Experience with a Fault-Tolerant Garbage Collector in a Distributed Lisp System

Final version sent to IWMM-92

David Plainfossé and Marc Shapiro

INRIA Project SOR, Rocquencourt BP 105, 78153 Le Chesnay CEDEX, FRANCE
David.Plainfosse@inria.fr

Abstract. In order to evaluate our fault-tolerant distributed garbage collection protocol, we have built a prototype implementation within a distributed Lisp system, *Transpive*, replacing Piquer's native indirect reference count distributed garbage collector. This paper presents our protocol and highlights implementation issues on Transpive. In particular, we describe the prototype and the alterations required to fit into the Transpive distributed programming model. The message and CPU performance of our protocol are measured and its fault-tolerance evaluated. We conclude that the cost of our protocol is close to Piquers's , although our protocol has greater functionality.

1 Introduction

Garbage collection (GC) has recently become of increasing interest in distributed systems [6, 9]. The motivations for such a service are numerous. First, transparency: Just as modern distributed systems support transparent, uniform placement of and invocation on both local and remote objects, so should they also support transparent object management, including reclamation. Second, storage management is a complex task, not to be managed by users. Distributed GC is even harder than local GC because the local collectors must be coordinated, to consistently keep track of changing references between spaces. This consistency problem is further complicated by the common failures of distributed systems such as lost, duplicated, and late messages, and crashes of individual spaces.

Distributed garbage collection poses a challenging problem: reclaiming all kinds of data structures while achieving efficiency, scalability and fault-tolerance. In spite of the difficulty, a number of proposals have attempted to design a distributed GC that fulfills all these requirements. The great number of incomplete proposals (see Sect. 6) reflects how difficult the challenge is. However, the combination of several complementary techniques may lead to an almost perfect algorithm. For instance, combining Lang *et al.* cyclic distributed GC [6] with our fault-tolerant algorithm could gain a fault-tolerant and cyclic garbage collector.

To address these issues, we have designed a fault-tolerant distributed garbage collector protocol, hereafter called the SGP protocol [13, 14] based on reasonable, weak assumptions. It scales to any number of nodes. It continues to function correctly

in the presence of lost, duplicated, or out-of-order messages, or of (fail-stop) node crashes; it allows objects to migrate or become deleted while referenced.

In order to evaluate the SGP protocol, we have prototyped it on a distributed Lisp, Transpive [12], implemented at INRIA, running on a multi-Transputer board hosted by a Sun server machine. For the purpose of this evaluation, we replaced Piquer's original Indirect Reference Count (IRC) garbage collector [11], provided with Transpive, with a prototype implementation of the SGP protocol. SGP provides all the functionality of Piquer's GC, and in addition is resilient to message or site failures. The motivations for this approach are the following:

- ease of prototyping the algorithm in a functional language,
- use of an existing, easy-to-use, clean, distributed programming environment,
- existence of a local tracing collector as required by SGP,
- possibility of comparison with Piquer's GC.

The organization of this paper is the following. Section 2 describes briefly the SGP protocol, and highlights mechanisms implemented on Transpive. Section 3 reviews the distributed programming model of Transpive and its implementation. In particular, issues relevant to the SGP implementation are highlighted. Then, in Sect. 4, we further describe the implementation itself. Section 5, presents performance measurements of our prototype implementation. Section 7 concludes the paper. We compare our performance results with Piquer's.

2 Brief Description of the SGP protocol

We consider a collection of *spaces* connected by an unreliable non-FIFO channels. A space is either a process, a processor or a group of machines. Spaces may contain one or more applications called *mutators* performing independant computations. Mutators allocates dynamically objects in their space. An Object is located in a single space but may migrate. Objects may contain references to other objects located in the same or in remote spaces. Local objects are accessed through standard pointers whereas remote objects are accessed via remote pointers. An object accessible from at least one remote space is called *public*, as opposed to *private* objects. A public object belongs to a single space, its *owner*. Public and private objects are dynamic sets. That is, any non-garbage private object may become public and vice-versa. For instance, a public object only remotely referred by a single space may migrate to that space. After migration, the object is considered as private. Basically, the distributed GC is in charge of tracking remote accessibility of public objects. Private objects do not concern us and are reclaimed by the local GCs.

The mutator rests upon two separate layers of object management (see Fig. 1). The bottom layer is independent of object semantics, structure, or programming language: this is the distributed garbage collection specified in [14] and described briefly herein. The distributed garbage collection only propagates accessibility information supplied by the upper layer.

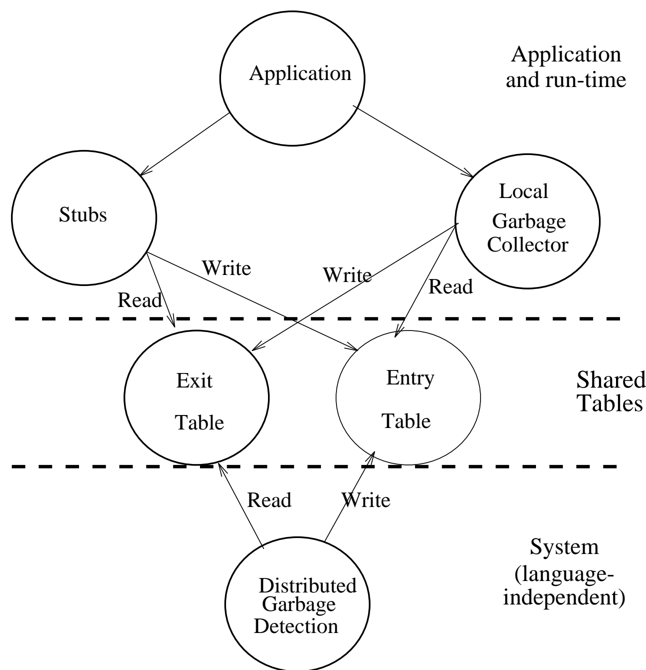

Fig. 1. Relationships between processes and main data structures

The upper layer is a (language-specific) run-time, extended to interface with our distributed GC. In the upper layer, one finds storage management (object allocation, and local tracing garbage collection) as well as remote invocation functions (i.e. communication stubs).

The two layers share information in the form of incoming and outgoing references. An incoming reference is called an *entry item* and an outgoing reference is called an *exit item*. Cooperation between layers is limited to simple interactions to maintain consistency between entry and exit items.

Mutators in different spaces communicate via RPC-style invocation, i.e. by messages. An invocation is mediated by mechanically-generated stubs for marshalling and unmarshalling messages; a stub interfaces between the application and the system, encoding typed information into a typeless form. The arguments and results in an invocation contain any mixture of pure data, references, and migrating objects. When sending or receiving a message, the stub creates either an entry or exit item for the reference or the object embedded in the message.

To provide fault-tolerance, extra time and ownership information is piggy-backed onto the existing mutator messages. Occasional control messages are exchanged, in the background, to remove inaccessible entry items.

The SGP protocol relies on the existence of any standard local tracing garbage collector. The distributed protocol is based on a conservative extension of reference counting. Each space maintains a list of potential incoming and outgoing references,

respectively called the *entry table* and *exit table*. Both the entry table and the exit table are conservative estimates. If two different spaces possibly refer to a single object of space A, each will be assigned an entry item in A. This differs from reference counting, and in particular from Piquer's IRC, because we need an entry per remote space to deal with unreliable communication. This policy renders entry item deletion an idempotent operation and permit tolerating lost or duplicated message. In the former case any subsequent control messages received will allow us to reclaim previously garbage entry items. The latter case will have no effect since all garbage entry items would have been been previously collected.

Local garbage collection proceeds from the union of the local root and the entry tables and removes object and entries in the exit table. Since local GC starts from the union of the local root with the (conservatively estimated) entry table, all non-reachable local objects are true garbage. Each local GC cleans the entry table of useless entry items. In turn, exit tables are used to clean remote entry table, yielding successively better estimates.

When an exit item on space A is deleted, the corresponding entry item on space B can be removed. To this effect, a *delete message* can be sent from space A to space B. However this message can be duplicated or lost. To guard against loss, periodic *use messages* are sent from A to B containing the list of all existing exit items on A pointing to B; by comparison space B can deduce entry items that are not reachable, and remove them. In the remainder of the paper, *control message* refers to both use and delete messages.

One common problem in distributed systems is the message delivery delay. Messages containing references must be taken into account to guard against unsafe reclamation. Suppose that one space B sends a message to a space A containing a reference to a given object, say x. At the same time, a control message is sent from space A to space B to inform that the remote pointer on object x has been discarded. If object x is not locally referenced upon receiving the delete message, it will be remove from the entry table and collected at the next local GC.

To avoid this problem, we keep on each space a vector of highest timestamps and we timestamp entry items. When sending a reference, the stub creates the entry items and store in it the value of the local clock. The same value is used to timestamp the mutator message. Upon receiving a mutator message, the receiver compares the timestamp value extracted from the message with the one found in the vector of highest timestamps. This vector contains a space identifier and an associated timestamp for each remote space. A timestamp is increased each time a message is received. If the corresponding entry in the vector does not yet exist the initial value can be taken from the message. Control messages carry the current value of the timestamp vector corresponding to the target space. Upon receiving a control message, the timestamp value found in the message is compared to the value in the entry items to detect messages in transit.

Since our distributed protocol is based on reference counting, it fails to collect cycles[1].

[1] A separate sub-protocol [14] copes with inter-space cycles but its description is out the scope of this paper as it has not been implemented on this prototype.

3 Transpive

A garbage collector interacts closely with the programming model, as shown in Sect. 2. In particular, the way references are created, copied and sent is a crucial issue. For this reason, we first describe the programming model of Transpive, concentrating on key points related to the SGP implementation. Transpive is a distributed Lisp designed to provide a programming model as close as possible to a centralized Lisp, and in particular:

- to provide location-transparent invocation,
- to supply the basic functionality required by a distributed application through a small number of concepts,
- to provide a set of extensions, easily portable to another Lisp or runtime systems.

Transpive is layered on a Lisp interpreter.[2] One Lisp interpreter runs on each Transputer processor and interacts with the others through message passing. The underlying runtime system ensures FIFO, reliable message channels. Consequently, we have simulated message failures to evaluate the fault-tolerance aspects of SGP.

3.1 Sending and Receiving Messages

Transpive provides a function, **`ext-send()`**, to send a typed message to Transpive thread, addressed by a identifier **`target_id`** and a port number **`port_n`**. The **`function`** argument is used for marshalling/unmarshalling the **`msg`** given as argument. We have extended this function to accept an added argument:

```
ext-send (msg function  thread_id port_n delay)
```

The last argument, **`delay`**, simulates messages failures: out-of-order, delayed, lost, or duplicated messages. The target thread **`thred_id`** receives the message by calling the function **`receive_from_any`**:

```
msg := receive_from_any()
```

A Transpive message is a structure composed of several fields :

```
struct msg {
     data        ; the message data
     source      ; the sender thread_id
     target      ; the target thread_id
     send_type   ; the type of the message
     function    ; function for marshalling and  unmarshalling
     ; additional SGP fields:
     timestamp   ; value of the sender Lisp local clock
     delay       ; simulates unreliable messages}
```

[2] The current implementation runs on Le_Lisp [4], a fast Lisp interpreter implemented at INRIA. But the distributed model of Transpive is generic and easily portable to another Lisp dialect or functional language.

All these fields have default values. The `function` is used to marshal and unmarshal the object referenced by the `data` field. Several alternative marshalling semantics are provided by Transpive. Transpive servers use an efficient marshalling function `low_level` which does not generate remote pointers but copies values to the target Lisp.

We have added two fields to the standard Transpive message structure. The SGP protocol timestamps mutator messages to protect against unsafe, late or duplicate messages (see Sect. 2). The `timestamp` field is managed by the stubs.[3] This extension has no consequence on the other message functions. The field `delay` is used to simulate message failures, and is set from the `delay` argument of `ext-send()`.

The effect of the `delay` value is the following:

0 corresponds to the default normal delivering FIFO order,
$+n$ corresponds to a delayed message,
-1 corresponds to a lost message,
-2 corresponds to a duplicated message.

The delay is enforced by the `receive_from_any()` function which delivers the message to the application according to the value of the `delay` field. The $+n$ value indicates the number of times the messages is read in the queue without being delivered to the corresponding thread.

3.2 Remote References

Transpive supports transparent fined-grained object sharing. Lisp is a typeless language which only manipulates cons cells. Consequently, Transpive allows one to pass and access remotely any cons cell. The creation of remote references is totally transparent to the programmer. The corresponding data structures are created as a side effect of message passing. Specifically, stubs are responsible for detecting cons cells in messages and creating the corresponding entry or exit items to access the remotely referenced objects. Lisp does make any distinction between references and plain objects. Therefore, in that model, a reference is created for each cons cell passed in the message. Thus, each cons cell of a list may be accessed independently from other cells. This policy is required to keep the same semantics as any local Lisp. However, it creates a large amount of exit and entry items and worsens locality.

Transpive provides a cache memory associated with remote references. On first access to a public object through a remote reference, a replica of the object is copied to the local cache of the referencing Lisp. All subsequent read accesses to this object will be local, in the cache.

Conversely, an attempt to write a replica invalidates all other replicas. Ownership of the object is migrated to the Lisp which has attempted the write access. This scheme is well adapted to functional languages, such as Lisp, where read accesses are much more frequent than writes.

[3] Actually, the timestamp field is initialized at creation of a message. Stubs are responsible for updating the timestamp associated with each descriptor as explained in Sect. 2.

A public object always points to a *descriptor*. For this purpose, Transpive objects have been extended with an extra field, called *back pointer*, to access their corresponding descriptor. In order to save space, plain private objects don't refer to any descriptor and their back pointer is set to NULL. Depending on the existence or not of a cached replica, a descriptor acts either as a local handler on its cached replica, or as a remote pointer to a public object. With respect to the SGP model, a Transpive descriptor acts partly both as an entry and an exit item. It contains the following fields when corresponding to an exit items:

- The identification of the owner Lisp where the object is located,
- an OID which uniquely identifies the object throughout the system,
- a status, indicating whether the cached replica is valid or not,
- a "weak pointer" to the local replica; this pointer is not taken into account by the local garbage collector. Initially, this pointer is set to NULL. It is updated upon receiving a copy.

An additional field is present when the descriptor acts as an entry item:

- a list of pairs ((lisp_id timestamp) ...). The first one identifies a Lisp holds holds a remote pointer to that particular object. The second one is increased each time a message containing a reference to that object is sent to that lisp.

Each Lisp maintains a table of valid descriptors (TOD) indexed by OID of public objects. As a descriptor contains only a weak pointer to the local replica, another data structure, the list of public objects (LPO), is maintained to prevent public objects from being collected by the local GC. When an object becomes public, it is added to the LPO. When the last remote reference to this object is discarded, and becomes again private, it is removed from the LPO.

Upon sending a Lisp list composed of cells, the stub generates an OID and allocates a descriptor for each cell of the list. For instance, a call to `ext-send((1 2 3))` will lead to generate three OIDs and three descriptors for `(1 2 3)`, `(2 3)` and `(3)`. This is inherent in the Lisp object model and unfortunately leads to large space overhead.

Transpive provides, along with descriptors, a number of functions to access and set each field of a descriptor. These functions will be used in the remainder of this paper and are introduced to improve readability of source Lisp code. They are listed below in the same order as the list of fields given above:

```
desc:= get_desc(obj)          ; reads obj's back-pointer and returns descriptor
get_owner(get_desc(obj))      ; returns owner field of obj's descrpitor
get_oid(desc)                 ; get oid's field of desc
get_replica (desc)            ; get cached replica through weak pointer
set_owner(desc, lisp_id)      ; set owner field with the Lisp identifier lisp_id
set_oid(desc, oid)            ; set oid field of the argument descriptor desc
get_timestamp(desc, lisp_id) ; get the timestamp value embodies in
the descriptor for a particular Lisp
```

The function `get_desc(obj)` takes an object as argument and returns its back pointer (i.e. its descriptor). The function `get_replica(desc)` returns the cached replica object (if it exists) of the descriptor `desc`.[4]

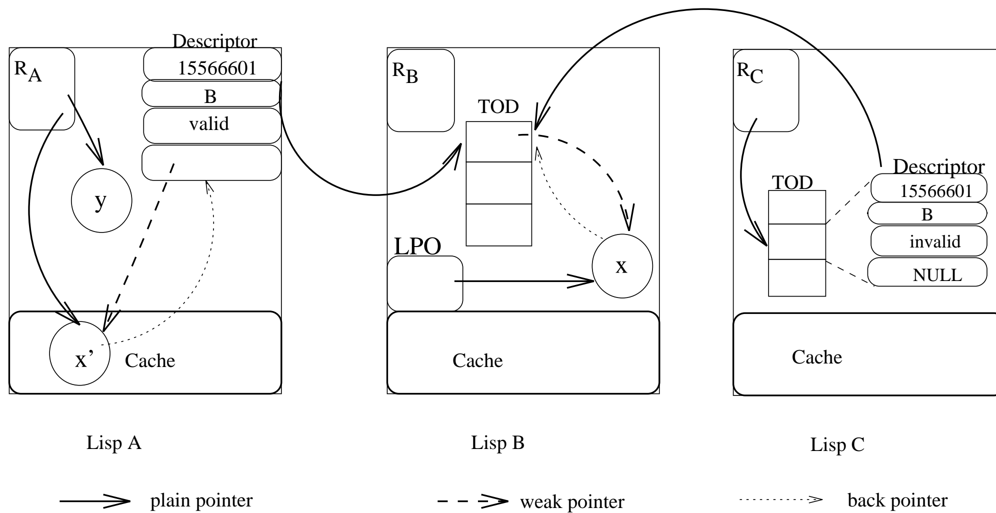

Fig. 2. remote references, descriptors and cache memory

Figure 2 shows three Lisps running on three different Transputers. Object x, owned by Lisp B is public and remotely accessible from Lisps A and C. Lisp A has already accessed object x from some root R_A and therefore has a replica of x in its local cache. A's reference to x points directly to the cached replica x'. In contrast, Lisp C has never accessed object x although it is accessible from its root R_C. Consequently, the reference to x points to the corresponding descriptor. Object y is a plain private object accessible from its local root R_A. Note that y does not refer to any descriptor because it is not public.

We have extended this descriptor to handle administrative information specific to the SGP protocol. As stated in Sect. 2, the SGP model assumes one entry item per remote space. In Transpive, a single descriptor may be referenced by several remote Lisps, and a counter associated to each descriptor embodies the corresponding reference count. We have adapted the SGP model to fit into the Transpive implementation of remote references. A new field has been added to each descriptor, pointing to a list of pairs. Each pair contains a Lisp identifier and a timestamp value. The Lisp identifier refers to a Lisp which remotely points to the corresponding public object. The timestamp value is updated each time a reference to the object is sent to this Lisp. The counter has been retained for compatibility but serves no useful purpose.[5]

[4] We tried to use the same variables names in the paper. In particular a descriptor will always be named as `desc` in pseudo code.

[5] However, we are in the process of removing them to compare memory consumption

4 Prototyping the SGP on Transpive

In the SGP protocol, a remote reference is created when a mutator passes a reference in a message. In other words, a creation message is a mutator message containing at least one reference. A *use message* (see Sect. 2) is sent by our collector to inform the owner Lisp which remote references have been discarded. We briefly describe here how we have implemented our protocol using the Transpive mechanisms introduced in Sect. 3.

4.1 Timestamps

Each Lisp maintains a vector of highest timestamps called the HTS vector. The HTS vector is updated each time a Lisp receives a mutator message. To handle the HTS vector, we have modified the original Transpive server of messages. This server receives all the messages exchanged between mutators and forwards them to the target thread. Actually, a call to the **ext-send()** function sends a message to a target thread via this server. Each time the message server receives a mutator message it extracts the timestamp, updates the corresponding entry of the HTS vector, then queues the message for the receiver.

```
PROCEDURE server_msg()
   msg : message;
   WHILE true do
        msg := receive_from_any()
        IF msg.timestamp GREATER THAN hts[msg.sender] THEN
           hst[msg.sender] := msg.timestamp
           queue the message to the target thread
        ELSE
           ignore the message
END END END
```

4.2 Cleanup of Public Objects

In Transpive, garbage collection of a descriptor occurs in several steps. Figure 3, shows the sequence of events involved in the collection of a public object. As stated earlier in Sect. 3, a descriptor is useless when the back pointer of its local replica does not reference it. Here are the relevant events:

1. On Lisp A the last reference to the local replica x' of x is discarded.
2. On Lisp A, a local GC occurs. The replica is collected and its descriptor pointer is updated.
3. The matching descriptor on Lisp A is then collected by the cleaning function **cleanup_tod**.
4. Consequently, a control (use) message is sent to the owner Lisp B.

---

between the SGP and IRC protocols.

5. On Lisp B, the SGP server receives the delete message and removes the corresponding object from the LPO.
6. On Lisp B, public object x is not locally referenced. It will be collected at the next local GC.

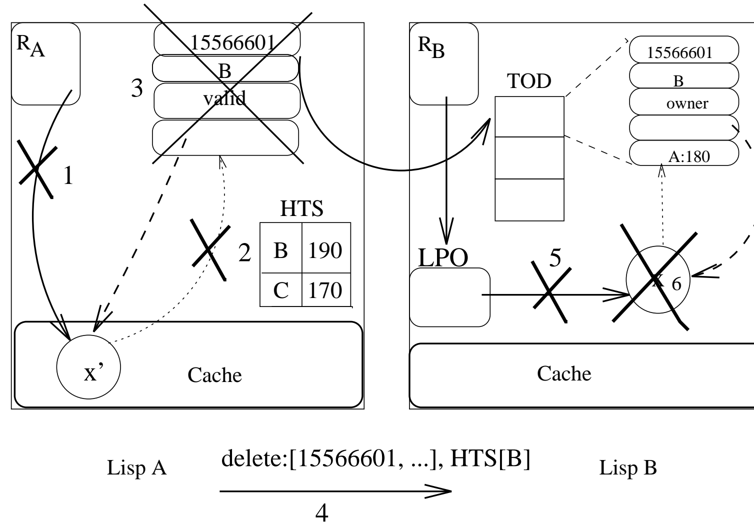

Fig. 3. Chronology of events in garbage collecting public objects x of Lisp B

SGP model assumed that entry items were collected by the local GC. The distributed programming model of Transpive enforces a totally different scheme for collecting descriptors. As stated earlier in Sect. 3.2, SGP's exit table is modelled by the TOD in the Transpive model and exit items are represented by descriptors. In contrast to exit items, a descriptor refers either a local object or a replica. Conversely each public object or replica points to its descriptor through its back pointer. Consequently, a descriptor cannot be collected as long as its replica is not collected. For this reason, garbage collection of descriptors proceeds in two steps. First, the local cached replica is collected and its back pointer is set to NULL. Later, the previously pointed descriptor is considered as garbage and will be collected by a cleanup function as shown by the code below:

```
FUNCTION cleanup_tod() : descriptors_list
  BEGIN critical section
      FOREACH desc IN TOD DO
         IF replica's back pointer refers to desc THEN
            add desc's OID to the use_list for the corresponding owner_lisp
         ELSE
            removes desc from the tod
      END END
   END critical section
```

```
   return TOD;
END
```

This function `cleanup_tod()`, cleans the TOD by removing useless descriptors. Each time a descriptor is detected, its OID is added to the `oids_list` if the descriptor is still valid. As an optimization, the messages are not sent at once to the corresponding Lisp, but rather buffered.[6] A high priority Transpive daemon `flush_msg_list` is responsible for traversing the list of messages and sending control messages as shown by the code below:

```
PROCEDURE flush_msg_list()
  BEGIN critical section
   FOREACH (target_lisp oids_list) pair IN use_list DO
     msg.timestamp := get_hts(target_lisp)
     msg.data := oids_list
     ext-send(msg, low_level(), target_lisp, GC_PORT, random())
   END
   reset use_list to nil
 END critical section
END
```

A control message is composed of the following fields:

- the corresponding entry of the vector of highest timestamps `hts`,
- the list, `l_obj_id`, of valid OIDs depending

Message lists are composed of pairs `(lisp_id l_obj_id)`. The first element is a lisp identifier and the second a list of OIDs. The function traverses the whole `use_list` and sends to each target Lisp a corresponding subset of the valid OIDs. Note that `ext-send` calls are done with a delay argument. The function `random()`, in the code fragment above, generates a random value corresponding to either a delay, a lost, or a duplicated message. The function `get_hts` is used to timestamp the control messages and prevents a public object to be discarding if a reference is in transit (see Sect. 2). All these control messages are sent to a specific Transpive port associated with our SGP server. Note that we have to bypass the normal reference marshalling scheme in order to avoid the creation of remote pointers when sending control messages. The `low_level` Transpive marshalling function is used to avoid the creation of descriptors. This function marshalles control messages as a vector of integers and bypassed the traditional reference sending layer.

4.3 SGP Server

Each Lisp runs a server dedicated to receiving and processing control (delete and use) messages. The former kind contains a vector of unreachable OIDs whereas the latter contains whole sublist of the reachable OIDs between two Lisps (see Sect. 2).

[6] We haven't tried yet to piggy back delete or use messages on mutator messages.

The sgp_server is activated each time a control message is sent from some remote Lisp by the flush_msg_list mentioned above. It extracts the relevant components of the control message and forwards them to appropriate function to update the local TOD.

```
PROCEDURE sgp_server()
   WHILE true DO
      ; wait for control message
      msg := receive_from_any()
      oids_list := msg.data
      msg_timestamp := msg.timestamp
      ; process control messages
      FOREACH oid IN oid_list
         desc := get_desc(oid)
         IF get_timestamp(desc) >= msg_timestamp THEN
            delete_desc(desc)
         ELSE
            a message in transit contains a reference to this descriptor
END END END END
```

4.4 SGP Interface with local GC

Le_Lisp provides an number of system signals. For instance, the signal gcalarm is activated just after each local garbage collection. This signal can be used to check the collection process to detect, for instance, a memory overflow. This signal invokes a user-defined function, which is a Null function by default. In our case, this function is responsible for cleaning the TOD and the LPO . Although it was stated in [14] that this cleanup could occur in parallel with other processing (only update of individual elements needs to be atomic), we implemented the whole procedure in a critical section as a quick first approximation.

```
PROCEDURE gcarlarm_sgp()
BEGIN
   BEGIN critical section
      ; removes from LPO previous public objects
      LPO := cleanup_lpo(LPO)
      ; cleanup the TOD of useless descriptors
      TOD := cleanup_tod(TOD)
   END critical section
END
```

The problem with this scheme is that the cleanup of remote pointers is bound to some local GC. This can lead to a memory overflow (in both implementations i.e Piquer's and SGP) since TOD cleanup is always delayed until after a local GC at the remote Lisp. This problem arises when a Lisp holds many remote pointers but uses only a small amount of its local memory. Since local GC is invoked on the basis of memory use, garbage remote references may not be collected for a long while. As a result, a high number of potentially garbage public objects are not collected,

leading to a memory overflow. To avoid this problem, the cleanup protocols should be called not only after local GC, but also periodically.

5 Experiments

In this section, we analyse the measured performance of our SGP prototype and compare it with the IRC implementation. Two kinds of performance are discussed: the number of messages and their frequency andthe CPU overhead due to both kind of distributed GC.

5.1 CPU overhead

We have measured the CPU overhead due to our SGP implementation. We compare these results with the native distributed GC of Transpive. We have run two applications: a merge sort and a matrix multiplication. The measurements were taken on a Parsytec board composed of four Transputers (T800) with one megabyte of memory each, hosted by a Sun. We have measured each application twice on the same data to take into account the copies of objects. Since Transpive copies public objects, results are always better the second time. However these measures have been repeated dozens of times to be sure of the results and have shown extremely low variance.

Table 1. CPU performance measurements

Application	CPU time in seconds						Overhead (%)	
	Without DGC		IRC		SGP		SGP/IRC	
(sort 100)	3.8	3.2	4.7	3.9	5.5	4.1	17%	5%
(sort 200)	5.6	4.4	6.7	5.2	8.1	5.9	20%	12%
(multmat 20 20)	11.1	7.8	12.0	8.7	13.5	9.8	11.9%	12.3%

Table 1 shows the performance measurements. The results conform with Piquer's. We have disconnected the function responsible for sending control messages on each Lisp in order to avoid interrupting applications, but we kept all the control data management in order to measure the overhead on mutators until sending control messsage. The overhead measured is due to managing control data structures this represents the mutator part of the SGP protocol. Our SGP implementation is on average 10% slower than Piquer's and 20% slower than without any DGC. This slight overhead is encouraging. First, our basic motivation was to evaluate the SGP prototype; as a consequence, we did not pay too much attention to optimizations and kept a big part of Piquer's data structure management (for compatibility reasons). Second, the fault-tolerance property of the SGP protocol requires a some additional work, compared with Piquer's approach which largely justifies some added cost. For instance, we update descriptor timestamps each time a reference is sent.

5.2 Message Overhead

A second kind of measurements concerns the number of control messages sent, and their frequency. Our message sending protocol is different from Piquer's and slows down local processing a little because group OIDs into a single structure, instead of sending a unique OID per control message. We have chosen, the former policy because it reduces message traffic. As shown on Table 2, this "buffering" strategy dramatically reduces the number of control messages sent in SGP compared with IRC protocol. Note that the number of control messages sent does vary a little between the two executions. Note also that we obtain the same results whatever the size of the list in the merge sort application. This shows that our message sending policy is somewhat independent of the number of objects sent between Lisps, although this buffering strategy may retain a big amount of floating garbage. For that reasons, this strategy should not have two much impact on control message frequency. This is particulary true when locality is very poor and the number of remote references is large, such as in Transpive.

Table 2. Control message measurements

Application	Control Messages				IRC - SGP	
	IRC		SGP			
(sort 100)	31	28	10	8	21	20
(sort 200)	41	39	10	8	31	31
(multmat 20 20)	101	96	20	18	81	78

6 Related Work

Distributed garbage collection is a difficult problem which has only been addressed partially. One key reason is that while most proposals rely on centralized techniques, adapting such techniques to distributed environments is not a straightforward task. Stop the world algorithms require costly termination mechanisms when facing distribution, whereas reference counting is completely defeated by common messages failures. In order to adapt those techniques to distributed environments, many recent proposals try to relax traditional invariants [2, 11, 15] whereas others rely on reliable communication protocols [3, 6, 10]. The former family algorithms is usually based on reference counting. Therefore they cannot garbage collect distributed cycles and must assume that such graphs are rare. The second family ensure better liveness but all known algorithms are not resilient to message failures [6], may be completely defeated by space failures [3], or fail to address large network [9]. Our protocol belongs to the former family and bears some similarities to a number of proposals based on reference counting [2, 11]. Unlike those approaches, however, we maintain an entry item per source space that permits us to tolerate message loss whilst avoiding the dangers of duplicated messages.

Dickman [2] proposes *Optimizing Weighted References Counting* improving traditional *Weighted Reference Counting* [1, 15] in two aspects: message failures resilience and indirection cells. Resilience to message failures is provided through a weak invariant that requires that each object weight (total weight) is always greater or equal to the sum of all remote reference weights (partial weights). The weak invariant permit tolerating message loss but duplicated message remains problematic. The algorithm avoid the creation of indirections cells when partial weights cannot be split. However, this is enforced through a special **`null weight`** value. In this case, the total weight is always greater than the sum of partial weights preventing the object from being reclaimed by error. However, liveness is not ensured for *weak* objects which conform only the weak invariant. For this reason, the author assumes than the algorithm is always used in conjunction with a global tracing collector to reclaim garbage distributed cycles and weak objects.

In [11] Piquer describes his Indirect Reference Count (IRC) algorithm which improves Weighted Reference Count [1] by avoiding indirection cells. The algorithm also eliminates the need for increment messages that may conflict with decrement messages in traditional schemes. Thus, creation and duplication of a remote pointer are performed locally without informing the space where the object is located. In order to achieve local creation/duplication, remote pointers have been extended with a new field, named an indirect pointer. The indirect pointer serves only distributed GC purposes, and refers either to an object or to another remote pointer. The whole set of remote pointers referencing a single object forms a distributed graph which can be traversed using indirect pointers. Mutators never use indirect pointers, instead relying on the direct pointers to access objects in a single hop. As with others proposals relying on reference counting, the IRC algorithm is not resilient to message failures: liveness is not enforced against message loss and safety is not preserved against duplicated message.

Mancini and Shrivastava [10] describes an efficient and fault-tolerant distributed garbage collector based on reference counting. Resilience to space or message failures is supported granted to an RPC mechanism extended with detection and killing of orphans. A special protocol is used to cope with duplication of remote references. This protocol makes an early short-cut of potential indirections even if they are never used. Two alternatives are proposed to deal with distributed cycles : traditional and inefficient global mark and scan, and per object cycle detection based on an heuristic. The first one is notoriously inefficient and the second one does not collect all cycles.

Hughes [3] describes an elegant algorithm based on timestamps and local tracing. The algorithm timestamps objects and relies on the premise that garbage objects' timestamps remain constant whereas non-garbage objects' timestamps increase monotically. A timestamp threshold is computed to distinguish garbage from non-garbage objects. Objects that carry timestamps less than the threshold can be safely reclaimed. Unfortunately, the threshold computation relies on a termination algorithm which is notoriously costly and not scalable. Moreover, the algorithm is not resilient to space failures since a failed space prevents increasing the threshold, hence blocking garbage collection on all other nodes.

In contrast to many proposals that attempt to compute on each space the global accessibility of objects. Liskov and Ladin [9] rely on their highly available central-

ized service to compute global accessibility of objects on a single space. This service is physically replicated, hence achieving high availability and fault-tolerance. All objects and tables are assumed to be backed up in stable storage. Clocks are synchronized and message delivery delay is bounded. These assumptions allow the centralized service to build a consistent view of the distributed system. Each local collector informs the centralized service about incoming and outgoing references, and about the paths between incoming and outgoing references. The path computation is expensive but necessary for reclamation of distributed garbage cycles. Based on the paths transmitted, the centralized service builds the graph of inter-site references, and detects garbage (including dead cycles) with a standard tracing algorithm. The centralized service informs LGCs of accessibility of objects.

In a later paper [5] Ladin and Liskov simplify and correct the deficiencies of the above proposal, adopting Hughes' algorithm and loosely synchronized local clocks. Hughes' algorithm eliminates inter-space cycles of garbage, thereby eliminating the need for an accurate computation of the paths and for the central service to maintain an image of the global references. Furthermore, the centralized service determines the garbage threshold date, making a termination protocol unnecessary.

Recently Lang *et al.* [6] describe an original proposal to combine reference count and mark and sweep. The algorithm collect distributed cycles within predefined groups. Groups are dynamic collections of spaces (i.e a space may be removed or added during garbage collection) and may overlap or include other groups. The algorithm relies both on counters and local GC to perform mark and sweep within a group. Reference counts must be kept accurate, hence message failures are not tolerated. Group GC is conservative with respect to inter-group references: any subgraph referenced from outside the group is not collected until a larger group is formed encompassing the entire graph; therefore liveness is not guaranteed. Thus, large cycle reclamation requires extending group size such that the group includes all spaces that hold a cycle vertex. Distributed garbage collection of very large networks is proposed through a hierarchy of included groups. Included groups benefit from larger groups GC that perform some of their work. However, large group GCs are longer than smaller ones and therefore retain more floating garbage. For that reason, the authors assume that large group GCs are rare compared to small group GCs.

In [8] Lins and Jones combine *Weighed Reference Counting* with Lins'local algorithm for *Cyclic Reference Counting* [7] to address distribution issues. As a result, they propose a simple algorithm to garbage collect cycles in a distributed environment. The general idea of the algorithm is to perform a local mark-scan whenever a reference to a shared graph is deleted. That is, a mark-scan is initiated each time an object is suspected of belonging to a garbage cycle (i.e when its counter is decremented down to one). The mark phase decrements counters each time it visits an object belonging to the subgraph. At the end, all nodes with counters equal to zero are part of a dead cycle and may be safely reclaimed. Lins [7] improves the basic idea to perform the mark-scan lazily. Spurious objects are not scanned at once but instead they are queued in a special list. When the allocator fails to supply memory the corresponding list is scanned in order to reclaim potential garbage cycles. Unfortunately, mark-scan of subgraphs must be computed in critical sections. In other words, two different spaces cannot invoke cycle detection concurrently.

7 Conclusion

We have experimented with the SGP protocol on Transpive. The choice of Transpive allowed us to quickly implement the SGP protocol and to learn few lessons, although the distributed model of Transpive is quite different from SGP's. The original SGP model did not take into account the replication of objects. Consequently, we have adapted the SGP protocol into the replication model of Transpive. As a result, collection of out-going references —descriptors in the Transpive model— is more complex and slower than we expected. This increases the conservative aspect of the SGP and can be troublesome if memory is heavily in demand. A solution to decrease the delay for collecting out-going references is to decoupled local GC from SGP. Moreover, the fine grained sharing support of Transpive is definitely an uncooperative environment for a distributed GC. In particular, memory consumption is heavy since it requires a huge number of entries in the control data structures. As a consequence, it increases the overhead of SGP on application and the frequency of local GC. The performance results are encouraging but need to be improved , to minimize the overhead on applications. The buffering policy reduces dramatically the number of control messages. The resilience to message failures has been demonstrated. This result validates our design guideline. However, the fault-tolerance to space failures and duplicate messages remain to be investigated. Although, the SGP design relies on a very different distributed programming model, the prototype behaves correctly with respect to the safety property. It demonstrates that the SGP protocol is generic and adaptable. Therefore, it is a good candidate for a system service.

Acknowledgments

The design of the SGP protocol has been done in collaboration with **Olivier Gruber** of INRIA/RODIN. We wish to thank our colleagues of INRIA/ICSLA for their help and their availability to answering questions on Transpive, in particular **José Piquer**, and **Luis Mateu**. Many thanks also to **Daniel R. Edelson** and **Peter Dickman** for commenting on drafts of this paper.

References

1. BEVAN, D. I. Distributed garbage collection using reference counting. In *PARLE'87—Parallel Architectures and Languages Europe* (Eindhoven (the Netherlands), June 1987), no. 259 in Lecture Notes in Computer Science, Springer-Verlag, pp. 117–187.
2. DICKMAN, P. Optimising weighted reference counts for scalable fault-tolerant dis tributed object-support systems. (submitted to publication), 1992.
3. HUGHES, J. A distributed garbage collection algorithm. In *Functional Languages and Computer Architectures* (Nancy (France), Sept. 1985), J.-P. Jouannaud, Ed., no. 201 in Lecture Notes in Computer Science, Springer-Verlag, pp. 256–272.
4. J. CHAILLOUX, M. DEVIN, J. M. H. Le_lisp : A portable and efficient lisp system. In *Proc. 1984 ACM Symposium on Lisp and Functionnal Programming* (Aug. 1984), pp. 108–120.

5. Ladin, R., and Liskov, B. Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Sys.* (Yokohama (Japan), June 1992).
6. Lang, B., Queinnec, C., and Piquer, J. Garbage collecting the world. In *Proc. of the 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.* (Albuquerque, New Mexico (USA), Jan. 1992).
7. Lins, R. D. Cyclic reference counting with lazy mark-scan. Tech. Rep. TR-77, University of Kent, Computing Labotory Canterbury (England), Aug. 1991.
8. Lins, R. D., and Jones, R. Cyclic weighted reference counting. Tech. Rep. TR-95, University of Kent, Computing Labotory Canterbury (England, Mar. 1992.
9. Liskov, B., and Ladin, R. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing* (Vancouver (Canada), Aug. 1986), ACM, pp. 29–39.
10. Mancini, L., and Shrivastava, S. K. Fault-tolerant reference counting for garbage collection in distributed systems. *The Computer Jounal 34*, 6 (Dec. 1991), 503–513.
11. Piquer, J. M. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe* (Eindhoven (the Netherlands), June 1991), vol. I of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 150–165.
12. Piquer, J. M. *Parallélisme et distribution en Lisp.* PhD thesis, Ecole Polytechnique, Massy France, Jan. 1991.
13. Plainfossé, D., and Shapiro, M. Distributed garbage collection in the system is good. In *Proc. of the International Workshop on Object-Orientation in Operating Systems* (1991), pp. 94–99.
14. Shapiro, M., Gruber, O., and Plainfossé, D. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), Nov. 1990.
15. Watson, P., and Watson, I. An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe* (Eindhoven (the Netherlands), June 1987), no. 259 in Lecture Notes in Computer Science, Springer-Verlag.

This article was processed using the LaTeX macro package with LLNCS style