Towards Distributed Garbage Collection in Distributed Real-Time Java

P. Basanta-Val, and M. García-Valls
Dpto. de Ingeniería de Telemática
Universidad Carlos III de Madrid
Avda. Universidad nº 30, Leganés- Madrid (Spain) 28911
{pbasanta, mvalls}@it.uc3m.es

Abstract—Java’s RMI (Remote Method Invocation) offers many facilities useful for a distributed application programmer. One of them is distributed garbage collection (DGC) that removes unreachable remote objects. Unfortunately, from the point of view of distributed real-time Java applications, DGC is underspecified (i.e., it may introduce unwanted interference on real-time remote invocations). Some researchers proposed there should be a real-time version for this mechanism in distributed real-time Java. This paper proposes a mechanism to turn DGC off and a real-time distributed garbage collector (RT-DGC) based on mechanisms currently available in RMI (such as leasing, and reference and unreferenced methods).

Terms.- RTSJ, DRTSJ, real-time Java, distributed garbage collection,

I. INTRODUCTION

Current complexity in developing real-time systems is increasing dramatically [1][2]. On one hand, ancient monolithic and small size systems are being interconnected to offer new applications that have end-to-end trans-node predictability requirements [3]. On the other, the average number lines of code of a typical real-time system is increasing too because new requirements from applications appear. To deal with this problem, real-time practitioners may opt for high-level programming languages that reduce the development and maintenance cost of their applications. One of these languages is real-time Java [4]. Currently, real-time Java comprises three core technologies: RTSJ [5] (The Real-time Specification for Java), DRTSJ [6] (The Distributed Real-time Specification for Java) and SCJS [7] (Safety Critical Java Specification). The first targets to single virtual machines and deals with problems related to centralized resource management; the second offers mechanisms to interconnect different virtual machines in a predictable way; and the third is intended for safety critical systems.

So far, they evolved at a different rate [8]. RTSJ has many implementations ready to be used. SCJS is in draft version. DRTSJ [9, 10] is moving slower and the community is lacking both a reference implementation and its corresponding implementation.

The list of problems that DRTSJ has to face with is extensive ([11]). It includes issues that range from low-level support to techniques (like end-to-end transactions, a predictable memory model) to the redefinition of existing services and new ones (see [12, 13]). Among the list of suggested services the most addressed is the remote invocation. Many researchers ([14, 15][16] [17][18]) proposed extensions to the API to interconnect different real-time networks, manage end-to-end priorities, and offer end-to-end predictable memory management models. New services (DEH [19]—distributed event handler—and SSS—synchronous scheduling service [20]) were proposed to manage distributed events and deal with new network models. Lastly other services, such as naming and the distributed garbage collector, defined in RMI [21] (the Java’s Remote Method Invocation mechanism which supports DRTSJ) were deprived from real-time performance. Many practitioners, especially those focused on high-integrity systems, have constrained its use to an initialization phase.

This paper is focused on the distributed garbage collection for distributed real-time Java issue. Its goals are to analyze the trade-off among removing distributed garbage collection and defining a real-time distributed garbage collector for distributed real-time Java. DRTSJ may profit from the results defined in this article to base on them its distributed garbage collector. The techniques are also interesting for other real-time RMI approaches.

The rest of the paper is organized as follows. Section II introduces the real-time Java framework used to support the model. Section III describes the support currently offered by Sun’s DGC. Section IV introduces support to leave out the current DGC algorithm and also mechanisms to introduce support for RT-DGC. These new algorithms are evaluated in Section V. Section VI is the related work section which connects the proposed contribution to other distributed real-time Java initiatives. Finally, the paper ends with conclusions and future work (Section VII).

II. BACKGROUND ON DISTRIBUTED REAL-TIME JAVA

In order to develop the rest of the model, this section refers to previous work. It introduces DREQUIEMI [22][22]: a framework for distributed real-time Java application development. It is especially focused on its real-time distributed garbage collection service.

DREQUIEMI ([22, 23][24]) has five layers: resources, infrastructure, distribution middleware, common services, and a component-based architecture with modules. Resources include memory, CPU and network. They are controlled via an extended RTSJ-VM (with network support) to offer common services which are used by application modules. Middleware configuration may be

{pbasanta, mvalls}@it.uc3m.es
carried out at three levels: centralized, distributed and locally on each service.

It supports four basic services: stub/skeleton (or just real-time remote invocation); DGC (distributed garbage collector); naming (which allows discovering remote objects); and a distributed event service with two flavors: i) asynchronous (like DRSJ’s DEH) and ii) synchronous ([12, 13]).

DGC is in charge of removing remote objects when they are not in use. It complements the garbage collector hosted in each node avoiding memory leaks due to remote objects that cannot be referenced from any machine. DREQUIEMI readapted Birrell’s algorithm [25, 26] currently included in RMI to be predictable.

An excellent characterization of the original algorithm is available on [25, 26] and [27]. The latter reference also describes race conditions and optimizations on the original Birrell’s algorithm.

```java
package es.uc3m.it.drequiem.rtrmi.server.dgc;
import java.rmi.server.dgc.*;
public interface RTDGCInterface extends java.rmi.Remote{}
public interface Server extends RealtimeUnicastRemoteObject{
    public Object remoteObjectLeak(){
        Server s= new Server();
        server.remoteObjectLeak();
        return s;
    }
}
```

Listing 1. DREQUIEMI’s DGC service

Listing 1 contains the basic remote interface of the RTDGC service described for DREQUIEMI. It contains two methods, one to create a remote reference from a remote node and another to unreference it. The internals of the middleware use the interface to add/remove remote references to a certain remote object.

III. MOTIVATION FOR A DGC SERVICE AND SUN’S IMPLEMENTATION SKETCHED

Not all distributed real-time Java applications require from a distributed garbage collector. However, other type of systems could incorporate them to offer additional flexibility to applications. One of these systems is DRTSJ who could include garbage collection to remove unused objects in its API.

A. Type of Memory Leaks Avoided by DGC

Listing 2 shows an example of a potential remote object memory leak. The code corresponds to a simple client and server. The server offers a LeakExample interface which is used by the client to invoke a remote method (Client: 06) at the server. The server’s side creates a remote server (Server: 07), which is accessible in the registry later (Server: 08). The client’s side looks up the server and invokes its unique remote method.

The example showed in Listing 2 creates a remote object (Server: 03) when the client invokes the remote object (from Client: 06). This object is automatically exported to the internals of the middleware, and can be invoked from its stub. The underlying middleware allocates the remote object (Figure 1 STEP1) in the internal table which contains all allocated remote objects, and then it exports the remote object to the table of exported remote objects (Figure 1-2).

```java
package es.uc3m.it.drequiem.rtrmi.server.dgc;
import java.rmi.server.dgc.*;
public interface RTDGCInterface extends java.rmi.Remote{}
public class Server extends RealtimeUnicastRemoteObject{
    public Object remoteObjectLeak(){
        Server s= new Server();
        server.remoteObjectLeak();
        return s;
    }
}
```

Listing 1. A remote method (remoteObjectLeak) that allocates a remote object at the server

Notice that it does not need to be accessible from the registry. If the object is returned as a result of the remote invocation (i.e., using Server 04: return s1) the client may invoke the client without further problems.

Notice that the code provided requires a distributed garbage collector to execute. If it is not supplied, then there is a memory leak each time the program invokes...
remoteObjectLeak because the objects are not removed from the internal registry table. Figure 1 shows the memory leak that corresponds to the example shown in Listing 2.

B. DGC and Naming Service Connection

There is a not trivial connection between the naming and DGC services: the naming service requires the DGC service to work properly. Figure 3 shows the relationship using the example described in Listing 2. The example binds a new remote reference to a logic name in the registry (Figure 3-STEP1). This name is used in the client to invoke the remote object later. The object is retrieved using lookup (Figure 3-STEP2).

To avoid the reference remote objects (Server) disappear while in the registry, the registry has to maintain a remote reference to the bound remote object. The remote reference stored in the registry lasts until another node removes it from the registry.

The example provided in Listing 2 does not provide us with this example. However, the client could destroy the remote reference invoking registry.unbind(server) in Client:07, just before exiting the main.

In total, the example has two remote references: one in the registry and another in the client. The reference allocated in the remote client avoids the server to be destroyed before the client ends.

Finally, notice that renouncing to the DGC requires additional changes in remote object programming and the underlying infrastructure. Remote object destruction should be explicitly be done by changing its remote object interface and the implementation in the registry.

Nevertheless, the changes are not as dramatic as turning off the centralized garbage collector because there are fewer remote objects in an application than plain Java objects.

C. Sun’s DGC algorithm sketched

RMI’s garbage collector ([21, 27]) is based on two mechanisms: leasing and reference lists. Reference lists are similar to reference counters; each remote object contains the list of remote references that point to the remote reference.

Birrell’s algorithm forces that each node that receives a reference to the remote object have a reference to node have to communicate it to the reference’s owner. Birrell proposed two primitives: dirty_call, clean_call to communicate the owner that the machine has a reference to the object and they object is no longer available. In Listing 2, before the client invokes remoteObjectLeak method, two dirty_call messages are used: i) one from the registry to notify that the reference is hosted there and ii) another in the client to notify that there is a new reference in the client node. Figure 4 shows the state of DGC algorithm at that time.

The transference of remote references has a special treatment at the node that receives them; it is called reference_copy treatment. If the reference is in the local list of references, then it is not necessary that the server calls the server with a dirty_call. However, the client has to count the number of local reference objects that point to the same remote object. Readers interested in the formal model of this mechanism are referred to [25].

RMI’s DGC considers the possibility of having failures in nodes. References have to be renewed periodically by default every 10 minutes. From the perspective of the middleware that means a periodic renewal transaction per each remote reference. In the default implementation, the middleware renews the reference each 5 minutes. Thus, Figure 5 shows the two new periodic processes allocated before invoking the “Leaky” remote reference. In the example, it is characterized as a periodic task.

DREQUIEMI’s implementation uses reference and unreference for remote communications between services. DREQUIEMI’s reference is similar to a dirty_call and unreferenced to a reference_copy. Leasing renewals are carried out with additional dirty_calls which substitute previous calls.

From the point of view of real-time systems, there are additional issues that should be characterized. The DGC algorithm does prescribe a characterization of the service useful in real-time applications. The additional messages required to transfer information about nodes do not define a worst-case execution scenario or scheduling parameters like priorities. The leasing mechanism of each node, which
renews references, is another source on indeterminism because its periodicity and interference is unbounded a priori.

IV. STRATEGIES FOR A RT-DGC SERVICE

This section studies two implementation approaches to the distributed garbage collector. It is described an interface to destroy remote object when a DGC is not available in the system. It also describes a predictable version for Sun’s DGC, namely RT-DGC.

A. Turning off DGC

The first approach, already mentioned in the paper, is to turn off the garbage collector. The obvious advantage is in performance terms. As shown in this section, turning off the garbage collector requires changes in the middleware. The creation of the remote object should automatically make it accessible from a remote node (i.e., via subscription on the table of exported objects). The implementation should not modify this mechanism to be backward compatible with RMI.

However, since DGC is turned off, the application needs a destruction mechanism that substitutes the remote object destruction carried out by DGC. Two mechanisms (unexportation of local objects and unreferenced of remote objects) were identified.

Unexporting local objects

RMI already offers a method to remove local objects from its internals. Applications may use it in exceptional situations to remove remote objects that may be referenced from other remote nodes. Nodes that host remote objects may use it to eliminate local remote objects.

Figure 6 shows how the server could use the API to unexport a local remote object. After invoking unexport, the DGC table removes the two references that point from the internals of the middleware to the remote object instance, being candidate for recycling. The remote object is going to be recycled by the local garbage collector when it decides to do so. The DGC algorithm allows its destruction by removing the references it holds to the remote object.

This basic mechanism could be insufficient and generate memory leaks because it requires a reference to the remote object. So that, this basic mechanism should be complemented with a method that lists all exported remote objects:

```
00: UnicastRemoteObject
01: static boolean
02: unexportObject
03: (Remote r, boolean force)
04: throws NoSuchObjectException;
```

B. Defining a Simple RT-DGC Service

Defining a real-time version algorithm for the described algorithm requires deciding when to call to reference and unreferenced methods in a way that no race conditions appear. It also requires the parameterization of leasing activities as real-time tasks so that they are scheduled with other tasks.

Reference method

The proposed RT-DGC invokes reference each time the remote reference leaves the local node subscribing the remote node to the list of nodes that have a reference to the remote object. The reference method is invoked with the reference of the client that allocates the object. In Listing 2, it happens twice before executing remoteObjectLeak. The first time occurs when the reference is allocated in the naming service (server→registry) and the second when it moves from the naming service to the client (registry→client).

```
00: UnicastRemoteObject
01: static Vector
02: listExportedRemoteObjects();
```
Figure 8. Reference to remote method abandoning its host node

Figure 8 shows the first process, when the client binds the remote object to the naming service. Before letting the reference go out from the local node, it invokes the DGC server of the remote reference’s owner. In the Figure 8 example, the owner of the Server remote is server. So that, server’s DGC is notified before continuing (Figure 9 STEP1). As a result, the DGC service allocates a reference in DGC_table.

The case of a transferring a reference to a non-local remote object occurs in Listing 2 after looking up the reference in the service. In this case, the naming service should notify the hosting node about a new reference in a remote node which is going to be transferred to a node called client. In Figure 9, it happens before transmitting the result of the “Leaky” look-up operation (STEP2) to the client. Before it transmits it back, it communicates the new reference to server (STEP3).

Figure 9. Reference to remote method abandoning a non-host node

Unreference method

Each reference method has its unreferenced counterpart that occurs in the remote node that is going to be released. The idea is that each node may hold a variable number of references to a remote reference. Once the last one has disappeared, it notifies the server using the reference method, which renews a reference from a node. If a reference is not updated periodically, then the naming service may remove it from the remote reference task set. From the point of view of the middleware, each remote reference is characterized as a periodic task (T, D, C) with a period (T), a deadline (D) and a cost (C). Since the renewal happens every T/2, the previous model is transformed into a periodic model with a variable release time that varies from [0,T/2] and a deadline which should not exceed T/2 from its release time (Figure 11).

Figure 10. Destruction of a local stub in a client and the corresponding notification to the server

The first line destroys the local reference to the stub that points to the remote object. The second forces the destruction of the object which notifies (Figure 9-STEP 1 and STEP 2) a remote communication with the node that hosts the remote reference.

Leasing model

To avoid machine failures each node notifies periodically to the owner of a reference that it has that reference. This is done using the reference method, which renews a reference from a node. If a reference is not updated periodically, then the naming service may remove it from the remote reference task set. From the point of view of the middleware, each remote reference is characterized as a periodic task (T, D, C) with a period (T), a deadline (D) and a cost (C). Since the renewal happens every T/2, the previous model is transformed into a periodic model with a variable release time that varies from [0,T/2] and a deadline which should not exceed T/2 from its release time (Figure 11).

Figure 11. Leasing process characterization for a local stub

Decoupling the DGC cost from the remote invocation process

Notice that reference and unreference processes are added to the local cost of a remote communication. From the point of the middleware, this could deprive the application from using this mechanism in high-efficient applications. For instance, the lookup mechanism could last the double because the remote communication described in the figure requires an extra communication. Furthermore, complex lookups with a huge number of communications may have serious communication overhead that translates into a linear relationship among
number of remote objects returned and its application response-time.

To decouple the DGC cost in reference operations, a first solution is to introduce additional elements on the DGC algorithm that decouple communications. For instance, in Figure 9 STEP2 could be delayed until the renewal of the local reference to the remote object. Figure 12 shows how the DGC communication is decoupled from the lookup statement to the background leasing mechanism.

![Figure 12. Deferring the communication cost to the registry](image)

V. EMPirical RESULTS

This section compares the different techniques described for supporting RT-DGC. All empirical results refer to the DREQUIEMI framework for distributed real-time Java applications. All nodes run DREQUIEMI on a 796Mhz processors, which are interconnected via a 100 Mbps Switched-Ethernet network.

The first experiment carried out compares synchronous RT-DGC to the deferred RT-DGC algorithm both described in Section IV. The results of the experiment are shown in Figure 13. It analyses three cases: (i) a real-time communication that does requires interaction with the garbage collector, (ii) the same remote invocation that returns a remote object; and a (iii) remote invocation that requires an interaction with the lookup service to search in which remote node is allocated the remote object. The results show the ability of the deferred algorithm to exclude the overhead of the DGC algorithm from remote invocations.

The second experiment refers to the overhead introduced by each remote reference. Each reference introduces a leasing task with blocking in the local node and consumes resources in the remote reference holder, which notifies the owner about a new reference, and the owner, who has to process the renewal. The client experience a block (U_bloq_Serv) and consumes a certain resources periodically which are modeled as a percentage in the utilization bound in the client and the remote object host (U_con(owner)). Figure 14 summarizes the main empirical performance results. It shows the evolution of the three types of cost with the leasing time; the shorter the leasing time is, the higher the amount of CPU required per period. Notice that the following inequations are true: 

\[ U_{\text{bloq}} \geq U_{\text{con}} \]

![Figure 13. Blocking time introduced in remote communications (796Mhz-100Mbps)](image)

VI. RELATED WORK

The related work refers to distributed real-time technology and its related interpretation of RT-DGC. Approaches like RTZen [28] and other CORBA’s attempts towards having a distributed garbage collector are set aside.

Regarding DRTSJ, it did not identify having a predictable distributed garbage collector as a goal for DRTSJ in any of its integration levels [19, 29]. DRTSJ may profit from the two options described in this paper; it may rely on manual remote object destruction and it may incorporate the RT-DGC service to its definition. In any case, the eventual specification should opt for an explicit default DGC management mechanism.

There are set of RT-RMI approaches that may benefit from the algorithms described in this paper to improve their approaches. Researchers from the University of York ([16]) worked in a framework for distributed real-time Java. Among the set of challenges they identified, having a predictable DGC mechanism is one of them. Thus, the
proposed solutions may be used to address the integration with a DGC mechanism. Researchers from UPM ([15, 30]) have defined rules for different distributed real-time Java platforms. All of their solutions forbid the use of a distributed garbage collector and some of them also the destruction of remote objects. The solution described for removing the garbage collector helps them to optimize its flexible framework.

Lastly, this piece of work continues our research on distributed real-time Java technologies named DREQUIEMI. Previous work focused on region-based memory management [31-33] within remote invocations, protocol optimizations [34], and other RTSJ enhancements [35]. The paper describes algorithms for supporting a predictable RT-DGC service, which are fully compatible with the previous framework.

VII. CONCLUSIONS AND FUTURE WORK
This paper dealt with problem of having a distributed garbage collector in a distributed real-time Java infrastructure. It identified the problems in current implemented approaches and proposed mechanisms to leave out DGC, defining extensions to the API, and specifying a real-time distributed garbage collector for RMI. The paper indentified a basic synchronous paper and asynchronous RT-DGC strategies. The proposed mechanism also incorporates the leasing mechanism, already available in RMI, which is parameterized as a real-time activity.

Our ongoing work is extending our evaluation to build a reference RT-DGC benchmark it can be a referent for distributed real-time Java applications. Other open research line is evaluating the integration of this service in RT-SOA architectures (following the approach described in [36, 37] and [38, 39][40][41] and producing application benchmarks based on [42]-[47] work.

ACKNOWLEDGMENT
This research was partially supported by the European Commission (ARTIST2 NoE, IST-2004-004527; iLAND ARTEMIS-JU Call 1) and by REM4VSS (TIN 2011-28339).

REFERENCES


