Distributed Garbage Collection by Timeouts and Backward Inquiry

Sung-Wook Ryu, B. Clifford Neuman
{ryu, bcn}@ISI.EDU

Information Sciences Institute
University of Southern California

Abstract

We present a practical and efficient garbage collection mechanism for large scale distributed systems. The mechanism collects all garbage including distributed cyclic garbage without global synchronization or backward links. The primary method used for local and remote garbage collection is timeouts: each object has a time-to-live, and clients which have a link to an object must refresh the target object within the time-to-live to guarantee that the link will remain valid. For cyclic garbage collection: objects suspected to be garbage are detected by last referenceable timestamp propagation; and cyclic garbage is reclaimed by backward inquiry (back-tracing). Since, without additional overhead, the information about backward references can be obtained during the refreshing process, and since messages necessary for cyclic garbage collection are bundled with the messages used for the refreshing, communication, computation and storage overhead is minimized. This mechanism has been implemented and evaluated on Prospero directory service, and the performance results show that it works well for large scale distributed systems.

Keywords: Distributed Garbage Collection, Distributed objects, Cyclic Garbage, Timeouts, Back-Tracking

1 Introduction

This paper suggests a practical and efficient garbage collection mechanism for large scale distributed systems. A critical problem for large scale distributed systems which manage distributed objects is deciding when objects are not reachable from any clients\(^1\). Since each system has limited size of storage spaces, unreachable objects should be reclaimed as soon as possible while the referential integrity of the distributed systems is maintained.

This process, called garbage collection, is important for the following reasons. First, manual garbage collection is error-prone because it is difficult for users to maintain the information about references correctly. In the current World Wide Web, because owners move or delete their web pages without considering incoming links, there are many dangling links, which cause "Not Found" error messages. Second, since distributed objects are dynamically created, deleted, migrated, and shared across the network (large scale), it is difficult to determine when an object is not reachable, and whether it is safe to reclaim it.

In the ideal distributed system, objects continue to exist as long as they are reachable from clients. In practice, this is difficult to support in large scale distributed systems because of the following reasons.

- Distributed systems are administratively decentralized, so their well-behaving cooperation cannot be required.
- Distributed systems are very large in scale, so it is impossible to get a global view of clients, objects and their references.
- Servers and clients can crash during garbage collection related operations.
- Messages can be lost, and the network can be partitioned for a while.

Many solutions for distributed garbage collection have been suggested, but they are not suitable for large scale distributed systems because their target systems are small in scale or they cannot collect cyclic garbage. Especially, it is a challenging problem to collect distributed cyclic garbage.

[Ladin and Liskov, 1992] uses a logically centralized service which kept all information about inter-systems refer-
ences, and [Jull and Jul, 1992] uses a global comprehensive tracing algorithm. Both of them do not scale well because they require global synchronization. [Neuman, 1992] uses a local lower bound instead of a global lower bound, that is, the algorithm sacrifices safety for performance. [Lang et al., 1992] introduced tracing within a group instead of global systems to avoid global synchronization. However, it cannot collect distributed cyclic garbage completely. [Fuchs, 1995] suggested a backtracing mechanism, but this mechanism assumed that all objects maintain backward references, which is not scalable. [Maheshwari & Liskov, 1997] suggested an enhanced backtracing mechanism which dynamically calculated backward references. However, the overhead of calculating and maintaining backward references is too heavy.

For the solution, we designed and implemented a practical and efficient distributed garbage collection mechanism. We have implemented a garbage collection mechanism which uses timeouts, last referenceable timestamp propagation [Neuman, 1992] and backward inquiry. The primary method used for local and remote garbage collection is timeouts, which is similar to leases in Java RMI [RMI, 1997] and pinging in DCOM [Chappell, 1996]. Each object has a Time-To-Live (TTL) and an expiration, and each link maintains an expiration time of its target object. Clients and objects which have a link to another object must send a refresh message to the target object before the expiration to guarantee that the link will remain valid. Timeouts collect acyclic garbage but not cyclic garbage.

For the cyclic garbage collection, local garbage (suspected to be garbage) is detected by last referenceable timestamp propagation (LRTS). When an object is accessed by a client, the LRTS of the object is set to the accessed time, and the LRTS (the accessed time) is propagated to all reachable objects recursively during refreshing process. There is no additional communication overhead for LRTS propagation because it is propagated with a refresh message by piggybacking. The LRTS shows the most recent time when the object was accessible by clients. Objects whose LRTSs are not more recent than a local threshold are suspects of garbage and they start backward inquiry (back-tracing) to confirm that they are garbage. Since all referencing objects send a refresh message to the suspect within the suspect's TTL, the suspect can obtain the information about backward links. Using the information backward inquiry is performed to examine if the suspect is not reachable from any live objects. The backward inquiry is performed during refreshing process, so the communication overhead is minimized.

2 The System Architecture

Our garbage collection mechanism has been implemented on the Prospero directory service [Neuman, 1992] which manages distributed information, but the definitions and assumptions do not restrict the generality of our mechanism.

A distributed object is a unit of information (e.g., an object of CORBA [Vogel and Duddy, 1997], DCOM [Chappell, 1996], Distributed OODB, web or Prospero) and may contain data and methods. Data includes attributes and links. A link consists of the target object's name and attributes of the link. Every object has a unique identifier, so a link always points to at most one object.

Objects are managed by a server which has two tasks, the directory service and the garbage collection service. Clients' requests (e.g., requesting object information, adding attributes to an object, and making a link to another object) are processed by the directory service, and garbage collection is performed by the garbage collection service. Although these must be two tasks in a single server, it is more convenient to think that there are two servers, a directory server and a garbage collection server. Clients are application programs or active objects (including root objects) which have links to objects.

![Distributed Object Model](image)

Figure 1: Distributed Object Model

In our system model, all objects are equal, and there are no special root objects. In practice, some distributed object spaces do not have special root objects which are always alive. For that reason, to collect garbage, our mechanism uses reachability from clients but not from root objects. Objects reachable from clients are alive, and unreachable objects are garbage. If there are root objects, all objects reachable from the root objects are
3 Timeouts

Information about both remote and local links are maintained by timeouts. When an object is created by a client, the client assigns a TTL (Time-To-Live) to the object. The new object also gets an expiration time which is the creation time plus the TTL. The TTL is the period for which the object is guaranteed to exist after being refreshed (receiving a refresh message). When a link is made to the object, the TTL is added to the current local time, and the resulting expiration time is stored in the object. Then, the TTL is sent to the new link, and the expiration time is calculated, which is its local time plus the TTL.

To guarantee that a link will continue to work (the target object remains valid), it must be refreshed before its expiration. This means that if a link is not refreshed before its expiration time, the target object can disappear. To reduce the overhead of garbage collection, each server performs local garbage collection periodically. During this time, the server examines the expiration times of links, and if a link is going to expire, the server refreshes the target object and the link. Therefore, all objects which have incoming links can get new expiration times before they expire.

The advantages of timeouts are: 1) it makes error recovery easy because referencing information is expired after a TTL time, that is, any wrong information is corrected after a TTL time; 2) and it is suitable for administratively decentralized distributed systems because well-behaving cooperation of all distributed systems are not required. Only cooperative links are considered as actual links.

All acyclic garbage objects eventually expire and are reclaimed. Once an object becomes unreachable from clients and other objects, it cannot be refreshed and will expire after its TTL time. Cyclic garbage objects, however, are not reclaimed by their expiration times because the objects in the same cycle will refresh each other, and they never expire. In Figure 1, when the client A disappeared and does not access the objects any more, all objects, w, x, y and z become garbage. The object w cannot be refreshed and expires soon. However, cyclic garbage objects, x, y and z cannot be reclaimed because the objects in the same cycle will refresh each other, and they never expire. The LRTS (Last Referenceable TimeStamp) propagation and backward inquiry are used to collect cyclic garbage.

4 LRTS Propagation

Each object maintains an LRTS (Last Referenceable TimeStamp) which shows the most recent time when the object was accessible by clients. When an object is accessed by a client, the LRTS of the object is set to the accessed time, and the LRTS (the accessed time) is propagated to all reachable objects recursively when links are refreshed. All reachable objects from the accessed object will get new LRTSs eventually. The LRTSs of inaccessible distributed cyclic garbage, however, will stabilize at a value that is less than or equal to the time at which the cycle became inaccessible. Objects whose LRTSs are not more recent than a local threshold are local garbage. The basic idea of LRTS was suggested by [Neuman, 1992], and the idea has been expanded here.

In Figure 1, when the client A accesses the object w, then the object w gets a new LRTS and the LRTS is propagated to the subsequent objects, x, y and z, during a refreshing process. If the object w deletes its link to the object x, then the objects x, y and z become cyclic garbage, and they cannot get new LRTSs from the object w. The LRTSs of inaccessible objects including cyclic garbage (x, y, z), will stabilize at a value that is less than or equal to the time at which the cycle became inaccessible.

If there is a root object, the local threshold is LRTS of the root object allowing for time for the LRTS to propagate. If there is no root object, the local threshold is selected by clients or system administrators based on type and number of clients, and frequency of clients’ access. For schools, one might expect that all enrolled students will access their objects during a semester, so the local
threshold can be 5 months\(^2\). This means that we can assume that all objects themselves or their ancestors will be accessed within 5 months. The local threshold affects only cyclic garbage collection but not acyclic garbage collection.

Since each system selects its own local threshold, each has a different threshold. Moreover, it is possible that LRTSs might not arrive at target objects on time because they are sent only during the link refreshing process. Therefore, before local garbage is reclaimed, it should be confirmed to be garbage. For this confirmation, back-tracing is performed by backward inquiry.

5 Backward Inquiry

The basic idea of backward inquiry is that, before a local garbage object (a suspect) is reclaimed, all referencing objects are examined to see whether they are garbage or not\(^3\). If all referencing objects are garbage, then the object is also garbage and can be reclaimed. The best way to check referencing objects is to ask them directly to examine themselves whether they are garbage or not, and to reply with results.

Since referencing objects refresh their links and target objects before their expiration times, objects can get refresh messages from all referencing objects. This means that objects can get information about all referencing objects (i.e., an implicit reference list) within their TTL times, even though they do not have an explicit reference list.

In Figure 1, the object \(w\) refreshes the object \(x\) before the object \(x\) expires. So, the object \(x\) knows that the object \(w\) has a reference to \(x\). When the object \(x\) became a local garbage, then the object \(x\) can ask the object \(w\) to examine whether the object \(w\) is garbage or not. Since local garbage objects continue to refresh their target objects, the object \(x\) does not know whether the object \(w\) is alive or not. If the object \(w\) is garbage, then the object \(x\) is also garbage. However, if the object \(w\) is alive, then the object \(x\) also alive.

If there is a cycle, the request might be forwarded without stopping. To avoid this unbounded looping in a cycle, objects are marked before sending a request. If an object is already marked and gets the same request again, the object can detect a cycle.

The benefits of backward inquiry are: 1) backward inquiry (back-tracing) is started from objects suspected to be garbage and only suspects take part in the back-tracing; 2) backward inquiry is performed backward, so it is easy to find the time when synchronization is finished; and 3) the incremental overhead of backward inquiry is low because messages necessary for the inquiry are bundled with the messages used for refreshing. The proof of correctness of the backward inquiry is attached at the end of this paper.

5.1 A simple example

Figure 2: Backward inquiry for a simple cycle

Figure 2 shows a simple example of backward inquiry. In this example, we assume that only one object starts backward inquiry. There is a simple cycle which consists of three local garbage objects, \(a\), \(b\) and \(c\). When \(c\) starts backward inquiry, a new AYG (Are-You-Garbage) message is created, and the name of starting object, \(c\), is assigned to the AYG message. Only the original starting object can assign its name on the AYG message. The AYG (c) message is added to its processing list. The processing list is introduced to avoid unbounded looping in cycles. When an object gets an AYG message, the object is marked with the starting object of the AYG message.

When \(b\) sends a refresh message to \(c\) (this is not shown in the Figure 2), then (1) \(c\) sends the AYG(c) message to \(a\) with the reply of the refresh message. The “c” in the AYG message is the starting object of the message. \(c\) is appended to the processing list of the object \(b\) to avoid unbound looping. When \(a\) sends a refresh message to \(b\), then (2) \(b\) sends an AYG(c) to \(a\) with the reply of the refresh message. \(c\) is appended to the processing list of the object \(a\). (3) When \(c\) sends a refresh message \(a\), \(a\) detects that there is a cycle since the AYG(c) message is from \(c\). The AYG(c) message is going backward and the

---

\(^2\)You may think that it is too long. However, all acyclic objects are collected by timeouts. Only cyclic garbage is collected after 5 months.

\(^3\)Cyclic garbage objects continue to refresh each other, so that they never expire.
refresh message is coming forward. If they meet at the same object, then a cycle is detected.

During the TTL time, a did not get any refresh message except one from the object c, so (4) a replies to b with \( R[c,b] \). "P" (Pending) means that there is no live object in the path. After receiving the reply, (5) b replies to c with \( R[c,b] \) that b is PENDING. When c receives a reply from the object b, c knows that c is garbage because the object is not reachable from any live object.

### 5.2 Starting backward inquiry

An object starts backward inquiry when the object is local garbage and should be reclaimed. A new AYG (Are-You-Garbage) message is created, and the starting object is noted in the AYG message. When an object receives an AYG message from another object, the object also starts backward inquiry for the message. The object should use the received AYG message instead of creating a new one.

A *processing list* is introduced to avoid unbounded looping in a cycle. If there is a cycle, an AYG message would otherwise traverse the network without stopping. When an AYG message visits an object for the first time, the starting object of the message is stored in the processing list. If a message whose starting object is same as previous one visits the object once more, the object detects a cycle by checking its processing list. The unbounded looping can be avoided by marking visited objects with AYG messages.

Objects refresh their links and target objects before they expire, so target objects can get refresh messages from all referencing objects before target objects’ expiration time. A refresh message contains an LRTS and a refresh type which is the garbage collection status of a referencing object. If a referencing object is garbage, the object does not send a refresh message. If a referencing object is alive, the refresh type is ALIVE, and if a referencing object is local garbage, the refresh type is LOCAL_GARBAGE. If a referencing object is performing backward inquiry, the refresh type is AYG_PROCESSING.

When an object which is performing backward inquiry receives a refresh message, the object examines the refresh message, and, if required, sends an AYG message to the refreshing object. When a refresh message type is ALIVE, then the object stops backward inquiry and becomes alive because the object is reachable from a live object and is alive. If a message type is AYG_PROCESSING, the object compares the sender with a processing list. If the sender name is already on the processing list, a cyclic is detected. By the AYG message, the object knows that there is a path from the object itself to the sender, and by the refresh message the object knows that there is a path from the sender to the object itself, which forms a cycle. If the sender name is not in the processing list, an AYG message is sent to the referencing object.

If the message type is LOCAL_GARBAGE, the object sends an AYG message to the referencing object to see if the referencing object is garbage or alive. The object receives and processes the refresh messages until receiving refresh messages from all referencing objects, that is, during its TTL time.

A *requesting list* is maintained to see if all replies to an AYG message have arrived. When an AYG message is sent to a referencing object, the referencing object name is stored in the requesting list, and the referencing object name is removed from the list when the object replies. Some objects in the requesting list can crash without replying AYG messages. When an object in the requesting list does not send a refresh message, the object is removed from the requesting list. When the requesting list becomes empty, backward inquiry finishes.

### 5.3 Multiple starting objects

Basically, each object performs backward inquiry for only one AYG message, because all AYG messages can share the result of an AYG message. For example, if the result of an AYG message is GLOBAL_GARBAGE, we know that the result of the rest of messages will be GLOBAL_GARBAGE. However, if more than two objects start the AYG algorithm, and each object performs backward inquiry only for one AYG message, it is possible that both of them cannot finish because of a deadlock.

To avoid the deadlock and to get performance improvement, each object maintains two lists, a *processing list* and a *reply list*. Currently processing AYG messages in a processing list and other AYG messages are stored in a reply list. When the processing list is empty, an AYG message is selected from the reply list based on the priority of the AYG messages. The starting object name of AYG message is a priority. Each object has a unique identifier, so each object has a unique priority. The selected AYG message is moved to the processing list, and backward inquiry is performed for the message. When a new AYG message arrives, the priority of the message and the lowest priority message in the processing list are compared. If the new message has a higher priority, then the new message is stored in the processing list, and backward inquiry is also performed for the message to avoid a deadlock. If
the new message has lower priority, the message is just stored in the reply list and is processed later.

Figure 3 shows an example that two objects start backward inquiry at the same time. The object \( a \) starts backward inquiry when the object is local garbage and should be reclaimed. The object \( a \) starts backward inquiry and (1) sends an AYG(a) message to the object \( x \). \( x \) is reserved for \( a \), and \( a \) is appended to the processing list. (2) The AYG(a) message is sent to \( y \) from \( x \), and \( y \) is also reserved for \( a \). The object \( b \) also starts backward inquiry, and (3) sends an AYG(b) message to \( z \). \( z \) is reserved for \( b \) and \( b \) is appended to the processing list.

When (4) AYG(b) message is sent from \( z \) to \( x \), \( b \) and \( a \) are compared because \( x \) was already reserved for \( a \). Since \( b \) is bigger than \( a \), \( b \) is stored in the processing list and continue to go forward. The AYG(b) message can go forward and until it detects a cycle (6,7).

When (5) AYG(a) message is sent from \( y \) to \( z \), the message is stored in the reply list and blocked because \( z \) was already reserved for \( b \) and \( b \) is bigger than \( a \). After \( b \) finishes backward inquiry, \( a \) continues backward inquiry.

![Figure 3: Processing list and reply list](image)

### 5.4 Processing replies

The reply of an AYG message can be ALIVE, GLOBAL_GARBAGE, or PENDING. When receiving an ALIVE reply, the object then stops backward inquiry and replies with ALIVE to all objects in the processing list and the reply list because the object is alive. The status of the object becomes alive. Otherwise, the object waits for other replies and processes them until receiving replies from all referencing objects to which the object sent AYG messages. If the object has received all replies, and if all replies are GLOBAL_GARBAGE, then the object is garbage, so it replies with GLOBAL_GARBAGE to all objects in the processing list and the reply list. The status of the object becomes garbage.

If the object has received all replies, and if more than one of replies are PENDING and rest of them are GLOBAL_GARBAGE, then the object replies with PENDING only to the sender of the AYG message. The object does not know whether or not the object itself is garbage, but it knows that it is not reachable from live objects except the starting object of the AYG message. The information is used only for the starting object of the AYG message.

If the result of an object of backward inquiry is GLOBAL_GARBAGE or ALIVE, then the object replies with the result to all senders of the AYG messages in the reply list. If the result is PENDING, then the object replies with PENDING to the sender of the AYG message, and the AYG message is removed from the processing list. If the processing list is empty, the object selects a new AYG message in the reply list by the priority of AYG messages for the next backward inquiry. The next AYG message is selected, and the message is sent to referencing objects when the object receives refresh messages.

### 5.5 A complex example

Figure 4 shows a more complex example of backward inquiry. There is a big cycle, \( (a, b, c, d) \) and a small cycle \( (a, b, d) \) is inside the big cycle.

The object \( a \) starts backward inquiry when the object is local garbage and should be reclaimed. (1) \( a \) starts backward inquiry, and sends AYG(a) to \( d \). (2) \( d \) sends AYG(a) to \( b \) and (3) sends AYG(a) to \( c \). When \( c \) sends an AYG message to \( b \), \( b \) replies PENDING since \( b \) received the same message already. \( b \) does not send its AYG message to \( a \) because the AYG(a) is from \( a \). (4) \( c \) replies to \( d \) with R[a,P], and (5) \( b \) replies with R[a,P]. Finally, (6) \( d \) replies to \( a \) with R[a,P], so \( a \) is global garbage because it is not reachable from any live objects.

### 6 Local garbage collection and OUT-REF Tables

A garbage collection server operates under two modes: a normal mode and a local garbage collection mode. During the normal mode, garbage collection servers process incoming refresh messages and AYG messages. At a specified interval the garbage collection server switches to a local garbage collection mode, and performs a local garbage collection algorithm. The next local garbage collection time is scheduled by the server before starting the cur-
1) AYG(a)  
2) AYG(a)  
3) AYG(a)  
4) R[a,P]  
5) R[a,P]  
6) R[a,P]  

Processing List={a}  
Processing List={a}  
Processing List={a}  

Detect a cycle  
Detect a cycle  
Detect a cycle  

Figure 4: backward inquiry algorithm for complex cycles

During the local garbage collection time, the garbage collection server sends refresh messages and performs reclamation (including backward inquiry) if needed. First, the server visits all objects in the system, and examines the expiration times of links. If a link which will expire before the next local garbage collection time is found, the link and its target object are refreshed. After refreshing is finished, the server visits all objects in the systems once more to reclaim global garbage.

The overhead of sending a refresh message to a remote system is heavy because network communication is required. To reduce the number of refresh messages, an OUT-REF table is introduced. During refreshing, if a garbage collection server finds a remote link which is going to expire before the next local garbage collection time, the server stores a refresh message in the OUT-REF table instead of sending the message directly. At the end of refreshing, the refresh messages are sent by batch, that is, messages whose destinations are the same host are gathered together on a message and the message is sent. The next local garbage collection time is also sent with the message, and the time is stored in the IN-REF table of the target system to provide fault-tolerant refreshing.

The frequency of local garbage collection is selected by system administrators. If a system has enough storage and needs a low overhead garbage collection algorithm, the local garbage collection algorithm can be performed infrequently. If a system has enough computing power and it wants to collect garbage objects as soon as possible, then the local garbage collection algorithm can be performed frequently. However, it is possible that TTLs of some links are less than a local garbage collection interval. For the solution, garbage collection servers maintain a list of the links whose TTLs are less than the interval, and send refresh messages before the expiration times regardless of a local garbage collection schedule.

7 Fault-tolerant features

In our mechanism, timeouts are used to distinguish live objects from garbage objects, but, due to network and server failures, refresh messages might not arrive at target objects on time. This means that live objects can be reclaimed as garbage because of the failures. For the solution, the IN-REF table is introduced.

Each host manages an IN-REF table, and the table contains information about remote hosts which have references to the local objects. The table contains 3 tuples: remote host name, previous refreshing time, and next refreshing time. The next refreshing time is the scheduled time when the remote server is going to send refresh messages. When a remote server makes a link to a local object or sends a refresh message at the first time, the remote host name is registered in the IN-REF table.

Before starting reclamation, the garbage collection server examines the next refreshing times in the IN-REF table to check whether refresh messages from all hosts have arrived. If all next refreshing times in the IN-REF table are greater than the current time, that is, if all refresh messages have arrived, garbage collection is performed normally.

If any next refreshing time is not more recent than a current time, that is, if any refresh message has not arrived yet, the garbage collection server requests a refresh message explicitly. If some remote hosts cannot send refresh messages because of server or network failures, reclamation is performed based on the oldest next refreshing time of the remote hosts that have not sent refresh messages yet, because the recent time when the status of all objects were correct is the oldest next refreshing time.

If a remote host notifies that it does not have references to local objects, or if a remote host has been unreachable for a long time, the remote host name is removed from the IN-REF table.

8 Automatic Control of Garbage Collection Frequency

Objects have various lifetimes. Some stable objects live for a long time; other temporary objects live only for a short time. Objects have a tendency that many of
them are short-lived, but objects live for a long time once they have survived for more than some period of time [Lieberman and Hewitt, 1983; Ungar and Jackson, 1988]. This means that the possibility of becoming garbage for a newly created object is higher than the possibility for an old one.

Baker et. al, 1991] measured life times of files in Sprite distributed file system, and the measurements showed that most files have short lifetimes. [Lieberman and Hewitt, 1983] suggested a garbage collection algorithm based on the lifetimes of objects. The basic idea was that the frequency of garbage collection was selected according to the age of objects. For younger objects, garbage collection was performed more frequently. [Ungar and Jackson, 1988] used tenuring policies for garbage collection algorithm to improve performance. Once an object has survived for more than some threshold, the object gets tenure, and the garbage collection algorithm is performed very infrequently on tenured objects.

In timeouts, the TTL of an object is the period during which the object is guaranteed to exist after it is refreshed. For links that are infrequently referenced, parent objects should refresh the links and target objects within their TTL period, so the frequency of garbage collection is decided by the TTL of a target object. It is hard to assign an appropriate TTL to an object. The shorter the TTL, the more refreshments are required, so the overhead of refreshment becomes heavy. On the other hand, the longer the TTL, the more time is needed to reload the object if the object becomes garbage, and storage will be wasted for a long time.

To solve this problem, we use garbage collection based on the lifetime and number of incoming links of an object. As [Lieberman and Hewitt, 1983][Baker et. al, 1991] showed, the possibility of becoming garbage for a newly created object is higher than the possibility for an old one, so garbage collection should be performed more frequently for newly created objects and less for old objects. In our algorithm, the TTL of an object starts from a small number and is increased when the object is accessed or receives a new last referenceable timestamp. The number of incoming links also correlates with the probability of becoming garbage. Objects which are referenced by many other objects will live for a long time. The TTL of an object is also increased when another object makes a link to the object.

The TTL is increased up to a maximum, which is assigned by the owner of the object. This algorithm also reduces network communication overhead because the TTL of an object will be increased during local operation.

When a remote object makes a link to a local object, the TTL of the local object becomes large enough to avoid frequent remote refreshing.

9 Performance

This section describes the result of the performance evaluation. In our system, objects are managed by a server which has two tasks, a directory server and a garbage collector. Since the directory server should work without disruption while the garbage collector is working, the task of the directory server has a higher priority than the task of the garbage collector.

We developed various configuration of distributed object spaces, and measured the overhead and performance. We measured three different performance aspects of our mechanism: 1) the performance degradation of the directory server caused by the garbage collector, 2) the overhead of the network communication, and 3) garbage collection latency times of acyclic garbage and cyclic garbage. For the measurement, Prospero servers are installed on 36 Sun workstations (4 Sun Ultra-2, 13 Sun Ultra-1, 9 Sun SPARCstation-20, 5 Sun SPARCstation-10, 3 SPARCstation-5, and 2 Sun SPARCsistem-600) which are scattered over 6 Ethernet.

9.1 The overhead of local GC

A garbage collector has two modes: a normal mode and a local garbage collection mode. During the normal mode, the garbage collector processes incoming refresh messages and AYG messages. At a specified interval the garbage collector switches to a local garbage collection mode, and performs local garbage collection algorithms step by step. The garbage collector visits all objects in the system and refreshes their links and target objects including remote target objects. After the refreshing process, it visits all objects in the system once again to examine their expiration times and last referenceable timestamps. If they are expired, they are reclaimed as garbage, and if their last referenceable timestamps are not more recent than a local threshold, they start backward inquiry to check whether they are garbage or not.

First, we measured the duration of local garbage collection using the object spaces configuration shown in the Figure 5. The object spaces are tree structure, and each object has 10 children if it is not a leaf. To simplify the evaluation, we do not add remote links on the object spaces. The Figure 6 shows the duration of local garbage collection. The x-axis shows the number of objects and
links in the spaces. If there are \( n \) objects in the spaces, there are also \( n-1 \) links since the spaces are tree structure. The y-axis shows the duration of local garbage collection time. The lower line shows the duration of local garbage collection when no links are about to expire and therefore no links are refreshed. The higher line shows the duration of local garbage collection when all links are about to expire and therefore target objects are refreshed. The local garbage collection time increases linearly in proportion to the number of objects and links to be refreshed.

![Figure 5: Object Spaces Configuration](image)

Second, we measured the performance degradation of the directory server caused by the garbage collector. In the Figure 7 the x-axis shows the size of objects and the y-axis shows the object retrieving time from a remote client. The line marked as “NO GC” indicates that the server does not support a garbage collection service. The line marked as “GC-Normal” indicates that the server supports a garbage collection service and the garbage collector is on normal mode. During the normal mode, when a client accesses an object, the garbage collector modifies the LRTS (Last Referenceable Timestamps) and the expiration time of the object, and stores the object. Since the garbage collector uses a caching mechanism, there is only a little garbage collection overhead during normal garbage collection mode. The line marked as “GC-Local-GC” indicates that the server supports a garbage collection service and the garbage collection server is on local garbage collection mode. There is about 14% overhead during local garbage collection time. Since the duration of local garbage collection mode is very short in comparison with that of normal mode, the overall overhead of garbage collection is very low.

![Figure 6: Duration of Local Garbage Collection Time](image)

![Figure 7: The overhead of the garbage collector](image)

### 9.2 Communication overhead

Our mechanism contains three operations which require network communication. They are refreshing, LRTS propagation, and backward inquiry. Table 1 summarizes the overhead of the garbage collection operations for remote objects in terms of the number of messages.

<table>
<thead>
<tr>
<th>Operation</th>
<th># of messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>refreshing a remote host</td>
<td>2</td>
</tr>
<tr>
<td>propagating LRTSs</td>
<td>0</td>
</tr>
<tr>
<td>sending backward inquiry</td>
<td>0</td>
</tr>
<tr>
<td>replying backward inquiry</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: The overhead of garbage collection operations
The OUT-REF table and the IN-REF table are introduced to reduce the remote refreshing overhead. When the garbage collector meets a remote link during the refreshing process, a refresh message is stored on the OUT-REF table instead of sending the message directly. After finishing the refreshing process, the garbage collector gathers messages whose destination servers are same and sends them by batch. Therefore, one refresh message and one reply message are required for each remote host regardless of the number of remote links. The LRTSs are propagated with refresh messages by piggybacking, so no additional message is needed. AYG messages are also sent with the replies of refresh messages, but one reply message for the result of backward inquiry is required.

![Figure 8: The remote objects refreshing overhead](image)

![Figure 9: The remote hosts refreshing overhead](image)

We measured remote objects refreshing overhead and remote hosts refreshing overhead. For the remote refreshing, the garbage collector is configured to send a refresh message every 0.5 second. In the Figure 8, the x-axis shows the number of remote links to a same remote host and the y-axis shows the object retrieving time from a remote client. Since the number of refresh message is only one regardless of the remote links, the refreshing process does not cause network congestion, and clients’ object reading time is stable. In the Figure 9, the x-axis shows the number of remote hosts to which local objects have links, and the y-axis shows the object retrieving time from a remote client. Since the garbage collector sends a refresh message every 0.5 second, the refreshing process does not cause network congestion, and clients’ object reading time is stable. If a host does not get a refresh message from a referencing host on time, the host explicitly ask the referencing host to send a message.

### 9.3 Garbage collection latency

To measure the latency times of acyclic and cyclic garbage, the Prospero servers and object spaces are configured as Figure 10 and Figure 12. Each object is stored on a different host, and each garbage collector is configured as: Local GC Cycle = 1 min; TTL = 3 min; Last Referenceable Threshold = 5 min; and Network Grace Period = 3 min. Each garbage collection process was run 10 times and the results were averaged.

Because refresh and AYG messages are propagated only during local garbage collection time, garbage is collected after some latency times which depends on the TTL and Local GC Cycle. If they are small, the garbage collection latency times decrease but the performance of servers is also degraded.

![Figure 10: Collection of Acyclic Garbage](image)

In Figure 10, after \(obj_{i}\) becomes unreachable from clients, it will be expired after its TTL plus Network Grace Period. \(obj_{j}\) cannot be reclaimed until \(obj_{i}\) is reclaimed because \(obj_{i}\) continues to refresh \(obj_{j}\). All subsequent objects, \(obj_{j+1}\) cannot be reclaimed until \(obj_{j}\) is
reclaimed. When \( obj_i \) is reclaimed, \( obj_{i+1} \) cannot get refreshed and is reclaimed as garbage after TTL plus Network Grace Period. However, if \( obj_{i+1} \) has already started backward inquiry, it will be reclaimed just after Local GC Cycle/2. The latency times to collect \( obj_i \) whose maximum depth in a chain is \( i \) is as follows:

\[
\text{TTL} + \text{Network Grace Period} + (\text{Local GC Cycle}/2) \times (i-1) + \text{Last Referenceable Threshold}
\]

Figure 11 shows the measurement of collection latency times of an acyclic garbage chain. \( Obj_1 \) and \( Obj_2 \) are reclaimed by timeouts (each took \( \text{TTL} + \text{Network Grace Period}=6 \text{ min} \)) and others are reclaimed by backward inquiry (each took \( \text{Local GC Cycle}/2=0.5 \text{ min} \)).

\[\text{AYG Propagation}(n) = \text{Last Referenceable Threshold} + (\text{TTL} - 1.5 \times \text{Local GC Cycle}) \times n.\]

and the PENDING result should return to \( obj_n \), and the latency is:

\[\text{AYG Reply}(n) = (n-1) \times \text{Local GC Cycle} / 2.\]

So, the latency time to break a garbage cycle is \( \text{AYG Propagation}(n) + \text{AYG Reply}(n) \). Figure 13 shows the result of measurement about latency time of cyclic garbage. We measured the latency times of garbage cycles whose size is 3, 6, ..., 33, 36.

\[\text{AYG Propagation}(n) + \text{AYG Reply}(n)\]

The result of the measurement shows that the latency time to collect acyclic and cyclic garbage increase linearly in proportion to the length of the garbage chain. The reasons why there are differences between measurement and theoretical latency are: 1) all clocks are not synchronized perfectly and 2) the starting time of local garbage collection on each server is not synchronized.

10 Discussion

The evaluation of our garbage collection mechanism is based on the following criteria:

**Safety:** Objects reachable from clients are not reclaimed as long as they are referenced by clients which try to refresh target objects. Network fault tolerance is provided by IN-REF tables. In our mechanism, only well-behaving links are considered as actual links.

**Liveness:** All garbage including distributed cyclic garbage is reclaimed completely by timeouts, last referenceable timestamp propagation, and backward inquiry.

Figure 12: Collection of Cyclic Garbage

Cyclic garbage objects such as objects in Figure 12 are collected by backward inquiry. In Figure 12, we gave \( obj_n \) the highest priority so that the object can finish backward inquiry first. To detect a cycle, an AYG message should be propagated to all objects in the cycle (from \( obj_n \) to \( obj_1 \)). Its latency is:

Figure 13: Collection Latency of Cyclic Garbage
Low overhead: The overhead of storage, computation, and communication is minimized. Each object does not maintain a backward reference list. Instead, each host maintains a list of referencing hosts on an IN-REF table, so not much storage is used. The information about backward references are obtained during refreshing process without additional computation. Communication overhead is also reduced by piggybacking. As Table 1 shows, the overhead of garbage collection operations in terms of the number of messages is not heavy.

Scalability: The number of messages which are required for garbage collection increase linearly in proportion to the number of links, and the latency time to collect garbage increases linearly in proportion to the length of the garbage chain. Our mechanism does not require global synchronization of all distributed systems, so it is suitable to large scale distributed systems.

Fault tolerance: By introducing IN-REF tables, safety and liveness are supported despite server and network failures under the assumption that the failures are recovered within a finite period of time (e.g., one month). IN-REF tables maintain a list of referencing hosts, so garbage collection servers can examine whether the remote serves are alive. If a remote server has crashed, the garbage collection is performed based on the previous time when the remote server was alive. The garbage collection process uses pair-wise communication, so even when some servers are unavailable, others will be able to move forward.

11 Related Work

Many distributed garbage collection algorithms have been suggested, and those algorithms fall into four categories: reference counting, reference listing, tracing, and migration.

Reference counting is simple and scalable, but the algorithm is not suitable for distributed systems because: 1) it cannot collect cyclic garbage objects, and 2) it is impossible to maintain the referencing link counts correctly. Since objects can be removed in a loosely synchronized distributed system without deleting their links (e.g., during a server crash), the link count of an object can be greater than zero, even though no other objects are referencing the object. CORBA [Vogel and Duddy, 1997] and DCOM [Chappell, 1996] use reference counting and they have the above problems.

[Bevan, 1987] suggested a weighted reference counting algorithm to avoid the problems caused by extra messages and race conditions. Each object maintains a reference count, and each reference to the object has a weight. When a reference is duplicated, the weight of the reference is halved and the remaining half is sent to the next reference. When a reference to the object is deleted, the reference count of the object is reduced by the weight of the deleted reference. When the reference count of the object reaches zero, the object is reclaimed as garbage. Each reference has a virtually unique weight, so potential race conditions are avoided. The shortcomings of this algorithm is the limited number of references and the above two drawbacks of reference counting.

[Piquer, 1991] suggested indirect reference counting to solve the problem of weighted reference counting. The algorithm localizes the creation and the duplication of references. For garbage collection, the algorithm maintains an inverted tree representing the diffusion tree of the references throughout the systems. The object itself is the root of the tree, and when a reference is created or duplicated, a node is added as a child of the creator. Each node keeps one pointer to its parent and a counter with the number of children. The tree is used only for garbage collection. The direct pointer is also created for accessing the target object. When a reference is deleted, the corresponding node is deleted from the tree only if it was a leaf. If not, the node is deleted when it becomes a leaf. When there is only a root node in the tree, the object is garbage. The shortcomings of the algorithm are the overhead of management of the inverted tree and two drawbacks of reference counting which were mentioned before.

Reference listing [Shapiro et al., 1990; Birrell et al., 1993] was introduced to solve the problem of reference counting. Instead of storing link counts only, each object keeps some information about referencing objects in its reference list, so that garbage collection servers can periodically examine whether the object is actually referenced by objects in its reference list. This algorithm works well in cases where there are only a few incoming links. The storage and management overhead for the reference list, however, is heavy, when a large number of incoming links exist. [Shapiro et al., 1990] uses migration to collect cyclic garbage, but [Birrell et al., 1993] does not collect cyclic garbage.

In tracing algorithms [Juul and Jul, 1992; Lang et al., 1992], each system starts marking all of the accessible lo-
object and their descendants are also marked. When a system gets all information about the remote incoming references, objects are examined for marks. Unmarked objects are reclaimed as garbage because they are unreachable from the root objects in the network. In order to see if all systems finish the local marking and the exchange of the reference information, a distributed termination detection algorithm or a centralized server is used. These algorithms can collect distributed cyclic garbage. The overhead of the termination detection, however, is extremely heavy, and in large scale distributed systems, the centralized server becomes a bottleneck.

In [Lang et al., 1992], both reference counting and tracing is used. Reference counting is a basic algorithm, and tracing is used to collect cyclic garbage. To avoid global synchronization, this algorithm introduced tracing within a group. Before starting tracing, group negotiation is performed to organize a group, and the tracing is performed only within that group. Cyclic garbage within the group is detected and reclaimed. After finishing the tracing, the group is disbanded. This algorithm is fault tolerant and does not need global synchronization, but cannot collect garbage completely.

[Full and Jul, 1992] uses a global comprehensive tracing algorithm. Their goal was to collect all garbage in the entire distributed systems. The local garbage collector performs local tracing, and during each global garbage collection cycle, the global garbage collector gets information about references from remote systems by cooperating with other global collectors. To determine when the global marking is finished, a distributed termination detection algorithm is used. This algorithm does not scale well.

Timestamp propagation uses timestamps instead of marks. Periodically, each system propagates timestamps to reachable objects from its root, and objects whose timestamps are less than a global lower bound are regarded as garbage. These algorithms do not need global synchronization, but the overhead of finding a global lower bound is extremely heavy. [Hughes, 1985] uses a distributed termination detection algorithm to find the global lower bound, but the algorithm was very expensive and did not scale well. [Ladin and Liskov, 1992] uses a logically centralized service which kept all information about inter-systems references, but this algorithm did not scale well either. [Neuman, 1992] uses timeouts and last referenceable timestamps for garbage collection. Objects whose last referenceable timestamp is not more recent than a lower bound are garbage, and they are reclaimed. He uses a local lower bound instead of a global lower bound, that is, the algorithm sacrifices safety for performance. DCOM [Chappell, 1996] uses pinging to examine clients’ status. Clients periodically send a pinging message to the referencing object. If an object has not got any timestamps from a client for a sufficient time interval, the client is assumed to have died.

[Fuchs, 1995] suggested a back-tracing mechanism, but this mechanism assumed that all objects maintain backward references. Since distributed objects usually do not maintain backward references, the mechanism is not practical. It is also unscalable to maintain backward references. [Maheshwari & Liskov, 1997] suggested an enhanced back-tracing mechanism. Each system maintains two tables, inrefs and outrefs which contain the information about remote references. The information about backward references are calculated using these tables and local tracing. The overhead of calculating backward references is heavy, and it is very hard to preserve safety and completeness in the presence of concurrent mutators and garbage collectors.

Migration [Bishop, 1977; Shapiro et al., 1990; Gupta, 1993] was also suggested to collect cyclic garbage. In this algorithm, all objects on a garbage cycle are migrated to a single system, and are collected during local garbage collection. This approach is not suitable for large scale distributed systems, because some objects may not be migrated, and some systems may not allow remote objects to be migrated.

12 Conclusion

We have described a practical and efficient distributed garbage collection mechanism which uses timeouts, last referenceable timestamp propagation, and backward inquiry.

Timeouts are suitable for large scale distributed systems because they make error recovery easy and do not require well-behaving cooperation of all distributed systems. Only cooperative references are considered as actual references for garbage collection. Last referenceable timestamp propagation and backward inquiry collect distributed cyclic garbage safely and completely without global synchronization or backward references. Since messages necessary for cyclic garbage collection are bundled with the messages used for refreshing, overhead is minimized.

The mechanism is fault-tolerant, so safety and liveness
are supported despite server and network failures. Each server maintains information about remote servers which have references to local objects, and uses this information to set a threshold for expiration of objects that have not been refreshed. By maintaining the last and expected next refreshing time from each server, garbage collection can be performed even when some of servers are unavailable.

References


APPENDIX: Proof of Correctness

The correctness of backward inquiry is proven by introducing an AYG graph which is defined as follows.

**Definition 1** An AYG graph is a directed acyclic graph which is represented by $G(V, E)$ where $V$ is a set of objects, and $E$ is a set of links. $G(V, E)$ has one exit object, $v_e$, which is reachable from all $v_i \in V$. $v_e$ does not have any outgoing links, and each $v_j \in V - \{v_e\}$ has only one outgoing link.

**Proposition 1** The result of backward inquiry for an AYG graph $G(V, E)$ with $v_e$ and $V^L = \{v_i \in V : v_i \text{ is alive}\}$ is correct. That is, if $V^L \neq \phi$, then ALIVE replies are returned to $v_e$. Otherwise, $v_e$ is global garbage.

**Proof.** The transpose of a directed graph $G(V, E)$ is the graph $G^T(V, E^T)$, where $E^T = \{(v_i, v_j) \in V \times V : (v_j, v_i) \in E\}$. Thus, $G^T(V, E^T)$ is $G(V, E)$ with all its edges reversed. The graph $G^T(V, E^T)$ is a tree whose root is $v_e$. The AYG algorithm traverses the tree $G^T(V, E^T)$ starting from $v_e$, and searches for live objects. The path of the AYG message is a subgraph of a tree $G^T(V, E^T)$. When the AYG message reaches an object which is alive or does not have outgoing links, the message (the reply) returns to the exit object $v_e$ using the path in $G(V, E)$. Therefore, backward inquiry for an AYG graph always terminates. Any object $v_i \in V^L$ returns an ALIVE reply to $v_e$; thus $v_e$ is global garbage iff $V^L = \phi$.

**Proposition 2** When backward inquiry is performed for a graph $G(V, E)$ with an AYG starting object $v_e$ and a set of live objects $V^L$, the path of the AYG message forms an AYG graph.

**Proof.** The path of the AYG message, $G'(V', E')$, can be generated by the following algorithm, when backward inquiry is performed for the graph $G(V, E)$ with an AYG starting object $v_e$ and a set of live objects $V^L$.

1. $V' - \phi, E' - \phi, V_t - V$
2. Choose $v_i \in V_t$ s.t. $(v_i, v_j) \text{ for some } v_j \in V'$
3. $V_t - V_t - \{v_i\}$
4. if $v_i \notin V^L$ then $V' - V' \cup \{v_i\}, E' - E' \cup \{(v_i, v_j)\}$; else do nothing
5. go back to 2 until $V_t = \phi$

By induction, it is proved that the graph $G'(V', E')$ which is generated by the above algorithm is an AYG graph.

Let’s assume that $G^n(V_n, E_n)$ is an AYG graph. In line 4 of the above algorithm, if $v_i \notin V^L$, then $v_i$ and $(v_i, v_j)$ is added to $G^{n+1}(V_{n+1}, E_{n+1})$. $v_i$ has only one outgoing link $(v_i, v_j)$, and $v_i$ is reachable to $v_e$ because $v_i$ is reachable to $v_j$ which is reachable to $v_e$. Therefore, $G^{n+1}(V_{n+1}, E_{n+1})$ is an AYG graph where $V_{n+1} = V_n \cup \{v_i\}$ and $E_{n+1} = E_n \cup \{e_i\}$.

**Proposition 3** Backward inquiry for any graph $G(V, E)$ with an AYG starting object, $v_e \in V$, is correct.

**Proof.** An AYG message is propagated from $v_e$ to objects in $V - \{v_e\}$. By proposition 2, the path of the message forms an AYG graph. By proposition 1, AYG backward inquiry is correct for that path. Hence, the algorithm is correct for the entire graph $G(V, E)$. 

The $G^n(v_e, \phi)$ is an AYG graph, which is trivial.