A Survey on Standards for Real-Time Distribution Middleware

HÉCTOR PÉREZ and J. JAVIER GUTIÉRREZ, University of Cantabria

This survey covers distribution standards oriented to the development of distributed real-time systems. Currently, there are many distribution standards that provide a wide and different set of real-time facilities to control the temporal aspects of applications. Besides giving a general overview of these standards, we describe the real-time mechanisms proposed by each standard to manage both processor and network resources, discuss whether the available facilities are sufficient to guarantee determinism throughout the whole application, and identify a set of features and deployment options that would be desirable in any real-time distribution middleware regardless of its distribution model and standard. The survey identifies open issues and key challenges for future research.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms: Performance, Design

Additional Key Words and Phrases: Middleware, distribution standards, networks, scheduling, real time

ACM Reference Format:
Héctor Pérez and J. Javier Gutiérrez. 2014. A survey on standards for real-time distribution middleware. ACM Comput. Surv. 46, 4, Article 49 (March 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2532636

1. INTRODUCTION

The concept of a distributed application is not new; it has existed since two computers were first connected and may consist of several tens of processors interconnected by one or more communication networks. However, the programming techniques used in these systems have evolved greatly, and they have become especially relevant in the past decade. Today, many services are provided transparently to the user and executed in a computer network: Automatic Teller Machines (ATMs), cable TV, and Web services are examples used in our daily lives.

Simple and homogeneous distributed applications can be developed directly using the communications services provided by operating systems. However, the direct use of such services by the programmer, even if it usually provides good performance, is error prone. Thus, a set of high-level abstractions (distribution models or paradigms) have been defined for these communication services in order to allow the programmer to specify interactions between components of a distributed application easily. These include Remote Procedure Calls (RPCs), distribution based on objects (DOM), distribution based on messages (MOM), distributed stream computing, or the Tuplespaces

Authors' address: Héctor Pérez and J. Javier Gutiérrez, Computers and Real-Time Group, University of Cantabria, Avda. de los Castros s/n, 39005 - Santander, Spain; emails: perezh@unican.es, gutierjj@unican.es.
This work has been funded in part by the Spanish government and FEDER funds under grant TIN2011-28567-C03-02 (HI-PARTES).
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 0360-0300/2014/03-ART49 $15.00
DOI: http://dx.doi.org/10.1145/2532636

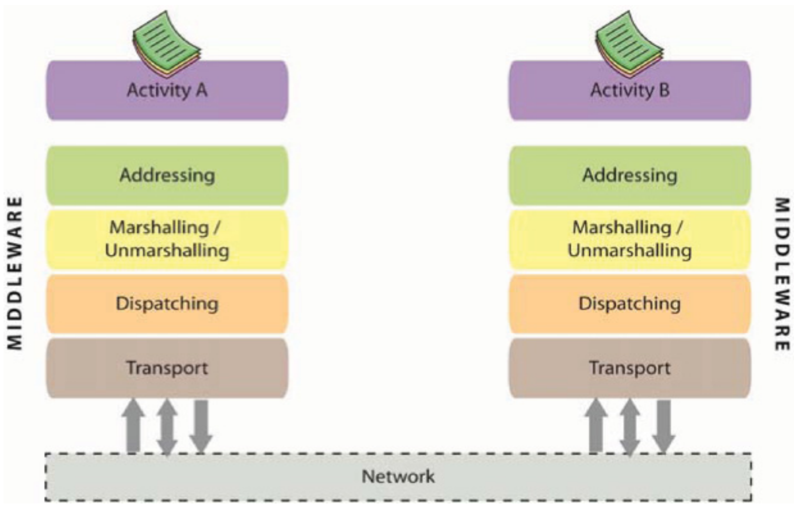

Fig. 1. Basic services provided by distribution middleware.

paradigm. Furthermore, recent trends in the industry have envisaged the use of a new distribution model based on data (also known as the data-centric model).

In the case of systems composed of dozens of computers with heterogeneous architectures, the development of distributed systems becomes complex and must (1) ensure communication between nodes and (2) address low-level communication details, such as the byte storage format (i.e., endianness), word size, or floating-point representation used. The management of this complexity can be maintained transparent to the user through the use of middleware technology, an intermediate software layer that simplifies the management and programming of applications and has become an essential tool in the development of distributed systems. Today, the concept of middleware is very broad and provides several features:

—*Communication middleware*, which is an abstraction of the low-level details related to distribution and communications
—*Component middleware* [Klefstad et al. 2002], which is usually based on a formal model that enables the development of systems by assembling reusable software modules (components) that have been developed previously by others regardless of the application that will be used
—*Model-driven middleware* [Gokhale et al. 2008], which mainly focuses on achieving a sustainable development process in terms of costs, development times, and quality by combining component middleware with model-based software development
—*Adaptive middleware* [Blair et al. 2001], which enables the reconfiguration of distributed applications to modify functionalities, resource usage, security settings, and so on
—*Context-aware middleware* [Rouvoy et al. 2009], which is able to interact with the environment where distributed applications execute and take action to make changes at runtime

We will focus on the first group described—communication middleware—which usually provides the basis for the development of higher-level middleware. This type of middleware internally handles the details of the interconnection process between nodes that usually consists of the following basic features (Figure 1): (1) addressing or assignment of identifiers to entities in order to denote their location, (2) marshalling or transformation of data into a representation suitable for transmission over the network, (3) dispatching or assignment of each request onto an execution resource for processing, and (4) transport or establishment of a communication link for exchanging network messages via unicast or multicast communications.

This survey focuses on standard-based distribution middleware due to its stability and impact on the industry. Currently, there are many standards that fall within any of the mentioned distribution paradigms. Thus, among the most representative examples of distribution models based on RPCs are the Open Software Foundation/Distributed Computing Environment (OSF/DCE) standard [The Open Group 1997] or the Distributed Systems Annex of Ada (DSA) [ISO/IEC 2006]. In relation to the DOM model, this paradigm is probably one of the most relevant in current industrial applications [Kim 2000], and an important example is the Common Object Request Broker Architecture (CORBA) standard [OMG 2011]. Other examples of the DOM model are the Java Remote Method Invocation (RMI) [Sun Microsystems 2004] or the previously mentioned Ada DSA, which also enables distribution based on objects. Examples of the MOM model are the Java Message Service (JMS) [Sun Microsystems 2002], a de facto standard, and the Data Distribution Service for Real-Time Systems (DDS) [OMG 2007]. However, the latter is often included in the data-centric category, as the distribution relies solely on knowledge of the data types to share; besides supporting generic messaging, messages are derived from the system data model (i.e., there is support for formally defined data types). Moreover, the contents of exchanged messages are not opaque to middleware and can be handled directly. The Tuplespaces paradigm is principally represented by JavaSpaces [Freeman et al. 1999]. Finally, and despite the significant number of emerging applications for the processing of large data streams such as S4 (Simple Scalable Streaming System) [Neumeyer et al. 2010] or S-NET [Grelck et al. 2012], there is no standardized solution for this paradigm at this time.

Unlike general-purpose systems, a real-time system is defined as a special kind of system whose logical correctness is based on both the correctness of the outputs and their timeliness. Thus, it is not sufficient for the software to be logically correct; the applications must also satisfy particular timing constraints. To this end, real-time applications rely on a scheduling scheme to specify a criterion for ordering the use of system resources (e.g., CPUs or communication networks) in such a way that the worst-case temporal behavior can be predicted. The problem of obtaining computationally feasible, reliable, and accurate timing predictions can be solved by applying different analytic techniques for single-processor, multiprocessor, or distributed real-time systems [Liu 2000; Sha et al. 2004; Davis and Burns 2011]. In the case of distributed systems, this process is challenging even for apparently simple distributed systems in which complex dependencies among data or threads allocated in different processors could be present, and therefore networks and processors should be scheduled together [Perathoner et al. 2007].

Although a wide set of distribution standards have been enumerated so far, not all of them are suitable for developing distributed real-time applications, as they require a set of mechanisms and capabilities to ensure determinism—for example, thread and network message scheduling, the assignment of scheduling parameters, or the use of synchronization protocols for a predictable access to shared resources. These mechanisms can be explicitly defined within the standard (e.g., DDS); can be added as an extension to the original distribution model (e.g., CORBA and RT-CORBA [OMG 2005]); or can be considered independent of distribution mechanisms, as in the case of Ada. The JavaSpaces specification presents a high-level abstraction for building distributed applications and so it relies on low-level communication middleware such as RMI.

Furthermore, high-integrity systems represent a special group within the real-time systems, as they are characterized by strict restrictions on their development.

Traditionally, designers have proved reluctant to incorporate recent programming techniques into the development process of these systems. However, this process has evolved rapidly in recent years, and they are starting to consider the application of technical advances that were inconceivable a few years ago, such as the use of priority-based scheduling or distribution middleware.

Given the high degree of attention and wide use of middleware technology in distributed real-time systems, we believe that it is necessary to analyze the current solutions and to critically assess their available real-time mechanisms. This article presents a survey of real-time distribution middleware based on standards with the aim of providing technicians and domain experts a comprehensive overview of the area and identifying significant open issues and future research directions. Likewise, it also addresses the initial steps taken in the use of distribution standards in high-integrity systems.

This document is organized as follows. First, Section 2 reviews the distribution architecture of real-time distribution middleware based on standards. Then, Section 3 focuses on the real-time capabilities included in those standards. Section 4 analyzes real-time networks and their relationship with distribution middleware. Section 5 deals with the use of distribution standards in high-integrity systems. Section 6 discusses whether the real-time mechanisms, included in distribution standards, are enough to ensure application predictability, and reviews the desirable features and properties for this type of middleware. Section 7 demonstrates how implementations address the open real-time issues left unresolved by distribution standards. Finally, Section 8 draws the conclusions.

2. REAL-TIME DISTRIBUTION MIDDLEWARE OVERVIEW

In general-purpose systems, the use of middleware technology aims to facilitate the programming of distributed applications. To this end, middleware provides a high-level abstraction of the basic services provided by operating systems, mainly those related to communications. Thus, developers are only responsible for defining which part of the application can be accessible remotely (e.g., through an Ada DSA interface or via a CORBA object), whereas middleware transparently establishes and manages communication between nodes within the distributed system. Furthermore, real-time systems also benefit from these high-level abstractions.

However, general-purpose middleware cannot be applied directly to real-time systems. In general, the distribution process (see Figure 1) presents several potential sources of indeterminism, including marshalling/unmarshalling algorithms, transmission/reception queues for network messages, delays in transport service, or dispatching of requests. Real-time middleware aims to solve these issues by implementing predictable mechanisms, such as the use of special-purpose real-time communication networks or the management of scheduling parameters. Consequently, this kind of middleware addresses not only distribution issues but should also provide developers with mechanisms that allow the temporal behavior of the distributed application to be determined. The remainder of this section introduces the most notable distribution standards for distributed real-time systems.

2.1. CORBA and RT-CORBA

CORBA [OMG 2011] is DOM middleware that follows the client-server paradigm and whose main feature is to facilitate the interoperability between heterogeneous applications (i.e., those coded in different programming languages, executed on different platforms, or even those middleware implementations developed by different companies). The specification was developed by an industry consortium called the Object Management Group (OMG). An overview of CORBA architecture is shown in Figure 2(a). It is comprised of the following components:

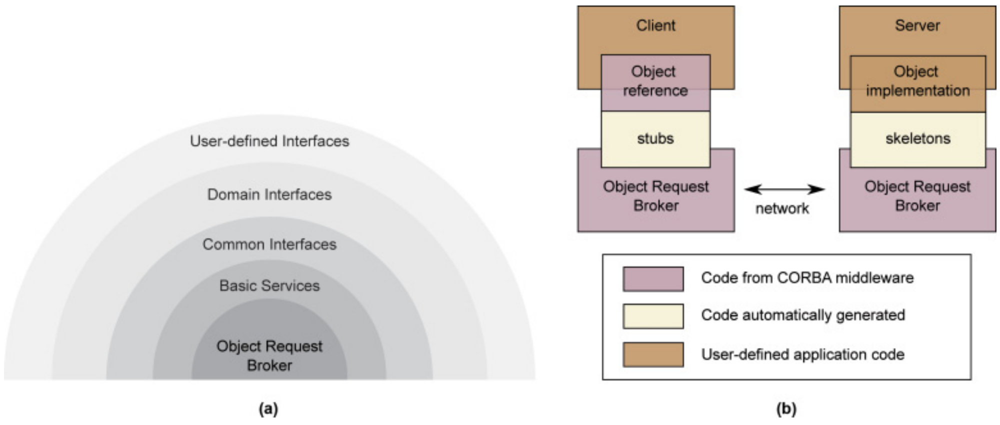

Fig. 2. Please provide the Figure and caption or renumber

—*Object Request Broker (ORB)*: ORB represents the core of middleware and is responsible for coordinating the communication between client and server nodes.
—*System Interfaces*: These consist of a set of interfaces grouped according to their scope, which include (1) a collection of Basic Services that support the ORB (e.g., location of remote objects, concurrency, persistence); (2) a set of Common Interfaces across a wide range of application domains (e.g., database management, data compression, authentication); (3) a group of interfaces for a particular application domain (Domain Interfaces) such as telecommunications, banking, and finance; and (4) User-Defined Interfaces (i.e., not standardized).

Since there is no software, operating system, or programming language that meets all industrial requirements, the main objective of CORBA is to provide solutions to support the heterogeneity of systems, relying on two basic aspects:

—*Language-Independent Middleware (Multilanguage)*: CORBA objects are defined by using a description language called Interface Definition Language (IDL). Currently, within the CORBA standard, there are specifications for the mapping of data types to multiple programming languages (Ada, Java, or C, for example).
—*Platform-Independent Middleware (Interoperable)*: CORBA defines a generic transport protocol called General Inter-ORB Protocol (GIOP). This protocol ensures interoperability between CORBA objects regardless of whether they are allocated to ORBs from different vendors or to different platforms. The Internet Inter-ORB Protocol (IIOP) is the specific mapping of the GIOP protocol over TCP/IP networks, which is considered the baseline transport for CORBA implementations.

Communication between nodes is performed by using several CORBA entities, which are illustrated in Figure 2(b) and described next:

—*Object Request Broker*: The ORB provides mechanisms to enable transparent invocation of a remote method as if it were a local method. Thus, the ORB abstracts the location of remote objects and the method of communicating with them.
—*Client Stubs and Server Skeletons*: These represent those parts of the code, which are usually automatically generated, in charge of redirecting the remote call through the ORB, as well as performing the marshalling and unmarshalling operations.
—*Object Reference*: This is an opaque reference that uniquely determines the location of a remote object and is called an Interoperable Object Reference (IOR). The IOR includes details of all network protocols and receiving ports that the ORB can use to

process incoming requests. This reference is generated and managed by the Portable Object Adapter (POA).
—*Communication Networks*: Both client and server nodes communicate through the ORB by using the GIOP protocol. This protocol is on top of the OSI transport layer and can be implemented on top of several network protocols; however, the CORBA standard only includes guidelines to implement it for networks based on IP.

Although CORBA provides comprehensive support for distributed objects, this standard does not include support for real-time applications. Therefore, this lack of support was addressed by the OMG through an optional set of extensions to CORBA called RT-CORBA [OMG 2005]. These extensions are described next:

—*RT-ORB*: An ORB extension that adds functions for the creation and destruction of specific real-time entities (e.g., mutexes, threadpools, or scheduling policies) and enables the assignment of priorities for their usage by internal ORB threads.
—*RT-POA*: Represents an extension to the POA [OMG 2011] and provides support for the configuration of the real-time policies defined by RT-CORBA. Such policies handle the end-to-end priority propagation models, the management of remote calls, the priority banded connections, or the selection/configuration of available network protocols.
—*Priority and Priority Mapping*: These represent an interface that both defines a generic priority data type (regardless of the underlying operating system) and provides operations to map native priorities onto RT-CORBA priorities (range 0 to 32,767) and vice versa. This mapping is implementation defined.
—*Mutex*: A portable interface for accessing the mutexes supplied by the RT-ORB. It provides synchronization mechanisms for controlling access to shared resources (e.g., sections of code).
—*RTCurrent*: An interface to determine the priority of the current invocation (i.e., it enables the priority of application threads to be handled).
—*ThreadPool*: A mechanism to control the degree of concurrency during the execution of remote calls on the server side.
—*Scheduling Service*: This is a service that simplifies the configuration of the timing aspects of the system. Through this service, RT-CORBA allows the application to specify its requirements based on various parameters such as priorities, deadlines, or expected execution time, whereas middleware will be responsible for setting up the required resources to meet them.

The use of these RT-CORBA entities enables the development of critical (e.g., real-time control systems) and noncritical (e.g., travel agencies or online shopping cart) real-time applications. Currently, RT-CORBA is employed in a wide range of scenarios such as Software Defined Radios [Bard and Kovarik 2007] or Industrial Robotics [Amoretti et al. 2006] and can be considered a very mature technology.

2.2. The Ada Distributed Systems Annex

The Ada programming language [ISO/IEC 2012] is an international standard that includes an annex dedicated to developing distributed applications: Annex E or Ada DSA. The major strength of the DSA is that the source code is written without regard for whether it will be executed on a distributed platform or on a single processor.

In the design of distributed systems, an application designed for a single processor can be divided into different functionalities that, when acting together, can provide a particular service to end users. The execution of each of these functionalities may be distributed across several interconnected nodes, while end users transparently invoke the service. In the Ada programming language, each part of the complete application

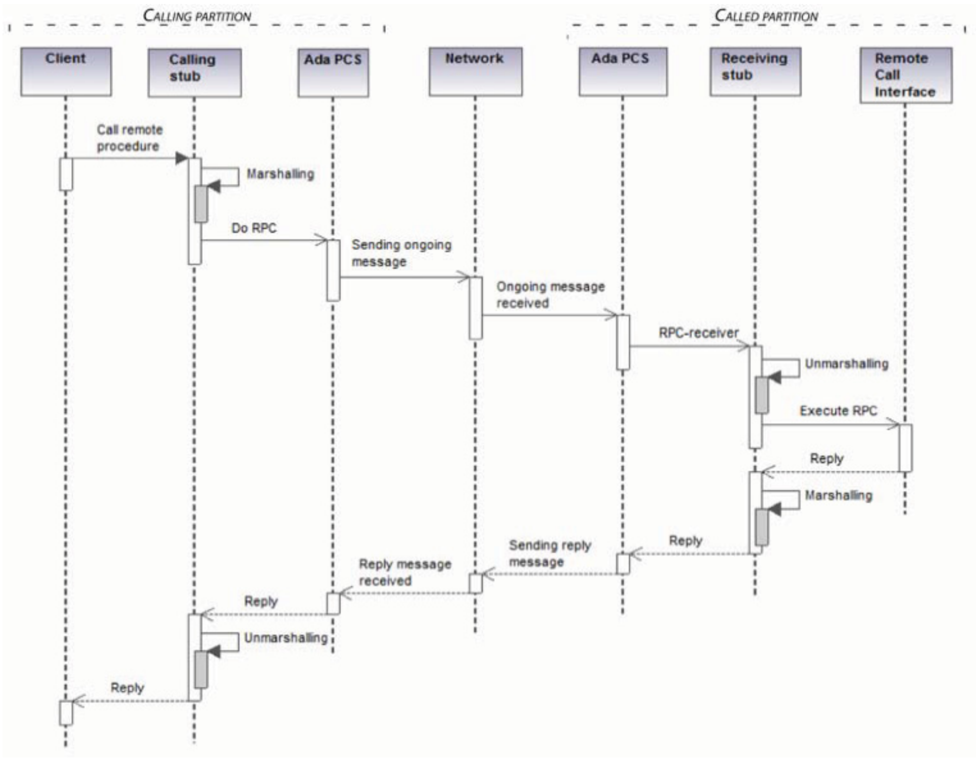

Fig. 3. Components of the DSA distribution model involved in a synchronous remote call.

that is independently assigned to each node is called a *partition*. Formally, according to the Ada Reference Manual, “a partition is a program or part of a program that can be invoked from outside the Ada implementation.” [ISO/IEC 2012]

The partitioning of an application through the DSA is not defined by the standard but is implementation defined. Partitions communicate with each other by exchanging data through RPCs (*Remote Call Interface*) and distributed objects (*Remote Types*). The DSA defines two kinds of partitions: active, which can execute in parallel with one another, possibly in a separate address space and on a separate computer; and passive, which are partitions without a task or thread of control (e.g., storage nodes). It is worth noting that the terms *task* and *thread* can be used indistinctly in the context of this survey.

Active partitions communicate through the Partition Communication Subsystem (PCS), a language-defined interface responsible for routing subprogram calls from one partition to another. Access to PCS should not be done directly from the application level, but from *calling* and *receiving stubs*. The PCS supports compilers used to generate stubs for a standard interface without being concerned with the underlying implementation. Despite this standardization effort, a recent revision of the programming language [ISO/IEC 2006] allows the use of alternative interfaces to PCS in order to facilitate the interoperability with other middleware (e.g., CORBA).

The high-level components of the distribution model proposed by the DSA are illustrated in Figure 3. This figure represents the sequence diagram of a synchronous remote call between two partitions: a partition that requires remote services (*calling partition*) and a partition that provides these services (*called partition*) through a remote call interface.

Although the DSA allows distributed systems to be built in a simple manner, it is not specifically designed to support predictable applications, and most of the issues that affect determinism have been left up to the implementation. However, there is some previous research in this line, and there are implementations that show that it can be used for real-time applications [Vergnaud et al. 2004; Campos et al. 2006]. Although this annex has not had a very significant commercial impact [Kermarrec 1999], Ada has traditionally been used and is still used to build real-time single-processor systems, so it is worth considering the analysis of this standard and its future development.

2.3. The Data Distribution Service for Real-Time Systems

Anonymous and asynchronous dissemination of information has been a common requirement for many different distributed applications, such as control systems, sensor networks, and industrial automation systems. The DDS [OMG 2007] aims to facilitate the exchange of data in these kinds of systems through the publisher-subscriber paradigm. Unlike other specifications that follow this paradigm, the communication model proposed by the DDS is data centric (i.e., the focus is on the data itself). A data-centric architecture must formally define the data type to be shared in the distributed system, and then information is exchanged anonymously by simply writing and reading samples of that data type. With a data-centric approach, middleware is aware of the content of the information exchanged and so it can directly handle it (e.g., data filtering).

As with most of the standards defined within OMG, DDS supports multilanguage and multiplatform capabilities by using the IDL language [OMG 2011] to define shared data types and the DDS Interoperability Wire Protocol (DDSI) [OMG 2009] to interoperate among different implementations, respectively. Beyond this, OMG has recently released the Extensible and Dynamic Topic Types specification [OMG 2012] to provide support for extensible and evolvable distributed systems using DDS. This specification allows data types to be dynamically defined (i.e., they can be used without compile-time knowledge) or modified (i.e., data fields can be added or removed). To this end, the specification provides DDS with a structural data type system, new data type representations, different serialization or encoding formats, and a new API for the management of data types at runtime.

The DDS conceptual model is based on the abstraction of a strongly typed *Global Data Space*, where publisher and subscriber respectively write (produce) and read (consume) data, leading to middleware focused on obtaining data independently from its origin. To better handle the exchange of data, the standard defines a set of entities involved in the communication process. Applications that wish to share information with others can use this Global Data Space to declare their intent to publish data through the *DataWriter* (DW) entity. Similarly, applications that need to receive information can use the *DataReader* (DR) entity to request particular data. *Publisher* and *Subscriber* entities are containers for several DWs and DRs, respectively, which share common QoS parameters. Likewise, these entities are grouped in *Participants* of a *Domain*. Only entities belonging to the same Domain can communicate. At a higher level of abstraction, the Participant entity contains all DWs, DRs, Publishers, and Subscribers that share a common QoS in the corresponding Domain.

To exchange information among entities, Publishers only need to know about the specific *Topic* (i.e., the data type to share) and Subscribers require registration of their interest in receiving particular Topics, whereas middleware may establish and manage the communication transparently. Within the definition of a Topic, one or more elements can be designed as a *Key*. This entity enables the existence of multiple instances of the same Topic, thus allowing DRs to differentiate the source of the incoming data for instance (e.g., a set of vehicles updating their position or a cluster of

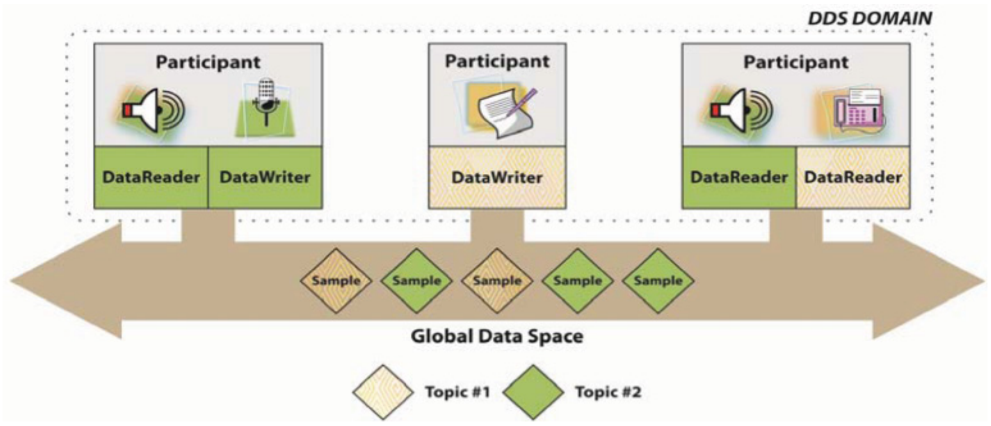

Fig. 4. Communication model for DDS.

temperature sensors providing information from different areas). The example in Figure 4 shows a distributed system that consists of three Participants in a single Domain and two Topics. Both Topics have a single DW in charge of generating new data samples. However, successive updates for Topic #1 will only be received by one DR, whereas new samples for Topic #2 will be received by two DRs.

Publishers and Subscribers are not required to communicate directly among themselves, but they are rather loosely coupled in terms of the following:

—*Time*, because data samples could be stored and retrieved later (e.g., when new Subscribers join the distributed system and require information about the previous state of the system)
—*Space*, because Publishers of data do not need to know about each individual receiver, whereas Subscribers do not need to know the source of the data samples—that is, Publishers and Subscribers are not known by each other

As was mentioned earlier, the development of distributed systems with DDS is bound to another specification that sets the main guidelines for performing the communication among entities: the DDSI. This protocol aims to guarantee the interoperability among different implementations by using the standard Real-Time Publish-Subscribe Wire Protocol (RTPS) [OMG 2009] together with the Common Data Representation (CDR) defined in CORBA [OMG 2011]. Although this specification is focused on IP networks, any other real-time network protocol could be used. For those interested readers, a detailed introduction to DDS can be found in Corsaro and Schmidt [2012].

Finally, although DDS has been designed to be scalable, efficient, and predictable, few researchers have evaluated its real-time capabilities [Pérez and Gutiérrez 2012]. Nevertheless, it is considered a mature technology and has already been deployed in several real-time scenarios such as Defense [Schmidt et al. 2008], Automation [Ryll and Ratchev 2008], or Space [Gillen et al. 2012].

2.4. The Distributed Real-Time Specification for Java

Besides the distribution standards, there are other nonstandard solutions that have attracted great interest among developers. This is the case of the Java programming language and its extensions for distributed real-time systems, which is considered a de facto standard by the community.

Java was initially designed as a programming language for general-purpose systems and, therefore, has several drawbacks for the development of predictable applications,

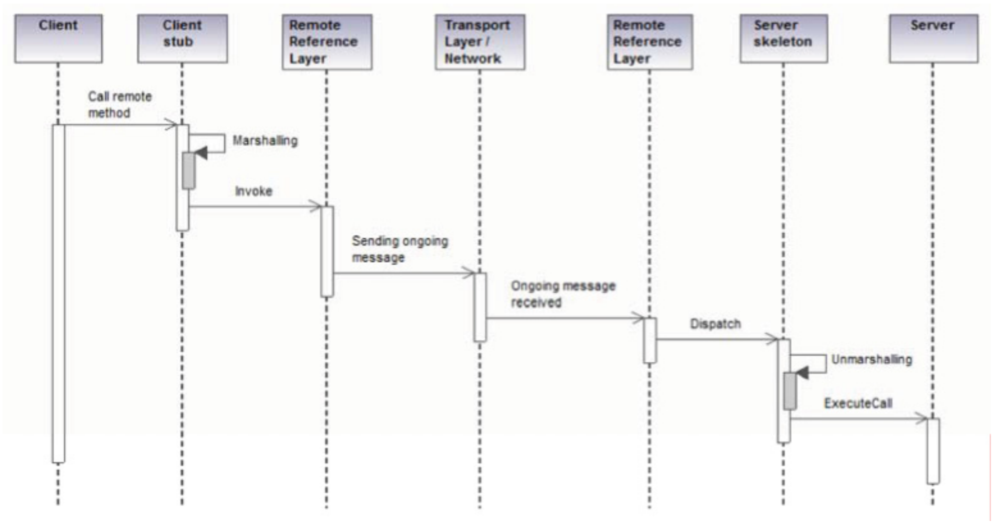

Fig. 5. Components of the RMI distribution model involved in an asynchronous remote call.

especially those aspects related to the management of internal resources such as memory or processor scheduling [Basanta-Val et al. 2010]. For distributed real-time systems, one of the most notable research works is the Distributed Real-Time Specification for Java (DRTSJ) [Sun Microsystems 2000], which integrates two existing Java technologies:

—*Real-Time Specification for Java (RTSJ)* [Bollella and Gosling 2000], which defines a Java specification to address the limitations of the language when used in real-time systems. As one of the guiding principles was to avoid making syntactic extensions to the language, real-time support was achieved through new libraries, refined Java mechanisms, and a real-time Java Virtual Machine (JVM) with support for both general-purpose and real-time applications. Nevertheless, this specification is only conceived for single-processor systems.
—*Remote Method Invocation (RMI)* [Sun Microsystems 2004], which defines a DOM model based on Java objects by defining a new interface, called *Remote*, which enables the differentiation of distributed objects from local ones. An overview of the high-level components involved in the RMI architecture is shown in Figure 5, which represents the sequence diagram of an asynchronous remote call between a client and a server. This architecture is comprised of the following components:
  —*Client Stubs or Proxies* and *Server Skeletons*: Represent the interface between the application layer and the rest of the RMI system. They are in charge of providing transparent distribution facilities.
  —*Remote Reference Layer*: Responsible for handling the semantics of remote invocations, both on client and server sides.
  —*Transport Layer*: Used to set up the connections and manage the low-level communication details. The specification defines the RMI wire protocol, which relies on two other protocols: Java Object Serialization and HTTP.

Even though Java is one of the most popular programming languages, no official DRTSJ specification has yet been released, although there is an early draft available [Sun Microsystems 2012] and the working group Web site [Sun Microsystems 2000] outlines the important features of the future specification. Nevertheless, several lines of research aim to adapt the language to a deterministic model not only for single-processor environments but also for distributed ones [Tejera et al. 2007; Basanta-Val et al. 2010].

3. ANALYSIS OF REAL-TIME DISTRIBUTION MIDDLEWARE BASED ON STANDARDS

The main objective in the design of distributed real-time systems lies in guaranteeing determinism over the whole application. For this purpose, distribution standards include different mechanisms to control the timing aspects of software and enable the application of analytic techniques to them. Basically, these mechanisms attempt to highlight implicitly how the available resources of the system should be used, mainly those concerned with the management of processors and communication networks. To this end, this section analyzes the distribution standards according to (1) the management of the processors, discussing the mechanisms provided to ensure a predictable timing behavior of tasks or threads, and (2) the management of networks to guarantee a deterministic network usage. In particular, this analysis reviews the distribution mechanisms proposed by RT-CORBA, Ada DSA, DDS, and Java DRTSJ, as they provide outstanding and standardized solutions for the development of distributed real-time systems.

3.1. RT-CORBA

The extension of the CORBA specification for real-time systems—RT-CORBA [OMG 2005]—adds new interfaces and mechanisms that aim to increase the predictability of applications distributed through CORBA. The standard is divided into two distinct parts: the first deals with those systems that are suitable for a priori timing analysis to determine whether their timing constraints are satisfied (static systems), whereas the second focuses on systems with variable workload and whose schedulability is guaranteed at runtime (dynamic systems).

The use of the set of real-time entities defined by RT-CORBA enables applications to configure and control the system resources explicitly, as is described next.

3.1.1. Managing Processor Resources. According to the static scheduling chapter of this specification, the main features of the RT-CORBA architecture are as follows:

—*Scheduling Based on Fixed Priority Scheduling Policy*: This first part of the specification includes only those systems scheduled by means of fixed priorities. This scheduling policy is implemented by the majority of real-time operating systems, especially those following the Portable Operating System Interface (POSIX) real-time standard [The Open Group 1998].

—*Use of Threads as Schedulable Entities*: In this case, RT-CORBA priority can be applied, and there are functions for conversion to the native priorities of the system on which they execute. According to this priority mapping, RT-CORBA defines three priority models:

- *Client_Propagated*, where the invocation is executed in the remote node at the priority of the client, which is transmitted with the request message.
- *Server_Declared*, when all the requests to a particular distributed object are executed at a priority preset in the server.
- *Priority Transforms Model*, which enables the user to define priority transformations that modify the priority associated with the server depending on different parameters, such as the current system workload or state. The transformation is done with two functions called *inbound* (which transforms the priority before running the server's code) and *outbound* (which transforms the priority with which the server makes calls to other remote services).

—*Definition of Threadpools to Control the Degree of Concurrency in the Server*: This mechanism enables different applications to share a number of threads. The configuration of this entity enables the specification of the number of threads that must be preallocated, the number of threads that may be created dynamically, and their default priority. It also allows groups of threads to be defined based on priority (ThreadpoolLanes).
—*Deterministic Access to Shared Resources*: RT-CORBA defines a local *Mutex* object to coordinate contention for shared resources. This mutex should implement a synchronization protocol based on priority inheritance. However, the standard does not specify any particular protocol, so implementations are responsible for setting which protocol or protocols may be used.

The specification of RT-CORBA incorporates a chapter dedicated to dynamic scheduling, which basically introduces two concepts:

—*Use of Different Scheduling Policies*: The possibility of introducing other scheduling policies in addition to the fixed-priority one, such as Earliest Deadline First (EDF) [Liu and Layland 1973], Least Laxity First (LLF) [Mok 1983], and Maximize Accrued Utility (MAU) [Jensen et al. 1985]. The scheduling parameters are defined as a container that can contain more than one simple value and can be changed by the application dynamically at runtime.
—*Use of Distributable Threads as a Schedulable Entity*: The Distributable Thread enables end-to-end scheduling by identifying scheduling segments and scheduling points that may be allocated in a separate address space. Scheduling segments represent pieces of code associated with a given set of scheduling parameters specifically set by the application. Scheduling points define points in time and/or code at which the scheduler is run and may result in schedule changes [OMG 2005].

3.1.2. Managing Network Resources. RT-CORBA does not explicitly consider the possibility of passing scheduling parameters to the communication networks, although it defines other mechanisms to mitigate the lack of predictability associated with the use of general-purpose communication networks. These are described as follows:

—*Protocol Properties*: RT-CORBA provides interfaces to specify the preferred protocol and to fine-tune the parameters of the protocol on both the client and server side. There are implementations that extend this interface to map the RT-CORBA priorities onto the underlying network [Schmidt 2005], although this is not standardized in the specification.
—*Use of Private Connections*: Ordinarily, given that GIOP is a connection-oriented protocol, the ORB is allowed to reuse or share a network connection to service multiple remote objects. However, multiplexing requests on a single connection implies that a client may be blocked while the connection is being used by another invocation. This mechanism removes this blocking by enabling the client to obtain a dedicated connection (that is, nonmultiplexed) per remote object.
—*Definition of Priority-Banded Connections*: This mechanism allows multiple connections between clients and servers to be established by associating each connection with a single or a range of priorities. This mechanism aims to reduce priority inversions when the underlying transport protocol is not deterministic.

3.2. The Data Distribution Service for Real-Time Systems

The DDS standard was explicitly designed to build distributed real-time systems. To this end, this specification adds a set of Quality of Service (QoS) parameters to configure nonfunctional properties. In this case, DDS provides high flexibility in the configuration of the system by associating a set of QoS parameters to each individual entity.

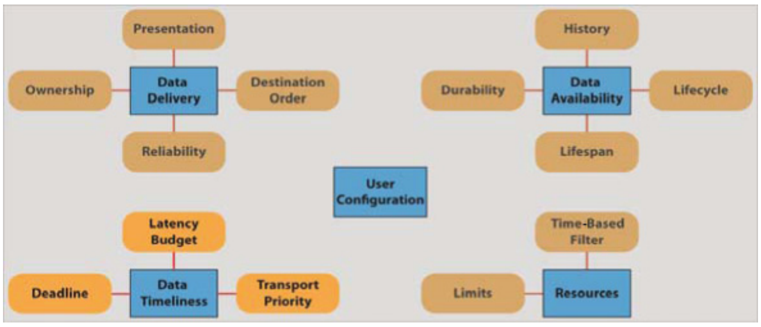

Fig. 6. QoS parameters defined by DDS.

Furthermore, DDS enables the modification of some of these parameters at runtime while performing a dynamic reconfiguration of the system. This set of QoS parameters allows several aspects of data, networks and computing resources to be configured and may be classified in the following categories (Figure 6):

—*Data Availability*: This comprises parameters for controlling queuing policies and data storage. The parameters that fall into this category are *Durability*, *Lifespan*, and *History*.
—*Data Delivery*: This specifies how data must be transmitted and presented to the application. The parameters that fall into this category are *Presentation*, *Reliability*, *Partition*, *Destination_Order*, and *Ownership*.
—*Data Timeliness*: This controls the latency in the distribution of data. The parameters that fall into this category are *Deadline*, *Latency_Budget*, and *Transport_Priority*.
—*Maximum Resources*: This limits the amount of resources that may be used in the system through parameters such as *Resource_Limits* or *Time-Based_Filter*.
—*User Configuration*: These parameters allow extra information to be added to each entity at application level.

Finally, this specification follows the *subscriber-requested, publisher-offered* pattern to set QoS parameters. By using this pattern, both publishers and subscribers must specify compatible QoS parameters to establish the communication. Otherwise, middleware must indicate to the application that communication is not possible.

3.2.1. Managing Processor Resources. The DDS specification does not explicitly address the scheduling of threads in the processors, as this is an implementation-defined aspect. However, a subset of the QoS parameters defined by the standard is focused on controlling the temporal behavior and improving the predictability of the application. The three parameters of Data Timeliness, which are highlighted in Figure 6, are particularly important in the management of resources for real-time systems. In particular, the specification has defined the following parameters for managing processor resources:

—*Deadline*: This parameter indicates the maximum amount of time available to send/receive data samples belonging to a particular topic. However, it does not define any associated mechanism to enforce this timing requirement; therefore, this QoS parameter only represents a notification service in which middleware informs the application that the deadline has been missed.
—*Latency_Budget*: This parameter is defined as the maximum acceptable delay in message delivery. However, the standard emphasizes that this parameter may not

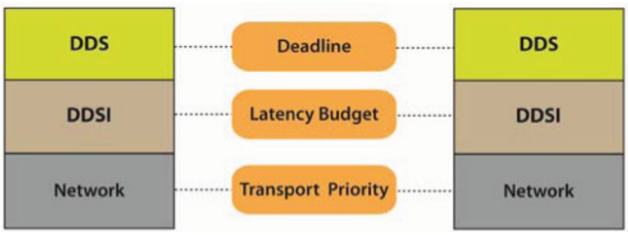

Fig. 7. Timing control in DDS.

be enforced or controlled by middleware; therefore, this parameter can be used to optimize the internal behavior of middleware.

These two QoS parameters, even if both share similar objectives, are applied at different levels, as illustrated in Figure 7. This figure shows how the Deadline parameter is monitored within the DDS layer, whereas the Latency_Budget is applied within the DDSI layer.

DDS defines different mechanisms to enable communication among entities. On the publisher side, the communication mechanism is straightforward: when new data are available, the DW performs a simple write call (e.g., write or dispose) to publish data into a DDS Domain. Then, the data sample is transmitted using asynchronous and one-to-one or one-to-many communication modes. However, DDS also provides support to block the calling thread until the data sample has been delivered and acknowledged by the matched DRs.

On the subscriber side, the reception of data can be performed in polling, synchronous mode, and asynchronous mode. These models are not only valid for the reception of data but also for the notification of any change in the communication status (e.g., nonfulfillment of requested QoS). In particular, the application could be notified through the following:

—*Polling*, as application threads can invoke the nonblocking operations to obtain data or changes in the communication status.
—*Listeners*, attaching a callback function to asynchronously access modifications in the communication status while the application keeps executing (i.e., middleware threads are responsible for managing any change in the communication status).
—*Conditions* and *Wait-sets*, which allow application threads to be blocked until one or several conditions are met. Both represent the synchronous mechanism to manage any change in the communication status.

3.2.2. Managing Network Resources. In relation to networks, this specification defines a set of features focused on guaranteeing determinism for communications, such as the use of scheduling parameters in networks and the definition of the format for the exchanged messages.

The passing of scheduling parameters to communication networks is performed through another QoS parameter included in the Data Timeliness category (see Figure 6):

—*Transport_Priority*: Unlike the Latency_Budget, which attempts to optimize the internal behavior of middleware, this parameter prioritizes the access to the communication network (see Figure 7). Furthermore, since communications are unidirectional, it is only associated with DW entities.

Moreover, the DDSI specification defines the set of rules and features required to enable communication among DDS entities. Although this specification is not particularly oriented to the use of real-time networks, it does not preclude their use and only lists a set of requirements for the underlying networks. The most important point addressed by the specification is the description of the RTPS protocol, which is responsible for specifying how to disseminate data among nodes. This requires the definition of the exchange information protocols and message formats. In particular, the structure of an RTPS message consists of a fixed-size header followed by a variable number of submessages. By processing each submessage independently, the system can discard unknown or erroneous submessages and thus facilitate future extensions of the protocol.

Another key feature of DDS is the overhead introduced by internal middleware operations. In this case, the standard defines a series of operations to be performed by implementations that may consume both processor and network resources. In particular, DDS provides a service for the management of remote entities called *Discovery*. This service describes how to obtain information about the presence and characteristics of any other entity within the distributed system. Although the standard describes one specific discovery protocol for the purpose of interoperability, it allows other approaches to be implemented. Under the required discovery protocol, implementations must create a set of DDS entities by default. These built-in entities are responsible for (1) establishing the communication transparently with the user and (2) discovering the presence or absence of remote entities (e.g., a plug-and-play system). This kind of network traffic, which is internal to middleware, is called *metatraffic* and should be considered in the timing analysis.

3.3. Distributed Systems Annex of Ada

DSA only deals with those mechanisms related to distribution, such as the configuration of the partitions of a program, the distribution models supported, or how to perform the communication between partitions. However, Ada DSA delegates the concurrent and real-time features to other parts of the language, such as those defined in the Real-Time Systems Annex (Annex D). Thereafter, since the use of DSA is closely linked to Ada programming, the analysis of the standard will consider the features included in the core of the language and the Real-Time Systems and Distributed Systems annexes.

The latest versions of the language—Ada 95, Ada 2005, and Ada 2012—have defined new mechanisms to develop predictable applications within the Real-Time Systems Annex and the Ada concurrency model. Therefore, the concurrency and the real-time mechanisms are supported by the language itself with the definition of the following:

—*Tasks*: These represent active entities that provide support for programming concurrent or parallel operations and interaction mechanisms. Furthermore, different scheduling parameters can be assigned to them, such as a priority or a scheduling deadline.
—*Protected Objects*: These provide a task-safe and deterministic access to shared data, as well as an event synchronization mechanism.
—*Timing Facilities*: These include different kinds of clocks and timers to measure real time and execution time of a single task or a group of tasks, as well as statements to suspend tasks with absolute, relative, or conditional delays.
—*Flexible and Extensible Scheduling Facilities for Tasks*: This includes standard scheduling policies based on fixed or dynamic priorities that can be simultaneously applied on the same system.

Although Ada defines a coherent real-time model for single-processor and multiprocessor systems, it does not address distributed systems. That is, the DSA is not specifically designed to support real-time applications. However, there are research works that demonstrate that it is possible to write real-time implementations within the standard [Gutiérrez and Harbour 1999, 2001]. The key aspects of the language for the management of processor and communication network resources are described briefly next.

3.3.1. Managing Processor Resources. The Ada concurrency model is supported by tasks and several interaction mechanisms, and it has the following features:

—*Flexible and Extensible Scheduling Model*: The language allows the use of different scheduling policies on the same partition, thus enabling the execution of applications with heterogeneous requirements. Ada includes the following scheduling policies within the Real-Time Systems Annex:

- *FIFO_Within_Priorities* is a preemptive scheduling policy based on fixed priorities that uses First-In-First-Out (FIFO) order for the same priority level.
- *Nonpreemptive_FIFO_Within_Priorities* is a nonpreemptive scheduling policy based on fixed priorities that uses FIFO order for the same priority level.
- *Round_Robin_Within_Priorities* is a preemptive scheduling policy based on fixed priorities in which tasks are time sliced for the same priority level.
- *EDF_Across_Priorities* is a preemptive scheduling policy based on dynamic priorities that uses deadlines for ordering tasks at the same priority level.

—*Support for Servicing Concurrent Remote Calls*: The specification requires support for executing concurrent remote calls and for waiting until the return of the remote call. As mentioned previously, the communication among active partitions is carried out in a standard way using the PCS, although the specification does not define how it is performed (i.e., it is implementation defined).

—*Predictable Access to Shared Resources*: Protected objects guarantee mutually exclusive access to shared resources, but they themselves do not provide bounded blocking times during access. For this, the standard has defined a set of protocols depending on the scope:

- Synchronization protocols (or locking policies according to Ada terminology), which enable deterministic access to shared resources. The Real-Time Systems Annex only obliges the implementation of the *Priority Ceiling Protocol* for both fixed priorities [Sha et al. 1990] and EDF [Baker 1991] versions, although it does not preclude the use of other policies.
- Queuing policies to specify the order in which tasks are queued for accessing shared resources. Two queuing policies are language defined: priority order (*Priority_Queuing*) and arrival order (*FIFO_Queuing*).

3.3.2. Managing Network Resources. Like RT-CORBA, Ada DSA does not consider the possibility of passing scheduling parameters to the communication networks, although there are some research works that have incorporated this concept [Gutiérrez and Harbour 1999, 2001].

Furthermore, Ada DSA does not have any mechanism for the transmission of priorities, as this aspect is open to implementation. Pautet and Tardieu [2000] propose a mechanism to handle the transmission of priorities following the same scheme defined by RT-CORBA. In Campos et al. [2004, 2006], some mechanisms for handling the transmission of priorities within DSA are proposed. These mechanisms are in principle more powerful than those of RT-CORBA, as they allow total freedom in the assignment of priorities both in the processors and in the communication networks at a configuration stage without modifying the application logic.

3.4. THE DISTRIBUTED REAL-TIME SPECIFICATION FOR JAVA

The development framework provided by the Java language is also an important technology within the real-time community. The model of distributed systems for real-time

Java is being developed by the JSR-50 Expert Group [Sun Microsystems 2000]. However, the lack of a definitive specification means that there is only an outline of its key elements. The major objective is to incorporate the concepts of distribution and real-time to Java instead of adapting the language to provide this support.

As in the case of Ada DSA, Java defines a real-time model for single-processor systems, which is defined by the RTSJ specification. DRTSJ builds on top of the mechanisms included in RTSJ, including the following:

—*Timing Facilities*: These allow the use of different types of clocks, such as high-resolution, absolute, and relative time clocks.
—*A New Memory Management Mechanism*: This avoids the effects of the garbage collector by defining different memory areas (*HeapMemory*, *ImmortalMemory*, and *ScopedMemory*).
—*Synchronization and Resource-Sharing Facilities*: The priority inversion problem is avoided by introducing the use of synchronization protocols (e.g., protocols based on priority inheritance and priority ceiling) and nonblocking communication mechanisms (e.g., *WaitFreeWriteQueue* and *WaitFreeReadQueue*).
—*A New Scheduling Framework*: This addresses real-time systems with both fixed and variable workload. The specification defines a new *Scheduler* class in order to support different scheduling algorithms, although it only obliges the implementation of the preemptive scheduling policy based on fixed priorities.
—*Use of Schedulable Objects*: RTSJ introduces two types of schedulable objects, *RealtimeThread* and *AsyncEventHandler*, which provide programming models based on threads and events, respectively. The former extends the expressiveness of regular Java threads to include real-time properties, such as:

- *SchedulingParameters*, which is used to determine the schedule and may be even changed at runtime.
- *ReleaseParameters*, which represents different timing characteristics of each schedulable object such as deadlines, maximum blocking times, processing costs, or release frequencies (e.g., periodic, sporadic or aperiodic).
- *MemoryParameters*, which specifies the maximum amount of memory required.
- *ProcessingGroupParameters*, which groups a set of schedulable objects whose execution relies on a virtual server (e.g., deferrable server).

The RTSJ specification is extended in the DRTSJ by providing the following key elements for the management of the processors [Anderson and Jensen 2006; Sun Microsystems 2012]:

—*Coherent Support for End-to-End Requirements in Distributed Applications*: Middleware must provide support for this type of requirements, not only for temporal constraints but also other necessities such as fault management or security. To this end, a new entity called *Distributable Thread* is introduced to provide an abstraction of the control flow of distributed applications. This concept is similar to the one defined in RT-CORBA.
—*An Easily Extensible Scheduling and Integrity Framework*: To facilitate the building of heterogeneous and complex systems, application designers may use appropriate user-defined policies for recovery in the presence of failures or scheduling distributable and local threads.

In relation to communication networks, DRTSJ does not provide any mechanism and leaves this matter for future versions of the specification. The current draft only states that the use of a real-time network protocol must not be mandatory by default, as this would allow the interoperability between general-purpose and real-time systems.

However, the applicability of this real-time model to distributed systems still represents an open research field [Tejera et al. 2007; Basanta-Val et al. 2010]. Therefore, since the DRTSJ specification is not yet complete and there are aspects that still have not been addressed, a more thorough analysis of it will not be attempted. Hereinafter, we will mainly focus on the study of the RT-CORBA, DDS, and Ada DSA standards.

4. REAL-TIME COMMUNICATION NETWORKS AND DISTRIBUTION MIDDLEWARE

Along with the distribution mechanisms provided by middleware, networks represent the other key element in the communications of a real-time system. Although most standards do not consider network scheduling (e.g., RT-CORBA or DSA) or provide limited support for it (such as DDS), the amount of time for sending or receiving messages determines the response time of a distributed system. From the perspective of real-time systems, communication networks are responsible for solving several problems, such as:

—*Transmission Order for Messages Available in a Network Device*: It is necessary to use a policy to schedule which message, among all those locally available, will be the next to be transmitted. This problem is especially relevant when using interconnection devices (e.g., switches) that should order incoming messages from different nodes prior to their transmission (e.g., by using priorities).
—*Shared Transmission Medium Among Several Network Devices*: In this case, it is necessary to use a policy to schedule which network device, among all those available, will be the next to transmit.

However, in general, the distribution standards that have been analyzed do not consider any of these problems, and therefore they do not specify the required properties of the underlying communication subsystem that may affect the temporal behavior of the distributed system. Thus, whereas Ada DSA does not address any characteristics of communication networks, the RT-CORBA and DDS specifications define two network protocols to facilitate interoperability between implementations—GIOP and DDSI, respectively. Although both protocols require the use of a message format by default, neither addresses how communications should be performed, as this aspect is defined by the underlying transport service.

In the first case, GIOP requires a reliable and connection-oriented transport service. This latter requirement has motivated the adaptation of some real-time protocols to comply with the standard, as is the case of the CAN protocol implemented by ROFES [Lankes et al. 2003]. However, the standard defines the IIOP protocol, which uses TCP/IP, as the reference protocol for the interconnection of CORBA subsystems. For instance, the TAO implementation [Schmidt et al. 1998] provides a mechanism to map RT-CORBA priorities onto the *Diffserv* data field [Nichols et al. 1998]. The mechanism proposed by *Diffserv* is based on the principle of traffic classification, where each network packet is placed in a different class of network traffic. Its main objective is to provide QoS guarantees in wide area networks such as the Internet. Under this approach, developers can specify the traffic class corresponding to the IP packet through a header data field whose length is 6 bits, thus allowing up to 64 different traffic classes. Each network device is configured to differentiate traffic based on its class, each traffic class being managed differently. However, *Diffserv* does not address what types of traffic should be given priority treatment, as this depends on each network device. Therefore, *Diffserv* cannot ensure a priori that packet processing will be uniform throughout the network. To partially mitigate this issue, the IETF RFC 2474 standard [Nichols et al. 1998] recommends certain values for this data field to ease interoperability between network devices (e.g., a value of 46, corresponding to the traffic class named Expedited Forwarding, will use a strict priority queuing above all other traffic classes).

However, the use of TCP/IP, even when using Switched Ethernet technology, is not appropriate for hard real-time systems [Felser 2001; Zhang and Tsaoussidis 2001]. This has motivated the development of an extension to the standard called *Extensible Transport Framework* (ETF) [OMG 2004], a framework that enables the integration of communication protocols other than TCP/IP with GIOP. However, there are hardly any developments using this framework to integrate real-time network protocols, probably because of its complexity [Foster and Aslam-Mir 2005]. The work in Losert et al. [2004] uses a preliminary version of ETF to implement a prototype that integrates the TTP/C communication protocol [Kopetz 2011] with RT-CORBA, and a similar solution also exists for TAO [O'Ryan et al. 2000].

The DDSI protocol is designed to use boundary-preserving and connectionless best-effort transport, and only requires a minimal set of services from the transport layer. Actually, it is sufficient that the underlying transport layer offers support to send/receive messages and detects errors during transmission (e.g., incomplete or corrupted messages). Moreover, since the size of messages is not sent explicitly by the DDSI protocol, the underlying transport must provide a mechanism to deduce the size of the received message. This latter requirement could be problematic for protocols that transmit data as an unstructured sequence of bytes (*stream oriented*) and do not preserve the boundary of messages from upper layers (e.g., TCP/IP). To address this issue, support for stream-oriented protocols in DDSI is being discussed at OMG meetings, although neither official nor draft documents have been released yet. Finally, although the DDS specification includes a QoS parameter to send network messages with different priorities, the underlying transport is not required to be capable of managing priorities or to support network scheduling based on priorities. Therefore, most implementations use UDP/IP networks, although there are some academic research works that integrate real-time communication networks, such as the CAN bus [Rekik and Hasnaoui 2009]. The OpenSplice,[1] RTI-DDS,[2] and OpenDDS[3] implementations, for example, mainly use a UDP/IP transport protocol and, in a similar way to what is done in TAO, map the *Transport_Priority* QoS parameter to the *Diffserv* data field [Nichols et al. 1998].

One of the main conclusions to be drawn is that most distribution standards and implementations for real-time systems currently use IP-based communication networks. Several factors may explain this willingness to use general-purpose instead of specific real-time communication networks, among which are their reduced costs and high data rates, as well as the evolution of the Ethernet technology to meet new bandwidth and market requirements, including the development of new standards (e.g., IEEE 802.1p [IEEE 2006]) that allow the prioritization of network traffic.

Although Ethernet was originally designed to interconnect general-purpose computers, the desire to incorporate a real-time element into this increasingly popular protocol has led to an evolving field of research during the past decade, mainly due to its features of low cost and high transmission speed (currently up to 10Gbps). As is defined in IEEE 802.3, Ethernet technology is not deterministic and thus it is unsuitable for real-time applications. The main problem is the MAC protocol named Carrier-Sense Multiple-Access protocol with Collision Detection (CSMA/CD), which uses nonpredictable back-off algorithms to avoid a message collision. However, Switched Ethernet introduces single collision domains and thus eliminates access contention. This has increased the volume of information that switches can receive simultaneously, and as a result, the existence of long bursts of messages or even an excess of multicast or broadcast messages may cause queue overflow for switches [Pedreiras et al. 2003]. This effect, which

[1]OpenSplice is available at www.prismtech.com.
[2]RTI-DDS is available at www.rti.com.
[3]OpenDDS is available at www.opendds.org.

is unacceptable for hard real-time systems, can be controlled by using flow control techniques for network traffic as is described in Vila-Carbó et al. [2008] or in the recent IEEE 802.1Qbb specification [IEEE 2011]. In the former, the authors limit network traffic through flow control mechanisms provided by operating systems in order to classify, schedule, and drop network messages when sending large volumes of information. The latter is a reference to a new standard that defines flow control mechanisms within network devices based on message priority. Another important factor to consider when using Ethernet devices for hard real-time systems is the network traffic generated by switches. This kind of traffic, which is caused by other network protocols such as the *Spanning Tree* protocol [IEEE 2004], must be disabled or modelled so that it can be taken into account in the timing analysis.

Thus, Switched Ethernet technology is presented as a valid alternative to traditional real-time networks as long as it is used under certain conditions (e.g., with controlled traffic loads). This is the case, for example, of the new ARINC-664 specification, Part 7, which is called Avionics Full-Duplex Switched Ethernet (AFDX) [ARINC 2009] and defines a hard real-time network based on Switched Ethernet for aircraft data networks.

Finally, distribution middleware provides a set of software services, as shown in Figure 1, to facilitate the distribution of one or more application among different nodes. However, when middleware is specifically designed to be used in real-time systems, it should also provide support for configuring networks (e.g., by allowing the assignment of scheduling parameters to network messages) and should define the required constraints on the underlying transport to ensure predictability (e.g., deterministic resolution or suppression of message collisions, identification of the additional traffic generated by network devices, and predictable routing).

5. HIGH-INTEGRITY REAL-TIME SYSTEMS AND DISTRIBUTION MIDDLEWARE

During recent decades, real-time systems have increased their complexity by means of adding dozens of processing nodes that host independent or coupled applications, most of them having nonfunctional requirements such as deadlines, QoS, or integrity. For instance, in high-integrity systems, a possible failure may lead to unacceptable consequences or damage (e.g., financial, environmental, or personal disasters). Therefore, these kinds of systems must undergo a certification process to verify their compliance with certain requirements imposed by different standards: DO-178B for avionics, IEC 880 for nuclear plants, MISRA for automotive, and so forth. Due to the high costs associated with the certification process, the development of safety-critical systems is characterized by the simplicity of source code—that is, it tends to minimize the software complexity to ease the certification. A common practice is to take advantage of subsets or profiles of the selected technology that restrict the use of those features that are difficult to certify. The challenge is greater for high-integrity distributed systems, as it is not sufficient to certify each part of the system individually and then combine them. Instead, the entire system must be certified.

Over the past years, the real-time community has attempted to boost flexibility to the development process of high-integrity systems. One of the most notable efforts in this sense is the evolution from the cyclic executive scheduling policy to a fixed priority scheduling scheme, which not only increases the flexibility in the development process but also facilitates the management of concurrency features in high-integrity software. Likewise, the complexity associated with the communication management of distributed systems is motivating the exploration of the use of distribution middleware in high-integrity systems. However, this kind of systems does not traditionally consider the use of middleware, because it involves the use of an extra software layer that makes the certification process more complex. Nevertheless, the use of middleware technology can provide a set of services that may be of interest for this kind of system, such as distribution transparency, network abstraction, communication management, or interoperability. For instance, Hugues et al. [2008] present PolyORB-HI, a minimal middleware development that is part of TASTE [Perrotin et al. 2010], a set of tools developed by the European Space Agency (ESA[4]) that supports the development of high-integrity systems. PolyORB-HI provides basic distribution services such as data marshalling/unmarshalling, request dispatching, and network transport. Under this approach, location addressing service is based on the automatic generation of source code from architectural descriptions (i.e., system models) instead of following any distribution standard.

As networking support is necessary for many safety-critical applications, current distribution standards may be seen as a feasible solution. However, distribution standards should overcome some challenges before they will be acceptable for use in high-integrity systems, mainly those related with the size and complexity of software. There are some efforts in this direction; for example, a DDS profile for safety-critical systems is being discussed at meetings of the OMG.

The Ada programming language provides several facilities to aid in the development of high-integrity systems such as *SPARK* [Altran Praxis 2011] or the *Ravenscar* profile [ISO/IEC 2006]. The former is a subset of the sequential part of Ada that restricts the use of certain features to facilitate the static analysis, whereas the latter defines a safe and analyzable subset of Ada concurrency facilities. The Ravenscar profile has become a useful tool for developing real-time single-processor systems, and it has recently been extended to be used with multiprocessors [ISO/IEC 2012], although its applicability to distributed systems still remains open to research [Audsley and Wellings 2001; Urueña and Zamorano 2007; Pérez et al. 2012].

The fact that some of the capabilities of the RTSJ specification are not certifiable for a safety-critical system has led to the proposal of a simpler profile named Safety Critical Java, of which an official early draft specification has already been released [The Open Group 2010]. As in the case of Ada, this profile is only aimed to single-processor systems. Nevertheless, the use of Java in high-integrity distributed real-time systems is currently being investigated [Tejera et al. 2007; Higuera-Toledano 2012] and will be a notable research field in the near future.

Regarding the CORBA standard, the research of Dubey et al. [2011] presents a real-time framework that contains a middleware implementation that is executed on top of a real-time operating system that fulfils the ARINC-653 (Avionics Application Standard Software Interface) avionics standard [ARINC 2006]. This specification defines a set of entities called *ports* to enable the interpartition and intrapartition communication, and that act as the input points to a real-time network (e.g., AFDX [ARINC 2009]). However, this framework extends the CORBA Component Model (CCM) [OMG 2011] instead of relying on the standard real-time facilities.

Although the use of middleware technology in high-integrity systems is attracting a high degree of interest within the real-time community, current research in this field is at an early stage and has not been sufficiently widely accepted to merit standardization. Nevertheless, it will be a point of interest during the coming years.

Finally, the use of a time-shared scheme in high-integrity systems has evolved in recent years to a more sophisticated paradigm referred to as partitioning or partitioned systems. Partitioning is a widespread technique that enables the execution of multiple applications in the same hardware platform with strong temporal and space isolation, thus allowing the coexistence of mixed-criticality applications. This technique is starting to be applied in a wide set of heterogeneous scenarios [MultiPARTES 2011] that may take advantage of using real-time distribution middleware. For instance, the

[4]http://www.esa.int/esaCP/.

work included in Pérez and Gutiérrez [2013] presents an early experience with the integration of distribution middleware into partitioned systems.

6. DISCUSSION ON REAL-TIME CAPABILITIES OF DISTRIBUTION MIDDLEWARE

After analyzing the different distribution standards aimed at the development of applications with timing requirements, this section attempts to find the similarities and differences among these specifications, as well as to assess their appropriateness for use in real-time systems. To better compare the different standards introduced earlier, it is necessary to define a set of requirements that a distribution standard for real-time systems should consider:

—*Support for Scheduling in Processors and Networks*: Real-time systems require strict control of the execution of threads and the transmission of messages. This includes support for scheduling policies in charge of ordering the concurrent access of threads and messages to processors and communication networks, respectively.
—*Control of the Scheduling Parameters*: Middleware should provide mechanisms to configure the scheduling parameters of each thread that may be executed in the processor. Similarly, the assignment of scheduling parameters to network messages should be supported.
—*Thread Management or Concurrency Pattern*: It represents the design pattern that deals with the multi-thread paradigm that controls and processes the dissemination of information. This feature is particularly important on the receiver side.
—*Controlled Access to Shared Resources*: This can be achieved through the implementation of synchronization protocols.

Two types of schedulable entities can be identified in a real-time system: threads for processors and messages for communication networks. Real-time engineers should be able to configure both entities, not only those defined within the application but also those created by middleware implementations. For instance, this is the case of Input/Output (I/O) decoupling threads, receiving threads, or built-in network messages, which should be configured through standardized mechanisms provided by middleware. Furthermore, those entities may also produce some kind of contention that must be taken into account in the real-time design.

6.1. Managing Processor Resources

The temporal behavior of distribution middleware is strongly determined by the scheduling policies and concurrency patterns [Pérez et al. 2008]. In the first case, it is necessary to identify which mechanisms are provided by middleware to select a specific scheduling policy and how to perform the assignment of the corresponding scheduling parameters to the schedulable entities responsible for attending to remote services. The second case deals with the options available to establish which thread is responsible for sending or receiving remote requests/data.

First, it is possible that schedulers are directly supported by the operating system. However, since these distribution standards are aimed at developing real-time systems, it would be desirable to include operations in their APIs to set a specific scheduling policy and the corresponding scheduling parameters for middleware threads.

Both Ada and RT-CORBA specifications provide support for different scheduling policies, including the Fixed Priorities Scheduling (FPS) policy. Similarly, the scheduling framework defined by DRTSJ can be extended in order to support different scheduling policies. However, the model proposed in DDS does not include the scheduling in the processors, which remains undefined. Although the DDS standard defines several timing parameters, none is suitable to schedule threads in the processors: the *Deadline* parameter could be used in some cases (i.e., EDF systems), but the standard does not consider such use. A similar situation exists with the *Latency_Budget* parameter and whose definition is not clear, although the specification proposes data batching (i.e., gathering a set of data samples to be sent in a single large network package) as an example of use.

RT-CORBA is the only specification that provides mechanisms to specify the scheduling parameters to be used during the execution of the requested operations on the remote node. The specification for static systems defines two policies, *Server_Declared* and *Client_Propagated*, which impose restrictions on the assignment of priorities and therefore reduce the schedulability of the system [Campos et al. 2006]. Furthermore, although the *Priority Transforms* model allows the modification of these policies, this is performed within the application-supplied code, so any change in the scheduling parameters may also require reviewing it.

Second, the processing of remote calls or incoming data represents a two-stage process that includes (1) the listening for I/O events in communication networks and the processing of network messages and (2) the execution of the application code associated with remote calls or incoming data, and a possible reply. The former is internally performed and controlled by middleware, whereas the latter relies on the interaction between middleware and the application. As discussed previously, distribution standards do not define which concurrency pattern should be used for the first stage but specify that implementations must service concurrent remote requests (e.g., the Ada DSA explicitly indicates this aspect, whereas RT-CORBA implicitly specifies it through the definition of *Threadpools*). However, the choice of one or another concurrency pattern is a factor that determines the temporal behavior of the application, so this issue will be addressed further in the analysis of the implementations (see Section 7). Concerning the second stage, RT-CORBA and Ada DSA lead the execution of application code in the context of an internal middleware thread, whereas DDS enables the use of internal middleware threads, by means of the Listener mechanism, or application threads, through Wait-set structures or by directly polling for data availability in a DR. Anyway, regardless of the thread or threads responsible for processing each stage, it is important that middleware provides the necessary mechanisms to control their scheduling parameters.

Finally, deterministic access to shared resources prevents the unbounded priority inversion problem [Sha et al. 1990]. RT-CORBA, Ada, and RTSJ include the use of synchronization protocols for access to critical sections, although only the latter two specify that implementations need to support specific protocols (e.g., the *Priority Ceiling Protocol*).

6.2. Managing Network Resources

In relation to communication networks, neither RT-CORBA nor Ada DSA include the possibility of assigning scheduling parameters; therefore, implementations are responsible for providing the necessary support for this. Concerning DRTSJ, the last report only states that both real-time and general-purpose networks must be supported. In the case of DDS, the specification only considers networks based on a fixed priority scheduling policy and excludes any other kind of predictable networks used in the industry (e.g., time-triggered networks). This can be dealt with by modifying the definition of the *Transport_Priority* parameter as proposed in Pérez and Gutiérrez [2012].

Although most of the standards analyzed are focused on Ethernet-based networks (e.g., RT-CORBA with TCP/IP and DDS with UDP/IP), this communication network is not itself suitable to provide deterministic response times, as was discussed in Section 4. However, the evolution of Ethernet technology in recent years, with the definition of new standards, such as 802.1p [IEEE 2006], which prioritizes different message streams, together with its low cost, has resulted in a growing interest within the industry in using this approach in the future development of real-time systems.

When distribution middleware is implemented on operating systems and network protocols with priority-based scheduling, it is easy to transmit the priority at which a remote service must be executed inside the messages sent through the network. For example, this scheme is used by the *Client_Propagated* policy in RT-CORBA. However, this solution does not work if more complex scheduling policies, such as flexible scheduling frameworks based on contracts [Aldea et al. 2006; FRESCOR 2006], are used. Sending the contract parameters through the network is inefficient because these parameters are large in size. In the same way, the dynamic change of the contract parameters in the remote node is also inefficient, so other configuration schemes are required for this kind of system.

Another important factor to consider is the size of network messages, which must be bounded and known before the timing analysis. This point is particularly critical in the design of predictable applications with DDS, since a DDSI message can comprise an undefined number of submessages, including not only metatraffic but also user data. Although this mechanism is quite efficient for minimizing the average response time, it is not usually suitable for real-time systems that aim at guaranteeing latency limits in each network stream. Therefore, it is up to implementations to provide the means to define the maximum size of a DDSI message.

Finally, the presence of messages and operations belonging to middleware may cause an increase in the response times of critical user applications. Although this overhead depends almost exclusively on each implementation, the effect seems to be more significant in standards such as DDS, which defines a set of built-in entities that may consume both processor and network resources.

6.3. Comparative Summary

This section has discussed the mechanisms provided by distribution standards for the management of processors and networks. Table I summarizes the analysis according to the degree of support for the requirements proposed at the beginning of this discussion (*scheduling policies*, *setting of scheduling parameters*, *concurrency patterns*, and *controlled access to shared resources*).

Most of these features, which are required to bound the worst-case temporal behavior, remain open to implementations, as is shown in Table I. Consequently, the choice of a particular middleware determines not only the application performance but also its predictability, and thus the ability to meet its deadlines. The choice of the concurrency pattern for the processing of remote calls or incoming data is particularly relevant, although this feature depends on implementations.

Finally, Figure 8 shows the evolution of these distribution standards for real-time systems over time. As can be seen, the RT-CORBA standard was mainly reviewed and updated until 2005, when the latest version of the standard was released. In this case, the RT-CORBA specifications have followed an unusual sequencing of version numbering, which includes RT-CORBA 1.0 (1999), RT-CORBA 1.1 (2002), RT-CORBA 2.0 (2003), and RT-CORBA 1.2 (2005). Ada DSA has evolved along with the programming language itself by introducing new real-time capabilities and safety-critical guidelines in the two revisions of the standard released since 1995. However, these new features are mainly standardized for single and multiprocessor systems, and current efforts in distributed systems are still at the research level. In the case of DDS, the first specification was introduced in 2004, and the technology has been updated more frequently with new features and extensions. Although the first draft of DRTSJ was introduced in 2000, this specification was not updated until 2012; an official specification has not yet been released.

Table I. Real-Time Capabilities of Distribution Standards

	Processor Resources				Network Resources	
Middleware	Scheduling policies	Setting of scheduling parameters	Concurrency pattern	Synchro. protocols	Scheduling policies	Setting of scheduling parameters
RT-CORBA	FPS EDF LLF MAU	Client propagated Server declared Priority_Transf.	Threadpool	Required	Impl. def.	Impl. def.
Ada DSA	FPS Nonpreemptable Round-robin EDF	Impl. def.	Impl. def.	Priority ceiling	Impl. def.	Impl. def.
DDS	Impl. def.	Impl. def.	Impl. def.	Impl. def.	FPS	Transport priority
Java DRTSJ	FPS User defined	Impl. def.	Impl. def.	Priority inheritance Priority ceiling	Impl. def.	Impl. def.

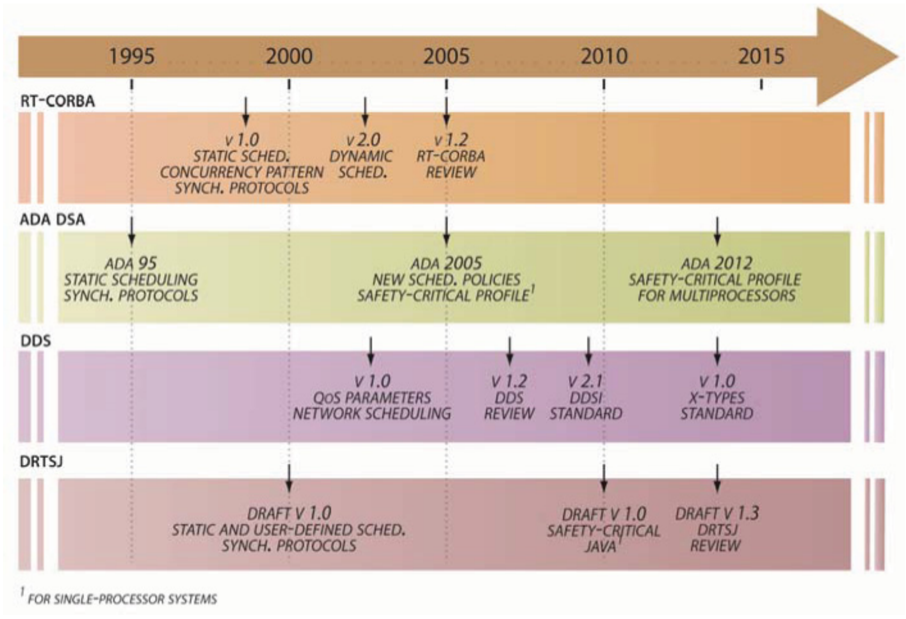
1995
2000
2005
2010
2015
RT-CORBA
v 1.0 STATIC SCHED. CONCURRENCY PATTERN SYNCH. PROTOCOLS
v 2.0 DYNAMIC SCHED.
v 1.2 RT-CORBA REVIEW
ADA DSA
ADA 95 STATIC SCHEDULING SYNCH. PROTOCOLS
ADA 2005 NEW SCHED. POLICIES SAFETY-CRITICAL PROFILE1
ADA 2012 SAFETY-CRITICAL PROFILE FOR MULTIPROCESSORS
DDS
v 1.0 QoS PARAMETERS NETWORK SCHEDULING
v 1.2 DDS REVIEW
v 2.1 DDSI STANDARD
v 1.0 X-TYPES STANDARD
DRTSJ
DRAFT V 1.0 STATIC AND USER-DEFINED SCHED. SYNCH. PROTOCOLS
DRAFT V 1.0 SAFETY-CRITICAL JAVA1
DRAFT V 1.3 DRTSJ REVIEW
1 FOR SINGLE-PROCESSOR SYSTEMS

Fig. 8. Timeline of real-time distribution standards history.

7. ANALYSIS OF THE REAL-TIME FEATURES OF IMPLEMENTATIONS

As discussed previously, distribution standards leave multiple aspects dependent on implementations that may affect the temporal behavior of applications. Therefore, it is of interest to complete the analysis by reviewing some of their reference implementations and their developments. The objective is to show how real implementations deal with these aspects that are not explicitly addressed by standards. Among other features, this part of the analysis will review the proposed concurrency patterns, the support provided for configuring the schedulable entities or the synchronization mechanisms used to control the access to shared resources.

Selected implementations are based on a twofold criterion: on the one hand, they are significantly compliant with their respective standards and thus can be considered as reference implementations; on the other hand, they are open-source implementations to allow a more thorough analysis of their real-time mechanisms to be performed. Although the DRSTJ specification mentions one reference implementation [Sun Microsystems 2012], it is not yet publicly available, and hereinafter we will focus on the study of the RT-CORBA, Ada DSA, and DDS implementations.

7.1. RT-CORBA Implementations

The distribution model proposed by CORBA is a mature technology that has led to numerous implementations, both commercial and open source. In the case of RT-CORBA, there are real-time versions of the commercial distributions ORBExpress,[5] e∗ORB,[6] and VisiBroker,[7] and open-source distributions such as ROFES (Real-time CORBA for embedded systems) [Lankes et al. 2003], TAO (the Ace ORB) [Schmidt et al. 1998], or PolyORB [Vergnaud et al. 2004]. PolyORB is characterized by supporting different

[5]ORBExpress is available at www.ois.com.
[6]e∗ORB is available at www.prismtech.com.
[7]VisiBroker is available at www.microfocus.com.

distribution models, including among others the aforementioned RT-CORBA or Ada DSA. Therefore, the analysis of PolyORB will be dealt with in the section related to the distribution model proposed by Ada. TAO may be regarded as the most popular open-source implementation of RT-CORBA and one of the most complete and efficient versions currently available for real-time systems [Schmidt et al. 2001], so the analysis will focus on this implementation in the following discussion.

TAO implements all of the mandatory features and services that have been defined in the latest version of RT-CORBA [Schmidt 2005] with the following exceptions:

—The priority transforms model
—The use of buffers to store remote requests in threadpools
—The borrowing of threads among threadpool lanes

In relation to the concurrency patterns, TAO defines several configurable properties depending on whether the application is acting as a server or a client. On the server side, these parameters establish concurrency constraints that are imposed by the server node during the processing of requests in a multithreading environment. TAO defines two levels of concurrency that are closely related:

—*Concurrency at Application Level*: These policies control which thread executes the call on the distributed object. Two values are defined:

- *Orb_Ctrl_Model*: This policy allows concurrent requests to a distributed object. In this case, the application developer is responsible for providing thread-safe access to the object (i.e., safe execution by multiple threads at the same time).
- *Single_Thread_Model*: By using this policy, all requests to the distributed object are called sequentially. Therefore, concurrent calls cannot occur within the scope of this policy.

—*Concurrency at ORB Level*: It represents a set of policies to define how threads receive and process requests. These policies are only available if the application-level concurrency is set to Orb_Ctrl_Model. In this case, TAO supports three concurrency patterns:

- *Reactive*: Through this policy, a single server thread is dedicated to handling multiple connections. In addition, other threads may also exist in the system to execute internal middleware operations.
- *Thread-per-connection*: In this case, the ORB creates a new thread to serve each new connection. This thread is dedicated to processing all requests performed on that connection, which will be processed sequentially. After closing the connection, the thread will be released.
- *Threadpool*: Under this policy, middleware creates a pool of threads that are responsible for processing concurrent incoming requests according to a concurrency pattern called *Leader & Followers* [Schmidt 1998; Pyarali et al. 2001]. In the leader-followers pattern, several threads take turns to monitor I/O operations and then process the requests once they have arrived. One thread becomes the leader and then takes responsibility for awaiting a new request and also processing it (i.e., I/O operations are not decoupled from request processing). The other threads in the pool are the followers. As soon as the leader thread receives a new request, one of the follower threads becomes the new leader. Once the thread finishes processing the request, it returns to the pool and waits to become the leader again.

Concerning the client side, these parameters affect the multithreading behavior of the client when a synchronous remote call is performed—that is, when a client thread must wait for a reply from the server. TAO defines a number of concurrency policies for waiting replies in client nodes, which are described as follows:

- *Wait-on-read*: According to this policy, when a client thread invokes a synchronous remote call, it is blocked waiting to read the reply from the server node.
- *Wait-on-reactor*: Under this policy, a single thread is responsible for performing all requests, although it can still perform other internal middleware operations while waiting for the replies (i.e., it is not blocked). When a reply is received, this single thread will be notified in order to process it.
- *Wait-on-leader-follower*: This policy enables client threads to wait for replies using the Leader & Followers concurrency pattern. Therefore, client threads waiting for replies become followers and can be used to perform other I/O or internal middleware operations. As with the Wait-on-reactor case, when a reply is received, the target thread will be notified in order to process it.

In relation to the scheduling for processors, TAO provides support for scheduling policies based on fixed priorities and the importance of threads (Most Important First [MIF]) [Schmidt 2005]. The latter policy is not included in the RT-CORBA standard and defines a parameter called *importance* to determine which thread should execute.

Although RT-CORBA does not consider the assignment of scheduling parameters to the communication networks, TAO provides a method to schedule IP-based networks. As an extension to RT-CORBA, TAO provides a mechanism to map RT-CORBA priorities to network priorities via the configuration protocol properties service. Thus, it is possible to differentiate classes of network traffic. To this end, it uses a data field within the IP header called *Diffserv* [Nichols et al. 1998]. In TAO, protocol properties can be set at the ORB, thread, or object level, so it is possible to enable the network priority mapping for all requests invoked (1) through a particular ORB, (2) through the thread itself, or (3) through the remote object itself, respectively.

Finally, the RT-CORBA standard does not address other kinds of real-time features, such as the priority mapping between native priority and CORBA priority or the synchronization protocol used for shared resources. For the former, TAO defines three priority mappings that are based on a one-to-one mapping (Direct mapping), one-to-one mapping but within a predefined range of CORBA priorities (Continuous mapping), and one-to-many mapping that covers the whole range of CORBA priorities (Linear mapping). For the latter, TAO does not oblige the use of any specific synchronization protocol, so the choice of this protocol will depend on what is provided by the underlying real-time operating system by default.

7.2. Ada DSA Implementations

Although distributed programming with Ada DSA is easier and more intuitive than with other technologies based on DOM and/or RPCs [Kermarrec 1999], the commercial impact of this annex has not been very significant and only a couple of implementations are relevant today: GNAT Library for Ada Distributed Environment (Glade) and PolyORB. Glade [Pautet and Tardieu 2000] is the original implementation of DSA offered by AdaCore to support the development of distributed applications with real-time requirements, although its maintenance has been discontinued today and its functionality has been replaced by PolyORB [Vergnaud et al. 2004]. PolyORB was introduced as middleware that can support different distribution standards such as CORBA, RT-CORBA, DSA, or Web Services. It is distributed with the GNAT compiler, and, in principle, it is envisaged for applications programmed in Ada. It currently supports CORBA, some basic notions of RT-CORBA (priorities and their propagation) and Ada DSA.

The architecture of PolyORB is divided into three separate layers: the application layer (referred to as application personality), the neutral layer or microkernel, and the protocol layer (referred to as protocol personality). Therefore, PolyORB provides a set of common components on top of which several personalities can be developed. This type of architecture allows different personalities to be combined, either at the application level or at the protocol level, within the same software system and thus enables interoperability and integration of different distribution paradigms under a single platform. Furthermore, this architecture allows different communication protocols to be used regardless of the application personality (e.g., the communication between DSA partitions can be done through the GIOP protocol, which is defined in the CORBA specification). The key features of this interoperability rely on (1) the use of a common network protocol for communications and (2) the conversion of any data to neutral data structures defined in the microkernel. This microkernel does not provide the same services as a conventional ORB, but it does include facilities for performing the conversion between distribution models.

For the management of remote calls, PolyORB supports different configurations to adapt the interaction between personalities and the microkernel. These configurable features include (1) the ORB tasking policies (which determine which tasks will execute requests from remote nodes), (2) the ORB controller policies (which determine which tasks will execute internal middleware operations such as I/O processing), and (3) the tasking runtimes (which represent a set of restrictions that must be fulfilled by system tasks). They are briefly described as follows:

—*Tasking Runtimes:* PolyORB defines three tasking profiles or runtimes to establish a set of restrictions on the concurrency model. The choice of a specific tasking runtime is a compilation-time parameter that can take the following values:

- *Full Tasking*: This runtime enables all middleware capabilities to manage and synchronize system tasks.
- *No Tasking*: Under this runtime, no tasking is required and therefore applications can hold a single task at most.
- *Ravenscar*: This runtime enables the concurrency facilities that are compliant with the Ravenscar profile [ISO/IEC 2006].

—*Tasking Policies*: These policies control the creation of tasks for processing incoming remote calls. PolyORB defines the following four policies:

- *No Tasking*: Under this policy, the environment task processes all incoming requests and internal middleware operations.
- *Thread Pool*: This policy defines a group of tasks or threadpool responsible for processing all jobs in middleware. As in the case of Glade, there are three configurable parameters: min_spare_threads, which indicates the minimum number of tasks created at start-up time; max_spare_threads, which represents a ceiling in the number of tasks available to process requests (i.e., tasks are deallocated if the number of tasks is greater than the ceiling); and max_threads, which indicates the absolute maximum number of tasks that the group may contain.
- *Thread Per Session*: This policy creates one task per network connection (i.e., when a new communication session is opened). The task terminates when the connection is closed.
- *Thread Per Request*: This policy creates one task per incoming request. The task is terminated when the request is completed.

PolyORB tasking policies and tasking runtimes have a dependency among themselves, so distributed applications must be configured with a coherent scheme (e.g., the No Tasking runtime implies the No Tasking policy).

—*ORB Controller Policies*: Four policies are defined that affect the internal behavior of middleware, such as the assignment of internal operations and I/O monitoring to middleware tasks:

- *No Tasking*: Under this policy, a loop monitors I/O operations and processes the requests.
- *Workers* [Schmidt 1998]: Under this policy, all threads are equal and monitor the I/O operations and process the incoming requests alternatively.
- *Half Sync/Half Async* [Schmidt and Cranor 1996; Pyarali et al. 2001]: This policy defines one single task to monitor the I/O operations and add the requests to a queue while the other tasks are responsible for processing them (i.e., I/O operations are decoupled from request processing).
- *Leader/Followers* [Schmidt 1998; Pyarali et al. 2001]: As in the case of TAO, this policy defines several tasks that take turns to monitor I/O sources and then process the requests once they have arrived (i.e., I/O operations are not decoupled from request processing).

This middleware is oriented to be used in real-time systems since it partially supports the RT-CORBA standard (static scheduling based on fixed priorities). Furthermore, the DSA personality, even when the standard does not include explicit support for real-time distributed systems, follows the same static scheduling scheme as RT-CORBA: Client Propagated and Server Declared.

This implementation does not explicitly consider the possibility of passing scheduling parameters to the communication networks or configuring synchronization protocols to control access to shared resources. For the latter, the Ada Real-Time Systems Annex allows the Priority Ceiling Protocol to be applied by means of a compiler directive (pragma) that configures by default all protected objects created by middleware. However, the appropriateness of this default configuration will depend on the target application.

7.3. DDS Implementations

The increasing interest within the industry in applying the distribution model defined by DDS has motivated the development of several implementations, both commercial (CoreDX[8] or RTI-DDS[9]) and open-source software (OpenSplice[10] or OpenDDS[11]). For our purposes, we have selected OpenSplice middleware because it is a reference open-source implementation and is considered one of the most efficient implementations of the standard.

PrismTech, one of the driving forces behind the DDS standard, develops and markets a software product called OpenSplice. It relies on a modular collection of pluggable services that provide the essential DDS functionality and a set of features such as networking, resource management, monitoring, or persistence. Additionally, DDS applications built using OpenSplice can be configured to use two different deployment architectures, namely, *federated* and *standalone*. Under the former architecture, applications and middleware services are decoupled (i.e., services are executed as a set of daemons shared by all applications) and interface directly through shared memory; for the latter architecture, DDS applications are built as standalone processes.

As discussed in Section 6, the DDS specification does not standardize the scheduling of threads. To handle this issue, OpenSplice implements two different scheduling

[8]CoreDX is available at www.twinoakscomputing.com.
[9]RTI-DDS is available at www.rti.com.
[10]OpenSplice is available at www.prismtech.com.
[11]OpenDDS is available at www.opendds.org.

policies or classes: *Timeshare*, in which threads have to regularly yield the processor to other threads of equal priority (i.e., round-robin within priorities), and *Realtime*, which represents a preemptive scheduling based on fixed priorities. In any case, the assignment of priorities to any internal threads spawned by OpenSplice is performed through a proprietary extension of the QoS parameters, which allows the priority to be specified as an absolute value or as a value relative to the priority of the parent thread.

In the case of communication networks, the assignment of scheduling parameters is performed through the Transport_Priority QoS parameter, which is mapped to the Diffserv field [Nichols et al. 1998] within the IP header in order to prioritize network traffic through capable network elements (e.g., routers or high-level switches). To handle the transmission and reception of data through the network, OpenSplice implements different networking services:

—*rt-networking Service*: Proprietary and noninteroperable with other DDS implementations, this service is able to handle different classes of network traffic through the definition of priority bands associated to proprietary entities called *network channels*. Once these network channels are created and configured, the rt-networking service selects the most appropriate channel for each incoming/outgoing network message based on the priority specified in the Transport_Priority QoS parameter.
—*DDSI2 Service*: The implementation of the OMG DDSI specification.
—*DDSI2E Service*: A commercial extension of DDSI2 that enhances the core service with support for some of the features included in the rt-networking service, such as the definition of priority bands.

To deal with the issue of unbounded size for DDSI messages mentioned in Section 6.2, OpenSplice implements a proprietary extension of the standard that allows the maximum size of DDSI messages to be configured. This improves the head-of-line blocking issue and provides a mechanism for controlling priority inversion.

Concerning the concurrency model for the dissemination of information, OpenSplice defines a threadpool composed of a variable number of internal threads depending on which services are being executed. To handle the reception and processing of incoming data samples, OpenSplice relies on two different services: the networking and the domain services. As explained earlier, OpenSplice supports different approaches for the former service; for instance, the rt-netwoking and the DDSI2E services handle the inbound and outbound network communication by means of the creation of dedicated receiver and transmitter threads per network channel, and the DDSI2 service uses one receiver and one transmitter thread for the processing of the I/O operations associated with user data.

The domain service implements the Wait-set and Listener mechanisms defined by the standard to allow data samples to be processed in the context of an application thread or middleware thread, respectively. Under this implementation, a listener thread is spawned per Participant.

Finally, the OpenSplice implementation supports the *Priority Inheritance* synchronization protocol through another proprietary extension of the standard. In this case, this feature is configured per Domain.

7.4. Discussion

As in the analysis of the standards, the following discussion focuses on the mechanisms provided by middleware implementations in terms of the management of processor and network resources.

7.4.1. Managing Processor Resources. The support provided by each implementation included in the analysis in relation to thread scheduling is quite diverse. The usual approach is to provide an interface to configure the scheduling parameters of application threads and delegate their execution sequence to the scheduler provided by the underlying operating system. In this case, the PolyORB-CORBA personality and TAO provide an interface compliant with the RT-CORBA specification, whereas OpenSplice supports a proprietary interface. In relation to PolyORB-DSA, it does not provide any interface and delegates the configuration and scheduling to the Ada runtime. Furthermore, TAO also includes support for another scheduling policy—MIF—which is not defined by the CORBA standard. Under this scheduling policy, middleware is responsible for determining which thread among all those available within the application should execute next.

In relation to the controlled execution of concurrent remote calls, the concurrency patterns implemented in TAO and PolyORB can be used as a reference for a large number of scenarios. However, the use of concurrency patterns based on the dynamic creation of threads, such as Thread-per-Connection or Thread-Per-Session, should be restricted to those situations where the creation of new threads does not jeopardize the determinism of the whole system (e.g., through an admission test at runtime). Moreover, other critical scenarios should also be considered; for example, in flexible scheduling frameworks [Aldea et al. 2006; FRESCOR 2006] where threads execute under contracts, middleware implementations should select a concurrency pattern that minimizes the dynamic change of the scheduling parameters [Pérez and Gutiérrez 2009], as the cost of negotiating or changing contracts is very high.

In general, those concurrency patterns that prevent the dynamic change of scheduling parameters and minimize context switches are used in hard real-time systems. In this case, not all implementations analyzed can be configured to meet these requirements. Thus, TAO allows the application to be configured to select the Leader & Followers pattern that, when applied together with some RT-CORBA mechanisms (e.g., ThreadpoolLanes, Private Connections, and Priority-Banded Connections), provides a set of threads to process the remote request while preserving the end-to-end priority assignment. However, even when PolyORB presents similar mechanisms to TAO, this implementation does not allow this type of configuration, because, among other reasons, it lacks support for some RT-CORBA facilities, such as Private Connections and Priority-Banded Connections. In the case of OpenSplice, the use of Wait-sets along with the concurrency pattern implemented in the rt-networking and DDSI2E services allows the collaboration of application and middleware threads to process incoming data samples while preserving end-to-end priority assignment.

Finally, TAO delegates the choice of the synchronization protocol to the underlying operating system. Nevertheless, the POSIX real-time standard [The Open Group 1998] does not dictate the use of any synchronization protocol by default, so middleware is responsible for configuring or providing the necessary mechanisms to configure the selected protocol. Furthermore, in the case of PolyORB, this aspect is delegated to the Ada language; therefore, applications may configure the predefined Priority Ceiling Protocol as long as the Real-Time Systems Annex is supported by the Ada compiler that is being used. Last, OpenSplice explicitly supports the Priority Inheritance Protocol.

7.4.2. Managing Network Resources. As in the case of the distribution standards, the implementations analyzed most often use general-purpose networks but incorporate some extensions to assign priorities in the communication networks. Thus, both TAO and OpenSplice provide an interface to define the scheduling parameters for the message streams in a proprietary or standard way, respectively. PolyORB does not consider in any case the use of scheduling parameters in the communication networks.

7.4.3. Comparative Summary. This section has discussed the mechanisms provided by middleware implementations for the management of the processing resources (i.e., processors and communication networks). Although the main objective is to analyze the capabilities of distribution standards for developing distributed real-time systems, the analysis of implementations has allowed the lacks and needs of current specifications to be identified.

In particular, the analysis of implementations has focused on the configuration mechanisms for threads and messages scheduling, as well as the concurrency patterns implemented for processing the inbound and outbound data. Table II summarizes the real-time features taken from the analysis of standards and integrates the solutions provided by each implementation. Thus, as in the case of the distribution standards, the implementations mostly support FPS and provide the required mechanisms for establishing the scheduling parameters, either through a standard API (e.g., TAO and PolyORB-CORBA), through a proprietary API (e.g., OpenSplice), or through the programming language (e.g., Ada for PolyORB-DSA). However, there are scenarios where the use of other policies, such as EDF or flexible scheduling based on contracts, could be more appropriate [Liu and Layland 1973; Fohler and Buttazzo 2002]. In relation to communications, the choice and configuration of a communication network may strongly affect the temporal behavior of a distributed system. Furthermore, there are scenarios where both networks and processors should be scheduled together with appropriate techniques [Liu 2000; Sha et al. 2004]. The analysis shows that most implementations use network scheduling based on fixed priorities over Ethernet technology, although these networks do not yet meet hard real-time requirements [Vila-Carbó et al. 2008; Pedreiras et al. 2003] except under very specific conditions [ARINC 2009].

The design and development of efficient concurrency patterns is a key factor in the temporal behavior of implementations. In this case, it is worth noting the large number of different concurrency patterns available in the implementations. TAO and PolyORB can be configured to fit in a wide range of scenarios, but only TAO considers the specific scenario in which avoiding the delay for the highest-priority invocation is required (i.e., by avoiding the dynamic update of the scheduling parameters and minimizing the context switches). OpenSplice decouples the I/O operations through the networking service, but only the rt-networking and DDSI2E services allow transmitter and receiver threads to be created per network channel to ensure an appropriate handling of messages with different priorities.

Finally, TAO delegates the use of synchronization protocols to the operating system. This is worthy of consideration because even the POSIX standard [The Open Group 1998], which can be considered a point of reference for real-time operating systems, does not dictate the use of any protocol by default. In this case, OpenSplice and PolyORB can be used as references, as they allow the configuration of synchronization protocols explicitly or through the mechanisms provided by the Ada programming language, respectively.

8. CONCLUSIONS

This survey has reported an analysis of distribution middleware options from the viewpoint of their suitability for the development of real-time systems. Specifically, the study has analyzed the RT-CORBA, Ada DSA, DDS, and Java DRTSJ standards, with emphasis on the scheduling of processors and networks. In particular, their real-time facilities were grouped into four categories (*scheduling policies*, *setting of scheduling parameters*, *concurrency patterns*, and *controlled access to shared resources*). Based on the previous analysis, we have isolated a set of features and objectives that all distribution standards for real-time systems and/or their implementations should incorporate:

—*Control in the Processing of Remote Calls or Data*: Regardless of the concurrency pattern used, the determinism of the application can only be guaranteed by controlling

Table II. Real-Time Capabilities of Middleware Implementations

	Processor Resources				Network Resources	
Middleware	Scheduling policies	Setting of scheduling parameters	Concurrency pattern	Synchro. protocols	Scheduling policies	Setting of scheduling parameters
TAO	FPS MIF	Client propagated Server declared Priority_Transf.	Reactive Thread per connection Threadpool	RTOS dependent	FPS over IP	Propietary extension
PolyORB	FPS Nonpreemptable Round-robin within priorities EDF	Client propagated Server declared	No_tasking Thread per request Thread per session Threadpool	Priority ceiling	Not defined	Not defined
OpenSplice	FPS Round-robin within priorities	Proprietary extension	Threadpool Dedicated threads per channel (rt-networking /ddsi2e)	Priority inheritance	FPS over IP	Transport priority

the scheduling parameters, even for threads created internally by middleware. This would avoid potential unbounded priority inversions.
—*Enabling Free Assignment of Scheduling Parameters*: Scheduling parameters should be assignable without restrictions throughout the chain of entities that compose the distributed system in order to maximize the schedulability of the system.
—*Support for Different Scheduling Policies*: Although the fixed priority scheduling policy is the most popular and widespread today, there are scenarios where the use of other policies, such as EDF or flexible scheduling based on contracts, could be more appropriate. Therefore, it would be desirable that real-time distribution standards can provide homogeneous support for the configuration of different scheduling policies.
—*Bound the Effect of Priority Inversion*: Middleware should provide sufficient support to guarantee the predictability of the distributed system. On the one hand, by providing mechanisms to facilitate the configuration of synchronization protocols in access to critical sections and, on the other hand, by ensuring a maximum size for network messages.
—*Documentation of the Overhead Introduced by Implementations*: In the analysis of a distributed real-time application, practitioners should be able to consider and evaluate each entity involved in the system, even those created internally by middleware (e.g., the internal threads for I/O management or the network messages belonging to metatraffic). The role and influence of these built-in entities must be clearly specified by the implementation, as these entities can increase the response times of the system by consuming processor and/or network resources. Therefore, distribution standards should include it as implementation requirements.
—*Enabling Timing Analysis of the Complete Application*: Although middleware is executed in the processor, the temporal behavior of the networks has a strong influence on the overall response times. Moreover, in many cases, both networks and processors should be scheduled together with appropriate techniques; therefore, middleware should have the ability to specify the scheduling parameters of both processing resources.

Although distribution standards play a central role in the current development of distributed real-time systems, they usually provide limited support for the real-time configuration of applications. Future advances in the research directions indicated in this survey should help resolve the key open issues identified, such as the real-time mechanisms to bound the effect of priority inversion, the support for different scheduling policies, or the free assignment of any of the scheduling parameters involved in a remote call (i.e., the parameters related to user-defined and built-in schedulable entities for both processors and communication networks).

Regarding high-integrity systems, traditionally this kind of system does not consider the use of distribution standards, because they were not designed with safety-critical systems in mind, and some of their features are hard to certify. This opens up an important research field to apply communication middleware based on distribution standards in, for example, ARINC partitioned systems.

ACKNOWLEDGMENTS

The authors want to thank the anonymous reviewers for their many detailed comments that have allowed the article to be enhanced significantly.

REFERENCES

M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. González Harbour, G. Guidi, J. J. Gutiérrez, T. Lennvall, G. Lipari, J. M. Martínez, J. L. Medina, J. C. P. Gutiérrez, and M.

Trimarchi. 2006. FSF: A real-time scheduling architecture framework. In *Proceedings of the IEEE Real Time Technology and Applications Symposium*. 113–124.

Altran Praxis. 2011. SPARK—the SPADE Ada Kernel (including RavenSPARK), Edition 7.2.

M. Amoretti, S. Caselli, and M. Reggiani. 2006. Designing distributed, component-based systems for industrial robotic applications. In *Industrial Robotics: Programming, Simulation and Applications*, Low Kin Huat (Ed.). ISBN: 3-86611-286-6, InTech, DOI:10.5772/4892.

J. S. Anderson and E. D. Jensen. 2006. Distributed real-time specification for Java: A status report (digest). In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'06)*. ACM, New York, NY, 3–9.

ARINC. ARINC Specification 653p1. 2006. Avionics application software standard interface (ARINC-653).

ARINC. ARINC Specification 664p7. 2009. Aircraft data network, part 7—Avionics Full Duplex Switched Ethernet (AFDX) network.

N. Audsley and A. Wellings. 2001. Issues with using Ravenscar and the Ada Distributed Systems Annex for high-integrity systems. *Ada Letters XXI*, 33–39.

T. P. Baker, 1991. Stack-based scheduling for realtime processes. *Real-Time Systems* 3, 67–99.

J. Bard and V. J. Kovarik. 2007. *Software Defined Radio: The Software Communications Architecture*. Wiley-Blackwell. ISBN: 0-47086-518-0.

P. Basanta-Val, M. García-Valls, and I. Estévez-Ayres. 2010. An architecture for distributed real-time Java based on RMI and RTSJ. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–8.

G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. 2001. The design and implementation of open ORB 2. In *IEEE Distributed Systems Online*, Vol. 2.

G. Bollella and J. Gosling. 2000. The real-time specification for Java. *IEEE Computer* 33, 6, 47–54.

J. L. Campos, J. J. Gutiérrez, and M. G. Harbour. 2004. The chance for Ada to support distribution and real-time in embedded systems. In *Reliable Software Technologies—Ada-Europe 2004*, A. Llamosí and A. Strohmeier (Eds.). Lecture Notes in Computer Science, Vol. 3063. Springer, 91–105.

J. L. Campos, J. J. Gutiérrez, and M. G. Harbour. 2006. Interchangeable scheduling policies in real-time middleware for distribution. In *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies*. Lecture Notes in Computer Science, Vol. 4006. Springer, 227–240.

A. Corsaro and D. C. Schmidt. 2012. The data distribution service—the communication middleware fabric for scalable and extensible systems-of-systems. In *System of Systems*, Dr. Adrian V. Gheorghe (Ed.). ISBN: 978-953-51-0101-7, InTech, DOI:10.5772/30322.

R. I. Davis and A. Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys* 43, 4, 35:1–35:44.

A. Dubey, G. Karsai, and N. Mahadevan. 2011. A component model for hard real-time systems: CCM with ARINC-653. *Software: Practice and Experience (SPE)* 41, 12, 1517–1550. DOI:10.1002/spe.1083. http://dx.doi.org/10.1002/spe.1083

M. Felser. 2001. Ethernet TCP/IP in automation: A short introduction to real-time requirements. In *Proceedings of the 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Vol. 2. 501–504.

G. Fohler and G. C. Buttazzo. 2002. *Introduction to the Special Issue on Flexible Scheduling*, Vol. 22. Springer Netherlands. 10.1023/A:1013489610047.

A. Foster and S. Aslam-Mir. 2005. Practical experiences using the OMG's Extensible Transport Framework (ETF) under a real-time Corba ORB to implement QoS sensitive custom transports for SDR. In *Proceedings of the SDR Technical Conference and Product Exposition*.

E. Freeman, S. Hupfer, and K. Arnold. 1999. *JavaSpaces: Principles, Patterns, and Practice.* Addison-Wesley, Reading, MA.

FRESCOR. Framework for Real-Time Embedded Systems Based on COntRacts. 2006. Project Web page. Retrieved September 2013 from http://www.frescor.org

M. Gillen, J. Loyall, K. Z. Haigh, R. Walsh, C. Partridge, G. Lauer, and T. Strayer. 2012. Information dissemination in disadvantaged wireless communications using a data dissemination service and content data network. In *Proceedings of the SPIE Conference on Defense Transformation and Net-Centric Systems*, Vol. 8405.

A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Tukay, J. Parsons, and D. C. Schmidt. 2008. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming* 73, 1, 39–58.

C. Grelck, J. Julku, and F. Penczek. 2012. Distributed S-Net: Cluster and grid computing without the hassle. In *Proceedings of the 12th IEEE / ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 410–418.

J. J. Gutiérrez and M. Harbour. 1999. Prioritizing remote procedure calls in Ada distributed systems. *Ada Letters XIX*, 67–72.

J. J. Gutiérrez and M. Harbour. 2001. Towards a real-time distributed systems annex in Ada. *Ada Letters XXI*, 62–66.

M. T. Higuera-Toledano. 2012. Adaptive distributed embedded and real-time Java systems based on RTSJ. In *Proceedings of the 15th IEEE International Symposium on Object / Component / Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*. 164–171.

J. Hugues, B. Zalila, L. Pautet, and F. Kordon. 2008. From the prototype to the final embedded system using the Ocarina AADL toolsuite. *ACM Transactions in Embedded Computing Systems* 7, 4, 1–25.

IEEE: The Institute of Electrical and Electronics Engineers STD 802.1D. 2004. Media Access Control (MAC) bridges. http://www.ieee802.org/1/pages/802.1D.html

IEEE: The Institute of Electrical and Electronics Engineers STD 802.1Q. 2006. Virtual bridged local area networks. Annex G. http://www.ieee802.org/1/pages/802.1Q.html

IEEE: The Institute of Electrical and Electronics Engineers STD 802.1Qbb. 2011. Priority-based flow-control. http://www.ieee802.org/1/pages/802.1bb.html

ISO/IEC. 2006. Taft, S. T., Duff, R. A., Brukardt, R., Ploedereder, E., and Leroy, P. 2006. *Ada 2005 Reference Manual. Language and Standard Libraries—International Standard ISO / IEC 8652 (E) with Technical Corrigendum 1 and Amendment 1.* Lecture Notes in Computer Science, Vol. 4348. Springer.

ISO/IEC. 2012. *Ada 2012 Reference Manual. Language and Standard Libraries—International Standard ISO / IEC* 8652:2012(E).

E. D. Jensen, C. D. Locke, and H. Tokuda. 1985. A time-driven scheduling model for real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*. 112–122.

Y. Kermarrec. 1999. CORBA vs. Ada 95 DSA: A programmer's view. *Ada Letters XIX*, 39–46.

K. H. Kim. 2000. Object-oriented real-time distributed programming and support middleware. In *Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE Computer Society, Washington, DC, 10–20.

R. Klefstad, D. C. Schmidt, and C. O'Ryan. 2002. Towards highly configurable real-time object request brokers. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*. 437–447.

H. Kopetz. 2011. *Real-Time Systems: Design Principles for Distributed Embedded Applications* (2nd ed., XVIII). Springer.

S. Lankes, A. Jabs, and T. Bemmerl. 2003. Integration of a CAN-based connection-oriented communication model into real-time Corba. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE Computer Society, Washington DC, 121–129.

J. W. S. Liu. 2000. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ.

C. L. Liu and J. W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* 20, 46–61.

T. Losert, W. Huber, K. Hendling, and M. Jandl 2004. An extensible transport framework for Corba with emphasis on real-time capabilities. In *Proceedings of the 2nd IEEE International Conference on Computational Cybernetics (ICCC)*. 155–161.

A. K. Mok. 1983. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. Ph.D. Dissertation. Massachusetts Institute of Technology.

MultiPARTES. Multi-Cores Partitioning for Trusted Embedded Systems (2011–2014). Project Web page. Retrieved September 2013 from http://www.multipartes.eu

L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. 2010. S4: Distributed stream computing platform. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. 170–177.

K. Nichols, S. Blake, F. Baker, and D. Black. 1998. Definition of the differentiated services field (DS field) in the Ipv4 and Ipv6 headers.

OMG. 2004. Extensible transport framework.

OMG. 2005. Realtime Corba Specification. v1.2. http://www.omg.org/spec/RT/1.2/

OMG. 2007. Data Distribution Service for Real-Time Systems. v1.2. http://www.omg.org/spec/DDS/1.2/

OMG. 2009. The Real-Time Publish-Subscribe Wire Protocol. DDS interoperability wire protocol specification. v2.1. http://www.omg.org/spec/DDSI/2.1/

OMG. 2011. Corba Core Specification. v3.2. http://www.omg.org/spec/CORBA/3.2/

OMG. 2012. Extensible and Dynamic Topic Types for DDS. v1.0. http://www.omg.org/spec/DDS-XTypes/1.0/

C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and . J. Parsons. 2000. The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*. Springer-Verlag, New York, 372–395.

L. Pautet and S. Tardieu. 2000. Glade: A framework for building large object-oriented real-time distributed systems. In *Proceedings of ISORC*. 244–251.

P. Pedreiras, R. Leite, and L. Almeida. 2003. Characterizing the real-time behavior of prioritized Switched-Ethernet. In *Proceedings of the 2nd Workshop on Real-Time LANs in the Internet Age (RTLIA)*.

S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. 2007. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07)*. ACM, New York, NY, 193–202.

H. Pérez and J. J. Gutiérrez. 2009. Experience in integrating interchangeable scheduling policies into a distribution middleware for Ada. In *Proceedings of the ACM SIGAda Annual International Conference on Ada and Related Technologies*, ACM, New York, 73–78.

H. Pérez and J. J. Gutiérrez. 2012. On the schedulability of a data-centric real-time distribution middleware. *Computer Standards and Interfaces* 34, 1, 203–211.

H. Pérez and J. J. Gutiérrez. 2013. Experience with the integration of distribution middleware into partitioned systems. In *Proceedings of the 17th Ada-Europe International Conference on Reliable Software Technologies*. Lecture Notes in Computer Science, Vol. 7896. Springer, 1–16.

H. Pérez, J. J. Gutiérrez, and M. Harbour. 2012. Adapting the end-to-end flow model for distributed Ada to the Ravenscar profile. *Ada Letters* 33, 1, 53–63.

H. Pérez, J. J. Gutiérrez, D. Sangorrín, and M. Harbour. 2008. Real-time distribution middleware from the Ada perspective. In *Proceedings of the 13th Ada-Europe International Conference on Reliable Software Technologies*, F. Kordon and T. Vardanega (Eds.). Lecture Notes in Computer Science, Vol. 5026. Springer, 268–281.

M. Perrotin, E. Conquet, P. Dissaux, T. Tsiodras, and J. Hugues. 2010. The TASTE toolset: Turning human designed heterogeneous systems into computer built homogeneous software. In *Proceedings of the European Congress on Embedded Real-Time Software (ERTS'10)*. Toulouse, France.

I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. 2001. Evaluating and optimizing thread pool strategies for real-time CORBA. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*. ACM, New York, NY, 214–222.

R. Rekik and S. Hasnaoui. 2009. Application of a can bus transport for DDS middleware. In *Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*. 766–771.

R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. 2009. MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, Vol. 5525. Springer-Verlag, Berlin, Heidelberg, 164–182.

M. Ryll and S Ratchev. 2008. Application of the data distribution service for flexible manufacturing automation. *International Journal of Aerospace and Mechanical Engineering* 2, 3, 193–200.

D. C. Schmidt. 1998. Evaluating architectures for multithreaded object request brokers. *Communications of the ACM* 41, 10, 54–60.

D. C. Schmidt. 2005. *TAO Developer's Guide: Building a Standard in Performance.* Object Computing, Inc.

D. C. Schmidt, A. Corsaro, and H. V. Hag. 2008. Addressing the challenges of tactical information management in net-centric systems with DDS. *Journal of Defense Software Engineering*, 24–29.

D. C. Schmidt and C. D. Cranor. 1996. *Pattern Languages of Program Design 2*. Addison-Wesley Longman, Boston, MA, 437–459.

D. C. Schmidt, D. L. Levine, and S. Mungee. 1998. The design of the TAO real-time object request broker. *Computer Communications* 21, 4, 294–324.

D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale. 2001. Software architectures for reducing priority inversion and non-determinism in real-time object request brokers. *Journal of Real-Time Systems* 21, 2, 77–125.

L. Sha, T. Abdelzaher, K. Arzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. 2004. Real time scheduling theory: A historical perspective. *Journal of Real-Time Systems* 28, 2–3, 101–155. DOI:10.1023/B:TIME.0000045315.61234.1e.

L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers* 39, 9, 1175–1185.

Sun Microsystems. 2000. JSR-50: Distributed Real-Time Specification. http://jcp.org/en/jsr/detail?id=50

Sun Microsystems. 2002. JavaTM Message Service Specification. v1.1. http://www.oracle.com/technetwork/java/docs-136352.html

Sun Microsystems. 2004. Java Remote Method Invocation (RMI). http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Sun Microsystems. 2012. Distributed Real-Time Specification (Early draft) http://jcp.org/en/egc/download/drtsj.pdf?id=50&fileId=5028

D. Tejera, A. Alonso, and M. A. de Miguel. 2007. RMI-HRT: Remote method invocation—hard real time. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'07)*. ACM, New York, NY, 113–120.

The Open Group. 1997. DCE: Remote Procedure Calls. v1.2.2. https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?catalogno=t151x

The Open Group. 1998. POSIX .13 IEEE Std. 1003.13-1998. Information Technology—Standardized Application Environment Profile—POSIX Realtime Application Support (AEP). DOI:10.1109/IEEESTD.1999.90558

The Open Group. 2010. Safety Critical Specification for Java. Draft Version 0.78. http://jcp.org/en/jsr/detail?id=302

S. Urueña and J. Zamorano. 2007. Building high-integrity distributed systems with Ravenscar restrictions. *Ada Letters XXVII* 2, 29–36.

T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. 2004. PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of Ada-Europe* (2004-05-04), A. Llamosí and A. Strohmeier (Eds.). Lecture Notes in Computer Science, Vol. 3063. Springer, 106–119.

J. Vila-Carbó, J. Tur-Masanet, and E. Hernández-Orallo. 2008. An evaluation of Switched Ethernet and Linux traffic control for real-time transmission. In *Proceedings of the ETFA*, 400–407.

C. Zhang and V. Tsaoussidis. 2001. TCP-real: Improving real-time capabilities of TCP over heterogeneous networks. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'01)*. ACM, New York, NY, 189–198.

Received October 2012; revised January 2013; accepted September 2013