# Detecting Distributed Cycles of Garbage in Large-Scale Systems

Fabrice Le Fessant
(Email: Fabrice.Le_Fessant@inria.fr)
INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

## ABSTRACT

Distributed scalable garbage collectors, mostly based on some kind of reference counting, fail to detect distributed cycles of garbage. This problem may lead to important memory leaks in distributed storage systems. In this paper, we present a new algorithm which detects and collects such distributed cycles of garbage.

Our algorithm is based on the propagation of marks along chains of remote pointers. It uses two new mechanisms: *min-max marking*, to propagate two different marks to each stub, and *sub-generation*, to build an acyclic graph on a cycle using back-tracking information. A new technique, called *optimistic back-tracking*, is also used to speed-up sub-generation.

The resulting algorithm is completely distributed, asynchronous, fault-tolerant and inexpensive. Moreover, it collects incrementally all distributed cycles of garbage, without partitioning the system. Thus, it is particularly well adapted to large-scale networks. Finally, it can be easily implemented with minor modifications of a local tracing garbage collector.

**Keywords:** distributed garbage collection, cycles, sub-generation, optimistic back-tracing, min-max marking.

## 1. INTRODUCTION

Most scalable distributed garbage collectors [15, 14, 4, 1] are based on some kind of reference counting, which fails to detect distributed cycles of garbage. To avoid important memory leaks in distributed storage systems, detection of distributed cycles of garbage has become an active area of research. Two kinds of algorithms address the problem: *per-cycle* algorithms [2, 10, 7, 12, 11, 13], i.e. algorithms triggered to verify that a particular suspected object belongs to a free cycle, and *all-at-once* algorithms [8, 9, 5], i.e. algorithms which detect all free cycles in a single mechanism, sometimes together with acyclic garbage.



### 1.1 Per-cycle detectors

Per-cycle detectors can be divided in two parts: a *heuristic*, to detect which objects are likely to belong to distributed cycles of garbage, and a *verifier*, which effectively verifies that suspected objects really belong to garbage cycles.

Lots of heuristics have been proposed, such as last access time (by the mutator), non-local reachability [12] or, more recently, distance from a local root [10, 11]. The choice of the heuristic is important: suspecting too many objects is expensive in time, since the verifier is erroneously triggered a lot of times, and suspecting too few objects is expensive in memory, since reclamation of real garbage is delayed.

When an object is suspected, the verifier is executed. Proposed mechanisms are: (1) migrating suspected objects to a single site [2, 10], where a tracing local garbage collector can collect the cycle; (2) back-tracking [7, 11, 13], i.e.tracing the references to the object (backward references) recursively until either a root is reached (if the object is erroneously suspected) or all backward references are traced without finding a root (the object is really garbage). However, both methods are quite expensive in time, require important modifications of the local garbage collectors, and have strong requirements on the system, such as object migration, extra fields in objects, or overlapping trace [16](objects traced several times in the same trace).

The main drawback of per-cycle detectors is their unitary cost: an algorithm must be started for each suspected object, consuming extra memory and messages.

### 1.2 All-at-once detectors

All-at-once detectors are less expensive, since a single algorithm is able to collect all cycles in one global mechanism. However, all all-at-once algorithms require some kind of consensus involving all the spaces. Since such a consensus can hardly be achieved on the whole system [6], i.e. by thousands of spaces, these algorithms are limited to partitions of the system.

Most of them are distributed tracing garbage collectors: once a partition of the system is selected — because it is likely to contain lots of garbage cycles — a global trace is triggered from local roots and scions associated with spaces outside the partition. Once the global trace is terminated on the partition, non-traced objects are reclaimed.

Such partitioned detectors differ either by the way they select the partition (partition can be created dynamically [12] or hierarchically [9]), or by the way traces are coordinated (traces can be sequential [12] or concurrent [8, 9, 5]). However, their main drawback is the consensus on the termination of the global trace, which limits both the number of spaces which can be involved in one partition and the tolerance of the algorithm to space crashes.

## 1.3 Our contribution

In this paper, we present a new detector of free cycles, designed for the JoCaml mobile agents system [5].

Our algorithm takes a medium approach between per-cycle and all-at-once detectors: from the per-cycle algorithms, it uses the distance heuristic, which is however improved by *min-max marking* so that some suspected cycles are immediatly detected as garbage, and a partial and lazy back-tracking mechanism, called *sub-generation*. From the all-at-once algorithms, it takes the global and uniform mechanism, since it does not require extra messages for suspected cycles. Moreover, it does not use any kind of consensus between spaces, and its requirements on the system are modest (no object migration needed, no extra space in objects, minor modification of the local tracing collector).

Finally, we propose a new mechanism, called *optimistic back-tracking*, to speed-up the sub-generation when a cycle is composed of many inter-connected objects located on very few spaces, a common case in practice.

## 1.4 Structure

The paper is organized as follow: section 2 describes our system and goal; section 3 gives an overview of the algorithm, incrementally detailed in sections 4 (basic algorithm with min-max marking); 5 (sub-generation) and 6 (optimistic back-tracking). Finally, the section 7 discusses the main properties of our detector.

# 2. MODEL

## 2.1 The distributed system

Our system consists of a set of *spaces*, which are the basic units of computation. Each space has its own local memory and its own local garbage collector. To access objects in remote memories, spaces can only send asynchronous messages, on an unreliable but fair medium of communication. Therefore, these messages may get lost, duplicated, or delivered out-of-order. Each space can access a local clock, which is either implemented by a distributed Lamport clock, or by loosely synchronized hardware clocks.

A remote reference is materialized by two objects, called *stub* (or *exit item*) and *scion* (or *entry item*) in the SSPC terminology [14]: Concretely, a reference $R$ from object $A$ in space $X$ to object $B$ in space $Y$ is represented by two pointers(Figure 1):

- a local pointer in $X$ from $A$ to the stub $stub_X(R)$ in $X$ and
- a local pointer in $Y$ from the scion $scion_Y(R)$ in $Y$ to $B$.

One scion is associated with each stub, and at most one stub is associated with each scion. Each remote reference has a unique identifier, called a *locator*, stored in its stub and scion.

Locators can be sent between spaces in messages, either as the target object of a remote method call, or as the parameters or reply of such a call. A new remote reference can be dynamically created: a scion is first created in the object space with a new locator; the locator is then sent to the remote space, where a new stub is created.

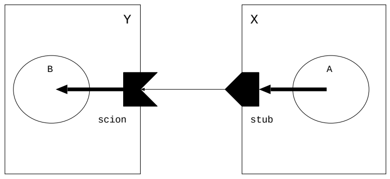


**Figure 1: A remote reference from space $X$ to space $Y$.**

## 2.2 Cycles

We write $\mathcal{O}$ for the set of all objects created in the system. If an object $O_2$ is reachable (either locally or remotely) from an object $O_1$, we write $O_1 \mapsto O_2$. If it is only *remotely* reachable from $O_1$, we write $O_1 \rightsquigarrow O_2$. An object is said to be *garbage*, when it is unreachable from a set of references, called the *roots*, which contains both local roots and messages in transit.

We are only interested here in detecting a particular kind of cycles, *Top Free Cycles*, which are strongly-connected components, with no incoming references:

DEFINITION 1 (TOP FREE CYCLE). *A set $G$ of objects is a* **Top Free Cycle** *if and only if:*

- $\exists O_1, O_2 \in G, O_1 \rightsquigarrow O_2$
- $\forall O \in G, \nexists r \in roots, r \mapsto O$
- $\forall O_1, O_2 \in G, O_1 \mapsto O_2$
- $\forall O_1 \in G, \forall O_2 \in \mathcal{O}, O_2 \mapsto O_1 \Rightarrow O_2 \in G$

We define the *perimeter* of $G$ at $O$ as the minimal length of all simple cycles in $G$ containing $O$. We define the *closure* of $G$, denoted $clos(G)$, as the set of objects $O$ such that $\exists O' \in G, O' \rightarrow O$. It is clear that any unreachable object which cannot be collected by an acyclic distributed garbage collector must be part of the closure of (at least) one top free cycle. By detecting top free cycles, and using the acyclic garbage collector for other objects, our garbage collector is *complete*. Thus, in the following, free cycles always refer to top free cycles.

# 3. OVERVIEW

Our algorithm is based on the propagation of marks along chains of remote pointers: marks are propagated locally from local roots and scions to reachable stubs by the local garbage collector, following a strict order on marks; then, stub marks are propagated to their associated scions by messages.

Our marks are complex: they have a distance, a range and a generator identifier. The distance is the number of stub-scion pairs the mark has been propagated along. The range is the maximal distance the mark can reach. When the mark range is reached, the marked scion becomes a mark generator with a strictly greater range. Local roots are special generators, all emitting the same mark identifier for a given local trace.

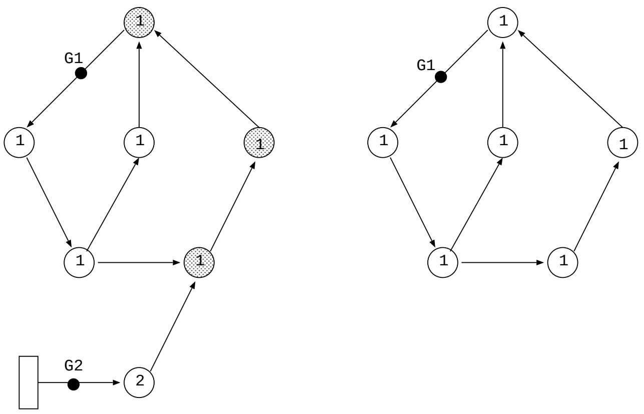


**Figure 2: The basic scheme: the left cycle is reachable, while the right one is unreachable. In the left case, generator G2 (a local root) can not receive its mark, while generator G1 of range 6 receives its mark, although gray (since mixed with mark of G2). In the right case, generator G1 receives its white mark: the cycle is detected and can be collected (by removing the generator G1).**

Generators can be removed either by mutator activity or by a greater mark propagated to their scion. Some important details are omitted here for simplicity, but it can be proved that any free cycle is eventually marked by one single generator, whose mark range is greater than the maximal subcycle length, and this generator remains forever.

We then extend marks by adding a color field: two marks are propagated to each stub, and the color of the mark propagated to its associated scion depends on them. A generator scion always emits its own mark with a white color. Other scions propagate the mark received from their associated stub. Local scions are sorted before each pair of local garbage collections: the first garbage collection traces scions in decreasing order of their marks, while the second one traces scions in increasing order. This is called *min-max marking* since each stub is first marked with the maximal mark that can be propagated to it, and then with the minimal one. Its associated scion is then marked with the maximal mark, colored in gray if the marks have different generators.

As a consequence, if a generator receives its mark, and the mark is white, the generator must be included in a free cycle (see Figure 2 for a basic example), since it can not be reachable from any local root nor other generator.

However, this basic scheme doesn't detect *all* cycles: if the final generator does not belong to an articulation edge of the cycle, orphan gray marks may appear on sub-cycles and never disappear, preventing the detection of the full cycle (see example on Figure 3).

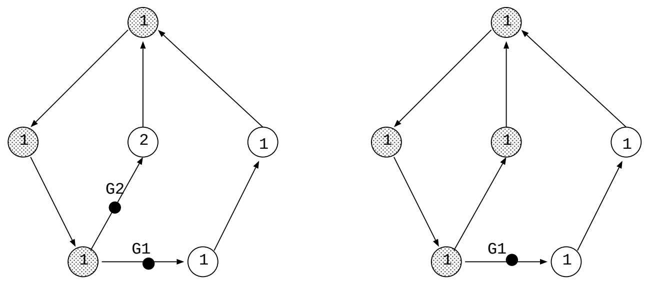


**Figure 3: The bad case: at the beginning, two generators G1 and G2 are present on the cycle. Consequently, gray marks appear on the sub-cycle, and will not disappear, even after removal of generator G2. Thus, the basic scheme of detection is not complete.**

The previous scheme is then extended with a partial back-tracking mechanism, called *sub-generation*: when a scion propagates the gray mark of the generator to the generator stub or one of its sub-generator stubs, the scion becomes a *sub-generator*, emitting the generator white mark. As a consequence, any orphan gray mark will eventually be removed, when a sub-generator appears on its sub-cycle (see Figure 5).

When all sub-generators and the generator only receive the generator white mark, the cycle is detected and can safely be reclaimed.

This extended algorithm is then improved by a mechanism called *optimistic back-tracking*: when a scion propagates the gray mark of the generator to the generator stub or one of its sub-generator stub, *all scions propagating the same* gray *mark in the space* become sub-generators. Indeed, all these scions are neighbors when scions have been sorted. Since a scion may erroneously become a sub-generator, a correction mechanism is used to remove such scions from the sub-generation.

# 4. THE BASIC ALGORITHM

## 4.1 Simple propagation of Marks

A mark is a record with a distance field and a generator record (see Figure 7). A generator record contains a creation time field, a range field and the locator of the mark generator. Marks are propagated from local roots and scions to stubs during local traces of the garbage collector, and from stubs to their associated scion by dedicated messages, called CYCLIVE, after each local trace.

At the beginning, the mark generators are the local roots; all local roots on all spaces use a common special locator and a common range, but the creation time in the generator record of a mark propagated from a local root by a local trace is always the starting time of that trace.

When a scion is created or used by the mutator, it behaves as a local root, propagating the same mark as other local roots. Otherwise, it propagates the mark received from its associated stub.

When a mark is propagated from a stub to its associated scion, its distance field is incremented. Thus, the distance field represents the number of stub-scion pairs the mark has been propagated along.

We define a strict order on generators and marks: generators use a lexicographic order on the values of their record fields (creation time first, range, and locators which are supposed to be strictly ordered with local roots locator as upper bound). For marks, the generator order is used first. If two

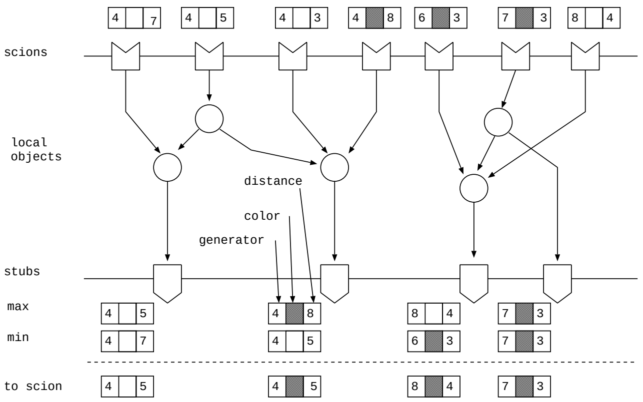


**Figure 4: Min-max marking: scions are sorted, and trace alternatively in decreasing and increasing order. Thus, each stub is marked with both the minimal and maximal mark which can be propagated from scions it is locally reachable from. The final mark is white, only if all marks which can be propagated to the stub are the same white one (only the distance may differ).**

marks have the same generator, the mark with the smallest distance is the greatest mark.

When the distance field of a mark reaches the range of its generator, the mark can not be propagated further. Instead, the marked scion becomes a new generator, emitting its own mark, using its own locator with a range strictly greater than the range of the previous received mark. If the scion was already a generator, a generator is created only if the previous generator was smaller than the generator of the received mark. This mechanism is similar to the distance heuristic [10], where the verifier is triggered if a mark has been propagated more than a suspicion threshold (here the range of local roots). Moreover, a generator is removed when a greater mark is propagated to its scion.

During the local trace, local roots are traced first, then scions are traced in decreasing order of the marks they propagate. Consequently, stubs are always marked by the maximal mark they are locally reachable from.

PROPERTY 1 (STABILITY). *Any top free cycle $G$ will eventually be marked by only one generator $S$, whose range is at least equal to the perimeter of $G$ at $O$, and this final generator will remain forever.*

With this simple algorithm, a generator can not detect if it is the final generator, since marks from other smaller generators may have been hidden during local traces. Thus, we extend this algorithm with a new mechanism called *min-max marking*.

## 4.2 Min-max marking

Mark are now colored in white or gray. The mark order is modified, so that gray marks from one generator are greater than white marks from the same generator. Otherwise, the order is unchanged.

A white mark means that the mark is pure, whereas a gray mark indicates that the mark was mixed with marks from different generators during a local trace (a stub was reachable from different roots marked with marks from different generators). The idea is that a generator that receives its own white mark must be the final generator (stability property).

We extend the previous algorithm by using *couples* of local garbage collections to propagate *two marks* to each stub: the greatest one and the smallest one. This is called *min-max marking*. This is simply implemented by sorting the scions once before each couple of local traces, and tracing them in decreasing order during the first trace, and in increasing order during the second trace [1](see Figure 8).

The mark propagated to the associated scion now depends on the two marks on the stub (see Figure 4 and 9):

- If both marks are white and from the same generator, the stub propagates the greatest mark.
- If both marks are from the same generator, but one is gray, the stub propagates the gray mark, but with the smallest distance field of both marks.
- If both marks are from different generators, the stub propagates the greatest mark, with a color set to Gray.

Since we keep the smallest distance from the two marks when they are from the same generator, the *stability property* is still true. We have now the following result:

PROPERTY 2 (RETURN OF THE white MARK). *If a generator receives its own* white *mark propagated from its stub, this generator belongs to a top free cycle.*

However, orphan gray marks may appear on top free cycles if the final generator is not on an articulation edge of the cycle (see Figure 3). As a consequence, this algorithm is not complete:

PROPERTY 3 (INCOMPLETENESS). *There are top free cycles where final generators never receive their own* white *marks.*

## 4.3 Coping with mutator activity

The mutator may change the reachability of objects while our algorithm is executing. This problem is solved by the use of timestamps in all messages, and *threshold-filtering*, a mechanism described in [5, 14].

When a reference has been sent in a message, the corresponding scion behaves as a local root, until a new mark is propagated from its associated stub. As in [5, 14], this new mark is used only if the threshold in the CYCLIVE message is greater than the timestamp of the scion. However, instead of increasing the threshold when a stub is collected, our threshold is increased when a white mark is propagated from a stub.

# 5. SUB-GENERATION

A *sub-generator* of a generator is a scion, which emits the white mark of the generator during local traces. Subgenerators are created in a recursive process started at the generator which receives its gray mark: when a generator

[1]Marks propagated from stubs are only commited at the end of the pair of local traces.

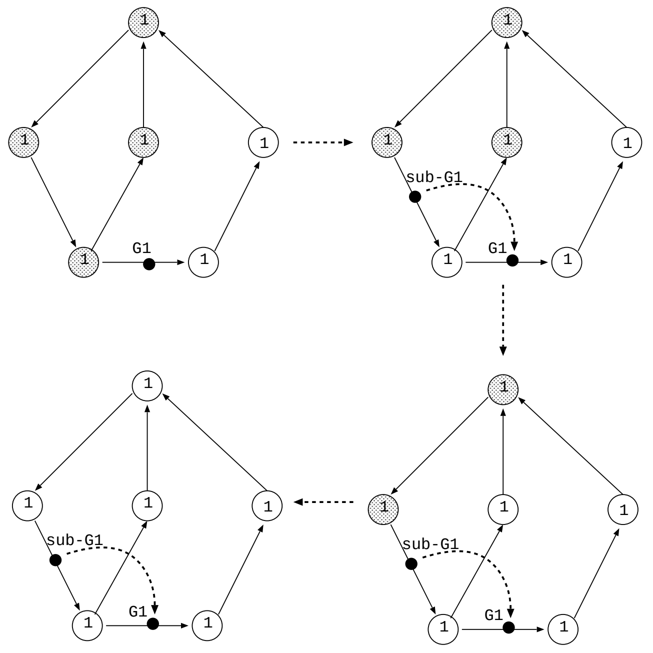


**Figure 5: Sub-generation: generator G1 receives its own mark, but gray. It starts a sub-generation, by creating the sub-generator sub-G1 in its back-trace. Sub-G1 always propagates the white mark of G1, thus removing the gray color from the sub-cycle. When both G1 and Sub-G1 receive G1 white marks, the cycle is detected and reclaimed (by removing pointers G1 and Sub-G1).**

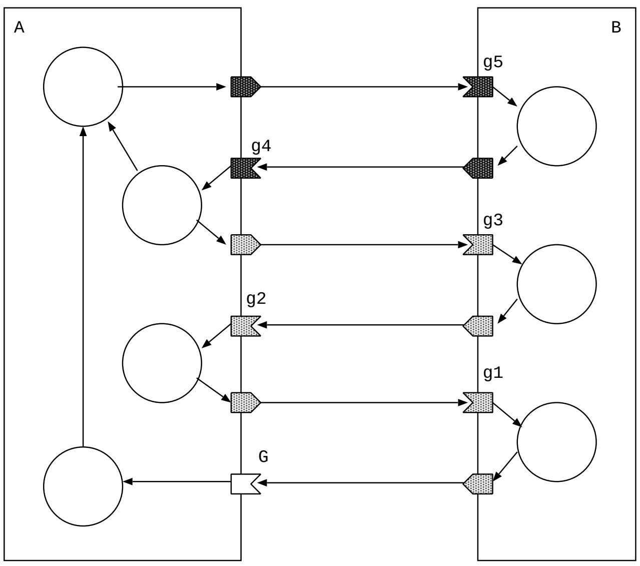


**Figure 6: Optimistic Back-tracking: Without optimistic back-tracking, the generator G will need at least four local traces in each space to create its four sub-generators g1,g2,g3 and g4 required to remove the orphan mark from the sub-cycle g4-g5. With Optimistic back-tracking, g1, g2, g3, g4, g5 become immediatly sub-generators of G and the cycle will be detected faster. This mechanism is efficient in most applications, where cycles contain lots of inter-connected objects distributed on very few spaces.**

stub, or one of its sub-generator stubs, receives the gray mark of the generator propagated from a scion ( which must be the greatest scion it is reachable from), this scion becomes a new sub-generator. The generator or sub-generator stub is called its *dominator*. The sub-generation status of each scion is then propagated to its associated stub in the CYCLIVE messages.

Thus, sub-generators are created by a lazy back-tracking process: if a generator is reachable from multiple scions marked with its gray mark, a new scion becomes a sub-generator at each pair of local traces, since new sub-generators emit the white mark, and thus, are smaller than all other scions that propagate the gray mark.

The mark propagated by a stub involved in the sub-generation process to its associated scion is computed from the marks received by the sub-generators created (dominated) by this stub from their associated stubs. If all these marks are white, and the stub mark is also white, the propagated mark is also white; otherwise, the mark is gray (see Figure 11).

The property 2 still holds. Indeed, a generator receives its white mark only if all its sub-generators also receive the white mark. However, any orphan gray mark in a sub-cycle cycle will eventually be removed when a sub-generator appears in the same sub-cycle (see Figure 5).

Property 4. *Completeness*
*The final generator of any top free cycle will eventually receive its own* white *mark.*

Once a top free cycle has been detected, the easiest and most efficient way to collect it is to remove its generator and sub-generators scions from the set of traced scions. As a consequence, the cycle will be collected by the acyclic garbage collector.

## 6. OPTIMISTIC BACK-TRACKING

The lazy back-tracking algorithm can be very slow, since it can only detect one sub-generator per pair of local traces. Thus, we propose a faster algorithm for back-tracking, which is unfortunately not exact. The basic idea is that, in a space where a sub-generator is present, all the scions which propagate the generator gray mark are probably only reachable from this generator. Thus, they can be immediatly added to sub-generators set.

Thus, when a generator or sub-generator stub is marked with the generator gray mark[2], all the scions in the local space which also propagate the same gray mark are included in the sub-generators set (unless they are already part of it). This is easy to implement since such scions are neighbors of the traced scion in the sorted array of scions.

Of course, this mechanism is not always correct: such scions may also be reachable from other generators, without being part of the cycle. As a consequence, we need do be able to correct such errors: if a scion is erroneously added to

[2]This can even be done when the stub *becomes* a sub-generator stub.

a sub-generators set, it will eventually receive a greater mark from another generator. As a consequence, the dominator of this scion will propagate to its associated scion a mark with a special black color, which will then be propagated again until the final generator.

When the generator receive its black mark, it increments a *new field* of its generator record, called the sub-generation counter. We immediatly update the order on marks: if two marks are from the same generator, but have different sub-generation counter, the greatest one is the mark with the highest sub-generation counter.

The mark change is important, since the white mark propagated by the incorrect sub-generator may erroneously lead to the collection of the graph. Moreover, this new field doesn't modify the order between marks from different generators: the scion erroneously included in the sub-generators set won't be included again, whereas other sub-generators will be included.

# 7. DISCUSSION

## 7.1 Fault-tolerance

Our algorithm is fault-tolerant: unreliable communications are supported, since the only message – the CYCLIVE message – can be lost, or delivered out-of-order. Indeed, this message always contains the full up-to-date information required for progress of the computation, and the computation is still conservative with older information. Race conditions are avoided thanks to an extension of the SSPC timestamp system.

Space failures can be easily handled by considering incoming references from suspected or crashed spaces as local roots. This approach is conservative, and only prevents the detection of cycles spanning on crashed spaces.

## 7.2 Ease of implementation

Our algorithm has been implemented in the JoCaml system [3], an extension of Objective CAML with mobile agents. The implementation only requires minor modifications of any local tracing garbage collector. In particular, local garbage collection has been kept incremental and locally optimal (no need for overlapping traces).

## 7.3 Resources consumption

### 7.3.1 Memory usage

Our algorithm does not add any space overhead for local objects. This is not the case of other cycle detectors, such as the back-tracing algorithm of Maheshawari and Liskov [11] which requires a Leader field per object to compute backward information.

Only stubs and scions structures are modified. Consequently, memory usage is only proportional to the number of stubs and scions in each space. Marks are quite complex, but most of them are root marks, which can be represented by long integers (for example 56 bits for time and 8 bits for distance).

### 7.3.2 Messages

In our algorithm, only one message is sent after each pair of local garbage collections. This message contains one mark for each stub (mainly root marks), and one mark for each scion involved in a sub-generation. Other cycles detectors also propagate some information for each stub, either a distance [10, 11], a timestamp [9, 5] or a list of locators [16]. Moreover, the CYCLIVE message can be shared with the acyclic garbage collector, for example to propagate the content of the LIVE message of the SSPC cleanup protocol [14].

### 7.3.3 Computation time

Scions must be sorted before each pair of local garbage collections. However, sorting scions before local garbage collections is also required by other detectors [10, 11, 9, 5] to correctly propagate either timestamps or distances.

Finally, a small amount of computation is needed for each stub to computes the mark it will propagate to its associated scion after a pair of local garbage collection. Thus, the total computation time is in $O(nstubs + nscions.\log(nscions))$

# 8. CONCLUSION

We have presented a new algorithm to detect free distributed cycles in large-scale networks. This new detector combines three new techniques: *min-max marking*, to detect different paths leading to a same object, *sub-generation*, to build a partial back-tracking graph upon a potentially free distributed cycle, and *optimistic back-tracing*, to retrieve back-references in an inexpensive but potentially erroneous way.

Our algorithm has good properties for large-scale networks: it is completely asynchronous, tolerant to space failures and unreliable communication, while preserving locality of detection; all cycles are detected in a same process, and it is economic in resources, such as computation time, memory usage and messages. Finally, it has been successfully implemented in our system, the JoCaml system [3].


### Acknowledgements

The author would like to thank Georges Gonthier, Damien Doligez, Marc Shapiro, Alan Schmitt and James Leifer for their valuable comments and suggestions on improving this paper.


# APPENDIX

# A. REFERENCES


[1] Birrell, A., Evers, D., Nelson, G., Owicki, S., and Wobber, E. Distributed garbage collection for network objects. Tech. Rep. 116, DEC SRC, Dec. 1993.

[2] Bishop, P. B. Computer systems with a very large address space and garbage collection. MIT Report LCS/TR–178, Laboratory for Computer Science, MIT, Cambridge, MA., May 1977.

[3] Conchon, S., and Fessant, F. L. Jocaml: mobile agents for objective-caml. In *Symp. on Agent Systems and Applications, Mobile Agents 1999. (ASA/MA99)* (Palm Springs, California (USA), oct 1999), IEEE Computer Society, pp. 22–29.

[4] Ferreira, P., Shapiro, M., Blondel, X., Fambon, O., Garcia, J., Kloosterman, S., Richer, N., Roberts, M., Sandakly, F., Coulouris, G., Dollimore, J., Guedes, P., Hagimont, D., and Krakowiak, S. PerDiS: design, implementation, and

use of a PERsistent DIstributed Store. Tech. Rep. INRIA RR 3525, Oct. 1998.

[5] Fessant, F. L., Piumarta, I., and Shapiro, M. An implementation of complete, asynchronous, distributed garbage collection. In *Conf. on Prog. Lang. Design and Impl. (PLDI)* (Montreal (Canada), June 1998), ACM SIGPLAN.

[6] Fisher, M., Lynch, N., and Patterson, M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2 (Apr. 1985), 274–382.

[7] Fuchs, M. Garbage collection on an open network. In *Proceedings of International Workshop on Memory Management* (Concurrent Engineering Research Center, West Virginia University, Morgantown, WV, Sept. 1995), H. Baker, Ed., vol. 986 of *Lecture Notes in Computer Science*, Springer-Verlag.

[8] Hughes, R. J. M. A distributed garbage collection algorithm. In *Record of the 1985 Conference on Functional Programming and Compute r Architecture* (Nancy, France, Sept. 1985), J.-P. Jouannaud, Ed., vol. 201 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 256–272.

[9] Lang, B., Queinnec, C., and Piquer, J. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Princip les of Programming Languages* (Jan. 1992), SIGPLAN Notices, ACM Press, pp. 39–50.

[10] Maheshwari, U., and Liskov, B. Collecting cyclic distributed garbage by controlled migration. In *Proc. of the Symp. on Principles of Distributed Computing* (1995).

[11] Maheshwari, U., and Liskov, B. Partitioned garbage collection of a large object store. In *Proc. of SIGMOD* (1997).

[12] Rodrigues, H. C. C. D., and Jones, R. E. A cyclic distributed garbage collector for Network Objects. In *Tenth International Workshop on Distributed Algorithms WDAG'96* (Bologna, Oct. 1996), O. Babaoglu and K. Marzullo, Eds., vol. 1151 of *Lecture Notes in Computer Science*, Springer-Verlag.

[13] Rodriguez-Riviera, G., and Russo, V. Cyclic distributed garbage collection without global synchronization in CORBA. In *OOPSLA '97 Workshop on Garbage Collection and Memory Management* (Oct. 1997), P. Dickman and P. R. Wilson, Eds.

[14] Shapiro, M., Dickman, P., and Plainfoss, D. Robust distributed references and acyclic garbage collection. In *Proc. 11th Symp. PODC* (Aug. 1992), ACM Press, pp. 135–146.

[15] Wollrath, A., Riggs, R., and Waldo, J. A distributed object model for the java system. In *Conf. on Object-Oriented Technologies* (Toronto Ontario (Canada), 1996), Usenix.

[16] Ye, X., and Keane, J. Collecting cyclic garbage in distributed systems. In *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)* (1997), IEEE.

## B. PSEUDO-CODE

```
type color = White | Gray

type generator = {
  creation : time;
  range : int;
  locator : locator;
}

type mark = {
  generator : generator;
  distance : int;
  color : color;
}

type stub = {
  scion_id : locator;
  last_recv : time;
  min_mark : mark;
  max_mark : mark;
  scion_mark : mark;
}

type scion = {
  scion_id : locator;
  last_sent : time;
  stub_mark : mark;
  mark : mark;
  is_generator : bool;
  target : object;
}
```

**Figure 7: Record structures for generators, marks, stubs and scions.**

```
min_max_trace(roots, scions)
{
  root_mark.creation := current_time();
  foreach r in roots, trace(r, root_mark);
  if(trace_min){
    for(nscion=last_scion; nscion>0; nscion--)
        trace(scions[nscion], scions[nscion].mark);
    trace_min := false;
    send_cyclive_msgs();
  } else {
    commit_new_received_marks(scions);
    sort(scions);
    for(nscion=1; nscion≤ last_scion; nscion++)
        trace(scions[nscion], scions[nscion].mark);
    trace_min := true;
} }

trace(r,mark)
{
  if(not_traced(r)) {
    if(is_stub(r)){
      if(trace_min) { r.min_mark := mark; }
      else { r.max_mark := mark; }
    } else {
      foreach o in object(r), trace(o,mark);
} } }
```

**Figure 8: The local garbage collector with min-max marking. This code can easily be modified to be executed incrementally. `commit_new_received_marks` commits `stub_mark` to `mark` in each scion before sorting the scions.**

```
scion_mark(stub)
{
  min_mark := stub.min_mark;
  max_mark := stub.max_mark;
  mark := max_mark;
  if(min_mark.generator = max_mark.generator){
    mark.distance :=
      min(min_mark.distance, max_mark.distance);
  } else { mark.color := gray; }
  return mark;
}
```

**Figure 9: The computation of the mark propagated by a stub**

```
type color = White | Gray

type generator = {
  creation : time;
  range : int;
  locator : locator;
  generation : int;
}

type mark = {
  generator : generator;
  distance : int;
  color : color;
}

type stub = {
  scion_id : locator;
  last_recv : time;
  min_mark : mark;
  max_mark : mark;
  scion_mark : mark;
  sub_mark : mark;
  sub_gens : scion list;
}

type scion = {
  scion_id : locator;
  last_sent : time;
  stub_mark : mark;
  mark : mark;
  is_generator : bool;
  target : object;
  sub_mark : mark;
}
```

**Figure 10: Record structures for the generators, marks, stubs and scions with sub-generation. Each stub keeps the list of its dominated sub-generators, and the mark of the real generator it is involved in a sub-generation.**

```
trace(r,mark)
{
  if(not_traced(r)) {
    if(is_stub(r)){
      if(trace_min) { r.min_mark := mark; }
      else {
        r.max_mark := mark;
        scion := scions[nscion];
        if(scion.sub_mark = NULL
           && r.sub_mark.generator = mark.generator
           && mark.color = gray) {
          r.sub_gens =
            r.sub_gens  ∪ { scion; }
          scion.sub_mark := r.sub_mark;
          scion.is_generator := true;
    } } } else {
      foreach o in object(r), trace(o,mark);
} } }

scion_mark(stub)
{
  min_mark := stub.min_mark;
  max_mark := stub.max_mark;
  mark := max_mark;
  if(min_mark.generator = max_mark.generator){
    mark.distance :=
     min(min_mark.distance, max_mark.distance);
    if(∃ scion ∈ stub.sub_gens,
        scion.stub_mark.color = gray) {
      mark.color := gray; }
  } else {
    mark.color := gray;
  }
  return mark;
}
```

**Figure 11: The trace with sub-generation. During each trace, scions may become sub-generators. The computation of the mark propagated from the stub to its associated scion in `scion_mark` takes both the mark received during the local trace and the marks on dominated sub-generators into account.**

```
commit_stub_mark(scion)
{
  stub_mark := scion.stub_mark;
  stub_mark.distance := stub_mark.distance + 1;
  if(scion.is_generator
    && stub_mark.generator > scion.mark.generator){
    scion.is_generator := false;
    scion.sub_mark := NULL;
  }
  if(not scion.is_generator){
    if(stub_mark.distance = stub_mark.generator.range){
      scion.is_generator := true;
      scion.mark := {
       distance = 0;
       generator = {
         locator = scion.locator;
         range = stub_mark.range + offset(stub_mark.range);
         creation = current_time();
       }
       color = White;
       generation = 0;
     }
    } else {
      scion.mark := stub_mark;
    }
  } else {
    if(stub_mark.generator = scion.mark.generator){
      if(scion.mark.locator = scion.locator
    && stub_mark.color = White) { cycle_detected(scion); }
      else { scion.sub_mark := scion.mark }
    }
  }
}
```

**Figure 12: Before sorting the scions, the latest received marks are commited. At this state, old generators may be removed, new generators created, or a sub-generation started from the generator.**