A Dual Programming Model for Distributed Real-Time Java

P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres

Abstract—More-and-more, the use of Java as a programming language for real-time applications is becoming of interest in industrial middleware. This is mainly because it offers advantages for the programmer as reduced deployment times, increased portability, and a number of APIs that may be integrated in large distributed applications. This article contributes a dual communication model for distributed real-time Java applications. Current efforts in distributed real-time Java (e.g., DRTSJ: The Distributed Real Time Specification for Java) are mainly focused on remote invocations and set aside other valuable approaches such as distributed events. The proposed model offers two choices for developing distributed real-time Java applications (one based on remote invocations, and another on distributed events) which may be used to develop applications. Both models include an additional support for asynchronism in communications, a feature that may speed up their communication performance. The article includes a description for the two models, the changes that are required in the current API to accommodate them, and an empirical evaluation of their performance on a reference implementation.

Index Terms—Real-time Java Middleware, Middleware for Factory Communications, RTSJ, DRTSJ

I. INTRODUCTION

By looking the trend in embedded and real-time systems, one may observe there is a continuous and unstoppable race towards having infrastructures with abstractions that reduce system development costs. Real-time systems, which in the past were both monolithic and isolated modules, have become cyber-physical infrastructures [1] interconnected via networks that participate in distributed and coordinated actions. From the software engineering perspective, that increase in complexity may be mitigated via (i) new design methodologies, including novel development models such as MDA (The Model Driven Architecture) [2], and (ii) via new programming languages [3] like RTSJ [4] (The Real-time Specification for Java) and DRTSJ [5] (The Distributed Real-Time Specification for Java).

The idea of using Java as a real-time technology [6] sounds very appealing because Java offers a portable bytecode model, which may reduce both deployment and maintenance costs remarkably in heterogeneous systems [7] and provides applications with independence from the underlying operating system. Furthermore, its programming community and an important number of libraries become synergic factors which should not be disdained in the evaluation of its potentiality. However, its generalized deployment is tough because many built-in mechanisms of Java collide head-on with constraints imposed by real-time systems [8] [9]. On the one hand, Java tries to abstract infrastructure as much as possible, what gets programming complexity reduced. However, on the other hand, many of its primary mechanisms (e.g., garbage collection [10], bytecode interpretation, and dynamic class downloading) involved in complexity reduction become indeterminism sources in real-time systems.

To date [11], real-time Java has achieved mature solutions for centralized systems but it lacks a similar support for distributed systems. For centralized systems it reached an interesting equilibrium point: RTSJ (The Real-Time Specification for Java) which is supported by implementations (e.g., Oracle’s JTRS [12]). However, for distributed systems, such support is still under development. For distributed real-time Java, there are two different approaches: one more conservative betting on the use of existing technologies (the RT-CORBA [13] model with RTSJ) and the aforementioned DRTSJ [5] which relies on Java’s RMI (Remote Method Invocation), a natural Java candidate for distributed systems.

One key issue in distributed real-time Java is the type of interface offered to the programmer [11] [14] when designing applications. Both, RT-CORBA and the real-time RMI based approaches ([15-19]), run on remote invocations and may be used in control-flow applications. However, they do not consider data-flow models based on the publisher-subscriber, which are included in non-Java centric initiatives like DDS (The Data Distribution Service) [20] and partially addressed in research projects like iLAND [21-23], and ServiceDDS [24].

This article extends the communication model to include a more general programming model; i.e., able to transfer user-data to the server by using distributed events and/or remote invocations on remote objects. By doing so, the programming model in DRTSJ becomes dual: it may use a control-flow approach based on remote invocations on remote objects; and it may support a data-flow distributed event model with publisher-subscriber interactions.

A. Impact on the State-Of-The-Art

In its initial design, DRTSJ [5] considered an event model similar to the event model described in this paper for notifying about failures to the other end of the communication. However, it does not allow transferring user-data from client to the server; its main goal was to notify about problems in distributable threads [5]. The work carried in the article improves the previous model with user-data, and several
asynchronous models. It also offers results on performance which may help the DRTESJ to evaluate if this support should or not be included in the next release of the specification.

The same rationale is true for the work carried out by other real-time RMI approaches. The support described by York University [17, 19] for remote invocations is mainly synchronous, so that the event model described in this article is an interesting improvement for its programming models. The work carried out by UPM [18], mainly based on remote invocations, could be extended by using the distributed event model defined in this article. Lastly, the work carried in DREQUIEMI [25][26] is also based on remote invocations and has only partially considered the publisher-subscriber model as a support for its SSS (Synchronous Scheduling Service) [27]. The work on the article generalizes and extends the previous results on the SSS.

Even RTZen [13], the main effort defined in RT-CORBA towards distributed real-time Java, may benefit from the dual programming models defined in this article. Its current programming model is based on remote invocations and it is silent on support distributed events. An approach to include this support is to use the model proposed in the article.

Lastly, the distributed event model described in this paper may be considered a successor of a previous mechanism described for centralized systems, namely the asynchronous event handler of RTSJ. The event based model was improved in [28] but the authors are silent on the impact of this mechanism into distributed systems. The results and API described in this article provide a way to extend the model they proposed to a networked environment.

The rest of this paper shows how to integrate both types of communications in DREQUIEMI. The integration includes the definition of the communication models, changes in APIs, and performance results. Section II fits both models in the DREQUIEMI framework. Section III formalizes the dual communication model. Section IV continues with API details for the models. Section V evaluates the performance for the models by using benchmarks. Finally, Section VI draws conclusions and relates our ongoing work.

II. THE ARCHITECTURE OF DREQUIEMI

The architecture (Fig. 1) is based on the model defined by Schantz et al. in [29] which considers a multi-domain layered architecture which consists of seven levels. DREQUIEMI particularizes the same model to a real-time Java platform with five layers that consider deployment of modules in remote nodes. The set of common services introduced by DREQUIEMI is also more specific (e.g., it includes a distributed garbage collector) than the set defined by Schantz. Therefore, the architecture proposed for DREQUIEMI could be considered as a particularization of the model proposed by Schantz to a case where all nodes are real-time virtual machines running the same distribution middleware.

A. Layers

- Resource layer. This layer defines the foundations of the model defining the list of resources involved. The model takes from real-time Java two key resources: memory and processor; and, from RMI, the network resource.

- Infrastructure layer. Memory, processor and network are typically accessed through interfaces (via an RT-OS or an RT-JVM) shaping an infrastructure layer. In DREQUIEMI, the access to the infrastructure is given through a set of primitives which are used by programmers to access to low-level resources.

- Distribution layer. On top of this layer, the programmer may use a set of common structures useful to build distributed applications. There is one hierarchy corresponding to each key resource, namely: ConnectionPool, MemoryAreaPool, and ThreadPool.

- Common service layer. This layer includes four services: a Stub/Skeleton service, a DGC (Distributed Garbage Collection) service, a Naming service, and eventually a Synch/Event service. The first allows carrying out a remote invocation while maintaining a certain degree of predictability. The DGC service eliminates unreferenced remote objects in a predictable way; that is, introducing a limited interference on other tasks of the system. The naming service offers a local white page service to the programmers, enabling the use of user-friendly names for remote objects. The synch/event service is a novel service (not included currently in RMI); it allows the establishment of a common temporal axis for a distributed application; it is based on the FIT [30] paradigm.

- Application. Lastly, the most specific parts of the application are found at the uppermost layer drawn as modules.

Fig. 1. The Architecture of DREQUIEMI

B. Primitives

In the architecture, programmers work at two levels: directly on the infrastructure middleware (controlling resources and simple communication facilities like sockets), or running on top of common services. In both cases, the communications with
the subsystem are carried out through primitives. Table I shows them grouped by their nature.

The centralized primitives define a set of mechanisms to control the three underlying resources (i.e., memory, CPU and network). Through its use, it is possible to allocate and deallocate memory, concurrent entities, and network communications. Moreover, these last two resources (concurrent entities and the network) are modelled in more detail with extra support to control the concurrent access to resources, the thread priority, and the reception and sending of messages through established connections.

<table>
<thead>
<tr>
<th>TABLE I: Primitives included in DREQUIEMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Infrastructure middleware</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Manager</td>
</tr>
<tr>
<td>Distribution middleware</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Manager</td>
</tr>
</tbody>
</table>

There are other primitives that grant access to common services and use management algorithms. The subscription and unsubscription to the stub/skeleton service is done explicitly with the creation of remote objects and stubs and defines one primitive to carry out remote invocations. The naming service offers three primitives: one to register and deregister clients from its internal tables, and another to look them up. The DGC service (which is in charge of carrying out distributed garbage collection) requires notifications on creation and destruction of remote references hosted in remote nodes. The fourth service supports a real-time event model which supports publish/subscriber communications to send messages, with a trigger primitive to activate remote nodes. As the centralized manager does, the distribution layer supports also a global manager, which may be changed through a set primitive.

From the programming perspective, the most import services are the stub/skeleton service which offers remote invocations (and communicates by using invoke) and the event model (which offers the trigger primitive). These two primitives are the core of the two programming models: remote invocations and distributed events; both are addressed in the next section in detail.

III. TWO COMMUNICATION MODELS FOR DISTRIBUTED REAL-TIME JAVA

This section describes the two models designed to implement communications in distributed real-time Java. One of them is the remote invocation; the other is the distributed event model. In addition, both communication models support asynchronism to decouple communications.

A. Control Flow on Remote Objects

The control flow model considers communications between client and server as a local invocation to a stub which translates the communication to the server transparently (see Fig. 2). The remote invocation is carried out following a sequence of seven steps \(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7\). The first step in the client (1) serializes the parameters to send them to the server, (2) which receives them (3), up-calls the remote object (4), and sends back a response to the client (5). Lastly, after receiving the parameters, the client deserializes the parameters (6) and the remote invocation finishes (7).

In this model, the invocation rate to the remote object (I.R.) is defined as follows:

\[
I.R_{sym} = \frac{1}{C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7} \quad (1)
\]

(Note: \(C_x\) refers to the cost of the \(x^{th}\) step of the invocation)

1) Client and Server-Side Asynchronism

In DREQUIEMI, if a remote invocation does not return any application-data generated from the remote invocation to the client, two new approximations are feasible [31], namely client and server-side asynchronous invocations.
Client-side asynchronous invocations (see Fig. 3) run server and client-side logic in parallel. Therefore, the execution flow has the following structure: \(((1 \rightarrow 2 \rightarrow 3'' \rightarrow 7) || (3 \rightarrow 4 \rightarrow 5))\). In addition, its I.R. reduces to the following expression:

\[
I.R_{\text{client-async}} = \frac{1}{\sum_{j=1}^{N} \sum_{i=1}^{j} C_{ij}} \quad (2)
\]

Server-side asynchronous (see Fig. 4) remote invocations divide the 3\(^{rd}\) and 5\(^{th}\) steps into two (3' and 3''; and 5' and 5''). The 3' step receives data from the client to then continue executing 3'' in parallel with the rest of the invocation. Likewise, 5'' returns the acknowledgment to the client and 5' is in charge of post-processing. The resulting invocation path is, therefore, as follows:

\(((1 \rightarrow 2 \rightarrow 3' \rightarrow 5' \rightarrow 6 \rightarrow 7) || (3' \rightarrow 4 \rightarrow 5'')\)

The corresponding client-side invocation rate is described by the following formula:

\[
I.R_{\text{server-async}} = \frac{1}{\sum_{j=1}^{N} \sum_{i=1}^{j} C_{ij}} \quad (3)
\]

B. Data-Flow on Distributed Event Objects

In the distributed event model, several nodes are subscribed to a remote event service (using the subscribe primitive). Each time this service receives any data, it calls on the remote objects subscribed in this service. In its most basic form, the service calls the subscribed objects (on the remote trigger method), which may execute certain application logic in the subscribed remote node object.

According to the model described in Section III-A, in a service with \(N\) subscribed nodes (Fig. 5), the execution has the following structure:

\[
\begin{align*}
( & 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7, \\
& 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7, \\
& 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 )
\end{align*}
\]

The corresponding I.R. of this service is described by the following formula:

\[
I.R_{\text{async-events}} = \frac{1}{\sum_{j=1}^{N} \sum_{i=1}^{j} C_{ij}} \quad (4)
\]

(Note: \(C_{ij}\) refers to the cost of the \(i^{th}\) step in the \(j^{th}\) node).

1) Client and Server Asynchronism in the Event Service

Following the approach described in the previous section, the middleware may cut the flux of the remote invocation just before sending the requested or after the request has been properly handled. Therefore, the execution in a pure client-side asynchronous event model (see Fig. 6) is described as follows:

\[
\begin{align*}
( & (1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7) \quad (1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7) \\
& (1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 ) \\
& (1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 ) \\
& (1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 )
\end{align*}
\]

The invocation rate of this asynchronous communication model is the following:

\[
I.R_{\text{client-side-async-events}} = \frac{1}{\sum_{j=1}^{N} \sum_{i=1}^{j} C_{ij}} \quad (5)
\]

\[\text{(Note: } C_{ij} \text{ refers to the cost of the } i^{th} \text{ step in the } j^{th} \text{ node).}\]

---

\(1\) This formula is a necessary condition to support an end-to-end feasibility test. The test should address additional application related issues like the nature of the communications (i.e. soft, or hard real-time), the minimum inter-instruction rate at client and server, and the definition of a stability condition (e.g., utilization \(\leq 1.0\)). The same rationale is also valid for equations (3), (4), (5), (6) and (7).
A. Remote Invocation Interfaces for Remote Objects

Regarding the stub/skeleton service, the middleware uses two main templates: `RealtimeUnicastRemoteStub` (List. 1) and `RealtimeUnicastRemoteObject` (List. 2). Table II explains each parameter in detail. The client side (List. 1) defines local parameters for the remote invocation controlled from the server (e.g., scheduling parameters); these parameters are used during the remote invocation to set the connection and memory managers used during the remote invocation. It also allows the application to configure the asynchronous mechanism used by the static `setAsync` method (when it is set to true, the invocation will follow the `client-side asynchronous model`).

At the server (List. 2), applications may configure the resource model used during the remote invocation (priority-sp, invocation pattern –rp-, parameters for the garbage collector -mp-, and group parameters for the invocation -ppp-), and specific parameters for pools: map and thp. Asynchronism at the server is also supported by another `setAsync` method which decides if void methods will be asynchronous at the server or not.

```
package es.uc3m.it.drequiem.rtrmi.server;
import javax.realtime.*;
import es.uc3m.it.drequiem.rtrmi.*;
import java.rmi.server.RemoteStub;
public class RealtimeUnicastRemoteStub {
    public static void setParameters(
        RemoteStub stub,
        boolean async,
        SchedulingParameters sp,
        ReleaseParameters rp,
        MemoryParameters mp,
        ProcessingGroupParameters ppp,
        ConnectionPool cpool)

    ...

    public static boolean getAsync(RemoteStub stub);
    public static void setAsync(
        RemoteStub stub,
        boolean async);
}
```

```
package es.uc3m.it.drequiem.rtrmi.server;
public class RealtimeUnicastRemoteObject extends java.rmi.server.UnicastRemoteObject {
    int port,
    SchedulingParameters sp,
    ReleaseParameters rp,
    MemoryParameters mp,
    MemoryAreaPool map,
    ThreadPool thp,
    ProcessingGroupParameters ppp;

    public void setAsync(boolean async);
    public void getAsync();
}
```

IV. AN API FOR THE TWO MODELS

This section defines an API for the two models described in the previous section. It is based on the remote object model of DREQUIEMI and supports the two models proposed and its asynchronous patterns.

Likewise, in a server-based asynchronous communication, the following communication models the service (see Fig. 6):

\[
\begin{align*}
I.R_{\text{server-side-async-events}} &= \frac{1}{\sum_{i=1}^{n} \sum_{j=1}^{m} C_{ij}} \\
I.R_{\text{resulting}} &= \frac{1}{(I.R_{\text{old}} + I.R_{\text{new}})}
\end{align*}
\]  

(6) (7)

The most general case combines arbitrary combinations of distributed events, each one with its own asynchronous model. In general, if it is added a new handler to the system (old), the resulting I.R. is described as follows:

To demonstrate this formula, note that the inverse of the invocation rate is the cost (i.e., \( \frac{1}{I.R} = c \)). Therefore, from eq. 7 it stands out that

\[
I.R_{\text{resulting}} = \frac{1}{(C_{\text{new}} + C_{\text{old}})}
\]

which corresponds to the definition of the resulting invocation rate.

An eventual note should be written about normalized invocation rates (N.I.R.). An N.I.R. is obtained dividing the invocation rate (I.R.) of a certain optimized pattern by the basic performance of its corresponding synchronous pattern.
Scheduling Parameters sp
R.I.
{EventCommonInterface ci}
throws RemoteException;

public void unsubscribe (EventCommonInterface ci)
throws RemoteException;

public Serializable trigger(Serializable data)
throws RemoteException;

Listing 3: EventCommonInterface details

B. A Common Event Service Interface for Remote Objects

Distributed events have their own interface that allows the basic functionality required to subscribe/unsubscribe remote objects to another remote object. This basic functionality is obtained via subscribe and unsubscribe remote methods (List. 3). Among other utilities, subscribe allows the remote object to define its own asynchronous or synchronous communication model for client-side behavior via an async parameter. The list of parameters that may be configured is described in Table II. Lastly, trigger enables a node to receive and send data which have to be serializable from publishers.

TABLE II: Parameters defined by the Remote Invocation and Distributed Events

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning in DREQUIEMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling Parameters</td>
<td>Priority used during the up-call at the server</td>
</tr>
<tr>
<td>Release Parameters</td>
<td>Invocation pattern at the server according to the real-time scheduling theory</td>
</tr>
<tr>
<td>Processing Group Parameters</td>
<td>Invocation pattern at the server according to the real-time scheduling theory</td>
</tr>
<tr>
<td>Memory Parameters</td>
<td>Parameters used by the garbage collector at the server</td>
</tr>
<tr>
<td>Thread Pool</td>
<td>Thread pool used at the server to manage remote invocations</td>
</tr>
<tr>
<td>Connection Pool</td>
<td>Connection pool used at the client to carry out the remote invocation</td>
</tr>
<tr>
<td>Memory Area Pool</td>
<td>Memory area pool used at the server to accept remote invocations</td>
</tr>
<tr>
<td>port</td>
<td>The IP port on which the remote object is accepting incoming messages</td>
</tr>
<tr>
<td>RemoteStub</td>
<td>The stub with the remote reference to the remote object</td>
</tr>
<tr>
<td>EventCommonInt.</td>
<td>A remote object which allows subscriptions and may be remotely triggered.</td>
</tr>
</tbody>
</table>

Another interesting feature of this model is that it enables multiple levels of subscription. One node may be subscribed to a certain node and transfer the triggered signal to nodes subscribed at the same time. For instance, in Fig. 7 ro2 is subscribed to ro1 but it also feeds ro3 with a trigger signal.

V. EMPIRICAL RESULTS AND EVALUATION

In order to evaluate the two proposed models, it was carried out a set of empirical tests that show the temporal performance of both programming models. All these result were obtained on DREQUIEMI [14, 32, 33] which runs on the JTime virtual machine [34] over a 2.4x real-time TimeSys kernel. Both machines are interconnected via a 100 Mbits Ethernet network. On this infrastructure the goals pursued were (i) to evaluate the performance delivered by each model under similar conditions; (ii) to evaluate the performance on an industrial benchmark; and (iii) to measure the overhead introduced by the distributed event model on the former distributed event model of the DRTSJ.

A. Performance Patterns

The first experiment evaluates the performance of the different communication models. First, it evaluates the performance without application load (i.e., when remote methods and handlers are empty methods) to then focus the attention on the performance offered by the asynchronism with load.

1) Results without Application Load

Fig. 8 shows the cost of sending the data measured at the client for the remote invocations (R.I.) and the distributed event model: D.E(x). There is a subfigure for each asynchronous communication pattern (the left subfigure for the synchronous client, the center subfigure for asynchronous communications at the client, and the right subfigure for asynchronism at the server-side). In all cases, the measured time is the time required to perform the communication (i.e., 1 ) at the client (y-axis). In addition, the experiment TR considers applications that send a certain amount of data (x-axis from 100 to 600 bytes) per communication.

The first obtained result is that the remote invocation (R.I.) and distributed events (D.E(x)) have a similar performance when there is a single subscriber (D.E(1)). This behavior is valid with independence on the type of communication carried out: sync, async-client, and async-server. From the perspective of the programmer, it is an important issue because it means that the programmer may use any of the two models (D.E and R.I.) to design applications.

The second important result refers to the case with several handlers subscribed (i.e., D.E(4) and D.E(16)). There is a linear relationship among the number of subscribers and the blocking experienced at the server, as suggested by the model proposed in Section III, which is supported by the empirical evidence.

The last remarkable result refers to the use of asynchronism in communications. The results show (Fig. 8) that the time required for the communication reduces remarkably when communications are asynchronous. For instance, the time required to send 100 bytes (by 1ms with a synchronous communication) is reduced to 200 µs with asynchronism at the client and to 800 µs with asynchronism at the server.
Asynchronism with Application Load

The previous experiments were extended taking into consideration an application load. This section uses speed-up instead of absolute costs, which eases the asynchronous pattern evaluation under server load constraints. The speed-up is measured as a N.I.R (normalized invocation rate) using the ratio between the asynchronous model and the synchronous model (see Section III). Since the asynchronous models offer better performance than the synchronous, the N.I.R. measured is always \( \geq 1.0 \), which is confirmed empirically.

For the cases analyzed (Fig. 9 and Fig. 10), the load at the server ranges from 0.0 ms to 2.0 ms (y-axis) and maintains the transferred data range used in Fig. 8 (i.e., from 100 to 600 bytes in the x-axis of each subfigure).

The obtained results showed that the use of asynchronism may reduce the cost of the communication remarkably, thus increasing the N.I.R. Therefore, the programmer should use this mechanism when the communication may be modeled as an one-way interaction.

For the client-side asynchronism (Fig. 9), the N.I.R. is maximum when the server-side has to process each event for 2.0 ms. For this case, the invocation rate of the client is 22 times more efficient than the synchronous model (with one subscriber). The N.I.R. increases with the number of subscribers (e.g., with D.E(1) the N.I.R. is 22.06; with D.E(4) the N.I.R. is 22.9; and with D.E(16) the N.I.R. is 23.7).

For the server-side asynchronism (Fig. 10), the N.I.R. results are similar to those described for the client-side asynchronism. The main difference is the scale in which the client-side asynchronism ranges. The maximum of the N.I.R. (4.6) corresponds to the maximum execution time at the server (i.e. 2.0 ms) and the minimum amount of data transferred (100 bytes). For the same scenario, the N.I.R. is 22.06, which is five times more efficient than the asynchronism at the server. As in the client-asynchronous case, the maximum increases when the number of subscribed nodes increases. For instance, with D.E(1) the N.I.R. is 4.6; with D.E(4) the N.I.R. is 4.7; and with D.E(16) the N.I.R. is 4.8.

B. Industrial Use-Case Evaluation

A network-centric industrial benchmark was developed to evaluate the performance of the communication models in a realistic application. It is based on an industrial use-case defined in [33] for AUTOSAR applications. The frequency of the tasks in the framework ranges from 83Hz to 0.25 kHz and all tasks have to meet their deadlines before their next release (i.e., \( T_i = D_i \)). For each configuration, the framework evaluates the reduction on the available bandwidth for the application. Ideally, communications should take a reduced amount of CPU per cycle; however, on a real platform and depending on their
frequenct, the blocking experienced in communications is relatively important.

For the evaluated use-case, the communications were modeled using remote methods (R.I.) and distributed events with one subscriber per distributed event (D.E(1)). Therefore, the benchmark allows comparing both models under similar conditions. For each communication, the evaluation measured the percentage of time required to handle the communications per cycle (i.e., I.R/T), called U blocking.

The obtained results (Fig. 11) corroborate several of the synthetic results shown in Section V-A. The first result is that from the perspective of the middleware there is not great difference between remote invocations (R.I.) and the distributed events (D.E(1)); the impact and overhead are rather similar in both models. Therefore, the programmer may adapt its programming style to either remote invocations or distributed events without suffering an additional penalty in performance. The second result is that, whenever it is possible, the developer should opt for asynchronous models. In an 83 Hz application, the percentage of time required for the communication may reduce the consumed utilization from 38% (with sync) to 17% (with serv_async) and 13% (with cli_async).

![Industrial benchmark results depending on the frequency](image)

**Fig.11. Bandwidth consumed by the communications**

(2x1 Ghz Machines- 100 Mbits Ethernet)

C. Distributed Events Proposed vs. the DRTSJ’s distributed events

As stated in the introduction, in its original design, the DRTSJ considered a simple distributed event model. DRTSJ’s events did not allow user-data to flow through remote nodes; a choice that constrains the flexibility offered to the programmer. The last experiment evaluates the overhead introduced by the proposed solution against the former model defined in DRTSJ.

In the experiment it is measured the extra time required at the client to communicate with the remote node. The results (Fig. 12) show that the proposed approach has a 5.9% (async_cli), 5.3% (async_serv) or 3.1% (sync) overhead when compared against the previous distributed event model of DRTSJ. This initial behavior changes with the work carried out at the server. In synchronous communications, the overhead diminishes because it is masked by the work carried out at the server. In the asynchronous communications, the overhead remains plain because the execution cost at the server does not have impact on the client-side cost.

VI. Conclusions And Future Work

This paper has extended the horizons of distributed real-time Java from a pure remote-invocation model to a model that also includes distributed events. This type of communication model complements current remote invocations with the possibility of having anonymous interaction based on publisher-subscriber interactions. This paper has explored the definition, implementation, and evaluation of both models on an architecture for distributed real-time Java.

The empirical results bounded and benchmarked the performance of the system on a common-off-the-shelf (COTS) infrastructure. They showed that both models may deliver a similar performance, so that the programmer may choose the one that fits better on the application. Regarding the asynchronism, the results suggest that the developer should consider it when the application does not require any result from the server.

Our future work is focused on how to exploit these techniques in a service-oriented (SOA) middleware and on low-level optimizations for the distributed event model. The SOA explored paths include the use of optimized solutions based on heuristics [35], operating system support [36], and even integrating XML parsers [37]. In all cases, the patterns described in this article will help define a basic model for communications among nodes for service-oriented approaches. In addition, an end-to-end characterization for IP networks [38] is to be used as the primary quality-of-service description model. For the low-level optimizations, several strategies are under evaluation; they include parallel event notification (in the event service), and specific policies for handling incoming events at the subscriber. The use of these enhanced policies will be evaluated against the reference distributed event model defined in the article.

ACKNOWLEDGEMENT

The authors would like to express their gratitude to the anonymous reviewers of the article.

REFERENCES


Currently, she is the technical coordinator of iLAND, a European project of the ARTEMIS JU.

**Iria Estévez-Ayres** born in Vigo, Pontevedra (Spain). She obtained her Telecommunication Engineering degree from Universidad de Vigo in 2001 and her PhD from Universidad Carlos III de Madrid in 2007. Currently, she is an assistant professor at the Telematics Engineering Department at Universidad Carlos III de Madrid, and she is member of the Distributed Real-Time Systems Lab. Her research is focused on the applicability of service oriented architectures to real-time and networked embedded systems.