Chapter 2
Using Real-Time Java in Distributed Systems: Problems and Solutions

Pablo Basanta-Val and Jonathan Stephen Anderson

Abstract Many real-time systems are distributed, i.e. they use a network to interconnect different nodes. The current RTSJ (*Real-time Specification for Java*) specification is insufficient to address this challenge; it requires distributed facilities to control the network and maintain end-to-end real-time performance. To date, the real-time Java community has produced the DRTSJ (*The Distributed Real-Time Specification for Java*), though a wide variety of other distributed real-time Java efforts exist. The goal of this chapter is to analyze these efforts, looking for the problems they addressed and their solutions, and defining (when possible) their mutual relationships.

2.1 Introduction

The next generation real-time systems are likely to be cyber-physical [73, 246], meaning that they are more distributed and embedded than before and have a coupled interaction with physical systems. From the point of view of real-time Java, it is not sufficient to rely solely on centralized timeliness specifications like the RTSJ (*Real-time Specification for Java*) [65]. Programmers require solutions that allow interconnection of these isolated virtual machines to offer real-time support on a myriad of networks and applications. The kinds of networks that should be addressed include, among others, automotive networking, industrial networks, aviation busses, and the Internet. Each system has different domain challenges.

P. Basanta-Val (✉)
DREQUIEM Lab. Universidad Carlos III de Madrid, Avda. de la Universidad 30
Leganés-Madrid, 28911, Spain
e-mail: pbasanta@it.uc3m.es

J.S. Anderson
Virginia Tech., ECE Department, Blacksburg, VA 24061, USA
e-mail: andersoj@anderson.org

M.T. Higuera-Toledano and A.J. Wellings (eds.), *Distributed, Embedded and Real-time Java Systems*, DOI 10.1007/978-1-4419-8158-5_2,
© Springer Science+Business Media, LLC 2012

The evolution of centralized and distributed real-time Java specifications has been conducted at different rates. Whereas there are commercial implementations for centralized systems based on the RTSJ, distributed systems still lack this kind of support.

On the one hand, centralized systems have an important infrastructure; they can use to develop their real-time Java applications using a common specification: the RTSJ (Real-Time Specification for Java). Designers can build their systems using Oracle's Java RTS implementation [64, 399], IBM's WebSphere RT [213], Aicas' Jamaica [372] or Apogee's Aphelion [16]. Each implementation extends the normative model in one or more ways. For instance, many of the aforementioned commercial implementations support real-time garbage collection, one feature that it is not mandatory in the RTSJ; many have plug-ins for common IDEs. There are also experimental platforms that include JTime [410] (the RT-JVM implemented as the reference implementation for the RTSJ), and free source-code implementations like JRate [108, 111], OVM [153], and Flex [327].

On the other hand, distributed real-time application practitioners suffer from a lack of normative implementations and specifications on which they may develop their systems. Most efforts toward a distributed real-time specification address one of two different technologies: Java's RMI (*Remote Method Invocation*) [398] and OMG's RT-CORBA (*Real-Time Common Object Request Broker Architecture*) [344]. Neither process is complete (i.e. no specification and implementation that developers may use have emerged).

For RMI, a natural choice for pure Java systems, the primary work has been drafted as a specification under the JSR-50 umbrella. The name given to the specification is the DRTSJ (*Distributed Real-Time Specification for Java*) [10, 225]. Unfortunately, the work is unfinished, inactive, and requires significant effort to produce implementations similar in quality and performance to those available in the RTSJ. In a wider context, some partial implementation efforts have been carried out in some initiatives (e.g. RT-RMI-York [72], RT-RMI-UPM [408], and DREQUIEMI-UC3M [43]) to test different architectural aspects; however, current implementations cannot be used in their current form in commercial applications.

RT-CORBA [344] was mapped to RTSJ [13, 219, 220] to take advantage of the real-time performance offered by the real-time virtual machine. The mapping is far from trivial because many features of RTSJ have to be combined with the real-time facilities of CORBA which offer similar support. The main implementation supporting RTSJ and RT-CORBA mapping [139, 314], RTZen, has produced a partial implementation which cannot be used in its current form to develop real-time applications, and requires additional implementation effort to produce a commercial product.

Apart from these two control-flow efforts there are other options for distributed real-time Java based on emerging standards like DDS [292] (*The Data Distribution Service*). For instance, the integration of distributed Java with the real-time distribution paradigm is being lively investigated in the context of the iLAND

project [24, 166, 168]. Many practical distributed real-time systems implemented with RTSJ include DDS products like RTI's DDS [336] or PrismTech's Open Splice [308].

The rest of this chapter explores previous work in differing depth depending on importance and the amount of publically accessible information. Before entering into the discussion of the main work carried out in distributed real-time Java, the chapter analyzes different Java technologies (Sect. 2.2) that may be used to support Java distribution. After that, it focuses the attention on DRTSJ and other real-time RMI (RT-RMI) approaches (Sect. 2.3). Then, extra support for distributed real-time Java that ranges from the use of specific hardware to the use of formal methods and multi-core infrastructure are discussed (Sect. 2.4). The chapter continues with a set of technologies that augment distributed real-time with additional abstractions and new services (Sect. 2.5). The chapter ends with conclusions and future work (Sect. 2.6).

2.2 Approaches to Distributed Real-Time Java

The section explores different alternatives for distributed real-time Java from a technological perspective identifying a set of candidate technologies. The leading effort, DRTSJ, is included as part of the discussion. For each category, the authors analyze advantages and inconveniences stemmed from the use of each technology.

Although there are many distributed abstractions and categories, from the point of view of real-time systems, they may be classified at least in the following types [226, 429, 430]:

- *Control-flow*. This category includes systems that use method invocation in simple request-response fashion. Java's RMI (*Remote Method Invocation*), remote invocations or any RPC (*Remote Procedure Call*) belong to this category. The basic idea of them is to move both application data and the execution point of the application among remote entities.
- *Data-flow*. This category includes the publish/subscribe model that is supported in technologies like the OMG's DDS [292]. The goal of DDS is the movement of data without an execution point among application entities following publisher/subscriber patterns which communicate through shared topics.
- *Networked*. Comprises systems that exchange asynchronous or synchronous movement of messages, without a clear execution point among their entities. IPC and message passing mechanisms included in many operating systems fall in this category. Java's JMS (*Java's Message System*) [395] could be included in the list too.

The list of distributed models for Java is more extensive, including: (1) *mobile objects*, (2) *autonomous agents*, (3) *tuple-spaces*, and (4) *web-services* (see [61]).

A comprehensive distributed real-time Java need not provide all of them; any (networked, data-flow, or control-flow) is powerful enough to develop a straightforward technology.

The impact of the three models on Java is quite different and it is in an uneven state. Control flow solutions, like DRTSJ and other RT-RMI technology, are the most addressed approaches because they were integrated previously in other technologies like RT-CORBA.

Data flow approaches are another alternative for building a real-time technology (with technologies like DDS). However, despite significant practical use, they have not been addressed by the real-time Java community in depth. Rather, the community has focused mainly on control-flow abstractions.

The use of networked solutions provides flexibility but requires cooperation from higher-level abstractions which should instead deal with end-to-end timeliness. Nevertheless, networked solutions are one of the most used approaches with several specific proprietary frameworks.

Other Java's technologies like Jini [146], JavaSpaces [401], and Voyager [319] offer augmented distribution models that enhance current infrastructures but they are silent about real-time issues. In many cases, these technologies require a predictable end-to-end communication model on which to build their enhanced models. For instance, Jini offers distributed service management (with three stages `discovery`, `join`, and `leasing`) in which communications rely on RMI. JavaSpaces offers an abstraction similar to object oriented databases (without persistence) with `read`, `write`, and `take` primitives which have to be accessed from remote nodes. Voyager supports mobile code and agents that may migrate in the platform at discretion among several virtual machines that communicate agents.

In the three cases described, the challenge is to characterize the augmented abstraction from the point-of-view of a real-time infrastructure. For instance, in one possible RT-Jini [167] the services could publish their real-time characterization which could be used in the discovery process by clients that require access to remote services. However, other alternative RT-Jini approaches may offer bounded discovery processes and enhanced composition.

Among these, three distributed technologies are of special interest for distributed real-time Java: RT-CORBA, RMI and DDS. The rest of this section analyses pros and cons of using each one of these three technologies when they are used as basic support for distributed real-time Java.

2.2.1 Main Control-Flow Approaches to Distributed Real-Time Java

Assuming a control flow abstraction, an implementation for distributed real-time Java may chose among two primary technologies: RT-CORBA and RMI.

2.2.1.1 RT-CORBA and RTSJ

The RT-CORBA standard enhances CORBA for real-time applications. The specification is divided into two parts: RT-CORBA 1.0 for static systems and RT-CORBA 2.0 for dynamic environments. One way to provide distributed real-time Java is to map these two models to RTSJ, as has been done previously for many other programming languages. For instance, a similar mapping has been carried out before for Ada and C/C++.

Defining a mapping from RT-CORBA to RTSJ is not as simple as it seems at a first glance. One may think that it is a simple process but it requires some kind of reconciliation between RT-CORBA and RTSJ in order to match and map abstractions for real-time parameterization, scheduling parameters and underlying entities.

For instance, using RT-CORBA with RTSJ, the programmer has two alternatives to control concurrency: one given by the RT-CORBA's mutex and another by using the `synchronized` statement and monitors included in RTSJ.

Choosing one or another alternative is equivalent to select one of the following target infrastructures:

1. RTSJ with a remote invocation facility given by RT-CORBA.
2. RT-CORBA enhanced with the predictability model of RTSJ.

The first choice corresponds to using the `synchronized` statement and the second to use a RT-CORBA's mutex (and probably discard the `synchronized` statement of Java).

Both approaches have implications from the point-of-view of the programming abstraction and architecture. The first choice distorts the abstract programming model of RT-CORBA, which is trimmed for a particular purpose. The second alternative is more conservative and maintains RT-CORBA interoperability among different programming languages and platforms, partially removing some support given by Java and the RTSJ.

In the second case, the price paid for it is high because many good properties of RTSJ have to be discarded. One example of this is the dual-thread model based on `RealTimeThreads` and `NoHeapRealtimeThreads` [65]. RT-CORBA only defines one kind of thread whereas RTSJ has two threads: real-time and non-heap threads.

The RTZen project [139] followed the second choice improving the internals of the ORB using RTSJ facilities. The project has generated also specific tools and programming patterns that may be used to build real-time systems. Among the main contributions carried out to distributed real-time Java are:

- A CORBA architecture that is compatible with the RTSJ [242, 314]. This architecture removes the garbage collector from the internals of the middleware using regions inside its internal architecture. However, it still maintains the GC's interference in the endpoints of the communication.

- Programming patterns [313] which are used in the implementation of a middleware and in other general programming abstractions.
- Tools for configuring applications [174]. They allow designers to select a subset from the whole middleware and reduce the application footprint.
- A component model (i.e., COMPARES [174, 175, 208]) for distributed real-time Java which relies on the RT-CORBA's resource model. It is based on the CCM (CORBA's Component Model) [285].

2.2.1.2 RMI and RTSJ

RMI [398] is the main distributed object abstraction for Java. It offers remote invocation and other services like connection management, distributed garbage collection, distributed class downloading and a naming service. Some of these services, like naming and connection management, are also available in the CORBA world; while others (class downloading and distributed garbage collection) have a clear Java orientation. This extra characterization reduces the development time in comparison to other alternatives.

The Java orientation of these services brings in advanced features like system deployment, automated update and avoids remote object memory leaks in distributed real-time systems. However, all these services are sources of unbounded interference and their interaction with the real-time Java virtual machine and runtime libraries must be better specified in order to avoid unwanted or unpredictable delay. This indeterminism has induced many researchers to avoid or forbid their use in specific environments like high-integrity applications.

Another interesting feature of RMI is the lack of a real-time remote invocation model that may produce a seamless integration with RTSJ. The remote objects may be characterized by using the model provided by RTSJ which includes scheduling, release, and processing-group parameters. From the point of view of a base technology for distributed real-time Java, a RMI-and-RTSJ tandem should be more synergic than a RTSJ-and-RT-CORBA solution.

It should be noted that the RMI-and-RTSJ in tandem approach is close to the first choice in the integration of RTSJ-and-RT-CORBA. In both cases the communication technology (i.e., RMI and RT-CORBA) is modified in order to be adapted to the model proposed by RTSJ. However, even in this case, RMI is closer to Java than RT-CORBA because RMI extends the Java's object model with genuine services like distributed garbage collection and class downloading, which are not available from CORBA.

2.2.2 A Data-Flow Approach: DDS and RTSJ

Alternatively, distributed real-time Java may be implemented with DDS-and-RTSJ in tandem, defining a distributed real-time technology based on these two technologies.

DDS offers another alternative to build real-time systems under the publish-subscribe paradigm with certain guarantees in message transmission and reception using *topics* and providing a quality-of-service model for messaging. However, it is unclear how to provide end-to-end Java abstractions on top of a publisher-subscriber abstraction efficiently. Such systems today require substantial developer intervention and explicit context management in application code to provide end-to-end abstractions and mechanisms for resource management and timeliness assurance.

One solution is to build simple request-response abstractions on DDS, similar to those available in RMI, or use isolated communication channels – as proposed in ServiceDDS [128] and the iLAND project [168, 214] – to replicate a synchronous communication. However, neither ServiceDDS nor the iLAND project has defined integration of RTSJ as its main goal. Both consider RTSJ to be a suitable programming language for use in their developments; however, the integration between RTSJ and DDS is not its primary goal.

Another solution is to provide optimized end-to-end models to run under the real-time publisher-subscriber paradigm. The authors consider this an open challenge in the distributed real-time Java context.

2.3 Distributed Real-Time Java

This section considers the main approaches defined for distributed real-time Java. It first describes the leading approach, DRTSJ, and then concentrates on other RT-RMI frameworks and approaches that are related to DRTSJ.

2.3.1 DRTSJ: The Distributed Real-Time Specification for Java

The leading effort is DRTSJ, its main goal is to enhance RTSJ with trans-node real-time end-to-end support, using as distribution mechanism RMI. In its initial position papers [10, 225, 226, 429, 430], DRTSJ is defined to have two goals:

- The primary aim is to offer end-to-end timeliness by guaranteeing that an application's trans-node has its timeliness properties explicitly employed during resource management. Provide for and express propagation, resource acquisition/contention and storage and failure management in the programming abstraction.
- The second goal is to enhance the programming model with control flow facilities that model real-time operations composed of local activities. The activity-control actions include: *suspend*, *abort*, *changes propagation*, *failure propagation*, *consistency mechanisms*, and *event notification*. The result of this activity is distributable real-time threads.

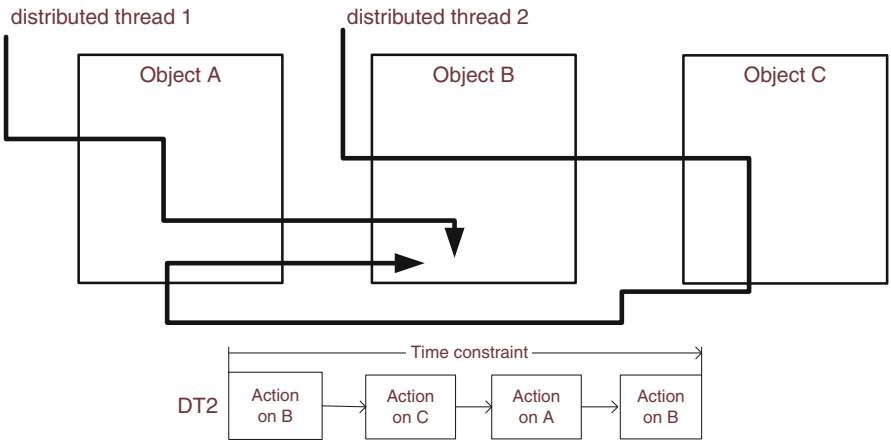


Fig. 2.1 Distributable threads used as an end-to-end flow abstraction

Among the general guiding principles of DRTSJ are:

- Bring distributed real-time to Java.
- Distributed objects and operations.
- Maintain the flavor of RTSJ.
- Not dictate the use of an RTSJVM for every node, nor any specific real-time transport.
- Provide coherent support for end-to-end application properties.

2.3.1.1 Distributable Real-Time Threads

The foundational abstraction for DRTSJ is the distributable thread, which is a single logical thread with a globally unique identity that extends and retracts through local and remote objects (see Fig. 2.1). Distributable threads extend and retract performing remote invocations (RPC's) to remote objects via RMI, and maintain an effective distributed stack.

It is noteworthy that distributable threads may have end-to-end time constraints, typically deadlines. However, other policies may be enforced in each segment through specific schedulers. Distributable threads are expected to carry their timeliness contexts (e.g., deadlines, priorities, or more complex parameters) with them as they extend and retract through the distributed system in the same manner as local threads.

Non-trivial distributed systems must consider themselves to be continually in a state of (suspected or real) partial failure; therefore fault detection, consistency, and

recovery policies must be provided. DRTSJ anticipates providing these recovery policies, but also anticipates application-specific policies which provide suitable engineering trades between consistency, performance, and timeliness.

Specification

The DRTSJ's expert group has drafted an internal specification for DRTSJ. It deals with the changes required in the current RTSJ infrastructure, new scheduling policies, new programming entities and asynchronous communications. Although describing the whole model is out of the scope of this chapter, the authors highlight several low-level details that clarify some key issues:

- It requires changes to RTSJ classes and some modifications to Java in order to support distributable threads, specifically marking some RTSJ classes as `Serializable`.
- It does not include any particular new scheduler. Its default behavior is based on the priority based scheduler currently included in RTSJ.
- It defines `DistributableThread` as its main abstraction.
- It is silent regarding use of the memory model offered by RTSJ, specifically scoped or immortal memory.
- In the past, it provided distributed `AsyncEventHandlers` as a lightweight mechanism to transfer failures. This mechanism is not currently considered defined as a part of the specification.

Software Suite

The JSR-50 expert group also described a software suite that consists of a modified RTSJ virtual machine with support for:

- *Distributable threads* with support for real-time performance.
- *Thread integrity mechanisms* like orphan detection and elimination policies which detect partial failures.
- *An optional meta-scheduler* component which allows the use of arbitrary scheduling policies.

Specific Problems and Solutions

Some of the core members of the JSR-50 Expert Group carried out a series of contributions to distributed real-time Java which are focused on the definition of scheduling algorithms for distributable threads:

- *Implementation of Distributable Threads via local proxy threads*, solving the ABA deadlock problem by servicing all local distributable thread segments using a single thread with re-entrant state management.
- *Scheduling algorithms for distributed real-time Java based on distributable threads*. Their contributions to this topic [10,11,27,27,316–318] use distributable threads as the main programming abstraction. The list of algorithms developed includes DUA-CLA [11], and ACUA [160]. DUA-CLA is a consensus-driven utility accrual scheduling algorithm for distributed threads, which detects [317] system failures and proposes recovery mechanisms for distributable threads [115, 318]. ACUA [160] is a distributed scheduling algorithm designed under a partially synchronous model; it allows for probabilistically-described message delays.
- *Distributable Thread Integrity policies*, sometimes called Thread Maintenance and Repair (TMAR). Building on precursor work from the Alpha and Mach projects, a number of timely thread failure and consistency management protocols have been proposed [173, 317].

2.3.1.2 Different Integration Levels

The position paper for DRTSJ was refined in [429] defining three integration levels (L0, L1 and L2). Each level requires some support from the underlying system and offers some benefit to the programmer.

- **L0** is the minimal integration level. The level considers the use of RMI without changes; this is its main advantage. In L0, applications cannot define any parameters for the server-side and a predictable end-to-end remote invocation is not feasible. The remote invocation should be used during initialization or non time-constrained phases of the distributed real-time application.
- **L1** is the first 'real-time' level. L1 requires mechanisms to bind the transmission and reception of messages and some kind of real-time remote invocation. To achieve this goal, DRTSJ would extend the remote object model with real-time characterization, and modified development tools. In L1, the remote invocation to a remote method creates objects that may be invoked from remote clients offering a predictable real-time remote invocation. For L1, additional changes are required in the serialization of some classes to be transferred through the network. As a result of this design, client and server do not require a shared clock (i.e., using clock synchronization) and failures in the server are propagated to the client as a result of the remote invocation (i.e. synchronously).
- **L2** offers a transactional model for RMI. The transactional model is based an entity called distributable real-time threads; these entities transfer information from one node into another. L2 requires clock synchronization among RMI nodes and changes in class serialization in some Java classes. The model also offers asynchronous events that provide an asynchronous mechanism and distributed asynchronous transfer-of-control. Using this support the server may notify changes to clients asynchronously.

2.3.2 Architectures for RT-RMI

Another set of works have targeted RT-RMI from a different angle producing a real-time version for RMI similar to RT-CORBA without having distributable threads as the main entities. This section introduces three frameworks: DREQUIEMI, a previous framework designed at the University of York, and a third work carried out at Universidad Politcnica de Madrid.

2.3.2.1 DREQUIEMI

Researchers at Universidad Carlos III de Madrid worked on extensions for distributed real-time Java [30, 43]. The resulting framework is named DREQUIEMI. DREQUIEMI integrates common-off-the-shelf techniques and patterns in its core architecture. The major influences to the model are RMI, the RTSJ, RT-CORBA, and some time-triggered principles. The reference architecture designed could be used to deploy distributed real-time threads on it and offers support for DRTSJ's L1. It also incorporates some elements taken from L2.

Its goal is to detect and analyze sources of unpredictability in a distributed Java architecture based on RTSJ and RMI. In DREQUIEMI many of the explored issues deal with a triple perspective: the impact on the programmer, the infrastructure and the run-time benefit stemmed from its adoption.

The current architecture has five layers (see Fig. 2.2):

- *Resources*. Collectively, they define the foundations for the architecture which includes the list of resources that participate in the model. The model takes from real-time Java two key resources: memory and processor and from RMI the network resource.
- *Infrastructure*. These three resources are typically accessed through interfaces (via an RTOS and a RT-JVM) shaping an infrastructure layer. In DREQUIEMI, the access to the infrastructure is given through a set of primitives. It is important to highlight that to be compliant with the model, the current RTSJ requires the inclusion of classes for the network. Fortunately, these classes are already available in standard Java (and in other packages), so that they may be added to a new network profile.
- *Distribution*. On top of this layer, the programmer may use a set of common structures useful for building distributed applications. There is one new type of element corresponding to each key resource, namely: *connection pool*, *memory-area pool*, and *thread pool*.
- Common services. There are four services:
  - A stub/skeleton service,
  - a DGC (Distributed Garbage Collection) service,
  - a naming service, and
  - a synch/event service.

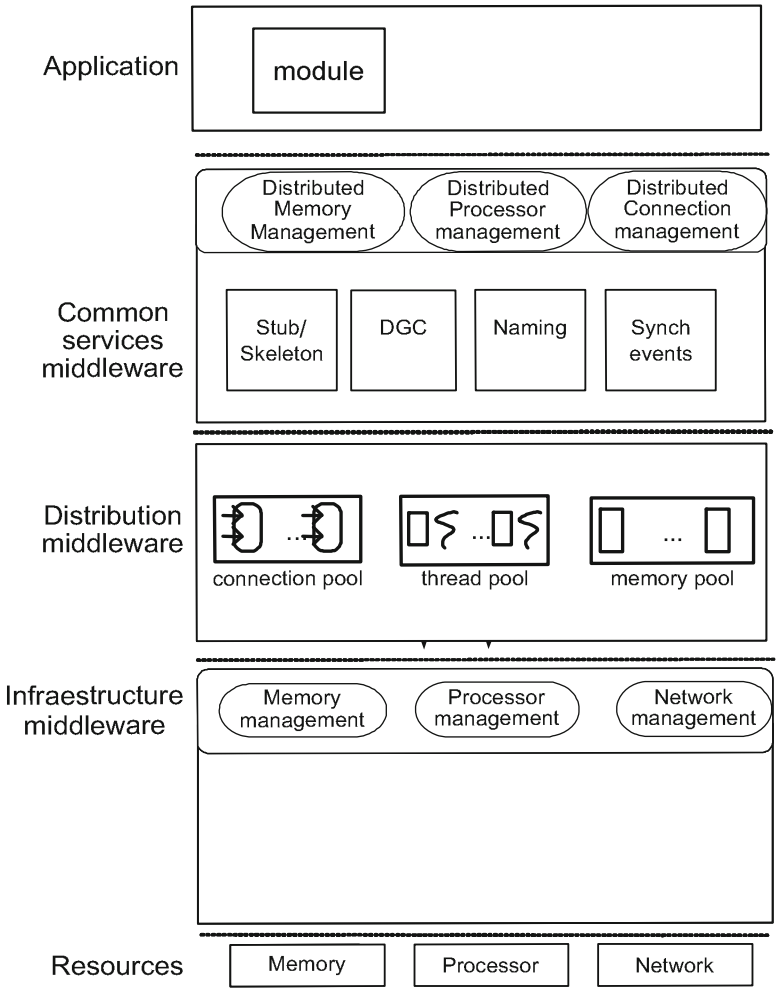


Fig. 2.2 DREQUIEMI's architecture for real-time distributed systems

The first allows the carrying out of 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) for data-flow communications.

- *Application*. Lastly, the specific parts of the application functionality are found at the uppermost layer, drawn as modules. The modules are based on components,

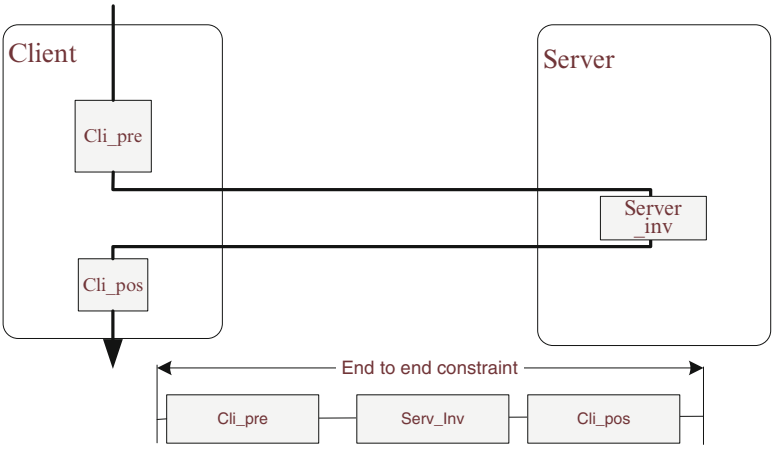


Fig. 2.3 End-to-end constraints in DREQUIEMI

promote reuse of pieces of other applications (see [44]) and may be supported by other augmented technologies.

The architecture defines two management levels:

- One centralized manager which handles memory, CPU and network resources at a centralized level (i.e. the real-time virtual machine).
- Another distributed manager which handles how these resources are assigned to distributed applications.

Scheduling and End-to-End Performance

From the point of view of end-to-end programming abstractions, in DREQUIEMI, there is no first class main entity such as DRTSJ's distributable threads. Instead, a real-time thread with certain deadline that invokes a remote method in a remote object establishes an end-to-end requirement for the application which comprises three stages (see Fig. 2.3):

1. The local client-side pre invocation (*Cli_pre*).
2. The server-side invocation (*Server_inv*).
3. The client-side post invocation (*Cli_pos*).

The characterization is enough to use well-known end-to-end scheduling models with precedence rules. Currently, the types of techniques addressed are based on end-to-end response time analysis for fixed priority-based scheduling (e.g., Direct Synchronization (DS), Release Guards (RG) and Sporadic Servers (SS) [181, 391, 392, 411]).

This model is also flexible enough to support other abstractions like the distributable threads described in the previous section on DRTSJ. However, its inclusion in DREQUIEMI requires extensions to the architecture to support new services similar to those defined for DRTSJ.

Specific Problems and Solutions

Using the DREQUIEMI framework, the following general distributed real-time Java issues have been addressed:

- *No-heap remote objects*. On this architecture [36, 39, 49] the authors designed a model, which is called no-heap remote object, to leave out the penalty of the garbage collector from the end-to-end path of the remote invocation. The model requires changes on the internals of the middleware and certain programming constraints called the no-heap remote object paradigm. From the point of view of distributed real-time Java, the no-heap remote object is useful to offer end-to-end real-time communications that do not suffer garbage collector interactions.
- *Non-functional information patterns*. Another problem addressed in DREQUIEMI is how to transfer non-functional information in distributed real-time Java. The authors identified different alternatives, evaluated their complexity and proposed a simple framework for priority-based systems [35, 45].
- *Asynchronous remote invocations*. The authors also analyzed the remote invocation model and proposed an asynchronous remote invocation which can be used to offer efficient signaling [42]. This mechanism avoids blocking at the client and requires changes in the transport protocol of RMI, namely JRMP.
- *Time-triggered integration*. Some real-time systems – like TTA (the Time Triggered Architecture) – use time-triggered communications to avoid network collisions and offer predictable communications. The authors developed a time triggered approach compatible with the unicast nature of RMI in [32] and [31]. By using it, programmers may increase the predictability of their communications.
- Protocol optimizations. The communication protocol of RMI, namely JRMP, has been optimized to offer efficient multiplexing facilities that reduce the number of open connections necessary to carry out a remote invocation [47, 48].

As a result of implementing RT-RMI with RTSJ, DREQUIEMI has revealed three optimizations for the RTSJ, namely: `RealTimeThread++` [34,41], `Extended-Portal` [40], and `AGCMemory` [37, 38]. The first extension generalizes the concurrency model of RTSJ proposing a unique type of thread that may be used by all applications; it also includes enhanced synchronization protocols (*Memory Ceiling Protocol* (MCP) and *Memory Inheritance Protocol* (MIP)). `Extended-Portal` offers an API mechanism able to violate the assignment-rule of RTSJ in a safe way, thus reducing the complexity of having to implement RT-RMI. Lastly, `AGCMemory` enhances the region model of RTSJ with a new type of region that enables partial object recycling.

2.3.2.2 A RT-RMI Framework for the RTSJ

Some members of the real-time systems group of York University are involved in the DRTSJ expert group and they carried out relevant work on how to develop a real-time RMI framework for DRTSJ's L1. In [69] and [72] they addressed the definition of a real-time RMI framework for RTSJ. The goal in this work is to build a model for the real-time remote invocation for RMI using the support given by RTSJ. The authors also defined a set of open issues related to the support for RMI.

For the remote invocation issue, the authors defined solutions for the client, the server and the network. The client-side remains almost unchanged while the server suffers several modifications. The authors proposed new classes to handle incoming remote invocations. At the server, the extra support added allowed defining scheduling parameters and a thread-pool which were attached to each remote object. The implementation considered assumes a TCP/IP network with a RSVP management infrastructure.

The framework supports propagation of parameters between client and server. The authors defined three policies to generate client-side parameters. These parameters are transferred to the server in each remote invocation. The set of parameters considered includes scheduling parameters and the relationship with the garbage collector.

The authors also considered memory issues regarding the implementation. At the client-side the implementation is easier than at the server, which requires the use of different types of memory.

Specific Problems and Solutions

The list of open issues described by York includes important issues related to certain L1 features and others stemmed from L2 requirements:

- *Class downloading*. RMI can download classes transparently from other virtual machines. However, this support introduces an important interference on the application that may have to wait for one of these classes to conclude its downloading. This is an important cost that should be removed from the remote invocation or at least be under application control.
- *Registry*. RMI has a naming mechanism that helps find other objects available in the system. However, the indirection given by this service is also another source of indeterminism and its integration in a real-time platform should be clarified.
- *Distributed garbage collector.* RMI extends the garbage collection support from a local virtual machine to multiple machines which recycle remote objects using a distributed garbage collector algorithm. However, the integration of this mechanism in a distributed architecture should be clarified in order to reduce its interference in real-time applications.
- *Object serialization*. RMI uses a general purpose object serialization model to marshal and unmarshal remote object parameters. From the computational point

overview, the mechanism is complex and its overhead for the remote invocation could be quite high.
- *Asynchronously interrupted exceptions* (i.e., AIEs). The support defined for L2 requires that one thread may raise an asynchronously interrupted exception in another. However, this action consumes network, memory, and CPU resources that have to be modeled and integrated in the core of RMI.
- *Asynchronous event handlers*. L2 allows a thread to raise an event that is handled in another node, requiring extra resources to carry out such an action. As in the previous case (AIEs) the platform should model and integrate the interaction of this interference in the distributed model.

2.3.2.3 The UPM Framework for RT-RMI

Another relevant contribution has been done by the Universidad Politecnica de Madrid (UPM) [408]. UPM worked in profile definition for RT-RMI and also addressed certain support issues related to improving the predictability and efficiency of RMI for real-time systems.

The authors considered that not all distributed real-time applications fit in the same distributed real-time Java profile; different applications may require different RT-RMI profiles.

Three different environments are considered:

- *Safety critical systems*. This type of system refers to systems where deadline misses can cost human lives or cause fatal errors. In this type of system, the behavior has to be correct and highly deterministic.
- *Business-critical systems*. This kind of profile refers to soft real-time systems whose anomalous behavior may have financial costs. In this kind of systems efficiency and robustness are extra-functional requirements.
- *Flexible business-critical systems*. The last profile is similar to the previous one but with a great deal of flexibility (e.g. future ambient intelligent systems and mission critical systems with survivability).

Four features define each profile:

- *Computational model*. The computational model refers to the abstraction used to design programs. In all cases, the abstraction chosen is a linear model described with an end-to-end predictable behavior. This end-to-end representation is used by the three models adjusting their computational features.
- *Adaptation required on RMI*. Each profile defines specific classes and hierarchies of objects for specific domains which have to be supported inside the core of the middleware.
- *Concurrency model in the remote invocation*. This feature refers to the rules that govern the handling of the remote invocation at both client and server.
- *Memory considerations*. Finally, each profile has different memory requirements and can use certain types of memory during its execution.

The first profile (RMI-HRT) is based on preemptive priority-driven scheduling and requires the use of priority ceiling protocols. It also considers two execution phases: initialization and mission; the first initializes the system and the second has real-time constraints. The profile defines the use of immortal memory for initialization and scoped memory in mission. An additional constraint introduced by the model in RMI is that clients cannot share the same reference to a remote object and dynamic allocation of objects is not allowed. The profile forbids the use of some dynamic mechanisms of RMI, removing among others: class downloading, firewalls, distributed garbage collection and security. RMI requires specific classes to handle real-time remote invocations and the interface compiler should be extended. The concurrency model defines handler threads at the server and a blocking thread at the client. The use of memory is constrained to regions – the garbage collector is disabled – and only one nesting level is allowed.

The second profile (RMI-QoS) has also end-to-end constraints. However, it does not have to rely on a worst-case response time analysis as RMI-HRT does. RMI-QoS accepts certain deadline misses if they fit in the quality-of-service pattern. The profile relies on the full-blown RTSJ and targets at soft-real-time applications. Its computational model has to support local and global negotiations. The computational model is also more complex than the previous; it includes: server initialization, reference phase, negotiation, and data transactions. Another key difference is that clients may share references to remote objects. The support given by the virtual machine has been extended with negotiations and specific classes to code remote objects. At the server side, the concurrency model has three threads: listener, acceptor and handlers. The memory restrictions placed on this model are fewer than in RMI-HRT.

The third profile considers a support based on OSGi with bundles that manage certain resources. Some features of RTSJ like the asynchronous interrupted exception are not considered in the framework. The profile may use real-time garbage collection to reduce the development cost of its applications.

Specific Problems and Solutions

A specific contribution has been done on the serialization currently available for RMI [407]. The authors distinguish two cases: one that transfers primitive types and another for references to Java objects. In the case of objects, the authors detected the problem of having acyclic structures which may require a complex analysis when they are used in a hard real-time system. The new mechanisms proposed have an impact on the serialization, which is extended with additional classes.

Many of the previous results described in the profiles have their origin in a previous article that defines solutions to make RMI real-time [124]. In this work the author based his solution on reservation protocols currently used in the Internet. The author also identified some indeterminism sources and proposed solutions for the client and server-side. As a result of his approach, the resulting RT-RMI architecture is more predictable.

2.3.3 Real-Time Component Models for RT-RMI

The list of RT-RMI related initiatives finishes with a component model designed at the Texas A&M University. As a part of a high mobility hard real-time component [321, 322] they proposed an approach for a real-time remote invocation server-centric service. The framework uses the total bandwidth server (TBS) algorithm to guarantee the allocation of CPU to tasks performed at the server side. The authors focus their analysis on CPU and are silent on network and memory management issues.

The impact of this work on distributed real-time Java is high. DRTSJ and RT-RMI may use the technique described, or another similar, to control and limit the CPU consumed at the server side.

2.4 Supplements for DRTJava

There are some supplements that enhance DRTJava with general support intended to offer enhanced support or simply better performance. The list included in this section refers to three functionalities: ability to use embedded and hardware platforms, the use of specific networks, and the use of formal models. They can be used in combination with many of the techniques described previously. For each of them the section explores key contributions carried out in each work and their specific contribution to distributed real-time Java.

2.4.1 Embedded Distributed Real-Time Java

Some pieces of work considered the issue of producing solutions for restricted performance and tight memory resources by using low-level communication facilities (e.g., [376, 377, 404]). Their work complements DRTJava with an embedded orientation and specific protocol communications.

One of these hardware optimized platforms is described in [405]. The platform is built on FemtoJava which was expanded with a RTSJ-based support. In this framework different Java nodes are connected using APICOM. This API provides communication via a network interface by using several services that establish connections, exchange messages, establish a logical local address, and broadcast messages. The model described in APICOM supports two main communication paradigms: point-to-point and publisher-subscriber.

Another alternative analyzed by the authors is the use of time-triggered models in their framework which could run on CAN [96] networks or the IEEE 802.154 wireless standard. These two communication facilities were integrated in hardware providing an interface to communicate Java nodes more efficiently.

2.4.2 Real-Time Networking

Implicitly, DRTSJ requires some predictability from the network: the messages exchanged among different nodes have to be time-bounded in order to offer bounded end-to-end response-time. Several authors have carried out different pieces of work that may contribute to this goal.

In order to enhance DRTSJ's communications there are two main approaches: local-area network integration and Internet communication facilities. The first includes integration of techniques such as time-triggered protocols (like the TTA [238]) and the second improves the predictability in packet transmission in open IP-based networks.

For the Internet approach, some pieces of work [46] addressed the definition of a technique to mark packets with a priority which is enforced in special routers using tools designed for Linux, once it is properly configured for hard real-time systems.

Another mechanism that has been analyzed in distributed real-time Java is RSVP, the resource reservation protocol of the Internet. In [124] the author defined a set of changes in RMI to use this transport protocol as part of his solutions for distributed real-time Java.

2.4.3 Multi-core Support for Distributed Real-Time Java

Many current infrastructures are multi-core systems with several processors that interconnect one each other using shared memory or have some kind of special bus that interconnect multiple cores. As a result of this change, current techniques defined for centralized real-time Java are being extended to profit from a multi-core infrastructure (see [284] or [426]). The changes have effects on current centralized infrastructures and scheduling algorithms that have to be redefined in order to consider multiple processors in a single chip and different core hierarchies.

From the perspective of distributed real-time Java two issues should be highlighted:

- The first is related to the scheduling techniques that both infrastructures share. To some extent, both models consider multiple nodes with precedence constraints on the activation of several tasks. The main difference is the possibility of having or not having shared memory. Systems like DRTSJ do not consider nodes that may have shared memory while traditional multi-core systems do have shared memory.
- The second important issue is that very probably future DRTJava implementations will run on multi-core infrastructures. So that, the internal scheduling algorithms that support DRTSJ should be designed with a multi-core infrastructure in mind. For instance, current end-to-end techniques defined for distributed DRTJava are silent on how to integrate a multi-core infrastructure as part of their infrastructures.

2.4.4 Models Based on CSP Formalism

As an alternative to the programming model defined by the DRTSJ, applications may use CSP (*Communicating Sequential Processes*). CSP is a formal description language for describing concurrent systems that can be used to describe distributed real-time Java systems written in CSP. Some researchers have produced an alternative model for distributed real-time Java entirely based on this formalism [204]. CSP supports sequential, parallel and alternative constructs that communicate using channels. The framework allows the use of priorities attached to processes so that users may use classical scheduling algorithms to assign priorities to concurrent entities. In this model, special channels transfer data among nodes using arbitrary communication frameworks like TCP/IP and CORBA.

DRTJava may profit from two characteristics of CSP: it may be modeling its behavior using CSP [206] and also may be modifying its communication model in order to include this abstraction within its core. In the first case, the choice enables the use of verification tools typically associated with CSP. In the second, the programming model is augmented with additional communication mechanisms which interconnect different threads and provide an additional programming abstraction not included currently in DRTSJ.

2.5 Augmented Technologies: Benefits and Approaches

In relationship to the approaches defined in the previous sections, there are other real-time Java technologies that can be part of cyber-physical infrastructures [44]. The current list of candidates includes names such as RT-OSGi, RT-EJBs, and RT-Jini. Each technology complements and augments distributed real-time Java in a different aspect, offering different enhanced facilities to RT-RMI. In addition, RT-RMI may be a requirement in the development of these technologies. For each technology, the rest of this section outlines its goals and the role played by distributed real-time Java in meeting them.

2.5.1 RT-OSGi

The first candidate is RT-OSGi [180] (real-time open services gateway initiative). OSGi offers life cycle management to deploy, start, stop and undeploy bundles automatically at runtime. These bundles may have interactions with local and remote entities. The real-time prefix should control local resources by assuming an underlying RTSJ framework and using DRTSJ in remote communications as required. In addition, the bundles have standard services and may define system dependencies.

Some specific contributions to the definition of this technology include: a hybrid model (C and Java) [180] which declares a real-time component model; and several platform optimizations to provide centralized isolation [325] and admission control [324]. So far, the integration of distributed real-time support in RT-OSGi has been addressed only partially in some preliminary works [33].

Readers interested in the integration of the RTSJ and OSGi are referred to Chap. 12 of this book, which considers in greater depth this issue.

2.5.2 RT-EJBs

In Java, Enterprise Java Beans (EJBs) [393] define a component model for distributed Java applications with persistence, naming and remote communication services. Internally, EJBs may use RMI to carry out a remote invocation. Unfortunately, the model lacks a real-time (i.e., RT-EJBs) specification that provide a quality-of-service framework and enhanced services, similar to those proposed for the CORBA's component model (CCM) [285].

Some interesting steps have been given in terms of characterizing a container model for RTSJ services [414] and other RTSJ-based component models (e.g., the one described in [302]). In both approaches, the container runs on a real-time Java virtual machine that offers real-time predictability. In addition to the centralized support, the container may introduce a distribution service based on distributed real-time Java technology.

Readers interested in component-oriented development for real-time Java are referred to Chap. 11 of this book, which considers in greater depth this issue.

2.5.3 RT-Jini

Jini offers the possibility of registering, finding and downloading services in a centralized service. When services are distributed, the application may use remote invocations to communicate remote virtual machines. A real-time version (RT-Jini) could improve the basic support in several ways including [167]: quality-of-service parameterization of services, predictable execution (which could be built upon RTSJ and DRTSJ) and predictable lookup and composition. Some initial steps in service characterization [167] and composition [151,152] have been given in CoSeRT [167].

In addition, some previous work on real-time tuples that use real-time Java [66] may help to refine the RT-Jini's model.

2.6 Conclusions

This chapter has covered the current state of distributed real-time Java, addressing different technologies and giving the status of each the main. It has also focused on DRTSJ, the leading effort towards a distributed real-time Java technology. It also introduces other related efforts that may be considered as alternative approaches, such as RT-CORBA alternatives or predictable infrastructures based on RT-RMI and the use of a DDS-and-RTSJ in tandem. For all of them, the chapter outlines primary issues addressed highlighting general contributions to distributed real-time Java.

For the support required to implement distributed real-time Java, the chapter addressed the use of hardware infrastructures, real-time networking, multi-core infrastructures and formal models that help validate applications. All these techniques may enhance distributed real-time Java in different ways.

Lastly, the chapter considered the use of distributed real-time Java as some kind of support to other upcoming real-time technologies. The list of outlined technologies includes RT-Jini, RT-EJBs and RT-OSGi.

A final note should be written regarding the next step to be given in distributed real-time Java. To date, there is no publicly-available distributed real-time Java reference implementation nor a specification. The next efforts should be driven in this direction, i.e. producing a public reference implementation for distributed real-time Java. To achieve this goal, much of the work described in this chapter (e.g, the distributable threads and the architectures for RT-RMI) may be helpful. This support is also crucial for other augmented Java technologies that require end-to-end predictable communications as part of their specification.

Acknowledgements This work was supported in part by the iLAND Project (100026) of Call 1 of EU ARTEMIS JU and also in part by ARTISTDesign NoE (IST-2007-214373) of the EU 7th FP programme. We also thank Marisol García-Valls (mvalls@it.uc3m.es) for her comments on a previous draft of this chapter.



http://www.springer.com/978-1-4419-8157-8

Distributed, Embedded and Real-time Java Systems
Higuera-Toledano, M.T.; Wellings, A.J. (Eds.)
2012, X, 378 p., Hardcover
ISBN: 978-1-4419-8157-8