A detection algorithm for distributed cycles of garbage

Fabrice Le Fessant, Ian Piumarta, Marc Shapiro

INRIA, Project SOR,
B.P. 105, 78153 Le Chesnay Cedex, France

{lefessan, piumarta, shapiro}@sor.inria.fr

September 16, 1998

Abstract

We present an algorithm that detects cycles of garbage in reference-based distributed systems. It is derived from Hughes' algorithm, in a simplified form that makes far fewer assumptions about the system. A local garbage collector marks incoming and outgoing references with timestamps that are propagated asynchronously between spaces. A central site computes the minimum reachable timestamp, allowing stale references to be identified and deleted. The coexistence of non-participating spaces, and spaces participating in collections controlled by more than one central site, is allowed.

Introduction

Remote references are often represented by "proxy" objects that transparently implement remote procedure calls. A reference to a remote object points to a *stub* which has the same interface as the remote object. The stub forwards procedure calls to a corresponding *scion* (or "server stub") that invokes the procedure directly on the target object.

Stubs and scions are well adapted to the support of distributed garbage collection [3]. Each space might contain a local garbage collector that treats scions as local roots. Periodically exchanging reachability information between spaces allows scions with no corresponding stubs to be reclaimed. The number of scions referencing an object therefore approximates a distributed reference count for the object. As with any collector based on reference counting, this is not sufficient to identify and reclaim cyclic garbage.

The next section presents our extension to the above scheme that detects cycles of garbage and allows them to be reclaimed. Section 2 explains how we have incorporated the algorithm into an existing reference-based distributed object system. Section 3 discusses the relationship of this scheme with Hughes' algorithm [1]. Finally, section 4 offers some conclusions about our approach.

1 Cyclic garbage detection

Our algorithm is based on timestamp propagation, as proposed by Hughes [1]. We therefore suppose the existence of a distributed clock in the system—for example, our implementation uses a Lamport logical "clock" [2] to order events within the distributed syste. Each space can thus use its local clock to determine the "current" time. We also assume some mechanism (such as a local garbage collector) that is able to identify stubs that are reachable from a given set of roots, to mark these stubs with a timestamp, and to do this in several passes using a different root set and timestamp in each pass.

We consider stubs and scions as belonging to chains of remote references. The problem is therefore to identify chains that are not locally rooted in any space through which they pass. We also assume that each stub references a single "matching" remote scion, and that each scion is referred to from a single "matching" stub. When referring to stubs and scions in the description below we will often refer to the "matching" space—this is the "downstream" space containing the scion (that matches a local stub) or an "upstream" stpace containins a stub (that matches a local scion).

As the spaces continue to trace, mark and propagate dates, stubs and scions that are part of chains will receive increasing timestamps—either directly from local roots, or indirectly via the scions from which they are reachable. Only chains that have no local roots in any space (and which are therefore cyclic garbage) will eventually stabilise at a constant date.

To recognise these chains we compute a time, called **globalmin**. Stubs and scions with timestamps before this date belong to chains that are not locally rooted in any space. This computation is made by the central site, and is the minimum of **localmin** dates, calculated and sent to it by each space after performing a "cyclic GC trace".

1.1 Data structures

Figure 1 shows the structures and messages used by our algorithm.

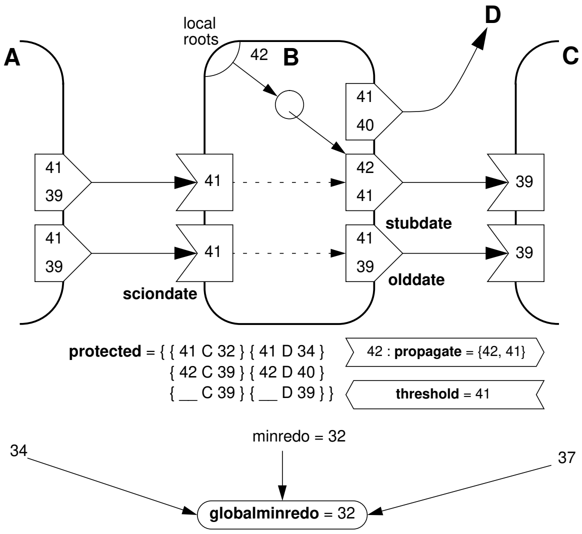

Figure 1: Structures and messages used by the cyclic garabage detection algorithm.

Stubs are extended with a timestamp which we call **stubdate**. This is the time of the most recent trace (possibly on a remote site) during which the stub's chain was found to be rooted. Stubs have a second timestamp, called **olddate**, which is the maximal value of **stubdate** for previous traces.

Scions are extended with a timestamp which we call **sciondate**. This is a copy of the most recently propagated **stubdate** from the scion's matching stub—i.e. the time of the most recent remote trace during which the scion's chain was found to be rooted. The **stubdate**s from a space are propagated to the matching in the matching space by sending a **propagate** message.

propagate messages are stamped with the time of the trace that generated them. Each site has a vector, called **cyclicthreshold**, which contains the timestamp of the last **propagate** message received from each remote space. The **cyclicthreshold** value for a remote space is periodically propagated back to that space by sending it a **threshold** message.

Each site calculates a time which we call **localmin**. This is sent to a central site, where the minimum **localmin** of all spaces is maintained in a variable which we call call **globalmin**.

Finally, each site has a table which we call **protected**. This table contains triples of the form

$$time \times space \times olddate$$

recording the minimum *olddate* for each matching *space* at a particular *time* in the past.[1]

1.2 The algorithm

We can now formulate the problem as calculating the correct value for **globalmin**: Any stubs or scions marked with a **stubdate** or **sciondate** (respectively) earlier than this are part of a chain that is not rooted in any space—a distributed cycle of garbage. The algorithm proceeds as follows.

Each space traces from its local roots, marking accessible stubs with the current time. The space then traces from any incoming scions: accessible

[1] The values in the table will be shown in italics to distinguish them from the values of the same name attached to stubs and scions.

stubs are marked with the largest **sciondate** from which they are reachable.

After the above tracing phase, any stubs for which **stubdate**$\neq$**olddate** are considered to be rooted. For each such stub we compare the stub's **olddate** with the corresponding entry in the **protected** table; i.e. the entry having the current *time* and the stub's matching *space*. If the stub's **olddate** is lower than the *olddate* in the table (or if there is no corresponding entry in the table) then value in the table is updated (or created).

The space can now calculate a new value for **localmin**. The updated **protected** table contains the minimum safe value of **localmin** for each matching *space* for any given *time*. The minimum of all these *olddate*s is therefore the only safe value for **localmin**, and so this becomes the new value of **localmin** that is sent to the central site.

After sending **localmin** to the central site, and awaiting an acknowledgement, the space can propagate the new **stubdate**s to any matching space. A **propagate** message is sent to each matching space, containing the new dates for each matching scion in that space. For each stub, the dates placed in the **propagate** message is as the maximum of the stub's **stubdate** and **olddate**s for each stub (which permits **propagate** message loss). When the remote space receives this message it updates the matching scions' **sciondate**s with the dates from the corresponding scions in the message.

Finally the space sends a **threshold** message to each matching space, containing the timestamp of the last **propagate** message received from that space. The purpose of this message will be explained below.

The **protected** table must be pruned periodically, otherwise the **localmin** for a space would never increase and cyclic garbage would never be detected. We prune the table whenever we receive a **threshold** message. Since this message is only sent after performing a trace, it indicates that the sending space has finished tracing from incoming scions and has sent its new **localmin** to the central site. We can therefore remove any entries in the local **protected** table that were previously responsible for protecting these remote scions, since the sending space will now be protecting them if necessary. These entries are identified as having a *space* equal to the sender of the **threshold** message, and a *time* that is smaller than the timestamp contained in the **threshold** message.[2]

1.3 Coping with mutator activity

There is one remaining problem: we must account for cycles in which mutator activity prevents a local root from ever being seen by a trace, because of spaces rapidly exchanging references, and deleting local roots, between traces. To address this problem we keep a set of "floating" entries in the **protected** table (these times are shown as "_" in Figure 1). Whenever a stub is used as an argument in an RPC its **olddate** is compared to the *olddate* of the floating entry for the destination space; if the stub's **olddate** is smaller then the floating *olddate* is replaced. When the space decides to perform a trace, the floating entries are first "fixed" at the current time—thereby protecting any scions that have been created in remote spaces due to recent mutator activity, regardless of whether the corresponding stubs are identified as rooted during the ensuing local trace.

1.4 Non-participating spaces

Scions are created due to references appearing in an incoming RPC message. Each time a scion is created its **sciondate** is set to a special value, which we call $UNKNOWN$, that always represents the current time. The **sciondate** for references sent to non-participating spaces are therefore always considered equal to the current time, and are therefore always considered rooted. A participating space will eventually send a **propagate** message to the local space, updating any **sciondate**s equal to $UNKNOWN$ with the correct value.

This mechanism allows spaces to alternate safely between "participating" and "non-participating", without risk of their scions being incorrectly identified as garbage.

[2] Recall that this is the timestamp of the last **propagate** message that we sent to the **threshold** message's sender space.

1.5 Fault tolerance

The above algorithm is tolerant to message loss. Since the **protected** set is only pruned when a **threshold** message is received from a remote space, and since the **threshold** message contains the date of the last **propagate** sent to the remote space, losing either of these messages will simply postpone the removal of the associated entries in the **protected** set. It is also tolerant to spaces that crash—they simply cease to participate in the distributed cyclic detection algorithm.

2 Implementation

Our algorithm has been incorporated into an implementation of the SSP Chains system [5] written in Objective CAML. The implementation contains 1300 lines of code, of which 200 are associated with the cyclic GC algorithm. The tracing phase of our algorithm was implemented by making minor modifications to the existing CAML garbage collector.

SSP Chains include a cleanup protocol that implements an acyclic distributed garbage collection algorithm. This works in parallel with our cyclic detection algorithm, and scions matching non-participating spaces (for example) will continue be reclaimed normally by this protocol. Once cyclic garbage has been detected by our algorithm, we need only break a link in the chain for the existing SSP cleanup protocol to reclaim the cycle of garbage.

Out-of-order delivery is also tolerated in the SSPC implementation by selectively imposing sequenced reception of messages. SSP Chains already address various race conditions by time-stamping each RPC message and keeping threshold tables to decide when a stale message must be dropped. This mechanism is used directly by the cyclic detection algorithm to periodically impose an ordering on message reception. This is done by copying the RPC timestamp of the last **propagate** message received from each space into the SSPC threshold table entry for that space.

Dropping messages in this manner is tolerable for two reasons. Firstly this ordering is only imposed at the time that a space decides to perform a cyclic trace. Only those messages that have been sent very recently, or which have been delayed for a very long time, risk being dropped. Secondly the application must have chosen an unreliable protocol (such as UDP) for its communication, and must therefore be willing to accept lost messages. An application which is not prepared for this would instead choose a reliable protocol, providing sequenced delivery, in which case its messages will never be dropped by our algorithm.

Finally, we reduce the number of messages sent due to cyclic GC by combining the **propagate** and **threshold** messages into a single "**cycliclive**" message, containing both the propagated **stubdate**s for the matching space and the timestamp of the last **propagate** message received from that space.

3 Related work

Our work is greatly inspired by Hughes' algorithm [1], with some simplifications and enhancements to make it easily implementable.

Hughes' algorithm

Hughes' algorithm makes impractical assumptions about the environment in which it runs, such as the availability of a synchronous global time and instantaneous communication.

Furthermore, Hughes' algorithm requires a variable in each space, called "redo", which is the earliest date which may not have been properly propagated by a local trace. This variable increases during collections, and decreases when receiving a message to update scion dates. Rana's termination 'algorithm [4] is then used to take a snapshot of all the "redo"s in the system at a given time in order to compute the minimum value "minredo" of live reference dates. To achieve this, spaces must keep a history of all their "redo" values since the latest snapshot.

Finally, Hughes' algorithm does not address fault-tolerance nor the existence of non-participating spaces.

Simplifications and enhancements

The idea behind our algorithm was to replace Rana's termination algorithm by a centralised computation of **localmin**. Global time was also replaced by Lamport's logical clock [2].

We initially designed a synchronous algorithm (described in a forthcoming INRIA Research Report), in which we still had to compute remote **redo** values by comparing new stub dates with old scion dates, before updating **localmin**.

To make this algorithm asynchronous, each stub was given a memory of its previous date—called **olddate**—which enables each space to recognise progressing stubs and to compute its **redo** value asynchronously at trace time. Using this additional information, **propagate** messages no longer need to be synchronous.

Finally, to cope with non-participating spaces and fault tolerance, we added a new special date *UNKNOWN* that identifies scions from non-participating spaces as local roots, therefore making the safe assumption that they don't belong to garbage cycles.

4 Conclusion

We have given an initial presentation of a simple algorithm that detects cycles of garbage in a distributed reference-based system. Our algorithm is non-intrusive, since it runs in parallel with mutators and garbage collectors. Its overheads are negligible, requiring three additional timestamps per stub/scion pair, infrequent messages between spaces, and the maintenance of a local set whose size is proportional to the number of connected spaces and the average number of **propagate** messages sent before receiving a **threshold**. Moreover its implementation is extremely simple, since only trivial modifications of the marking function of a standard tracing garbage collector are needed to update the timestamps on stubs.

This algorithm seems to be well adapted to large-scale distributed systems, since it supports non-participating spaces and even failures of participating spaces. Spaces can be organised in dynamically-evolving clusters and participate with more than one central site, and even failures of the central site can be supported by interleaving clusters or introducing redundant copies of the central site. The algorithm is also tolerant to messages being lost, duplicated, or delivered out of order.

A formal proof of the safety of the algorithm has been constructed, an outline of which is included as an appendix to this paper. A prototype implementation, within the Objective-CAML version of SSP Chains, is already running.

Our algorithm is a "work-in-progress" (it was evolving even during the writing of this paper). Further progress will be reported at the workshop.

References

[1] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985. Springer-Verlag.

[2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[3] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Second Closed BROADCAST Workshop*, pages 211–249, Bruxelles (Belgique), November 1994. Broadcast Basic Research Action.

[4] S. P. Rana. A distributed solution to the distributed termination problem. *Information Processing Letters*, 17:43–46, July 1983.

[5] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1992. `http://www-sor.inria.fr/publi/SSPC_rr1799.html`.

A Safety proof

Notation

$name_{S_k}(T)$ is the value of *name* on space S_k at time T. $\mathsf{MESSAGE}_{S_k(T_k)\rightarrow S_l(T_l)}$ is the message sent by S_k at time T_k and received by S_l before T_l.

NOW is a special date, which is always replaced by the current time during tracing.

Hypothesis

Let S be the central site
Let $S_{1..n}$ be the participating and non-participating spaces
Let S_i be one participating space
Let T_i^0, T, $T_{1..n}$ be times such that $globalminredo_{S_i}(T_i^0) = globalminredo_S(T)$
and $globalminredo_S(T)$ was computed from $minredo_{S_k}(T_k)$

Lemma 1

S_k and S_l are two participating spaces.
If $\mathsf{CYCLICLIVE}_{S_l(T_l')\rightarrow S_k}$ is received before T_k, then $T_l' \leq T_l$.

Proof:

$\mathsf{MINREDO}$ messages must be acknowledged by the central site before any $\mathsf{CYCLICLIVE}$ message is sent during a local trace. So, $\mathsf{MINREDO}_{S_l(T_l')\rightarrow S}$ has been received by S before $\mathsf{CYCLICLIVE}_{S_l(T_l')\rightarrow S_k}$ has been sent. Since $\mathsf{CYCLICLIVE}_{S_l(T_l')\rightarrow S_k}$ is received by S_k before $T_k \leq T$, $\mathsf{MINREDO}_{S_l(T_l')\rightarrow S}$ is received by S before T. But the latest message received from S_l by S before T is $\mathsf{MINREDO}_{S_l(T_l)\rightarrow S}$, so we have $T_l' \leq T_l$.

Lemma 2

S_k is a participating space and S_l a space.
If ($stub_{S_l}$, $scion_{S_k}$) is a pair such that $scion_{S_k}.date(T_k) < globalminredo_S(T)$, then:

- *At time T_l, $stub_{S_l}$ is only accessible from scions satisfying:*
  $scion_{S_l}.date(T_l) < globalminredo_S(T)$
- $max(stub_{S_l}.date(T_l), stub_{S_l}.olddate(T_l)) < globalminredo_S(T)$

Proof:

$scion_{S_k}.date(T_k) < globalminredo_S(T)$
and $globalminredo_S(T) \leq minredo_{S_k}(T_k)$
and $minredo_{S_k}(T_k) \leq T_k$
$\Rightarrow scion_{S_k}.date(T_k) < T_k$
$\Rightarrow scion_{S_k}.date(T_k) \neq \mathsf{NOW}$

Since $scion_{S_k}.date(T_k) \neq$NOW, S_k must have received a message $\mathsf{CYCLICLIVE}_{S_l(T_l')\rightarrow S_k}$ from S_l, sent at time T_l', and received before T_k, which previously set the value of $scion_{S_k}.date$. Therefore, S_l is a participating space and, from Lemma 1, $T_l' \leq T_l$.

Since $scion_{S_k}.date$ is an increasing value, and the date sent by $\mathsf{CYCLICLIVE}_{S_l(T_l')\rightarrow S_k}$ for $stub_{S_l}$ is $max(stub_{S_l}.date(T_l'), stub_{S_l}.olddate(T_l'))$, we have:
$max(stub_{S_l}.date(T_l'), stub_{S_l}.olddate(T_l')) \leq scion_{S_k}.date(T_k)$

$\Rightarrow max(stub_{S_l}.date(T_l'), stub_{S_l}.olddate(T_l')) < globalminredo_S(T)$

Now we must prove that, for any time T_l'', $T_l' \leq T_l'' \leq T_l$, we have $stub_{S_l}.date(T_l'') < globalminredo_S(T)$:

Suppose T_l'', $T_l' \leq T_l'' \leq T_l$, is the first time such that $stub_{S_l}.date(T_l'') \geq globalminredo_S(T)$. Then, since $stub_{S_l}.olddate$ is the maximum of all previous $stub_{S_l}.date$ values ($stub_{S_l}.olddate$ may also be modified by an application message containing the location of $scion_{S_k}$, but this is not the case since $scion_{S_k}.date$ would have been set to NOW), we have $stub_{S_l}.olddate(T_l'') < globalminredo_S(T)$.

$\Rightarrow stub_{S_l}.olddate(T_l'') < stub_{S_l}.date(T_l'')$
$\Rightarrow protected_{S_l}(S_k, T_l'') \leq stub_{S_l}.olddate(T_l'')$
$\Rightarrow protected_{S_l}(S_k, T_l'') < globalminredo_S(T)$

Moreover, $stub_{S_l}.date(T_l'')$ is contained in $\mathsf{CYCLICLIVE}_{S_l(T_l'')\rightarrow S_k}$, and greater values are contained in subsequent CYCLICLIVE messages . Since $scion_{S_k}.date(T_k) < globalminredo_S(T)$, none of them have been received by S_k before T_k.

From Lemma 1, at time T_l, only $\mathsf{CYCLICLIVE}_{S_k\rightarrow S_l}$ messages sent before T_k have been received by T_l. All these messages have a threshold strictly lower than T_l'', since $\mathsf{CYCLICLIVE}_{S_l(T_l'')\rightarrow S_k}$ and subsequent messages have not been received by S_k at time T_k.

So, at time T_l, $protected_{S_l}(S_k, T_l'')$ is still in the $protected_{S_l}(T_l)$ set:

$\Rightarrow minredo_{S_l}(T_l) \leq protected_{S_l}(S_k, T_l'')$
But $protected_{S_l}(S_k, T_l'') < globalminredo_S(T)$
$\Rightarrow minredo_{S_l}(T_l) < globalminredo_S(T)$
which is a contradiction since $minredo_{S_l}(T_l)$ is used to compute $globalminredo_S(T)$ on the central site at time T.

We have proved that $max(stub_{S_l}.date(T_l), stub_{S_l}.olddate(T_l)) < globalminredo_S(T)$.

From this, at time T_l, $stub_{S_l}$ is not reachable from any local roots. It is only reachable from scions whose date at time T_l is inferior to $globalminredo_S(T)$

Proof

Let ($stub_{S_j}$, $scion_{S_i}$) be a stub-scion pair such that $scion_{S_i}.date(T_i^0) < globalminredo_{S_i}(T_i^0)$: We must prove that it belongs to a garbage cycle before erasing it.

As $scion.date()$ is increasing, and $T_i < T < T_i^0$, we have: $scion_{S_i}.date(T_i) < globalminredo_S(T)$

From **Lemma 2**, we know that $stub_{S_j}$ is reachable at time T_j only from scions satisfying the same property at time T_k.

This property characterises garbage cycles if and only if, for a stub-scion pair ($stub_{S_k}$, $scion_{S_l}$) in the cycle, no local reference may have been created with $scion_{S_l}$ by an application message sent from $stub_{S_k}$ at time T'_k before T_k and received by space S_l at time T'_l after time T_l.

This event must belong to one of the following two cases: either a $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ message sent after the application message is received before T_l by S_l, or no $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ message sent after the application message is received before T_l by S_l.

In the first case, all application messages received after T_l, but sent before the last $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ received before T_l, will be rejected, since their timestamps are lower than the threshold at time T_l, which is set to be greater than the timestamps of all $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ received before T_l.

In the second case, from Lemma 1, at time T_l, all $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ messages received were sent before T_k. Since no $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ messages sent after the application message have been received before T_l, the last $\mathsf{CYCLICLIVE}_{S_k \to S_l}$ received before T_l was sent before the application message. Thus $protected_{S_k}(T_k)$ still contains a protected time inferior to $stub_{S_k}.olddate(T'_k)$ (to cope with mutator activity).

$\Rightarrow minredo_{S_k}(T_k) \leq stub_{S_k}.olddate(T'_k)$
From Lemma 2, $stub_{S_k}.olddate(T'_k) < globalminredo_S(T)$
$\Rightarrow minredo_{S_k}(T_k) < globalminredo_S(T)$
which is a contradiction since $minredo_{S_k}(T_k)$ is used to compute $globalminredo_S(T)$

So, we have proved that $scion_{S_i}$ belongs to a garbage cycle at time $T_i^0 \geq T$ and can be safely deleted.