Cyclic Distributed Garbage Collection Without Global Synchronization

Preliminary Examination Report
April 28, 1995

Ph.D. Candidate: Gustavo Rodriguez-Rivera,

Comittee Members:
Vincent Russo (Advisor)
Douglas Comer,
Tony Hosking,
Michal Young,
Vernon Rego

Computer Science Department
Purdue University West Lafayette, IN 47907

Abstract

In this report we show an algorithm for cyclic distributed garbage collection that does not require global coordination and is tolerant to message, network, and process failures. Most of the cyclic distributed garbage collectors use mark-and-sweep to find the transitive closure of objects that are reached from the global roots. The main disadvantage of mark-and-sweep is that it requires global coordination to know when the mark-phase has ended. Our algorithm takes a more radical approach and uses back-tracing instead of mark-and-sweep to eliminate the need for global synchronization. Starting from an object that is suspected to be garbage, the algorithm recursively traces back the references that point to the suspect to find the transitive closure of objects that reach the suspect. If this closure does not contain any root, then the objects in the closure are collected. Local garbage collectors have to be modified to provide back-references for local proxies (representation of remote objects) and local principals (exported objects). One interesting feature of the algorithm is that all the state necessary for garbage collecting one closure is stored in a single message. This allows the algorithm not to require global synchronization and to make it scalable for usage in very large distributed systems. The cost of the algorithm is minimum. It only adds small overhead in the local garbage collector to compute the back-references plus some extra space in the process to store them. We show how this algorithm is implemented and a proof of its correctness.

1 Introduction

1.1 Why Distributed Garbage Collection?

The same reasons that make automatic garbage collection useful for uniprocessor systems are also valid for distributed systems as well. Using explicit deallocation instead of garbage collection leads to program errors such as memory leaks and dangling references. Moreover, there is one reason that makes distributed garbage collection important in a distributed environment: A program that does not use garbage collection and uses explicit deallocation needs to keep track of object reachability to know when to deallocate an object. Keeping track of object reachability may not be difficult in a single program. However, in a large distributed system where different programs written by different programmers exchange references to objects, explicit object deallocation requires the programmers to *explicitly* agree on when to deallocate an object. This approach is impractical and error prone as the number of programs increases (not scalable).

Distributed garbage collection is a difficult problem. Besides collecting *all* the garbage of a system promptly, a distributed garbage collector should deal with lost and duplicated messages, process crashes, long lasting network failures, and problems of scalability. It is not a surprise that many algorithms exist. Each of these algorithms solves some problems but leaves other problems unsolved. For example there are algorithms that collect cycles of garbage but require global coordination of some kind, making the algorithm unscalable. Other algorithms are efficient passing references around without telling the owner, but are not fault-tolerant. Other algorithms are scalable and deal with process crashing and lost and duplicated messages, but are not cyclic.

The algorithm we propose here is an extension of reference listing algorithms such as [BEN$^+$93], and [SDP92]. These algorithms are already scalable and tolerant to lost and duplicated messages, but can not collect cyclic data. The algorithm we describe uses reference listing plus additional mechanisms that allow the collection of garbage cycles without the need of global synchronization. It was implemented as part of the Renaissance distributed system [MR93]. However, it can be used as an extension for systems that already use reference listing to collect garbage cycles.

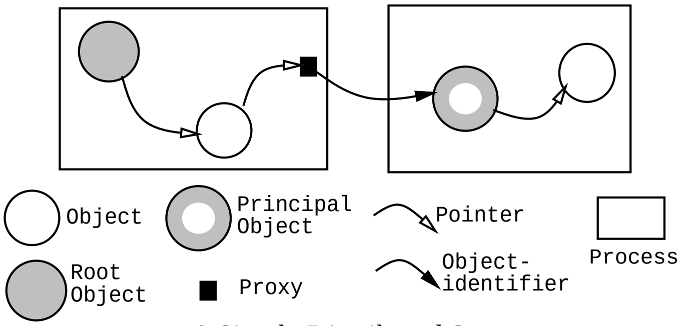
The paper is organized in the following way: First, we explain the terminology used in this paper and the basic ideas behind the algorithm. Then the formal algorithm and a proof of its correctness are described. Finally, we give an explanation of how the algorithm is fault tolerant and a list of related work.

1.2 Terminology

In Renaissance *objects* are entities that have *state* and *implementation*. The implementation of an object has a set of entry points called *methods* that allow external interaction with the object. A *host* is a piece of hardware with an operating system that provides independent *address-spaces* and execution paths or *threads*. The union of an address-space and one or more threads running on this address space is called a *process*. A process is the place where objects reside. The threads in a process execute the implementation of the objects. A *distributed system* is a set of objects that reside in one or more processes (in the same or different hosts) that interact with each other through *method invocation*. During a method invocation, *object-identifiers* can be passed or returned. If a process receives an *object-identifier* that identifies an object that resides in a process different than the recipient, then a *proxy* or local representation of the object is created in the recipient's address-space. All methods invoked on a proxy are forwarded to the object they represent called the proxy's *principal*. An object is considered *garbage* when no thread can execute the object's implementation. Every process has a special set of objects that are never garbage. This set is called the *local roots*. The union of all the local roots in the distributed system is called the *global roots*. An object A can *directly-reach* object B if: 1. A contains a pointer to B; or 2. A directly-reaches a third object C and C contains a pointer to B. An object A can *reach* object B if: 1. A directly-reaches B or B's proxy; or 2. A reaches a third object C and C directly-reaches B or B's proxy[1]. Objects that are not reachable from the global roots are garbage. This is because they can not be reached by any thread, and therefore their implementation can not be executed. A *local garbage collector* reclaims local objects that are not directly-reachable from the local roots or the local principals. *A distributed garbage collector* collects and reclaims

[1] Note that even when object A *reaches* object B, it is possible that A does not *directly-reach* B.

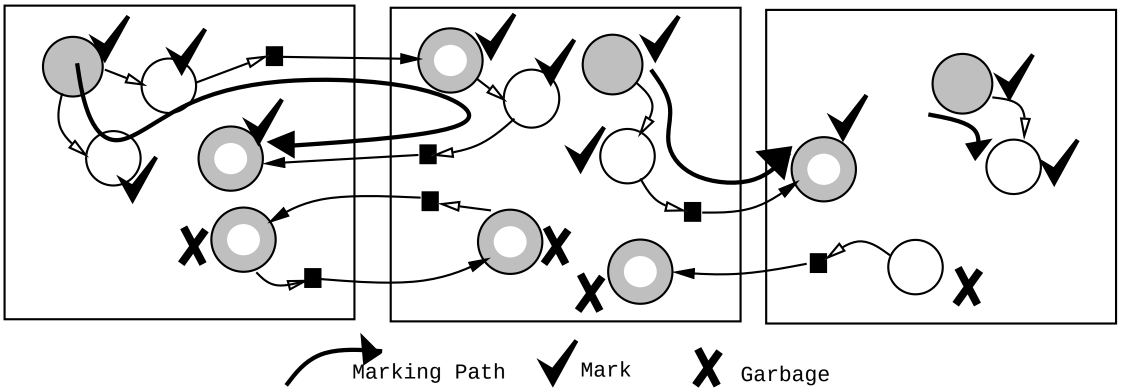
objects that are not reachable from the global-roots. The next figure shows an example of a distributed system and the symbols used to represent the different components.



A Simple Distributed System

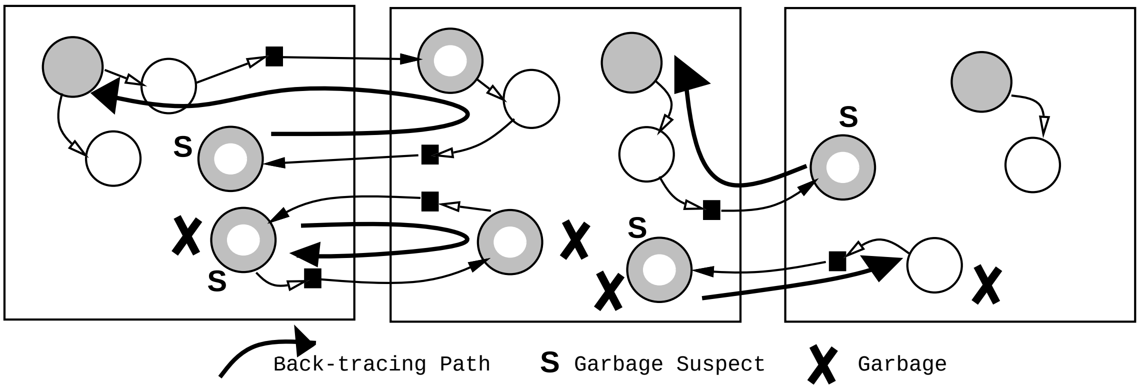
1.3 Basic Idea

The algorithm we describe here uses trial-error and back-tracing instead of mark-and-sweep. Most of the cyclic distributed garbage collectors are variants of the mark-and-sweep algorithm. They find alive objects by finding the transitive closure of objects reachable from a root set. The main disadvantage of mark-and-sweep algorithms is that some kind of global synchronization is needed. In mark-and-sweep, garbage is defined as "all the objects that are left unmarked after a mark-phase". This implies that all the processes in the system need to detect when *all* the processes have completely propagated the mark-bits to distinguish alive objects from garbage.



Mark and Sweep

Our algorithm uses trial-error and back-tracing. When an object is suspected to be garbage, the references that point to the suspect are recursively back traced. The back-tracing continues all the way until the closure of all the objects that reach the suspect is found. If this closure does not contain any root, then the objects in the closure are reclaimed. Notice that back-tracing involves *only* processes that own objects in the closure, compared to mark-and-sweep that involves *all* the objects in the system.



Back Tracing

However, back-tracing does not come for free. There are four basic problems: 1. Back-references need to be computed before doing back-tracing; 2. How to determine garbage suspects. 3. It is necessary to allow multiple back-tracings to run at the same time; and 4. Back-references may change during back-tracing.

There are two kinds of back-references: *local-back-references* and *remote-back-references*. Local-back-references connect objects in the same process. Remote-back-references connect objects in different processes. Remote-back-references are obtained by keeping track of the processes that have proxies to any of the local principals. Local-back-references are computed by the local garbage collector using auxiliary tracing. Since only proxies and principals are relevant in a distributed garbage collection, local-back-references are computed only for proxies and principals and not for internal objects. The local garbage collector takes care of the internal objects when they are no longer directly-reachable from neither principals nor roots.

The second problem is solved by considering as suspects the principals that are not longer directly-reachable from local roots. After a local garbage collection, for every principal that is no longer directly-reachable from a local root a back-tracing phase is started. In this way, the frequency of the back-tracings is related to the frequency of the local garbage collections.

The third problem is solved by keeping independent state for every back-tracing. In our implementation of the algorithm the state of a back-tracing is stored in a token message that goes from one process to the other. The message keeps a list of objects that have been back-traced and a list of objects in the forming closure pending for back-tracing. The back-tracing ends when there are no objects pending for back-tracing.

The fourth problem is solved by using sequence numbers. Every principal has a sequence number that is incremented when a method is invoked remotely or when a new proxy is created for this principal. A closure of principals that is not reached by local roots can have its back-references modified only through remote method invocations to any of its principals, or by creating new proxies to its principals. If after computing the closure any of the sequence numbers of the principals in the closure have changed, then the collection stops without collecting because the back-references may have changed. Otherwise, if the closure is complete, and it does not contain a root, and none of the sequence numbers of the principals has changed, then the principals in the closure are reclaimed. In the next section we explain in detail how the algorithm is implemented.

2 Algorithm

The algorithm is divided in two parts. In the first part the local garbage collectors compute the local-back-references. In the second part the local-back-references and remote-back-references are back-traced to find inter-process garbage. Neither part requires global synchronization and can run concurrently. Garbage is collected only when there is enough information to do so. First, we explain some basic data structures used in this algorithm, and how new remote references to objects are created and deleted. Later, we explain how the local-back-references are computed by the local garbage collector. Finally, we explain how the distributed algorithm finds inter-process garbage.

Even though the implementation of the algorithm in the Renaissance system was done in C++, the language used in this paper to illustrate the algorithm is Python [Ros94].

2.1 Process Data Structures

The algorithm needs the following data structures. For each process p there is the set *p.principals* of all local objects that are currently exported by p. Also for each process p there is the set *p.proxies* of all proxies that exist in this process. Both *p.principals* and *p.proxies* are implemented using weak pointers. This prevents the local garbage collector from considering them as part of the local roots during garbage collections.

For each principal o there is a list *o.processes* that contains a list of all the processes that have a proxy to o. *o.processes* gives the remote-back-references needed for back-tracing.

Also for each principal o there is a variable *o.sequence* that is incremented in the following cases: 1. There is a remote method invocation to o; 2. A new proxy is created for o in some other process; and 3. An object-id for o is passed back to the process that owns o through a remote method invocation. This sequence number is used to verify that a closure of principals is valid after it is formed.

A mutex variable *p.mutex* in each process prevents inconsistencies in the different data structures and is also used to stop incoming remote method invocations.

2.2 Creation and Deletion of Remote References

For the purpose of the algorithm there are only two operations on references that matter in the system: create remote reference and delete remote reference. We explain next how these operations are performed in our algorithm.

New references are created by passing arguments or as a result of a remote method invocation. In order to create a new reference to a remote object, an object-id is transmitted over the wire. If the recipient process is not the owner of the object, then the process has to create a local proxy. Notice that in order for a process to pass an *object-id* to another process, the sender process needs either to be the owner of the object or to have a proxy for that object. In any case the following invariant has to hold:

Invariant 1 If there is a proxy x for object o in process p, then p should be in *o.processes*.

In order to insure this invariant even in the presence of lost and duplicated messages we use an approach similar to the one described in [BEN+93]. Assume that process p wants to send an object-id *oid* for object o to process q, and r is the process that owns o. Also assume that p, q, and r are different processes. First we have to make sure that q is in *o.processes* before a proxy is created in q. In order to do so the following has to be done: 1. p sends to q the object-id *oid* and waits for an acknowledge. 2. Before q creates a proxy for o, it sends a registration message to r. 3. r adds q to *o.processes* and then it sends an acknowledge back to q. 4. When the acknowledge is received, q creates the proxy and then it sends an acknowledge back to p. If it happens that q and r, or p and r are the same process, or q already has a proxy for o, then some of the steps are skipped.

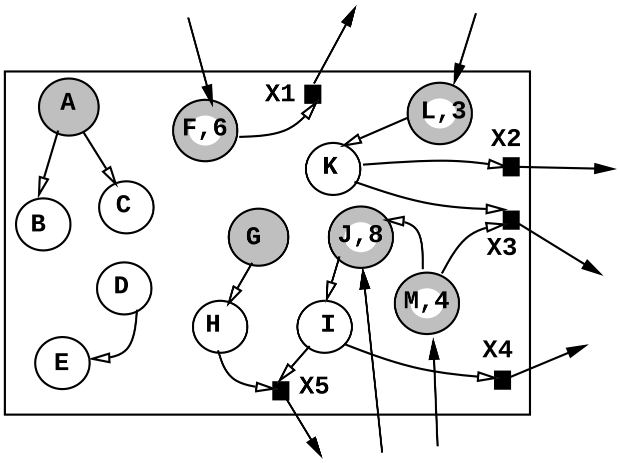
Delete reference operations are executed when proxies are removed as a result of local garbage collections.

In order to make the creation and deletion calls tolerant to delayed, lost, or duplicated messages, each operation is labeled with a sequence number. For each process p there is a variable *p.sequence* that is incremented and included in every reference operation. In the other side, a process p keeps in a table $p.recent\{q\}$ the greatest sequence number that another processes q has used in their reference operations. If p receives a register or delete message with sequence number *seq* from process q, it performs the operation only if $seq > p.recent\{q\}$. In this way reference operations are tolerant to lost and duplicated messages. Periodically each process p checks if the process q still exists (using a special server described later). If the processes has died, then it removes that entry from $p.recent\{q\}$. A more detailed description of these operations and a proof of their correctness are described in [BEN+93].

2.3 Local Garbage Collection and Local Back References

The local garbage collector has to be modified to find the local-back-references of local principals and local proxies. Local-back-references are kept for every proxy and principal z in the variable *z.roots*. This variable is the list of all the principals u that directly-reach z. In order not to include u in the local roots, u is stored as a weak pointer.

If z is directly-reachable from a local root, then the special object R is appended to *z.roots*. R is a special object that represents the local roots. We show in the next figure an example of how the data structures may look like, after a local garbage collection.




```
At the end of a local garbage
collection we have:

p.principals = [ F, J, L, M ]
p.proxies = [ X1, X2, X3, X4, X5 ]

F.roots = []
J.roots = [ M ]
L.roots = []
M.roots = []

X1.roots = [ F ]
X2.roots = [ L ]
X3.roots = [ L, M ]
X4.roots = [ J, M ]
X5.roots = [ R, J, M ]

Also D and E are collected
```

Example of a Local Garbage Collection

Note that *z.roots* is only defined for proxies and principals, and not for internal objects. The reason for this is that only principals and proxies are relevant in a distributed garbage collection. Internal objects that are reachable only by garbage principals are collected by the local garbage collector after the distributed collector reclaims these garbage principals.

We also need a flag *o.dirty* for every principal o that is cleared before every garbage collection and is set when *o.sequence* is incremented. *R.dirty* has a value of *true* always.

Now we show under which conditions a list of local-back-references *z.roots* computed during a garbage collection is *consistent* with the current state.

Invariant 2 Given a proxy or principal z, *z.roots* is *consistent* with the current state if *z.roots* does not include R and for every principal o in *z.roots*, *o.sequence* has not changed since the time *z.roots* was computed.

This holds because a proxy or principal that is not directly-reachable from a local root can have their local-back-references modified only through method invocations to principals in *z.roots*. Therefore, a variable *z.roots* is *consistent* if *z.roots* does not include R, and for every principal u in *z.roots*, u exists and *u.dirty* is equal to *false* (*u.sequence* has not changed since the time *z.roots* was computed). The following figure summarizes how the function *consistent(z)* is computed.

```
def consistent( z ):
     if R in z.roots:
          return 0
     for u in z.roots:
          if u not in p.principals or u.dirty:
               return 0
     return 1
```

Definition of consistent(z)

Now lets describe how the local garbage collection is performed. When a local garbage collection starts, the collector stops all the other threads and atomically clears the *mark-bits*. Then the collector traces the local roots and recursively marks all the objects directly-reachable from them. After that, for every proxy or principal z that is directly-reachable from the local roots, the special object R is added to *z.roots*. There is going to be a flag *o.potential-garbage* in every principal that is set to *true* if the principal is not marked at this point. Then these not marked principals are also recursively marked to prevent the collection of objects that are not reachable from local roots but are reachable from some global root. The objects that are left unmarked are reclaimed.

In a second phase, for each principal *o* that has *o.potential-garbage* set to *true*, there is going to be a *private* mark phase to determine the proxies and principals *o* directly-reaches. First all *mark-bits* are cleared and then *o* is traced. All the objects *o* directly-reaches are marked recursively. At the end of this *private* mark phase, for every proxy or principal *z* that *o* directly-reaches, *o* is added to *z.roots*.

To eliminate clearing *mark-bits* in every *private* garbage collection, it is possible to use a *mark-number* instead of a *mark-bit* for every object. The *mark-number* is set to a *mark-value* when an object is marked. An object is considered marked if its *mark-number* is equal to *mark-value*. This *mark-value* is different in every mark phase. In the Renaissance local garbage collector the *mark-number* was not difficult to implement because a *mark-byte* was already used for every object instead of a *mark-bit*. The only problem is that after running out of *mark-values* (after 255 *private* mark phases in this case), all *mark-bytes* have to be reset to *0*.

At the end of a collection all the other threads are restarted.

We have used a simple stop-the-world garbage collection algorithm here to simplify the explanation. Any other incremental garbage collection technique can be adapted to also find the local-back-references without long pauses. In the Renaissance local garbage collector we have used an incremental garbage collector based on the one described in [BDS91].

New proxies and principals are created with *z.roots* equal to *[R]*. Also new principals are created with the flag *o.dirty* set to *true*, and *o.sequence* equal to *0*.

The next figure shows how the modified local garbage collector looks like.

```
def local_gc( p ):

     p.mutex.lock()
     p.stop_other_threads()
     p.clear_mark_bits()
     p.mark_objects_reachable( p.local_roots() )

     for z in p.proxies + p.principals:
          if p.marked( z ):
               z.roots.append( R )

     for o in p.principals:
          if p.marked( o ):
               o.potential_garbage = 0
          else:
               o.potential_garbage = 1
               p.mark_objects_reachable( o )

     for o in p.all_objects():
          if not p.marked( o ):
               if o in p.proxies:
                    p.proxies.remove( o )
               p.reclaim( o )

     for o in p.principals:
          if o.potential_garbage:
               p.clear_mark_bits()
               p.mark_objects_reachable( o )
               for z in p.proxies + p.principals:
                    if p.marked( z ):
                         z.roots.append( weak(o) )

     p.create_tokens_for_potential_garbage()
     p.start_other_threads()
     p.mutex.unlock()
```

Definition of local_gc(p)

2.4 Collecting Distributed Garbage

Now that we have explained how local-back-references are computed, we explain how this information is used to collect distributed garbage. Assume that all processes that participate in a cycle of garbage have performed local garbage collection after the garbage was produced. Since by definition no thread can execute the implementation of a garbage object, then no method invocations could have been done in any of the objects that participate in the graph of garbage. This implies that every proxy or principal z in this graph has *consistent(z)* equal to *true*. At this point all the information necessary to collect this garbage graph is already available.

The main idea of the algorithm is to find the closure of principals that reach a exported object called *suspect* that is considered potential garbage. If this closure does not include any roots, then the principals that form the closure are collected. Otherwise the algorithm halts without collecting.

The algorithm has three phases. In the first phase a local garbage collector creates a new token from a principal *suspect* that is suspected to be garbage. The token holds all the state of the forming closure. This token is passed around every process that has a principal that reaches *suspect* to add the object-id of the principal in the token. If the algorithm finds out that a global root reaches any of the principals in the token, or there is not information available to say so, then the token is discarded because the graph may not be garbage. Otherwise in a second phase, after all the reachable principals are exhausted, their sequence numbers are checked to verify that there has not been any method invocations that could have changed the back-references in the closure. If any of the sequence numbers has changed, the token is discarded and the collection halts because the information in the token may not be correct anymore. Otherwise, all the principals in the closure are considered garbage. If this is the case, in a third phase a delete token is sent to every principal to reclaim the space they use.

A token message t may have the following three types depending of the phase of the algorithm the token is in.

The first token type is the triple *("FIND-CLOSURE", principals, proxies)*, where *"FIND-CLOSURE"* is a string constant that labels this type of tokens. *Principals* is a list of pairs *(principal, sequence)* that are the principals that form part of the closure, and the sequence numbers they had when they were added to the token. *Proxies* is a list of pairs *(process,principal)* that lists the proxies that are not yet part of the closure, but point to some principal in the forming closure.

The second token type is the triple *("CHECK-SEQUENCE", principals, ith-principal)*. Where *"CHECK-SEQUENCE"* is a string constant that differentiates this type of tokens. *Principals* is a list of pairs *(principal, sequence)* as described above. And *ith-principal* is an index that points to some object in *principals*.

The third type of token is *("DELETE", o)*. Where *"DELETE"* is a string constant, and o is a principal to delete in the recipient process.

The algorithm goes as follows. First after a local garbage collection, all the principals that are not directly-reachable from the local roots are considered suspects of being potential garbage and a new token is generated for each of them. The token includes the suspect and all the principals that directly-reach the suspect. It also includes all the proxies of the suspect and the principals. So, for every suspect s that is potential garbage, a new token *("FIND-CLOSURE", principals, proxies)* is generated. *Principals* is initialized to *[(s, s.sequence)]* and *Proxies* is initialized to empty. Then for every process q in *s.processes*, the pair *(q,o)* is appended to *proxies*. Also, for every pair *(u, useq)* in *s.roots*, the pair *(u, useq)* is appended to *principals*, and for every process q in *u.processes*, the pair *(q,u)* is appended to *proxies*. Then the token is sent to the first process in *proxies*. This procedure is shown next and is called at the end of the local garbage collection.

```
def create_tokens_for_potential_garbage( p ):
     for s in p.principals:
          if s.potential_garbage and p.consistent( s ):
               principals = [(s,s.sequence)]
               proxies = [ ]
               for q in s.processes:
                    proxies.append( (q, s) )
               for u in s.roots:
                    principals.append( (u,u.sequence) )
                    for q in u.processes:
                         proxies.append( (q, u) )
               t = ( FIND_CLOSURE, principals, proxies )
               p.pass_token( t, proxies[ 0 ][ 0 ])
```

Definition of create_tokens_for_potential_garbage(p)

In every process there is going to be an active daemon thread that process incoming token messages. When a token *("FIND-CLOSURE", principals, proxies)* arrives, it reads the first pair *(process-x,principal-x)* in *proxies*. This pair defines a proxy x for *principal-x* located in *process-x* (that should be the current process). The daemon adds to the token all the principals that reach x and their proxies. If x exists and *consistent(x)* is true, then for every principal o in *x.roots*, *(o,o.sequence)* is appended to *principals* (only if o is not already there). Also for every process p in *o.processes*, the pair *(p,o)* is appended to *proxies*. However, if *consistent(x)* is false, then no accurate back-references are available for x and the token is discarded. After that, if *proxies* is not empty, then the token *("FIND-CLOSURE", principals, proxies)* is passed to the next process in *proxies*.

When *proxies* is empty, the second phase starts. An empty *proxies* means that potentially all the objects in *principals* form a closure that does not include any global root. However, before concluding this, it is necessary to check that the sequence numbers have not changed in any of the principals since the time they were added to *principals*. In order to do so, the daemon creates another token *("CHECK-SEQUENCE", principals, ith-principal)*, where *ith-principal* is equal to *0*, and it sends the token to the process that owns the first object in *principals*. When a process receives the token *("CHECK-SEQUENCE",principals,ith-principal)*, it reads from *principals* the pair *(o,oseq)* that corresponds to *ith-principal* (note that the current process owns o). If *o.sequence* is different than *oseq*, then it discards the token. Otherwise it checks if *(o,oseq)* is the last pair in *principals*. If this is true, then the third phase starts. Otherwise it increases *ith-principal* and it sends the token *("CHECK-SEQUENCE",principals,ith-principal)* to the process that owns the *ith-principal* object in *principals*.

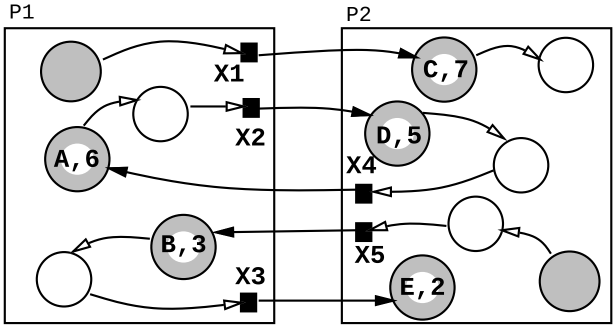
In the third phase, for every pair *(o,seq)* in *principals*, the daemon sends a token *("DELETE", o)* to the process that owns o When a daemon receives this token it reclaims the space used by o. An example of a distributed garbage collection and the daemon we have just described are shown in the following figures.

```
def receive_tokens_daemon( p ):
     while 1:
          t = p.receive_token()

          if t[ 0 ] == "FIND_CLOSURE":
               p.mutex.lock()
               (cmd, principals, proxies) = t
               (process_x, principal_x ) = proxies[ 0 ]
               if p.exists_proxy_for( principal_x )
                    x = p.proxy_for( principal_x )
                    if not p.consistent(x):
                         p.mutex.unlock()
                         continue
                    for o in x.roots:
                         for (u,useq) in principals:
                              if u == o:
                                   break
                         else
                              principals.append( (o,o.sequence) )
                              for p in o.processes:
                                   proxies.append( (p,o) )
               proxies.remove( (process_x, principal_x) )
               if len( proxies ) <> 0:
                    p.pass_token( ("FIND_CLOSURE", proxies, principals ),
                                   proxies[ 0 ][ 0 ] )
               else:
                    p.pass_token( ("CHECK_SEQUENCE", principals, 0 ),
                                   p.owner( principals[ 0 ][ 0 ] ) )
               p.mutex.unlock()

          elif t[ 0 ] == "CHECK_SEQUENCE":
               p.mutex.lock()
               (cmd, principals, ith) = t
               (o, oseq) = principals[ ith ]
               if o not in p.principals or o.sequence <> oseq:
                    p.mutex.unlock()
                    continue
               ith = ith + 1
               if ith < len( principals):
                    p.pass_token( ("CHECK_SEQUENCE", principals, ith),
                                   p.owner( principals[ ith ][ 0 ] ) )
               else:
                    for (o, oseq) in principals:
                         p.pass_token( ("DELETE", o), p.owner( o ) )
               p.mutex.unlock()

          elif t[ 0 ] == "DELETE":
               p.mutex.lock()
               (cmd, o ) = t
               p.principals.remove( o )
               p.mutex.unlock()
```

Definition of receive_tokens_daemon(p)

1. Initially proxies and principals z have z.roots = [R]. Then P1 performs local garbage collection.

```
A.roots  = []
B.roots  = []
X1.roots = [R]
X2.roots = [A]
X3.roots = [B]
```




```
C.roots  = [R]
D.roots =  [R]
E.roots  = [R]
X4.roots = [R]
X5.roots = [R]
```

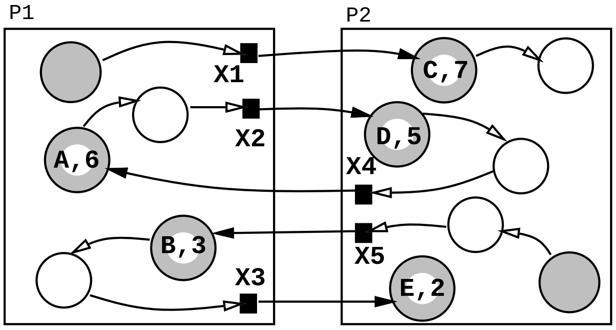
2. P2 generates token messages t1 and t2 for principals A and B that are considered garbage suspects.

```
t1 = ("FIND_CLOSURE", [(A,6)], [X4])
t2 = ("FIND_CLOSURE", [(B,3)], [X5])
```

Note: Names of proxies are used instead of their representation (process, principal) to make the example more clear.

3. t1 is passed to the owner of X4 (P2). Since R is in X4.roots, P2 discards t1.
4. t2 is also passed to the owner of X5 (P2). Since R is in X5.roots, t2 is discarded.
5. P2 performs local garbage collection

```
A.roots  = []
B.roots  = []
X1.roots = [R]
X2.roots = [A]
X3.roots = [B]
```




```
C.roots  = []
D.roots =  []
E.roots  = []
X4.roots = [D]
X5.roots = [R]
```

6. P2 generates token messages t3, t4, and t5 for principals C, D, and E that are considered garbage suspects.

```
t3 = ("FIND_CLOSURE", [(C,7)], [X1])
t4 = ("FIND_CLOSURE", [(D,5)], [X2])
t5 = ("FIND_CLOSURE", [(E,2)], [X3])
```

7. t3 is passed to the owner of X1 (P1). Since R is in X1.roots, P1 discards t3.
8. t4 is passed to the owner of X2 (P1). P1 adds A and X4 to t4. P1 also removes X2 from t4.

```
t4 = ("FIND_CLOSURE", [(D,5),(A,6)], [X4])
```

9. t4 is passed to the owner of X4 (P2). P2 tries to add principal D to t4. Since D is already there, it is not added. Then P2 removes X4 from t4. Since there are no more proxies in t4, t4 is set to the value:

```
t4 = ("CHECK-SEQUENCE", [(D,5),(A,6)], 0)
```

11. t4 is passed to the owner of D (P2). D.sequence is still equal to 5, so P2 increments the counter in t4.

```
t4 = ("CHECK-SEQUENCE", [(D,5),(A,6)], 1)
```

12. t4 is passed to the owner of A (P1). A.sequence is still equal to 6, so P2 increments the counter in t4. Since the counter is now equal to the number of principals in the token, then D and A are garbage and can be safely deleted. P1 sends the following tokens to P1 and P2:

```
("DELETE", A) --> P1
("DELETE", D) --> P2
```

13. When P1 and P2 receive these tokens, P1 removes A from P1.principals and P2 removes D from P2.principals. In the next local garbage collections A and D and all the objects they directly reach are reclaimed.
14. Now we go back to t5. t5 is passed to the owner of X3 (P1). P1 adds B and X5 to t5. P1 also removes X3 from t5.

```
t5 = ("FIND_CLOSURE", [(E,2),(B,3)], [X5])
```

15. t5 is passed to the owner of X5 (P2). Since R is in X5.roots, P2 discards t5.

Example of a Distributed Garbage Collection

3 Correctness and Completeness

Now we prove that this algorithm is *correct* and *complete*. *Correctness* means that the algorithm collects only garbage objects. *Completeness* implies that any garbage produced in the system is sooner or later collected. In this section we first extend the definition of *consistency* for a list of principals in a token. Next, we describe some helpful invariants of the algorithm. Finally we prove the correctness and completeness of the algorithm.

We use Invariant 2 to define *consistency* for a list of principals.

Definition 1 The list *principals* in a token is *consistent* if for every pair *(o,oseq)* in *principals*, *o* exists and *o.sequence* is equal to *oseq*.

This means that if *principals* is *consistent*, then the back-references of these principals have not changed since the time these principals were added to the token. *Consistency* is used to describe the following invariants.

Invariant 3 Let token *("FIND-CLOSURE", principals, proxies)*: if *principals* is *consistent* and there is a root that reaches an object in *principals*, then the root reaches the object through a proxy in *proxies*.

This invariant holds in the algorithm and can be proven by induction in the number n of processes that the token visits. *Proof*: At $n = 0$, a new token $t_0 = (\text{"}FIND-CLOSURE\text{"}, principals_0, proxies_0)$ is created from a principal *suspect* that is considered potential garbage (not directly-reachable from any local root). Next, *suspect* and its proxies are added to t_0. Also the principals that reach *suspect* and their proxies are added to t_0. Since none of the principals that reach *suspect* are reachable from a local root, if there is a root that reaches *suspect* or any of the principals, then the root has to use any of the proxies that are already in $proxies_0$. Then the invariant holds for t_0 at this point of the algorithm.

After that, the algorithm unlocks the data structures, and t_0 is passed to the next process. While the token is in transit, the only way to change the reachability graph between proxies in $proxies_0$ and the objects in $principals_0$, is through method invocations to proxies in $proxies_0$, or by creating new proxies for principals in $principals_0$. Since that increments the sequence numbers of the objects in $principals_0$, then $principals_0$ would not be *consistent*. Therefore the invariant holds when t_0 is in transit.

As induction hypothesis we assume that the invariant holds for the token t_{n-1}= *("FIND-CLOSURE",* $principals_{n-1}$, $proxies_{n-1}$*)* received by process n. Using the induction hypothesis, in the induction step we prove that the invariant still holds for a token t_n = *("FIND-CLOSURE",* $principals_n$, $proxies_n$*)* that is passed to the next process.

First when t_{n-1} is received, the daemon locks the process data structures to prevent changes, and to stop remote method invocations to the local principals. Then t_n is initialized to *("FIND-CLOSURE",* $principals_{n-1}$, $proxies_{n-1}$*)*. At this point the invariant holds for t_n by the induction hypothesis. Then the first proxy x in *proxies* is read. If *x.roots* is not *consistent*, then the token is discarded; so there is nothing to prove. Otherwise, x is directly-reachable only from local principals and no from local roots. Then the local principals that directly-reach x are added to $principals_n$ and their proxies are added to $proxies_n$. In this way, if a root reaches any of the principals recently added to $principals_n$, the root reaches the principal through any of the proxies just added. Therefore the invariant holds at this point.

Later the daemon deletes x from $proxies_n$. We have to prove that also the invariant holds in this case. First if x was not used by a root to reach a principal in the token, then we can safely remove x without breaking the invariant. Otherwise, if x was used by a root to reach a principal in the token, then since x is directly-reachable only by principals that were recently added to the token, then at least one of these principals is reachable from a root. Since all the proxies that reach these principals were also recently added to the token, then one of these proxies can be used by a root to reach the principal. Therefore x can be removed from t_n without breaking the invariant.

When t_n is in transit, the only way the reachability graph between proxies in $proxies_n$ and principals in $principals_n$ can be modified, is by method invocation on any of the proxies in $proxies_n$ or by creating new proxies for principals in $principals_n$. Since that increments the sequence numbers of principals that are already in $principals_n$, then $principals_n$ would not be *consistent*. Therefore the invariant holds for t_n $\Box$.

As a corollary of the previous invariant we have that:

Invariant 4 Let token *("FIND-CLOSURE", principals, proxies)*: if *principals* is *consistent* and *proxies* is empty, then the objects in *principals* are unreachable from any root and are garbage.

Proof: Lets assume that there is a root that reaches any of the objects in *principals*. By invariant 3, if a principal in the token is reachable from a root, then there should be a proxy in *proxies* used for that. Since *proxies* is empty then no object in *principals* is reached from a root, a contradiction. This implies that the objects in *principals* can be safely reclaimed because they are not reachable from any root□.

Now we are ready to prove that the algorithm is correct. *Proof*: After a process receives a token t = *("FIND-CLOSURE", principals, proxies)* where *proxies* is empty, all the objects in *principals* are checked for consistency. If all the objects in *principals* are consistent, then by Definition 1 *principals* was consistent at least when t was completely formed. Since *proxies* is empty and by Invariant 4, the objects in *principals* were already garbage when t was completely formed. Therefore, the algorithm collects only garbage□.

Now to prove completeness we assume that the local garbage collectors are correct, complete, and that they run garbage collections periodically. *Proof*: After a graph of garbage is generated, there is going to be a time that all the processes involved run a local garbage collection. After this time, there are not going to be method invocations to any of the principals in the garbage graph. Therefore, for every principal and proxy z in the graph of garbage, *z.roots* is *consistent* and does not change. At this point all the back-references that are needed for back-tracing are available.

We have to prove now that progress is made is made at this point. Since the local garbage collectors run periodically, a process in the graph of garbage generates a token *("FIND-CLOSURE", principals, proxies)* that starts the back-tracing. We can learn from the algorithm that a principal can be inserted in *principals* only once. As a result, proxies can be inserted to *proxies* also only once. We can learn also from the algorithm that every time a token *("FIND-CLOSURE", principals, proxies)* visits a process, one proxy is removed from *proxies*. Since proxies are inserted only once, one proxy is removed every time a process receives the token, and since the number of proxies in a garbage graph is bounded, then the token is passed around the processes until the proxies are exhausted and the graph is collected. This proves the completeness of the algorithm□.

4 Fault Tolerance

In this section we show in which way the algorithm is tolerant to lost and duplicated messages, process failures, and network failures.

4.1 Lost and Duplicated Messages

The algorithm deals with lost and duplicated messages gracefully. The token messages are self-contained. It implies that all the state required for the back-tracing of a closure is stored in the token itself; so no state is kept in the processes. If a token is lost, another token is generated later after some process in the closure runs a garbage collection. If a token is duplicated, then one of them may succeed collecting garbage and the others are discarded. The algorithm was designed in such a way that no clean up is necessary after a garbage collection is halted.

4.2 Process Failures and Network Failures

Process failures are detected using a well-known server in each host. This server gives information about the status of the processes in that host. Whenever the passing-token operation times-out, the sender process queries the server in the recipient host. If that process has crashed, then the sender removes the process from *o.processes*. If the process has not crashed, then it continues trying to pass the token. If the server does not respond, then it assumes that a network failure has happened and it discards the token. When the server (or the network) comes back again, another token is generated by some process in the closure.

5 Related Work

In this section we enumerate a list of previous works on distributed garbage collection. For a thorough survey of local garbage collection techniques, we invite the reader to refer to [Wil92].

As we have explained before, there are a lot of algorithms for distributed garbage collection that have very different characteristics. We categorize them in three basic techniques: Reference Counting, Tracing, and Hybrid. Reference Counting algorithms keep a count of the number of references that are being used for an object. When this counter drops to zero, the object is safely removed. Tracing techniques use variations of mark-and-sweep to collect garbage objects. Reference counting techniques do not need global coordination but do not collect cycles of garbage. Tracing techniques collect cycles of garbage but require global coordination. Hybrid techniques use both reference counting and tracing.

In the reference counting category there are algorithms that have reduced overhead at passing references. Some of these algorithms pass references in a single message. However, they are unable to collect objects after a process that was helding references to them has crashed.

Weighted reference counting is one of these algorithms [Bev87], [WW87]. In weighted reference counting each reference has an associated integer weight. When a process passes a copy of the reference, part of this weight goes with the copy. When a reference is deleted, the owner of the object is informed and it subtracts this value from the original weight. When the original weight goes to zero, the object is deleted. Notice that if a process crashes helding a reference to an object, the object is never deleted. Also this algorithm is not tolerant to message lost and duplication.

Reference listing is a variation of reference counting that is tolerant to failures [SDP92], [BEN+93], and [MS91]. In reference listing, principals keep a list of the processes that have references to them. This is similar to what our algorithm does. When a reference is transmitted, the owner of the principal has to be notified to update the list for that principal. By including also a monotonically increasing sequence number for every reference operation in each process, the operations are made *idempotent*. In this way reference operations are tolerant to message lost and duplication. Reference listing scales up very well, however it is not capable of collecting garbage cycles.

Tracing with time stamps [Hug85] is an elegant extension to the mark-and-sweep garbage collector for a distributed environment. The basic idea is that each local garbage collection propagates a GC-time-stamp from the local roots to the objects they reach. These time stamps are also propagated from proxies to principals. If an object can be labeled with more than one time-stamp, the highest one is used. Time-stamps of Garbage objects never increase. In this way, objects with time-stamps below some global lower-bound (highest time-stamp completely propagated in the system) are garbage collected. The main problem of this algorithm is that it requires global coordination to compute this lower-bound. Also the garbage of the whole system may be held because a lazy process does not run garbage collection.

A logically centralized but physically distributed reference service is used in [LL89] to centralize all the reachability information of the system. Each local garbage collector sends the reachability information they gather to this service. Using all the reachability information, the service builds the reachability graph of the system and determines which objects are garbage. Later in another paper by the same authors [LL92], they reduced the amount of information that the centralized server needs to hold by implementing Hughes' algorithm. The centralized server propagates time-stamps internally and it determines the global lower-bound. Because of the need of a centralized service, this algorithm does not scale up well.

Garbage collection between groups is suggested in [LQP92] to solve the problem of global synchronization. Complete garbage collection is performed inside each group, and inter-group references are treated conservatively. First the algorithm performs group negotiation to build up a group. Then it performs a marking phase that requires coordination between the processes in the group. Objects inside the group that are referenced by other groups are considered marked, and mark messages are propagated from proxies to principals. When all the mark messages have been completely propagated in the group, unmarked objects are collected. Even though the algorithm addresses the issue of global coordination, it only solves the problem partially. Services that have a large number of clients should include all the clients in their garbage collections.

Some other techniques are a hybrid of reference counting and tracing techniques. For example [Bis77] suggests the idea of using reference counting plus object migration to collect cycles of garbage. When a group of objects is suspected to form a garbage cycle, they are moved to a single process where the local garbage collector can collect them. A major drawback of the algorithm is that not all distributed systems allow object migration.

Another hybrid technique traces locally subgraphs rooted by potential garbage objects [JL92]. Objects in the subgraph that are reachable from an external root are kept. Objects that are only reachable from within the subgraph are deleted. The drawback of this algorithm is the overhead introduced by tracing the subgraph every time a reference is deleted. In a following paper [JL92], they reduce the tracing overhead by delaying the tracing of several graphs and grouping them into one. However, still some kind of global synchronization is needed to allow multiple auxiliary tracings to run at the same time. Our algorithm has some similarities to this one. However, as we have explained, our algorithm traces a graph backwards instead of forwards.

The algorithm we have described in this paper is a hybrid of reference listing and tracing. However, the tracing is done in a different way. Instead of following references from the top to the bottom, as normal mark-and-seep tracing algorithms do, our algorithm follows back-references from the bottom to the top. These back-references are provided in two ways: 1. For every principal o there is a list of processes that have proxies to o; and 2. For every proxy and principal z there is a list of principals $z.roots$ that reach z. These kind of tracing finds the roots that reach a garbage suspect. In contrast, normal mark-and-sweep finds the objects that are reached by a given root. We have called this kind of tracing *back-tracing*. Back-tracing has the advantage that it does not require global synchronization.

6 Conclusion

In this paper we have described an algorithm for distributed garbage collection that collects cycles of garbage and does not require global synchronization. The algorithm is based on back-tracing the references that a potential garbage object has. If the back-trace finds a root, the collection stops. Otherwise the entire traced graph is garbage collected. The algorithm stores in a single message all the state required to back-trace a graph. This makes the algorithm not to require global synchronization and tolerant to lost and duplicated messages. We proved the correctness of the algorithm and described its implementation.

References

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[BEN+93] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed Garbage Collection for Network Objects. Technical Report 116, Digital Equipment Corporation Systems Research Center, 1993.

[Bev87] David I. Bevan. Distributed Garbage Collection Using Reference Counting. In de Bakker et al. [dBNT87], pages 176/117–187.

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

[dBNT87] Jacobus W. de Bakker, L. Nijman, and Philip C. Treleaven, editors. *Parallel Architectures and Languages Europe*, number 258, 259 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, June 1987. Springer-Verlag.

[Hug85] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *ACM Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy, France, September 1985. Springer-Verlag.

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

[LL89] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault- tolerant distributed garbage collection. In *Fifth ACM Symposium on the Principles of Distributed Computing*, pages 29–39, 1989.

[LL92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *Principles of Distributed Computing*, pages 708–715, 1992.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming*, pages 39–50, Albuquerque, New Mexico, January 1992.

[MR93] Patrick A. Muckelbauer and Vincent F. Russo. The Renaissance Distributed Object System. Technical Report TR.93-022, Department of Computer Science, Purdue University, 1993.

[MS91] Luigi Mancini and S. K. Shrinivastava. Fault-tolerant reference counting for garbage colection in distributed systems. *Computer Journal*, 34(6):503–513, December 1991.

[Ros94] Guido Van Rossum. Python Reference Manual. Technical report, CWI, Amsterdam, October 1994. Available from http://www.cwi.nl/ guido/Python.html.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Symposium on Principles of Distributed Computing*, pages 135–146, Vancouver (Canada), August 1992. ACM.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[WW87] P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. In de Bakker et al. [dBNT87], pages 432 – 443.