time stamping. Our presentation is formal and rigorous and we have proposed solutions to problems which have not been discussed before, especially during the process of establishing inter-processor pointers.

Our approach also has several advantages over previous work in this area. First, our algorithm works with finite but unbounded delay. Previous algorithms either ignore network delay[5] or assume a known upper bound for network delay[7]. Second, our algorithm does not rely on synchronised clocks, a condition which was assumed by most previous work[5, 7]; this complicates our termination detection protocol.

In section 2, models used in our algorithm are defined. The algorithm and its correctness are discussed in section 3 and section 4 concludes this paper.

2 Basic Assumptions

2.1 Network Model

We assume a group of processors communicating with each other over a message passing network with finite but unbounded delay. Messages reach their destinations and are acknowledged eventually in the order they are sent. Processors have clocks which are not synchronised. Every processor has its local objects, which are inaccessible from other processors. Associated with each processor are two computing agents, the *mutator* and the *collector*[4]. For the sake of convenience, when the objects, the mutator and the collector all belong to the same processor, we refer to the object as the mutator's objects or the collector's objects, we also call the mutator the object's mutator and the collector the object's collector. The mutators perform computation in the network. A mutator only has access to those objects reachable from its roots via local pointer dereferencing. If a mutator has to access a remote object, it can only do so through message passing, i.e. by sending a message to the remote object's mutator requesting that the operation be done on its behalf. In addition, a mutator can only receive pointers from other mutators.[2] The *collectors*, based on either a *marking* or *copying*[2, 10] scheme, reclaim objects which are inaccessible from any of local roots nor non-local objects.

[2]Since a collector can see garbage, it can send a message containing a pointer to a garbage object to a mutator and revive the object. Therefore we have to prohibit mutators from receiving pointers from collectors.

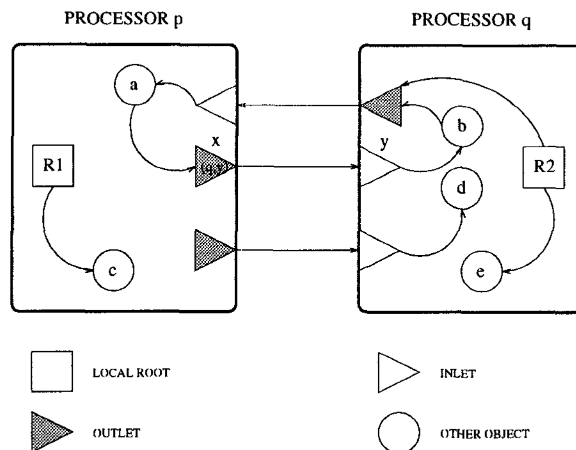

Figure 1: Inlets and Outlets

2.2 Inlets and Outlets

Two kinds of special objects, namely *inlets* and *outlets*, (Fig 1) are used for inter-processor pointer indirection. An *inlet* contains a pointer to a local object which is known to be pointed to by some inter-processor pointers. An inlet y on processor p is identified by a *reference (p,y)*. An *outlet* contains a reference to an inlet. An inter-processor pointer from an object a in processor p to another object b in processor q is represented by a local pointer from a to an outlet x in p with a reference (q, y) where y is the inlet in q pointing at b (Figure 1). Thus inter-processor pointers can be manipulated as if they were local pointers. This indirection technique is common in distributed garbage collection. We say inlet x and outlet y are *matched* if y stores a reference to the x.

2.3 Operations of Mutators and Collections

We classify the operations of the mutators and the collectors into the following categories. Implementations will be described later.

1. Creating New Objects
2. Changing Local Pointers
3. Sending References

4. Receiving References
5. Local Garbage Collections
6. Sending Time Stamps
7. Receiving Time Stamps

These operations model computation and distributed garbage collection in the network.

3 Algorithm

There are two types of garbage, *local garbage* and *distributed garbage*. Local garbage can be reclaimed by local garbage collectors without involving inter-processor pointers. Distributed garbage is garbage involving inter-processor pointers. In particular, it is sufficient that only inlets are considered by the distributed garbage collector. As outlets which are inaccessible from local roots or inlets can be reclaimed by the local garbage collectors, they would not be considered distributed garbage.

3.1 Distributed Reference Counting

We now outline a distributed reference counting scheme capable of reclaiming acyclic distributed garbage similar to the one described in [8]. Under this scheme, every inlet is associated with a reference counter reflecting the number of matching outlets. The reference counter is updated whenever matching outlets are created or destroyed. Normally, as soon as the reference counter of an inlet reaches zero, an inlet can be reclaimed. However in some cases, owing to network delay, an inlet with zero reference count may not be garbage because there may still be some messages containing reference to this inlet. We say an inlet is *in use* if a message containing a reference of the inlet has been sent but not acknowledged. Similarly, an outlet is in use if the particular reference stored in the outlet has been sent but not acknowledged.

Each mutator records references it has sent but not acknowledged. A *message counter* is added to every inlet and outlet for finding out whether an object is in use. The counter is zero when an object is created, incremented when a reference associated with the object is sent and decremented when the message is acknowledged. Thus an object is in use if its message counter is non-zero.

Our distributed reference counting algorithm is as follows. This algorithm is only invoked when an inter-processor pointer is created or removed. During the creation of a new inter-processor pointer, a mutator, say M_A, sends a reference to another mutator, say M_B, and a new outlet for this reference has to be created in M_B's processor if the referenced inlet is not in M_B's processor. The reference counter of the inlet matching this new outlet is incremented. Therefore M_B does not have to reply to M_A and instead it sends an *increment reference message* to the collector of that inlet. This increment reference message contains the reference, M_A and the ID of the original message from M_A. When that inlet's collector receives this message, it increments the reference counter of that inlet and replies to both M_A and M_B. The outlet or the inlet associated with reference in M_A's processor is kept *in use* until the reply from the collector to M_A is received.

When an inter-processor pointer is removed, an outlet is reclaimed and the reference counter of the matching inlet is decremented accordingly. Therefore, the outlet's collector, say C_A, sends a *decrement reference count* message to the the inlet's collector. However, if the outlet is in use, the sending of the decrement reference count message as well as its reclamation will be delayed until it is not in use. Upon receiving the decrement reference message, the collector C_B decreases the reference counter of the inlet. The inlet will be reclaimed if its reference counter reaches zero and it is not in use.

3.2 Time Stamping

As reference counting cannot reclaim cyclic garbage, we augment it with time stamping. The combined algorithm reclaims cyclic garbage and also reclaims acyclic garbage faster than algorithms using time stamping alone. Time stamps are introduced to all inlets, outlets and roots. We denote the time stamp of an object x by $time(x)$. Loosely speaking, $time(x)$ denotes the "time" until which x remains alive and thus $time(x)$ is non-decreasing as time passes. They are propagated from roots to inlets and outlets.

Intuitively, our algorithm works like this. Roots are sources of increasing time stamps which are propagated to inlets and outlets. Thus any live inlet or outlet also has an increasing time stamp. If an inlet or an outlet becomes garbage, it is disconnected from all roots and hence its time stamp would stop increasing. A threshold is found such that time stamps of all live objects are of at least this value. All objects with time stamps less than the threshold must be garbage and hence removed.

Because of asynchronous clocks, one may ask whether it makes sense in propagating time stamps originating from asynchronous sources. It is easier to

understand our algorithm if we view time stamps as version numbers, which are ordered like time stamps. If we view the time stamping algorithm as a collection of concurrent distributed mark-and-sweep algorithms, then $time(x)$ denotes that object x is live during "all" the distributed garbage collection processes with version numbers $\leq time(x)$. Thus if object x receives several time stamps, it makes sense for higher time stamps to override lower ones as the liveness of x in a distributed collection process of higher version number implies that of lower ones.

3.2.1 t_{prop} t_{trans} and t_{min}

In order to determine the threshold for live objects, we have to ensure that all time stamps not greater than the threshold are propagated properly throughout the whole network. Intuitively, every processor p keeps track of two time thresholds, $t_{prop,p}$ and $t_{trans,p}$, where $t_{prop,p}$ is the time threshold that all time stamps are propagated properly within processor p[5] and $t_{trans,p}$ is the time threshold that all time stamps are transmitted properly from processor p to other processors. The time threshold is defined as

$$t_{min} = \min_{p}(\min(t_{prop,p}, t_{trans,p}))$$

As local roots are live when the collection starts, their time stamps are set to the starting time of the local collection. Time stamps are then propagated from local roots or inlets to outlets within a processor during a local collection, after which the time stamp of any outlet y is not less than that of any local root or inlet x from which the outlet is accessible, i.e., $time(y) = \max\{time(x)|y \text{ is reachable from } x\}$. Thus $time(y)$ is at least $time(x)$ but no more than the time threshold for proper time stamp propagation, $t_{prop,p}$.

Invariant 1 *For any inlet or local root x and outlet y of a processor p such that y is accessible from x via intra-processor pointers,* $time(y) \geq \min(t_{prop,p}, time(x))$.

To uphold this invariant, $t_{prop,p}$ has to be adjusted constantly. It is set to the starting time of the local collection, i.e. the time stamp of the roots, after the collection and this is the only time when $t_{prop,p}$ is increased. During the period between two instances of garbage collection, if inlet x receives a higher time stamp and $t_{prop,p}$ is also higher than outlet y's time stamp, we reduce $t_{prop,p}$ to maintain the time threshold property and Invariant 1.

Invariant 2 *For any processor p and local root x,* $time(x) \geq t_{prop,p}$

Time stamps are also propagated from outlets to inlets via inter-processor pointers. We assume that a time stamp message is sent from an outlet to its matching inlets when the outlet's time stamp is increased Owing to network delay, the message does not arrive at its destination immediately. This is similar to the case that time stamps cannot be propagated immediately within a processor. Therefore every processor p also has another time threshold $t_{trans,p}$ for the highest time stamp value it has sent properly to other processors. This means time stamp messages ever sent out by p with values not greater than $t_{trans,p}$ are guaranteed to be received.

Invariant 3 *For any outlet x or a processor p and its inlet y so that x and y are matching,* $time(y) \geq \min(t_{trans,p}, time(x))$.

The propagation of time stamp among processor is a continual process. Unlike propagating time stamps within a processor, generally there does not exist a convenient moment when all time stamps are propagated properly, i.e. all received and acknowledged. Hence the increase of $t_{trans,p}$ cannot be done as easily as that of $t_{prop,p}$ and we must take into the account of the unacknowledged time stamps.

The above invariants are vital to the correctness of our distributed garbage collection algorithm. Let t_{min} denote the global minimum of $t_{prop,p}$ and $t_{trans,p}$ of all processors. Intuitively, any time stamp value not greater than t_{min} are propagated correctly throughout the whole network and time stamps not greater than t_{min} remain unchanged forever. We now claim that the time stamp of any live inlet is at least t_{min}.

Theorem 1 *The time stamp of any live inlet is at least t_{min}, the global minimum of $t_{prop,p}$ and $t_{trans,p}$ for all processor p.*

Owing the space constraint, the proof of the theorem from the three invariants is not presented here. A detail proof is given in [6].

Theorem 1 shows that the invariants guarantee the safety of our algorithm and that live objects would not be erroneously deleted if only those object with time stamps smaller than t_{min} are deleted.

To guarantee the correctness of our distributed garbage collection algorithm, we preserve these invariants all the time during the operations of the mutators and collectors. In section 2.3 we have listed seven operations for the mutators and the collectors, we now the show they do not affect the invariants. [3]

[3] Operations pertaining to distributed reference counting are neglected here for the sake of simplicity of our presentation.

3.2.2 Creating New Objects

New objects are only accessible from the local roots and they do not contain pointers. Therefore all invariants are not affected.

3.2.3 Changing Local Pointers

The set of accessible outlets from an inlet may be changed during local computation but this does not affect Invariant 1 because any outlet involved must be accessible from the local roots and hence its time stamps is at least $t_{prop,p}$ (Invariant 2). As local computation does not include increasing time stamps of outlets, Invariant 3 is not affected.

3.2.4 Sending References

When a *local* reference is sent, the mutator creates a new inlet if no inlet for the referenced object exits. The time stamp of the inlet (new or old) is set to the current time without violating Invariant 1 and is also sent to the other mutator together with the reference.

When a *remote* reference is sent, the minimum of $t_{trans,p}$ and the time stamp of the outlet containing the reference is sent along with the reference as a lower bound of the inlet's time stamp.

3.2.5 Receiving References

On receiving a *local* reference, the corresponding inlet is found $t_{prop,p}$ is reduced to no more than the time stamp of the inlet as we do not know if that higher time stamps than this are properly propagated. After adjusting $t_{prop,p}$, we also increase the time stamp of the inlet to the current time because the inlet is live now. This increase has no effect on Invariant 1.

When a *remote* reference is received and there is no outlet containing this reference, a new outlet is created with its time stamp set to the current time. $t_{trans,p}$ is reduced to the time stamp received together with the reference, which is a lower bound of the matching inlet's time stamp[4].

If an outlet x containing the reference exists, we increase $time(x)$ to lower bound of the matching inlet's time stamp without sending a message to its matching inlet y because $time(y)$ is guaranteed to be not smaller than the lower bound. Since the outlet is now accessible (previously may be inaccessible) from the local roots, we reduce $t_{prop,p}$ to $\min(t_{prop,p}, time(x))$ to keep Invariant 1.

[4]This is true even if the inlet is local to p.

3.2.6 Local Garbage Collection

Besides reclaiming local garbage, local garbage collection also propagates time stamp within a processor. [5, 7]. It does so as follows. (1) Time stamps of local roots are set to the starting time of local collection. (2) Time stamps of inlets in use (if lower) are set to the starting time of the local collection. (3) Time stamps of local roots and inlets are propagated to their descendants by local graph traversal in descending order of their time stamps. When an outlet is first seen during a local collection, its time stamp is set to that of the local root or inlet from which it is reached if its time stamp is lower. (4) The time stamps of all outlets in use are also increased to the starting time of the local collection if they are lower. (5) When the graph traversal finishes, the time stamp of an outlet in not smaller than the maximum time stamp of all local roots or inlets from which it is accessible. We take this chance to advance $t_{prop,p}$ to the starting time of this local collection[5].

3.2.7 Sending Time Stamps

When processor p's collector increases the time stamp of an outlet x with a matching inlet y, it sends the new time stamp value of x to y's collector. Invariant 3 may be violated in the period after the time stamp of x is increased till y's new time stamp is received. Therefore, $t_{trans,p}$ is reduced to the minimum of $t_{trans,p}$ and the original time stamp of x.

3.2.8 Receiving Time Stamps

When a time stamp for an inlet is received, the time stamp of the inlet (if lower) is increased to the received value. Invariant 1 may be violated by this increase. To guard against this, the value of t_{prop} is set to the minimum of the old time stamp of the inlet and the value of t_{prop} before the increase (section 3.2.5).

3.3 Updating $t_{trans,p}$

We have shown that our algorithm does not reclaim any live inlet erroneously. However, nothing is said about the progress of the algorithm. To ensure that garbage is eventually reclaimed, t_{min} is required to be increasing. We add two things to get a complete distributed garbage collection algorithm, First, we find a way to increase $t_{trans,p}$ of all processors. Second, we find a way to calculate t_{min}.

[5]We cannot increase $t_{prop,p}$ to a value greater than this because local computation changes accessibility of outlets.

From the definition of $t_{trans,p}$, $t_{trans,p} \leq$ old time stamps of an outlet whose new time stamps have been sent and not acknowledged (section 3.2.7). We let $t_{trans,p}$ be the minimum of all such old time stamps. Thus $t_{trans,p}$ is decreased or increased when a time stamp message is sent or is acknowledged. These operations can be implemented efficiently using a heap.

3.4 Estimating t_{min}

Because of asynchronous clocks, we find the minimum t_{min} in a period of time instead of that in a specific moment. A leader processor initiates the calculation of t_{min}. First, it broadcasts to all processors, including itself, an initialization message. When the leader has got all replies, it broadcasts another message. Upon receiving the second message, each processor calculates and replies $\min(t_{prop,p}, t_{trans,p})$ during the period between the receiving times of two messages. Using the returned values, the leader finds and broadcasts to all processors the global minimum t_{min}, which is the threshold for reclaiming garbage.

Calculation of t_{min} as described above runs continuously at the background it can be shown that t_{min} is non-decreasing. Based on the fact that the time stamp of any garbage inlet does not increase indefinitely, all new inlets and outlets are stamped with current time, progress of our distributed garbage collection algorithm can be guaranteed.

4 Conclusions

Our algorithm reclaims distributed garbage in a message passing processor network with unsynchronised clocks and non-zero network delay. To decrease the latency for reclaiming acyclic garbage, reference counting mechanism is included in the algorithm.

Our algorithm is different from algorithms depending only on reference counting [8, 9] or graph traversal [5, 7]. It combines the fast reclamation of acyclic garbage associated with reference counting and the effective reclamation of cyclic garbage associated with graph traversal. Although similar hybrid schemes [1, 3] have been proposed, our scheme addresses some problems neglected by them. In [1] an hybrid algorithm is presented. This algorithm uses marking where ours uses time stamps which are more efficient owing to temporal overlapping. Also, we address some specific issues like message delay and clock synchronisation. The algorithm described in [3] is intended for shared memory architecture where message delay and asynchronous clocks does not exist.

Acknowledgements

The authors would like to thank Dr. Olin Shivers at MIT and Mr. Kelvin Kwan at University of Hong Kong for their comments on the draft of this paper.

References

[1] J. K. Bennett, "The Design and Implementation of Distributed Smalltalk," *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, pp. 318-330, Dec. 1987.

[2] J. Cohen, "Garbage collection of linked list data structures," *Computing Surveys*, 13(3):341-367, Sep. 1981.

[3] J. DeTreville, *Experience with Concurrent Garbage Collectors for Modular-2+*, System Research Centre Research Report 64, DEC, Nov. 1990.

[4] E. Dijstra at el, On the fly garbage collection, an exercise in cooperation. *CACM 21*, 11, Nov. 1978, pp. 966-975.

[5] J. Hughes. *"A Distributed Garbage Collection Algorithm". Proc of Functional Languages and Computer Architectures* 1985, pp. 256-271.

[6] D. Kwan. Master Thesis, in preparation.

[7] R. Ladin and B. Liskov. *"Garbage Collection of a Distributed Heap". Proc. of IEEE Symposium on the Principles of Distributed Computing.* 1992, pp. 708-715.

[8] C. W. Lermem and D. Maurer. A Protocol for Distributed Reference Counting. Proceedings of 1986 ACM Conference on Lisp and Functional Programming, MIT, 343-350

[9] D. Plainfossé and M. Shapiro. Experience with a Fault-Tolerant Garbage Collector in a Distributed Lisp System. *IWMM 1992 International Workshop of Memory Management*, pp. 116-133, Springer Verlag, LNCS series.

[10] P. Wilson, Uniprocessor Garbage Collection Techniques. *IWMM 1992 International Workshop of Memory Management*, pp. 1-42, Springer Verlag, LNCS series.