A synchronous scheduling service for distributed real-time Java

Pablo Basanta-Val, Iria Estévez-Ayres, Marisol García-Valls, and Luis Almeida

Abstract—Current trends in real-time systems identify Java as a new alternative to develop both centralized and distributed real-time systems. Many efforts have been devoted to develop the Real-Time Specification for Java (RTSJ), and there is substantial ongoing activity to produce a straightforward and valuable Distributed Real Time Specification for Java (DRTSJ). The current paper provides a contribution to this latter activity defining, from different angles, a synchronous scheduling service aligned with principles of some popular real-time architectures. This service orchestrates the system in such a way that it provides end-to-end guarantees in the distributed transactions, guaranteeing their timely execution across the network and nodes. The service is described from two points of view: the system one, characterizing a portable model; and the programmer one, defining a distributed object-oriented implementation of a model based on Real-Time Remote Method Invocation (RTRMI). Finally, it also presents results of an implementation carried out to judge the efficiency of the service, offering a preliminary predictability and performance assessment of a distributed real-time Java technology.

Index Terms—real-time, synchronous scheduling, distributed real-time Java, RTRMI, RTSJ

I. INTRODUCTION

Since its birth, real-time technology has undergone several changes in motivations, constraints and targets. In their first decades, influenced perhaps by the high cost of the hardware, resource efficiency was one of its main goals. Today, with the extraordinary reduction in hardware cost, and increase in computing and communication capabilities, we are witnessing an exponential growth of the applications complexity, making reusability and flexibility of programming technologies a more important goal. One of the technologies that favors such goals is Java, making it particularly suited to deploy a new generation of distributed, embedded, and real-time applications [29]. Briefly during the last decade we saw the birth of real-time Java technologies such as the Real-Time Specification for Java (RTSJ) [9], which harmonize the requirements of the classical real-time systems with the peculiarities of the Java virtual machine model. In the coming years, we will probably see the spreading of the Distributed Real-Time Specification for Java (DRTSJ) [17] [3].

Many issues still need to be addressed on the way towards a DRTSJ, one of them being the way time is handled across the distributed system. The issue is whether nodes will share (directly or indirectly) a global knowledge of time or if, instead, each node will evolve independently asynchronously. It is well-known that both schemes offer advantages and disadvantages [21]. The support for asynchronous models is typically easier to implement and configure but it introduces a high variability in program executions, whereas synchronous models are more difficult to implement but offer higher predictability to the programmer. Therefore, it is our opinion that a general purpose distributed real-time Java technology should give a good support to both paradigms.

In the related literature there is a substantial and mature work on synchronous models. The TTA (Time Triggered Architecture) provides a tightly coupled approach to the problem of synchronizing a distributed system [23]. A clock synchronization protocol provides a common clock across the system that is used to drive the transmissions in the network together with the tasks executions in the nodes according to a global synchronous schedule. This technology provides a well-defined communication mechanism based on shared memory that isolates the temporal behavior of nodes and network. This results in a very predictable behavior that allows optimizing end-to-end delays with low jitter and building low-latency fault-tolerance mechanisms. However, the rigidity inherent to the tightly-coupled global schedule has been criticized as excessive for applications with certain demands of operational flexibility, either to support live reconfiguration or even adaptation to changing resource availability or variable demand. In this type of applications, the admission control, which was traditionally relegated to an off-line analysis, becomes an on-line need. One approach that has followed this path towards support for on-line admission control is the FTT (Flexible Time Triggered) paradigm [1] in which a central master synchronizes the whole system.

However, both FTT and TTA targeted local communications, for tightly coupled distributed applications, providing a relatively low software abstraction level, for example, when compared with technologies like Java/RMI (Java’s Remote Method Invocation) and TCP/IP. Nevertheless, these technologies can still take advantage from the TTA or FTT operational models and some of their principles to improve real-time Java predictability without damaging Java’s portability. Particularly, having an implicit or explicit global knowledge of a system-wide time may be interesting in a distributed real-time Java application to coordinate external actions (e.g. cooperating autonomous robots, sensing and actuation in a process plant...), thus improving the overall predictability of a distributed system. However, other operational features of FTT and TTA conflict with RMI [6], such as the use of broadcast communication, e.g., the trigger-message in FTT, which clashes with the unicast nature of RMI.

These principles of real-time coordination of a distributed system have been incorporated in RMI via DREQUIEM (a Java’s real-time distribution middleware from DREQUIEM Lab. [1]). This is an RTRMI middleware that includes support to the most common techniques used by distributed real-time applications. Among others, it already provides global priority-based support for remote

1http://www.it.uc3m.es/drequiem

Manuscript received January 20, 2008; revised August 13, 2007.
Pablo Basanta-Val, Marisol García-Valls and Iria Estévez are with Universidad Carlos III de Madrid (SPAIN). Luis Almeida is with Univerdade de Porto (PORTUGAL).
invocations similarly to the one already defined in RTCORBA (Real-Time Common Object Request Broker Architecture) and it also provides the concept of thread-pool or memory-pool [8] [7].

In this paper we analyse some principles from the FTT framework in particular and propose their integration in DREQUIEMI to further improve the timeliness of distributed Java applications. The paper is organized as follows. Section II presents a brief survey of related work, focusing on the sources of inspiration to DREQUIEMI. Section III deals with the major integration issues or what we have called the multicast vs. unicast disjunction, defining what, why and how the FTT mechanisms have been adapted to the unicast environment of DREQUIEMI. Going into more detail, Section IV presents the analytical model used by the DREQUIEMI’s synchronous scheduling service as well as the concept of choreography. Sections V, VI, and VII cover the Java-related aspects of the model. The first one gives an overview of the specific architecture used to provide support for the model. The second one outlines the programmer perspective, whereas the third presents the results obtained with an empirical evaluation of a model prototype. Section VIII shows two illustrative use cases developed with the synchronous scheduling service. The conclusions are presented in Section IX together with an outline of our related ongoing efforts.

II. Related work

At the middleware level, CORBA distribution middleware offers support for both synchronous and asynchronous techniques. Its real-time specification, RTCORBA [30], makes use of real-time priorities to reduce the priority inversion of the most urgent tasks but at the cost of introducing high penalties in the lower priority ones. On the other hand, the Time Service of CORBA [26] provides a high resolution global clock that is available to the different nodes of the system as well as a Real-Time Event Service [14] that may be used to provide a global synchronization signal across the network. Global time-triggered approaches such as the TTA [22] or its middleware counterparts TMO [19] and ROFES [24] rely on such global synchronization to prevent contention at the access to shared resources, particularly the network at which level a TDMA access scheme is enforced.

The requirements for such a synchronous middleware, namely TMO, are described in [18]: (A) timely activation and execution of the time triggered methods, (B) timely transmission of remote invocations between nodes, (C) timely recognition of an arrival and (D) timely execution of each remote method. In [20], these requirements are met in a non-intrusive way through the definition of two new services: TMOES (TMO Execution Support) and CNCM (Cooperating Network Configuration Management). The former, running on top of an ORB, offers support to A, C, D while the latter supports B maintaining RT computing groups of cooperating nodes.

One approach that is clearly oriented to reconcile flexibility and timeliness as needed in distributed embedded systems is the FTT (Flexible Time Triggered) paradigm [27]. It concentrates all system scheduling information in a central node, the master, which may modify the evolution of the entire distributed system dynamically by means of triggering events that are sent regularly to the remaining nodes, the slaves. This paradigm was initially designed to work over CAN buses (FTT-CAN [1]) but it has also been applied to Ethernet, both shared (FTT-Ethernet [28]) and switched (FTT-SE [25]). Moreover, a combination with Java technology was proposed in which small pieces of Java code (FTTLets [10]) run atop a virtual machine and are activated transparently by a master engine. More recently, it was also combined with the CORBA object-oriented middleware [11]. However, its integration with the new emerging generation of unicast distributed middleware, mainly based on the real-time Java language and especially those that combine RMI and RTSJ paradigms, is still pending.

DREQUIEMI [6], on the other hand, is an RTRMI middleware that provides such RMI and RTSJ combination. Although the scheduling service of DREQUIEMI shares principles with almost all previously referred works, FTT is perhaps the one with the highest degree of coincidence. Particularly, both are based on the master-slave paradigm, having a master that is a shared entity provided with a certain global knowledge of the system, and a number of slaves executing the application specific logic. However, as shown in Table I, they also differ in several key aspects. Whereas FTT understands a distributed system as the distribution of raw-memory among slaves, DREQUIEMI scheduling service understands this information as structured data originated always from the master.

The contribution of this paper is to bring FTT principles into DREQUIEMI to provide a globally synchronous scheduling service for distributed real-time Java. Thus, the ideas proposed in this paper may be used to improve DRTSJ with a real-time service (synchronous scheduling service) that

- considers the temporal characteristics of the subscribed slaves,
- may send data to a central master,
- may receive data from a central master,
- and defines a set of basic choreographies (or primitive transactions) to provide a functionality similar to that of FTT.

The resulting middleware maintains the high level of flexibility with timing guarantees that characterizes both approaches and which is not typically met by the other referenced middlewares.

III. Integration of FTT-principles in DREQUIEMI

Some of the key characteristics of the FTT paradigm are based on the existence of a data link layer broadcast service to distribute the messages exchanged among the different system nodes. In this case, this shared medium (e.g. Ethernet) or protocol-level
emulation (e.g. IP-Multicast [11]) enables a low-cost mechanism to disseminate data efficiently among all nodes in the network that helps maintain a replica of certain data in a set of nodes because such mechanisms copy data automatically and very efficiently, from one node to the rest of the network.

However, from the point of view of a middleware such as DRTSJ, or other RTRMI alternatives based on unicast/transactional communications such as DREQUIEMI, this kind of support is not easy to develop because their programming paradigm, inherited from RMI, mainly relies on unicast/transactional remote invocations. From the integration point of view, and in order to maintain certain performance, this forces one to choose between the shared memory model of FTT or the client-server model of RMI. The use of an IP-multicast paradigm would bring us closer to the operating model of FTT but it would also constrain the model to local area network (LAN) infrastructures. On the other hand, although a client-server model requires the redefinition of the operating model of FTT toward a unicast/transactional remote invocation, it also avoids having a shared vision of the whole distributed system in every slave.

In this work, the decision was to maintain the integrity of RMI (unicast/transactional) instead of integrating multicast. This option gives independence and portability from the low-level communication protocols and allows having a full control over the messages that are sent to the nodes from the master side. Besides, it may help DREQUIEMI define specific strategies (application-dependent) that satisfy the requirements of certain applications. In this case, the use of the transactional nature of RMI allows a master to know if after the end of a transmission the message has been delivered (or not) to a slave. Such behavior is not common in multicast systems in which transmissions are typically unacknowledged. On the other hand, using a unicast/transactional model may yield a performance penalty, mainly in the cases in which there are many slaves requesting the same data. But this should be compensated by having a more reliable and flexible infrastructure based upon RMI.

In the absence of such broadcast service at the data link level, several FTT mechanisms must be readapted. In particular, this section shows the adaptation carried out in three important features of FTT, namely:

- the periodic activation signaling (trigger message);
- the direct communication among slaves;
- the support for synchronous and asynchronous traffic with temporal isolation.

These three mechanisms, upon readaptation to a unicast network, have been adopted by the synchronous scheduling service of DREQUIEMI.

A. Periodic activation signalling

In FTT the master organizes the traffic in fixed duration slots called Elementary Cycles (ECs), which are triggered by a periodic signaling mechanism called Trigger Message (TM) that conveys the traffic schedule for the respective EC (Figure 1). The remaining nodes, i.e., the slaves, decode the TM and transmit the traffic scheduled therein. The traffic scheduling for each EC is carried out on-line, by the master, in the previous EC according to any desired scheduling discipline.

The transmissions of the slaves within each EC are either arbitrated by the underlying native medium access control protocol or separated in time by a transmission control mechanism. For example, in the case of the FTT-SE protocol for switched Ethernet, the transmissions of the slaves are carried out immediately after decoding the TM and are queued in the switch.

A straight implementation of the FTT synchronization mechanism in a unicast network would require the replication of the TM and its transmission to all slaves one by one, in each EC, leading us to an application level multicast. This would result in a strong penalty in consumed bandwidth and network performance because of the high number of TM replicas that had to be sent every EC. Thus, this synchronization mechanism is not adequate for a unicast network. Therefore, DREQUIEMI uses instead a common master-slave synchronization in which the master signals each slave directly and independently when some type of action is required from it (Figure 2). In this case, the master waits for the completion of each transaction, thus preventing access conflicts in both the network and slaves and there is not an elementary cycle that governs the evolution of the transmissions. Each transaction is triggered independently at its own rate by the master.

B. Slave-to-slave communications

In the FTT paradigm, the slaves transmit when triggered by the TM and such transmissions are addressed directly to the respective receivers in a connectionless manner. However, in a connection-oriented unicast network, as used in DREQUIEMI, direct slave-to-slave communication would require a potentially large number of logical connections (with n slaves the number of connections could raise up to \((n-1)\frac{(n+3)}{2} + n\)). Therefore, DREQUIEMI follows a master-centric approach (Figure 3), in which all communications are carried out through the master, requiring just n logical connections, i.e., between the master and each slave.

Conversely, the master-centric approach presents a larger message delivery time because of the forwarding mechanism that the master must carry out to transfer messages between slaves.
Another problem of master-center approaches, which may be solved with a backup master [13], is their tendency to have a single failure point in the master.

C. Synchronous and asynchronous traffic

In order to cater for the requirements of common distributed real-time applications, many real-time networks provide support for two types of traffic, namely synchronous and asynchronous. Typically, the former class is used to support periodic traffic with system-wide synchronism, e.g., using a common clock, while the latter aims at handling communications associated to asynchronous events, such as alarms, logging and self-triggered components.

The FTT paradigm includes support for both classes using two separate phases (windows) inside each EC, with strict mutual temporal isolation. This option increases the robustness of the system and provides QoS guarantees to each traffic type (Figure 4 - top). However, the existence of such disjoint phases can be a source of inefficiency because one phase can be overloaded while the other phase may still have some slack. This is attenuated in the FTT framework by allowing the asynchronous phase to reclaim unused bandwidth in the synchronous one.

As seen previously, DREQUIEMI has not adopted the cyclic EC structure and consequently, the multi-phases concept does not apply. It is the master that schedules all transactions independently, one by one, either synchronous or asynchronous, in the latter case using polling (Figure 4 - bottom).

IV. COMPUTATIONAL AND ANALYTICAL MODELS

As Figure 5 shows, the actors involved in DREQUIEMI’s scheduling service are only two. To maintain certain similarity with FTT, we have named them master and slave. The master represents the communication mechanism among the different nodes of the system. From the point of view of an application programmer the master is a networked shared object accessed by the different nodes (slaves) of the system. In some sense, it may be also understood as some kind of broker because its knowledge is enough to transfer the information from one slave to any other on the network guaranteeing a timely delivery of the information. To achieve it, it has full control on the messages transmitted through the network; no message may be sent without its previous consent.

The other actor of the model is the slave. From the point of view of a programmer, a slave is a local entity that may put and/or get data in/from the master. Its knowledge of the remaining system comes through the master, only. It has no direct connection with any other entities in the system.

This simple model can be supported by different topologies. Figure 5 in the lower part shows a simple deployment architecture with each node residing in one computer, and a switch interconnecting the different nodes of the system. Altogether the system contains one master and three slaves.

A. Types of transactions

By default, a master emulates the behavior of FTT and may execute six so-called primary transactions (Table II), namely T, S, P, C, AP, and AC. The first two are intended to specific application domains that only require a simple temporal coordination of slaves. On the other hand, P and C provide support to synchronous communications among several nodes, while AP and AC offer the corresponding support for asynchronous communications.

<table>
<thead>
<tr>
<th>Acro</th>
<th>Denomination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Trigger transaction</td>
<td>It causes the execution of a slave.</td>
</tr>
<tr>
<td>S</td>
<td>Synchronization transaction</td>
<td>It synchronizes the master and slave clocks.</td>
</tr>
<tr>
<td>P</td>
<td>Produce transaction</td>
<td>It retrieves data from a slave.</td>
</tr>
<tr>
<td>C</td>
<td>Consume transaction</td>
<td>It pushes data from the master to the slave.</td>
</tr>
<tr>
<td>AP</td>
<td>Asynchronous produce transaction</td>
<td>It permits a slave to push data to a master.</td>
</tr>
<tr>
<td>AC</td>
<td>Asynchronous consume transaction</td>
<td>It permits a slave to get data from a master.</td>
</tr>
</tbody>
</table>

These transactions, which we will call choreographies following the terminology of W3C, are all initiated by the master, whereas the slave plays a passive role.

- **Trigger transaction (T-Choreography).**

In this case, the information sent from the master triggers
the execution of a method in a slave. This type of message, in principle, does not contain any application information. It may contain, however, a delay parameter to delay triggering the method execution correspondingly. In the example of Figure 6, the master sends a delay trigger message to the slave, indicating that its activation should not be immediate but delayed 500 units of time counted from the start of the respective transaction. At the master side, the execution of the choreography ends just after the reception of the confirmation from the slave, without having to wait for the activation or termination of the runnable object.

This transaction pushes data from the master into the slave (figure 9). This transaction must have an associated P-Choreography from which buffer, in the master, the data will be fetched and the value retrieved is always the most recent.

Fig. 6. Details on the T-transaction

- Synchronization transaction (S-Choreography).
  In this case, the master sends its own current clock to allow the slave to carry out clock synchronization (Figure 7). Many possible algorithms can be used such as those referred in [4] or the more recent IEEE1588. In turn, the slave sends back the offset of its local clock with respect to the newly received master one, before any correction is applied, which allows the master to be aware of the performance of the slave clock synchronization.

Fig. 7. Details on the S-transaction

- Produce transaction (P-Choreography).
  In this case, the transaction pulls information from a slave into the master (Figure 8). Retentive buffers are used in both sides, producer and master. Note that consecutive executions of the same P-Choreography always fetch the most recent data in the producer and overwrite the contents of the respective buffer in the master.

Fig. 8. Details on the P-transaction

- Consumer transaction (C-Choreography).

Asynchronous Producer transaction (AP-Choreography)
This transaction is similar to the P-Choreography but the transfer of data is conditional. In this case, the buffer in the producer is non-retentive and if it is empty there will be no effective data transfer and the transaction ends sooner. The data buffer in the master is still retentive, to allow sending the same information to multiple consumers. The conditional transfer allows reclaiming unused bandwidth that can be allocated by the master to other asynchronous transactions as long as they fit within certain windows defined in the schedule, called asynchronous windows (Figure 10), which typically correspond to intervals of time between synchronous choreographies.

Fig. 9. Details on the C-transaction

- Asynchronous Consumer transaction (AC-Choreography)
  This transaction is similar to its synchronous counterpart, the C-Choreography, except that it is preceded by a message exchange in which the master informs whether the associated AP-Choreography transferred new data in its previous cycle. If so, its data is pushed similarly into the slave. Otherwise, the data transfer is skipped releasing bandwidth as in the AP-Choreography case (Figure 11).

Fig. 10. Details on the AP-transaction

Fig. 11. Details on the AC-transaction
B. Choreography scheduler

In order to support timely communication every choreography must be time-bounded. Then, a real-time scheduler running in the master schedules the choreographies which are handled as periodic non-preemptive tasks. The definition of a choreography as a non-preemptable task helps the implementation skip many problems that stem from the indivisibility of the messages transmitted through the net or the possibility of having collisions during their transmission. Nevertheless, it also forces a choreography to be bounded and short, in order to give good performance.

There are several results available in the real-time scheduling literature for non-preemptive scheduling (e.g. [5], [16], [15] or [2]) that may be applied to a non-preemptive set of choreographies. One scheduling algorithm that is known for its high scheduling efficiency is EDF, which is known to be optimal for single processor preemptive systems. However, it turns out that non preemptive EDF (NPR-EDF) in single processor systems is still optimum [16] in the sense that if the set of concrete non-preemptive tasks is schedulable by a non-idling algorithm, it is also schedulable by NPR-EDF. Therefore, we decided to include this kind of scheduling in DREQUIEMI to improve its efficiency.

1) Temporal model of the choreography:
Let \( \chi \) be the set of choreographies that must be executed by the master.

\[
\chi \equiv \{\xi_i\} \forall i = 1..N
\]

Let each choreography be characterized as follows

\[
\xi_i \equiv \{T_i, D_i, C_i, O_i\} \forall i = 1..N
\]

where \( T_i \) is its period, \( D_i \) its relative deadline, \( C_i \) its duration and \( O_i \) its initial offset.

2) Admission control:
Given the NPR-EDF scheduling, we can use an adequate schedulability test to decide about the admission of a set of choreographies to guarantee they meet their time constraints. The admission control mechanism used by DREQUIEMI scheduling service is based on the one developed by Baruah for multiprocessor systems [5] but simplified for one processor, only. Note that the master is a single processor that executes choreographies in sequence.

This results in an upper bound to the (master) utilization factor directly deduced from the deadlines of the choreographies in \( \xi \) and their maximum duration. Let \( \rho \) be:

\[
\max\{C_i\} < \rho \times \min\{D_i\} \forall i = 1..N
\]

Then, the set of choreographies \( \xi \) is schedulable if (sufficient condition):

\[
U = \sum_{i=1}^{N} \frac{C_i}{D_i} \leq (1 - \rho) \leq 1
\]

Note that this admission control is more efficient for shorter choreographies. Therefore, if the network has high-bandwidth and the nodes are performant, or if the choreographies are kept short in terms of maximum duration, the efficiency of the test increases, allowing a higher utilization with guaranteed timeliness. In the limit, when the choreographies durations tend to zero the efficiency asymptotically grows up to an ideal utilization factor of 100%. Moreover, the efficiency of this test also increases when the deadlines are equal to the periods of the transactions, which is a common situation in practice (\( D_i = T_i \forall i \)).

Another advantage of this test is that it is computationally light and thus it can be executed on-line to decide on the admission of new choreographies that can be added at runtime.

Note that this test applies to the master, only. To assess the schedulability of the whole system, a more complex test must be considered that incorporates the load in the slaves, too, in a holistic fashion. If the choreographies schedule is such that their execution in the slave nodes is strictly periodic, then this test can also be applied to each slave independently, considering the respective subset of choreographies plus the set of methods that the slave executes. If the choreographies schedule in the master presents scheduling jitter, then this test must be adapted in the slave nodes, to include such jitter effect. Such adaptation is currently under development.

3) NPR-EDF scheduler:
The non-preemptive EDF scheduling service orchestrates the different choreographies of the distributed application in the master, providing an adequate admission control and sequencing of messages through the network. Algorithm 1 shows its pseudocode. Basically, the scheduler stores a set of active choreographies in an internal structure. A new choreography is pushed by the method addIFFeasible into the system and removed by the cancelChoreography method. As we can see, the pseudocode has been optimized to decide in constant time about the schedulability of the whole system maintaining the state of the current utilization factor in internal variables. The remaining operations have a linear dependency, \( \Theta(n) \), with the number of scheduled choreographies.

V. IMPLEMENTATION ARCHITECTURE

In order to give support to the computational model described in the previous section, a platform based on real-time Java technologies and the RTRMI middleware was developed. As Figure 12 shows, it consists of three layers: (1) a real-time Java virtual machine that allows to control the primary resources such as memory or the processor; (2) a real-time Java distribution middleware based on the RMI paradigm (DREQUIEMI), able to provide some control over the resources involved in a remote invocation; and (3) a convergence layer, in charge of the specific support for the master-slave programming model on top of a RTRMI node.

Fig. 12. Layered architecture for the master-slave model
Algorithm 1: Details on the NPR-EDF scheduler

```java
public class NPREDFScheduler {
    double rho=0;
    long double minDeadline=inf;
    double maxCost=0;
    double utilization=0;
    ChoreographySet coreographySet= new ChoreographySet();

    public void addChoreography (Choreography chr) {
        coreographySet.put(chr); /* Complexity: O(1) */
    }

    public void runScheduler () {
        while true do
            executeNextChoreography();
        end
    }

    private void executeNextChoreography () {
        Choreography chr = coreographySet.waitForTheReleaseTime();
        /* Complexity: O(n) */
        chr.run();
    }

    public void cancelChoreography (Choreography chr) {
        coreographySet.remove(chr);
        if (coreographySet.size()==0) {
            coreographySet=null;
        } else if (coreographySet.size()==1) {
            coreographySet=null;
        } else {
            coreographySet.remove(chr);
        }
    }

    private void recalculateRho () {
        foreach i in coreographySet do
            if (coreographySet(i).deadline<minDeadline) then
                minDeadline=coreographySet(i).Deadline;
            end
            if (coreographySet(i).cost>maxCost) then
                maxCost=coreographySet(i).cost;
            end
            rho=maxCost/minDeadline;
        end
    }
}
```

trol on object allocation, the new real-time Java threads, the event mechanism and the hardware access) the proposed master-slave architecture uses a small subset of them, namely those more predictable: ImmortalMemory, the LTMemory and the NoHeapRealtimeThread. Other characteristics, such as RealtimeThreads or the HeapMemory, are not considered because of their higher unpredictability.

As DRTSJ does, the synchronous scheduling service also requires from RTSJ that the classes for time (HighResolutionTime) are serializable. This extra requirement simplifies the transmission of time information among nodes remarkably.

### B. RTRMI required support

On top of the RTSJ layer, there is another one in charge of providing a predictable remote invocation for RMI. Internally, it uses the RTSJ facilities to modify the priority of clients and servers that give support to the master-slave paradigm, being able to remove the temporal objects allocated during a remote invocation and also to provide some control over network connections allocation.

From the point of view of the master-slave model, the most important characteristics that DREQUIEMI offers are the following:

- **Initialization and mission phase.** The execution of remote applications is divided into two phases: initialization and mission. In the initialization phase all slaves subscribe themselves to the master. After this initialization phase, the mission phase begins. This phase corresponds to the normal system operation with the master sending the command messages to the subscribed slaves. In this case, the master is allowed to use auxiliary regions to remove the temporary objects required to send the command messages to the proper nodes.

- **End-to-end priority management.** In a similar way to the RTCORBA model, it is possible to use two priority models, one where the remote object executes at the client priority and another in which it is possible to define a priority to execute the logic of a remote object. The master-slave architecture uses the latter, enabling the execution priority of remote objects to be configured in the server during their initialization.

- **Memory-Area Pools.** To eliminate the objects created during the remote invocation, the server side of the middleware offers the possibility of using an entity, i.e., the MemoryAreaPool, to control the type of memory that is used during the remote invocation. In this way, the programming paradigm for the slaves will be the NoHeapRemoteObject [7] where all the objects dynamically created during the remote invocation are collected after the execution of the server.

- **Connection Pools.** Also, in a similar way to the RTCORBA model, it is possible to perform an initial reservation of a set of connections that are reutilized during several consecutive remote invocations. During the reservation of these connections it is possible to define a common priority used by the server to start the processing of a remote invocation. The master-slave architecture uses this feature to establish a connection with each one of the slaves in the system, thus minimizing the communication latencies and the generated interferences.

### C. Convergence layer

The main problem addressed by this layer is the absence of asynchronous remote invocations in RTRMI. Following the principles of the RMI middleware, the communication paradigm of DREQUIEMI is synchronous, meaning in practice that a master should be blocked until the termination of each invoked slave method and only then continue its execution. But from the point of view of the master-slave architecture this kind of behavior is inappropriate because it does not allow the parallel execution of master and slaves. This problem is solved providing a desynchronization layer at the slave side, as shown graphically in Figure 12.

In total, the process of invoking a slave involves three main concurrent entities:

- **Master.** This is the thread running at the master node that sends command messages to the different slaves of
the distributed system. In order to reduce the jitter of this operation, this thread should run with the highest priority in the system.

- **Slave-clerk.** The clerk is one of the two entities of a slave node that is in charge of interacting directly with the master. This entity waits blocked in a socket for incoming messages and activates the slave code with the data received from the master when necessary. After this activation, it sends a confirmation message to the server (master) in order to notify it of the success of the processing of the data. Similarly to the master thread, the clerk should also execute at the highest priority of the node, to contribute to minimizing the processing jitter.

- **Slave-handler.** This entity contains a handler thread that executes the asynchronous code of the slave triggered by the master, e.g., a wait for an activation in a T-choreography. Its priority should be lower than the one of the clerk. Typically, the relationship among the priority of these threads should be as follows:

\[
P_{\text{master}} \geq P_{\text{clerk}} > P_{\text{handler}}
\]

This schema favors the predictability of the communications by reducing the potential interference on the master and clerks, thus also contributing to the predictability of the handlers triggering mechanism. If this relationship is not followed, e.g., with \( P_{\text{clerk}} < P_{\text{handler}} \), the clerk can be blocked by an application handler and as a consequence delay the activation of another handler with higher priority than the one running, which would otherwise preempt it.

VI. PROGRAMMER PERSPECTIVE

The use of the master-slave model still requires a set of specific extensions to DREQUIEMI. The proposed extension is object-oriented. It defines a basic interface, FTTRemote, extended by both master and slave, a raw structure to interact with messages (FTTData), and a common template Choreography designed to offer a basic support to the concept of choreographies in the master.

A. Hierarchy overview

As Figure 13 summarizes, there are two common roots for the classes: java.rmi.Remote interface, common for the objects that may be remotely invoked; and java.io.Serializable, a common class for all data exchanged among the different nodes in the network.

![Class hierarchy](image)

Both master and slaves share a common interface that models the behavior of a more general entity, the FTTRemote. This interface is a generalization of the FTT paradigm that consists of four methods: two for the subscription and unsubscription of a slave, one to modify these parameters dynamically, and a forth related to the generation of the trigger message from a master to a slave. All the other interfaces are particularizations of this one.

B. Details on the master and slave templates

The FTTMaster is a template for the master that gives the basic support for periodicity and slave management. It may be used directly by the programmer or indirectly by advanced specializations such as the EDFMaster which was mentioned in the previous section. Figure 14 shows details of this API.

![Master and skeleton interfaces](image)

To instantiate an FTTMaster entity, it is necessary to define its execution priority, and a block of memory used to eliminate objects allocated during the execution of a choreography. The dynamic management of slaves subscription uses the methods subscribe and unsubscribe to introduce, remove or modify the set of registered slaves.

The master life cycle is controlled using the start and stop methods, defining three stages. The first is an initialization phase where the master waits for incoming slaves and ends with the start method invocation. The second is the phase where the master sends the command messages to the subscribed nodes, according to their deadlines, until the stop method is executed. Lastly, the third, starting with the stop method invocation, stops the master activity, namely the communications among the entities of the network.

The slave support complements the one of the master, providing mechanisms for the processing of command messages. Figure 14 also shows the internals of the FTTSlave template. During the instantiation of a slave it is necessary to define the execution priority for the internal clerk; the memory to process incoming messages at the slave side; the priority of the handler thread and the block of memory where data are finally allocated.

Currently, the allocation of master and slaves to physical nodes is carried out by the code of the application. From the programmer perspective, both are remote objects and thus can be instantiated in any RTSJ-enabled node with the described support for the synchronous scheduling service. Although master and slaves will typically reside in different physical nodes, the model is flexible enough to support their allocation in the same processor if needed, in a transparent way.

C. Choreography and FTTData

Figure 15 gives a brief description of a choreography, from the programmer perspective. This class, as any other RTSJ schedulable entity, offers the possibility of getting and modifying the requirements of a single communication or a set. In its most primitive form, it may encapsulate S, T, P, C, AP and AC transactions grouped in a certain manner. Typically, the code
of the programmer uses the `setSchedulingParameters` to define the temporal properties of a certain choreography, and the master uses the `getSchedulingParameters` to access to the properties of the system.

![Choreography and FTTData skeleton interfaces](image)

**Fig. 15.** Choreography and FTTData skeleton interfaces

The figure also gives an overview of the FTTData basic structure. It defines the basic interface used to read and write data in a synchronous fashion (`getData` and `setData`), which forms the basic communication mechanism. A call-back function (`waitMessage`) is also available that allows waiting for an incoming message and thus synchronize the application code in the slaves with the choreography schedule in the master. The `getRealtimeClock` method, already available in RTSJ, allows accessing a global shared clock that can be used to coordinate distributed actions in a system. Finally, the interface for asynchronous communication is provided by the `sendAsync` and `receiveAsync` methods.

**VII. Empirical evaluation**

In order to obtain an empirical performance evaluation of this service as a proof of concept, two types of experiments have been carried out. The first set aimed at determining the cost of executing each type of choreography in terms of processing and memory in a networked environment. The second set of experiments aimed at assessing the predictability of the system in terms of activation jitter of the master and slaves.

**A. Execution time**

In order to evaluate the cost of executing the proposed choreographies we used a two-machine environment connected with an Ethernet network: the master resides in one node while the slave is in another machine. All our results refer to 735MHz Celeron PCs with 120MB of memory running the jTime virtual machine (version 1.0) [31], the Timesys 2.4.7 real-time operating system [32] in a Knoppix kernel 2.4.22-xfs egcs-2.91.66 Linux distribution and the network is an isolated 100-Mbps switched Ethernet.

The first experiment consisted in determining the time required by the master to execute the basic choreographies: `P`, `C`, `AP`, `AC`, `T`, and `S`. Since the cost of executing `P`, `C`, `AP` or `AC` strongly depends on the type of data transferred, an artificial benchmark was generated containing five different cases: 0, 10, 25, 50 and 100 `java.lang.Double` objects stored in an array structure.

Figure 16 summarizes the time required by the master to execute each choreography. `C`, `P` and `AP` choreographies are very close in performance for the same amount of transferred data, the reason being that the data transmission time and the processing carried out by the middleware are dominant and mask the differences of the specific logic of each choreography. In the case of an `AC`, the execution time shows a nearly constant overhead of 945µs for any data size, which arises from the extra initial interaction between master and slave, in which the former informs the latter of the presence of fresh data or not in the associated producer buffer.

Another important result stems from the asynchronous choreographies (`AC` and `AP`) that may allow recovering an important amount of time to execute more choreographies of that kind when there is no actual data transfer. For example, in the case of an `AP` choreography for 100 data items, its execution time can be 10925µs when the data is ready or just 944µs if it is not, an order of magnitude reduction. This reduction is even larger in a corresponding `AC` choreography, from 11862µs to 945µs.

Finally, the execution times of the `T` and `S` choreographies are very similar and under 1ms.

**B. Dynamic memory consumption**

The analysis of latencies, despite important is not the only relevant one. Another important analysis, especially in real-time embedded systems, is the memory that is dynamically required to send the command messages by the master and the one that is required by the slave to process them. Furthermore, in RTJava (RTSJ) this analysis is even more important to configure the real-time garbage collector, or to dimension the regions used to remove temporal objects created during the execution of a choreography.

In this analysis we also cover the set of `C`, `P`, `AC`, `AP`, `T`, and `S` choreographies. Figure 17 shows the maximum memory allocated dynamically in the master and in the slave to accomplish the transactions. In `P`, `C` and `AP` cases, the memory consumption is relatively high, ranging from 9284 B to 16788 B. As expected, the amount of required memory increases for large data structures.

In the case of having asynchronous choreographies, the amount of consumed memory increases because the polling or initial interaction adds an extra overhead of 9248 B. As in the previous experiment related to execution time, the relative overhead of this mechanism is reduced as the amount of data transmitted in a choreography increases.

**C. Jitter and synchronization**

A different kind of experiments was carried out to assess the level of jitter in the activation of the master and slave methods. This is particularly relevant for `T` and `S` choreographies giving an idea of the achievable precision in the related time synchronization with a simple infrastructure with parsimonious resources. To this end, the testing infrastructure was enlarged with more three similar PCs, one master and four slaves.

At the master side, the main lesson learnt is related to the way the underlying operating system manages activations generated by timers. With older kernels such as the 2.4.22 that we used, the temporal resolution is defined by compilation at installation time, being typically 10ms, 4ms or 1ms. In our case the resolution was set to 10ms, thus any periodic activity generated by the kernel is achieved with 10ms activations with skips to reach the desired value on average. This also means that any other value that is not integer multiple of the kernel resolution will suffer large jitters (topmost part of Figure 18). Conversely, for periods that are integer multiples a rather precise activation is possible.

Thus, we adapted the activation of any thread to the nearest smaller integer multiple of the kernel temporal resolution, resulting in an impressively low jitter of ±50µs. This simple technique
uses more bandwidth due to the use of potentially shorter periods than needed, in the exchange for higher precision. Alternatively, more recent 2.6.x kernels could be used, which already come with a real-time patch that supports activations with any periods and relatively high precision.

For the purpose of our work, the most important result is that it is indeed possible to achieve an activation jitter below 100μs despite the high complexity of the underlying software architecture.

The main lesson learnt concerning the slaves is that, in order to have time accurate synchronizations or activations, it is necessary to eliminate the accumulated jitter which is typically generated by the execution of a set of choreographies in sequence. Even in simple set-ups, like the one shown at the bottom of Figure 18, namely using four S-choreographies with the same scheduling parameters: \((O_i = 0, D_i = T_i = 20\text{ms}, C_{\text{max}} = 2\text{ms}, \forall i = 1..4)\) the amount of accumulated jitter may grow to ±200μs at the 4th slave. Nevertheless, carrying out a simple correction in the master in the offset of the S-choreographies allows decoupling them in time, avoiding the jitter accumulation and improving the quality of the synchronization with jitters below ±100μs.

### VIII. Two use cases

To illustrate the use of the synchronous scheduling service, two simple applications are analyzed in detail along this section. The first is a producer-consumer application with an end-to-end transmission deadline. The second is a distributed application in which two nodes need to share a common clock to carry out a temporal synchronization. Particularly, the first application illustrates how to use C and P choreographies or alternatively two asynchronous choreographies (AC and AP) to communicate between two nodes in a distributed application. The second example looks at the use of S or T choreographies to maintain certain inter-slave temporal synchronization using the synchronous scheduling service as reference clock. In both application, the maximum transmission delays of the network are bounded and known.

#### A. A producer consumer distributed application

Figure 19 shows a very simple use case of a synchronous scheduling service: a producer-consumer application. In such an application new data is generated by a producer node every 10ms and have to be transmitted to the consumer node in no more than 20ms.

1) Approach with C and P choreographies:

We firstly follow a synchronous approach in which two choreographies are sufficient to model the system:

- one named \(PC\) that pulls the data from the producer into the master,
- and another named \(CC\) that pushes the data from the master to the consumer.

To use the synchronous scheduling service, one needs identifying the period \((T)\) and a deadline \((D)\) for each choreography. In our current case, the period is directly deduced from the description of the system, i.e., \(T_{PC} = T_{CC} = 10\text{ms}\). The deadline \((D)\) of each choreography is not so direct to deduce
because the only constraint is $D_{PC} + D_{CC} < 20ms$. There are several schemes to breakdown an end-to-end deadline constraint in individual deadline constraints that can be applied to each component of a given transaction, e.g., even distribution, weighted based on cost, weighted based on load, etc. For the sake of simplicity, we consider an even breakdown in this case, thus leading to $D_{PC} = D_{CC} = 10ms$.

The cost of transferring data from a client to a server varies with the network and processor characteristics. In this case, it is considered that a cost of 2 milliseconds is enough to send or receive data to or from the master. Thus, in the current example: $C_{CC} = C_{PC} = 2ms$.

The settings of the offsets have a considerable impact in the end-to-end latency of a transaction. Typically, one choreography should be activated right after the termination of the preceding choreography in a transaction. In a trivial scenario like the one considered here, the minimum end-to-end delay can be achieved making $O_{CC} = O_{PC} + C_{PC}$. Nevertheless, any offset used in this case will always meet the end-to-end requirement of 20ms. Thus in our example we used $O_{PC} = 5ms$ and $O_{CC} = 10ms$.

After defining the choreographies with the parameters referred above, they must be allocated to the respective nodes and the master must be correspondingly configured to forward the data from the producer to the consumer (Figure 20). The code of the master (Figure 21) shows how the master creates and initializes each choreography ($pCh$ and $cCh$). Each choreography is configured with release parameters (lines 16 and 34 respectively) and the target slave (lines 30 and 36 respectively). To enable the transfer of data among the two nodes, the code of the choreographies overwrites preTransmission and postTransmission respectively. Both methods share an attribute (buffer) which contains the last produced data.

The two slaves (producer and consumer) also require certain application logic to produce and consume data from the master. Figure 22 shows the code executed by the producer and Figure 23 details the code of the consumer. In order to be discovered, both slaves register themselves in the registry (Figure 22 - 05 and Figure 23 - 05 ) before starting to transmit (Figure 22 - 10) or receive (Figure 22 - 10) data. Once registered, the master discovers them (Figure 21 - 19 and 35) using the RMI registry service.
between these two activities, i.e., the producer runs unsynchronized with respect to the master, this interval can be as large as $T_{PC}$. Thus, in this example, the end-to-end delay can be as large as 17 ms, which is still below the specified deadline of 20 ms.

Fig. 24. Message sequence chart of the producer-consumer application using synchronous choreographies

2) Approach using AC and AP choreographies: In those applications that do not generate data periodically the use of asynchronous choreographies instead is more efficient, thus we show here how the producer consumer application can be developed using the asynchronous choreographies APC and ACC.

In this example, the characterization of APC is similar to the characterization of PC with the following scheduling parameters $O_{APC} = 5 ms, T_{APC} = D_{APC} = 10 ms$, and $C_{APC} = 2 ms$.

The characterization of ACC requires a further explanation because its cost maybe higher than that of its CC equivalent since an extra pair of messages is needed for the master to inform the slave whether new data is available. If data is present, the transaction will then include the actual data transfer as in CC.

Thus, we will consider a worst case execution time of 3 ms, i.e., the 2 ms as in CC plus 1 millisecond for the extra initial interaction. The ACC scheduling parameters are then $O_{ACC} = 10 ms, D_{ACC} = T_{ACC} = 10 ms$, and $C_{ACC} = 3 ms$.

Figure 25 shows the case when the producer has no data to transmit. Note the shorter execution times given the absence of the actual data transfer.

Fig. 25. Producer-consumer application using asynchronous choreographies: case without data transfer

Figure 26 shows the case when the producer has actually produced some data. This data is retrieved by the master at $t = 5 ms$ and stored internally at $t = 7 ms$. At $t = 10 ms$ the master informs the slave that the data is ready so that the slave can retrieve it. The data becomes available at the slave side at $t = 13 ms$. As in the synchronous case, the worst case end-to-end delay since the data is actually generated by the producer until it

---

Fig. 21. Source code for the master in the producer-consumer application using synchronous choreographies

```c
/* Snippet of code that shows how a master creates a communication with two slaves (producer and consumer) using choreographies (P and C).
   Both slaves are located through the registry of RM.
   Start of the code of the producer application using synchronous choreographies:
*/
01: Master ma = new Master(); //Creation of the master
02: Object buffer; //Creation of an auxiliary variable
03: Object of = new Object(); //Creation of the first choreography
04: PC = new PC(buffer);
05: Overwriting the data is going to be stored.
06: void postTransmission(Object obj){ //invoked at runtime by the scheduler
07: BufferOpt(); //attracts the data which is going to be transferred
08: }
09: //Defining the properties of the choreography
10: PeriodicParameters pp = new PeriodicParameters();
11: new AbsoluteTime(10,0), //StartTime + 5ms
12: new RelativeTime(10,0), //Period=10 ms
13: new RelativeTime(12,0), //Count=2 ms
14: new RelativeTime(10,0), //Deadline=10 ms
15: null, null, //No handlers
16: pc.setReleaseParameters(pp); //Looking up the first node
17: Registry registry = LocalRegistry.getRegistry();
18: FTSLave.produceProc(); (FTSLave.registry.lookup("producer");
19: master.subscribe(produceNode, pc);
20: Creation of the second choreography
21: ACC = new ACCChoreography();
22: Overwriting which data is going to be transferred
23: Object postTransmission();
24: return Buffer;
25: 
26: PeriodicParameters pp = new PeriodicParameters();
27: new AbsoluteTime(10,0), //StartTime + 10ms
28: new RelativeTime(10,0), //Period=10 ms
29: new RelativeTime(12,0), //Count=2 ms
30: new RelativeTime(10,0), //Deadline=10 ms
31: null, null, //No handlers
32: acc.setReleaseParameters(pp);
33: FTSLave.consumeProc(); (FTSLave.registry.lookup("consumer");
34: master.subscribe(consumeNode, acc);
35: Starting the service
36: master.start();
*/

Fig. 22. Source code for the producer in the producer-consumer application (synchronous choreographies)

```c
/* Snippet of code that shows how a producer creates a communication with the consumer
*/
01: Source code of the producer application (synchronous choreographies)
02: Creation of the producer slave
03: Binding the producer
04: Registry registry = LocalRegistry.getRegistry();
05: registry.bind("producer", producer);
06: Producing logic
07: 
08: waitUntilNextPeriod(); // It waits 10 ms
09: Object producedData = produceSomeData(); // It generates the data
10: slave.produceData(new FIOTData(producedData);
11: while(true);
*/

Fig. 23. Source code for the consumer in the producer-consumer application (synchronous choreographies)

```c
/* Snippet of code that shows how a consumer creates a communication with the producer
*/
01: Source code of the consumer application
02: FTSLave consumeProc(); new FIOTSlave();
03: Binding the object
04: Registry registry = LocalRegistry.getRegistry();
05: registry.bind("consumer", consume);
06: Consumer logic
07: 
08: waitUntilNextPeriod(); // It waits 10 ms
09: FIOTData data = slave.getFIOTData();
10: Object obj = data.getData();
11: consumeData(obj); // It consumes the produced data
12: while(true);
*/

example, the application at the producer side generates a datun labeled 10 at $t = 4 ms$. At $t = 5 ms$ the master triggers the PC choreography that pulls such data into the master itself, where it becomes available at $t = 7 ms$. At $t = 10 ms$ the master triggers the CC choreography that pushes such data into the consumer, where it becomes available at $t = 12 ms$.

The end-to-end delay related strictly with the master relaying mechanism is $7 ms$. However, the total end-to-end delay concerning the actual data transfer must also account for the interval between the data generation by the producer application and that data being pulled by the master. If there is no synchronization
becomes available at the consumer is given by a producer polling period $T_{APC}$ plus the transaction maximum duration that in this case is $8\text{ms}$, leading to a total of $18\text{ms}$, which is still below the specified deadline.

Note that the absence of data to transfer releases bandwidth that the master can use to carry out other asynchronous transactions. This possibility to reclaim unused bandwidth can have a strong positive impact in the efficiency of the system in the case of large asynchronous data transfers.

B. Synchronizing two nodes

In this application the purpose is to execute two methods in two different nodes in a synchronized way. For this end, we use the scheduling service to create a a common timeline. The first method has to be activated at $t = 5\text{ms}$, $15\text{ms}$, $25\text{ms}$ and so forth. The second has the same period but its initial activation is delayed until $t = 15\text{ms}$ (Figure 27).

![Fig. 26. Producer-consumer application using asynchronous choreographies: case with data transfer](image)

**Fig. 26.** Producer-consumer application using asynchronous choreographies: case with data transfer

1) Using two $T$ choreographies:

In a first approach we will use two trigger choreographies, $T1C$ and $T2C$, with scheduling parameters trivially deduced from the specification, i.e., $T_{T1C} = T_{T2C} = D_{T1C} = D_{T2C} = 10\text{ms}$ and $C_{T1C} = C_{T2C} = 1\text{ms}$. The offsets require some care in order to have the execution of the methods in the slaves triggered at the desired instants. In fact, these choreographies have an extra parameter, i.e., the trigger delay, which must be considered together with the offset. Thus, we consider $O_{T1C} = 0$ with $delay_{T1C} = 5\text{ms}$ and $O_{T2C} = 5\text{ms}$ with $delay_{T2C} = 10\text{ms}$ (Figure 28).

Figure 29 illustrates the first $15\text{ms}$ of the system execution with the methods in the slaves being triggered at the desired instants.

![Fig. 27. Example of an application with temporal synchronization requirements](image)

**Fig. 27.** Example of an application with temporal synchronization requirements

2) Using two $S$ choreographies:

Another alternative to cater for the activation requirements of the application is to use two synchronization choreographies $S1C$ and $S2C$, one for each slave, to maintain a global timeline. Note that these choreographies are only used to synchronize slave clocks with the master clock. Then, it is up to the slaves to use such global timeline to execute their methods in a coordinated way. Therefore, the period of the $S$ choreographies is related with the desired precision and algorithm for the clock synchronization and not directly with the invocation period of the methods to execute.

In this particular case we used $T_{S1C} = D_{S1C} = T_{S2C} = D_{S2C} = 100\text{ms}$, an interval of time during which the relative drifts of master and slave clocks should be negligible compared with the jitters introduced by the middleware infrastructure (Section VII-C). The execution time is also $O_{S1C} = O_{S2C} = 1\text{ms}$.

The offsets in this case should be such that the choreographies are executed before the desired start time for the slaves methods to give time for the global timeline to be established. In this case we used $O_{S1C} = 0$ and $O_{S2C} = 1\text{ms}$. Thus, as shown in Figure 30, after $t = 2\text{ms}$ both slaves have a local estimation of the master clock $(mc)$ which they use to then trigger their methods internally.

![Fig. 29. First execution of each $T$ choreography and respective method](image)

**Fig. 29.** First execution of each $T$ choreography and respective method

**Fig. 28.** Satisfying activations with two $T$ choreographies

IX. CONCLUSIONS AND FUTURE WORK

The next generation of distributed real-time Java technologies may take advantage of the results generated in the scope of distributed real-time systems. Many such results can be easily integrated in the core of distributed real-time Java without excessive efforts, defining only the additional interfaces for the programmer. The flexibility of Java language and its distribution middleware offered by the virtual machine and RMI make the process even easier than in other platforms. However, this process is not always trivial and it requires additional efforts to adapt the existing techniques to the new scenarios.
In this paper we proposed and analyzed a synchronous scheduling service for the distributed real-time Java. It is based on the Flexible Time-Triggered communication paradigm adapted to the unicast/transactional environment provided by the DREQUIEMI middleware. This service allows coordinating a distributed application from a central point, the master, which controls the activation of the different tasks in the system. This can be particularly well suited to applications with low jitter requirements as well as to cope with dynamic reconfigurations with timeliness control. The experimental results are very promising in terms of temporal predictability while being parsimonious in terms of dynamic memory requirements, i.e. a few kilobytes.

Our ongoing work is directed towards the full characterization of new types of choreographies. Particularly, we wish to support choreographies that correspond to COTS components, particularly services, and to support their dynamic composition following a similar approach to the one proposed in [12].

REFERENCES


