Extending .NET Remoting with Distributed Garbage Collection

Paulo Pereira
CCISEL
Rua Conselheiro Emídio Navarro N^o 1
1959-007 Lisboa, Portugal
palbp@cc.isel.ipl.pt

Luís Veiga and Paulo Ferreira
INESC-ID / Technical University of Lisbon
Rua Alves Redol N^o 9
1000-029 Lisboa, Portugal
luis.veiga@inesc-id.pt paulo.ferreira@inesc-id.pt

Distributed Systems Group, INESC-ID Lisboa, Portugal
http://www.gsd.inesc-id.pt/

Abstract. The memory management of distributed objects, when done manually, is an error-prone task. It leads to *memory leaks* and *dangling references* causing applications to fail. Avoiding such errors requires automatic memory management, named *distributed garbage collection* (DGC).
Current DGC solutions are either *not safe*, *not complete* or not portable to widely used platforms such as .NET. As a matter of fact, most solutions either run on specialized environments or require modifications of the underlying virtual machine (e.g. Rotor) hindering its immediate utilization.
This paper describes the architecture, implementation and performance measurements of a DGC algorithm that: i) is capable of reclaiming both *acyclic* and *cyclic garbage*, while ii) being portable in the sense that it does not require the underlying virtual machine to be modified.
The distributed garbage collector was implemented on top of two realizations of the Common Language Infrastructure (.NET virtual machine specification): Common Language Runtime (CLR) and Shared Source CLI (SSCLI), commonly known as Rotor. The implementation requires no modification of the environment, it makes use of the aspect-oriented functionalities provided, and the preliminary results are encouraging.

1 Introduction

There are several arguments that justify the existence of a system provided memory recycling service, named *garbage collector* (GC). These arguments, extensively presented in the context of local garbage collection [1], range from the description of the consequences of errors that result from manual management, namely unreclaimed memory (*memory leaks*) and premature reclamation (*dangling references*), to, according to Wilson [2], the classification of the existence of such a service as a fundamental requirement for accomplishing program modularity.

The strength of the previous arguments has been acknowledged with the inclusion of garbage collection services in platforms with wide industrial usage, such as .NET and Java.

In distributed object systems the same arguments apply, simply because these systems result from the extension of the programming model offered in non distributed ones. Considering that a local garbage collector exists, a natural extension would be to provide automatic reclamation of distributed memory. From the previous ideas, a question emerges: Why not *distributed garbage collection* (DGC)?

The CLI[1] includes a *distributed object system*, known as .NET Remoting. This distributed object system does not support automatic recycling of distributed memory. Alternatively, it offers a configurable object lifetime management service, based in renewable leases. This approach has several drawbacks since it places the problem of distributed memory recycling at the application programming level, leading to the previously pointed errors.

This document describes an extension of .NET Remoting with a distributed garbage collection service. The proposed solution is characterized as *safe* and *complete*, even in the presence of temporary failures (node or network). Permanent failures are not considered. Additionally, the presented solution is portable, in the sense that it does not require modification of the underlying virtual machine, and is valid in both considered realizations of the CLI (CLR[2] and Rotor[3]).

Although our main contribution is the previously characterized solution, another approach has also been used. This alternative approach differs from the former by the method used to collect the information required for DGC operation. The first resorts to techniques based in the existing AOP[4] support. The latter, by contrast, resorts to virtual machine source code modification and, for this reason, is only targeted at Rotor.

In both solutions the main design goal was the minimization of imposed application pause times due to DGC operation.

2 System Model

We consider systems that offer a distributed memory model based in the work done for the Orca language [3] and further refined for the Modula-3 [4] and Java [5] languages. These systems are commonly known as distributed object systems.

In the considered distributed memory model, the global address space (*global space*) is partitioned into disjoint address spaces (*space*). Each individual space is composed of objects and those that are globally accessible are named *remote objects*. For an object to be globally accessible it must have a *global identifier*, usually constructed using the hosting space identifier and the object local identifier.

Communication between spaces is modelled as remote object method calls, or simply *remote calls*. In each remote call two spaces are involved, the caller and called object hosting spaces. The former is named *client space* and the latest *server space*. These roles

[1] Common Language Infrastructure, .NET virtual machine specification.
[2] Commercially used CLI realization.
[3] Shared source CLI realization.
[4] Aspect-oriented Programming.

are established in a per remote method call basis (i.e. a given space can be both client and server space). For a remote call to be made, the client space must hold a reference to the remote object. This reference contains the remote object global identifier and is named *remote reference* or *inter-space reference.*

Furthermore, remote objects are *passive* (they do not have an internal thread of execution) and remain in the same hosting space for all their lifetime (not *mobile*). Applications are comprised of threads that invoke both local and remote objects.

For memory recycling purposes, objects are considered *live* if they are *reachable* (through graph transversal) from the running program *root set.* Otherwise, they are *garbage* (also called *dead*). A garbage collector function is to reclaim memory associated with garbage objects while preserving live ones. The root set is composed of all global variables and the local variables (residing at the activation stack and registers) of all existing threads of execution. Once an object is marked as garbage, it will always stay that way, meaning, unreachable objects cannot become reachable.

Finally, garbage collection algorithms are classified as *safe* if all live objects are preserved, and *complete* if all garbage objects are eventually recycled.

3 Distributed Garbage Collection

Individual spaces are assumed to have a local garbage collector (LGC) that is responsible for reclaiming dead objects. Each space also contains a DGC component that collaborates with the LGC to prevent reclamation of objects that are only reachable from remote spaces (through inter-space references). In practice, this collaboration takes place through local root set extension, meaning, the DGC simply prevents reclamation of an object by referencing it explicitly, leaving the existing LGC unmodified. With this approach the DGC component function is to *detect* distributed garbage and update the root set extension accordingly.

3.1 Algorithm

The presented DGC algorithm is classified as *hybrid*, since it combines a reference-listing [6, 4] algorithm, that is prevalent, for acyclic distributed garbage detection and, to ensure completeness, a centralized detector of distributed garbage cycles [7].

In the remainder of this section, we present the data structures maintained by the algorithm and the main conceptual aspects of its operation, with respect to acyclic and cyclic garbage detection.

Inter-space reference representation For distributed garbage detection purposes, inter-space references are represented as *stub-scion* pairs. The *scion* resides at the server space and represents an incoming inter-space reference. Each scion points to its corresponding locally hosted remote object. The *stub* resides at the client space and represents an outgoing inter-space reference. Each stub is associated with exactly one scion. A stub-scion pair represents all inter-space references that exist between a given client space and a particular referred remote object.

Stubs and scions are grouped in sets. All stubs in a set have their corresponding scions in the same server space. Conversely, all scions in a set have their corresponding stubs in the same referring space. Each space contains a table of stub sets and a table of scion sets. For simplicity, in the remainder of this document, the former will be called *stub table* and the later *scion table.* The number os sets in each table is determined by the number of referred and referencing spaces, respectively.

Local root set extension In order to prevent locally hosted remote objects from being reclaimed by the LGC when exclusively reachable through inter-space references, the considered local root set must be extended. This is the purpose of the stub and scion tables.

The stub table is a conservative estimative of all remote references held by its owning space (outgoing inter-space references). The scion table is a conservative estimative of all existing incoming inter-space references and explicitly references locally hosted remote objects as long as they are reachable from other spaces. Conservative estimations are used to ensure safety, that is, an object is considered live until proved otherwise.

The main function of both components (acyclic and cyclic) of the DGC algorithm is to update the previously described data structures, leaving actual object reclamation to the existing LGCs.

Acyclic Collector The creation of scions and stubs is performed incrementally, according to inter-space reference creation. This occurs whenever an application message bearing a remote object reference is exchanged between spaces. In this case, the sender space is said to *export* the reference, and the receiver is said to *import* it.

Since in distributed object systems communication is modelled as remote method calls, both intervening spaces (caller and callee) assume the roles of sender and receiver for each remote method call. The client space (caller) sends a message with the method parameters to the server space (callee). In response, the server space sends a message with the method result to the client.

Whenever a reference to a remote object located in a given space is exported, the corresponding scion must be created. Every time a reference to a remote object is imported into another space, the corresponding stub must be created. To accomplish this, all application messages must be (conceptually) scanned in search of remote references. This operation is critical since it imposes an application delay with the duration of the scan.

Besides the creation of remote references, another result of application execution is the abandonment of remote references, as they become useless. This leads to remote objects becoming unreachable. As a consequence, stub and scion tables must be updated to allow the reclamation of these objects.

DGC Protocol: When a LGC execution cycle ends, the stub table is updated according to the outgoing remote references that were dropped. The resulting table contains the stubs corresponding to the remote references that are still reachable from the local root set. The details of this process are explained in section 5.

Changes in the stub table trigger the creation of `NewSetStubs` messages. The number of created messages is determined by the number of stub sets that were altered (stubs removed) as a result of the LGC cycle execution. Each message contains information relative to the surviving stubs on that set. Once created, each of these messages is sent to the server space to which it is related (i.e. where the referred objects are hosted).

Upon reception of a `NewSetStubs` message, the space matches the carried information against the corresponding scion set. Scions for which a corresponding stub is not included in the `NewSetStubs` message, are removed from the set. Remote objects are recycled when all its scions are removed from the scion table. Naturally, objects are only recycled if they are also unreachable from the non extended local root set.

If the execution of the previously described tasks implied application suspension, the application performance penalties would be prohibitive. For this reason, the algorithm was designed to enable deferred execution of the tasks related to the DGC protocol, while maintaining its safety. Note that deferred execution may lead to concurrent execution.

In order to eliminate race conditions resulting from protocol tasks and application work being done concurrently, upon creation, scions are time-stamped using a per-space monotonic global counter, as prescribed in [8, 9]. Additionally, each space maintains a time-stamp vector, or vector clock [10], containing one entry for each referred space. Each entry has the highest known scion time-stamp value for the corresponding space. When a `NewSetStubs` message is sent to a given space, its corresponding entry in the vector clock is also sent.

Algorithm safety is maintained by associating the view of outgoing inter-space references with the time it was taken, ensuring its consistency. This prevents the receiving space from incorrectly eliminating scions whose corresponding stubs were not yet created when the `NewSetStubs` message was generated (e.g. a `NewSetStubs` message is generated when the reply of an ongoing remote method call has not yet been received).

The previously described protocol does not require global synchronization, since scion tables are updated incrementally, according to the reception of `NewSetStubs` messages. It is also fault tolerant, since message loss does not compromise safety, it only leads to delays in garbage detection.

Cyclic Collector Distributed cycle detection is performed by a centralized algorithm. Although in the presented solution the algorithm is performed in one of the participating spaces, selected through DGC configuration, for simplicity, the following description assumes it is performed in a dedicated space, called *distributed cycles detector* (DCD). Note that this no effect in algorithm safety and completeness.

The centralized algorithm operates on a global object graph view, constructed incrementally at the DCD from individual local object graph snapshots. These snapshots are taken at each participating space, without coordination, and sent to the DCD, where cycle detection is performed by using an adapted mark-and-sweep on the locally existent global object graph view. Although coordination between spaces is not required, the process of taking a local graph snapshot forces suspension of all application threads in that space.

Note that since the global graph view is constructed incrementally, it can in fact be a partial view, i.e. snapshots of one or more participating spaces have not yet been received and hence are not included in the global graph view. In this case, the DCD simply will not be able to detect distributed cycles that comprise objects belonging to those spaces. Nevertheless, cycles completely included in the partial view of the global graph, will be detected.

To ensure global graph view consistency, for cycles detection purposes, messages bearing local graph snapshots also include the space DGC related information, namely, time-stamp vector and scion and stub tables.

Snapshot Compression: The size of messages bearing local object graphs snapshots and DGC related information may lead to significant bandwidth usage, penalizing application performance. Additionally, the global graph views, constructed from individual snapshots, can occupy a large amount of memory space, limiting scalability (i.e. relative to the number of participating spaces).

These problems are solved through *snapshot compression*. This is done at each participating space ensuring that, from the point of view of the DCD operation, there is no loss of relevant information. This idea results from the observation that the internal details of each local graph are not relevant to distributed cycles detection. To this purpose, the only relevant information is the one regarding inter-space references. In particular, which outgoing remote references are reachable from the unextended local root set and which are reachable from incoming remote references (i.e. from the local root set extension). This information can be expressed as reachability relations between scions and stubs, and between local roots and stubs. Global graph views are created by connecting each stub with its corresponding scion.

This solution has the merit of making the sizes of the exchanged information and of the global graph views proportional to the number of existing remote references, as opposed to being proportional to local graph sizes, favoring scalability.

Compressed Snapshot based Mark-and-sweep: Cyclic garbage detection is performed trough tracing, starting at a conservative estimative of the *global root set*. This set is composed of individual unextended local root sets and, for safety, all the scions that match the following criteria: i) their corresponding stubs are not yet included in the global graph view; ii) their time-stamp has a value greater than the highest time-stamp (regarding the space where its associated object resides) known by the space holding the corresponding stub.

The tracing phase produces two groups of scions and stubs: marked or unmarked. Those that are unmarked may belong to a distributed garbage cycle. Note that both DGC collectors (acyclic and cyclic) are uncoordinated. After identifying the existing cycles, each cycle is broken through explicit scion deletion. The remaining objects in broken cycles are reclaimed by the acyclic collector.

In order to support explicit scion deletion, the DGC Protocol described in section 3.1 was extended with a new type of message, `ScionDelete`. This extension preserves the fault tolerance property of the protocol, since these messages are idempotent.

3.2 Architecture

The solution architecture is based in a clean separation between relevant event detection, required information gathering, and the DGC service implementation. The solution is composed of two layered modules: i) the lower module named *Instrumentation* and ii) the upper module named *DGC Service*. A layered organization was adopted in order to simplify the usage of the DGC algorithm in other realizations of the target distributed object system, since the only module that requires modification (if any) is the one named *Instrumentation*.

The function of the Instrumentation module is to gather the required information for the execution of the *DGC Service* module function, distributed garbage detection. In the remainder of this document, and for implementation description purposes, the stub and scion tables will be referred to as *External Reference Table* (ERT) and *Object Data Table* (ODT), respectively.

Before describing the implementation techniques used in each module, which is done in section 5, a functional decomposition is in order. The functionalities provided by the Instrumentation module are: i) Detection of remote reference import, for ERT update; ii) detection of remote reference export, for ODT update; iii) detection of LGC cycle termination, and, for ERT update, consequent gathering of information relative to unreachable outgoing remote references; iv) local object graph snapshot creation, for cycles detection.

The DGC Service module includes the following elements: i) an extension of the local root set, in order to include information about locally hosted objects that are remotely reachable (ODT); ii) a conservative estimative of locally held remote references (ERT) used to update remote ODTs; iii) a well-know communication endpoint for DGC protocol message exchange.

The previously enumerated functionalities are included in all participating spaces. Additionally, one of the participating spaces is responsible for performing cycle detection. This space includes a well known communication endpoint that receives snapshots from participating spaces and the component responsible for DCD operation.

4 .NET Support

This section describes the features of the target system that are fundamental to understand the proposed solution. For more details consider [11] and [12].

The target system exposes the distributed memory model described in section 2. In the exposed model, and using .NET Remoting terminology, types are categorized as *nonremotable* or *remotable*. Only instances of types in the last category are permitted to cross space[5] boundaries. Note that an object is said to cross space boundaries each time it is included in the actual parameters list, or return value, of a remote method call.

Remotable types are further refined into two subcategories: *marshal-by-value* and *marshal-by-reference*. The subcategory to which a given type belongs determines its

[5] In .NET terminology a space is also named *Application Domain*.

instances behavior in space boundary crossings. As their names suggest, marshal-by-value instances are passed by value, that is, a copy is used. On the other end, marshal-by-reference instances are passed by reference, meaning, the instance global identifier is used.

A type is categorized as marshal-by-reference if it derives (directly or indirectly) from `System.MarshalByRefObject`. Global identifiers are represented by instances of `System.Runtime.Remoting.ObjRef`, itself a marshal-by-value type.

For a type to be classified as marshal-by-value it must be annotated with the `Serializable` attribute. This requirement is justified by the process used for copy creation. This process, commonly known as *serialization*, produces a byte sequence containing the internal state of all objects included in a given graph.

4.1 Communication Infrastructure

Although communication is modelled as remote method calls, it is ultimately achieved through message exchanges between client and server spaces. When a method call is made, a message composed of the parameters is sent from the client space to the server space (these parameters include object and method identifiers). The method call result is obtained from the received return message.

The previous observation forms the basis of the .NET Remoting communication infrastructure architecture, depicted in figure 1.

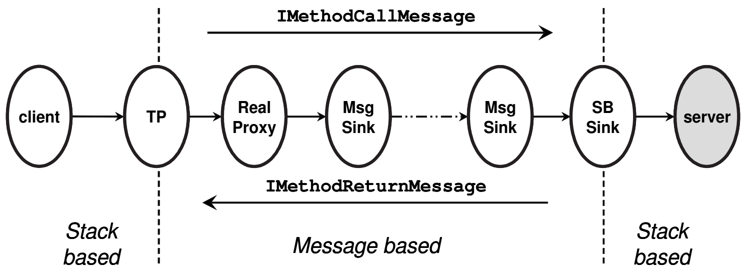

Fig. 1. Call chain composition

As shown, communication is supported by a configurable chain of message processing nodes[6], generally called *message sinks* (MsgSink). The chain endpoints, *transparent proxy* (TP) and *stack builder sink* (SB Sink), are responsible from converting between stack frames and messages. The remaining nodes, with the exception of *real proxy* (RealProxy), constitute the node set that supports the communication protocol. This node set is named *channel*.

The transparent proxy function is to represent the server object at the client space. In other words, the client object makes method calls on the transparent proxy, which

[6] Inspired in the name usually used to refer to computer network participants.

in turn, converts them into messages forwarded to the next node in the chain. The method call result is then produced based in the received return message.

The stack builder sink represents the client object at the server space. Received messages, passed along the node chain, are converted to local calls on the server object. The return value is converted to a message that is sent back through the chain.

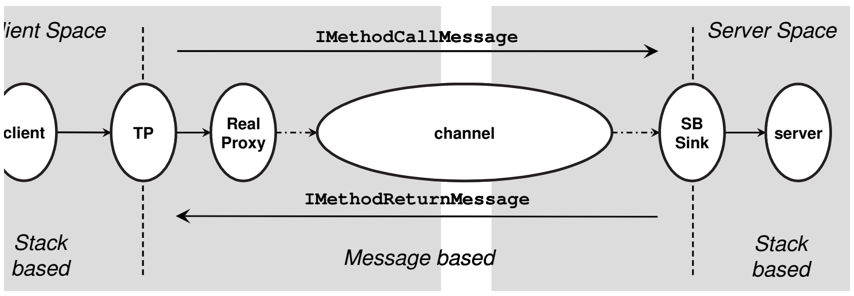

Fig. 2. Call chain distribution

As shown in figure 2, the chain is physically split across participating spaces. The half that resides at the server space is constituted upon object activation. The other half is created at the client space upon reception of the corresponding `ObjRef` instance. In order to enable client side chain creation, the received `ObjRef`, sent by the server space, contains all the required information (eg. location information, remote object identifier).

The previously described architecture offers several customization opportunities, in particular, the possibility to define new channels and new real proxy types. Considerations about their inadequacy for solving the problem at hand can be found in section 5.

4.2 Lifetime Management

Memory associated with remote objects is reclaimed using a lease based algorithm. The lease value determines if the associated remote object is alive. In each remote call for a given remote object, its lease is automatically renewed. When an object is considered dead (if its lease has expired) the corresponding memory is reclaimed.

The previously described approach is characterized as not safe since remote objects may be, erroneously, considered dead. For example, consider a client holding a remote reference and not making use of it for a time interval greater than the lease time. In this scenario, the client space will obtain an error when it tries to use the remote reference (i.e. by calling a method on the referred remote object).

To prevent such errors, the application has the opportunity to register sponsors that will decide about lease renewal. Each space contains a *lease manager* that tracks leases that are associated with remote objects. Periodically, the lease validity is checked and, if it has expired, a registered sponsor is queried about the lease renewal. Although interesting, this solution only shifted the problem of memory reclamation to the application programmer domain.

4.3 AOP Support

Aspect-oriented Programming is a paradigm proposed by Kiczales and colleagues in [13]. In this paper, the authors argue that Object-oriented Programming (OOP) failed to deliver the promised clean problem decomposition. Quoting them, "we have found many programming problems where OOP techniques are not sufficient to clearly capture all the important design decisions the program must implement. Instead, it seems that there are some programming problems that fit neither the OOP approach nor the procedural approach it replaces".

In fact, as noted, applications tend to be polluted with code snippets not directly related to the problem domain, but instead, related to system level concerns (e.g. security, transaction management, concurrency control). A similar observation, with respect to memory management, leaded to the pro garbage collection argument used by Wilson [2], in which the existence of such a service was considered a fundamental requirement for accomplishing program modularity.

In AOP, system properties are classified as *components* or *aspects*. Components contain the implementation of system properties directly related to the problem domain and that result from its functional decomposition. Aspects are the implementation of properties related to system level concerns and that are transversal to problem domains. The overall system is the resulting combination. The process of combining components and aspects is named *aspect weaving*.

Although there are several approaches to aspect weaving, usually performed via code instrumentation (e.g. source code instrumentation), only runtime aspect weaving will be considered in the current document.

The target system[7] provides support for runtime aspect weaving, performed through method call interception. The provided method call interception infrastructure, well described in [14], is based in the .NET Remoting communication infrastructure, described in section 4.1, and is known as *context architecture*.

Context Architecture In this architecture, spaces are subdivided in *contexts*. A context is the runtime environment that results from the addition of a particular set of aspects to the base system properties. This is performed through method call interception using the specific chain composition established at instance creation[8].

To selectively accomplish the afore mentioned call interception, marshal-by-ref types were further refined into two subcategories: *context-agile* and *context-bound* types. Only types in the later category are considered in the context architecture. A type is classified as context-bound if it derives (directly or indirectly) from `System.ContextBoundObject`, itself a `System.MarshalByRefObject` derivate.

[7] In both considered realizations, CLR and Rotor.
[8] Also named *activation*.

The association between component (context-bound type) and aspects (sets of nodes in the interception chain) is performed by annotating the component with a *context attribute*. This type of *custom attribute* [9] is responsible for specifying the required context for the annotated type instances.

Figure 3 depicts the general composition of the call interception chain. The chain is partitioned into four distinct regions, namely, *envoy sinks* (ESink), *client context sinks* (CCSink), *server context sinks* (SCSink) and *object sinks* (ObjSink). Each region is delimited by its corresponding system provided terminator sink (e.g. ETSink). The existence of these terminator sinks suggest that the chain is not implemented as a list of message sinks. In fact it is not, alternatively, each terminator sink is responsible for forwarding messages to the first node of the next existent region (set of nodes).

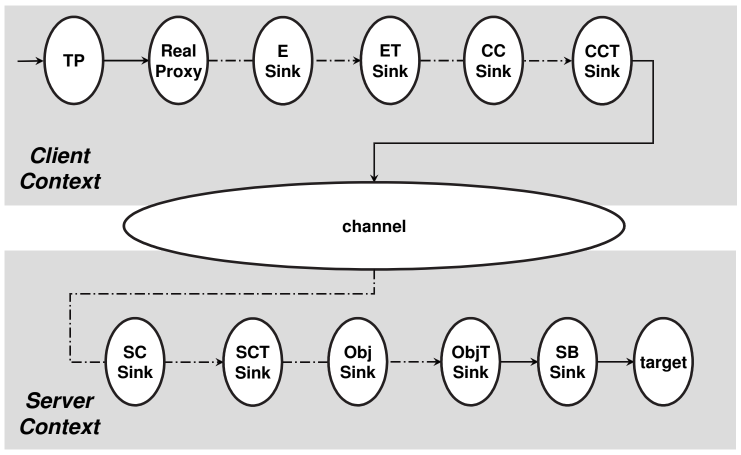

Fig. 3. Context architecture

To help understand the provided interception opportunities, consider the existence of two contexts (client and server) and the propagation in the chain of the message that corresponds to a method call. The sequence of events for the return message are reversed.

While still in the client context, two sets of sinks participate in call interception: envoy sinks and client context sinks. Although envoy sinks reside in the client context, they were specified by the server and are target object specific. They present an opportunity for collecting information regarding the client, for usage at the server context.

The next sinks that are given the opportunity to intercept the call are client context sinks. This set is specified by the context from which the call is originating.

The remaining two sets will participate in call interception upon arrival at the server context. Both sets are specified by the server and their names, server context sinks and object sinks, suggest their purpose. The first set is shared by all instances residing in

[9] .NET support for metadata extension.

the server context. The second is specific to each instance. As an example, consider the automatic renewal of a remote object lease, described in section 4.2. In this case, the .NET Remoting infrastructure uses an object sink in order to renew the lease each time the object receives a remote call.

5 Implementation

For the purpose of distributed garbage collection, marshal-by-reference types are classified as *dgc types* or *non dgc types*. Only instances of dgc types are subject to DGC. This classification provides a choice between using the proposed .NET Remoting extension, or the default lifetime management service. A type is classified as dgc type if it derives (directly or indirectly) from `System.Runtime.Remoting.DGC.RemoteObject`.

5.1 DGC Service module

Local root set extension is performed by using the default lifetime management service, in particular, by registering a local sponsor. This sponsor is shared by all dgc type instances hosted in the space. When queried about lease renewals, the sponsor responds according to the instance inter-space reachability status, maintained in the ODT.

The conservative estimative of locally held remote references (ERT), is updated upon LGC execution cycle termination. This event triggers the execution of a low priority background thread that discards stubs relative to unreachable outgoing remote references. This is performed by using *weak references*, a system provided mechanism for referring objects without preventing their reclamation. Each stub holds weak references for all locally held transparent proxies to its associated remote object. Note that, at the client space, and for a given remote object, more than one call chain may exist. The stub is discarded when all its weak references are invalid, meaning, all referred transparent proxies were collected.

The previously described techniques require additional event detection, namely: i) LGC execution cycle termination, to trigger the low priority background thread execution; ii) transparent proxy creation, to refer the created proxy from the corresponding stub, using a weak reference.

The remaining elements in this module are the required well-known communication endpoints. They are implemented as stateless remote objects that exist for the entire space lifetime.

5.2 Instrumentation module

As previously stated, the presented solution does not require modifications to the underlying virtual machine. Nevertheless, an alternative approach has been used, one that resorts to virtual machine source code modification. The differences between both approaches are restricted to the current module and are described in subsections 5.3 and 5.4. The only functionality that is identical in both approaches is the detection of LGC cycle execution termination. In order to achieve it, an unreachable singleton instance of type `System.Runtime.Remoting.DGC.LocalGcDetector` is used. This type redefines the `Finalize` method in order to generate the required event. Since the singleton instance is never reachable from the local root set, it is considered garbage. As a consequence, it is eventually collected and its `Finalize` method is called, generating the required event and creating a new unreachable instance.

5.3 Runtime Instrumentation

As described in section 4.1, the .NET communication infrastructure provides two extensibility points: channels and real proxies. Although it would be possible to detect remote reference import and export through the definition of a custom channel, that would perform the required message scanning, this solution would force the usage (by the application) of that particular communication channel. Alternatively, the DGC service is considered an *aspect*, and the provided context architecture is used. This approach does not impose a specific communication channel.

The base type for dgc types (`RemoteObject`) derives from `ContextBoundObject`. It is annotated with a context attribute that adds an envoy sink to the call interception chain. Additionally, the context attribute performs proxy creation detection, since it has the opportunity to inject a custom real proxy in the interception call chain.

Although the envoy sink can be used to scan all messages that traverse the call chain, this is not the proposed solution, due to the performance penalties that would result from intensive message scanning. Based in the observation that envoy sinks are part of the client side call chain, a conclusion can be drawn: envoy sinks must be an extension of the `ObjRef` instances that represent the remote object. In fact, they are! By detecting serialization and deserialization of the inserted envoy sink, we can detect remote reference export and import, respectively. Upon serialization, the envoy sink transports the DGC required information (e.g. scion identifier).

With the previously described solution, detection of remote reference boundary crossings only imposes performance penalties when application messages *effectively* contain remote references.

Finally, and for distributed garbage cycle detection, snapshot creation is performed by specific application request. To the effect, the application programmer contributes with the information of what transparent proxies are still reachable from the non extended local root set.

5.4 Source Code Instrumentation

This approach is still a work in progress. At the time of writing, the instrumentation required for acyclic garbage detection is fully implemented. The work in progress is related to cycles detection, in particular, to the extension of the local GC with the code necessary to obtain the local root set snapshot at each participating space.

In this approach, instead of using an extension (envoy sink) of the call interception chain as a way to extend `ObjRef` instances with the required information, we actually extended the `ObjRef` type with the required code. Additionally, in order to detect proxy creation, the method `SetOrCreateProxy` of type `RemotingServices` was also modified. This method is called each time the space receives an `ObjRef` instance and, as a consequence, the corresponding proxy must be created.

5.5 Preliminary results

Due to the deferred nature of the implemented DGC algorithm, the majority of the DGC related tasks are executed in a low priority background thread. The solution viability is not conditioned by the performance of these tasks, since they do not penalize application performance. Nevertheless, it is essential that, eventually, these tasks are given the opportunity to be executed. We assume that they will. Failure to comply leads to undetected garbage accumulation.

Note that undetected garbage accumulation is not actually a problem, since its growth (beyond configured limits) can trigger a priority boost of the background thread, forcing its detection and consequent reclamation. Naturally, this is has a negative impact (but necessary) on application performance.

The DGC related tasks that are considered critical are those that impose application pause times (i.e. no application related work is being done, regardless of the urgency). These pause times occur when DGC book-keeping is performed in application enrolled threads (i.e. synchronously). Our implementation main performance requirement was the minimization of these pause times, imposed by the following tasks: i) Remote reference usage, by calling a method of the referred remote object; ii) Remote reference import and export detection; iii) Local snapshot creation, for cycle detection.

For the time being, hence the subsection name, we have focused in measuring the performance penalties imposed by the first two, when using runtime instrumentation.

In the next subsections, the presented times are the average of 100 samples of the execution time of each sequence of actions to evaluate. Efforts have been made to ensure that no LGC and DGC collections occur while evaluating the costs of each sequence of actions.

Although these measurements are presented as absolute times, their single purpose is to underline the working temporal scale. Conclusions were only drawn from the observed variations between measured times with, and without DGC related operations.

Remote reference usage Here, the goal is to evaluate time overhead imposed by the usage of the extended call chain to make remote invocations. Note that an additional node exists in the call chain associated with dgc types instances. Since this additional node is a pass-through, meaning, it just forwards messages to the next node in the chain (it is not actually scanning passing messages), the expected additional cost is low.

In the following scenarios, two spaces are used: a client and a server space. The client performs a remote invocation on an object hosted at the server space. The elapsed time is measured at the client, hence it includes round-trip and remote method service times. Table 1 summarizes the results.

Scenario A This is an unrealistic worst case scenario. Both spaces are hosted in the same computer and the called method does not perform useful work (i.e. empty method body).

Scenario B Both spaces are hosted in the same computer and the called method does perform some work (simulated and $\approx 1ms$ long)

Scenario C Each space is hosted in a separate computer and the called method does not perform useful work.

	A	B	C
Non dgc type ($\times 100ns$)	2603	10214	66188
Dgc type ($\times 100ns$)	3104	10314	67191
Δ (%)	19.25	0.98	1.52

Table 1. Remote reference usage costs

With the previous scenarios we intend to show the effects of the inclusion of realistic factors (i.e. network latency or useful work execution) in the relative cost associated to the usage of the extended call chain. This relative cost is masked by the cost of each factor introduced. From the previously stated, we conclude that, as expected, in realistic scenarios there is no significant penalty for using the proposed DGC solution.

Remote reference import and export detection Here, the goal is to measure time overhead associated to the export and import of a remote reference to an instance of a dgc type, as opposed to remote reference export and import to an instance of a non dgc type. The measured time includes client and server data structures update times and remote reference propagation.

This measurements will be the cost associated with, conceptually, scasnning messages that contain remote references, and as a consequence, updating DGC data structures.

	A	B	C	
Non dgc type ($\times 100ns$)				
Dgc type ($\times 100ns$)				
Δ (%)				

Table 2. Remote reference usage costs

6 Related Work

This work focuses distributed garbage collection, applied to real-world platforms, without intrusive modifications that hinder portability and prevent wide deployment of the proposed solutions. Distributed garbage collection has been extensively described in the literature [15–19], comparing different algorithms, based on parameters such as asynchrony, message traffic, space and time overhead.

In these systems, distributed garbage, including distributed cycles, is frequent and has been characterized in [20, 21]. We use the term GC-solution to designate the set of components and algorithms involved in performing garbage collection, both local and distributed, and their actual implementation. Incomplete solutions are typically based on distributed reference-counting and reference-listing [22, 23]. Detection of distributed cycles has been addressed using i) object migration: explicit [24] and via indirection [28] (train algorithm), ii) trial deletion [30], iii) propagation of marks or time-stamps: global [31–33], within groups [35, 36], iv) distributed back-tracing [38, 39, 33], v) centralized detection: loosely-synchronized [40], and asynchronous [41], and vi) cycle detection algebra [42].

Nonetheless, most of the solutions found in the literature are developed towards very specific systems, namely research prototypes, where it is assumed the DGC developer has complete control over the runtime. When applied to a standard, or widely deployed runtime (as Java and .NET), these solutions frequently require significant modifications to the underlying runtime.

Thus, we briefly evaluate existing work on a different perspective, with portability in mind instead, in the sense that we have described. We present a qualitative overview of two main issues that may hinder the adoption of a complete GC-solution to any widely adopted runtime: i) *runtime intrusion*, and ii) *coupling* between different components of the GC-solution. Each of these aspects is decomposed in sub-aspects and, for each of them, we introduce a scale of approaches with *increasing degrees* of portability and/or flexibility. Some solutions may be mentioned at different degrees because of different techniques they employ.

6.1 Runtime Intrusion

Runtime intrusion is defined as the need to deviate from an existing runtime, in order to provide it with a specific garbage collection solution. These deviations may be caused by different GC components, and have different degrees. Naturally, the optimum degree is not requiring any intrusion at all, and this is the case when a specific solution is not explicitly mentioned.

Local GC The most inflexible technique, w.r.t. LGC, when adopting a GC-solution is to impose an *heterodox LGC* [28, 32], substantially different from those typically included in the runtime. A GC-solution may require the *extension of reachability encoding* of an existing LGC. This is the case of solutions that require the LGC to incorporate, in object headers, more bit-colors [35, 36] or additional marks, like time-stamps [31–33] or reachability-maps [40, 38]. An existing GC may also be subject to *extension of operation* that is less intrusive that the previous technique, either pre-pending or appending operations to the ones already performed by the existing LGC, such as generating stub sets [22, 23], or calculating backward references [39]. A solution may impose *direct instrumentation*, in that the existing LGC must be blocked [38, 36] or triggered at specific moments (e.g., when coordinating with other GC components), possibly for a partial collection over a fraction of the objects [36]. *Indirect instrumentation* consists in using indirect mechanisms to detect when a local garbage collection has taken place (e.g., using `finalizer` methods on a dummy object). This is portable and is used in our solution.

Acyclic DGC The most inflexible technique to implement a distributed garbage collector is to *modify the communication protocol*, or impose the use of a specific one provided by a non-standard system [24, 31, 30, 22, 35, 28, 32, 36, 33] (e.g., Thor [40, 38]). Alternatively, intrusion may be confined to *modifying remoting mechanisms* and its code [33, 41, 42]. If it is possible and allowed, DGC may be implemented resorting to *interception of library loading* performed by the dynamic linker, either by extending or overriding the functionality of components regarding communication and remote method invocation, without modifying code [39]. Portable techniques include *extended communication mechanism*, resorting to extensions allowed by the runtime, such as custom sockets.

Finally, even non-intrusive extensions may be independent of the communication protocol and restricted to *extended remoting mechanisms*, such as sink chain extensions (as this solution provides).

Cycle Detection Some solutions, depending on the adopted algorithm(s) may require additional *direct intrusion* in the runtime, for the purpose of cycle detection, without possibility of delaying the disruptive operations. Examples include blocking the mutator, while performing bit-color propagation in some situations [36], and applying barriers to inter-space invocations when back-tracing information is being calculated [38]. In general, most solutions also require information of the *local root-set* of each space, in order to differentiate objects targeted by local references, or just by inter-space references. This may be achieved by modifying the LGC, or indirectly via hints provided by the programmer. This is required because existing runtimes, neither inform about different levels of reachability, nor provide reflection services that offer information about stack variables.

6.2 Coupling of GC components

Coupling is defined as the degree of interdependency among different GC components (namely LGC, acyclic DGC, and cycle detection), in the sense that the adoption of one approach for one component, will mandate the adoption of the same or related approach to one, or both the others. In essence, this assesses how monolithic a GC approach is, or how it may be flexibly combined with others. This will determine the difficulty of deploying the algorithm if it is not possible to modify the runtime, namely its LGC. Furthermore, this may hinder application performance and/or delay garbage reclamation since garbage of the three kinds is not created at similar rates, and thus should be addressed with specialized approaches.

One-size-fits-all The most inflexible solutions are those that mandate the use of the *same algorithm*, a specific one for all three GC components [28, 31–33], i.e., the use of a acyclic DGC algorithm, or cycle detector, effectively mandates the use the same algorithm for LGC purposes. Naturally, this seriously undermines the adoption of these algorithms to an existing runtime, if one of the components cannot by modified or extended.

LGC and Acyclic DGC Some solutions demand strong integration of the components that perform LGC and acyclic DGC. They may require the LGC to *propagate information*, through the object graph, received by the acyclic DGC component, namely marks [35, 36] and time-stamps [31–33], or otherwise provide inter-space *reachability information* of objects to the DGC [40].

Acyclic DGC and Cycle Detection There are solutions that, while avoiding intrusive modifications to the LGC of an existing runtime, use the *same algorithm* for acyclic and cyclic DGC [31, 40, 28, 32, 33]. This is not as prejudicial as with the case of LGC, but it may prevent the use of a cycle detector if it imposes changes to an existing acyclic DGC algorithm (e.g., reference-listing) integrated in the runtime . Furthermore, using the same algorithm may delay the identification of acyclic garbage that should be performed more frequently (e.g., [31, 28, 32]). At an intermediate level, the DGC must be able to *cooperate* with the cycle detector, e.g., performing simulated deletions [30]. Other solutions use *specialized cycle detectors* that do not interfere with normal, more frequent, acyclic DGC operation, namely [24, 35, 38, 39, 36, 41, 42].

LGC and Specialized Cycle Detection The coupling between LGC and cycle detection, in the context of solutions that use the same algorithm for acyclic and cyclic DGC was already addressed in the second headed paragraph. With respect to solutions with specialized cycle detectors, those based on migration techniques must be able to *detach objects* from the local graph and create the appropriate inter-space references to preserve their reachability [24, 28]. Trial deletion for cycle detection requires the LGC to provide *tentative reachability information* about the outcome of simulated deletions [30]. Cycle detectors that need to be informed about local root-sets, do not necessarily preclude the use of the runtime built-in LGC [35, 38, 39, 36, 41, 42].

In this section, we have analyzed the virtues and shortcomings of a number of the most relevant GC-solutions found in the literature, wit respect to runtime intrusion and coupling among their components.

There is no optimal solution, i.e., one that does not require any modification nor extension of the runtime. Nonetheless, we believe that those with increased degrees of portability (i.e. runtime intrusion) and flexibility (i.e. component decoupling), as the one described in this paper, can be deployed realistically. Besides our proposal, the other work we found where the concerns about portability are paramount, despite some runtime intrusion, is the implementation of distributed back-tracing cycle detector for CORBA objects [39].

7 Conclusions

References

1. Jones, R., Lins, R.: Garbage Collection, Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons (1996)
2. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proc. Int. Workshop on Memory Management. Number 637, Saint-Malo (France), Springer-Verlag (1992)
3. Bal, H.E., Tanenbaum, A.S., Kaashoek, M.F.: Orca: A language for distributed programming. SIGPLAN Notices **25**(5) (1990) 17–24
4. Birrell, A., Nelson, G., Owicki, S., Wobber, E.: Network objects. Software–Practice and Experience **25**(S4) (1995) 87–130
5. Wollrath, A., Riggs, R., Waldo, J.: A distributed object model for the Java system. In: 2nd Conference on Object-Oriented Technologies & Systems (COOTS), USENIX Association (1996) 219–232
6. Shapiro, M., Dickman, P., Plainfoss'e, D.: Robust distributed references and acyclic garbage collection. In: In Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing, Vancouver (Canada) (1992) 135–146
7. Veiga, L., Ferreira, P.: Complete distributed garbage collection: an experience with rotor. In: IEE Proceedings - Software. Volume 150. (2003) 283–290
8. Hughes, J.: A distributed garbage collection algorithm. In Jouannaud, J.P., ed.: Functional Languages and Computer Architectures. Volume 201 of Lecture Notes in Computer Science. Springer-Verlag, Nancy (France) (1985) 256–272
9. Shapiro, M., Gruber, O., Plainfossé, D.: A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320 (1990)
10. Mattern, F.: Virtual time and global states of distributed systems. (In: Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel Computing)
11. Rammer, I.: Advanced .NET Remoting. Apress (2002)
12. McLean, S., Naftel, J., Williams, K.: Microsoft .NET Remoting. Microsoft Press (2003)
13. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
14. Box, D., Sells, C.: Essential .NET, Volume 1: The Common Language Runtime. Addison-Wesley (2003)
15. Plainfossé, D., Shapiro, M.: A survey of distributed garbage collection techniques. In: Proc. Int. Workshop on Memory Management, Kinross Scotland (UK) (1995)
16. Jones, R., Lins, R.: Garbage Collection, Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester (GB) (1996) ISBN 0-471-94148-4.
17. Abdullahi, S.E., Ringwood, G.A.: Garbage collecting the internet: a survey of distributed garbage collection. ACM Computing Surveys (CSUR) **30**(3) (1998) 330–373
18. Shapiro, M., Fessant, F.L., Ferreira, P.: Recent advances in distributed garbage collection. Lecture Notes in Computer Science **1752** (2000) 104
19. Ferreira, P., Veiga, L.: Garbage collection curriculum. Msdn academic alliance curriculum repository, object id 6812, Microsoft (2005)
20. Wilson, P.: Distributed garbage collection general discussion for faq. GCList Mailing List (gclist@iecc.com) (1996)
21. Richer, N., Shapiro, M.: The memory behavior of the WWW, or the WWW considered as a persistent store. In: POS 2000. (2000) 161–176
22. Shapiro, M., Dickman, P., Plainfossé, D.: Robust, dist. references and acyclic garbage collection. In: Symposium on Principles of Dist. Computing, Vancouver, Canada (1992)
23. Birrell, A., Nelson, G., Owicki, S., Wobber, E.: Network objects. In: SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (1993) 217–230
24. 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. (1977)

25. Shapiro, M., Gruber, O., Plainfossé, D.: A garbage detection protocol for a realistic dist. object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt (1990) Superseded by [46].
26. Gupta, A., Fuchs, W.K.: Garbage collection in a distributed object-oriented system. IEEE Transactions on Knowledge and Data Engineering **5**(2) (1993) 257–265
27. Maheshwari, U., Liskov, B.: Collecting cyclic dist. garbage by controlled migration. In: Proc. of PODC'95 Principles of Dist. Computing. (1995) Later appeared in Dist. Computing, Springer Verlag, 1996.
28. Hudson, R., Morrison, R., Moss, J.E.B., Munro, D.: Garbage collecting the world: One car at time. In: Conf. on Object-Oriented Programming Systems, Languages, and Applications, Atlanta (U.S.A.) (1997)
29. Lowry, M.C., Munro, D.S.: Safe and complete distributed garbage collection with the train algorithm. In: 9th International Conference on Parallel and Distributed Systems (ICPADS 2002), 17-20 December 2002, Taiwan, ROC. (2002) 651–658
30. Vestal, S.C.: Garbage collection: an exercise in distributed, fault-tolerant programming. PhD thesis, Seattle, WA, USA (1987)
31. Hughes, J.: A distributed garbage collection algorithm. In Jouannaud, J.P., ed.: Functional Languages and Computer Architectures. Number 201 in Lecture Notes in Computer Science, Nancy (France), Springer-Verlag (1985) 256–272
32. Louboutin, S.R., Cahill, V.: Comprehensive dist. garbage collection by tracking causal dependencies of relevant mutator events. In: Proc. of ICDCS'97 Int'l Conf. on Dist. Computing Systems, IEEE Press (1997)
33. Fessant, F.L.: Detecting distributed cycles of garbage in large-scale systems. In: Conference on Principles of Distributed Computing(PODC). (2001)
34. Philippsen, M.: Cooperating distributed garbage collectors for clusters and beyond. Concurrency: Practice and Experience **12**(7) (2000) 595–610
35. Lang, B., Quenniac, C., Piquer, J.: Garbage collecting the world. In: Conf. Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages. ACM SIGPLAN Notices, ACM Press (1992) 39–50
36. Rodrigues, H., Jones, R.: Cyclic distributed garbage collection with group merger. Lecture Notes in Computer Science **1445** (1998) 260
37. Fuchs, M.: Garbage collection on an open network. In Baker, H., ed.: Proc. of Int'l W'shop on Memory Management. Volume 986 of Lecture Notes in Computer Science., Concurrent Engineering Research Center, West Virginia University, Morgantown, WV, Springer-Verlag (1995)
38. Maheshwari, U., Liskov, B.: Collecting cyclic dist. garbage by back tracing. In: Proc. of PODC'97 Principles of Dist. Computing. (1997)
39. Rodriguez-Rivera, G., Russo, V.: Cyclic distributed garbage collection without global synchronization in corba. In: OOPSLA'97 GC & MM Workshop. (1997)
40. Liskov, B., 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), ACM (1986) 29–39
41. Veiga, L., Ferreira, P.: Complete distributed garbage collection, an experience with rotor. IEE Research Journals - Software **150(5)** (2003)
42. Veiga, L., Ferreira, P.: Asynchronous complete distributed garbage collection. In: 19th IEEE International Parallel and Distributed Processing Symposium, Denver, CO, USA (2005)
43. Ferreira, P., Shapiro, M.: (1993) IEEE Computer Society Press.
44. Shapiro, M., Plainfossé, D., Gruber, O.: A garbage detection protocol for a realistic distributed object-support system. Technical report, INRIA (1990) INRIA 1320.
45. Stutz, D.: The microsoft shared source cli implementation. MSDN Library Article, Microsoft Corporation (2002)
46. Shapiro, M.: A fault-tolerant, scalable, low-overhead dist. garbage collection protocol. In: Proc. of the Tenth Symposium on Reliable Dist. Systems, Pisa (1991)