Submitted to Special Issue (ACM) JTRE

No-Heap Remote Objects for Distributed Real-Time Java

PABLO BASANTA-VAL, MARISOL GARCÍA-VALLS, AND IRIA ESTÉVEZ-AYRES

Universidad Carlos III de Madrid

This paper presents an approach to provide real-time support for Java's Remote Method Invocation (RMI) and its integration with the RTSJ memory model in order to leave out garbage collection. A new construct for remote objects, named *No-heap Remote object* (*NhRo*), is introduced. The usage of a NhRo guarantees that memory required to perform a remote invocation (at the server side) does not use heap-memory. Thus, the aim is to avoid garbage collection in the remote invocation process, improving predictability and memory isolation of distributed Java-based real-time applications. The paper presents the bare model and the main programming patterns which are associated with the NhRo model. Sun RMI implementation has been modified to integrate the NhRo model in both static and dynamic environments.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—Memory management (garbage collection); C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—Distributed systems;

General Terms: Real-time remote objects, region-based memory management

Additional Key Words and Phrases: real-time Java, distributed real-time Java, RTSJ, DRTSJ

This research was partially supported by the European Commission (ARTIST2 NoE, IST-2004-004527).
Authors' addresses: {pbasanta, mvalls, ayres}@it.uc3m.es, Department of Ingeniería Telemática, The University Carlos III de Madrid, Spain 28911.

1. INTRODUCTION

Currently, work on providing real-time support in middleware technologies is an active field of study and research. The advantages that this will offer for the development of distributed real-time applications are obvious. The main technology in this domain has been CORBA and specially the RT-CORBA [Fay-wolf et al. 2000] effort. However, on the way to simpler architectures, and in the area of one of the most popular programming languages of the moment as Java, new horizons appear. Java is spreading fast in the domain of the distributed applications. Java-RMI [Wollrath et al. 1996] [Sun 2004] is a very popular object-oriented middleware that includes many interesting features of Java technology, such as object orientation, strong typing, code mobility, virtual machines, class downloading, or security.

However, real-time applications require middleware that includes real-time support for predictable management of threads, memory, and network communication. Although there is an ongoing research effort to obtain the specification for a real-time RMI [Anderson et al. 2006] by merging RMI with the RTSJ (The Real-Time Specification for Java) [Bollella et al. 2000], currently Java RMI does not have a straightforward support for distributed real-time applications.

Among the main obstacles of Java for real-time systems are its complex memory model and its garbage collection mechanism. The later introduces unpredictable pauses in program execution. There are some proposals to solve the garbage collection problem, but none of them is perfect. On the one hand, turning the garbage collector off introduces changes in the coding paradigm, and it complicates the usage of the available Java implementations (libraries) [Al-Jaroodi et al. 2005]. On the other hand, real-time garbage collection techniques [Siebert 2008] [Bacon et al. 2003] clean up the memory in a predictable way, but at the cost of introducing an execution overhead. Eventually, region-based solutions (like scoped-memory, defined in the RTSJ, or similar solutions [Andreae et al. 2007]) require that the programmer be aware of the life duration of objects to then group them appropriately into the same region instance; somehow, this complicates the programming tasks.

The integration of region-based memory in middleware such as CORBA or RMI is challenging. There are approaches such as [Raman et al. 2005] that successfully use scoped-memory in the internals of the middleware to reduce the sweeps of garbage collection; however, the consequences of full replacement in the end-to-end path have not been studied well enough.

In this paper, a memory management solution for distributed real-time Java using scoped-memory is proposed. This solution avoids using the garbage collector. It introduces some changes in the middleware architecture and programming paradigm, but it offers a simple and efficient model for remote invocations that are self-cleaning with respect to their memory consumption. This model is targeted to simple distributed real-time applications with the following characteristics: (1) they require to keep their internal state (as a result of a remote invocation their internal state changes that needs to be stored), and (2) they create a well-known number of objects of a known size (each remote invocation has a bounded memory usage).

This paper is organized as follows. Section 2 contains the related work. Section 3 is a background section which introduces the memory management of RTSJ. Section 4 presents the model by explaining how the information of the memory model has been merged with the model of a remote object. Section 5 presents the model, from the point-of-view of a programmer, and Section 6 shows the changes which need to be introduced to support the model. Section 7 gives empirical evidences on the behavior of the model, which is compared together with the garbage collector. Eventually, Section 8 draws some conclusions and points out the direction of our related efforts.

2. RELATED WORK

This section shows how current distributed real-time Java solutions, based either on CORBA or RMI, are using scoped-memory in their architectures. Also, it presents the memory management solutions based on scoped-memory that are able to leave out the garbage collector in centralized and distributed systems. Table I contains the most relevant related works.

Denomination	**Technologies**	**Goals**
RTZen	RT-ORBA+RTSJ	Implementation of RT-CORBA with RTSJ
DRTSJ	RMI+RTSJ	A specification for distributed real-time Java
RT-RMI-York	RMI+RTSJ	A framework for distributed real-time Java.
RT-RMI-UPM	RMI+RTSJ	Profiles for distributed real-time Java
DRTJava-on-CSP	CSP+Java	Produce an alternative based on CSP formalisms
APICOM	RTSJ+CAN	A distributed real-time object-oriented platform
Scratchpad	RTSJ	A no-heap component model for real-time Java
RTJ-COM	RTSJ	A component framework for embedded real-time Java

Table I: Memory management for distributed real-time Java systems (related work)

2.1 Distributed real-time Java

On the one hand, one approach to produce a distributed real-time Java solution is to use the CORBA model and the RT-CORBA specification. A priori, the existence of a RT-CORBA specification seems to facilitate the task, since it is only required to provide a language mapping (as it was already done with C and C++). However, the scoped-memory and the existence of multiple types of threads make the mapping non-trivial.

The RTZen project [Raman et al. 2005] presents a hybrid strategy for performing this mapping. It relies on the usage of the garbage collector and also on scoped-memory. It uses the scoped-memory inside the middleware while the parameters of the remote invocation passed to the remote object (servant, in RT-CORBA jargon) are created in the heap. Thus, it improves the efficiency of the middleware, but the garbage collector is maintained in the end-to-end path.

On the other hand, distributed real-time Java may be based on RMI. RMI incorporates features like distributed garbage collection and remote-code downloading since RMI is the natural distribution model of Java. This is not the case for CORBA, where such features are not available. However, there is no current real-time RMI specification and the definition of the solution will have to tackle many of the problems that RT-CORBA has addressed already. Therefore, one of the main advantages of real-time RMI is that it may produce a more integrated solution in the Java domain.

There is an effort which is working on the definition of a Distributed Real-time Specification for Java (DRTSJ) [Anderson et al. 2006]. This specification extends RTSJ to create a real-time RMI which provides support for predictability and end-to-end timeliness of trans-node activities. There is no public draft available yet, but some work in this direction has already appeared. In relation to the memory management issues there is a tagging interface, `NoHeapRealtime`, which ensures that the underlying transport of messages and remote object invocation do not use heap memory. However, in this work the way scoped-memory is used to obtain these semantics is not addressed; it is just a design requirement. The no-heap real-time remote object (NhRo) approach handles this issue.

A framework for the RMI and RTSJ integration [Borg et al. 2003] was proposed by the University of York. This framework suggested the use of scoped-memory in remote invocations. However, the framework is focused on the problems of the integration, not analyzing the impact that an intensive use of scoped-memory has in the programming

paradigm or inside the middleware. The NhRo approach presents and discusses these changes in detail.

In [Tejera et al. 2006], the real-time group of the Universidad Politécnica de Madrid addresses the integration of network resource reservation mechanisms in the RMI architecture and its implications in the internals of RMI. However, it does not consider the penalties of the garbage collector in the end-to-end communications.

A third alternative [Hilderink et al. 2000] relies on the Communicating Sequential Processes (CSP), a formal language for describing patterns of interaction in distributed systems. This framework covers real-time aspects related to the assignment of priorities which are attached to the communication channels. However, the issue of producing predictable memory management models for their infrastructure seems to be beyond the scope of the paper, which only suggests the use of a real-time garbage collector.

Finally, other related work as [Silva et al. 2006], defines a communication API based on RTSJ (APICOM) for embedded systems. However, the use of region-based memory models and the serialization of Java objects are out of their scope.

2.2: Programming with regions

Although there is a number of patterns for the scoped-memory and its efficient implementation and usage (e.g. [Corsaro 2004] or [Higuera-Toledano 2006]), there are not many solutions that consider the use of scoped-memory as a real alternative to the garbage collector.

One notable exception is [Bollella et al. 2003] defining the scratchpad, a real-time component model; this is a high-predictable construct for components in centralized systems. This component model has been extended by Plsek [Plsek et al. 2008], introducing additional support for real-time threads in embedded systems. Another exception is [Basanta-Val et al. 2005a], where the authors originally proposed NhRo for distributed systems. Although the scratchpad targets centralized (mono-virtual machine) systems and the NhRo focuses on distributed systems, both share some similar principles. Both make a difference between the state of a component (or remote objects in the case of the NhRo) and the set of temporal objects that change this state; they are placed in different memory-areas (or memory-contexts, in the jargon of NhRo).

3. BACKGROUND ON THE MEMORY MANAGEMENT OF RTSJ

Before explaining the details of NhRo, this section presents the memory model of RTSJ, focusing on the memory types it defines to avoid garbage collection. Furthermore, it explains the structure used to manage the non-collectable memory types: the scope-stack.

3.1 Memory areas

On the opposite side to the memory model of Java, where only the heap memory may be used to allocate objects, the memory model of RTSJ is based on the concept of memory-areas. Each memory-area defines an allocation-context, with different properties in terms of life and predictability, where programs may allocate Java objects. RTSJ includes a memory-area for the heap, adding two new allocation-contexts for real-time systems: *immortal* and *scoped*. As a whole, immortal and scoped are known as no-heap memory.

The main properties of each memory-area are the following ones:

- **Heap-Memory:** this is the classic Java heap. It supports the allocation and de-allocation of objects. In this memory-area, the de-allocation of objects is performed by the garbage collector automatically. Therefore, threads that use this memory suffer the penalties of the garbage collector.
- **Immortal-memory**: this memory-area is provided by the virtual machine. It does not support the de-allocation of objects. Once an object is allocated in immortal-memory, the object lives forever until the virtual machine ends its execution. Therefore, this memory is typically used to store static objects which have to be maintained during the entire life of an application.
- **Scoped-memory**: this memory has to be created by the programmer. The programmer creates scoped-memory instances and allocates objects in them. Scoped-memory instances support the allocation and de-allocation of objects. De-allocations are performed by an internal counter which is stored in each scoped-memory instance. Its granularity is in between heap-memory and immortal-memory, thus enabling predictable run-time collection of temporal objects.

3.2 Scope-stack

The existence of multiple memory-areas requires a mechanism to keep track of the regions where objects are allocated in. In the particular model of RTSJ, this mechanism is indirect and is based on an internal structure associated to each thread: the *scope-stack*

(Figure 1). The scope-stack contains information about the memory-areas which are accessed from a thread. Besides, the API of RTSJ includes methods (`enter` and `executeInArea`) that allow pushing and popping memory-areas in the local scope-stack of the current thread.

To control the allocation of objects, there is an element in each thread, named *default allocation-context*, which decides in which memory-area an object is allocated by default. For instance, in Fig. 1, the instantiation of a new object `o1` reserves its memory in the current allocation context (i.e. $\texttt{scope}_b$).

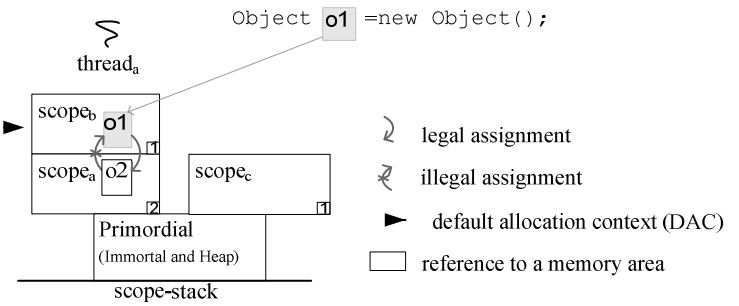

Fig. 1. Scope-stack

The destruction of objects that are created in a scoped-memory is performed by a counting algorithm. Each scoped-memory instance has an internal counter that is incremented (+1) each time a scoped-memory is pushed in the local scope-stack of a thread, and decremented (-1) when popped. When the counter reaches zero (1→0), all objects stored in the scoped are collected. In Figure 1, `o1` is not reclaimed until its related scoped-memory counter reaches zero. In the example, this happens when $\texttt{scope}_b$ is popped out from the local scope-stack.

RTSJ defines two rules that guarantee a safe use of the scoped-memory instances pushed in and popped from the scope-stack. These are the *assignment rule* and the *single-parent rule*. The assignment rule forbids references to objects stored in inner scopes, and references are only possible to objects stored inside the same scope or in outer scopes. For instance, in Figure 1, the object `o1` can reference only objects allocated in its outer regions (named the *creation-context* of the object), namely $\texttt{scope}_b$, $\texttt{scope}_a$, `heap`, and `immortal`. The single-parent rule states that the scoped-memory instance that is just below one scope is always the same. In the example of Figure 1: the parent of $\texttt{scope}_b$ is

`scope`$_a$; the parent of `scope`$_a$ is the `primordial` scope (i.e. the combination of heap and immortal memory); and the parent of `scope`$_c$ is also the `primordial` scope.

4. DEFINITION OF SCOPE-STACKS FOR THE REMOTE INVOCATION

Integrating no-heap memory into RMI may be directed at two kinds of scenarios: *static* and *dynamic*. In this paper, a static scenario is one where, once created in immortal memory, a remote object is never destroyed; besides, each remote object may receive or perform an unlimited number of remote invocations, depending on whether it is a remote server or a remote stub. In a dynamic scenario, they can be created also in scoped-memory and deleted when they are no longer needed. Our experience shows that in the case of static scenarios [Basanta-Val et al. 04], the middleware can be implemented without having to include mechanisms that violate the assignment-rule; in general, dynamic scenarios require the integration of mechanisms that violate the assignment-rule within the middleware infrastructures to facilitate its implementation [Basanta-Val et al. 05a].

In addition, the definition of a solution has to deal with the issue of associating each object created in the middleware context with a particular type of memory-area of RTSJ. The RMI context creates Java objects each time a remote entity: (1) is created, (2) receives a remote invocation, (3) invokes a remote method, (4) sends back a response to the client, or (5) is destroyed. Hence, its model allows deciding in which memory-area should these objects be allocated; this may be immortal-memory, a specific scoped-memory instance, or heap-memory. Currently, the programmer can decide about the creation and destruction of remote entities; however, the API is insufficient to decide in which type of memory ought to be created the parameters for the remote invocation; this is due to the fact that the interfaces of RMI do not support such a facility.

Therefore, in order to be able to combine the use of the memory model of RTSJ and the remote object technology in a middleware such as RMI, it is necessary to merge the remote invocation and the information of the scope-stack. The internals of this middleware are able to copy objects from client to server, and back from server to client, in order to perform the remote invocation. However, they are not able to decide where these objects are allocated; such a decision relies on the scope-stack of the thread and its default allocation-context.

4.1 Definition of scope-stacks of the invocation: creation and invocation contexts

In the NhRo approach, both middleware and programmer are aware of a scope-stack at client and server. The middleware uses two scope-stacks for the remote invocation: one for the client's side and another one for the server's side. At the client, the scope-stack defines an interface with the invoking thread. It decides where the results of the remote invocation are allocated. At the server side, the scope-stack provides the interface for the communication with the remote object instance.

At the client, the model for a NhRo uses the scope-stack of the invoking thread to allocate the results of the remote invocation, as it happens in any invocation to a local method. The middleware uses the default allocation-context of the invoking thread to allocate the objects returned by the remote invocation. This solution is, therefore, compliant with the characteristics of the remote invocation: the thread that initiates the remote invocation does not differentiate whether the invocation is remote or local. In both cases, the objects which are returned as result of the remote invocation are created in the default allocation-context of the invoking thread. For instance, in Figure 2, the invocation of a remote method in the local stub RS_o allocates the results of the invocation to the `add` method (in the example, the object `c`) within the default allocation-context of the invoking thread (in the figure, $scope_r$).

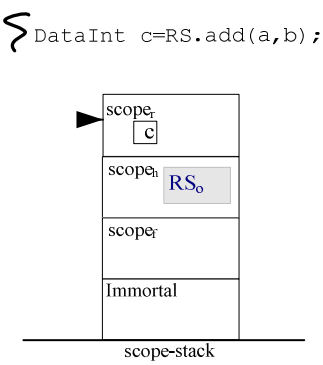

Fig. 2. Scope-stack at the client side

The scope-stack that is used at the server side during the remote invocation is more complex than the one used at the client. Figure 3 shows this idea with two situations. The one in the left side illustrates the creation of the remote object which will be invoked remotely; it shows the scope-stack of the thread that creates the remote object. The figure on the right presents the remote invocation at the server, showing the scope-stack of the handler thread (the thread that invokes the remote object at the server).

There is a common part between scope-stacks: $\texttt{scope}_b$, $\texttt{scope}_a$, and `primordial`. This common part is called the creation-context of the remote object (`Ro`), and it stores the state of the remote object. There is also a new set of memory instances (in the example $\texttt{scope}_c$), which are taken from a new entity of the middleware called memoryarea-pool, pushed on top of the creation-context. They are named invocation–context and store the parameters of the remote invocation and other auxiliary objects of the invocation.

Notice that the safety in the access to the remote object instance is maintained for a remote invocation due to the handler thread that reconstructs the creation-context of the remote object (`Ro`) before pushing a new invocation-context in its scope-stack (Figure 3). This new invocation context should not have any parent, and it is taken from a memory-area pool which is a new entity of the middleware in charge of provisioning new invocation contexts (see Section 6 for more details).

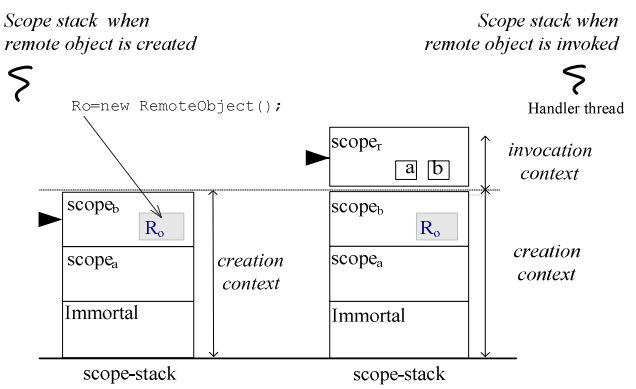

Fig. 3. Scope-stack at the server side of the remote invocation of a NhRo

As a consequence of the scope-stack chosen for the remote invocation at the server side, a new set of constrains appear in the remote invocation. Collectively, they are called the no-heap remote object paradigm (NhRo-paradigm). The programmer has to pay a special care during the coding of an application and in the internals of the middleware which have to be modified to give a proper support for this model. Next sections (5 and 6) cover these issues in detail.

5. PROGRAMMING WITH NO-HEAP REMOTE OBJECTS

The differences between using the traditional remote object paradigm and the no-heap remote object paradigm are mainly two. The first one is that the memory structure

perceived by a programmer is not plain; it is fragmented into two contexts. The programmer should be aware of the existence of two contexts and a special assignment rule between them. The second difference is that the programming model has a few more limitations than the one based on heap-memory; not all remote objects based on heap-memory may be directly supported by the NhRo paradigm.

5.1 Fragmentation in nested contexts

From the programmer's perspective, the memory is divided in two nested contexts: the creation-context and the invocation-context.

- ***Creation-context***: it contains all the objects that represent the state of the remote object. Objects belonging to this context are not de-allocated before the remote object instance. The internals of the middleware maintain the creation-context of the remote object until the remote object is no longer in use.
- ***Invocation-context***: it stores the objects that are created during the remote invocations. The memory for the invocation–context is provided by the middleware, and it is recycled after the remote invocation, also by the middleware.

5.2 No-heap Remote Object assignment rule: NhRo-rule

There is an assignment rule between the two contexts. This rule is called the *no-heap remote object assignment rule* (NhRo-rule), and it is a consequence of the traditional assignment rule when applied from a scope to an outer memory-area. The NhRo-rule forbids references from objects stored in the creation-context to objects stored in the invocation–context. The opposite is possible: objects allocated in the invocation–context may reference objects allocated in the creation–context.

The NhRo memory-contexts definition is non-restrictive; each context may be a single memory-area instance or more than one, depending on the type of application which is been developed. This allows multiple specializations of the model depending on the scenario where the NhRo is to be deployed. Thus, in static environments, it is reasonable to use only immortal-memory, whereas in dynamic environments, it may use scoped-memory or immortal. In both scenarios, the NhRo-rule keeps its semantics: the life of objects created in the invocation–context is shorter than the life of objects created in the creation–context.

5.3 Programming under the no-heap remote object paradigm

The no-heap remote object paradigm introduces two main constraints when compared against the traditional one. The first one is inherited from the use of scoped-memory instead of heap. The second one comes from the division of memory into two contexts and the NhRo-rule.

5.3.1 Constraints due to scoped-memory

Under this paradigm and during the remote invocation, the programmer should not allocate objects in the creation-context, but s/he may allocate a bounded number of them in the invocation-context.

- **Allocation in creation-context**. The allocation of objects during the remote invocation in the creation context may lead to memory leaks; therefore, this practice should be avoided. Memory leaks occur due to fact that contents of the creation–context cannot be recycled while the remote object instance is being invoked remotely. So, all memory that is required for the creation-context has to be pre-allocated during the creation of a remote object, or it has to be taken from a shared pool-of-objects which is managed by some external API or other specific mechanisms. This practice helps programmers to avoid the problems of the memory leaks in the code of their remote objects.
- **Allocation in invocation-context**. The creation of objects in the invocation contexts does not introduce such memory leaks. The middleware recycles the objects created in an invocation-context after the end of a remote invocation. However, the amount of memory that a remote method may need to allocate in the invocation–context has to be finite and bounded; the scoped-memory instance that is used in the invocation context may not be partially recycled while the remote invocation is active. Fortunately, from the point-of-view of a real-time application, such restriction is not new and may be integrated within the worst case execution analysis of the distributed system.

5.3.2 Constraints introduced by the NhRo-rule

The second type of constraints comes from the definition of the NhRo-rule. The NhRo-rule separates objects that belong to the state of a remote object from those that are created during a remote invocation. This separation enables the recycling of the invocation–context after the remote invocation; however, this kind of behavior does not

always suit the needs of the application. A problem arises when the programmer wants that any object created during the remote invocation is not recycled after it has finished. The NhRo cannot, as done in traditional remote objects, maintain a reference to the object. This kind of reference is forbidden by the NhRo-rule since the referenced object is in the creation–context whereas the referenced object is in the invocation-context.

There are two main techniques to mitigate this problem:

- **Copy pattern**. A first solution is to copy the contents of the object from the invocation-context to another object pre-allocated in the creation–context. Unfortunately, this mechanism does not work in all cases because not all objects of Java may be copied from one memory to another. An example of an object which cannot be copied so easily is a real-time thread or a memory-area.
- **Reference pattern**. A second option is to avoid the collection of the invocation–context after the remote invocation. One easy way to carry out this second option is to create an auxiliary remote object, acting as a wedge thread, in the invocation–context. The creation of the auxiliary remote object guarantees that the object is not reclaimed while the auxiliary remote object is able to receive remote invocations. Besides, in the case of a remote object, the remote interface of the auxiliary remote object allows accessing and modifying the object. A more interesting choice is the use of a special mechanism, for example a mechanism named `ExtendedPortal` [Basanta-Val et al. 2006], to break the NhRo-rule and store in its creation-context a reference to an arbitrary object. The goal of this extension is similar to the pinned scoped memory of RTSJ 1.1 [JSR-128 2009]; in both cases the aim is to have easy-to-use access to arbitrary Java objects.

5.4. An example of NhRo

This section shows a simple example for the set of applications that are the target of the NhRo model. Figure 4 is a representative example of the distributed real-time applications that: (1) need to keep their internal state, and (2) produce a limited number of objects as a result of their operation. This example also illustrates the changes introduced in the programming model, i.e., how to store the changing state of an application as a result of a remote invocation. In this particular case, the remote object corresponds to a simple calculator that requires maintaining in its internal state the result of the last operation, and it helps illustrate the concept of creation and invocation contexts and the NhRo-rule together with its associate programming patterns.

5.4.1 Identifying the creation and invocation contexts

The assignment of the objects to either creation or invocation context is implicit in the definition of the remote object.

DataInt.java

```
public class DataInt implements Serializable{
 private  int internalvalue=0;
 public   DataInt(int i){internalvalue=i;};
 public   int intValue(){
            return internalvalue;}
 public   void setIntValue(int i){
             internalvalue=i; }
}
```

Calculator.java

```
public interface Calculator extends java.rmi.Remote{
 public     DataInt add(DataInt i, DataInt o)
              throws java.rmi.RemoteException;
 public     DataInt lastResult()
              throws java.rmi.RemoteException;
 public     void doNothing()
              throws java.rmi.RemoteException;
 public     Calculator getNewCalculator()
              throws java.rmi.RemoteExceptioo;
}
```

Invocation Context

CalculatorImpl.java

```
01: public class CalculatorImpl extends
02:   es.uc3m.it.drequiem.rtrmi.NoHeapRealtimeRemoteObject
03:   implements Calculator{
04:     private DataInt lastInteger=new DataInt(0);//Last result
05:     private ExtendedPortal ep= new ExtendedPortal(null);
06:     public CalculatorImpl()
07:               throws java.rmi.RemoteException{
08:               super(0, false, null, null, null,
08:                     new LTMemoryAreaPool(5,5,1000,1000) //IC factory
09:               ,null);}
```

Creation Context

```
10:     public  DataInt add(DataInt a, DataInt b)
11:       throws java.rmi.RemoteException{
12:                     int aux1=a.intValue();
13:                     int aux2=b.intValue();
14:                     DataInt tmp=new DataInt(aux1+aux2);
15:                     lastInteger=tmp; //violates NhRo-rule !!
16:                     return lastInteger;
17:     }
18:     public  DataInt lastResult()throws java.rmi.RemoteException{
19:               return lastInteger; }
20:     public void  doNothing()throws java.rmi.RemoteException{
21:         }
22:     public  Calculator getNewCalculator() throws ... {
23:         Calculator newC=(Calculator)new CalculatorImpl();
24:         return newC;
25:     }
26: }
```

Invocation Context

Fig. 4. A remote calculator implemented with NhRo paradigm

According to the NhRo model, the objects that belong to the state of the remote object are in the creation-context. Thus, the remote object instance and all objects that are referenced from its attributes are implicitly created in the creation-context. In the example of Figure 4, `CalculatorImpl` instance and the object referenced by `lastInteger` and `ep` are all in the creation–context of the remote object.

According to our model, the objects which are used during the remote invocation belong to the invocation–context. Thus, when a method of the remote object is remotely invoked, all its objects are implicitly in its invocation–context. In the example of Figure 4, when `add`, `lastResult`, `doNothing`, and `getNewCalculator` are remotely invoked, `a`, `b`, `tmp`, and `newC` are allocated within the invocation–context of the remote object. The invocation-context of the object is specified in the constructor of the remote object; in the constructor the programmer defines the memoryarea-pool that is used in the remote invocation.

In this particular example (line 8 in Figure 4), the invocation-context corresponds to an `LTMemory` instance which is taken from an `LTMemoryAreaPool`. It may offer up to 5 simultaneous invocation contexts (each one of 1000 bytes) to the remote object. In a more complex application, several remote objects could share the same memory-area pool, if needed, using in their constructors the same reference.

The pre-allocation of memory in the creation context is implicit. The objects allocated during the creation of the remote object are in the creation–context. In the example of Figure 4, pre-allocation in the creation-context is performed using the `new` statement in the following lines: 04, 05, and 08.

5.4.2. Fulfillment of the NhRo-rule

Unfortunately, in this example, the `add` method violates the NhRo-rule (line 15), so that it has to be modified. This example illustrates the type of applications that need to maintain their state; in this case, the calculator has to keep (remember) the result of the last operation that it performed. This result is allocated in the invocation-context (in `tmp`), and it is assigned to an attribute of the calculator allocated in the creation–context (in `lastInteger`). Therefore, in the following statement: `lastInteger=tmp;` the NhRo-rule is contravened.

A solution, as stated before, is to copy the contents (not the references) from the invocation-context to objects which are pre-allocated in the creation–context. In the example of the calculator, to use this pattern to store the state of the remote object, the above statement should be replaced with a slightly more complex one: `lastInteger.setIntValue(tmp.intValue())`.

There is also another interesting alternative which relies on the violation of the RTSJ assignment-rules using an extended `ExtendedPortal` to maintain a forbidden reference. In this case, the statement at line 15 (`lastInteger=tmp`) should be replaced with the

following one: `ep.setPortal(tmp)`. Besides, to be fully operative, this approach requires an additional change at line 19 (`return lastInteger`), which should be replaced by a slightly more complex sequence which is required to copy the resulting object from the extended portal to a temporal object allocated in the invocation-context of the remote object:

```
19.1: DataInt tmp2= new DataInt();
19.2: ep.enter (new Runnable(){
19.3:   public void run (){
19.4:   tmp2.setIntValue(((DataInt)ep.getPortal()).intValue());
19.5:   });
19.6: return tmp2;
```

To return the result of the last operation, one cannot access directly to the portal contents; this would raise an illegal access exception. Instead, a specific pattern for the extended portal [Basanta-Val et al. 2006] has to be used; the pattern consists of two methods (`enter` and `getPortal`) that collectively guarantee a safe access to the object referenced from the portal (`ep`). The `enter` method accesses the creation-context of the object referenced from `ep`, while `getPortal` accesses the reference of the object that stores the result of the last operation (i.e. `tmp2`).

Another alternative is to modify the code of the `lastReturn` method, in order to return a simple object. In this case, the signature of the remote object should be `public Object lastReturn()` instead of `public DataInt lastReturn()`. Introducing this change in the interfaces, it is still needed to change the return statement at line 19 (`return lastInteger`) with the following one: `return ep`. Internally, the serialization of an extended portal transfers a copy of the referenced object, instead of a copy of the extended portal.

Eventually, it is important to notice that `getNewCalculator` does not violate the NhRo-rule. There is no reference from the creation to the invocation-context. However, in this case, the invocation–context plays a double role. The invocation–context for the remote object processes the remote invocation, and it is also inside the creation–context of the newly created remote object. In this particular example, the invocation-context of the parent remote object is not released before the creation-context of its descendant does. This behavior is similar to the creation of a nested thread in RTSJ: the child thread inherits the scope-stack of the parent and the common regions are not destroyed before the child thread is alive.

6. SUPPORT FOR NO-HEAP REMOTE OBJECTS

In this section, the architecture of the integration of the NhRo model and RMI is presented for static and dynamic environments. In this context, static means that is not possible to destroy a NhRo once it has been created, whereas in a dynamic environment such constraints are relaxed.

6.1. Minimum middleware for NhRos

As a minimum support for the NhRo model for static environments, we observe the following constraints:

- The NhRo's creation-context contains only immortal-memory, and it cannot contain any scoped-memory instances.
- The NhRo's invocation–context uses a single `LTMemory` instance, which is obtained from an `LTMemoryAreaPool`.
- Threads that perform up-call are the RTSJ's no-heap real-time threads.

Figure 5 shows a minimalist architecture for the integration of NhRo model into RMI for static environments. All objects are created in immortal-memory and only those objects needed to serve remote invocations are created in scoped-memory (`LTMemory`).

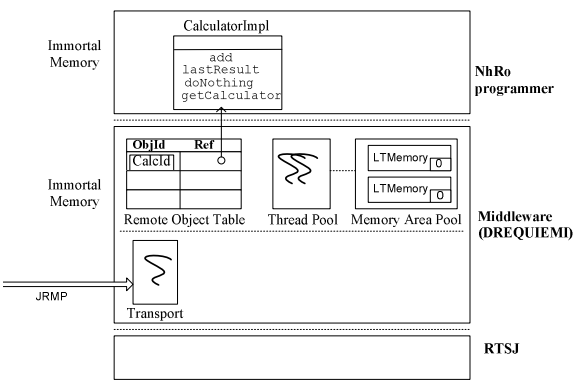

Fig. 5. Server-side view of NhRo-RMI integration

Sun RMI middleware solution has been slightly adapted to implement the support required for NhRo, as part of DREQUIEMI's [Basanta-Val et al. 2007] architecture for networked real-time systems. The key parts of the middleware are: the transport component, the remote-object table, the thread-pool, and the memoryarea-pool.

- **Transport**. It allows communications between two Java virtual machines using Java's Socket API. Each transport has an associated RTSJ's no-heap real-time thread that monitors connections.

- **Remote-object Table**. It contains information of NhRo instances that can be remotely invoked. Among others, this information contains a remote–object identifier (`ObjID`) and a pointer to the remote object instance. Each time a new NhRo instance is created, its constructor attempts to register itself in this table.
- **Thread Pool**. This structure is based on RT-CORBA's thread-pool. It limits the number of concurrent invocations at the server and improves server side remote invocation efficiency and predictability. DREQUIEMI's thread-pool requires that the programmer sets a single parameter: the number of threads stored in the pool.
- **Memory-area Pool**. This structure is a completely new idea that comes out of the architecture. It complements the thread-pool structure in the middleware for distributed real-time Java applications. The main idea beneath a memory-area pool is parallel to the thread-pool one. Each time a scoped-memory instance is created, some memory resources in the operating system are reserved. After using these resources to perform a remote invocation, it is interesting to release them, instead of allocating new ones.

As thread-pools do, memory-area pools also need to be dimensioned. The programmer must set up the number of instances of the pool, its size and relationship between thread-pool instances and memory-area pool instances. The default policy of the memory-area pool requires that the programmer supplies a single parameter: the size of the memory-area instances stored in the pool. All memory-areas stored in the pool have the same size. The number of memory-area instances stored in the memory-area pool equals the number of threads in the thread-pool. Hence, the existing association between threads and memory-areas is one-to-one; each thread uses a single `LTMemory` instance in a remote invocation.

6.2. Functionality supplied to NhRos

The modified RMI provides to NhRo objects the following functionality, just as the standard RMI:

- **Exportation**. The registry mechanism is maintained; a NhRo is registered in it in the same way a normal remote object does.
- **Remote Invocation**. A NhRo can be remotely invoked in the same way as a normal RMI remote object.

6.2.1 Creation and invocation of NhRos in static environments

Each time an instance of a NhRo is created, its constructor tries to register itself in the RMI internals in order to make it remotely accessible. RMI internals verify that the NhRo has been created in immortal-memory. If not, it throws a remote exception notifying that the remote object instance must be created in immortal-memory. The registration process implies the creation of a new `ObjID` instance; this `ObjID` is used by clients to refer to a particular remote object instance. After a successful registration, the NhRo instance may be invoked from other virtual machines.

6.2.2 Remote Invocation of NhRos

Each time the transport thread detects that a client has started a remote invocation process, it selects a handler thread from the thread-pool. This thread will complete the remote invocation process.

This handler thread gets an `LTMemory` instance from an `LTMemoryAreaPool` and enters it (using the `enter` method). This changes its default allocation-context to the specified `LTMemory` and increments (+1) its internal counter. From this point on, all objects will be created in this memory-area instance. After entering the specified memory-area, the handler thread de-serializes the object identifier (`ObjID`), the method identifier, and the remote method parameters. All these objects are implicitly created in `LTMemory`. After that, the thread uses the `ObjID` to query the remote-object table and gets a reference to the right NhRo instance. Objects created in the remote object's method are created by default in `LTMemory` (e.g. the object referenced by `tmp`). After performing the up-call, it sends back a response to the client. Objects created at this stage are also created in `LTMemory`. Eventually, after the remote invocation has finished both, the handler thread and `LTMemory` instance, which is decremented (-1), return to their corresponding pools.

6.2.3 Implementation of the NhRo-rule

In DREQUIEMI, the NhRo-rule is implemented as a result of RTSJ assignment rules. All NhRo instances and the objects referenced from their attributes are created in immortal-memory, while objects of the invocation–context are in scoped-memory. RTSJ assignment-rule does not allow that a reference to an object created in `LTMemory` be stored in immortal-memory. It should be noted that an instance to a remote object could always be referenced from the RMI internals because it is placed in immortal-memory.

`LTMemory` usage does not violate the single-parent rule because `LTMemory` instances are never shared between two threads at the same time.

Eventually, the complexities of manipulation of the scope-stack structure have been hidden to the NhRo programmer. Before invoking the user-level code, threads change their default allocation-contexts using `enter`. The programmer only needs to know that there are two allocation-contexts and that objects created in remote object methods disappear after the method has finished, thus fulfilling the model.

6.3 Modifications for dynamic environments

The basic support described in sections 6.1 and 6.2 may be insufficient for dynamic environments due to two assumptions that are not valid in these environments:

- Each thread in the thread pool has a unique `LTMemory` instance which is associated with a specific `LTMemoryAreaPool`.
- All remote objects are allocated in immortal-memory and cannot be de-allocated.

6.3.1 Extension to the LTMemoryAreaPool

DREQUIEMI defines a default policy for an `LTMemoryAreaPool` in dynamic environments; it is based in pre-allocation and requires two configuration parameters (note: in a static environment, the second parameter is enough to configure the thread-pool [Basanta-Val et al. 04]):

- The number of invocation–contexts that the memory-area pool may provide concurrently. This information is supplied with two configuration parameters: a minimum number of instances which are guaranteed by the middleware (line 8 in Figure 4), and a maximum number of memory-areas which may be instantiated to satisfy temporal peaks in remote object demands.
- The maximum amount of memory that any remote invocation may consume. Once more, this information is specified with two values: one minimum guaranteed value and a ceiling value, which is associated to maximum demands in concurrency.

Using these two parameters, the default policy of the memory-area pool pre-allocates $invocation_{maxnumber}$ invocation-contexts and each one contains $invocation_{size}$ amount of memory. As shown in figure 6, it is implemented with a linked list of scoped-memory instances. When the counter of an invocation context changes from one to zero the invocation–context is recycled and it is added at the end of the list. When a handler

thread performs the remote invocation, it gets the memory for the invocation-context from the head of the list.

The number of threads in the thread-pool and the maximum number of invocation contexts are not the same, as in the architecture for the static environment. The difference between the two numbers fixes the maximum number of invocation-contexts that may be used at the same time to hold (using for example extended portals) data without reducing the concurrency of the server.

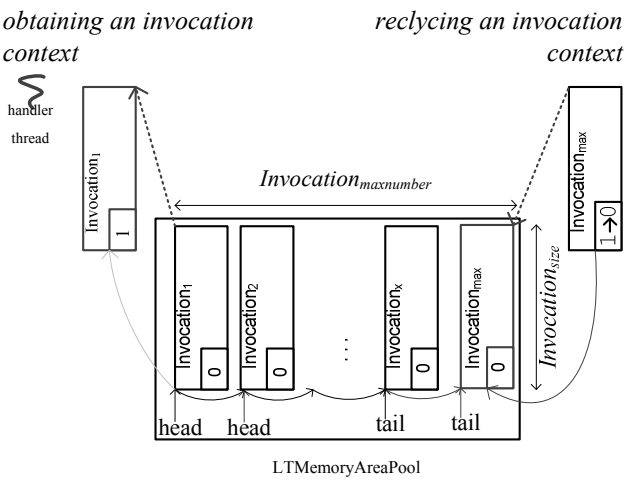

Fig. 6. Default policy for the management of invocation-contexts (dynamic environments)

6.3.2 Management of creation-contexts

In the case of not enforcing the creation of a remote object in immortal-memory, extra requirements have to be demanded from the middleware.

In order to maintain the semantics, during the creation and destruction of the remote object, the counters of scoped-memory instances that are in creation-context are modified. When a remote object is able to receive remote invocations (creation), they are incremented (+1). When a remote object is not able to receive more remote invocations (destruction), they are decremented (-1). In our implementation, this process is done automatically allocating in the middleware an extended portal which refers to the remote object instance (Figure 7).

The handler thread uses the remote-object table and the memory-area pool during the remote invocation to build its scope-stack. The handler thread first reads the `ObjID` from the parameters of the remote invocation sent by the client. Using `ObjID`, it queries the

remote-object table to get its corresponding creation-context (step 1 in Figure 7), which is pushed in a new scope-stack.

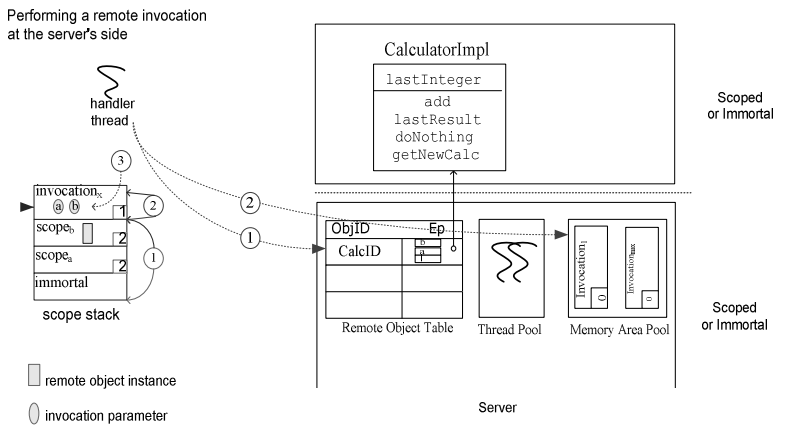

Fig. 7. Handler thread just before invoking the method in the remote object instance (dynamic invocation)

After that, it obtains the memory for the invocation-context and pushes it on top of the scope-stack (step 2 in Figure 7). Once it has created the scope-stack, it allocates the parameters for the remote invocation in the invocation-context. Figure 7 shows the state of the handler thread at this time. After that, the handler thread invokes the remote object. In order to do that, it reads the reference (`ep`) to the remote object from the remote-object table and invokes the appropriate method of the remote object. After invoking the method of the remote object, the thread sends back the results of the remote invocation to the client and pops the invocation and creation contexts.

To end this section, Table II shows how the counters of the region model of RTSJ are modified by the NhRo-paradigm. The table differentiates between counters that are modified directly by the middleware (*middleware*, in the actor column) from those actions that can be initiated from the logic of the application (*program*, in the actor column). The table shows how the creation-context of a remote object is incremented by the middleware during the creation and invocation of a remote object, while the middleware only increments the invocation-context of the object before performing the up-call (see section 6.3 for additional explanations). Likewise, the counters of the creation-context are decremented by the middleware whenever the remote object is destroyed and after performing the up-call. Lastly, as stated in section 5.3 and 5.4.2, the use of an extended portal and nested remote objects also alters the value of the counters.

In all these cases, whenever the counter of scoped-memory instance reaches zero, all the objects that it stores are destroyed.

Actor	Kind of region	Action	
		Incremented (+1)	**Decremented (-1)**
Middlew.	Creation-Context	During the creation of a remote object Before performing the up-call	During the destruction of a remote object After performing the up-call
	Invocation-Context	Before performing the up-call	After performing the up-call
Program.	Both, Creation and Invocation Contexts	During the creation of a nested remote-object in the invocation context. Setting an extended portal reference to an object allocated in the invocation-context	During the destruction of a nested remote-object Unsetting an extended portal reference to an object allocated in the invocation-context

Table II: How the counters of a region are modified by the NhRo-paradigm

7. EMPIRICAL EVALUATION

The current model, which is part of the DREQUIEMI architecture for networked real-time systems, has been validated empirically through an implementation done from the JTime virtual machine [TimeSys 2004] and the classes of RMI for J2ME. This implementation allows establishing parallelisms, which are summarized in this section, between the performance of classical garbage collection and regions under identical conditions.

7.1 General behaviors

The main advantage given by a region-based memory model is that it offers the programmer a plain model where the cost of invoking a certain remote object is almost constant. As shown in Figure 8, the use of a scoped-memory (an `LTMemory` instance in the experiment case) is able to avoid the latencies of the garbage collector, associated with the heap-memory. The garbage collector (in this case, the generational garbage collector which comes with JTime [TimeSys 2004]) introduces high penalties of almost ten times the average response time. In our experiments, the worst case penalty (Figure 9) may reach almost one order in magnitude (from 1 to 10 ms) depending on whether the NhRo-paradigm is or not used. It is expected that the differences between a real-time garbage collector and regions are much smaller, but unfortunately the implementation of

JTime (the virtual machine used during the implementation) does not support this kind of facilities yet.

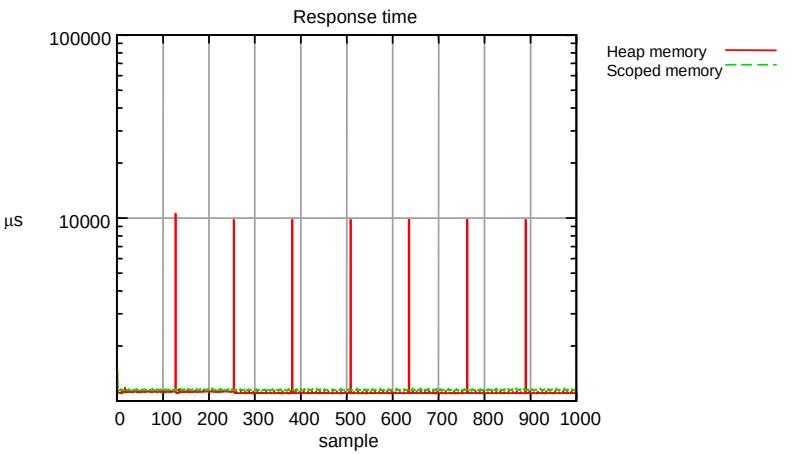

Fig. 8. Garbage and region-based collection in remote invocations (976 MHz)

Zooming into the best case response times (Figure 10), we may see how (in the best cases) the performance of a garbage collector is slightly better than the one given scoped-memory since it does not introduce the extra burden of manipulating creation-contexts, the extra checks required in the validation of the single-parent rule, or the increment and decrement of the counters of the `LTMemory` instances. In this case, the differences between the two techniques (50 µs) may be understood as the cost of recycling the regions in each invocation versus having infinite memory.

7.2 Average overheads

Another experiment carried out was to calculate the average overhead introduced in the best cost of the remote invocation which is caused by each automatic memory management technique. To this end, a small synthetic benchmark was built. It consists in the transmission of a variable number of expensive-to-destroy objects as parameters of a remote invocation. Each one of these objects executes a single assignment within a loop ten thousand times in its finalizer. Our findings (Figure 11) reveal the garbage collector (with 2 Mb of heap) as the best technique, in average terms. Even when the remote invocation is empty, i.e. it does not contain any object, the difference in average terms is 2 % worse than the use of regions. This initial difference increases with the number of objects because the garbage collector may execute the finalizers more efficiently than a

region-based model. However, once more, it is important to note that this experiment considers the average performance instead of the worst-cases (see Figure 9), which are the most relevant cases in a real-time system.

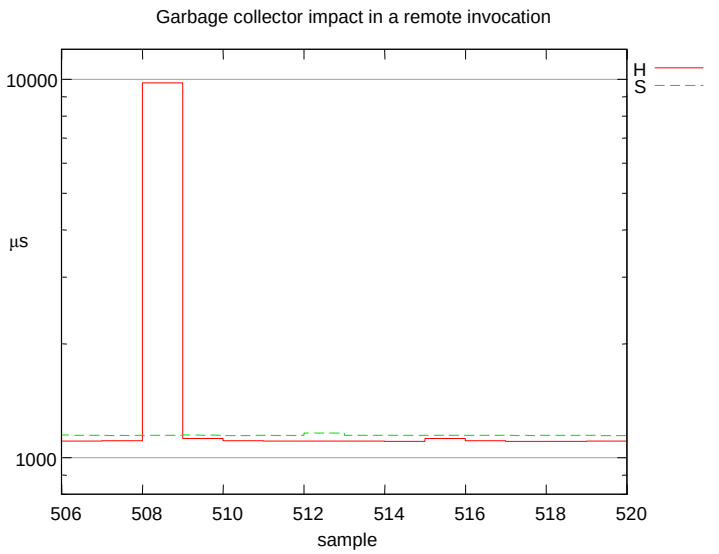

Fig. 9. The worst case of a garbage collector in a remote invocation (976 MHz)

7.3 Consumption of memory in remote invocations

Eventually, a third analysis which is relevant for both garbage collectors and region-based models is the amount of memory consumed in a remote invocation. For the first technique (garbage collector), it allows calculating the maximum allocation rates of an application; and in the case of a region-based memory management, it helps the programmer to dimension the size of the regions which are stored in the `LTMemoryAreaPool`. Our results (Figure 11) show that a remote invocation consumes a tremendous amount of memory. Even in the least demanding cases (with empty remote invocation) the dynamic allocation of memory reaches 5 kilobytes.

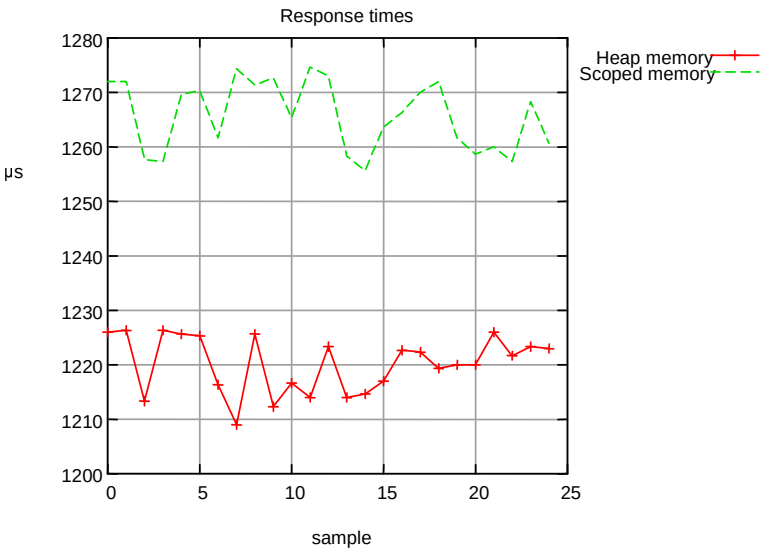

Fig. 10. Garbage and region-based collection best responses (976 MHz)

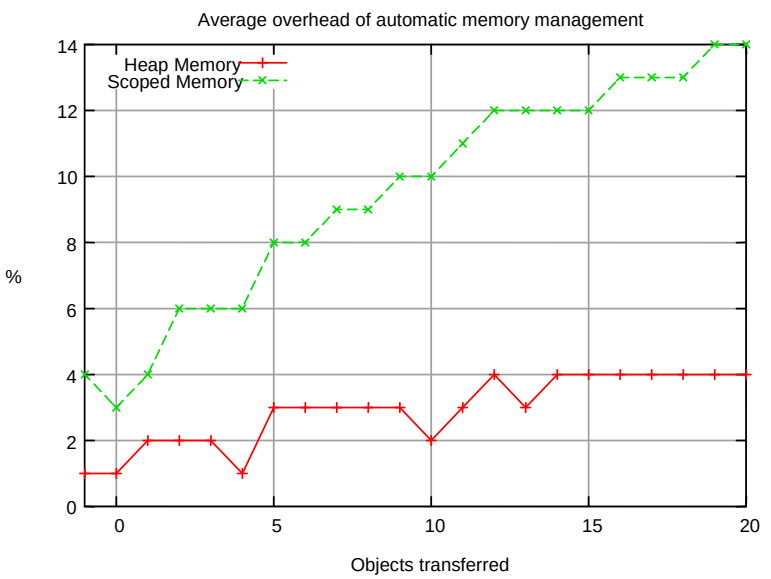

Fig. 11. Overheads of the automatic memory-management algorithms

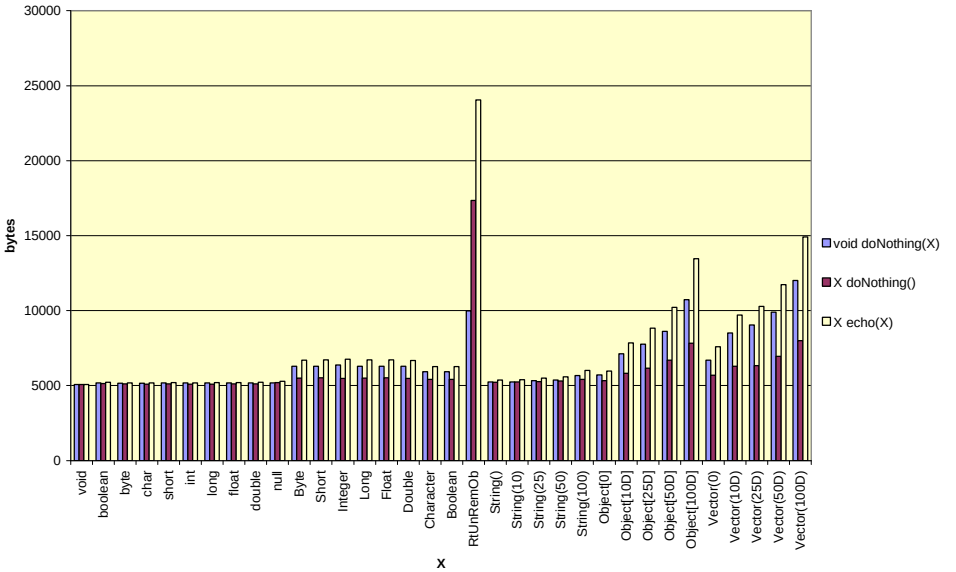

Fig. 11. Dynamic allocation of memory in server

8. CONCLUSIONS

There is still a long way to go to obtain valuable architectures and solutions which make easier the development of distributed real-time Java systems. This article provides an insight into the high-predictable automatic memory-management for remote invocations, exploring solutions which allow a programmer to skip over the penalties of garbage collection in a networked application.

The proposed technique, named NhRo, is able to avoid the garbage collector penalties, producing more predictable remote invocations. However, this is not for free, it requires a compromise from both remote object practitioners and middleware designers. The first ones will have to code their applications according to more constrained rules, and the second ones, will have to include additional facilities within their current middleware infrastructures. This is the shared price that both will have to pay for having more predictable infrastructures.

Our ongoing efforts are evaluating the integration of advanced garbage collection techniques, which relies on the use of advanced region-models, such as `AGCMemory` [Basanta-Val et al. 2005b] within networked infrastructures. Besides, it is also being considered the integration of algorithms for dynamic composition of real-time systems [Estevez-Ayres et al. 2007] to select which memory-area pool is going to be used in the remote invocation.

REFERENCES

ANDREAE, C., COADY, Y., GIBBS, C., NOBLE, J., VITEK, J., AND ZHAO, T. 2007. Scoped types and aspects for real-time Java memory management. *Real-Time Syst.* 37, 1 (Oct. 2007), 1-44.

AL-JAROODI, J. AND MOHAMED, N. 2005. Object-Reuse for More Predictable Real-Time Java Behavior. In *Proceedings of the Eighth IEEE international Symposium on Object-Oriented Real-Time Distributed Computing* (May 18 - 20, 2005). ISORC. IEEE Computer Society, Washington, DC, 398-401.

ANDERSON, J. S. AND JENSEN, E. D. 2006. Distributed real-time specification for Java: a status report (digest). In *Proceedings of the 4th international Workshop on Java Technologies For Real-Time and Embedded Systems* (Paris, France, October 11 - 13, 2006). JTRES '06, vol. 177. ACM, New York, NY, 3-9.

BACON, D. F., CHENG, P., AND RAJAN, V. T. 2003. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for Java. *SIGPLAN Not.* 38, 7 (Jul. 2003), 81-92.

BASANTA-VAL, P., GARCÍA-VALLS, M., AND ESTÉVEZ-AYRES, I. 2004. No Heap Remote Objects: Leaving Out Garbage Collection at the Server Side. *Second International Workshop on Java Technologies for Real-Time and Embedded Systems* JTRES'04. 25-29 October 2004, Larnaca, Cyprus. ISBN: 3-540-23664-3. ISSN: 0302-9743

BASANTA-VAL, P., GARCIA-VALLS, M., AND ESTEVEZ-AYRES, I. 2005a. Towards the Integration of Scoped Memory in Distributed Real-Time Java. In *Proceedings of the Eighth IEEE international Symposium on Object-Oriented Real-Time Distributed Computing* (May 18 - 20, 2005). ISORC. IEEE Computer Society, Washington, DC, 382-389.

BASANTA-VAL, P., GARCÍA-VALLS, M., AND ESTÉVEZ-AYRES, I. 2005b. AGCMemory: a new real-time Java region type for automatic floating garbage recycling. *SIGBED Rev.* 2, 3 (Jul. 2005), 7-12.

BASANTA-VAL, P., GARCÍA-VALLS, M., ESTEVEZ-AYRES, I., AND DELGADO-KLOOS, C. 2006. Extended portal: violating the assignment rule and enforcing the single parent rule. In *Proceedings of the 4th international Workshop on Java Technologies For Real-Time and Embedded Systems* (Paris, France, October 11 - 13, 2006). JTRES '06, vol. 177. ACM, New York, NY, 30-37.

BASANTA-VAL, P., ALMEIDA, L., GARCIA-VALLS, M., AND ESTEVEZ-AYRES, I. 2007. Towards a Synchronous Scheduling Service on Top of a Unicast Distributed Real-Time Java. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium* (April 03 - 06, 2007). RTAS. IEEE Computer Society, Washington, DC, 123-132.

BOLLELLA, G., CANHAM, T., CARSON, V., CHAMPLIN, V., DVORAK, D., GIOVANNONI, B., INDICTOR, M., MEYER, K., MURRAY, A., AND REINHOLTZ, K. 2003. Programming with non-heap memory in the real time specification for Java. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, 361-369.

BOLLELLA, G. AND GOSLING, J. 2000. The Real-Time Specification for Java. *Computer* 33, 6 (Jun. 2000), 47-54.

BORG A. AND WELLINGS A. 2003. A Real-Time RMI Framework for the RTSJ, *Proc. of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, Portugal, pp 238-248

CORSARO, A. 2004 *Techniques and Patterns for Safe and Efficient Real-Time Middleware*. Doctoral Thesis. UMI Order Number: AAI3162124, Washington University.

ESTEVEZ-AYRES, I., ALMEIDA, L., GARCIA-VALLS, M., AND BASANTA-VAL, P. 2007. An Architecture to Support Dynamic Service Composition in Distributed Real-Time Systems. In *Proceedings of the 10th IEEE international Symposium on Object and Component-Oriented Real-Time Distributed Computing* (May 07 - 09, 2007). ISORC. IEEE Computer Society, Washington, DC, 249-256.

FAY-WOLFE, V., DIPIPPO, L. C., COPPER, G., JOHNSTON, R., KORTMANN, P., AND THURAISINGHAM, B. 2000. Real-Time CORBA. *IEEE Trans. Parallel Distrib. Syst.* 11, 10 (Oct. 2000), 1073-1089.

HILDERINK, G. H., BAKKERS, A. W., AND BROENINK, J. F. 2000. A Distributed Real-Time Java System Based on CSP. In *Proceedings of the Third IEEE international Symposium on Object-Oriented Real-Time Distributed Computing* (March 15 - 17, 2000). ISORC. IEEE Computer Society, Washington, DC, 400-408.

HIGUERA-TOLEDANO, M. T. 2006. Hardware support for detecting illegal references in a multi-application real-time Java environment. *Trans. on Embedded Computing Sys.* 5, 4 (Nov. 2006), 753-772

JSR-128. 2009. JSR 282: RTSJ version 1.1. Available online at http://jcp.org/en/jsr/detail?id=282

PLŠEK, A., LOIRET, F., MERLE, P., AND SEINTURIER, L. 2008. A component framework for java-based real-time embedded systems. In *Proceedings of the 9th ACM/IFIP/USENIX international Conference on Middleware* (Leuven, Belgium, December 01 - 05, 2008). V. Issarny and R. Schantz, Eds. Middleware Conference. Springer-Verlag New York, New York, NY, 124-143

RAMAN, K., ZHANG, Y., PANAHI, M., COLMENARES, J. A., AND KLEFSTAD, R. 2005. Patterns and Tools for Achieving Predictability and Performance with Real-Time Java. In *Proceedings of the 11th IEEE international Conference on Embedded and Real-Time Computing Systems and Applications* (August 17 - 19, 2005). RTCSA. IEEE Computer Society, Washington, DC, 247-253.

SIEBERT, F. 2008. Limits of parallel marking garbage collection. In *Proceedings of the 7th international Symposium on Memory Management* (Tucson, AZ, USA, June 07 - 08, 2008). ISMM '08. ACM, New York, NY, 21-29.

SILVA JR., E. T., FREITAS, E. P., WAGNER, F. R., CARVALHO, F. C., AND PEREIRA, C. E. 2006. Java Framework for Distributed Real-Time Embedded Systems. In *Proceedings of the Ninth IEEE international Symposium on Object and Component-Oriented Real-Time Distributed Computing* (April 24 - 26, 2006). ISORC. IEEE Computer Society, Washington, DC, 85-92.

SUN. 2004. The RMI specification. Available online at http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html

TEJERA, D., ALONSO, A., AND DE MIGUEL, M. A. 2007. RMI-HRT: remote method invocation - hard real time. In *Proceedings of the 5th international Workshop on Java Technologies For Real-Time and Embedded Systems* (Vienna, Austria, September 26 - 28, 2007). JTRES '07, vol. 319. ACM, New York, NY, 113-120.

TIMESYS. 2008. JTIME virtual machine. Available online at http://www.timesys.com

WOLLRATH, A., RIGGS, R., AND WALDO, J. 1996. A distributed object model for the java™ system. In *Proceedings of the 2nd Conference on USENIX Conference on Object-Oriented Technologies (Coots) - Volume 2* (Toronto, Ontario, Canada, June 17 - 21, 1996). Conference on Object-Oriented Technology and Systems. USENIX Association, Berkeley CA, 17-17