Incremental Distribution of Timestamp Packets: A New Approach To Distributed Garbage Collection

Marcel Schelvis

ParcPlace Systems †
1550 Plymouth Street
Mountain View, CA 94043
(415)691-6750
schelvis@ParcPlace.COM

Abstract

A new algorithm for distributed garbage collection is presented. This algorithm collects distributed garbage incrementally and concurrently with user activity. It is the first incremental algorithm that is capable of collecting *cyclic* distributed garbage. Computational and network communication overhead are acceptable. Hosts may be temporarily inaccessible and synchronization between hosts is not necessary. The algorithm is based on asynchronous distribution of timestamp packets each containing a list of last-access times of some relevant remotely referenced objects. Finally, the correctness and time complexity of the algorithm are discussed.

Key words and phrases

Incremental distributed garbage collection, cyclic garbage, multiple-timestamp packets.

1. The problem: distributed garbage

In Smalltalk [Goldberg83], Lisp [Allen79], and similar systems, storage is dynamically allocated from a *heap*. In such systems, chunks of data, which we will call *objects*, are the nodes (or vertices) of a directed graph. Pointers between objects are the edges of this graph. Some objects are predefined *roots*. They are directly accessible from outside the graph. Objects *live*, if they have a root. This is the case if they are accessible via a path of inter-object pointers starting from a root. The problem of *garbage collection* is that of reclaiming space occupied by *dead* objects, which is data that has become inaccessible and therefore useless. Some requirements complicate the design of garbage collectors (see also section 3):

† Part of this work was done while the author was at Océ-Nederland B.V., the Netherlands.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 089791-333-7/89/0010/0037 $1.50

- ☐ Obviously the main requirement for a garbage collection algorithm is that it should collect *all* objects that are dead, and *only* those objects that are dead. *Cyclic garbage* is difficult to detect and many algorithms fail in this respect.
- ☐ Garbage collection should be *concurrent* with user activity (called the *mutator* as opposed to the *collector*).
- ☐ The *fragmentation* of the heap must be handled.
- ☐ The *computational overhead* of garbage collection must be acceptable.

In distributed object-oriented systems like DistributedSmalltalk [Schelvis88], objects are distributed over a number of hosts on a network. Each host keeps his objects in one or more local spaces. Collection of garbage that spans more than one host (distributed garbage) introduces some additional problems:

- ☐ Gaining access to objects on remote hosts is sometimes impossible, because a remote host may be down (or just inaccessible) and even if possible, such access is relatively expensive because of network communication overhead.
- ☐ A number of processors must cooperate and may have to be synchronized.
- ☐ Distributed *cyclic* garbage should be collected also. Until now no algorithm is known that is capable of doing so in an incremental fashion.

2. A new solution: asynchronous distribution of timestamp packets

We developed, implemented and tested a new algorithm for distributed garbage collection with the following properties:

- ☐ Dead objects are *incrementally* collected within finite time, *including cyclic distributed garbage*. Living objects are not collected.
- ☐ Collector and mutator activities are concurrent.
- ☐ Garbage collection is a set of independent local activities. Any host may start such an activity at any time without any synchronization with other hosts being necessary.
- ☐ Hosts may be down. During the down-period of some host, only the collection of distributed garbage part of which is on this host, is blocked (temporarily). Thus, a host can always collect its local garbage.
- ☐ Communication between hosts for garbage collection purposes is minimal and computational overhead is acceptable.

The algorithm is based on the following idea: all information needed to decide if an object lives or not is present in the system, although distributed across different hosts. A merge of some pieces of the information may render a complete enough picture of the relevant part of the distributed object graph for some object. This merge may be accomplished step by step (i.e., incrementally), by a process of repeated asynchronous distribution of information. All hosts receive information from some related hosts, combine this information with local information, and send the results to another set of related hosts. The flow of information will correspond with the pattern of remote references. Each unit of information is a *timestamp packet* containing a list of last access times of a few relevant remotely referenced objects. Our algorithm is explained in section 4.

The relevance of our algorithm is not restricted to the *distribution*-aspect of distributed object-oriented systems. For example, it facilitates the design of *persistent* object-oriented systems or even (distributed) object-oriented database systems: units of persistency or database objects can be small to medium sized object spaces (that may migrate, be replicated, etc.). Garbage collection of such a space can be done without the need to immediately access other spaces on disk or on remote hosts.

3. Previous work on garbage collection

A number of garbage collection techniques exist that can be classified as follows:

- ☐ *Object-based methods* which concentrate on the *death of individual objects*, such as *reference counting*. Object-based methods keep track of the incoming pointers of each object, for example, by dynamically updating a count of them. When this reference count becomes zero, the object is dead and its space is reclaimed. A distributed object-based method is *weighted reference counting* [Bevan87].
- ☐ *Graph-based methods* which concentrate on the *liveness of object graphs*, such as *mark & copy*, where the graph of living objects is traversed starting from the roots and traversed objects are moved to a free part of the heap. The old part may then be reused. Note that no dead objects are accessed.
- ☐ *Hybrid methods* (a mixture of object- and graph-based methods) such as *generation scavenging* [Ungar84] and some *incremental distributed* garbage collection algorithms [Ali84]. Also the algorithm presented in this paper is such an incremental distributed scheme.

- *Cyclic garbage*
An important difference between object- and graph-based methods is the availability of information with respect to the state of liveness of objects. The information available to object-based methods reflects only those nodes of the graph that have *direct* references to the node in question. This information is a list of such nodes like *remembered tables* used for generation scavenging or it is a simple reference count. Object-based methods fail to collect *cyclic garbage*, since every node on a cycle is referenced and hence cannot be distinguished from a living node. Therefore, object-based systems periodically must invoke some graph-based method which does not fail in this respect. This is even true for *cyclic reference counting* methods in combinator machines [Mago79][Treleaven82]. Graph-based methods for non-distributed systems have no problem with cyclic garbage. *Distributed* graph-based methods like network-wide *mark & sweep* need the synchronized cooperation of *all* hosts. However, this is difficult to realize in a typical distributed system, where usually at least some hosts are not able or willing to cooperate.

- *Heap compaction*
Because of heap fragmentation the living objects must be compacted periodically. Heap compaction is inherent to graph-based methods. Object-based methods must compact separately.

- *Concurrent garbage collection*
Concurrency of collector and mutator activity, which allows users to continue their activities during garbage collection, is inherent to object-based methods (apart from compaction and collection of cyclic garbage). For graph-based methods, concurrency is possible by traversing the object graph stepwise and breadth-first [Dijkstra78][Baker78].

- *Computational overhead*
when relatively many objects are created that die soon, object-based methods are expensive because each object is handled individually although sometimes the death of some short-lived objects can be foreseen and the overhead avoided. On the other hand, when many objects keep on living for a long time, graph-based methods are expensive, since all these long-lived objects are moved around the heap many times. Unfortunately, most systems have both characteristics at the same time.

Lieberman and Hewitt proposed a hybrid method to solve the computational dilemma. This method is based on the *lifetimes of objects* [Lieberman83] and is ususally called *generation scavenging* [Ungar84/88]. Objects of about the same age are kept in a separate space (generation), which is marked & copied independently from other generations. The older the generation, the less frequently it is garbage collected. Intra-space garbage collection is breadth-first mark & copy starting from objects that are referenced from other spaces. In order to detect these objects, inter-space references are kept track of

dynamically. As a result of garbage-collection frequency tuning, generation scavenging has a low computational overhead. Generation scavenging inherits the cyclic garbage problem from its object-based component: cyclic garbage that spans more than one space is not collected. By aging, distributed cyclic garbage is supposed to end up in the oldest generation of objects, and thus become local cyclic garbage, which can then be collected.

A distributed hybrid scheme is proposed by [Ali84]. Local spaces are like Lieberman's generations. Hosts cooperate, but in a very loose way. No synchronization is needed. Distributed cyclic garbage cannot be collected, because of the object-based component in the algorithm. Our new algorithm is also a hybrid method, but we succeeded in overcoming the distributed cyclic garbage problem. In the remainder of this section [Ali84]'s method will be discussed. Our algorithm will be discussed in the following section.

Incremental asynchronous collection of *acyclic* distributed garbage is straightforward. Hosts perform garbage collections on their local object space whenever they want, and gather remote references at the same time. Afterwards these remote references are sent to the appropriate remote hosts, each of which updates a table of remotely referenced local objects. We will call this table the host's *entrance*. A remote reference is a tuple (*host identifier, index in entrance*). Each entry in the entrance table contains a pointer to a local object, and a set of identifiers of hosts that reference this object. Entrance tables are root objects, so the objects they reference are not collected: remotely referenced objects are said to live by *prevention*. When the set of remote host id's for some entry is empty, the entry is removed, including the pointer to the formerly remotely referenced object. If the object has no other root it will be collected during the next garbage collection on its host.

A solution for the distributed cyclic garbage problem, which is analogous to the aging solution in generation scavenging, is to keep track of the last-access time of remotely referenced objects and to move subgraphs that are not accessed for a long time to some host from which they are referenced. In our algorithm we use these timestamps, but we don't need to move objects around before they are collected.

4. The timestamp packet distribution algorithm

A first, simple packet distribution algorithm resembles the scheme used for distributed deadlock detection in the R* system [Mohan86]. This single-node packet algorithm (see fig. 2) will be discussed in more detail later in this section. It detects *absence of roots* in the *entrance graph G* in a similar way as *deadlock* is detected in the *wait-for graph* of distributed transactions.

Fig. 1 (left) shows an object graph distributed over 4 hosts. Objects painted on the edge of a host are roots of the object graph. Gray or black objects are remotely referenced. Fig. 1 (right) shows the corresponding entrance graph. Entrance nodes are shown as objects, namely the objects for which they act as entrance.

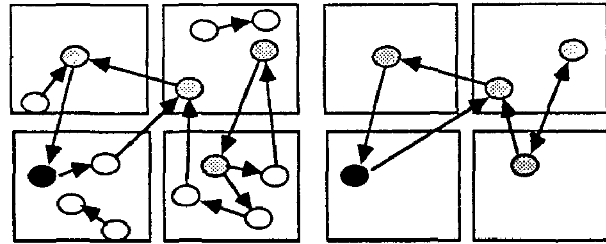

Figure 1. An object graph and its entrance graph

An *edge* in G represents the set of local paths from some entrance node to an object with a remote reference to another entrance node. Node n is an entrance *root* (painted black in fig. 1) if a path exists in some local object graph, from a local root to an object with a remote reference to n. During the incremental breadth-first object-graph traversal for local garbage collection, G's edges are gathered. First the local roots are traversed and then the entrance table. Remote references detected during the first phase determine entrance roots. The packet to be sent later, called a *root-packet*, contains the node id. Remote references detected during the second phase determine entrance edges. The packet to be sent later is based on the entrance entry which is currently the root of the traversal. During this second traversal, objects with remote references are not copied to their new space yet, but are remembered in a table instead, so that all paths from the local entrance table to these objects will be detected. The table of remembered objects is copied last. Our scavenge algorithm is incremental and therefore allows for concurrent garbage collection. Further discussion of the scavenge algorithm is outside the scope of this paper.

Packets are asynchronously and repeatedly sent to each remotely referenced host; that is, after some (one or more) local garbage collections. If there is an entrance edge $n \rightarrow m$, then the set of packets n's host sends to m's host will contain a packet n (associated with sender n and address m). P_n is the set of packets received for entrance node n. Although hosts actually send and receive sets of packets, it is more convenient to think in terms of entrance nodes receiving, storing and sending their own packets, therefore *send* p in *sendfrom*(n) means that for each entrance pointer $n \rightarrow m$, n sends a packet p to m. When an entrance pointer $n \rightarrow m$ is removed as a result of either mutator or collector

activity, then next time m receives packets from n's host, there will not be a packet sent by n anymore. In our algorithms this is expressed by m receiving the empty packet ε.

```
Collector:

sendfrom(n)
      if every p∈P_n : p = n or p = ε
      then
         "absence of roots is detected"
         send ε;
         remove n
      else
         if every p∈P_n ≤ n
         then
            send n
         else
            if every p∈P_n = m and m>n
            then
               send m
            else
               send nothing
            fi
         fi
      fi

receive(m, p, n)
      if p = ε
      then
         remove p_n from P_m
      else
         replace p_n in P_m by p
      fi

Mutator:
      when a remote reference m → n is created,
      a packet m is added to P_n
```

Figure 2. A single node packet distribution algorithm

If all packets on a node n are "n" then this implies that all packets received by n in the past departed from n. Therefore n (and all nodes from which it is accessible) are dead and can be removed. Because of the root packet, an entrance root and all nodes accessible from it will not be removed.

In table 1, "$(i)j$" for node n at time t indicates that the last packet received by node n from node i at time t is j. "$\Rightarrow k$" indicates that as long as the set of packets at node n is not changed, then after each garbage collection on the host of node n a packet k will be sent to all nodes that n references. In fig. 3 (left) and table 1, at time 0, node 1 has packets 2 from node 2 and 3 from node 3. Since these packets are different, node 1 is blocked. Node 4 initially has received 1 from node 1, but since its own id is higher, it sends packets 4. Nodes 2 and 3 receive 4, their own id is lower, so they also send 4. When node 1 has received the packets 4 from both nodes 2 and 3, all its packets have the same source (apparently node 4), which is higher than the id of node 1, so node 1 is unblocked and will send packet 4. When this packet is received by node 4, all packets on node 4 (only one in this case) also started from node 4, and therefore node 4 is garbage, as are all nodes from which node 4 is accessible (nodes 1, 2 and 3).

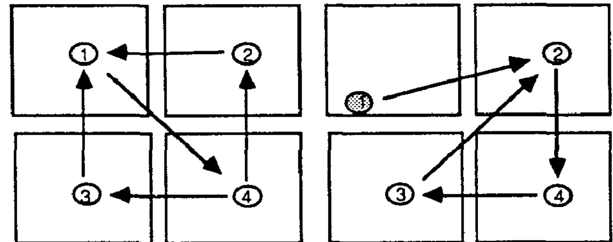

Figure 3. A dead and a living distributed cyclic graph

Table 1 shows 6 snapshots of the packet distribution process, at times when significant progress has been made. In fig. 3 (right) and table 2, 1 (gray) is a root of the object graph, and hence 2 is a root of the entrance graph 2→4→3→2. As long as the graph does not change, the packets on node 2 will be different (1 from node 1, 4 from node 3). Thus node 2 will be blocked, node 4 will not receive packet 4, and no garbage collection will occur.

nodes / time	1	2	3	4
0	(2)2 (3)3	(4)4	(4)4	(1)1
1	blocked	(4)4⇒4	(4)4⇒4	(1)1⇒4
2	(2)4 (3)4⇒4			
3				(1)4⇒ε removed
4		(4)ε⇒ε removed	(4)ε⇒ε removed	
5	(2)ε (3)ε⇒ε removed			

Table 1. Packet distribution process of a dead graph

A non-cooperating host (for whatever reason, e.g., it is down, or there is no garbage collection needed) will also block packets, but only temporarily. In fig. 3 (left),

for example, the host of node 1 might be down for some period, before sending packet 4 to node 4. But eventually packets with the highest identifier will return home via all incoming pointers. At this time the host of this object has complete enough information: its object has no roots and can be collected.

nodes / *time*	2	3	4
0	(1)1		
	(3)3	(4)4	(2)2
1			(2)2 ⇒ 4
2		(4)4 ⇒ 4	
3	(1)1		
	(3)4 blocked		

Table 2. Packet distribution process of a living graph

Packets are added by the mutator only, namely, when it atomically transfers references from one host to another. In our DistributedSmalltalk system [Schelvis88] all mutator activity for garbage collection purposes is incorporated in an atomic process-migration mechanism. Collectors update or remove packets. Therefore the only consequence of not receiving information from other collectors is that some dead nodes are not removed until new information is received.

The packet exchange is the only interaction necessary between hosts. The frequency of it can be dynamically modified: low when there is little distributed garbage, otherwise higher. A natural policy is to send some packets along with each mutator access of a remote host. Thus, collector packet exchange may partly be incorporated in the process-migration mechanism, thus minimizing network communication overhead.

Since there are no rules prescribing some time order or any other dependency between the garbage-collection activities of different hosts, no synchronization is necessary.

The single node packet algorithm is suitable to do incremental distributed garbage collection in graphs that may contain cycles, but no *subcycles* in these cycles. Nodes on a subcycle are always blocked (see fig. 4 and table 3; the blocking node is painted gray). The only way to know if the subcycle does not contain roots (in which case the node could be unblocked) would be to unblock and let a packet pass and traverse the subcycle! In order to solve the dilemma, we will introduce *multiple node packets*. In case of a potential subcycle a new packet is created that starts from the same node where the old type is blocked. When all received packets are of the old or the new type, the node is unblocked for the old type of packets. The contents of the new packet is the old packet extended with the current node id.

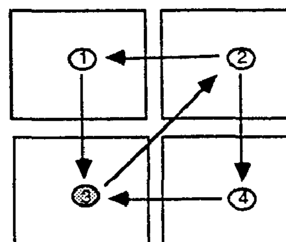

Figure 4. A cyclic graph containing a subcycle

Another problem with the node packet algorithm is that the roots of living structures may not move at will from one host to another. Absence of roots can be incorrectly detected when the roots in such a graph would play a pathological "hide and seek": just before packets would be blocked at a node, the root moves to another host.

nodes / *time*	1	2	3	4
0	(2)2	(3)3	(1)1	(2)2
			(4)4	
1		(3)3⇒3		
2	(2)3⇒3			
3			(1)3	
			(4)4 blocked	

Table 3. A failing packet distribution process

The solution is to use *timestamps* instead of object identifiers as the unit of graph information. Each entrance entry is associated with a unique timestamp. Every time the entry is accessed, this timestamp is updated. In a living graph, as long as it is not accessed, packets will be blocked. As soon as there is access (with the possibility of moving roots) the timestamp of the accessed node is updated. The death of a distributed graph will be detected by the node with the highest timestamp at the time of death. In order not to be detected, a moving root is bound to access this node. But then, the timestamp of this node is updated, so the absence-of-roots detection process has to start all over again and no premature garbage collection occurs.

The multiple-timestamp packet distribution algorithm (*MTP*) is shown in fig. 5. Nodes are identified by means

of their timestamp, so node n is the node with timestamp n. Each packet is a list of one or more timestamps. A timestamp is a (*localtime.hostid*) tuple (the hostid makes timestamps unique). Each host has one local root that is associated with timestamp 0 (zero). Root packets therefore consist of (0.hostid). The last timestamp of a packet may carry an accent. For example *fe'* means *there is no path from a root via e to f*, whereas *fe* means *there may be a path from some root via e to f*. The former is called an *answer* packet, the latter a *question* packet. Some examples of timestamp packet operations used in the *MTP* algorithm:

- *timestamp ordering*: $(12.3) > (6.10) > (6.9)' > (6.9)$
- $first(feb) = f$, $last(feb) = b$, $length(feb) = 3$
- *lexical packet ordering*: $fed > fecb > fec$, $fec' > fec$
- *prefixing*: $\varepsilon, f, fe, fec \subseteq fec$
- $strip(fecb, c) = fec$
- *timestamp adding*: $fec+b = fecb$, $fec+c = fec$, $fe+e' = fe'$
- if $fec' \in P$, then $fec, fecb, fecba'$, etc. are called *obsolete*.

For convenience, all nodes n have the empty packet $\varepsilon \in P_n$. Note that

$$MTP(\{\varepsilon\}, n) = n'.$$

Also note that for a node n at time t with packets received P_t and $p_t = MTP(P_n, n)$, the following invariants hold:

$$p_t = t_1 t_2 .. t_k \Rightarrow t_1 > t_2 > .. > t_k$$

This invariant holds, because *MTP* strips timestamps smaller than n before adding n or n'.

$$p_t \geq max(P_t)$$

This invariants holds because p_n consists of $max(P_t)$ with timestamps smaller than n stripped and possibly n or n' added. Note that sometimes n is updated in order to maintain this invariant.

$$p_{t_2} \geq p_{t_1} \Rightarrow t_2 \geq t_1$$

This invariant states that packets sent by a node are equal or larger than the packets it sent in the past. This invariant holds because the second invariant implies that packets in P_t cannot be replaced by smaller packets and because the invariant is explicitly maintained when packets are removed.

```
Collector:

sendfrom(n)
    p ← MTP( P_n, n);
    send p;
    "if no roots, remove node"
    if answer( p) and length( p) = 1
       then remove n
    fi

MTP(P_n, n)
    p ← strip(max(P_n), n);
    if for all non-obsolete q∈P_n: q=p+t' or q⊆p
       "t some timestamp"
    then
       if answer( p)
       then
          return( p)
       else
          return( p+n')
       fi
    else
       if answer( p)
       then
          n ← localtime ← max(localtime, last( p)+Δt);
          return( strip( p, n) + n)
       else
          return(p+n)
       fi
    fi

receive(m, p, n)
    "p_n∈P_m is the packet, that was previously
    received from n; p_m is the last packet created by m"
    if p = ε
    then
       remove p_n from P_m;
       if p_m > MTP(P_m, m)
       then
          m ← localtime ← max(localtime, first( p)+Δt)
       fi
    else
       replace p_n in P_m by p
    fi

Mutator:
    When a remote reference m→n is created,
    a packet m is added to P_n;
    When an entrance entry is created,
    it is assigned a unique timestamp;
    When an entrance entry is accessed,
    its timestamp is updated;
```

Figure 5. The multiple-timestamp packet algorithm

Table 4 shows the garbage collection process of the multiple-timestamp algorithm for the distributed graph of fig. 6.

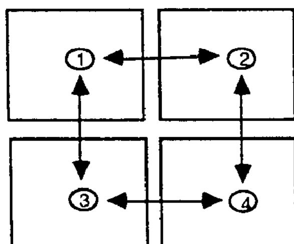

Figure 6. Distributed cyclic garbage containing subcycles

5. Indication of correctness

An entrance graph G is like a living organism, the lifetime of which is a sequence of mutations performed by the mutator and the collector. The mutator adds nodes and packets. The collector removes nodes and updates and removes packets. For convenience we think of the mutation sequence as one mutator event (the birth of the first node) followed by zero or more collector events, followed by one mutator event, etc. A series of collector events is thought of as being a process, that starts after a mutator event m_i and terminates at the next mutator event m_{i+1} or, in case of a dead graph, when the graph is removed.

During a garbage collection process an entrance graph can be represented by a hypergraph, the nodes of which represent maximal strongly connected (MSC) entrance subgraphs (a directed graph is strongly connected if there is a path from each node to each other node. A subgraph of a graph is *maximal* strongly connected if no more nodes can be added maintaining the strong connectivity). Note that this hypergraph is *acyclic*, since a cycle would imply that the graphs on this cycle are strongly connected, and therefore these graphs would not be *maximal* strongly connected.

We will give an indication of the correctness of the *MTP* algorithm as follows:

- We present an invariance that holds for all packets generated in an MSC subgraph (*MTP* invariance).
- We show that mutator events do not violate the *MTP* invariance.
- We show that collector events do not violate the *MTP* invariance and also, by complete induction, that during a garbage collection process an MSC subgraph G has the same *MTP* behavior as a one node graph $\{n\}$, where $n = max(G)$; n is called G's *parent*. The pointers to G are redirected to n, the packets associated with these pointers assigned to n, and finally, the pointers out of G are assigned to n.
- The entrance graph's *MTP* behavior now is comparable to that of the acyclic entrance hypergraph, therefore the correctness of the *MTP* algorithm would follow from its correct manipulation of an acyclic entrance graph.

time \ nodes	1	2	3	4
1				(2)2 (3)3 ⇒ 4
2			(1)1 (4)4 ⇒ 4:3	
3	(2)2 (3)4:3 ⇒ 4:3:1			
4		(1)4:3:1 (4)4 ⇒ 4:3:2		
5	(2)4:3:2 (3)4:3 ⇒ 4:3:2:1'			
6		(1)4:3:2:1' (4)4 ⇒ 4:3:2'		
7	(2)4:3:2' (3)4:3 ⇒ 4:3:2'			
8			(1)4:3:2' (4)4 ⇒ 4:3'	
9	(2)4:3:2' (3)4:3' ⇒ 4:3'			
10		(1)4:3' (4)4 ⇒ 4:3'		
11				(2)4:3' (3)4:3' ⇒ 4' removed
12		(1)4:3' (4)4'⇒4' removed	(1)4:3' (4)4' ⇒ 4' removed	
13	(2)4' (3)4' ⇒ 4' removed			

Table 4. A multiple-timestamp packet distribution process

We first prove the correctness of *MTP* for acyclic entrance graphs. A node n of an acyclic entrance graph is either

- a dead leaf: if n is dead then in finite time $P_n = \{\varepsilon\}$ and n is removed (after sending packet n').
- not a dead leaf: (n has incoming pointers or is entrance root; in both cases the number of packets (ε is not counted) $|P_n| > 0$. If

$$s = strip(max(P_n), n) = \varepsilon,$$

then

$$MTP(P_n, n) = n$$

(since the graph is acyclic, n can not receive packets containing n, such as nm', which is necessary to

satisfy the requirement to produce packet n' instead of n (see also fig. 5)). If $s \neq \varepsilon$, then

$$MTP(P_n, n) = sn \quad (|P_m| > 1) \text{ or}$$

$$MTP(P_n, n) = sn' \quad (|P_m| = 1)$$

In neither of the three cases will n be removed.

QED

Before we discuss the remaining part of the proof, we will analyse the structure of an MSC graph. Each MSC subgraph consists of a parent node, which is the node with the currently highest timestamp, and a number of semi-MSC subgraphs. Semi-MSC means that these subgraphs are strongly connected *only* via the parent node. Each semi-MSC subgraph itself consists of one parent node and a number of semi-MSC subgraphs, and so on recursively.

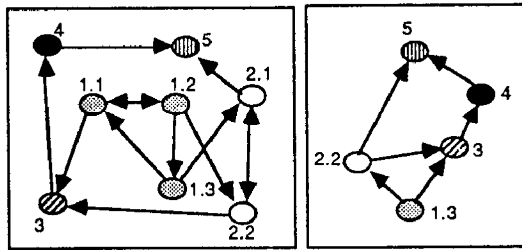

Figure 7. Entrance graph and its hypergraph

In fig. 7 an entrance graph is shown along with the corresponding hypergraph. MSC graphs are represented by their respective parent nodes, for example, node 1.3 represents MSC graph (1.3, 1.2, 1.1). Entrance node 4 is accessible from outside the graph, and is therefore a root. Nodes 4 and 5 live, the other nodes are dead and will be removed. In MSC graph (1.3, 1.2, 1.1), 1.3 is parent and (1.2, 1.1) is the (only) semi-MSC subgraph. In (1.2, 1.1), 1.2 is parent and (1.1) is the (only) semi-MSC subgraph. The hypergraph of the graph in fig. 6 consists of node 4 only, which represents parent 4 and the semi-MSC subgraph (1, 2, 3). Node 3 is the parent of this subgraph, etc.

The parent-child relationships within a semi-MSC graph can be represented by a tree. When a tree corresponding with a semi-MSC graph has depth k, the graph itself is said to be k deep. The graph in fig. 7 has depth 3, the one in fig. 6 has depth 4.

A semi-MSC graph does not contain roots if the parent is not a root and if none of its child semi-MSC subgraphs contains a root. The absence-of-roots detection algorithm is based on this recursive definition. The parent n asks his children (the semi-MSC subgraphs) if they have roots by means of a packet n. A child in its turn contains a parent node (say m) and a number of semi-MSC subgraphs. This parent node will ask the question nm to its children, and so on recursively. If eventually all children of m respond nml' (no roots), then m responds nm' to its parent n.

The MTP invariance

Given a semi-MSC graph with parent n, then for all $m \in G$,

$$MTP(P_m, m) \leq MTP(P_G, n) \text{ or}$$

$$MTP(P_G, n) \subseteq MTP(P_m, m)$$

(every packet generated in G is smaller than (or equal to) the packet that n would create if it would receive G's packets, or the latter packet is a prefix of the former). Initially, that is, before any mutator or collector events, the invariance holds, since for all nodes

$$m \in G \,:\, m \leq n \text{ and } max(P_m) \leq max(P_G).$$

Mutator events

A node $r \in G$ which is accessed by the mutator is an entrance root per definition, because the mutator is the root of the local object graph from which the entrance node is apparently accessible. The mutator updates r's timestamp. As a result the tree of semi-MSC subgraphs is rearranged: r may rise in the tree and may even become parent of the MSC subgraph it is a node of. If r does become parent of G, the *MTP* invariance holds: since $r > n$,

$$MTP(P_G, n) < MTP(P_G, r),$$

and, because of the invariance before the mutation, for all $m \in G$,

$$MTP(P_m, m) \leq MTP(P_G, r)$$

If r does not become parent, then nothing has been changed for nodes $m \in G$, except for node r. But then, since $r < n$, the invariance holds for r also.

Collector events

Note that because of the *MTP* invariance, at time t_1 packets sent by G,

$$p_G^{t_1} \leq MTP(P_G^{t_1}, n) \text{ or}$$

$$MTP(P_G^{t_1}, n) \subseteq p_G^{t_1}$$

From the induction proof that is given hereafter, it follows that collector events do not violate the *MTP* invariance (see induction hypothesis).

Induction hypothesis

Given a semi-MSC graph G of depth k with parent node n and a set of packets received from nodes outside G at time t_1, $P_G^{t_1}$, then there is a time $t_2 > t_1$, that all packets sent by nodes of G,

$$p_G^{t_2} = MTP(P_G^{t_1}, n) \text{ or}$$

$$MTP(P_G^{t_1}, n) \subseteq p_G^{t_2}$$

where $MTP(P_G^{t_1}, n)$ is the packet created at node n by the multiple-timestamp packet algorithm after redirecting G's incoming pointers and packets to n. Note that the first condition implies

$$\text{answer}(MTP(P_G^{t_1}, n))$$

for $G-n$ (nodes other than n).

Induction base

Trivial, since an MSC graph of depth 1 *is* its parent node.

Induction proof

We have to prove the hypothesis for a semi-MSC graph of depth $k+1$. This graph consists of a parent node and a number of semi-MSC subgraphs each of depth $\leq k$. Because of the induction hypothesis these subgraphs may be replaced by their parent nodes (incoming and outgoing pointers and packets are redirected). Note that the resulting graph is strongly connected. Also note that the nodes other than the parent node do not contain cycles (otherwise the subgraphs they represent wouldn't be *maximal* strongly connected).

We first will introduce the notion of *self-reproducing properties* of packets, which, again, are reminiscent of living organisms. We then will show how packets with such properties are propagated throughout G in three phases: a *question-propagation*, a *response-generation* and a *response-propagation* phase.

Suppose a packet p has some property ξ (e.g., $\xi(p) \equiv first(p) = t$) and suppose that for all $m \in G$,

$$(\text{there is a } p \in P_m : \xi(p)) \Rightarrow \xi(MTP(P_m, m))$$

(when a node has received a packet with property ξ, it will create and send packets with property ξ: the property has reproduced itself). In addition, if G is strongly connected and

$$(\text{there is a } n \in G \text{ and } p \in P_n) : \xi(p),$$

and if all $m \in G$ call *MTP* and send packets within finite time, it then follows that *all* packets produced in G will have property ξ within finite time. Call ξ *strongly self-reproducible in G*. Note that when a new property is introduced that violates the first condition for the old property, then the old property loses its self-reproducingness: its propagation fades out and instead the new property is propagated. On the other hand, a new property may also *imply* an old property.

Property ζ is *weakly* self-reproducible in G, if for all $m \in G$,

$$(\text{all } p \in P_m : \zeta(p)) \Rightarrow \zeta(MTP(P_m, m)),$$

and if G is a strongly connected graph with some node n such that $G-n$ is acyclic and

$$\zeta(MTP(P_n, n)),$$

and if all $m \in G$ call *MTP* and send packets within finite time. As soon as *all* packets residing at a node have the property ζ, then it will create and send packets with property ζ. The conditions for the propagation of a weakly self-reproducible property throughout G are comparable to those of an *activity network*: an activity will start when all activities that it depends on are finished. Necessary and sufficient conditions for all activities to be carried out are that they all take finite time, that the activity network is acyclic, and that the root activity is carried out.

Question-propagation

We will show that property

$$\xi(p) \equiv s \subseteq p$$

$$\text{with } s = strip(max(P_G^{t_1}), n) = n_1 n_2 .. n_l,$$
$$n_1 > n_2 > .. > n_l > n,\ n \text{ parent of } G$$

is strongly self-reproducible in G. Suppose a node $m \in G$ has received a packet p with property ξ, then also

$$\xi(MTP(P_m, m))$$

since there is no packet $q \in P_m$ such that $q > s$. If such a packet would exist, a prefix of it would be present in $P_G^{t_1}$, and from that would follow that

$$s \neq strip(max(P_{t_{1G}}), n)$$

Because G is strongly connected, ξ is strongly self-reproducible, and thus within finite time n will receive a packet p, with $n_1 n_2 .. n_l \subseteq p$.

Response-generation

Because both P_n and $P_G^{t_1}$ contain a packet with property ξ,

$$MTP(P_n, n), MTP(P_G^{t_1}, n) = sn, sn' \text{ or } s$$

where question(sn), answer(sn') and answer(s).

- $MTP(P_G^{t_1}, n) = sn$:
  $MTP(P_n, n)$ can not be larger than $MTP(P_G^{t_1}, n)$, so $MTP(P_n, n) = sn$ and response-generation is finished.
- $MTP(P_G^{t_1}, n) = MTP(P_n, n) = sn'$ or s:
  Response-generation finished.
- $MTP(P_G^{t_1}, n) = sn'$ or s; $MTP(P_n, n) = sn$:

The last case is slightly more complicated. We will show that with packet sn a new *weakly* self-reproducing property is introduced in G,

$$\zeta(p) \equiv p = sn \text{ or } snr', r \in G$$

(Note that $\zeta \Rightarrow \xi$). A node m with packets P_m all having the ζ property, will reproduce a packet such that

$$\zeta(MTP(P_m, m))$$

Initially ζ holds for $MTP(P_n, n)$, and since $G-n$ is acyclic ($G-n$ are the nodes that represent semi-MSC subgraphs), the property will propagate throughout G. Note that the activity network corresponding to this phase is isomorphic to G, except for the parent node, the outgoing pointers of which correspond to the first activity and the incoming pointers of which correspond to the final activity (see fig. 8).

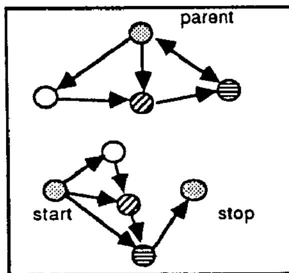

Figure 8. A semi-MSC graph and its MTP activity network

When all packets $p \in P_n : \zeta(p)$, the next packet n will produce is sn' or s (depending on s being an answer packet or not).

Response-propagation

This phase is similar to the question-propagation phase. The response generated in the previous phase, sn, sn' or s, introduces a new strongly self-reproducible property θ,

$$\theta(p) \equiv sn \subseteq p \text{ (or } p = sn' \text{ or } p = s)$$

(Note that θ violates ζ). If the response is sn (question packet), then θ is strongly self-reproducible for a reason similar to ξ. If the response is s or sn' (answer packet), then θ is strongly self-reproducible because all nodes $m \in G$ only have packets $p \subseteq snr'$ (see response-generation), so that within finite time, say at t_2, θ has propagated throughout G and therefore

$$p_G^{t_2} = MTP(P_G^{t_1}, n) \ (= s \text{ or } sn')$$

$$\text{or } MTP(P_G^{t_1}, n) \ (=sn) \subseteq p_G^{t_2}$$

QED

6. Indication of time complexity

From a single host perspective, it is interesting to know what percentage of cpu cycles is used for garbage collection. Because of the *locality of reference* phenomenon, the number of remote references will only be a fraction of the number of local references in the typical distributed system. As a result, the local scavenger's overhead will by far outnumber the *MTP* overhead. We therefore will not discuss the time complexity of an individual *MTP* algorithm.

From a distributed system perspective however, it is interesting to know how long it will take before distributed garbage is collected. We will consider some simply structured graphs among which the worst case.

A dead entrance subgraph G consisting of k nodes, n_1, .., n_k, is removed within $\underline{\omega}$ time, where $\underline{\omega}$ is a stochastic variable with expectation

$$E_{\underline{\omega}} \leq \frac{1}{2} \gamma \frac{1+p}{1-p} f(G)$$

- $\gamma = max(\gamma_1, .., \gamma_k)$ with $\gamma_i < \infty$ the mean time between successive packet distributions by the host of node n_i.
- $p = max(p_1, .., p_k)$, with $p_i < 1$ the probability that at an arbitrary time the host of node n_i is down or otherwise inaccessible.
- $f(G)$ is the number of garbage collections *resulting in some progress* of the garbage collection process.

Suppose that at time t_{gc} host i just has collected garbage and distributed his packets. Also suppose that host i has remote pointers to host j. Then the next time host j will receive packets from host i is $t_{gc} + \underline{\psi}_{ij}$, where $\underline{\psi}_{ij}$

is a stochastic variable with expectation

$$E_{\Psi_{ij}} = \gamma_i (1-p_j) + (\gamma_i + E_{\Psi_{ij}}) p_j$$

At *random time* t_r, the next time host j will receive garbage information from host i, is $t_r + \underline{\omega}_{ij}$, where $\underline{\omega}_{ij}$ is a stochastic variable with expectation

$$E_{\underline{\omega}_{ij}} = \frac{\gamma_i}{2} (1-p_j) + (\frac{\gamma_i}{2} + E_{\Psi_{ij}}) p_j$$

Therefore

$$E_{\underline{\omega}_{ij}} \leq \frac{1}{2} \gamma_i \frac{1+p_j}{1-p_j}$$

It is clear that $E_{\underline{\omega}_{ij}} < \infty$. It now follows that

$$E_{\underline{\omega}} \leq max(E_{\underline{\omega}_{ij}}) f(G) \leq \frac{1}{2} \gamma \frac{1+p}{1-p} f(G)$$

$f(G)$ depends on G's size and structure. E.g. $f(G) = 13$ for the graph in fig. 6 (see also table 4). In between these garbage collections, other garbage collections can occur that do not result in such progress (e.g., in table 4, between time 1 and 2, all garbage collections on hosts 1, 2 and 4). The graphs that we will discuss in this section, have a simple structure, and therefore it is easy to determine $f(G)$. For more complex graphs it will be more difficult. However, since the worst case $f(G)$ is computable, we can compute a rough upper bound for the time complexity of more complex graphs as well.

- *Single cycle*
  Suppose there are k nodes with timestamps

  $$n_1, .., n_k, \quad n_1 > .. > n_k,$$

  then if the nodes form a *single cycle*, $f(G) = 2k$. A packet with prefix n_1 starts from node n_1 and passes the other nodes, before returning to n_1, accounting for k steps. Then the packet n_1' follows the same route, while the nodes it passes are removed, accounting for another k steps.
- *List structure*
  In case of a non-cyclic list structure, $f(G) = k$. The tail of the list sends a packet t' and is removed, and this continues until the head of the list is removed.
- *The worst case*
  The worst case is a doubly linked list,

  $$n_1 \longleftrightarrow n_3 \; . \; . \; n_k \; . \; . \; n_4 \longleftrightarrow n_2$$

  First a packet with prefix n_1 goes from n_1 to n_2, accounting for $k-1$ steps, then a packet with prefix $n_1 n_2$ goes from n_2 to n_3 in $k-2$ steps, etc. Finally packet $n_1 n_2 .. n_{k-1}$ reaches node n_k, so that n_k unblocks and returns $n_1 n_2 .. n_{k-1} n_k'$ to n_{k-1}. This continues until $n_1 n_2'$ goes from n_2 to n_1. Now n_1 sends the final packet n_1' and is removed, then node n_3 does the same, and finally n_2. The total number of steps is

  $$f(G) = 2\sum_{i=1}^{k}(i-1) + k = k^2$$

If all dead nodes would be removed without distributing their final packet, then the number of steps required in the worst case would be $k^2 - k$ to remove the first node, then the absence-of-roots detection process would start all over again for the remaining graph of $k-1$ nodes, etc., so the total number of steps required would then be:

$$\sum_{i=1}^{k}(i^2 - i) = \frac{1}{3}k^3 - \frac{1}{3}k$$

A way to prevent this is to check if answer packets of length 1 are successfully sent and, if not, to postpone the removal of the dead node.

In fig. 6, if each host has a 50% probability of not being accessible, and if the mean time between packet distributions is 1 min., then the expected garbage collection time is 19.5 min.

7. Conclusions

A new algorithm for distributed garbage collection was presented. This algorithm collects distributed garbage incrementally and concurrently with user activity. It is the first incremental algorithm that is capable of collecting cyclic distributed garbage. Computational and network communication overhead are acceptable. Hosts may be temporarily inaccessible and synchronization between hosts is not necessary. The algorithm is based on asynchronous distribution of timestamp packets each containing a list of last-access times of some relevant remotely referenced objects. The correctness and time complexity of the algorithm were discussed.

8. References

[Ali84]
Mohamed Ali K. A-H, Object-oriented Storage Management and Garbage Collection in Distributed Processing Systems, The Royal Institute of Technology, Dept. of Telecommunication Systems - Computer Systems, Sweden, Report TRITA-CS-8406, December 1984

[Allen79]
J. Allen, Anatomy of Lisp, McGraw-Hill, New York, 1979.

[Baker78]
H.G. Baker, List Processing in Real Time on a Serial Computer, *Commun. ACM* 21, 4 (April 1978) pp. 280-294

[Bevan87]
D.I. Bevan, Distributed Garbage Collection Using Reference Counts, Parallel Architectures and Languages Europe Conference Proceedings Vol. II, pp. 176-187, *Lecture Notes in Computer Science*, Springer Verlag (1987)

[Brownbridge85]
D.R. Brownbridge, Cyclic Reference Counting for Combinator Machines, Functional Programming Languages and Computer Architecture, *Lecture Notes in Computer Science*, nr. 101, pp. 273-288, Springer Verlag (1985)

[Decouchant86]
D. Decouchant, Design of a Distributed Object Manager for the Smalltalk-80 System, *OOPSLA '86 Proceedings*, pp. 444-452

[Dijkstra78]
E.W. Dijkstra, On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM* 21 (11) (1978) pp. 966-975.

[Goldberg83]
A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation, Addison Wesley, 1983.

[Knuth68]
Knuth, D.E., The Art of Computer Programming 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA

[Lieberman83]
Lieberman H. and Hewitt C., A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, Vol. 26, No. 2, June 1983, pp 419-429

[Mago79]
G.A. Mago, A Cellular Computer Architecture to Execute Reduction Languages, *Int. J. Computer and Information Science*, Vol. 8(5) pp. 349-385 (1979)

[Mohan86]
Mohan C., Lindsay B., Obermarck R., Transaction Management in the R* Distributed Database Management System, *ACM Transactions on Database Systems*, vol. 11, nr. 4, pp. 378-396, Dec. 1986

[Schelvis88]
Schelvis M. and Bledoeg E., The Implementation of a Distributed Smalltalk, ECOOP'88 proceedings, *Lecture Notes in Computer Science*, nr. 322, Springer-Verlag, pp. 212-232

[Treleaven82]
P.C. Treleaven, D.R. Brownbridge, and R.P Hopkins, Data Driven and Demand Driven Computer Architecture, *ACM Computing Surveys*, Vol. 14(1) pp. 93-143 (February 1982)

[Ungar84]
D. Ungar, Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm, *ACM Software Engineering Notes*, April 1984, pp 157-167

[Ungar88]
D. Ungar and F. Jackson, Tenuring Policies for Generation-Based Storage Reclamation, *OOPSLA '88 Proceedings*, pp. 1-17, San Diego