Extending Distributed Real-Time Java with Remote Memory Areas

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

Abstract—Current approaches towards distributed real-time Java are mainly based on the idea of having user-defined remote objects, allocated in servers that may be invoked from clients. This article extends this support included in real-time Java with an extension called Remote Memory Areas (RMAs). RMAs offer a method that allows running user-defined code in a generic server that may be reused by several applications (i.e., it does not require defining one type of remote sever per application). The paper describes the abstraction, which is backward compatible with main approaches in distributed real-time Java, and provides empirical evidence on its performance on a use-case application.

Keywords— DRTSJ, real-time Java, real-time middleware, remote memory areas

I. INTRODUCTION

Industrial applications may benefit from high-level programming languages, e.g. real-time Java, that reduce their development maintenance costs [1][2]. Java offers key features such as garbage collection, a portable byte-code model, and a well-defined [3] programming language that may speed up the next generation applications development. However, the same mechanisms are also source of indeterminism that may introduce arbitrary delays in the industrial applications, requiring being readapted to the special and specific nature of real-time applications.

Current efforts in real-time Java -mainly focused on centralized, safety critical, and distributed specifications-progress at different rates [4]. The leading effort is RTSJ [5] (The Real-Time Specification for Java), which offers a stable API and a number of implementations (e.g. Jamaica [6], Oracle JRTS [7], and IBM RT-WebSphere [8]) ready to be used in applications. The second main effort is SCJS [9, 10] (Safety-Critical Specification for Java) and proposes an extension to deal with the specificities of high-integrity applications and it is producing implementations [11] currently. Lastly, there is a distributed version for RTSJ namely DRTSJ (The Distributed Real-Time Specification for Java) [12] that it is still progressing slower, lacking implementations and reference models [13].

Regarding DRTSJ, most approaches [13] are based on Java’s RMI [14] (Remote Method Invocation) that transports information from one RTSJ-enabled node into another via remote invocations. The basics of the remote invocation model proposed by Java’s RMI are simple ([14]). First, the developer defines a user-defined (a remote interface) contract for the remote interface which is supported by the server after. Typically each application requires one type of server. After server instantiation, the client uses stubs to serialize, send data to the server, and receive information from the remote invocation.

Current support does not include a generic set of mechanisms that allows remote execution of a piece of code in generic servers that could be reused in other applications. In this context, the Remote Memory Area (RMAs) extension provides a mechanism to execute a piece of code in a remote object allocated in a server. RMAs extend an existing mechanism included in RTSJ and named memory-memories, to transform them into remote objects that may be used to building distributed application. By using RMAs, the application may select the remote node and scheduling properties of the remote object sent to the remote node. The RMA approach is a general approach which works on a scalable number of nodes, and for more than one application.

The main RT-RMI frameworks (UPM’s RT-RMI [15], York’s RT-RMI [15], and UC3M’s DREQUIEM [16]) may benefit from having this type extension included as part of their APIs. Currently, none of these frameworks offers a similar support in their cores to run arbitrary code in remote servers. In addition, leading effort DRTSJ [13] may also offer flexible support for distributable threads that it is compatible with current real-time remote invocation model.

The rest of the article develops RMAs. Section II introduces the RMAs. Section III exemplifies a type of industrial application that may be developed by using the extension. Section IV offers performance on an industrial application described as use-case. Section V connects this work to other different approaches in centralized and distributed real-time Java. Lastly, Section VI concludes and relates ongoing work.

II. REMOTE MEMORY AREAS

Current memory model of RTSJ [5] (Listing 1) offers an enter method as a part of an entity called memory area (javax.realtime.MemoryArea) that allows to change the relationship with the underlying virtual machine. An invocation on the enter method changes the default allocation context of the remote object, executes the run method of the Runnable object and restore the previous status (i.e. default allocation context).

RMAs extend this simple model into a distributed ecosystem. The approach to transform the former memory area model into a distributed model is based on transforming a memory area into a remote object allocated
in a remote server. This way, the client interacts with the remote memory area by using its corresponding local stub (Figure 1).

RMAs also add the possibility of defining the parameters of the remote invocation at the server. This is allowed by allowing the execution of schedulable objects that change their relationship with the remote scheduler at runtime.

Listing 1. API of a Memory Area in RTSJ

```
01: public abstract class MemoryArea{
02:   MemoryArea(long size)
03:   void enter(Runnable logic)
04:   void executeInArea(Runnable logic)
05:   Object newInstance();
...  
06: }
```

Figure 1. Local memory area vs. Remote Memory Area

A. A Remote Interface for Remote Memory Areas

Transforming the interface of a memory area into a remote object involves dealing with the definition of a proper interface for RMAs compatible with remote invocations. Ideally, the interface should be as similar to the API of the memory area as possible. However, it is not as simple as it seems a priori due to a number of issues related to RMI and real-time distribution.

Before entering into these issues, Listing 2 shows the API of a RMA. This API results from dealing with the specific issues that are explored in the rest of the subsection.

Listing 2. Remote Memory Area interface

```
01: RemoteMemoryArea extends java.rmi.Remote{
02:   implements Runnable, Serializable{
03:     Scheduler enterScheduled(Schedulable s)
04:     void enterAsyncScheduled(Schedulable s)
05:     throws RemoteException;
06: }
```

The first difference is the type of object used in the invocation. In RTSJ, the primary objects are Runnable objects that have a run method that may be invoked in a remote server. In RMAs, the main object is a Schedulable object that inherits the behavior of a schedulable object. All schedulable objects are runnable objects (and they also have a run method).

Another difference has to do with the by-reference and by-value semantics of RMI. In RMI all references to objects in the method are sent by value. The only object that may be returned at runtime is the result object if any.

As a result, the API provided by RTSJ for entering a memory area cannot be used directly into RMI. In RTSJ, the enter method of a memory area has a void enter(Runnable r) signature and all modifications on r remain accessible after the invocation. However, in RMI, the r object is lost after the remote invocation and results stored in r are not properly returned to the client. A simple solution to this issue is to define the remote version of the remote method returning a non void object (Line 02 in Listing 1).

The third change has to do with whether the client waits or not for a response from the server until the remote invocation ends. In proposed API, the developer has two options: the first is to wait for a response from the server (via enterScheduled in Listing 2) and the second option is to use an asynchronous invocation where the client does not wait for a response (via enterAsyncScheduled in Listing 2). The advantage of asynchronous invocations is that they give better performance and lower response-times (see [17] for empirical evidences).

The fourth issue, also shown in Listing 2, is that all remote methods have an option of raising a remote exception. So that, the client thread has to define how handle this exception (e.g. by using a try-catch statement). This try-catch statement is not required in a plain remote invocation to an enter method of a local memory area.

The fourth and last issue is that the objects that have to be sent in enterScheduled and enterAsyncScheduled methods have to be Serializable objects. In RMI, not all objects may be sent as a parameter of a remote invocation. Only those objects that may be serialized may be sent and received in a remote server. Therefore, all runnable objects sent should implement the Serializable interface (see example included in Listing 3).

Listing 3. An example of a Serializable and Runnable class. The samples array is transferred atomically as the remote object is sent as a part of a remote invocation.

```
01: public class NormalizerFilter
02:   implements Runnable, Serializable{
03:     long samples[1024]; //Input and Output
04:     public void run(){
05:       for (int i=0; i<1024; i++){
06:         samples[i]=normalize(samples, i);
07:       }
08:     }
```

B. Types of Remote Memory Areas

Once an interface has been proposed, the next step is to define general and useful behaviors for the remote memory areas according to RTSJ and DRTSJ. Each behavior represents a type of remote object that may be invoked at the server from different clients.

There are two main behaviors of real-time Java that may be replicated in a distributed scenario:

heap: The heap behavior matches the HeapMemory of RTSJ. This type of memory area is interesting because it allows allocation of remote objects into heap memory. It
suffers from the advantages (flexible programming model) and drawbacks (latency and overhead of the garbage collector) typically associated to the Java’s heap.

**No-heap** The second behavior of interest is the non-heap behavior of RTSJ. This behavior allows an application to avoid the garbage collector penalty. However, it also means removing objects that are created for the remote invocation. Providing a solution to this problem means introducing limitations to the programming model of the application. Our particular choice was to use the no-heap remote object paradigm, previously developed by the authors in [18][19]. In this paradigm a memory area pool is in charge of removing temporal objects allocated at the server.

Listing 4. Partial implementation of a RemoteHeapMemory class

```java
00: class RemoteHeapMemory implements RemoteMemoryArea, Schedulable{
01:     public RemoteHeapMemory() {
02:     }
03:     public Schedulable enterSchedulable(Schedulable ss) {
04:         Schedulable th = set_thread_parameters(ss);
05:         ss.run();
06:         restore_thread_parameters(th);
07:         return ss;
08:     }
09:     public void enterAsyncSchedulable(Schedulable ss) {
10:         RealtimeThread th = threadpool.getThread();
11:         th.runAsync(ss);
12:         return;
13:     }
14: }
```

Listing 5. Partial implementation of a RemoteLTMemoryAreaPool instance. All objects created during the remote invocation are allocated in a private block of memory released after the remote invocation (LTMemory).

Listing 4 shows the code for the server side of a remote heap memory instance. The server provides an implementation for enterSchedulable and enterAsyncSchedulable remote methods that may be reused from different clients. The remote heap memory allows attaching Schedulable information to the remote object instance. In this particular implementation, this schedulable information is used if the client does not provide the server with schedulable parameters (i.e. release, scheduling, scheduler, memory and/or processing group parameters). The implementation of the asynchronous invocation is implemented using a thread taken from a thread pool (Listing 04:11).

The implementation of a no-heap version requires using special mechanisms that allow the remote object is not destroyed before it ends its life (e.g. pinned-scowps) and memory area pools that provides fresh blocks of memory that may be reused in each invocation (see Listing 5). The memory required for the remote invocation is stored in an LTMemoryAreaPool instance (see [18][19]) that provides memory for the run method of the Schedulable object.

C. Serialization of Schedulable Information

In RMAs, developers send non-functional information to the server (in its simple form the priority used in the server for the remote invocation). A simple way to transfer this information is to attach this information to the remote invocation by sending a schedulable object in each remote invocation.

However, new changes are required because the information related to the priority of a real-time thread is not serializable in RTSJ ([20][21]). A backward compatible solution is to design a wrap class that carries out this action (this class may require a slight adaptation for each type of application). Another solution is to change their definition in RTSJ, transforming the schedulable parameters into Serializable classes.

Listing 6. Partial overview of the HookSchedulable class.

```java
00: class HookSchedulable implements Schedulable, Serializable{
01:     private void writeObject (...) throws {
02:         //serialization code !!!
03:     }
04:     public MemoryParameters getMemoryParameters();
05:     public void setMemoryParameters(MemoryParameters);
06: }
07: }
```

The particular choice was to define a new wrapper class: HookSchedulable (Listing 6). This class implements the Schedulable and the Serializable interfaces. The Schedulable interface forces the method to include a run method and methods for reading scheduling, release processing group, memory and scheduler parameters. The Serializable interface forces the application to implement code for serializing these parameters.

Some properties like the application deadline and period do not require extra treatments from the application; they may be transformed into Serializable objects easily. Others like the offset, cost, and priority may require taking into account additional decisions about their particular nature. The offset may be serialized and transmitted if the two nodes are synchronized (globally by
using any clock synchronization protocol or indirectly by using a modified version of JRMP). Likewise the cost, the priority, and the memory parameters may be transmitted if the parameters are specific for the remote node.

Another option is to use some type of global parameterization for any of these parameters. With the RT-CORBA model as reference model in mind, the priority may be transformed into a global priority honored in each different node (see [20] for detailed discussion). The same rationale is valid for the remaining parameters (cost and memory parameters).

Lastly, the transmission of a scheduler object is rather complex because RTSJ does not define a mechanism for identifying different schedulers. Two options are feasible: the first is to transform the scheduler into another remote object and the second is to perform a user-specific serialization. The first option allows automatic object deployment at the cost of high latencies. The second option requires application assisted and customized efforts for each application (but it is more efficient too).

III. DEVELOPER PERSPECTIVE

Previous section addressed the issue of producing interfaces and extensions according to the constraints introduced by real-time Java. This section addresses the developer’s perspective. To this end, it includes a simple distributed application for control that reads data, processes these data, and stores the data into and output (see Figure 6). Communication with remote Java nodes is done via remote memory areas.

![Figure 2. Distributed application running on three RMAs from a single client](image)

The industrial application (Listing 7) is declared in a Schedulable object and consists of three steps. Each step refers to an invocation to the run method of the object. The first time the code is executed, it initializes the array; the second time it performs a remote operation; and the third time stores a trace with the result of the last element.

To select the right section to execute is controlled via an internal attribute (Listing 7:02). To run the general code of Listing 8 in three different machines (see Figure 2), the example requires mechanisms to lookup target nodes and extra code to send the information to the corresponding node. This action is performed in statements 06-09 in Listing 8. Each line looks up for a remote node in the naming service of RMI (e.g. rmi registry). After looking for the right node, the example shows how to invoke the nodes in order to carry out the three actions. All invocations use the same Runnable object (tsc) that traverses all nodes with has a user-globally defined priority of 8.

```
00: public class ThreeStepsApp
01: extends Schedulable, Serializable{
02:     int scount=0;
03:     ...
04:     boolean local=true;
05:     long[] array= null;
06:     ThreeStepsApp(...){
07:     array=new long[1024];
08:     }
09:     public void run(){
10:     scount ++;
11:     switch(scount)
12:     {
13:         case 1: //First execution
14:             for (int i=0; i<1024; i++)
15:                 ( array[i]=input();
16:             }
17:         }
18:     break;
19:         case 2: //Second execution
20:             for (int i=0; i<1024; i++)
21:                 array[i]=process(array[i]);
22:     }
23:         case 3:            output(array[1023]);
24:     break;
25:         }
26:     private trace(long){ … }
27:     }
28: }
```

Listing 7. Application code packed in a unique Schedulable object

```
01: new RealtimeThread(){
02:     public void run(){
03:         setSchedulingParameters
04:         ((PriorityParameters(32)));
05:         ThreeStepsApp tsc= new ThreeStepsApp();
06:         RemoteMemoryArea src=lookupSrc();
07:         RemoteMemoryArea proc=lookupDest();
08:         RemoteMemoryArea dest=lookupRemoteHeap();
09:         tsc.setSchedulingParameters
10:         ((PriorityParameters(8)));
11:         do{
12:             tsc=(TestSchedulable3)src.enterSchedulable(tsc);
13:             tsc=(TestSchedulable3)proc.enterSchedulable(tsc);
14:             tsc=(TestSchedulable3)dest.enterSchedulable(tsc);
15:             waitForTheNextPeriod();
16:         while(true).
17:     }
```

Listing 8. Running the example within a client thread that runs the example

A final note on Listing 7: applications may change the code running in the server without having to change the particular implementation of the remote memory area allocated in the server. This is because the server-side implementation invokes the run method of the schedulable object that it is sent to the server. This kind of behavior characterizes RMAs.

IV. EMPIRICAL RESULTS

The proposed extensions were evaluated on Oracle JRTS [7]. The RMI stack of JRTS was partially modified
(with pre-allocation of connections and the priority of the handler thread at the server) to perform an implementation valid for RT-RMI (see Figure 3). All results in this section refer to an infrastructure with two virtual machines running on an 800 MHz processor interconnected via a 100 Mbps Ethernet. The underlying kernel is a real-time Linux 3.0 rt-patched kernel on a Debian 6.0 distribution.

![Figure 3. Application stack: operating system, communication middleware and application](image)

### A. Distributed Control Application Performance

For the evaluation a benchmark was built. It is based on the application described in Figure 2 and includes parametric control on the number of elements that may be sent packed in a remote invocation. This number moves from $10^2$, $10^3$, $10^4$ and $10^5$. The end-to-end response time has strong dependency on the amount of data. This dependency is shown in Figure 4.

![Figure 4. End-to-end response of the distributed control application (800 Mhz-100 Mbits)](image)

Results show (see Figure 4) the total end-to-end cost of the application changing from 16 ms (with an array of 102 elements to 100 ms for 105 elements. The overhead, when this solution is compared to traditional RT-RMI is small because serialization introduces high penalties in the end-to-end path (Figure 5).

![Figure 5. Overhead due to the abstraction(800 Mhz-100 Mbits)](image)

V. RELATED WORK

The main approaches towards distributed real-time Java are mainly based on user-defined remote objects [13]. This is the case of RTZen [22] and DRTSJ [12] approaches. This same rationale is true for RT-RMI UPM [15], RT-RMI York [23], and DREQUIEMI [16][24] frameworks since they are based on RMI.

Only in specific works, researchers consider other models. One of them is the SSS (Synchronous Scheduling Service) of DREQUIEMI ([25][26]) described in the scope of time-triggered distributed communications. This service provides new classes (namely master and client) that offer control access to a time-triggered distribution model. The second is an effort focused on producing a distributed event model for distributed real-time Java [27], also defined in the DREQUIEMI context. This second model provides a remote interface that enables to use a publish-subscribe model in distributed real-time Java applications. In both cases the remote API does not need to be redefined for each application.

The possibility of modifying or adding additional functionality associated to the `enter` method has been widely used in real-time Java (see [28][29][30]). In [31] the authors proposed extended portals with an `enter` method as a mechanism to provide a way to access forbidden objects with hard semantics (extending weak semantics proposed by previous work [32]). Also, the authors produced new types of memory areas (e.g. AGCMemory [33][34]) that particularize the `enter` method of a particular memory area.

Some researchers [35][36][37] also analyze the use of DDS and RTSJ together as backbone for distributed applications. Their programming model may benefit from the programming model proposed in this article by extending their interfaces with generic mechanisms that allow remote execution of distributed applications.

VI. CONCLUSIONS AND FUTURE WORK

This paper extended current support for distributed real-time Java with a mechanism to perform remote invocations via remote memory areas. The programming model of the remote memory areas allows user application code without introducing changes in the remote objects allocated in remote nodes. The empirical evidence offered clues on the performance this extension may provide on a simple control application. Our future and ongoing work is focused on using the proposed RMA extension as a building block for a reconfiguration service for distributed real-time Java [24]. In addition, we also plan to address different implications on security concerns stemmed from the RMA model and its integration within industrial architectures ([38][39][40]).

ACKNOWLEDGEMENTS

This research was supported by the European Commission (iLAND ARTEMIS-JU Call 1) and by the national project REM4VSS (TIN-2011-28339).
REFERENCES


