University of Cantabria
Electronics and Computers Department

Integrating a Real-Time Model in Configurable Middleware for Distributed Systems

Ph. D. Thesis
Héctor Pérez Tijero
Santander, April 2012

University of Cantabria
Electronics and Computers Department

Integrating a Real-Time Model in Configurable Middleware for Distributed Systems

Ph. D. Thesis
submitted for the title of
Doctor
at the University of Cantabria

Héctor Pérez Tijero

University of Cantabria
Electronics and Computers Department

Integrating a Real-Time Model in Configurable Middleware for Distributed Systems

Ph. D. Thesis
submitted for the title of Doctor at the University of Cantabria following the *Science, Technology and Computers* postgraduate programme by

Héctor Pérez Tijero

Supervisor:
Dr. J. Javier Gutiérrez García
Associate Professor

I hereby declare
that the work in this thesis has been undertaken under my supervision in the Electronics and Computers Department at the University of Cantabria and it is suitable for submission.

Santander, April 2012

Héctor Pérez Tijero

J. Javier Gutiérrez García

ABSTRACT

This thesis describes the integration of the end-to-end flow real-time model, which is defined in the MARTE (Modeling and Analysis of Real-Time and Embedded Systems) standard, into distribution middleware, as it can facilitate the development process of distributed real-time systems based on the Model-Driven Engineering (MDE) paradigm. The study focuses on how distribution standards and their implementations guarantee the real-time behaviour of these kinds of applications, thus providing a set of features required to develop analyzable distributed real-time systems. The standards studied are RT-CORBA (Real-Time Common Object Request Broker Architecture), the DSA (Distributed Systems Annex) of Ada, and DDS (Data Distribution Service for real-time systems). The features analysed will contribute to the definition of the endpoints pattern, a new proposal that, when integrated with distribution middleware, enables the use of MDE and schedulability analysis techniques more easily. This thesis also presents a distributed real-time platform supporting different distribution standards, and scheduling policies, and several examples or case studies to validate the features and usability of the endpoints pattern. In addition, this thesis deals with the use of the end-to-end flow model in high-integrity systems by adapting the endpoints pattern to the Ravenscar profile, and also explores the integration of the proposal into a toolset for MDE to enable the automatic generation of Ravenscar-compliant distribution code. Finally, specific implementations of the endpoints pattern are presented for full and restricted Ada.

TABLE OF CONTENTS

LIST OF FIGURES

Acronyms and abbreviations

AFDX:	Avionics Full-Duplex Switched Ethernet
APC:	Asynchronous remote Procedure Call
API:	Application Programming Interface
BIP:	Basic Priority Inheritance Protocol
BTM:	Industrial Robotic Arm
C/S:	Client / Server
CAN:	Controller Area Network
CASE:	Computer-Aided Software Engineering
CDR:	Common Data Representation
CORBA:	Common Object Request Broker Architecture
CSMA / CD:	Carrier-Sense Multiple-Access protocol with Collision Detection
DDS:	Data Distribution Service for Real-Time Systems
DOM:	Distributed Object Model
DR:	Data Reader
DRTSJ:	The Distributed Real-Time Specification for Java
DSA:	Distributed Systems Annex
DTM:	Distributed Transaction Manager
DW:	Data Writer
EA:	Exclusive Areas
EDF:	Earliest Deadline First
ETF:	Extensible Transport Framework
FPS:	Fixed Priority Scheduling
FRESCOR:	Framework for Real-time Embedded Systems based on COntRacts
GA-HI:	Guide for the use of the Ada Ravenscar Profile in high integrity systems

GIOP:	GENERAL INTER-ORB PROTOCOL
HDRT:	HIGH INTEGRITY DISTRIBUTED REAL-TIME APPLICATIONS
HL:	HIGHEST LOCKER PROTOCOL
I/O:	INPUT AND OUTPUT OPERATIONS
IDL:	INTERFACE DEFINITION LANGUAGE
IIOP:	INTERNET INTER-ORB PROTOCOL
IOR:	INTEROPERABLE OBJECT REFERENCE
IP:	INTERNET PROTOCOL
JMS:	JAVA MESSAGE SERVICE
JVM:	JAVA VIRTUAL MACHINE
LLF:	LEAST LAXITY FIRST
MAC:	MEDIA ACCESS CONTROL PROTOCOL
MARTE:	MODELING AND ANALYSIS OF REAL-TIME AND EMBEDDED SYSTEMS
MAST:	MODELING AND ANALYSIS SUITE FOR REAL-TIME APPLICATIONS
MDA:	MODEL-DRIVEN ARCHITECTURE
MOM:	MESSAGE ORIENTED MODEL
OMG:	OBJECT MANAGEMENT GROUP
ORB:	OBJECT REQUEST BROKER
OSI:	OPEN SYSTEM INTERCONNECTION
P/S:	PUBLISHER / SUBSCRIBER
PCP:	PRIORITY CEILING PROTOCOL
PCS:	PARTITION COMMUNICATION SUBSYSTEM
PIM:	PLATFORM-INDEPENDENT MODEL
POA:	PORTABLE OBJECT ADAPTER
PSM:	PLATFORM-SPECIFIC MODEL
QOS:	QUALITY OF SERVICE
RMA:	RATE MONOTONIC ANALYSIS
RMI:	THE REMOTE METHOD INVOCATION
RMS:	RATE MONOTONIC SCHEDULING
RPC:	REMOTE PROCEDURE CALL
RTA:	RESPONSE TIME ANALYSIS
RT-EP:	REAL-TIME ETHERNET PROTOCOL
RTPS:	REAL-TIME PUBLISH-SUBSCRIBE WIRE PROTOCOL
RTSJ:	THE REAL TIME SPECIFICATION FOR JAVA

TCP:	TRANSMISSION CONTROL PROTOCOL
TDMA:	TIME DIVISION MULTIPLE ACCESS
TTA:	TIME TRIGGERED ARCHITECTURE
TTP / C:	TIME TRIGGERED PROTOCOL FOR CLASS C
UB:	UTILIZATION BOUND TEST
UDP:	USER DATAGRAM PROTOCOL
WCET:	WORST CASE EXECUTION TIME
WCRT:	WORST-CASE RESPONSE TIME

LIST OF PUBLICATIONS AND AWARDS

The research work included in this thesis has led to several publications and awards which are detailed below.

Publications

(1) PÉREZ, H. AND GUTIÉRREZ, J. J. "ON THE SCHEDULABILITY OF A DATA-CENTRIC REAL-TIME DISTRIBUTION MIDDLEWARE", JOURNAL OF COMPUTER STANDARDS & INTERFACES, VOLUME 34, ISSUE 1, 2012, PP. 203-211. ISSN: 0920-5489.

(2) PÉREZ, H., GUTIÉRREZ, J. J. AND GONZÁLEZ HARBOUR, M. "ADAPTING THE END-TO-END FLOW MODEL FOR DISTRIBUTED ADA TO THE RAVENSCAR PROFILE", IN PROCEEDINGS OF THE 15TH INTERNATIONAL REAL-TIME ADA WORKSHOP (IRTAW), LIÉBANA (SPAIN). TO BE PUBLISHED IN ADA-LETTERS, 2012. ISSN: 1094-3641.

(3) PÉREZ, H., GUTIÉRREZ, J. J., ASENSIO, E., ZAMORANO, J. AND DE LA PUENTE, J. A. "MODEL-DRIVEN DEVELOPMENT OF HIGH-INTEGRITY DISTRIBUTED REAL-TIME SYSTEMS USING THE END-TO-END FLOW MODEL" IN PROCEEDINGS OF THE 37TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, OULU (FINLAND), 2011, PP. 209-216. ISBN: 978-0-7695-4488-5.

(4) PÉREZ, H., GUTIÉRREZ, J. J. AND GONZÁLEZ HARBOUR, M. "SUPPORT FOR A REAL-TIME TRANSACTIONAL MODEL IN DISTRIBUTED ADA", IN PROCEEDINGS OF THE 14TH INTERNATIONAL REAL-TIME ADA WORKSHOP (IRTAW), PORTO VENERE (ITALY), ADA LETTERS (XXX), 2010, PP. 91-103. ISSN: 1094-3641.

(5) Sangorrín, D., González Harbour, M., Pérez, H. and Gutiérrez, J. J. "Managing Transactions in Flexible Distributed Real-Time Systems", in Proceedings of the 15th International Conference on Reliable Software Technologies, Valencia (Spain), Springer, LNCS 6106, 2010, pp. 251-264. ISSN: 0302-9743.

(6) Pérez, H. and Gutiérrez, J. J. "Revisión del DDS y sus capacidades para tiempo real", in Proceedings of the 3rd congreso español de informática (cedi), Valencia (Spain), 2010, pp. 33-40. ISBN: 978-84-92812-64-6.

(7) Pérez, H. and Gutiérrez, J. J. "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, Florida (USA), ACM, Ada Letters (XXIX), 2009, pp. 73-78. ISSN: 1094-3641.

(8) Pérez, H., Gutiérrez, J. J., Sangorrín, D. and González Harbour, M. "Real-Time Distribution Middleware from the Ada Perspective", in Proceedings of the 13th International Conference on Reliable Software Technologies, Venice (Italy), Springer, LNCS 5026, 2008, pp. 268-281. ISSN: 0302-9743.

(9) Pérez, H. and Gutiérrez, J. J. "CORBA & DSA: Análisis y evaluación de sus implementaciones desde la perspectiva de los sistemas de tiempo real", in Proceedings of the 2nd congreso español de informática (cedi), Zaragoza (Spain), 2007, pp. 11-17. ISBN: 978-84-9732-608-7.

Awards

(1) OUTSTANDING PAPER AWARD FOR THE PAPER "EXPERIENCE IN INTEGRATING INTERCHANGEABLE SCHEDULING POLICIES INTO A DISTRIBUTION MIDDLEWARE FOR ADA" IN THE ACM ANNUAL INTERNATIONAL CONFERENCE ON ADA AND RELATED TECHNOLOGIES (SIGADA), 2009.

(2) XVII ADA-SPAIN AWARD FOR THE BEST ACADEMIC PROJECT RELATED TO THE ADA PROGRAMMING LANGUAGE: "ADAPTACIÓN Y OPTIMIZACIÓN PARA UNA PLATAFORMA DISTRIBUIDA DE TIEMPO REAL DE UN MIDDLEWARE BASADO EN LOS ESTÁNDARES DE RT-CORBA Y ADA", ADA-SPAIN ASSOCIATION, 2009.

1 APPROACH AND OBJECTIVES

This chapter describes the scope of this thesis. First, in Section 1.1, we briefly review the concepts of distributed systems, real-time systems, communication networks and schedulability analysis. Section 1.2 introduces the challenges and the specific features of distributed real-time systems. The most relevant distribution standards for the development of real-time systems are introduced in Section 1.3. We subsequently review the main software tools for the development of predictable systems, as well as some of the most widely used design strategies in Section 1.4. Next, in Section 1.5, we describe the motivations that have led to the development of this thesis, as well as its major objectives. Finally, in Section 1.6, we present the organization of the remaining chapters of the thesis.

1.1 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 of these systems have evolved greatly and they have become especially relevant in the last decade. Today many services are provided transparently to the user and executed in a computer network: Automatic Teller Machines (ATM), cable TV or web services are examples used in our daily lives.

A distributed system is primarily intended to promote collaboration and exchange of information between applications and users. To this end, these systems are characterized by easing access to resources in a transparent and homogeneous way, hiding some of the complexity associated with the distribution of resources and functionalities over a computer network. Since the whole network acts as a single element for the user, these systems must be easily scalable, that is, they must be able to be adapted to changes in the size or in the geographic location of the network. As a drawback, a distributed system necessitates more complexity for design, implementation, debugging and maintenance.

The implementation of a distributed application requires the use of communication services provided by operating systems to exchange information among computers. The direct use of such services by the programmer, even if it usually provides good performance, is a complex and error-prone procedure. 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. According to the mechanism used for distribution, a distributed system can be classified as follows:

- **Distribution based on the direct use of communication services**

  Under this model, the programmer is responsible for performing the distribution explicitly, that is, using the communication services provided by the network and / or operating system.

- **Distribution based on remote procedure calls (RPC)**

  A remote procedure call allows an application to transparently invoke a procedure located on another processing node, that is, solving all the issues related to distribution: location of services, transmission of parameters and results, heterogeneous systems, etc.

- **Distribution based on objects (DOM)**

  This is a higher level abstraction than RPCs and allows object-oriented programming in distributed systems, integrating both technologies and their main features.

- **Distribution based on messages (MOM)**

  In this case, communication between nodes within the distributed system is performed by exchanging asynchronous messages while using, for example, some kind of storage buffers. When the message content is not opaque to the software system and it can operate directly on the content, it can be considered as a new distribution model based on data (also known as *data-centric* model).

In addition to the paradigm or model used to perform the distribution, interaction models should be also considered in the design of distributed applications. In fact, some important quality parameters such as flexibility, decoupling and efficiency strongly depend on which interaction model is selected. This feature has also led to a new classification of distributed systems, which is orthogonal to the distribution model used. Within the context of this thesis, two interaction models have been considered, which are depicted in Figure 1-1 and described briefly below:

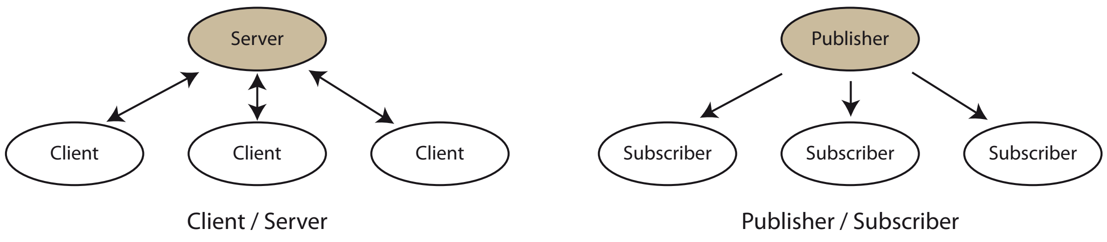

Figure 1-1: Common interaction models for distributed systems

- **The Client / Server (C/S) interaction model**

  Nowadays, distributed systems are mostly based on the *client – server* paradigm. According to this model, clients make requests to the servers and thus both entities are known in advance. An example illustrating this behaviour is the web browser, which acts as a client and requests the web page information. Such information is usually located in a central node known as a server.

- **The Publisher / Subscriber (P/S) interaction model**

  Unlike the previous interaction model, some distributed systems require the distribution of the information through several data producers and consumers. Most of these systems focus on decoupled entities, meaning that entities do not refer to each other to enable communication. In such cases, the model that fits best is based on the *publisher–subscriber* paradigm. According to this model, subscribers do not make requests to a specific publisher, but register their interest in receiving a particular data type. For example, imagine a system to monitor the temperature of several rooms in a building. In this case, subscribers would be interested in receiving the data type *temperature*. Whenever new data of this specific type is available, it will be sent over the network through one or more publishers (in our example, all those publishers which can read a temperature sensor).

Furthermore, real-time systems, those whose logical correctness is based on both the correctness of the outputs and their timeliness, are more and more usual in our daily lives. For example, if the case of a car, you may not know we are using several tens of processors in charge of controlling functions as diverse as radio, injection, brakes, fluid level checks or vehicle air conditioning. However, some of these functions are prioritized over others. For example, the brake control in an emergency stop or the stability control in a skid must take precedence over other actions, such as a passenger who is changing the temperature of the cabin area, and satisfy explicit (bounded) response-time constraints. Thus, the system should not only successfully perform an action, but must execute it in a certain amount of time. Formally, a real-time system is characterized not only by its logical result but also by the time at which the results are produced, what requires a predictable behaviour throughout the system (*software* and *hardware* components).

1.1.1 REAL-TIME SYSTEMS

1.1.1.1 Main concepts and classification

Unlike general purpose systems, a real-time system is defined according to a set of terms that characterize not only its logical behaviour but also its temporal behaviour. Throughout this thesis, we will refer to the processing capacity usage which is required for the execution of a piece of code (for example, reading a file or a simple arithmetic operation) as *operation*, and to the software entity responsible for executing it as *task* (or thread). Each *operation* is executed on a processing resource, typically in a processor (*CPU*) or a communication network, and must be completed within a certain amount of time known as a *deadline*. Finally, every operation has an associated worst-case execution time (*WCET*), defined as the maximum time required to complete the operation in an environment which provides exclusive access to all resources.

Another important timing feature relies on the activation pattern, which can be *periodic* or *aperiodic*, depending on whether operations are triggered at regular intervals of time or not, respectively. In the case of *aperiodic* activations, there are different types of this pattern, such as the sporadic activation pattern (characterized by having a minimum interarrival time between activations) and the *bursty* activation pattern (characterized by having an upper bound on the number of activations that may arrive in a given interval).

A real-time system can be classified according to the consequences of missing a deadline. Real-time systems which cannot fail to meet any deadline are called *hard real-time systems*, as this causes a total system failure. The flight control system of an aeroplane should have these features, for example. However, not all real-time applications are critical. Thus, a video conference could tolerate the loss of certain video quality due to the delay in the arrival of new data to our media player. These systems are called *soft real-time systems*, as they still operate correctly although deadlines are occasionally missed (i.e. their performance is degraded). Such systems must guarantee a certain quality of service (*QoS*). Lastly, there are some mixed real-time systems [BUR09] in which a few missed deadlines are acceptable, but missing more than a few may lead to system failure.

When multiple tasks require simultaneously running operations in a single-processor system, it is necessary to specify a criterion for accessing the CPU, called *scheduling policy*. It consists of a set of rules to determine the exact order for tasks to be executed in the processor. To determine whether a system is schedulable or not (i.e. if all operations can be completed within their deadline when they are executed concurrently), schedulability analysis techniques should be applied to the system. These techniques are used to confirm that system timing requirements are satisfied by predicting the worst-case response time (*WCRT*). WCRT is defined as the maximum time required to complete a specific operation in a shared environment, or an upper bound of it.

In distributed systems, networks can be considered as another processing resource, and messages and their transmission over the communication link are treated as tasks and operations for processors, respectively.

1.1.1.2 Scheduling policies for real-time systems

Static systems are those where the total workload is within known bounds such that *a priori* timing analysis can be performed, and *dynamic system*s are those which do not have a sufficiently predictable workload and therefore require performing some kind of analysis or admission test at runtime. Some common approaches to schedule static systems include the following scheduling policies [KLE93] [LIU00]:

- *Cyclic Executive scheduling*. In this case, the basic scheme is to go cyclicly through a repeating sequence of operations. One possible implementation is to specify the start and finish times of each operation in one or more tables (*table-driven*) at compilation time.
- *Round-Robin scheduling* enables that all tasks to be fairly executed by assigning them the same time slot.
- *Fixed Priority Scheduling* (*FPS*) uses priorities to determine which should be the next task executed on the processor. The priorities assigned to each task are fixed and do not vary with time.
- *Dynamic priority scheduling* also uses priorities but, in this case, priorities may change at runtime. The main algorithms are *Earliest Deadline First* (*EDF*), which assigns higher priorities to tasks with the shortest (nearest) deadline, and *Least Laxity First* (*LLF*), which assigns higher priorities to tasks according not only to their deadline but also their remaining execution time.

Although most hard real-time systems are static and are scheduled through the policies described above, there are systems whose workload varies with time, thus preventing the application of *a priori* timing analysis. In recent years, the *flexible scheduling* approach can be seen as one of the most representative examples of the scheduling of such systems. This scheduling policy is based primarily on the concept of resource reservations through an entity called *contract*. Contracts are negotiated at runtime through an admission test, which can accept or reject the requested resource reservation. This type of scheduling can also be applied to static systems when application requirements are verified at design time. The FIRST [ALD06] and FRESCOR [FRSH11] projects are examples of this type of scheduling.

1.1.1.3 Synchronization protocols

The previous section has reviewed some common approaches for scheduling a set of tasks, that is, to determine the exact order for tasks to execute operations in the processor. However, operations may require accessing other kinds of resources to continue their execution, such as data from external hardware, an area of memory or a file. When several tasks modify the state of these resources (i.e. *shared resources*), it is important to maintain data consistency because most of the resources do not allow simultaneous access, but require mutual exclusion access. Operating systems provide this safe access via synchronization mechanisms which are responsible for ensuring mutually exclusive access to shared resources. Examples of synchronization mechanisms include mutexes, semaphores, monitors or protected objects [BUR09].

From a real-time perspective, mutual exclusion can lead to an *unbounded priority inversion* problem [KLE93]. This problem occurs when a higher priority task “A” is blocked waiting for the release of a shared resource that is being used by a lower priority task “B”. Given this scenario, intermediate priority tasks could be activated in this system that would not allow the processing of task “B” and, consequently, so too task “A”, even though the latter has the highest priority in the system. A real and illustrative example of this problem and its consequences can be found in [WIL97].

The unbounded priority inversion problem can be solved by using appropriate *synchronization protocols* to control the access to shared resources. The main protocols developed for real-time systems using FPS include the *Priority Ceiling Protocol* (PCP), the *Highest Locker Protocol* (HL) and the *Basic Priority Inheritance Protocol* (BIP). These protocols are discussed in detail in the work described in [SHA90] and [RAJ89].

1.1.1.4 Schedulability analysis for real-time systems

To determine whether a set of tasks scheduled by means of a specific scheduling policy can meet their deadlines, a *schedulability analysis* should be applied to the set since it is difficult to estimate the temporal behaviour through simulation for any non-trivial application [XU93]. For single-processor systems, the scheduling process for static real-time systems using FPS integrates the following techniques:

- *A priority assignment strategy*, which is able to perform an automatic and optimized assignment of priorities to a set of tasks.
- *A schedulability analysis,* which predicts the worst-case behaviour of the system when the priority assignment is applied.

Traditionally, *Rate Monotonic Analysis* (RMA) [LIU00], which comprises both priority assignment and analysis techniques, is the most widely used technique in such systems. The RMA theory originated in 1973 when Liu and Layland introduced *Rate Monotonic Scheduling* (RMS), a scheduling algorithm with optimum priority assignment for systems with independent tasks whose deadlines are equal to their periods. This algorithm assigns higher priority to tasks with shorter period, and is *optimal* in the sense that if a system is not schedulable with this priority assignment, then no other assignment will be schedulable either. Under the conditions imposed by RMS, Liu and Layland developed a schedulability test, called the *Utilization Bound Test (UB)*, which makes the schedulability of the system conditional on a maximum percentage of system load. Although these techniques are applied to systems that are very restrictive, they served as a basis for further work that successively removed these restrictions [LEH89]. These works developed other analysis techniques (*Response Time Analysis, RTA*) based on calculating the *Worst-Case Response Time* (WCRT) [JOS86][AUD93][TIN94A].

Although RMA techniques are only suitable for single-processor systems, they can also be applied to distributed systems by modelling communications networks as if they were processors, and network messages as if they were tasks [TIN94B] [KLE93].

Finally, the development of analysis techniques for systems scheduled by dynamic priorities is also important. The analysis techniques proposed in [BAR90] and [SPU96] represent significant contributions in this field.

1.1.1.5 High-integrity systems

During the last 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 non-functional requirements such as deadlines, QoS or integrity.

A high-integrity system can be defined as a system in which a failure may lead to catastrophic consequences (i.e. financial, environmental or personal disasters). Among the most common examples are control systems for commercial aircraft or railway signalling, in which the lives of hundreds of passengers depend on the correctness of the safety-critical system.

A high-integrity system must provide operational guarantees for both hardware (mechanical components, electronics, electrical connections, etc.) and software components. Unlike everyday computing applications, high-integrity software must undergo a *certification process* to verify compliance with certain requirements imposed by a regulatory authority. These requirements are generally reflected in specialized standards for different industries, becoming a basic document in the design and

Table 1-1: Summary of main standards for high-integrity systems

INDUSTRY	ORGANIZATION	STANDARD	NAME
AVIONICS	RTCA	DO 178B	SOFTWARE CONSIDERATIONS IN AIRBORNE SYSTEMS AND EQUIPMENT CERTIFICATION
RAILWAY	BS	EN 50128	RAILWAY APPLICATIONS. COMMUNICATION, SIGNALLING AND PROCESSING SYSTEMS. SOFTWARE FOR RAILWAY CONTROL AND PROTECTION SYSTEMS
NUCLEAR	IEC	880	SOFTWARE FOR COMPUTERS IN THE SAFETY SYSTEMS OF NUCLEAR POWER STATIONS
AUTOMOTIVE	MISRA	ISO/TR 15497	DEVELOPMENT GUIDELINES FOR VEHICLE BASED SOFTWARE
SPACE	NASA	NASA-STD-8719.13	NASA SOFTWARE SAFETY STANDARD
MILITARY	IEEE/EIA	ISO 12207	U.S. SOFTWARE LIFE CYCLE PROCESS

development of high-integrity systems. Table 1-1 lists some of the most widely used standards related to each sector. In general, each of these documents establishes a set of criticality levels according to the software influence in system safety and reliability. For example, within the avionics industry, the standard DO-178B defines five criticality levels depending on the damage that software failure can cause to the system, from *no effect* (level E) to *catastrophic* (level A), requiring a more stringent certification process for the latter.

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 a programming language that restrict the use of those features that are difficult to certify, such as MISRA-C [MIS04] or SPARK [SPA10].

High-integrity systems are usually scheduled by the cyclic executive policy. However, this policy lacks the flexibility necessary to adapt the schedule to changes (due to software errors, changes in the requirements or incorporating additional functionality) and increases the difficulty of designing complex systems. This has motivated the real-time community to attempt to evolve toward a fixed priority scheduling scheme. This change would not only increase the flexibility in the development process, but would also introduce concurrency features in high-integrity software. As a drawback, this also introduces new sources of errors which makes the process of certification harder. One of the proposed solutions is to create safe and analyzable subsets of concurrency facilities. Among the most notable contributions is the *Ravenscar profile* [ADA05], which defines a concurrent but certifiable model for Ada.

1.1.2 DISTRIBUTION MIDDLEWARE

Simple and homogeneous distributed applications can be developed directly using the communications services provided by operating systems. However, in the case of systems composed of dozens of computers with heterogeneous architectures, this development becomes complex and must (1) ensure communication between nodes and (2) address low-level communication details, such as the byte storage format (i.e. *endianess*), word size or floating-point representation used. This complexity can be managed transparently to the user through the use of middleware technology, an intermediate software layer that simplifies the management and programming of applications and which 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*, which is usually based on a formal model that enables the development of systems by assembling reusable software modules (*components*) which have been developed previously by others regardless of the application that will be used.
- *Adaptive middleware*, which enables the reconfiguration of distributed applications to modify functionalities, resource usage, security settings and so on.
- *Context-aware middleware,* which is able to interact with the environment where distributed applications execute and take action to make changes at runtime.

The development of this thesis focuses 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 which usually consists of the following basic features (see Figure 1-2): (1) *addressing* or the assignment of references to objects to denote their location, (2) *marshalling* or the transformation of data into a representation suitable for transmission over the network, (3) *dispatching* or the assignment of each request onto an execution resource for processing, and (4) *transport* or the establishment of a communication link for exchanging network messages.

Furthermore, our approach only considers distribution middleware that is based on standards due to its stability and impact on the industry. Currently, there are many standards that fall within any of the above distribution paradigms. Thus, among the most representative examples of distribution models based on RPCs are the OSF / DCE standard [DCE97] or the Distributed Systems Annex of Ada language (DSA) [ADA05].

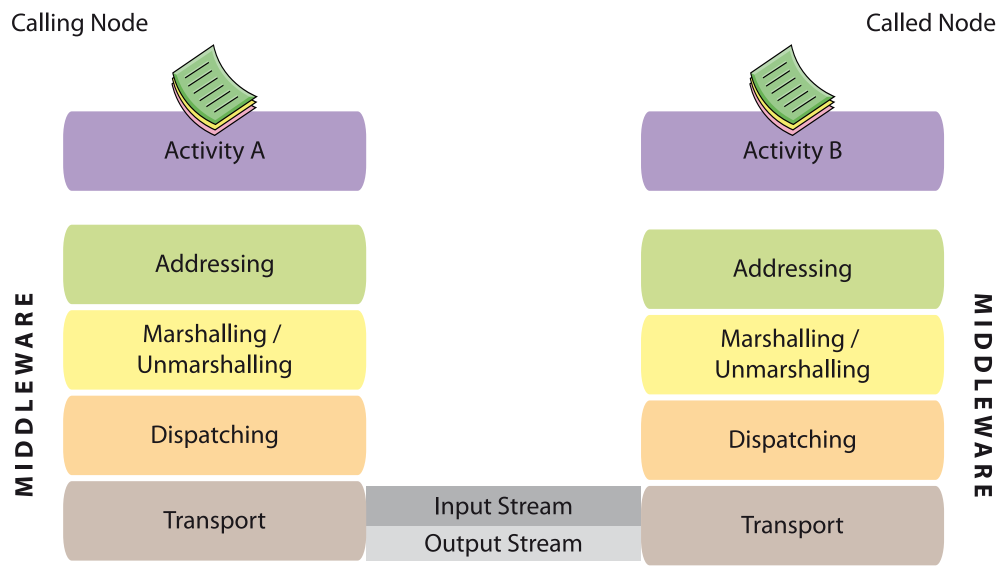

Figure 1-2: Basic services provided by distribution middleware

In relation to the DOM model, this paradigm is probably the most relevant in current industrial applications, and an important example is the CORBA standard [COR03]. Other examples of the DOM model are the Java Remote Method Invocation (RMI) [RMI04] or the previously mentioned Ada DSA, which also allows distribution based on objects. Examples of the MOM model are the Java Message Service (JMS), a *de facto* standard, and the Data Distribution Service for Real-Time Systems (DDS) [DDS07]. However, the latter is often included in the *data-centric* category, since the contents of exchanged messages are not opaque to middleware and can be handled directly. Table 1-2 summarizes these standards and their main features.

However, not all these standards are suitable for developing distributed real-time applications, as they require a set of mechanisms and capabilities to ensure determinism: for example, task 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), be added as an extension to the original distribution model (e.g. CORBA and RT-CORBA [RTC05]) or be considered independent of distribution mechanisms, as in the case of the Ada language.

Table 1-2: Classification of distribution standards

STANDARD	DISTRIBUTION PARADIGM	INTERACTION PARADIGM	OBSERVATIONS
OSF/DCE	RPC	C/S	
ADA DSA	RPC DOM	C/S	
CORBA	DOM	C/S	
JAVA RMI	DOM	C/S	
JMS	MOM	P/S	
DDS	MOM	P/S	DATA-CENTRIC

1.2 DISTRIBUTED REAL-TIME SYSTEMS

The strong growth in the volume of data and events that a current system must process, along with the fast development of technologies for communication networks, has provoked the need to extend the functionality of real-time applications over distributed environments as was previously announced by J. Stankovic in the nineties [STA92].

Within this scenario, middleware should provide mechanisms to guarantee predictability over the whole application. In particular, distributed real-time systems introduce the following new challenges for the developer:

- *The influence of communication networks on the system response times.* Although time constraints are usually associated with task execution, these constraints are directly reflected on the messages exchanged between tasks in a distributed system, that is, the execution time of a task depends on the time spent by this task in sending and / or receiving messages as networks can be viewed as shared resources. Furthermore, the evaluation of the end-to-end communication delay is difficult as it depends on the network topology used.
- *Appropriate techniques for schedulability analysis.* Techniques developed for single-processor systems cannot be applied, and should be extended to incorporate diverse factors that may affect the temporal behaviour of the whole application, such as:

- *Optimal allocation of resources.* In distributed systems, it is necessary to allocate n tasks to a number p of processors so that they can satisfy the time constraints of the critical tasks. The process of obtaining an optimal allocation belongs to the family of problems known as NP-complete (non-deterministic polynomial-time) [BUR91].
- *Allocation of scheduling parameters.* Although there are techniques for the optimum assignment of scheduling parameters in single-processor systems, these techniques are not optimal for distributed systems where the problem becomes NP-complete [MOK78] [BUR91].
- *Jitter.* It represents variations in the activation of tasks or in the transmission of messages. This variation, which could be caused by the inaccuracy of hardware, affects the temporal behaviour of real-time distributed systems, such as in multimedia systems where jitter produces undesirable effects (e.g. audio distortion or annoying flickering in the video).

• *Appropriate real-time model to integrate both the previous challenges.* Middleware should integrate a real-time model that allows schedulability analysis techniques to be applied and network messages and tasks scheduling to be configured.

An overview of the basic differences in the scheduling of single-processor, multiprocessor and distributed real-time systems can be found in [BUR91]. This thesis will only address those distributed environments with bounded latencies, which is the traditional scenario for hard real-time systems. The issues described above are now discussed in turn.

1.2.1 REAL-TIME SYSTEM MODEL

The objective of this section is to introduce a model of the behaviour of a real-time system which will be used throughout the rest of the thesis. Viewed from the perspective of an outside observer, a real-time distributed system can be modelled as a set of distributed *transactions* [TIN94B] [HAR01], as illustrated in Figure 1-3. A transaction is defined as an entity that holds a set of tasks and network messages with some precedence relationship between them, sharing either functional or temporal attributes (e.g. tasks that are activated with the same period). Under this model, each transaction may be triggered by the arrival of one or more external event thereby causing the execution of operations on the processors or in the networks. In the example shown in Figure 1-3, the external event is represented by a timer. After the timer expires, the sequence of Asynchronous remote Procedure Calls (APC) is performed, which implies

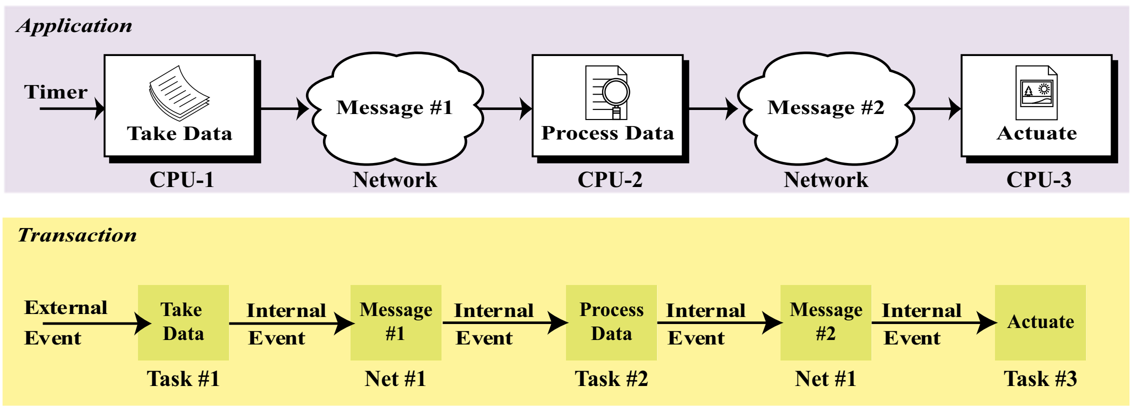

Figure 1-3: Modelling applications according to the end-to-end flow model

the execution of the following operations: *Take Data*, *Process Data* and *Actuate* subprograms and the transmission of two network messages. From the perspective of the real-time transactional model, on completion of *operations* events are generated internally to the transaction which may in turn activate other operations as illustrated in Figure 1-3.

A system representation using this model is analyzable through different schedulability analysis techniques, such as those included in MAST[1] (Modelling and Analysis Suite for Real-Time Applications) [HAR01]. MAST is a software toolsuite that offers an open set of tools for modelling, analysis and design of real-time systems. It proposes a system model also based on the aforementioned *real-time transactional model* [TIN94B] [HAR01]. However, the real-time transactional model is currently known as *end-to-end flow model* in the MARTE (Modeling and Analysis of Real-Time and Embedded Systems) modelling standard [MAR08] and, since MAST is being aligned with MARTE's standardized terminology [HAR12], the latter terminology will be used throughout this thesis. Therefore, the end-to-end flow model consists mainly of the following entities (see Figure 1-4):

- *Steps*. This entity includes two concepts: firstly, the operation to be executed (for example, the execution of a piece of code on a processor or sending a message through a network); secondly, the schedulable entity to execute the operation.
- *Events*. These are the elements responsible for triggering the execution of a *step*. There are two types of events: *Workload_Event*, which characterizes the nature of end-to-end flows (i.e. periodic or aperiodic), and *Internal_Event*, which represents the flow of *steps* within the end-to-end flow.

1. MAST is available at http://mast.unican.es

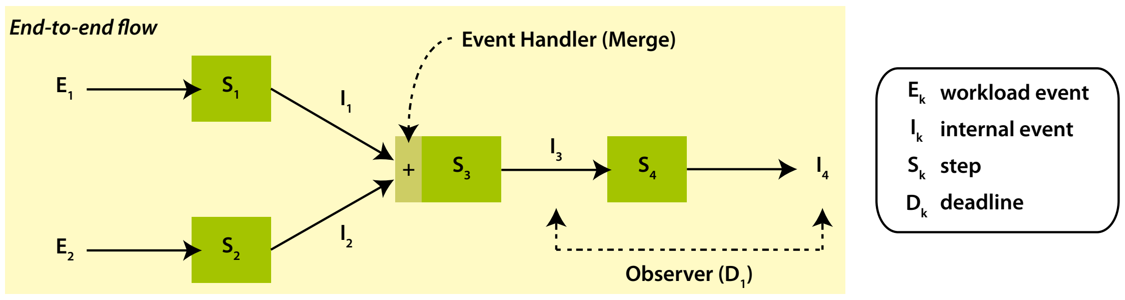

Figure 1-4: End-to-end flow model proposed by MARTE standard

- *Event Handlers*. They represent the actions to be executed on the arrival of an event. These actions may be linear, in which a single input event generates a single output event (for example, the execution of a *step*) or nonlinear, in which a single action can be activated by the combination of one or more events or can generate multiples output events (e.g. sending a multicast message is modelled by a *Fork* or the use of various sources of data can be matched with a *Merge* (see Figure 1-4)).
- *Observer*. The Observer is an entity responsible for monitoring a set of parameters associated with *Internal_Events*. For example, it allows temporal parameters to be evaluated (e.g. deadlines or maximum jitter) at a specific point in the end-to-end flow.

Therefore, the real-time transactional model or *end-to-end flow model* plays a central role in the development of real-time distributed systems, as it is part of a relevant modelling standard within the real-time community and also includes Computer-Aided Software Engineering (CASE) tools such as MAST to facilitate the development process for real-time engineers. Furthermore, this real-time model provides a simple and complete representation of system entities where schedulability analysis techniques can be directly applied.

1.2.2 SCHEDULABILITY ANALYSIS FOR DISTRIBUTED REAL-TIME SYSTEMS

Schedulability analysis techniques proposed for single-processor systems are not directly applicable to distributed systems and should be revised (1) to include the influence of communication networks in response times and (2) to consider the precedence relationships between tasks allocated in different nodes.

Unlike single-processor systems, distributed systems requires the assignment of tasks to appropriate nodes. Tasks can be allocated in a static or dynamic way to the nodes. If we focus on systems with static allocation of tasks and messages to processors and networks, respectively, then the problem is simplified and two fundamental issues must be considered: (1) the schedulability analysis necessary to determine the schedulability of processing resources (processors and communication networks), that is, whether the system can meet deadlines even in the worst-case scenario; and (2) the allocation of scheduling parameters to schedulable entities (tasks for processors and messages for communication networks) to maximize the possibility of meeting these requirements.

Previously, in section 1.1.1, a set of scheduling policies for real-time systems was listed. For distributed systems, the easiest scheme is to use the cyclic executive approach by laying out a complete schedule to cycle through a sequence of steps. This type of scheduling, which is decided at compilation time, can be applied to single-processor, multiprocessor or distributed systems. In the latter case, it must also use a cyclic approach for communications (e.g. through TDMA techniques). Once the schedule has been constructed, then no further schedulability test is required (*proof by construction*). One example of this kind of architecture is the Time-Triggered Architecture (TTA) [KOP11], which uses a complex heuristic function to guide the search for a feasible and optimized schedule (e.g. *phase aligned*[1]). Drawbacks of this scheme include the lack of flexibility for adaptation to non-periodic end-to-end flows and the high cost associated with the redevelopment of the schedule when software is modified [BUR09].

Cyclic executive scheduling is mainly used in high-integrity systems. Currently, avionics systems follow the ARINC 653 standard [ARINC06], which allows multiple applications of different software levels to be hosted on the same hardware. Each piece of application software is called a *partition*, and it has its own memory space and one or more dedicated time slots. Within each partition, multitasking is allowed and thus other scheduling policies can be applied.

Other alternative approaches have also been considered. These determine which task should execute at each time by the use of one or more scheduling parameters. Although the RTA analysis is exact (that is, it obtains the exact WCRT of a given task) for single-processor systems, this technique is no longer optimal for multiprocessors or distributed systems [MOK78], mainly due to the problem of deferred activation *or jitter.*

Jitter can significantly affect the schedulability of the system [KLE93]. Among other factors, this effect is associated with the inaccuracy of the hardware used such as the resolution (granularity) of the system clock. There is also another major source of jitter in systems with precedence relationships. In this case, the activation time of tasks is not perfectly periodic, but depends on the completion time of the triggering task, which is variable. The effect of deferred activation is usual in distributed systems where tasks are often activated from messages [GUT96].

1. Send slots in the communication subsystem are available immediately after the WCET of the previous processing action belonging to the same sequence

Therefore, schedulability techniques for distributed systems should consider how the variability in the execution of a task or the transmission of a message will affect another part of the system [BUR09]. For fixed priorities, Tindell and Clark [TIN94C] [PAL97] proposed an algorithm to calculate an upper bound of response times for distributed systems. This is called *holistic schedulability analysis* and takes into account the *jitter* effect by using an iterative algorithm. This technique assumes that all system tasks and messages are independent and therefore it leads to pessimistic results. To reduce this pessimism, Tindell introduced the concept of *offsets* (activation phase) [TIN94B] to avoid simultaneous activation of tasks / messages with precedence relationships. This technique only considered *static offsets*, but this restriction was eliminated by Palencia and Harbour in [PAL99]. However, the exact calculation of WCRT through these techniques is intractable for large systems [TIN94B] so alternative methods are applied to obtain upper bounds for the response times [PAL99][RED04][MAKI08].

For systems with EDF scheduling, Spuri [SPU96] [SPU96-2] adapted *holistic* schedulability analysis, while [PAL05] extended the *offset-based* schedulability analysis techniques. Moreover, the authors in [RIV10] proposed another technique based on the use of local deadlines[1].

Finally, the work included in [RIV11] integrated these techniques and presented a set of tools able to analyse heterogeneous systems (i.e. processor and communication networks having either fixed priority or EDF schedulers). These works are the basis for the transactional model [TIN94B], the real-time model previously discussed, which will be used in the context of this thesis.

1.2.3 REAL-TIME COMMUNICATION NETWORKS

Nowadays, a real-time distributed system may consist of several tens of processors interconnected by one or more communication networks (for example, the Volvo XC90 architecture consists of a minimum of 4 communication networks which interconnect up to 40 microprocessors [HRI05]). Software for this type of systems consists of several concurrent tasks that communicate with each other by exchanging messages over communications networks. Similarly to what happens with tasks in processors, networks are shared resources that can be simultaneously accessed by multiple nodes in a distributed system. A collision is the result of two or more nodes on the same network attempting to transmit a message at exactly the same time. Hence, it is necessary to establish an order for nodes to access the transmission medium. This is known as network scheduling policy, and is one of the major differences between general-purpose and real-time communication protocols. The latter are specifically designed to eliminate or deterministically avoid the existence of collisions on the network.

1. Deadlines are referenced to the local clock

Although a large number of network protocols are suitable for real-time systems, the literature of real-time computing [BUR09] [KOP11] usually classifies them according to the technique used to solve the problem of access to the transmission medium:

- *Time Division Multiple Access protocols (TDMA)*. This is a static scheduling scheme in which each node is assigned a time slot for transmitting network messages. Therefore, this technique eliminates the collisions only if there is a precise timing synchronization of all clocks of the distributed system.
- *Token passing protocols*. In this case, the protocol defines a particular message type, called token. The token is passed between nodes and authorizes the node to communicate. Since there is only one token within the distributed system, it avoids the existence of collisions.
- *Master-Slave protocols*. Such protocols are based on a master node that controls which node can access the communication network. Therefore, this master node ensures the suppression of collisions and so this technique is often implemented with redundant master nodes to avoid a single point of failure.
- *Priority-based protocols*. Under this scheme, nodes must specify a priority associated with each network message. These protocols tend to have two phases. In the first phase, known as priority arbitration, each node indicates the priority of the message it intends to transmit, and the node that owns the highest priority message will be awarded the right to transmit its message. Subsequently, the message is transmitted in the second phase.

Nowadays, a wide variety of communications networks exist for real-time systems. In the last decades, the trend in the industry was toward the development of special-purpose networks capable of providing quality of service in a specific scenario: CAN [CAN91] and FlexRay [FLEX05] for automotive systems, PROFIBUS [IEC00] for industrial automation and control systems or ARINC-629 [ARINC99] for avionics. However, the increasing need to reduce costs and development time has promoted the use of commercial hardware and protocols such as PROFINET [IEC07] or ARINC-664 Part 7 [AFDX09], which include the use of Ethernet technology that is widely used in general-purpose systems.

The remainder of this section focuses on reviewing a subset of networks which is able to provide soft or hard real-time guarantees, which are classified according to their usage sector.

1.2.3.1 Specific-purpose real-time communication networks

Automotive systems

The CAN bus (Controller Area Network) [CAN91] was developed in the mid 80s by Robert Bosch as a serial communication bus for applications in the automotive field and subsequently standardized in ISO 11898. The Media Access Control protocol (MAC) used is CSMA / CD + AMP (Carrier-Sense Multiple-Access protocol with Collision Detection and Arbitration on Message Priority) with deterministic resolution of collisions. This protocol is optimized for small size messages (up to 8 bytes) that are transmitted with a bit rate of up to 1Mbps using a fixed priority scheduling policy.

The FlexRay protocol [FLEX05] was developed recently by a consortium that includes, among others, BMW, DaimlerChrysler, Motorola, GM, Bosch and Philips, as a natural replacement for the CAN bus in automotive control systems. The main feature of FlexRay is the efficient integration of communications based on time division (TDMA) and event-driven techniques. Moreover, it has a bit rate up of to 20Mbps and can transmit messages with a size between 2 and 354 bytes.

TTP / C (Time-Triggered Protocol for Class C) [KOP93] is a communication protocol developed to support critical applications that require real-time guarantees and fault tolerance in the automotive field. The medium access is performed using TDMA techniques and the information about which node should transmit and when it should do so is stored in a static table belonging to each node. This protocol supports transmission speeds of 500Kbps, 1Mbps, 2Mbps and 25Mpbs.

Industrial automation and control systems

The PROFIBUS protocol [IEC00] is included in the IEC61158 standard and is designed to support deterministic communications between computers, programmable logic controllers and devices such as sensors and actuators. As with MAC protocol, it uses a token-passing architecture that guarantees the exclusive use of the communication bus, with a bit rate of up to 12Mbps and a fixed-priority scheduling policy.

Avionics

The ARINC-629 [ARINC99] standard describes communication networks for avionics systems. This specification defines a multilevel protocol for data communication via a bidirectional multiple-access bus. The MAC algorithm uses CSMA / CA (Carrier-Sense Multiple-Access protocol with Collision Avoidance) and TDMA techniques for aperiodic and periodic network traffic, respectively. Furthermore, it supports a bit rate of up to 2Mbps.

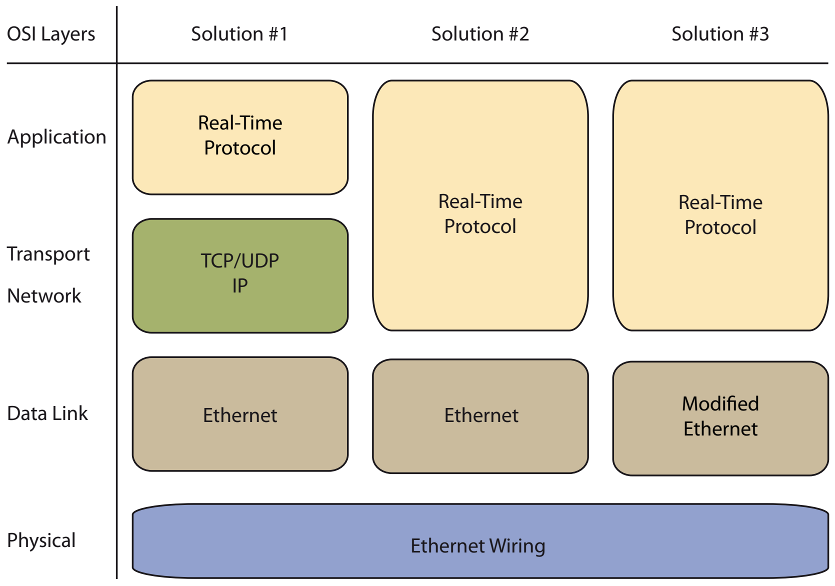

Figure 1-5: Structure of Ethernet-based solutions for real-time systems

Spacecraft systems

SpaceWire [SPW08] is a standard promoted by the European Space Agency (ESA) that enables the interconnection of aerospace systems through a full-duplex high-speed network (from 2 to 200Mbps). This network uses point-to-point serial connections between nodes, but also supports the use of special routers to implement more complex networks. Therefore, this type of connection eliminates collisions in the transmission medium. A fixed-priority scheduling policy is used for network scheduling.

1.2.3.2 Real-time communication networks based on Ethernet technology

Ethernet is a suitable networking technology for local area networks. Standardized as IEEE 802.3, Ethernet was originally designed to interconnect general-purpose computers. However, the desire to incorporate a real-time element into this increasingly popular protocol has led to an evolving field of research during the last 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 CSMA / CD (Carrier-Sense Multiple-Access protocol with Collision Detection), which uses non-predictable back-off algorithms to avoid a message collision. To overcome the lack of predictability of CSMA / CD, several solutions have been proposed. In [DEC05], the author classifies these solutions according to which part of the protocol stack has been modified, as is shown in Figure 1-5. In particular, three solutions are presented which are described briefly below.

Real-time networks on top of TCP/IP or UDP/IP

There are some solutions that include some kind of temporal control over the messages but, by themselves, do not guarantee a strictly bounded transmission time since they do not change the media access control protocol defined in Ethernet. Some examples include:

Real-Time Publisher Subscriber protocol (RTPS) [IEC04] [RTPS09], based on the publisher - subscriber paradigm, integrates QoS parameters into communications. These parameters are used to detect critical delays in the transmission of packets, configure the availability and use of resources, adjust the system reliability, etc.

Real-Time Protocol (RTP) [RFC3550] is oriented to the management of multimedia content and allows the QoS over the communication to be synchronized and adapted.

PROFINET [IEC07] is a protocol divided into profiles according to the required level of determinism. In the real-time profile, it is a protocol based on asymmetric architecture with a master node that controls multiple slave nodes. Although it can be used on commercial switches, the hard real-time profile requires the use of specific hardware.

Real-time networks on top of modified Ethernet

This group consists of protocols that modify the Ethernet standard and its infrastructure to provide temporal guarantees including, for example, those that modify the original MAC layer.

The authors of [LEE98] combine a token-bus (IEEE 802.4) with the physical layer of IEEE 802.3. In this way, a token-bus network is obtained but using Ethernet technology. This technique reaches a bit rate of 5Mbps over 10Mbps Ethernet links.

The CSMA-DCR protocol [LELAN93] proposes deterministic avoidance of collisions by creating a hierarchy of priorities as a binary tree composed of all the nodes on the network.

Real-time networks on top of Ethernet

These solutions do not make any modification to Ethernet but implement control mechanisms to suppress the collisions over the transmission medium. In general, they require a dedicated Ethernet segment to guarantee bounded transmission times. The following protocols can be included:

Real-Time Ethernet Protocol (RT-EP) [MAR05], a token-passing protocol over a logical ring that uses a fixed priority scheduling policy.

Flexible Time-Triggered Ethernet protocol (FTT-Ethernet) [PED02] combines both time-driven and event-driven communications. It is a master-slave protocol where the master node controls the system requirements, the scheduling policies, and the admission control at runtime. The possibility of failure on the master node is dealt with by using redundancy of master nodes, thus avoiding a single point of failure.

RTnet protocol [KISZ05] presents an abstraction layer that allows different techniques to be used to access the transmission medium. The existing version provides support for two options: the standard CSMA / CD and a TDMA technique. As we said earlier on in this section, the former cannot be applied for hard real-time systems. Furthermore, RTnet also includes deterministic implementations of other higher-level protocols, such as UDP / IP or ARP (Address Resolution Protocol), to facilitate the encapsulation of general-purpose network traffic.

Finally, Switched Ethernet technology is presented as an alternative that has become particularly relevant in recent years. This technology integrates the use of standard Ethernet switches and full-duplex communications to avoid collisions in the transmission medium. Furthermore, it can use a fixed priority scheduling policy [VBLAN06] which provides real-time guarantees under certain conditions [PED03] [VILA08]. For example, the *Avionics Full-Duplex Switched Ethernet* protocol (AFDX), which is defined in the ARINC-664 specification [AFDX09], uses this technology for aircraft data networks.

1.3 REAL-TIME DISTRIBUTION MIDDLEWARE

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), while middleware transparently establishes and manages communication between nodes within the distributed system.

However, general-purpose middleware cannot be applied directly to real-time systems. In general, the distribution process (see Figure 1-2) presents several potential sources of indeterminism, including marshalling / unmarshalling of data, transmission / reception queues for network messages, delays in transport service or requests dispatching. Real-time middleware aims to solve these issues by implementing predictable mechanisms, such as the use of specific-purpose real-time communication networks or the management of scheduling parameters. Consequently, this kind of

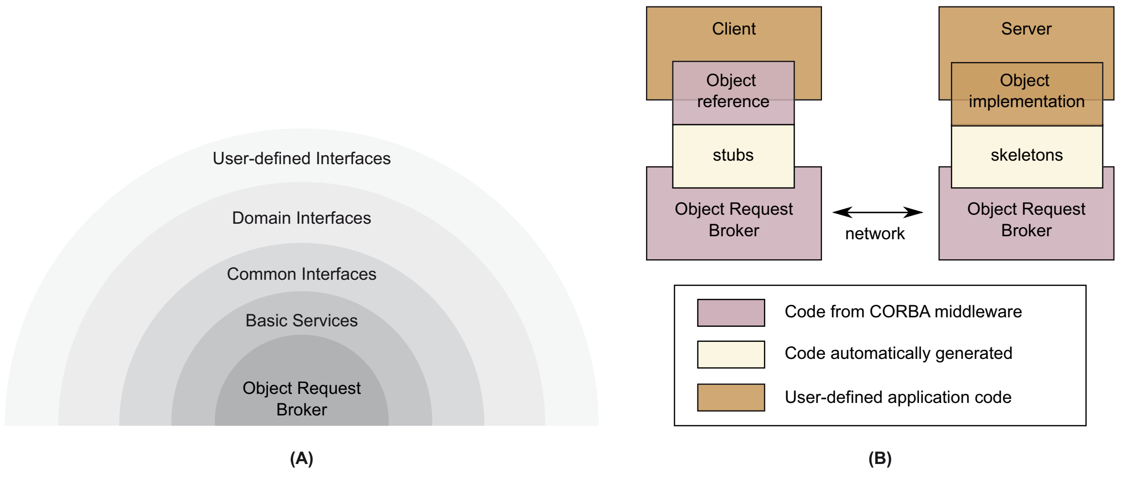

Figure 1-6: Components and architecture for CORBA distribution model

middleware addresses not only the distribution issues but also should provide developers with mechanisms which allows the temporal behaviour of the distributed application to be determined.

As we said earlier on in this chapter, this thesis will focus exclusively on middleware technologies based on standards, due to their impact and stability. The remainder of this section introduces the most notable distribution standards for real-time systems.

1.3.1 CORBA AND RT-CORBA

The Common Object Request Broker Architecture (CORBA) [COR03] 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[1]). An overview of the CORBA architecture is shown in Figure 1-6 (A). It is comprised of the following components:

- **Object Request Broker** (ORB). It represents the core of middleware and is responsible for coordinating the communication between client and server nodes.
- **System interfaces**. They consist of a set of interfaces grouped according to their scope which include: (1) a collection of Basic Services which support

1. http://www.omg.org/

the ORB (e.g. location of remote objects, concurrency, persistence, etc.); (2) a set of Common Interfaces across a wide range of application domains (database management, compression, authentication, etc.); (3) a group of interfaces for a particular application domain (Domain Interfaces) such as telecommunications, banking, finance, etc; and (4) User-Defined Interfaces (i.e. not standardized).

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

- Language-independent middleware (multi-language).

  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 of 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 1-6 (B) and described below:

- *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*. They 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*. It is an identifier 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, although 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, which is called RT-CORBA [RTC05]. This set of extensions includes new mechanisms such as the RT-ORB, priority mappings or scheduling policies that enable its use for both non-critical (e.g. travel agencies or on-line shopping cart) and critical systems (e.g. real-time control systems).

1.3.2 THE ADA DISTRIBUTED SYSTEMS ANNEX (DSA)

The Ada programming language [ADA05] is an international standard that includes an annex dedicated to developing distributed applications: Annex E or Ada Distributed Systems Annex (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 which, 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 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*”.

Partitions communicate with each other by exchanging data through remote procedure calls (*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). The partitioning of an application through the DSA is not defined by the standard but is implementation-defined.

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 allows compilers to generate stubs for a standard interface without being concerned with the underlying

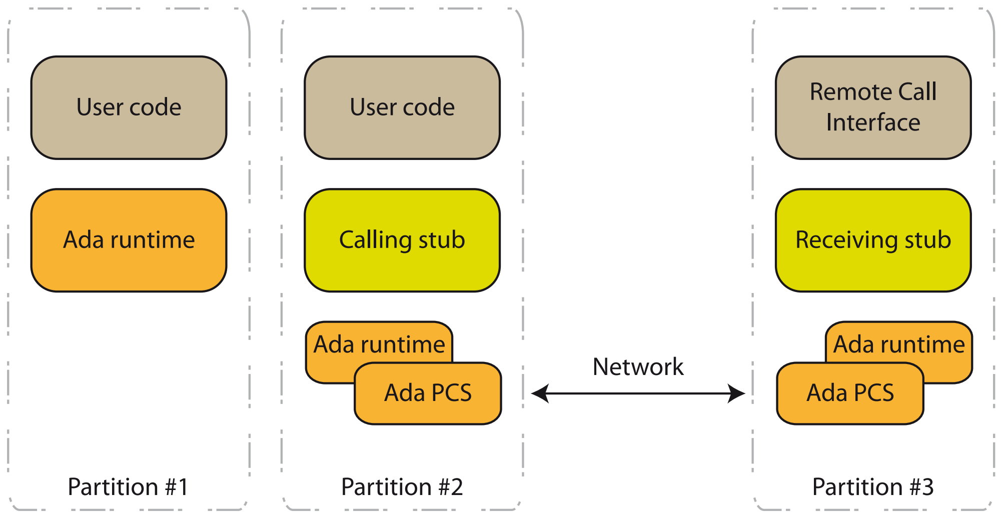

Figure 1-7: Components and architecture for DSA distribution model

implementation. Despite this standardization effort, the latest revision of the programming language [ADA05] allows the use of alternative interfaces to facilitate the interoperability with other middleware (e.g. CORBA).

The components of the distribution model proposed by the DSA are illustrated in Figure 1-7. This figure represents three types of partitions: a local Ada partition (*partition #1*), a partition that requires remote services (*partition #2*) and a partition that provides these services (*partition #3*) through a remote call interface.

Lastly, the specification also defines a set of attributes and rules to check the internal consistency of a distributed application. Since these applications can be executed in separate nodes, these mechanisms are intended to ensure the use of the same version of source code for generating each partition.

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 are some previous works that show it can be used for real-time applications [LOP04][LOP06].

1.3.3 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 Data Distribution Service for Real-Time Systems (DDS) [DDS07] aims to facilitate the exchange of data in these kinds of systems through the publisher-subscriber paradigm. Unlike other

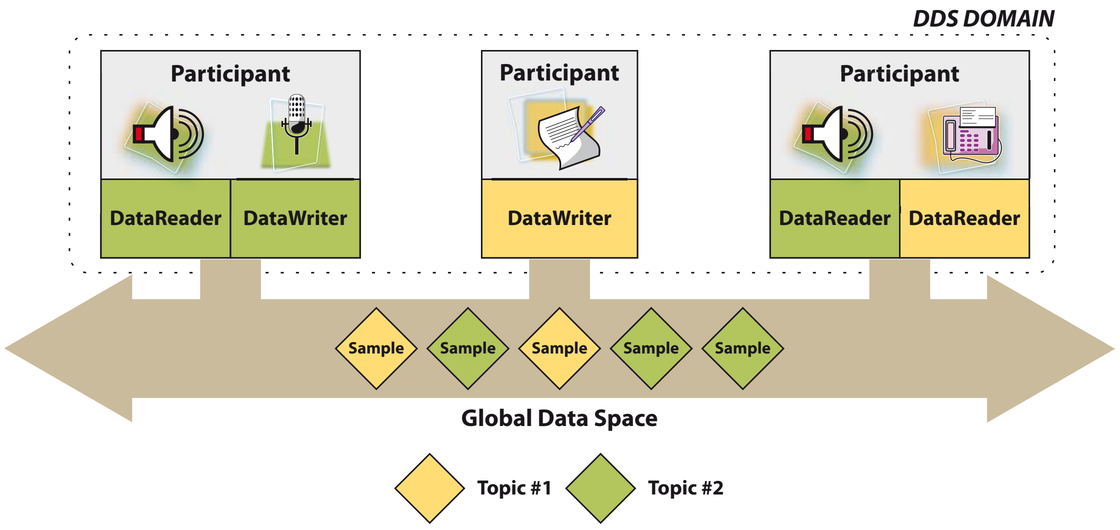

Figure 1-8: Communication model for DDS

specifications that follow this paradigm, the communication model proposed by the DDS is *data-centric*. This implies that middleware is aware of the content of the information exchanged and several QoS can be applied to it (e.g. data filtering).

As with most of the standards defined within the OMG, the DDS supports multi-language and multi-platform capabilities by using the IDL language [COR03] to define shared data types and the DDS Interoperability Wire Protocol [RTPS09] to interoperate among different implementations, respectively.

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 a 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 whish to share information with others can use this Global Data Space to declare their intent to publish data through the *Data Writer* (DW) entity. Similarly, applications that need to receive information can use the *Data Reader* (DR) entity to request particular data. *Publisher* and *Subscriber* entities are containers for several DWs and DRs, which share a common QoS, respectively. 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, while middleware will establish and manage the communication almost transparently. The example in Figure 1-8 shows a distributed system which 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:

- *Time,* because data samples could be stored and retrieved later (for example, 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 while 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 which sets the main guidelines for performing the communication among entities: the DDS Interoperability Wire Protocol. This protocol aims to guarantee the interoperability among different implementations by using the standard Real-time Publish-Subscribe Wire Protocol (RTPS) [RTPS09] together with the Common Data Representation (CDR) defined in CORBA [COR03]. Although this specification is focused on IP networks, any other real-time network protocol could be used.

Finally, although DDS has been designed to be scalable, efficient, and predictable, few researchers have evaluated its real-time capabilities.

1.3.4 THE JAVA APPROACH

Besides the distribution standards, there are other non-standard solutions which 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, especially those aspects related to the management of internal resources such as memory or processor scheduling [BAS07]. Nevertheless, several lines of research aims to adapt the language to a deterministic model not only for single-processor environments but also for distributed ones [BAS07] [TEJ07]. For distributed real-time systems, one of the most notable research works is The Distributed Real-Time Specification for Java (DRTSJ) which integrates two existing Java technologies:

- The Real-Time Specification for Java (RTSJ) [BOL00] defines a new Java specification to address the limitations of the language to be used in real-time systems. It is based on modifying the Java Virtual Machine (JVM), and supports both general-purpose and real-time applications, but only for single-processor systems.
- The Remote Method Invocation (RMI) [RMI04] defines a DOM model based on Java objects by defining a new interface in which the methods of remote objects can be invoked from other JVMs, possibly on different hosts.

Even though Java is becoming one of the most popular programming languages, it has not released any official DRTSJ specification or draft yet. The working group website [DRTSJ00] only outlines the important features of a future specification, so this approach will not be considered as an objective of this thesis.

1.4 DEVELOPMENT TOOLS AND STRATEGIES FOR REAL-TIME SYSTEMS

In general, any software development process aims to build applications able to meet certain functional requirements. However, the complexity of current systems has meant that this process should be structured in some way, leading to a sequence of steps which represent different levels of abstraction. For example, the development process defined in [BUR09] is divided into (1) requirement specification, (2) system design, (3) implementation and (4) testing.

However, although the methodology used in the development of real-time systems does not differ from other applications, it requires additional techniques for specifying, analysing and verifying the timing requirements. The evolving complexity of such systems has lead to the need for using more sophisticated development processes. Initially, structured programming strategies were used (e.g. DARTS [GOM84]) although the success of object-oriented programming led to the shift to object-based strategies (for example, HRT-HOOD [BUR94] or [MED05]). Likewise, new strategies have recently been proposed such as component-based [LOPZ10] [PLA08] or model-driven (e.g. those developed within the ASSERT project [MAZ09] [PERR10]) development processes, which are mainly focused on achieving a sustainable development process in terms of product costs, development times and quality.

Unlike these approaches based on programming languages, objects and components, the model-driven methodology aims to raise the level of abstraction for software development by setting the *model* concept as the basic entity for each stage of the development process (from requirement specification to design, analysis, configuration and implementation). Under this approach a software system is built from a set of high-level models, which undergoes a series of transformations that finally results in the executable code. Moreover, these transformation operations generate derived models to which different types of analysis can be applied (e.g. schedulability analysis).

One of the main supporters of these methodologies is the OMG, which has developed a set of standards called Model-Driven Architecture (MDA). MDA consists of a hierarchy of modelling levels:

- *Platform-Independent Models* (PIM) are used to specify a system independently of the computer platform on top of which it will run.
- *Platform-Specific Models* (PSM) are derived for specific execution platforms by transforming the PIM taking into account the particular characteristics of the chosen platform, abstracted in the form of a platform model (PM). The implementation code is automatically generated from the PSM, ideally with no human intervention.

The integration of middleware into an MDA strategy can enable the automatic configuration of certain critical parameters for real-time applications [MAH04] (e.g. concurrency, resource allocation or scheduling parameters). However, such integration should not be performed directly but requires middleware to support the same real-time model used within the MDA strategy. This facilitate not only the development but also the analysis of predictable applications at different stages of the development process (from the initial prototypes to the release application), as well as integration with other MDA tools.

1.4.1 ANALYSIS AND VERIFICATION OF REAL-TIME REQUIREMENTS

For real-time systems, it is not sufficient for the software to be logically correct; the applications must also satisfy particular timing constraints. Therefore, the verification of such requirements distinguishes the development of real-time systems from other kinds of applications. The verification of a real-time system can be usefully divided into a two-stage process [BUR09]:

- Verifying requirements / designs. This stage checks whether system timing requirements are fully coherent and consistent. It may require the use of formal methods for real-time systems, such as Real-Time Logic [JAH86].
- Verifying the implementation. This stage aims to verify whether system timing requirements can be satisfied in a specific execution platform (that

is, with a finite set of resources). RMA and holistic schedulability analysis are notable examples of this kind of techniques.

While there are research works that integrate formal methods into both stages [CHO95][MOK96][HEN06], this field is still open to research. Therefore, although the use of formal methods can be considered relevant in the development of real-time systems, the remainder of this section will focus on the second issue (implementation verification).

One of the major challenges in the development process of real-time systems is to accurately predict the worst-case temporal behaviour of the application. The complexity and heterogeneity of today's distributed systems has recently lead to the need for using techniques capable of obtaining reliable and accurate predictions which are also computationally feasible. Among the most common approaches used by these techniques are the solutions based on simulations, stochastic methods or analytic techniques.

In general, simulation often suffers from insufficient application coverage which may lead to an underestimation of the WCRT [XU93]. While this situation might be tolerable for some soft real-time systems, it is not for hard real-time systems. Representative examples of simulation-based tools are SIM-MAST [LOPZ04] or Extend [KRAH01].

Similarly, stochastic methods propose statistical analysis to determine the temporal behaviour of the system and they usually apply to soft real-time systems as well [VILA08-B]. However, these methods are attracting a high degree of interest within the real-time community and new analysis tools are being developed such as Stochan [LOPE08] or MOTOR [BOHN07].

In relation to analytic techniques, they have been traditionally applied in the analysis of hard real-time distributed systems. So far, this thesis has focused on schedulability analysis through the RMA classical theory and its extensions for distributed systems. However, there are other analytic techniques and tools for distributed systems, such as Modular Performance Analysis with Real-Time Calculus [THI00], based on queuing theory, or SymTA/S [HAM04], based on the compositional scheduling analysis of the system. The work included in [PERA07] reviews and evaluates different techniques and temporal analysis tools for distributed systems, and concludes that *one size does not fit all* (that is, the use of either technique depends mainly on the type of real-time application to be analysed).

For RMA, there are many software applications capable of modelling and analysing real-time systems, both commercial (e.g. TimeWiz [TIM02] or Rapid RMA [RAP03]) and open-source (e.g. Cheddar [SIN04] and MAST [HAR01]). MAST, as was discussed above, provides a set of tools to perform various types of timing analysis: schedulability analysis, sensitivity analysis or optimized scheduling parameters assignment techniques. As this software is open-source and follows the end-to-end flow model, it will become an useful tool within the context of this thesis.

A prerequisite for MAST and any schedulability analysis tool is knowledge about the WCET of each operation defined in the application. Estimating such values is still a major challenge, as they depend on the complexity of the system in terms of both software and hardware. However, currently there are some commercial tools such as AiT[1], which uses an abstract model of the processor on which the code is executed, or RapiTime[2], which uses timing measurements on the actual hardware together with coverage analysis to provide accurate estimates of WCET. Finally, RapidET [LU11] is another tool for calculating WCET based on stochastic analysis.

1.5 MOTIVATION AND OBJECTIVES

As we said earlier, the continuous increase in the complexity of modern real-time systems has lead to the need for new development processes that minimize development times and costs while still maintaining satisfactory levels of quality. The use of model-driven approaches is being accepted for the development of mission-critical and real-time systems, as they allow timing analysis techniques to be applied at different stages of the development. This type of development process usually relies on the automatic generation of the source code from a set of high-level models, which implies that the use of middleware adapted to the real-time model being applied could facilitate the development process, as well as the integration of different CASE tools. This becomes particularly important in high-integrity real-time systems in which new restrictions and requirements apply, such as the use of schedulability analysis techniques or the compliance with certain properties amenable to certification.

In the design of distributed real-time systems, one of the most critical stages is schedulability analysis; this stage is responsible for calculating the WCRT. Timing analysis for distributed systems usually tends to be performed independently for processors and communication networks, that is, both resources are scheduled separately. However, this strategy could be improved when precedence relationships are present in the real-time application. The end-to-end flow model, which has recently been standardized in the MARTE specification, is suited to this type of systems and allows distributed systems to be modelled in a simple and complete way. After modelling the system, schedulability analysis techniques can be applied to verify whether end-to-end timing requirements are satisfied.

1. AiT is available at http://www.absint.com/ait
2. RapiTime is available at http://www.rapitasystems.com

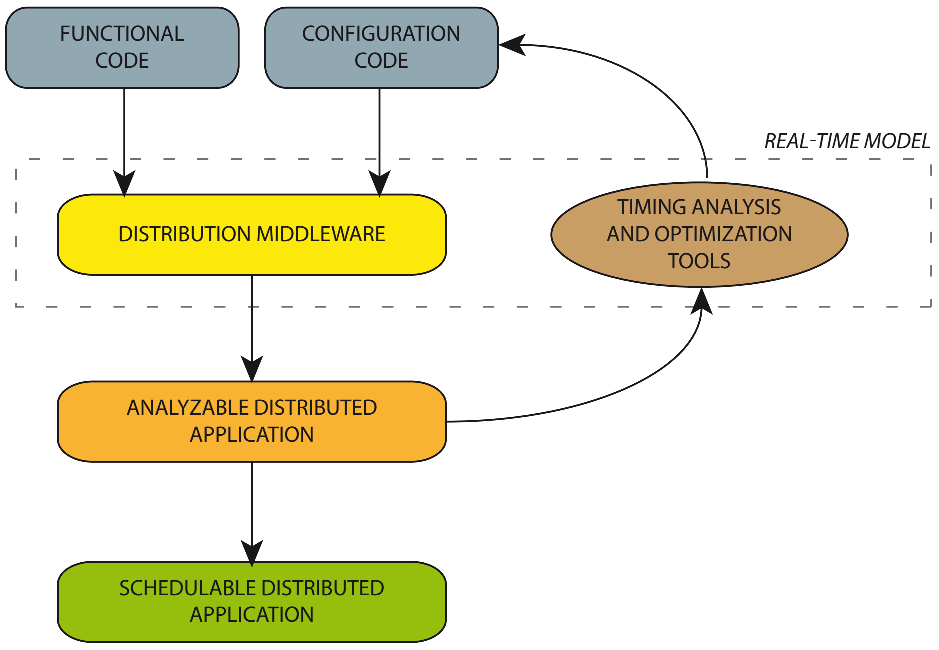
FUNCTIONAL CODE
CONFIGURATION CODE
REAL-TIME MODEL
DISTRIBUTION MIDDLEWARE
TIMING ANALYSIS AND OPTIMIZATION TOOLS
ANALYZABLE DISTRIBUTED APPLICATION
SCHEDULABLE DISTRIBUTED APPLICATION

Figure 1-9: Distribution middleware and analyzable distributed applications

Furthermore, distribution middleware usually provides little or no support to bound end-to-end response times in distributed systems. This lack of support could be overcome by integrating the real-time *end-to-end flow* model into middleware to facilitate the modelling, configuration and scheduling of distributed systems, as is illustrated in Figure 1-9.

Although Ada defines a coherent real-time model for single-processor systems, this standard provides limited support for the development of real-time distributed systems. Actually, there is no integration between the Distributed Systems Annex and the Real-Time Annex. Current research in this field has not been sufficiently widely accepted to merit standardization [BUR09]. This thesis also aims to fill this gap by proposing the development of distributed real-time applications designed to be analyzable via traditional schedulability analysis techniques for distributed systems.

1.5.1 OBJECTIVES

This thesis aims to make contributions in the field of distributed real-time systems by adding a real-time model suitable for schedulability analysis to distribution middleware. This objective has two fundamental premises:

- **The use of distribution standards**. Standards play a central role in the current development of complex real-time systems, and help to make products accessible and understandable for the community and the industry. Moreover, the implementation and testing of our approach would also benefit from using *open-source* software.

- **The application of the end-to-end flow model.** This model has been traditionally used for calculating response times in distributed real-time systems, and it has recently been included as a part of the MARTE modelling standard.

In particular, the main objective of this thesis lies in the integration of the *end-to-end flow* model into distribution middleware. This integration would facilitate schedulability analysis of real-time applications which have been distributed through middleware, as well as its incorporation into model-driven development processes. Within this development, the following specific objectives should be considered:

Objective #1: Analysis of the main distribution standards oriented to real-time systems.

Currently, there is a wide set of standards that support the development of distributed real-time applications. However, most of these standards provide limited support for the configuration of the temporal aspects of the application, as well as the verification of compliance with the timing requirements at runtime. Therefore, the first objective is to analyse the mechanisms proposed by each standard, as well as whether the available facilities are sufficient to guarantee determinism over the whole application.

Objective #2: Identification of a set of middleware features required for the development of analyzable applications.

Given the wide variety of standards, the second objective will be to identify a set of features and deployment options that would be desirable in any distribution middleware. The implementation of these features will enable the application of schedulability analysis techniques regardless of the distribution model and / or standard used.

Objective #3: Integration of the real-time *end-to-end flow* model into distribution middleware.

To this end, the proposed mechanisms should be flexible enough to allow the use of different scheduling policies and concurrency patterns. Furthermore, they must also be easily integrated with different distribution paradigms, such as those proposed by RT-CORBA or Ada DSA.

Objective #4: Adaptation of the *end-to-end flow* model to the Ada standard.

Ada was initially designed to support the development of real-time applications for single-processor systems and it has become an important programming language for safety-critical systems. Furthermore, this language also addresses distributed systems through one of its annexes, the DSA, although the standard does not provide details about the issue of distributed real-time programming. In view of these facts, this thesis proposes the use of Ada and its facilities as the base technology on which to develop our proposal, due to both its relevance within the real-time industry and its need to address the development of distributed real-time systems.

Objective #5: Development of a distributed real-time platform.

This platform will allow the real-time model and the proposed architecture to be validated. To this end, the platform should integrate several components such as middleware, operating system or communication networks while preserving system predictability.

Objective #6: Applying the *end-to-end flow* model to high-integrity systems.

High-integrity systems are characterized by imposing severe restrictions on the design and software development, as well as their need to undergo complex certification processes. The validation of the *end-to-end flow* model for this kind of systems aims to explore the flexibility and generality of our approach.

1.6 OUTLINE OF THE THESIS

The rest of this thesis is structured in five chapters, which correspond almost directly with the objectives listed above, as follows:

- **Chapter #2** analyses the distribution standards from the real-time systems perspective, with particular emphasis on both the mechanisms included in the specifications and the issues that remain open to implementations. This chapter also describes the features that should be supported by distribution middleware to allow the development of predictable applications.
- **Chapter #3** introduces the distributed real-time model, which is the key issue of this thesis. The elements that compose the model, its configuration

and operation are described throughout this chapter. Additionally, a specific implementation for Ada DSA is proposed.

- The main objective of **Chapter #4** is to validate the use of the real-time model in several scenarios and to apply different distribution models. This chapter includes the development of the distributed real-time platform for testing and its integration with the real-time model.
- **Chapter #5** focuses on adapting the model to high-integrity systems. It describes the most common restrictions in these systems and discusses how to apply the model according to these guidelines. Additionally, a specific implementation is proposed for middleware, useful for high-integrity applications developed in Ada.
- **Chapter #6** discusses the results of the thesis and explains the main conclusions. It also proposes the lines for future work.

Finally, the bibliography and two appendixes are included. Appendix "A" briefly introduces the main objectives and results of the national and international projects with implications for this research work, as well as its specific contributions to them. Appendix "B" lists the bindings developed to provide an Ada interface to a contract-based scheduling framework.

2 ANALYSIS OF THE REAL-TIME MECHANISMS INCLUDED IN THE DISTRIBUTION STANDARDS AND THEIR IMPLEMENTATIONS

This chapter is focused on the analysis of the main distribution standards from the real-time systems perspective. First, in Section 2.1, the general context is introduced. Then the following three sections present the real-time capabilities of the distribution middleware based on RT-CORBA, Ada DSA and DDS standards. Section 2.5 analyses real-time networks and their relationship with distribution middleware. Section 2.6 discusses whether the real-time mechanisms, included in distribution standards, and their implementations are enough to ensure application predictability, and reviews the desirable features and properties for this type of middleware. Finally, Section 2.7 summarizes the contributions of the chapter.

2.1 INTRODUCTION

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 schedulability analysis 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.

Although there are some previous works which deal with a performance analysis of distribution middleware based on jitter, latency or throughput features [SCH01] [XIO03], none of them address it from the schedulability analysis perspective. This chapter focuses on this topic, and describes the key features of distribution standards in terms of the management of resources, their proposed scheduling models, their adaptation to the current schedulability analysis techniques and their link to real-time communication networks. In particular, this analysis reviews the distribution mechanisms proposed by RT-CORBA, Ada's DSA and DDS, as they provide outstanding and standardized solutions for the development of distributed real-time systems. Then, the analysis is completed by reviewing some of their reference implementations as they can strongly influence the real-time behaviour of applications. 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.

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 [DRTSJ00]. 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. The key elements to emphasize are:

- Coherent support for end-to-end requirements in distributed applications.

  Support for these kinds of requirements (e.g. not only temporal constraints but also other ones such as fault management or security properties) must be included in middleware. 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 end-to-end flow described in the MARTE specification.

- An easily extensible scheduling and integrity framework must be provided.

  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.

- The use of a real-time network protocol must not be obligatory by default.

  This would allow the interoperability between general-purpose and real-time systems.

Although Java defines a real-time model for single-processor systems, its applicability to distributed systems still represents an open research field [BAS07][TEJ07]. Therefore, and since the DRTSJ specification is not complete yet and there are aspects that still have not been addressed, a more thorough analysis of it will not be made, and hereinafter we will focus on the study of the RT-CORBA, DDS and Ada DSA standards.

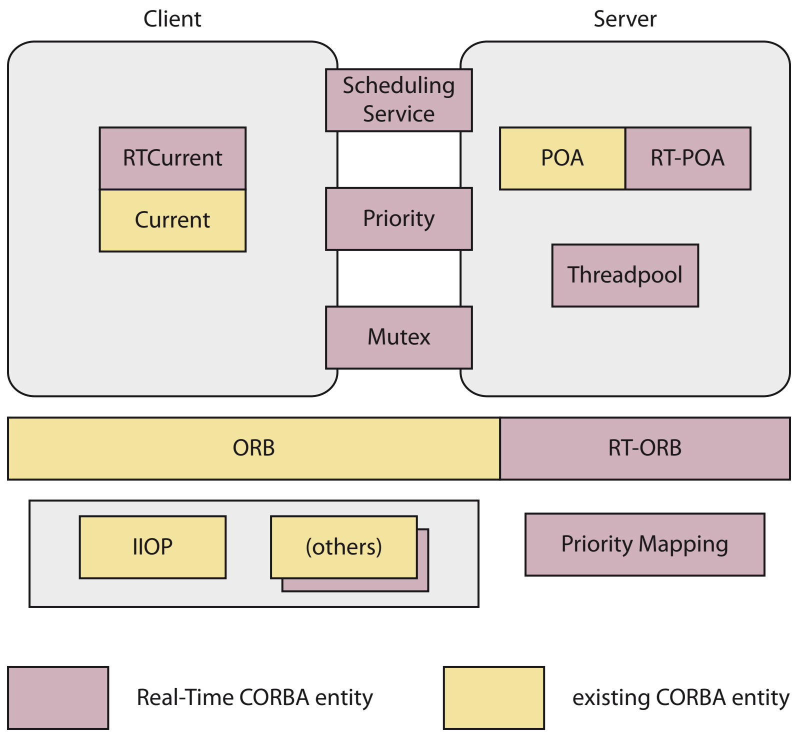

Figure 2-1: RT-CORBA extensions

2.2 RT-CORBA

The extension of the CORBA specification for real-time systems, which is called RT-CORBA [RTC05], 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 which are suitable for *a priori* timing analysis to determine the schedulability of the system (*static systems*), while the second focuses on systems with variable workload and whose schedulability is guaranteed at runtime (*dynamic systems*).

Figure 2-1, which is taken from [RTC05], illustrates the RT-CORBA architectural overview. This figure shows how key entities defined by the RT-CORBA extensions relate to the standard CORBA architecture. They are described below:

- *RT-ORB*, which is 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 tasks.

- *RT-POA*, which represents an extension to the POA [COR03] 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*, which 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 - 32767). This mapping is not standardized.
- *Mutex*, which is 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*, which is an interface to determine the priority of the current invocation (i.e. it enables the priority of application tasks to be handled).
- *ThreadPool* as a mechanism to control the degree of concurrency during the execution of remote calls on the server-side.
- *Scheduling Service*, 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, while middleware will be responsible for setting up the required resources to meet them.

By using these RT-CORBA entities, applications are able to configure and control the system resources explicitly as is described below.

Managing processor resources

According to the static scheduling Chapter of this specification, the main features of the RT-CORBA architecture are:

- **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 POSIX real-time standard [POS98].
- **Use of tasks as schedulable entities,** by which an RT-CORBA priority can be applied and by which 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.
- A *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 tasks or threads. The configuration of this entity enables the specification of the number of tasks that must be preallocated, the number of tasks that may be created dynamically, and their default priority. It also allows groups of tasks 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:

• **The use of different scheduling policies.** The possibility of introducing other scheduling policies in addition to the fixed-priority policy, such as, EDF (Earliest Deadline First), LLF (Least Laxity First), and MAU (Maximize Accrued Utility). 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.

• **The use of Distributable Threads as 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 [RTC05]. As in the case of DRTSJ, the concept of *distributable thread* is similar to the end-to-end flow used by response time analysis techniques [TIN94C] [PAL99].

Managing network resources

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

- **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 [SCH05], 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 get a dedicated connection (that is, non-multiplexed) per remote object.
- **Definition of Priority-Banded Connections.** This mechanism allows multiples 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.

2.2.1 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[1], e*ORB[2] and VisiBroker[3], and open-source distributions such as ROFES [LAN02] [LAN03], TAO [SCH98-2] or PolyORB [VER04]. PolyORB is characterized by

1. ORBExpress s available at http://www.ois.com
2. e*ORB s available at http://www.prismtech.com
3. VisiBroker s available at http://www.borland.com

supporting different distribution models, including among others the aforementioned RT-CORBA or Ada's 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 implementation of RT-CORBA and one of the most complete and efficient versions currently available for real-time systems [SCH01], so the analysis will focus on this implementation in the following.

TAO implements all the mandatory features and services that have been defined in the latest version of RT-CORBA [SCH05] with the following exceptions:

- The priority transforms model
- The use of *buffers* to store remote requests in threadpools
- The borrowing of tasks among threadpool lanes

In relation to the management of remote calls, TAO defines several configurable properties depending on whether the application is acting as a server or a client.

Concurrency patterns for server nodes

These parameters establish concurrency constraints that are imposed by the server node during the processing of requests in a multitasking environment. TAO defines two levels of concurrency which are closely related:

- **Concurrency at application-level.** These policies control which task 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 task-safe access to the object (i.e. safe execution by multiple tasks 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 tasks 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 task is dedicated to handling multiple connections. In addition, other tasks may also exist in the system to execute internal middleware operations.

- *Thread-per-connection*. In this case, the ORB creates a new task to serve each new connection. This task is dedicated to processing all requests performed on that connection, which will be processed sequentially. After closing the connection, the task will be released.
- *Threadpool*. Under this policy, middleware creates a pool of threads which are responsible for processing concurrent incoming requests according to a concurrency pattern called *Leader & Followers* [SCH98] [PYA01]. In the leader-followers pattern, several threads take turns to monitor input / output (I/O) operations and then process the requests once they have arrived. One task 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). Other tasks in the pool are the followers. As soon as the leader task receives a new request, one of the follower tasks becomes the new leader. Once the task finishes processing the request, it returns to the pool and waits to become the leader again.

Concurrency patterns for client nodes

These parameters affect the multitasking behaviour of the client when a synchronous remote call is performed, that is, when a client task must wait for a reply from the server. TAO defines a number of concurrency policies for waiting replies in client nodes, which are described below:

- ***Wait-on-read.*** According to this policy, when a client task 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 task 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 task will be notified in order to process it.
- ***Wait-on-leader-follower***. This policy enables client tasks to wait for replies using the *Leader & Followers* concurrency pattern. Therefore, client tasks waiting for replies becomes *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 task 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 tasks (Most Important First, MIF) [SCH05]. The latter policy is not included in the RT-CORBA standard and defines a parameter called *importance* to determine which task should execute.

Although RT-CORBA does not consider the assignment of scheduling parameters to the communications networks, TAO provides a solution 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* [RFC2474]. In TAO, protocol properties can be set at the ORB, task 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 task 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 which 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 all the 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.

2.3 THE ADA DISTRIBUTED SYSTEMS ANNEX (DSA)

The Distributed Systems Annex of Ada only deals with those mechanisms concerning distribution, such as the configuration of the partitions of a program, the distribution models supported or how to perform the communication between partitions. However, the DSA delegates the concurrent and real-time features to other parts of the language, such as those defined in the Real-Time Systems Annex (i.e. Annex D). Thereafter, since the use of the 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, which are named *Ada 95* and *Ada 2005*, 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:

- *Tasks*, which 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 deadline.

- *Protected objects*, which provide a task-safe and deterministic access to shared data.
- *Timing facilities*, such as different kinds of clocks and timers to measure real time and execution time of a single task or a group of tasks, and 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 which can be simultaneously applied on the same system.

As in the case of Java, Ada defines a coherent real-time model for single-processor and multiprocessor systems, but 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 [GUT99] [GUT01]. The key aspects of the language for the management of processor and communication network resources are described briefly below.

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*, a preemptive scheduling policy based on fixed priorities which uses first-in-first-out (FIFO) order for the same priority level.
    - *Non_Preemptive_FIFO_Within_Priorities*, a non-preemptive scheduling policy based on fixed priorities which uses FIFO order for the same priority level.
    - *Round_Robin_Within_Priorities*, a preemptive scheduling policy in which tasks are time-sliced for each priority level.
    - *EDF_Across_Priorities* is a preemptive scheduling policy based on dynamic priorities which 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 was previously mentioned, the communication among active partitions is carried out in a standard way

using the Partition Communication Subsystem (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 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 or HL for both the fixed priorities [SHA90] and EDF [BAK91] 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*).

Managing network resources

Like RT-CORBA, Ada DSA does not consider the possibility of passing scheduling parameters to the communications networks, although there are some research works that have incorporated this concept [GUT99][GUT01].

Furthermore, Ada DSA does not have any mechanism for the transmission of priorities, as this aspect is open to implementation. [PAU00] proposes a mechanism to handle the transmission of priorities following the same scheme defined by RT-CORBA. Moreover, in [LOP04] and [LOP06] some mechanisms for handling the transmission of priorities within the DSA are proposed. These mechanisms are in principle more powerful than that of RT-CORBA, as they allows total freedom in the assignment of priorities both in the processors and in the communication networks.

Finally, although Ada provides powerful and flexible mechanisms, the development of real-time systems is still considered a challenging, complex and costly process (for example, due to the certification process in high-integrity systems). Over the last decade, one of the major challenges for the real-time community has been to demonstrate that concurrent programming is a useful, safe and applicable facility even in safety-critical systems. To this end, the latest revision of the language introduced the *Ravenscar profile* [ADA05], a subset of the Ada tasking and real-time features designed to facilitate the development of complex safety-critical hard real-time applications while simplifying schedulability analysis. This profile has become a useful tool for developing real-time single-processor systems, although its applicability to distributed systems still remains open to research. This issue is the topic of Chapter 5, which will concentrate on how to adapt the real-time end-to-end flow model to high-integrity distributed real-time systems.

2.3.1 DSA IMPLEMENTATIONS

Although distributed programming with the DSA is easier and more intuitive than with other technologies [KER99], the commercial impact of this annex has not been very significant and only a couple of implementations are relevant today: Glade and PolyORB.

Glade [PAU00] is the original implementation of the DSA offered by AdaCore[1] to support the development of distributed applications with real-time requirements. The scheduling is done through fixed priorities and it implements two policies for distribution of priorities in the style of RT-CORBA (*Client Propagated* and *Server Declared*). This implementation does not consider scheduling for communications networks.

For the management of remote calls, Glade defines a taskpool to process the requests. The number of tasks in the pool can be configured through three parameters: *minimum size*, which indicates the number of preallocated tasks; *high size*, which represents a ceiling in the number of available tasks to process requests (i.e. tasks are deallocated if the number of tasks is greater than the ceiling); and *maximum size*, which represents the absolute maximum number of tasks in the pool, which therefore indicates the number of requests that can be concurrently processed (i.e. not queued). Glade also uses another intermediate task to await the arrival of requests, perform an initial processing and select one of the tasks of the pool to finally execute the remote call.

Glade maintenance has been discontinued today, and its functionality has been replaced by other middleware developed by the same company, called PolyORB [VER04]. PolyORB is introduced as a 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 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 (called *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

1. Ada-Core Technologies, The GNAT Pro Company. http://www.gnat.com/

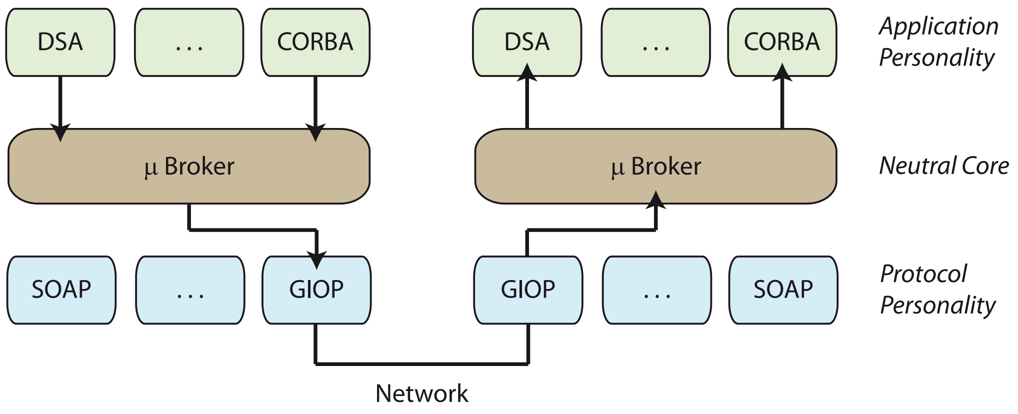

Figure 2-2: Application and protocol personalities in PolyORB

enables interoperability and integration of different distribution paradigms under a single platform. Figure 2-2 illustrates this behaviour. Let's assume that the application invokes a remote call to perform an arithmetic operation. In this example, both the invoking and the invoked application can be distributed according to the DSA and CORBA, but the remote call is handled by the same middleware in any case. Furthermore, this architecture allows different communication protocols to be used regardless of the application personality. Thus, in the example shown in Figure 2-2, the communication between DSA partitions can be done through the GIOP protocol, which is defined in the CORBA specification. The key feature of this interoperability relies 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*. Not only does this *microkernel* provide the same services that a conventional ORB does, but it also includes facilities for performing the conversion between distribution models. The decoupling of application and protocol personalities, and the support for multiple simultaneous personalities within the same running middleware, has led to it being presented as a *schizophrenic* middleware.

For the management of remote calls, PolyORB supports different configurations to adapt the interaction between personalities and the *microkernel*. According to Figure 2-3, such 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 below.

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 which can take the following values:

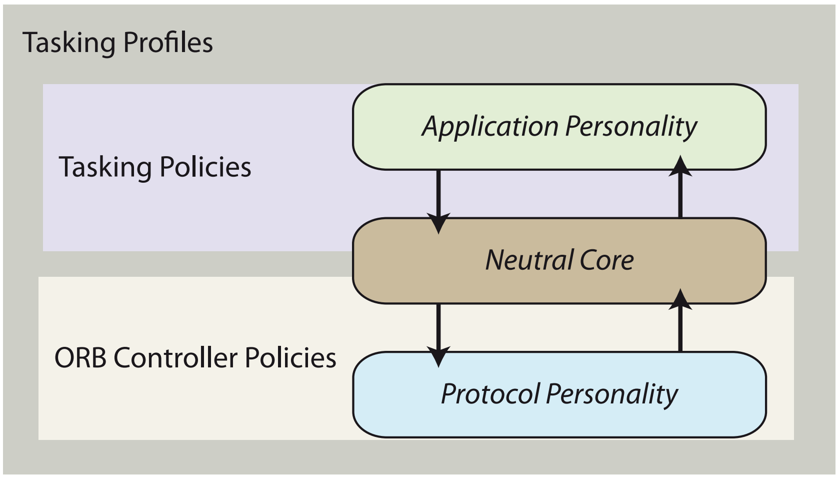

Figure 2-3: Tasking model in PolyORB

- *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 which are compliant with the Ravenscar profile [ADA05].

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 available tasks 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 them 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 behaviour 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* [SCH98]. Under this policy, all the threads are equal and they monitor the I/O operations and process the incoming requests alternatively.
- *Half Sync/Half Async* [SCH96][PYA01]. 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* [SCH98][PYA01]. 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 support for real-time distributed systems, follows the same static scheduling scheme as RT-CORBA which was already implemented in Glade: *Client Propagated* and *Server Declared*.

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

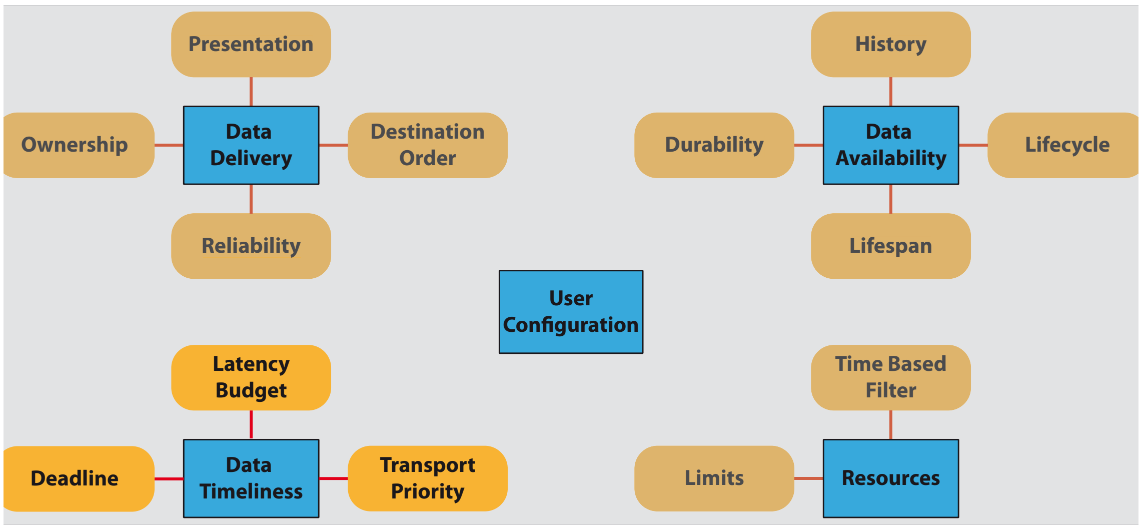

Figure 2-4: Qos parameters defined by DDS

2.4 THE DATA DISTRIBUTION SERVICE FOR REAL-TIME SYSTEMS (DDS)

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 non-functional 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. 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 (see Figure 2-4):

- **Data availability**. It comprises those parameters for controlling queuing policies and data storage. The parameters that fall into this category are *Durability*, *Lifespan* and *History.*
- **Data delivery.** It 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.** It controls the latency in the distribution of data. The parameters that fall into this category are *Deadline*, *Latency_Budget* and *Transport_Priority.*

- **Maximum Resources.** It 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.

Managing processor resources

The DDS specification does not explicitly address the scheduling of tasks 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 behaviour and improving the predictability of the application. The three parameters of Data Timeliness, which are highlighted in Figure 2-4, 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 and 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 must not be enforced or controlled by middleware and, consequently, indicates the urgency in the processing of data samples. Therefore, it can be considered as a *best-effort* parameter to configure the internal behaviour of middleware.

These two QoS parameters, even if both share similar objectives, are applied at different levels as is illustrated in Figure 2-5. This figure shows how the

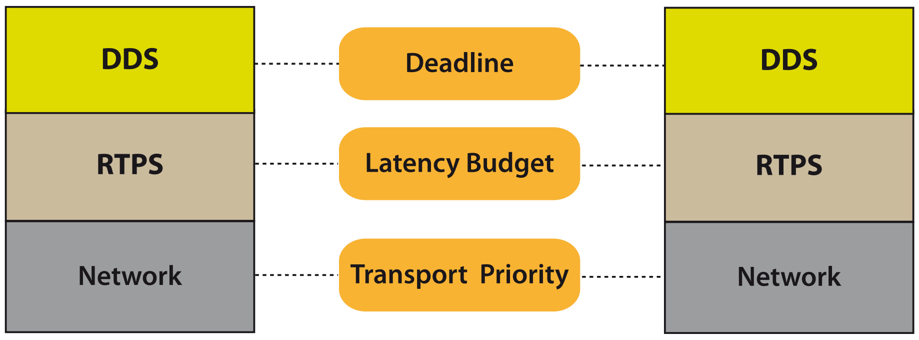

Figure 2-5: Timing control in DDS

Deadline parameter is monitored within the DDS layer, while the *Latency_Budget* is applied within the RTPS 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 Data Writer (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 task until the data sample has been delivered and acknowledged by the matched Data Readers (DR).

On the subscriber side, the reception of data can be performed in both synchronous 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. non-fulfilment of requested QoS). In particular, the application could be notified through:

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

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 2-4):

- **Transport_Priority**. Unlike the *Latency_Budget* which attempts to optimise the internal behaviour of middleware, this parameter prioritizes the access to the communication network (see Figure 2-5). Furthermore, since communications are unidirectional, it is only associated with DW entities.

Moreover, the DDS Interoperability Wire Protocol 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 usage 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 sub-messages. By processing each sub-message independently, the system can discard unknown or erroneous sub-messages and thus ease 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 which consumes both processor and network resources. In particular, DDS provides a service for the automatic management of entities called *Discovery*. This service describes how to obtain information about the presence and characteristics of any other entity within the distributed system. As a result, 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 (such as a *plug-and-play* system). This kind of network traffic, which is internal to middleware, is called *metatraffic* and should be considered in the schedulability analysis.

2.4.1 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[1] or RTI-DDS[2]) and open-source software (OpenSplice[3] or OpenDDS[4]). For our purposes, we have selected RTI-DDS middleware because it is a reference implementation and is considered one of the most efficient implementations of the standard [XIO03].

RTI, one of the driving forces behind the DDS standard, develops and markets a software product called “RTI Data Distribution Service” (RTI-DDS). As would be expected from the analysis performed in this section, the DDS standard does not address any aspect of the concurrency model of distributed applications, nor the scheduling of tasks either. In this case, the RTI-DDS tasking model uses three types of internal middleware tasks whose priorities are assigned through a proprietary extension of the QoS parameters and cannot be changed dynamically:

- **Database**: This task is responsible for storing updated information about locally-created and remotely-discovered entities.
- **Event**: This task is dedicated to checking the condition of many different time-triggered events.
- **Receive**: One or more tasks dedicated to processing the I/O events received via the underlying network transport. The static assignment of priorities to *Receive tasks* can only be applied at Participant level, that is, all the receiving tasks belonging to the same Participant will share the same priority (both for *metatraffic* and user data).

The RTI-DDS implementation has created a concept called *Exclusive Areas* (EA) to encapsulate mutexes and critical sections. RTI-DDS defines an ordering of the EAs and a set of accessing rules (e.g. they are always accessed in the same order), which prevents deadlocks even for synchronization protocols susceptible to this problem such as BIP [SHA90].

Finally, in the case of communications networks, the assignment of scheduling parameters is performed through the *Transport_Priority* QoS parameter. Under this implementation, this parameter allows different priorities to be allocated to the data sent on a per-DW basis. To this end, the priority is mapped to the *Diffserv* field [RFC2474] within the IP header in order to prioritize network traffic through capable network elements (e.g. routers or high-level switches).

1. CoreDX is available at http://www.twinoakscomputing.com/
2. RTI-DDS is available at http://www.rti.com/
3. OpenSplice is available at http://www.prismtech.com/
4. OpenDDS is available at http://www.ociweb.com/

2.5 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. While 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:

- *The 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. A wide range of techniques that solve the problem of message collision have already been introduced in Chapter 1.

However, in general, the distribution standards that have been analysed do not consider any of these problems and, therefore, they do not specify the required properties of the underlying communication subsystem which may affect the temporal behaviour of the distributed system. Thus, while the DSA does not address any characteristics of communications networks, the RT-CORBA and DDS specifications define two network protocols to facilitate interoperability between implementations, called GIOP and RTPS, 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 [LAN03]. 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, as was previously discussed in this chapter, provides a mechanism to map RT-CORBA priorities onto *Diffserv* data field [RFC2474]. 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 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 it is depended on each network device. Therefore, *Diffserv* cannot assure a priori that packet processing will be uniform over the network. To partially mitigate this issue, the IETF RFC 2474 standard [RFC2474] recommends certain values for this data field to ease interoperability between network devices (for example, 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 [FEL01] [ZHA01]. This has motivated the development of an extension to the standard called *Extensible Transport Framework* (ETF) [ETF04], a framework that allows 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 [FOS05]. The work in [LOS04] uses a preliminary version of ETF to implement a prototype that integrates the TTP / C communication protocol with RT-CORBA.

The RTPS protocol is designed to use a *multicast* 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 offers support to send / receive messages and detects errors during transmission (for example, incomplete or corrupted messages). Moreover, since the size of messages is not sent explicitly by the RTPS protocol, the underlying transport must provide a mechanism to deduce the size of the received message. This latter requirement can be problematic for protocols which 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). 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 in [REK03]. In the case of RTI-DDS, this implementation mainly uses an UDP/IP transport and, in a similar way to what is done in TAO, maps the *Transport_Priority* QoS parameter to the *Diffserv* data field [RFC2474].

Therefore, 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 [VBLAN06]) that allow the prioritization of network traffic.

As we said earlier on Section 1.2.3.2, shared Ethernet is unfeasible as a real-time network due to the non-deterministic resolution of collisions. 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 [PED03]. This effect, which is unacceptable for hard real-time systems, can be controlled by using flow control techniques for network traffic as is described in [VILA08] or in the recent IEEE 802.1Qbb specification [FLOWC11]. 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 the 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 [MACB04], 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 (for example, 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) [AFDX09] 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 1-2, 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 (for example, 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, predictable routing, etc).

2.6 ANALYSIS OF DISTRIBUTION MIDDLEWARE FROM THE REAL-TIME PERSPECTIVE

2.6.1 ANALYSIS OF THE REAL-TIME FEATURES OF DISTRIBUTION STANDARDS

After analysing the different distribution standards aimed at the development of applications with timing requirements, this section attempts to find the analogies and differences among these specifications, as well as to assess their appropriateness for use in real-time systems.

According to the model used in the analysis, two types of schedulable entities can be identified in a real-time system: tasks for processors and messages for communications networks. As was introduced in Chapter 1, there are situations in which processors and networks should be scheduled together, so middleware should provide sufficient mechanisms to configure both entities.

2.6.1.1 Managing processor resources

The temporal behaviour of distribution middleware is strongly determined by scheduling policies and concurrency patterns [PER08]. 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. The second case deals with the options available to establish which task is responsible for sending or receiving remote requests.

Firstly, 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 system tasks.

Both Ada and RT-CORBA specifications provide support for different scheduling policies, including the FPS policy. 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 of them are suitable to schedule tasks 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 [LOP06].

Secondly, the processing of remote calls represents a 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. In general, regardless of which task or tasks are responsible for processing each stage, it is important that middleware provides the necessary mechanisms to control their scheduling parameters. However, how to manage and process a remote call is implementation-defined. As has been discussed above, distribution standards do not define which concurrency pattern should be used for sending and / or receiving remote calls, but specify that implementations must service concurrent remote requests (for example, 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 behaviour of the application, so this issue will be addressed further in the analysis of the implementations.

Finally, deterministic access to shared resources prevents the unbounded priority inversion problem [SHA90]. Both Ada and RT-CORBA include the use of synchronization protocols for access to critical sections, although only Ada specifies that implementations need to support a predefined protocol (in this case, the HL or Priority Ceiling Protocol).

2.6.1.2 Managing network resources

In relation to communications networks, neither RT-CORBA nor Ada's DSA include the possibility of assigning scheduling parameters and therefore implementations are responsible for providing the necessary support for it. In the case of DDS, the specification only considers networks based on a fixed priority scheduling policy while it excludes any other kind of predictable networks used in the industry (e.g. time-triggered networks). It is more flexible to modify the definition of the *Transport_Priority* parameter to include a wider range of network scheduling policies (e.g. *Transport_Scheduling_Parameter*).

Although most of the standards analysed are focused on Ethernet-based networks (such as RT-CORBA with TCP/IP and DDS with UDP/IP), this communication network is not suitable to provide deterministic response times itself as it was discussed in Section 2.5. However, the evolution of IP technology in recent years, with the definition of new standards, such as 802.1p [VBLAN06] 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, [ALD06] [FRSH11], are used. Sending the contract parameters through the network is inefficient because these parameters are large in size.

Another important factor to consider is the size of network messages, which must be bounded and known before the schedulability analysis. This point is particularly critical in the design of predictable applications with DDS since an RTPS message can comprise an undefined number of sub-messages, 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 which aim at guaranteeing latencies limits in each network stream.

Finally, the presence of messages and operations belonging to middleware may cause an increase of 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.

2.6.1.3 Comparative summary

This section has focused on discussing the mechanisms provided by distribution standards for the management of processors and networks. Table 2-1 summarizes the main real-time features defined by these standards, which are grouped into the following categories:

- *Scheduling*, which includes policies responsible for ordering the concurrent access of tasks and messages to processors and communication networks, respectively.
- *Concurrency patterns* for the execution of remote requests, or those strategies defined for controlling and processing remote requests on the node called.
- *Controlled access to shared resources* through the implementation of synchronization protocols.
- *Setting of scheduling parameters* to remote calls, including those mechanisms defined for the transmission of scheduling parameters.
- Support for the *end-to-end flow model* or distributable thread.

Table 2-1: Real-time capabilities of distribution standards

MIDDLEWARE	SCHEDULING		CONCURRENCY PATTERN	SHARED RESOURCE ACCESS CONTROL	SETTING OF SCHEDULING PARAMETERS	END-TO-END FLOW MODEL
	PROCESSORS	NETWORKS				
RT-CORBA	FPS EDF LLF MAU	NOT DEFINED	THREADPOOL	REQUIRED	CLIENT PROPAGATED SERVER DECLARED	DISTRIBUTABLE THREAD
ADA DSA	FPS NON PREEMPTABLE ROUND-ROBIN EDF	NOT DEFINED	NOT DEFINED	PRIORITY CEILING PROTOCOL	NOT DEFINED	NOT DEFINED
DDS	NOT DEFINED	FPS	NOT DEFINED	NOT DEFINED	NOT DEFINED[A]	NOT DEFINED

A. NOT APPLICABLE FOR P/S SYSTEMS

Most of these features, which are required to perform the schedulability analysis, remain open to implementations as is shown in Table 2-1. 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 execution of remote calls is particularly relevant, although this feature is set by implementations and therefore it will be dealt in more detail with the next section.

Since the end-to-end flow model has been traditionally used in calculating the response times of distributed systems [TIN94B], its integration into distribution middleware would facilitate the development of real-time systems in several aspects. Firstly, it would allow the straightforward application of CASE tools, such as MAST [HAR01], for analysis and / or optimization. Secondly, given that the end-to-end flow model is defined within the MARTE standard [MAR08], it would facilitate the automatic generation of distribution and real-time source code when it is incorporated into model-driven development processes, such as the ASSERT development process [MAZ09] [PERR10].

Furthermore, the Ada language does not consider in any case the existence of the end-to-end flow model, but there are research works that show that it is possible to integrate it into the DSA, such as the proposals included in [LOP06] and this thesis, part of which is included in [PER10]. RT-CORBA defines a similar concept to end-to-end flow, the *Distributable Thread*, which allows end-to-end scheduling for distributed applications. However, the high complexity of this part of the specification means that only two implementations [SCH05] [LI04] provide the required support, although there are no references to practical applications that use it up to now. In the case of DDS, the standard does not consider the use of the end-to-end flow model.

Finally, although this may be considered a subjective comment, distribution middleware should target the ease of programming, for example by allowing the separation of concerns among the application logic, the distribution code and the real-time configuration. In this case, neither RT-CORBA nor DDS fulfil this requirement.

2.6.2 ANALYSIS OF THE REAL-TIME FEATURES OF IMPLEMENTATIONS

This analysis aims to assess the solutions provided by implementations about those features that standards leave as implementation-defined and which may affect the temporal behaviour of applications.

2.6.2.1 Managing processor resources

The support provided by each implementation included in the analysis in relation to task scheduling is quite diverse. The usual approach is to provide an interface to configure the scheduling parameters of application tasks and delegate their execution sequence to the scheduler provided by the underlying operating system. In this case, the PolyORB-CORBA personality and TAO provide a compliant interface with the RT-CORBA specification, whereas RTI-DDS 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 named MIF which is not defined by the CORBA standard. Under this scheduling policy, middleware is responsible for determining which task among all those available within the application should execute next.

In general, the processing of remote calls is a two-stage process that includes: firstly, waiting for requests arriving from the network and their initial processing; and secondly, the execution of the user code associated with the requested service. Regardless of which task or tasks are responsible for processing each stage, it is important that middleware can provide the necessary mechanisms to control their scheduling parameters.

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 tasks, such as *Thread-per-Connection* or *Thread-Per-Session*, should be restricted to those situations where the creation of new tasks does not jeopardize the determinism of the whole system (e.g. through an admission test at run time). Moreover, other critical scenarios should also be considered; for example, in flexible scheduling frameworks [ALD06] [FRSH11] where tasks execute under contracts, middleware implementations should select a concurrency pattern that minimizes the dynamic change of the scheduling parameters [PER09] 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 analysed implementations 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 (such as *ThreadpoolLanes*, *Private Connections* and *Priority-Banded Connections)*, makes available a set of tasks 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 RTI-DDS, the use of *Listeners* along with its implemented concurrency pattern allows tasks to process incoming requests without using other intermediate tasks. In general, the restrictions imposed by DDS implementations in the assignment of scheduling parameters may cause quite variable response times depending on the target application [PER12].

Finally, both TAO and RTI-DDS delegate the choice of the synchronization protocol to the underlying operating system. Nevertheless, the POSIX real-time standard [POS98] 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 and, 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.

2.6.2.2 Managing network resources

As in the case of the distribution standards, the implementations analysed mostly use general-purpose networks but incorporating some extensions to assign priorities in the communications networks. Thus, both TAO and RTI-DDS 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 communications networks.

2.6.2.3 Comparative summary

This section has focused on discussing the mechanisms provided by middleware implementations for the management of the processing resources (i.e. processors and communications networks). Although the main objective is to analyse the capabilities of distribution standards for developing distributed real-time systems, the

Table 2-2: Real-time capabilities of middleware implementations

MIDDLEWARE	SCHEDULING		CONCURRENCY PATTERN	SHARED RESOURCE ACCESS CONTROL	SETTING OF SCHEDULING PARAMETERS	END-TO-END FLOW MODEL
	PROCESSORS	NETWORKS				
TAO	FPS MIF	FPS OVER IP	REACTIVE THREAD PER CONNECTION THREADPOOL	RTOS DEPENDENT	CLIENT PROPAGATED SERVER DECLARED	DISTRIBUTABLE THREAD[a]
POLYORB	FPS NON PREEMPTABLE ROUND-BOBIN EDF	NOT DEFINED	NO TASKING THREAD PER REQUEST THREAD PER SESSION THREADPOOL[b]	PRIORITY CEILING PROTOCOL	CLIENT PROPAGATED SERVER DECLARED	NOT DEFINED
RTI-DDS	FPS	FPS OVER IP	THREADPOOL	RTOS DEPENDENT	NOT DEFINED	NOT DEFINED

a. Not tested

b. Several control patterns can be applied: Workers, Half Sync/Half Async or Leader & Followers

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 tasks and messages scheduling, as well as the concurrency patterns implemented for processing concurrent remote calls. Table 2-2 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), a proprietary API (e.g. RTI-DDS) or through the programming language (e.g. Ada for PolyORB-DSA). In relation to communications, most implementations use network scheduling based on fixed priorities over Ethernet technology, although these networks do not meet hard real-time requirements yet [VILA08] [PED03] except under very specific conditions [AFDX09].

The design and development of efficient concurrency patterns for managing concurrent invocations is a key factor in the temporal behaviour of implementations. The concurrency patterns implemented in TAO and PolyORB can be used as a reference for 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). Although the concurrency pattern implemented in RTI-DDS would also be suitable for this latter scenario, the lack of flexibility in the assignment of scheduling parameters may penalize the temporal behaviour of certain types of applications.

Finally, most implementations delegate the use of synchronization protocols to the operating system. This is worthy of consideration because even the POSIX standard

[POS98], 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, PolyORB can be used as a reference as it allows the configuration of synchronization protocols through the mechanisms provided by the Ada language.

2.7 CONTRIBUTIONS OF THIS CHAPTER

This chapter has reported an analysis of distribution middleware options from the viewpoint of their suitability for the development of real-time systems, and has also discussed some solutions adopted according to the proposals of this thesis. Specifically, the study has analysed the RT-CORBA, the Ada DSA and the DDS standards, with particular emphasis on the scheduling of processors and networks. Based on the analysis above, we have isolated a set of features and objectives that all distribution standards for real-time systems and / or their implementations should incorporate:

- **Control of remote calls.** Regardless of the concurrency pattern used, the determinism of the application can only be guaranteed by controlling the scheduling parameters even for tasks 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 end-to-end flow 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 [LIU73] [FOH02]. 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 on 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 tasks 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.

- **End-to-end flows or distributable threads.** The integration of this concept into distribution middleware would facilitate the application of CASE tools for analysis and optimization, as well as its integration into model-driven development processes.
- **Enabling schedulability analysis of the complete application.** Although middleware is executed in the processor, the temporal behaviour 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 [LIU00] and, therefore, middleware should have the ability to specify the scheduling parameters of both processing resources through suitable models.

3 PROPOSAL FOR AN ANALYZABLE REAL-TIME MODEL IN DISTRIBUTION MIDDLEWARE

This chapter is organized as follows. First, in Section 3.1, the basic features that distribution middleware should support to develop real-time applications are introduced. The integration of the end-to-end flow concept into the distribution models analysed in Chapter 2 is addressed in Section 3.2. Section 3.3 discusses in detail the proposal for integrating the real-time model into distribution middleware. A detailed description about the interface designed to support the real-time model is dealt with in Section 3.4. The development of the proposed interface for Ada is included in Section 3.5, while Section 3.6 discusses the mechanisms required to enable the automatic generation of the real-time configuration code from high-level system models. An example of use for the proposed interface is included in Section 3.7. Section 3.8 discusses the use of concurrency patterns with the real-time model. Finally, Section 3.9 summarizes the contributions of the chapter.

3.1 INTRODUCTION

Chapter 2 reviewed the distribution standards usually applied for real-time systems, as well as the analysis of their real-time capabilities and the mechanisms currently supported. These standards follow different interaction paradigms (e.g., *client - server* or *publisher - subscriber*) which can be easily modelled through the end-to-end flow model. Once the system is represented as a set of end-to-end flows, timing analysis techniques can be directly applied to verify whether the distributed application meet its deadlines. Within the real-time community, the end-to-end flow model has traditionally been used in calculating the response times of a distributed system, and it has recently been included in the MARTE standard for modelling and analysing real-time systems [MAR08]. Therefore, the incorporation of this model into distribution middleware would facilitate the development process for distributed real-time systems, as it would provide the required support to apply timing analysis, optimization and automatic code generation tools. Furthermore, this would also allow its integration with model-driven development processes. In particular, it would be desirable that the usage of the end-to-end flow model along with distribution middleware could support the following characteristics:

- **Separation of concerns between the logic of the application and the real-time aspects.** This would require hiding the details related to scheduling from the software engineers developing the functional parts of the application. Moreover, the responsibility of programming the real-time aspects might even rely on another kind of developer specialized in real-time systems.
- **Support for heterogeneous scheduling policies and parameters.** Distribution middleware should provide mechanisms to allow the use of interchangeable scheduling policies. Likewise, it would also be desirable to be able to specify the scheduling details through a uniform methodology which should be independent of the scheduling policy or parameters used.
- **Control in the identification and / or assignment of scheduling parameters.** Distribution middleware should document the existence and nature of each schedulable entity within the system, as well as providing mechanisms flexible enough to configure them.
- **Infrastructure for CASE tools**. The configuration of the real-time aspects should be performed automatically by means of software tools that generate the required configuration code from high-level system models.

In order to incorporate these features into distribution middleware and facilitate the application of timing analysis techniques, a new configuration mechanism has been developed. It is called the *endpoints pattern* and it represents a set of mechanisms which allow the real-time requirements and capabilities of a distributed system to be specified according to the end-to-end flow model. Nevertheless, before we go into the details about the *endpoints pattern*, it is worth examining how to integrate the end-to-end flow model into distribution middleware and, in particular, into the distribution models which were discussed in Chapter 2.

3.2 DISTRIBUTION MIDDLEWARE AND THE END-TO-END FLOW MODEL

As we saw in the previous chapter, it would be desirable that distribution middleware can provide the required mechanisms to configure the schedulable entities. However, this configuration process is not trivial and requires the application of analysis and optimization techniques to satisfy its timing requirements. For example, although the developers can assign priorities based on their experience and according to certain application features, the problem becomes intractable as soon as the system is composed of dozens of tasks and more than three or four processors. In this case, the application of automated tools for timing analysis and feasible assignment of scheduling parameters is required as there are multiple and complex interactions between tasks in distributed systems. This precedence relationship between tasks means that the assignment of scheduling parameters to a single task (or message, in the case of networks) may influence the temporal behaviour of other system tasks. A reliable mechanism to determine whether the scheduling policy and the priority assignment applied can meet the required timing requirements consists of an *a priori* schedulability test, since simulation can in general underestimate the worst case response time of applications [XU93].

This section aims to explore the integration of the end-to-end flow concept into the distribution models proposed by the standards for developing distributed real-time systems. To continue the same line of research followed by the previous chapter, this section will focus on the following standards, which are classified according to their distribution paradigms: Ada DSA for remote procedure calls, CORBA for distribution of objects and DDS for data-centric distribution.

The distribution paradigm included in DSA defines two communication modes in the calling node: synchronous mode (i.e. RPCs), in which the calling task is blocked on the communication network until the remote call is completed and the reply received; and asynchronous mode (APCs), in which the task is allowed to return before the completion of the remote call. For CORBA, the available communication modes are similar: remote requests are performed through the ORB asynchronously (called *Oneway* requests in the CORBA standard) or synchronously. However, the decoupling features of the message-oriented distribution paradigm cause significant differences compared to the previous cases. For instance, DDS only provides a single mechanism to publish data, which is asynchronous and unidirectional (i.e. it allows one-to-one or one-to-many communications).

From the viewpoint of called nodes, neither DSA nor CORBA define any communication modes, and they only specify that the system must provide support to process requests concurrently. However, DDS does include multiple communication modes. Actually, Subscribers can receive data either asynchronously (i.e. through *Listener* entities), in which the internal middleware tasks are responsible for processing received data, or synchronously (i.e. through *Wait-set* entities), in which the internal middleware tasks are responsible for receiving data, but application tasks are in charge of processing them.

As we said earlier on Chapter 1, real-time systems can be modelled as a group of end-to-end flows. In a distributed real-time system, this model enables the calculation of the worst-case response time for each end-to-end flow that has been defined. The

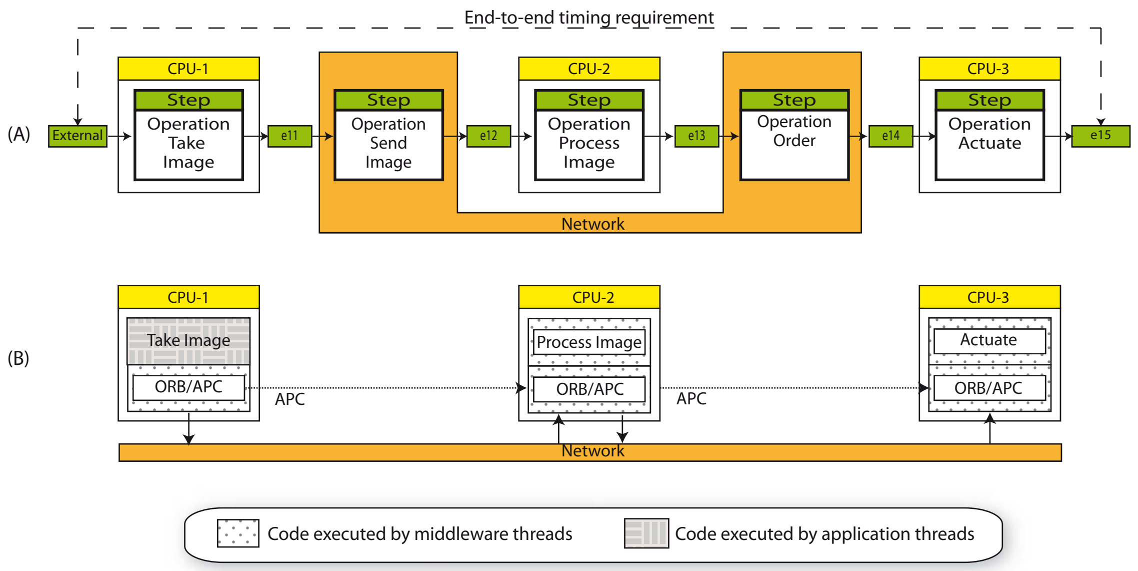

Figure 3-1: Asynchronous end-to-end flow model and distribution model based on ORB and APCs

optimization of these response times requires a thorough assignment of the scheduling parameters to each schedulable entity (tasks or messages), which are responsible for processing each step.

3.2.1 MODELLING OF ASYNCHRONOUS REMOTE CALLS

To illustrate the modelling of Asynchronous remote Procedure Calls (APCs), once again we consider the linear end-to-end flow example used in Chapter 1 and illustrated in Figure 1-3; in this example, CPU-1 has to take an image of the environment, send it to another processor to be analysed, and finally cause an action to occur in CPU-3. Figure 3-1 represents the end-to-end flow model for schedulability analysis (A) and the distribution model based on ORB/APCs (B).

The theoretical model shown in Figure 3-1-A defines a linear distributed end-to-end flow performing asynchronous remote calls whose timing requirement is an end-to-end deadline from the creation of the image until the actuation executed in CPU-3. The model shows all the events, steps and processing resources (the processors or the network) defined in the end-to-end flow. In particular, the end-to-end flow consists of five steps:(1) taking an image, (2) sending the image to be analysed through the network, (3) analysing the image to determine the actuation, (4) sending the command for actuation through the network and (5) performing an action depending on the command received.

The implementation of this example using the model based on ORB/APCs (Figure 3-1-B), requires the definition of *Process Image* and *Actuate* as remote

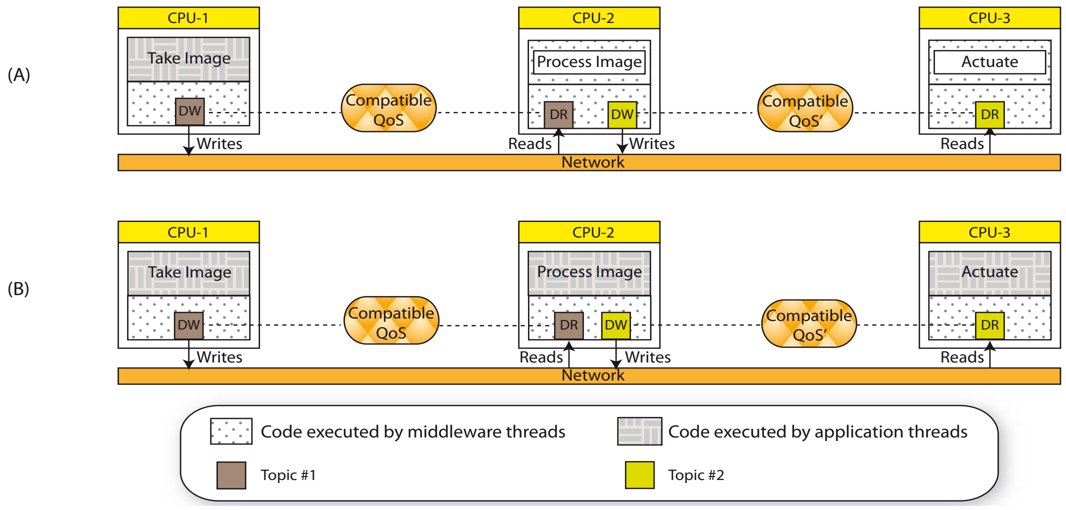

Figure 3-2: Asynchronous end-to-end flow model based on data-centric DDS model

procedures or methods belonging to a remote object. According to this model, this part of the user code will be executed by internal middleware tasks, and therefore middleware should provide support to explicitly assign their scheduling parameters [PER08].

On the other hand, the data-centric model used in DDS would require the definition of (see Figure 3-2):

- Two topics to describe both the image and the command to be executed.
- Four entities to perform the distribution, a DW-DR pair is required for each topic registered in the system.

According to the model followed by DDS, each DW-DR pair has an associated set of specific QoS parameters, and the configuration of each entity must be performed individually: the scheduling parameters for the network messages are configured explicitly in each DW through the *Transport_Priority* parameter, while the configuration of the processing tasks is not considered by the standard and therefore it remains implementation-defined. Under this model, the remote procedures *Process Image* and *Actuate* could be executed using either internal middleware tasks, by means of the *Listener* mechanism (see Figure 3-2-A), or application tasks, through *Wait-set* structures (see Figure 3-2-B).

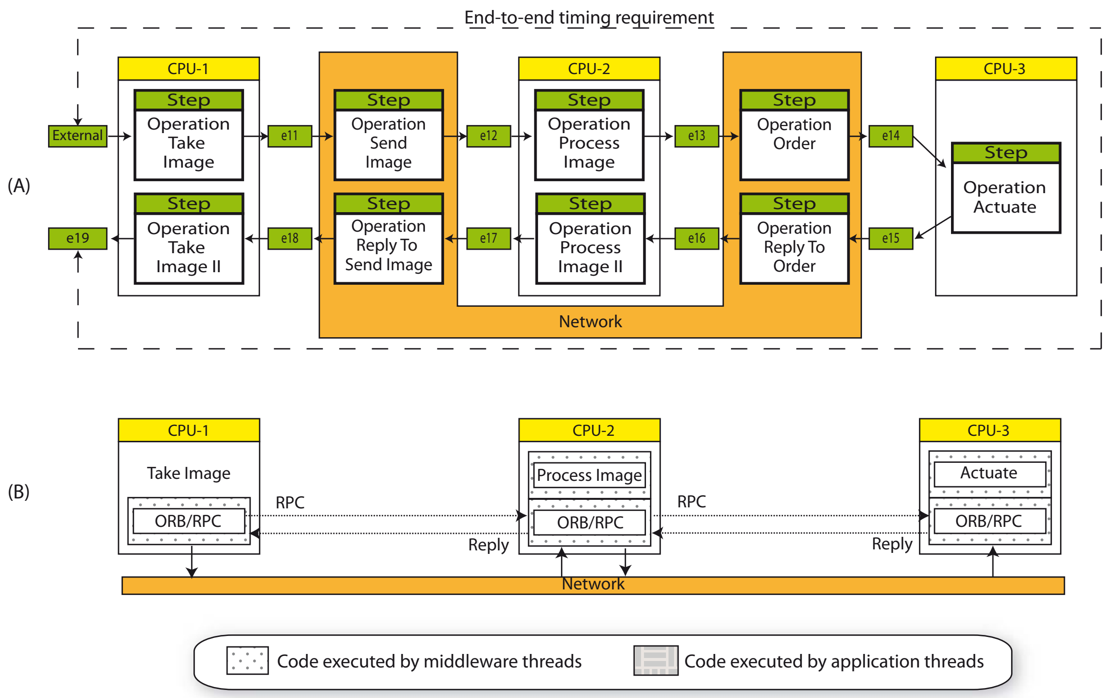

Figure 3-3: Synchronous end-to-end flow model and distribution model based on ORB and RPCs

3.2.2 MODELLING OF SYNCHRONOUS REMOTE CALLS

In end-to-end flows that represent synchronous remote calls, the theoretical model is extended with a second chain of activities and events representing the return path. Figure 3-3-A shows the same example used in the previous section but, in this case, the system has been modified to return the result of the remote calls. Figure 3-3-B shows the model based on ORB/RPCs, which is similar to the asynchronous case but requires the creation of two new message streams for the replies. However, the DDS model, which is essentially asynchronous, requires the definition of two new topics to determine each result and their corresponding entities for the distribution besides the two message streams. Figure 3-4 shows two possible approaches to build this type of synchronous end-to-end flow using DDS: *Listeners* are used in Figure 3-4-A while *Wait-sets* are used in Figure 3-4-B. In this case, it is important to remark that this model:

- Requires the definition of a second part of the procedures to process the results (*Take Image II* and *Process Image II*). This approach is quite similar to the end-to-end flow model used in the analysis and therefore facilitates the transition from the model to the real system.

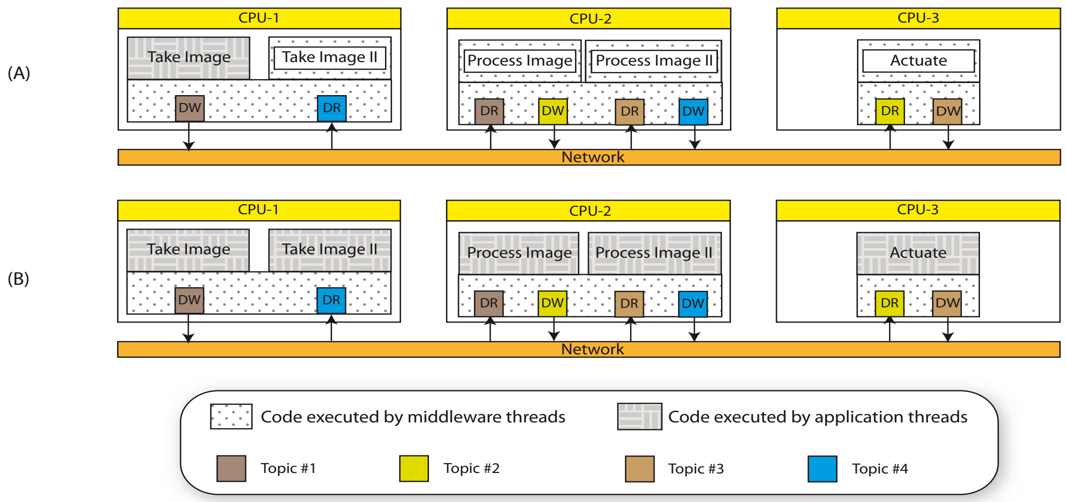

Figure 3-4: Synchronous end-to-end flow model based on data-centric DDS model

- Provides more flexibility to build the system. For example, if the system described above is modified to return the result of the action performed only to CPU-1, in this case the adaptation of the DDS application is straightforward (see Figure 3-5) while the ORB/RPCs model requires the reply to come through CPU-2 or the nature of the application to be changed to asynchronous.

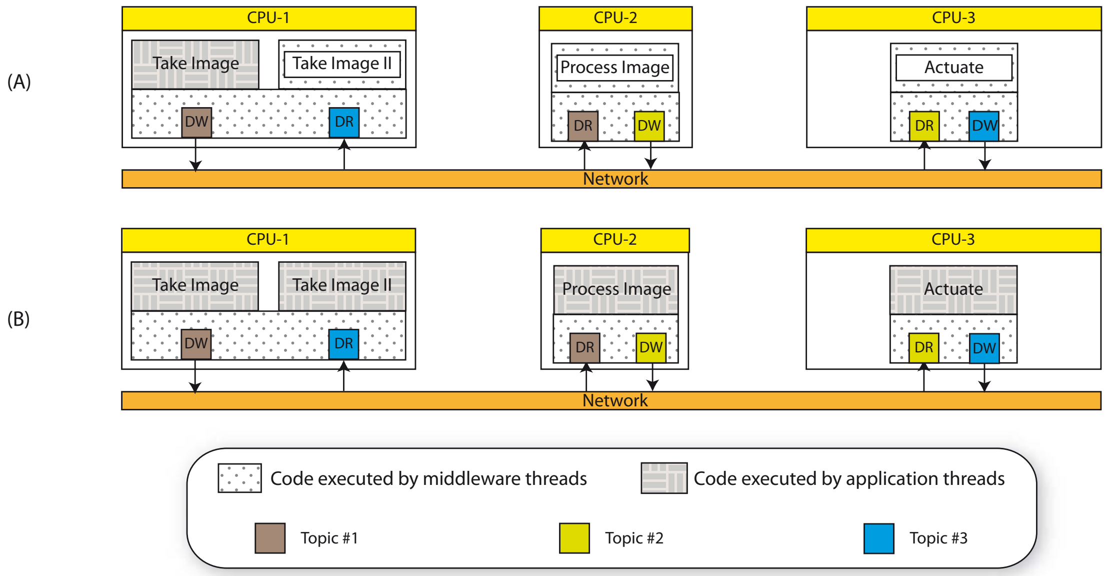

Figure 3-5: Synchronous and asynchronous end-to-end flow model based on data-centric DDS model

3.2.3 VALIDATION OF THE END-TO-END FLOW MODEL IN DISTRIBUTION MIDDLEWARE

After completing the analysis, we can conclude that the end-to-end flow model can be applied to the three specifications, although the distribution model proposed by DDS displays greater similarity to the theoretical model. Therefore, a distributed application that conforms to any of these standards, either using synchronous or asynchronous communication modes, can be represented as a set of end-to-end flows. Although this study has only focused on the basic communication mechanisms between entities, its results can validate the use of the end-to-end flow model in a distributed system following the RT-CORBA, DSA or DDS standards. However, we cannot conclude that any mechanism included in these standards (for example, those defined by DDS to provide a certain QoS) is suitable to be modelled with the end-to-end flow model. This part of the study, which represents an interesting line of research, is beyond the scope of this thesis and it is proposed as future work.

Furthermore, the modelling and analysis of current distributed systems, which can be composed of dozens of tasks and several processors, is a complex and error prone process that usually requires CASE tools to be applied. To this end, not only should the real-time model be abstracted from the distributed application, but distribution middleware should also be adapted to it. This integration would incorporate the capability to explicitly create, configure and manage those entities related to the end-to end flow model within distribution middleware. For this purpose, a set of entities to support the real-time model should be identified. This point constitutes one of the objectives of this thesis, whose proposal defines a set of entities, configuration interfaces and new services which are called the *endpoints pattern*.

3.3 THE ENDPOINTS PATTERN

The *endpoints pattern* aims to provide a homogeneous solution for creating and configuring the entities within the end-to-end flow model. As we explained earlier in Chapter 1, an end-to-end flow consists of a sequence of interrelated events and steps whose execution must usually satisfy end-to-end timing requirements. While the steps represent either the execution of a piece of code by means of a task or the transmission of a message through a network, the event is the entity responsible for triggering the execution of a step with specific scheduling parameters. The *endpoints pattern* incorporates the event concept by defining a new element called *Event_Id*. This element identifies an end-to-end flow in its execution as it is transmitted through the network within each remote call thus allowing the scheduling parameters associated with each step to be applied.

Once the Event_Id has been incorporated into the model, the next step in the modelling of distributed applications through end-to-end flows is the identification of the schedulable entities. In the case of the *endpoints pattern*, two kinds of schedulable entities are defined:

- For the processing nodes, the *handler tasks* intended to execute remote calls. These handlers are created explicitly with the appropriate scheduling information.
- For the network, the *endpoints* or communication points are used to transport messages through the network. There are two types of endpoints: those related to the transmission (*send endpoints*), which are associated with specific scheduling parameters; and those related to the reception (*receive endpoints*), which constitute the waiting mechanism for handlers tasks or those tasks that perform a synchronous remote call. As in the case of the handler tasks, the endpoints must be created explicitly and with a set of scheduling parameters associated with the transmission of messages through the network for send endpoints.

A send endpoint contains information about the network to be used, the destination address and the communication port that is used to establish the link between the send and the receive endpoints. A receive endpoint contains information about the network and the communication port to be used. In order to support both asynchronous and synchronous remote calls, the *endpoints pattern* distinguishes different types of send and receive endpoints. In particular, the following endpoints are considered:

- *Send endpoint* to send messages with specific scheduling parameters. It has an associated Event_Id to identify the specific message stream to use.
- *Receive endpoint* to listen for incoming requests. It is not directly associated with any Event_Id and thus it can process different end-to-end flows depending on the incoming events.
- *Send reply endpoint* to send the reply message of a synchronous remote call with specific scheduling parameters. It has an associated Event_Id to identify the specific message stream to use.
- *Receive reply endpoint* to listen for the reply message provoked by a synchronous remote call. It has an associated Event_Id that allows the calling task to wait for the reply message.

In Chapter 1, Figure 1-2 showed the set of basic services provided by distribution middleware to enable communications between nodes to be performed transparently from the developers' viewpoint. However, there is a loss of abstraction in a real-time system because, among other requirements, developers may have to specify the

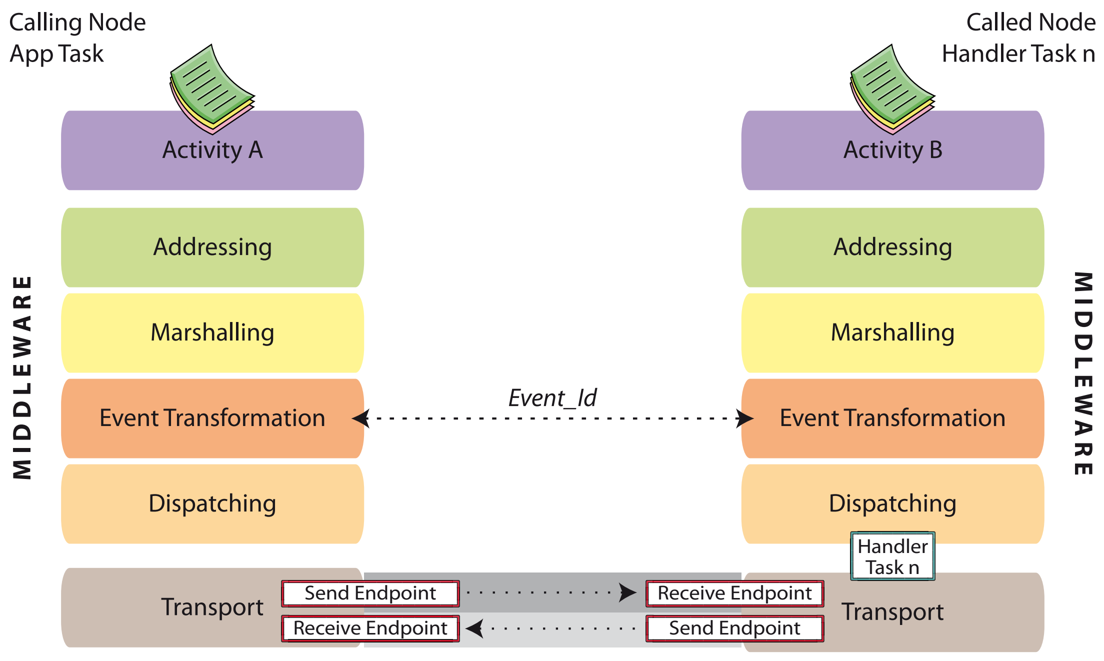

Figure 3-6: The endpoints pattern and distribution middleware

scheduling parameters of network messages. Thus, in order to guarantee the schedulability of the system, distributed calls must be known in advance by developers, who should configure them appropriately.

Figure 3-6 shows the entities and services added by the *endpoints pattern* to distribution middleware. First, the send and receive endpoints are created explicitly and configured to process one or more end-to-end flow. Similarly, the developer or real-time engineer controls both the handler tasks and their scheduling parameters, and is responsible for associating each handler task with the end-to-end flows to be processed.

Moreover, in addition to these entities, the *endpoints pattern* adds a new functionality to distribution middleware: the *event transformation service*. This service provides internal support for the end-to-end flow model, as it hides the management of the real-time details from the application code. In order to build an end-to-end flow, not only should the active components of an end-to-end flow be indicated but their precedence relationship should also be stipulated. Under the *endpoints pattern*, this precedence relationship is established by the aforementioned Event_Id element. The Event_Id is transmitted through the network with each remote call and is managed by the event transformation service to configure the end-to-end flow (see Figure 3-6).

To explain how the end-to-end flow model is related to distribution middleware and the *endpoint patterns*, we again consider the linear and asynchronous end-to-end flow example used in Section 3.2.1 and illustrated in Figure 1-3. For this end-

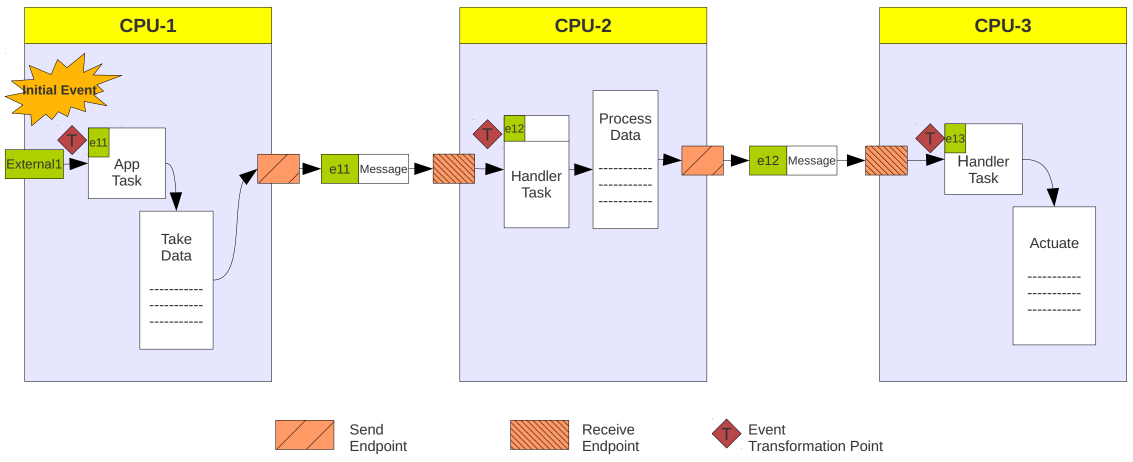

Figure 3-7: The endpoints pattern and the end-to-end flow model

to-end flow, each step generates a single internal event as output which, depending on its value, will trigger the execution of the next step with the appropriate scheduling parameters. Under the *endpoints pattern*, each of these internal events is identified through Event_Id. Figure 3-7 shows the details of the events, steps and processing resources (the processors or the network) according to the *endpoints pattern*. Furthermore, the event transformation service is represented through the *Event Transformation Points in* Figure 3-7. By setting the initial event *External1*, the application task triggers the activation of the end-to-end flow that generates the output event *e11*. This event causes a message to be sent to perform the first remote call. In CPU-2, distribution middleware identifies the incoming event at the receive endpoint and transforms it to the corresponding event *e12*. Then, middleware sets the appropriate scheduling parameters for the handler task depending on the event received. After processing the data, a message is sent to perform the remote call to CPU-3 with event *e12*. Finally, middleware executed in CPU-3 will transform the incoming event into event *e3* and will set the corresponding scheduling parameters for the actuation. This action completes the execution of the end-to-end flow.

By using the *endpoints pattern*, distributed real-time applications can be developed and configured separately in a flexible way. Among the major advantages of the proposed approach are:

- **Support for the management of complex linear end-to-end flows.** For instance, nested remote calls can be appropriately executed by means of the Event_Id and the event transformation service. Moreover, this event management is particularly useful in end-to-end flows that invoke the same remote operation two or more times, and which may require different scheduling parameters for each call. This case is illustrated in Figure 3-8

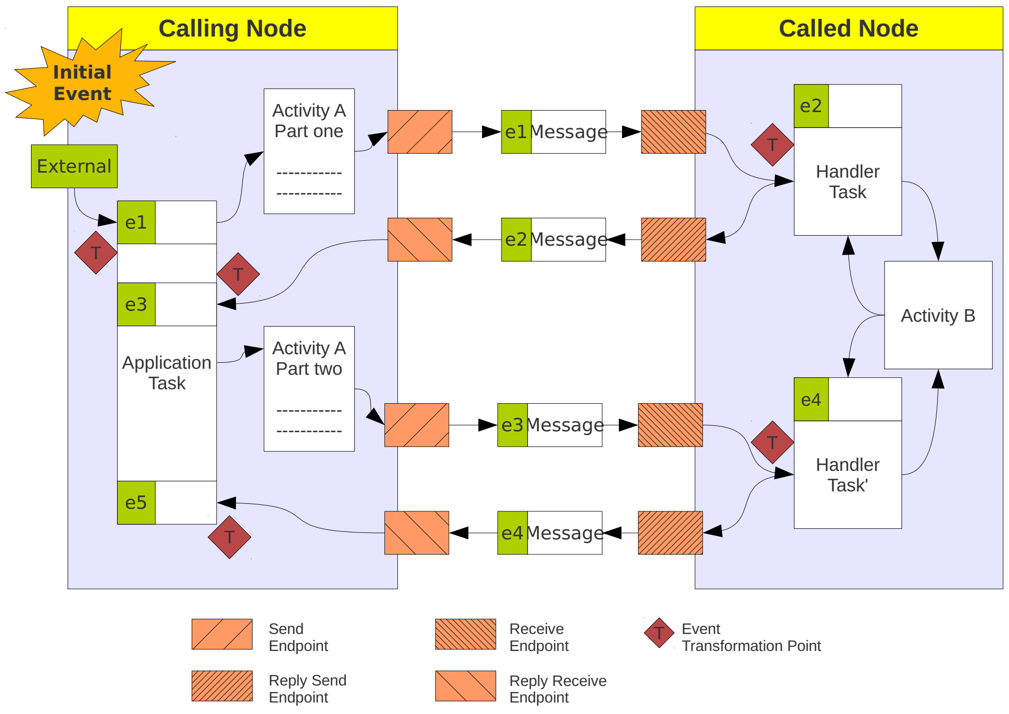

Figure 3-8: Event flow in complex linear end-to-end flows

where the application task makes several synchronous requests to the same remote service. Setting the initial Event_Id is sufficient to enable the middleware to identify the particular point in the end-to-end flow and therefore the scheduling parameters required in each case.

- **Support for the sharing of handler tasks by several end-to-end flows.** The *endpoints pattern* also allows a handler task to be shared among different end-to-end flows executing in turn with different scheduling parameters, which is interesting in order to reduce the total number of tasks in the system. Moreover, this feature enables the model to support complex non-linear end-to-end flows.
- **Support for separating the logic of application and the real-time aspects.** A desirable feature for a real-time model is the separation of concerns between the logic of application and the real-time aspects. Under the proposed real-time model based on the *endpoints pattern*, the creation of schedulable entities and scheduling parameters is performed by means of a set of interfaces that represent a simple configuration operation. Once the initial stage for the creation and configuration of entities is finished, middleware will be responsible of updating the scheduling parameters and managing the chain of events within the end-to-end flow in a transparent

way. Moreover, this real-time configuration process can be automated by applying MDE techniques based on the end-to-end flow model.

3.3.1 RELATED WORK

Over the last years there has been a significant effort to apply the end-to-end flow model to distribution middleware, which has led to several research works that constitute the basis of the *endpoints pattern*. These works include studies about modelling, analysis and integration of the end-to-end flow model with the distribution mechanisms mainly proposed for the Ada programming language. Thus, a previous work about the modelling of distributed applications with the DSA was presented in [GUT02], while [GUT99] discusses different methods for the assignment of priorities to optimise response times for an end-to-end flow. Both research works provided an initial integration of the end-to-end flow model into distribution middleware. Moreover, these initial proposals were extended and implemented in [LOP04] and [LOP06]. While the former enables the free assignment of priorities in remote calls (both in the processors and in the communication networks), the latter supported different scheduling policies and proposed a way to incorporate the end-to-end flow model into the DSA. In particular, the authors in [LOP06] proposed an interface for DSA to customize the communication layer and handler tasks and thus it can be considered a first step in the direction of the *endpoints pattern* presented in this thesis.

Compared to the approach defined in [LOP06], this thesis incorporates several features which provide the proposal with new services and more flexible support of the end-to-end flow model. Unlike the interface defined by [LOP06], the *endpoint pattern* considers the use of different scheduling policies for processors and networks. Moreover, while [LOP06] associates each receive endpoint with a single event, the *endpoint pattern* allows multiples events to be handled by each receive endpoint (i.e. it allows handler tasks to be shared by several end-to-end flows). The *endpoint pattern* also adds a new event transformation service to avoid the management of real-time aspects within the application code. Lastly, it can support different distribution models and can be used in high-integrity systems as will be shown later.

3.4 THE ENDPOINTS API

The proposal included in this thesis defines a set of interfaces (API) to provide complete support for the end-to-end flow model. These interfaces enable distribution middleware to create and configure the schedulable entities, as well as the management of resources in the processors and communication networks. The *endpoints pattern* includes separate interfaces for handler tasks and communication endpoints to ease the integration of new, possibly different, scheduling policies both in processing nodes and networks. Moreover, the proposal also defines a third interface to allow middleware and applications to manage event associations through the event transformation service. The following sections of this chapter describe the functionality of each of these interfaces.

3.4.1 NETWORK SCHEDULING INTERFACE

Meeting real-time requirements in a distributed application requires the networks to be scheduled with appropriate techniques. The *endpoints pattern* considers the scheduling of communication networks by making the communication endpoints visible, and by associating scheduling parameters to the messages sent through them.

The proposed interface is abstract, that is, it should be extended by each scheduling policy according to its own scheduling parameters (e.g. priorities, deadlines or more complex parameters for contract-based scheduling). The network scheduling interface defines the following primitives:

- *Create_Send_Endpoint*. Subprogram to create a new endpoint to send requests through a particular network. This kind of endpoint is associated with specific scheduling information.
- *Create_Receive_Endpoint.* Primitive to create an endpoint in a particular network to receive requests. Handler tasks are responsible for awaiting the arrival of incoming requests in such endpoints.
- *Create_Reply_Send_Endpoint*. Subprogram to create a communication endpoint to send replies through a particular network in a synchronous request. This kind of endpoint is associated with specific scheduling information.
- *Create_Reply_Receive_Endpoint.* Primitive to create an endpoint in a particular network to listen to replies to a synchronous remote call. Therefore, the task responsible for awaiting the reply is the same as the one that performed the remote call.

Finally, corresponding subprograms are provided by the interface to destroy the endpoints created.

3.4.2 PROCESSING NODE SCHEDULING INTERFACE

In a distributed real-time system, the arrival of concurrent remote requests with different degrees of urgency is frequent. Handler tasks are responsible for awaiting arriving requests and processing them.

As in the case of networks, the interface that allows the handler tasks to be created and configured with appropriate scheduling parameters is abstract, and therefore it should be extended for each scheduling policy to be used. The processing node scheduling interface includes the following primitives:

- *Create_Handler_Task*. Primitive to create a handler task associated with specific scheduling parameters. This handler task is responsible for processing incoming requests arriving at a single receive endpoint (i.e. it is bound to a particular receive endpoint).

Since a single handler task could process several requests matching different end-to-end flows, each of them with its specific scheduling information, this interface defines two primitives for the dynamic update of scheduling parameters:

- *Set_Event_And_Sched_Params_Association*. Subprogram to link a particular event to the scheduling parameters provided.
- *Update_Scheduling_Parameters*. Subprogram to update the scheduling parameters for a selected handler task.

Finally, corresponding subprograms are provided by the interface to destroy the handler tasks created.

3.4.3 EVENT MANAGEMENT INTERFACE

The event transformation service completes the support for the development of the end-to-end flow within distribution middleware. In the end-to-end flow model, external events trigger the end-to-end flows, and the setting of an identifier of those events is the only operation that our model requires the application code to perform. Once the application has set this external event at the beginning of the end-to-end flow, all the subsequent steps are scheduled according to the associated internal event at each moment. These internal events are transmitted through the end-to-end flow by means of the Event_Id parameter, which enables middleware to identify a particular point in the end-to-end flow and therefore the scheduling parameters required in each case. Furthermore, these internal events are automatically set by middleware at the transformation points, which match the receive endpoints used by both handler and application tasks.

Therefore, the developer or real-time engineer should configure the event flow within an end-to-end flow, and middleware will be in charge of automatically setting the appropriate event at the transformation points of the remote call. For this purpose, the event management interface provides the following primitives:

- *Set_Event_Assocation*. Subprogram to set associations between input and output events. Middleware will use this link to select an appropriate output event at the specified transformation points.

- *Get_Event_Association*. Primitive to return the output event associated with the specified input event.
- *Set_Event_Id*. Subprogram to set the current event associated with a task.
- *Get_Event_Id*. Primitive to get the current event associated with a task.

3.4.4 USING THE CONFIGURATION INTERFACE

The integration of the proposed interfaces with distribution middleware enables the developer to explicitly specify all the details related to system scheduling. To this end, the developer should use the proposed configuration interfaces (i.e. processing node scheduling, network scheduling and event management interfaces) to specify all the end-to-end flow elements belonging to the application, regardless of the selected scheduling policy.

Within the initialization stage of a system, the application is responsible for creating the schedulable entities and configuring the scheduling parameters associated with tasks, network messages and the execution of remote calls. All these operations can be performed within a new phase which is called *real-time configuration stage*. This real-time configuration is completely independent of the functional parts of the application, and its programming may even be done by another developer specialized in real-time systems. Those aspects related to the coordinated and deterministic initialization of nodes in a distributed system are beyond the scope of the real-time model and, therefore, the *endpoints pattern*. Nevertheless, this aspect is really important in a distributed system and so its study remains open to future work.

Once the system has been initialized, the distributed real-time application will start to execute and then only distribution middleware will make use of the API to automatically manage the events transformation and the update of scheduling parameters if required.

Furthermore, although the *endpoints pattern* is suitable for use in static systems (i.e. their workload is within known bounds), it can also be used in dynamic systems whose workload varies with time. This would require higher level software tools to be applied which are responsible for guaranteeing determinism during the creation of new entities, as well as during the execution of new end-to-end flows (i.e. new end-to-end flows will not affect the temporal behaviour of the rest of the system). The validation of the use of the *endpoints pattern* in both types of systems is part of the next chapter of this thesis, and is included in the following research works: [PER09] and [SAN10].

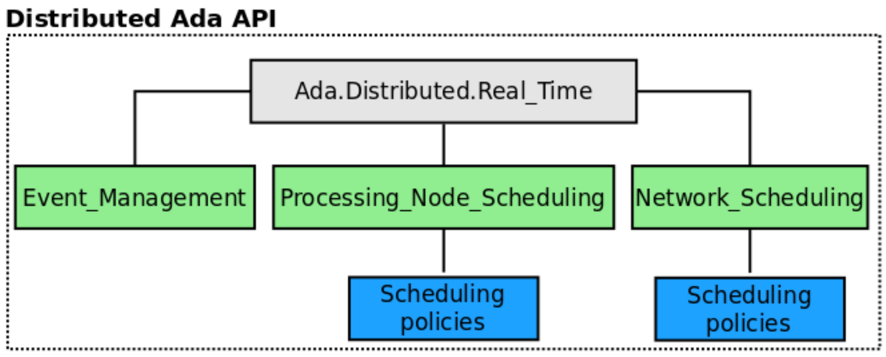

Figure 3-9: Package hierarchy for end-to-end flow DSA systems

3.5 INTEGRATION INTO THE ADA STANDARD

Once the functionality of the *endpoints pattern* has been described, the next step is to define a specific API for Ada that allows the development of distributed real-time applications. In particular, we have defined a set of Ada packages that can be integrated into the DSA, as is illustrated in Figure 3-9.

Firstly, the package defined in Listing 3-1 includes the identifiers for the basic elements to be set up in a distributed system: communication networks, send/receive ports and nodes.

The second package, which is described in Listing 3-2, contains the identifiers of common elements in the proposed end-to-end flow model for real-time environments. The *Send_Endpoint_Id* and *Receive_Endpoint_Id* are the types used to identify endpoints, which are used as links to enable the communication among different nodes. The type *Event_Id* implements the event defined by the endpoints pattern and it is used to attach an identifier which associates scheduling parameters to the schedulable entities within the end-to-end flow (send endpoints or handler tasks) and to identify the endpoint where a task performing an RPC should await the reply. This *Event_Id* is the only data

Listing 3-1: Package a-distributed.ads

```
package Ada.Distributed is
  pragma Pure;
  type Port_Id is range implementation-defined;
  type Node_Id is range implementation-defined;
  type Network_Id is range implementation-defined;
end Ada.Distributed;
```

Listing 3-2: Package a-distributed-real_time.ads

```
package Ada.Distributed.Real_Time is
  pragma Pure;
  type Send_Endpoint_Id is private;
  type Receive_Endpoint_Id is private;
  type Event_Id is range implementation-defined;

private
  type Send_Endpoint_Id is ...;
  type Receive_Endpoint_Id is ...;
end Ada.Distributed.Real_Time;
```

type which is defined within the public part of the Ada specification (see Listing 3-2). As was discussed in Section 3.4, this seems reasonable since this parameter is mainly managed within the application code (for example, on setting the initial event).

The interface responsible for network scheduling is defined in the following Ada package (see Listing 3-3). This interface is abstract and holds the primitives in charge of creating and destroying the communication endpoints. Each of these subprograms receives the minimum information necessary to establish communication as input parameters (i.e., the network and the communication port). Additionally, these subprograms responsible for the creation of the send endpoints also receive an identifier for the destination node and an abstract tagged private type called *Message_Scheduling_Parameters* as input parameters. Extensions of the *Message_Scheduling_Parameters* tagged type will contain the specific network scheduling parameters that must be associated with a specific send endpoint. Furthermore, each scheduling policy must implement subprograms to map its own scheduling parameters (e.g., priorities, deadlines, or contract-based parameters) onto extensions of this private type.

As in the case of the network scheduling interface, the Ada package responsible for the processing node scheduling (which is defined in Listing 3-4) includes an abstract tagged private type called *Task_Scheduling_Parameters*. This tagged type should be extended with appropriate data for the scheduling policy supported (e.g., deadlines for EDF or more complex parameters for contract-based scheduling). Furthermore, supported policies must implement subprograms to map their own scheduling parameters onto this private type. This interface allows end users (1) to create handler tasks, specifying both the initial scheduling parameters and the receive endpoint associated with this new handler task, and (2) to associate the processing of an incoming event with specific scheduling parameters.

In addition to the public interface to create and configure handler tasks, middleware will require a primitive to internally manage the scheduling parameters associated with a handler task. This primitive is only used by the middleware, so a new Ada package is provided to separate the application user interface from the middleware's

Listing 3-3: Package a-distributed-real_time-network_scheduling.ads

```
package Ada.Distributed.Real_Time.Network_Scheduling is
  type Message_Scheduling_Parameters is abstract tagged private;
  type Message_Scheduling_Parameters_Ref is access all
    Message_Scheduling_Parameters'Class;

  procedure Create_Receive_Endpoint
   (Net         : Network_Id;
    Port        : Port_Id;
    Endpoint    : out Receive_Endpoint_Id) is abstract;

  procedure Create_Send_Endpoint
   (Param       : Message_Scheduling_Parameters_Ref;
    Dest_Node   : Node_Id;
    Event       : Event_Id;
    Net         : Network_Id;
    Dest_Port   : Port_Id;
    Endpoint    : out Send_Endpoint_Id) is abstract;

  procedure Create_Reply_Receive_Endpoint
   (Net         : Network_Id;
   Event_Sent   : Event_Id;
   Port         : Port_Id;
   Endpoint     : out Receive_Endpoint_Id) is abstract;

  procedure Create_Reply_Send_Endpoint
   (Param       : Message_Scheduling_Parameters_Ref;
    Dest_Node   : Node_Id;
    Event       : Event_Id;
    Net         : Network_Id;
    Dest_Port   : Port_Id;
    Endpoint    : out Send_Endpoint_Id) is abstract;

  procedure Destroy_Receive_Endpoint
   (Endpoint    : Receive_Endpoint_Id) is abstract;
  procedure Destroy_Send_Endpoint
   (Endpoint    : Send_Endpoint_Id) is abstract;
private
  type Message_Scheduling_Parameters is abstract tagged ...;
end Ada.Distributed.Real_Time.Network_Scheduling;
```

one. The description of this package is shown in Listing 3-5. The subprogram *Update_Scheduling_Parameters* will update scheduling parameters for a selected handler task, therefore allowing different concurrency patterns to process remote requests as will be discussed in Section 3.8.

Finally, the last proposed interface, which is defined in Listing 3-6, is responsible for providing the necessary subprograms to (1) configure the event flow

Listing 3-4: Package a-distributed-real_time-processing_node_scheduling.ads

```
package Ada.Distributed.Real_Time.Processing_Node_Scheduling is
  type Task_Scheduling_Parameters is abstract tagged private;
  type Task_Scheduling_Parameters_Ref is access all
     Task_Scheduling_Parameters'Class;
  type Handler_Id is range implementation-defined;

  procedure Create_Handler_Task
   (Default_Params    : Task_Scheduling_Parameters_Ref;
    Endpoint          : Receive_Endpoint_Id;
    Handler       : out Handler_Id) is abstract;

  procedure Destroy_Handler_Task
   (Handler       : Handler_Id) is abstract;

  procedure Set_Event_And_Sched_Params_Association
   (Params            : Task_Scheduling_Parameters_Ref;
    Event             : Event_Id) is abstract;

  function Current_Handler_Id return Handler_Id;

private
  type Task_Scheduling_Parameters is abstract tagged ...;
end Ada.Distributed.Real_Time.Processing_Node_Scheduling;
```

within an end-to-end flow and (2) handle event identifiers (e.g. for setting the initial event that triggers the start of the end-to-end flow).

3.6 AUTOMATIC GENERATION OF THE REAL-TIME CONFIGURATION

The creation and configuration of communication endpoints and handler tasks, and the assignment of scheduling parameters to them, may be seen as a complex task that requires the application developer to add a lot of code to an already complex application. However, most of the work can be done at initialization time, and all the information

Listing 3-5: Package a-distributed-real_time-processing_node_scheduling-internals.ads

```
package      Ada.Distributed.Real_Time.Processing_Node_Scheduling.Internals is
  procedure Update_Scheduling_Parameters
   (Params       : Task_Scheduling_Parameters_Ref;
    Handler  : Handler_Id := Current_Handler_Id) is abstract;
end        Ada.Distributed.Real_Time.Processing_Node_Scheduling.Internals;
```

Listing 3-6: a-distributed-real_time-event_management.ads

```
package Ada.Distributed.Real_Time.Event_Management is

  procedure Set_Event_Association
   (Input_Event       : Event_Id;
    Output_Event      : Event_Id);

  function Get_Event_Association
   (Input_Event       : Event_Id) return Event_Id;

  procedure Set_Event_Id (New_Event : Event_Id);

  function Get_Event_Id return Event_Id;

end Ada.Distributed.Real_Time.Event_Management;
```

needed can be automatically obtained from the real-time end-to-end flow model. From this model it is possible to make a transformation to automatically generate the configuration code that needs to be run at initialization time, thus easing the application of CASE tools. Following this approach, the application code would only need to include a simple *Set_Event_Id* call to set the initial event that triggers the end-to-end flow.

With the increasing complexity of software applications, the use of model-driven development processes such as MDA is becoming more and more frequent. This type of methodologies requires a model of real-time behaviour in which the operations, tasks, messages, triggering events and their interactions and deployment on a particular hardware platform are explicitly described. This model allows the application developer to perform a real-time analysis in which the timing requirements of the applications can be validated. This model can be generated simply for the purpose of analysis, or it could be obtained from design information of the application developed, for instance, with the MARTE UML profile for embedded real-time systems [MAR08].

The model proposed by MAST [HAR01] allows the real-time behaviour of an application to be represented and it may be obtained from the *Schedulability Analysis Modeling* (SAM) subprofile included in MARTE [MED11]. The MAST model contains descriptions of the execution and communications platform, the concurrent architecture of the application, its operations including any synchronization, and the end-to-end flows that describe the flow of events in the system. This software includes tools for performing schedulability analysis as well as automatic assignment of scheduling parameters, or sensitivity analysis.

Although the MAST model contains most of the information that is required to automatically generate the configuration of each architectural element defined in the *endpoints pattern*, the integration of this model into distribution middleware is not straightforward. Therefore, the current MAST model would need to be augmented to describe a few additional properties that are required for this automatic configuration.

- *Identification of the nature of each task*: Unlike the *endpoints pattern*, MAST does not differentiate whether tasks are dynamically created by middleware or not, that is, if it is an application task or a handler task. The latter is associated with a receive endpoint and is usually created by middleware.
- *Identification of the nature of messages.* MAST models synchronous remote calls as if they were asynchronous, and it does not distinguish between the call and the reply of an RPC. However, the *endpoints pattern* clearly distinguishes between RPCs and APCs because they use a different configuration for the endpoints.

Once these features have been incorporated, the application of a set of rules would be enough to generate the real-time configuration code automatically:

- For each message transmission operation, create a send endpoint in the sending node (with the scheduling parameters associated with the message stream), and a receive endpoint in the receiving node.
- For each handler task, create the corresponding handler with its specific scheduling parameters.
- For each external event, add the corresponding *Set_Event_Id* call.
- For each input event to a step, generate the mapping between the event and the corresponding scheduling parameters, as well as the association between input and output events.

3.7 EXAMPLE OF USE

This section describes a simple example to illustrate the usage of the proposed configuration interface. We once again consider the example described in Section 3.2.1 but, in this case, with two linear end-to-end flows performing asynchronous remote procedure calls through three processors and using one communication network. The elements belonging to each end-to-end flow are the same as were defined in Section 3.3 and depicted in Figure 3-7. The MAST model of this system is shown in Figure 3-10 and it includes the details of the events, steps and processing resources (the processors or the network). Each processor contains only a single Ada partition, so for our purposes,

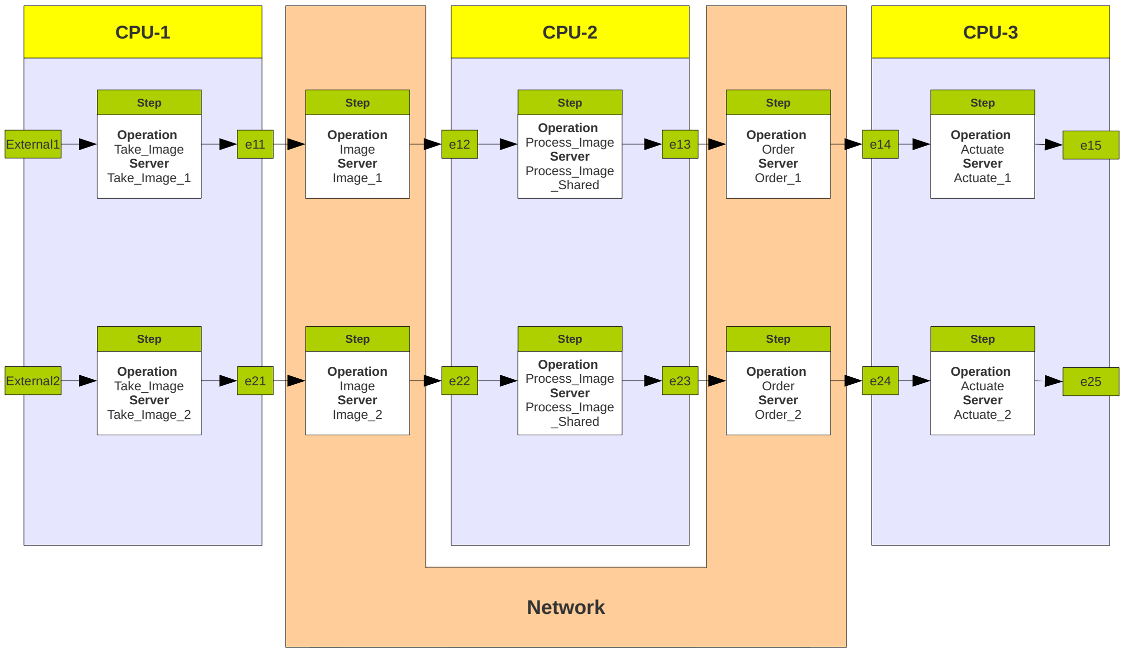
CPU-1
CPU-2
CPU-3
Network
External1
External2
Step
Operation Take_Image Server Take_Image_1
Operation Take_Image Server Take_Image_2
Operation Image Server Image_1
Operation Image Server Image_2
Operation Process_Image Server Process_Image _Shared
Operation Order Server Order_1
Operation Order Server Order_2
Operation Actuate Server Actuate_1
Operation Actuate Server Actuate_2
e11
e12
e13
e14
e15
e21
e22
e23
e24
e25

Figure 3-10: MAST model for the example system

partitions and processors are equivalent. Note that in CPU-2 the *Process_Image* step is performed by the same handler task (*Process_Image_Shared*) for both end-to-end flows.

As can be seen in Figure 3-11, our approach only contains the entities implemented in the application. In our case, the external events *External1* and *External2* trigger the execution of the steps in CPU-1 which, in turn, generate output events *e11* and *e21*. Those events cause messages to be sent to perform the remote call to the *Process_Image* operation. CPU-2 identifies the incoming events (*e11* or *e21*) in the receive endpoint and transforms them to the corresponding events *e13* or *e23*. Then, middleware sets the appropriate scheduling parameters for the handler tasks depending on the event received. After executing the steps in CPU-2, messages are sent to perform the new remote call to CPU-3 with the current events. Finally, middleware in CPU-3 will transform the incoming events into events *e15* and *e25* which will be used to set the corresponding scheduling parameters for the steps. This ends the execution of the end-to-end flows.

Figure 3-12 shows the Ada packages used in this example. These packages can be classified into two different groups of files: (1) real-time configuration files and (2) application code, which is divided in turn into the *remote call interfaces* and the *main procedures*. The former are developed as Ada packages for CPU-3 and CPU-2 (see Listing 3-7 and Listing 3-8, respectively). Listing 3-8 also shows how the remote call to CPU-3 is performed. The latter are represented in Listing 3-9, Listing 3-10 and Listing 3-

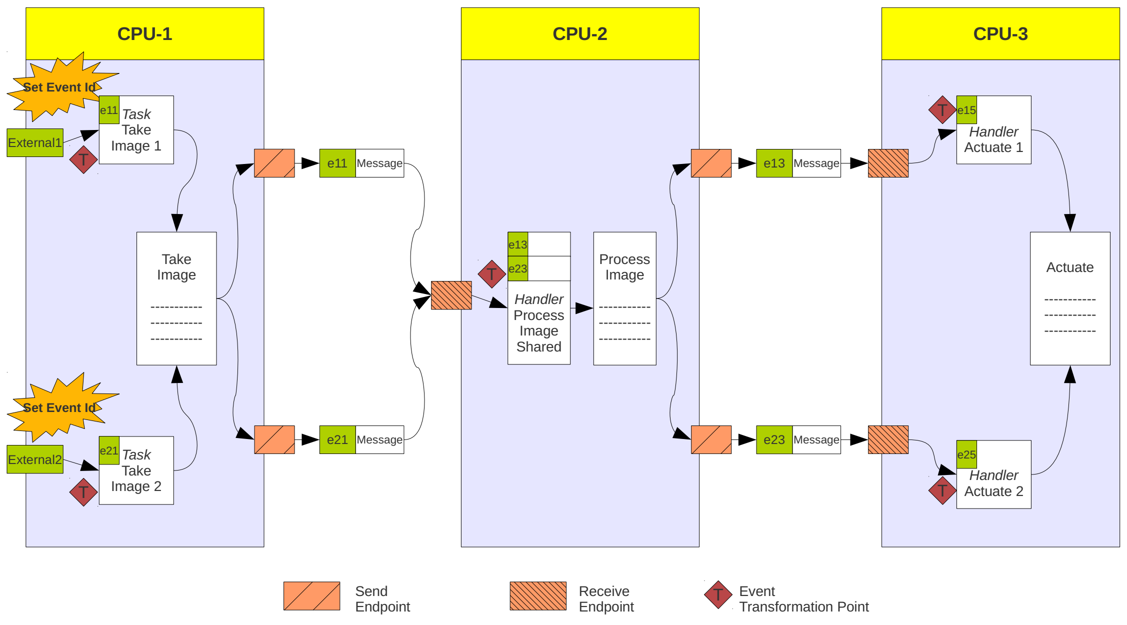

Figure 3-11: Simplified model for the example system

11 for CPU-3, CPU-2 and CPU-1, respectively. While CPU-3 and CPU-2 only require the real-time configuration to be performed for the end-to-end flows, CPU-1 also requires the initial event to be set and the *Process_Image* remote request to be performed (see Listing 3-11). As a result, we can appreciate the separation of concerns achieved by the proposal between the application logic and the real-time configuration code.

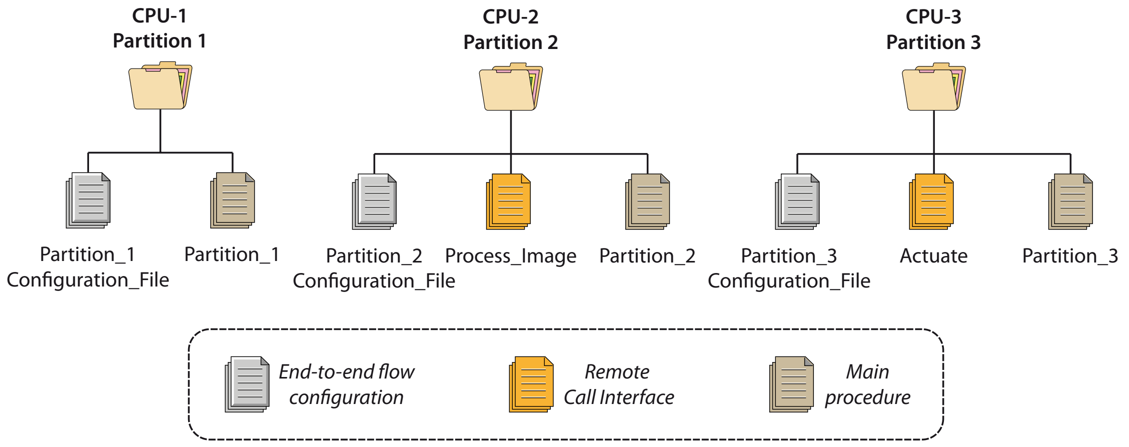

Figure 3-12: Package structure for the MAST example

Listing 3-7: Packages actuator.ads and actuator.adb

```
package Actuator is
  pragma Remote_Call_Interface;
  procedure Actuate (The_Command : Command_Type);
  pragma Asynchronous;
end Actuator;

package Actuator is
  procedure Actuate (The_Command : Command_Type) is
  begin
    ... -- Application code for the actuation
  end Actuate;
end Actuator;
```

Listing 3-8: Packages image_analysis.ads and image_analysis.adb

```
package Image_Analysis is
  pragma Remote_Call_Interface;
  procedure Process_Image (The_Image : Image_Type);
  pragma Asynchronous;
end Image_Analysis;

with Actuator; -- Other dependences
package body Image_Analysis is
  procedure Process_Image (The_Image : Image_Type) is
  begin
    ... -- Process current image and decide the actuation
    Actuator.Actuate (Current_Command);
  end Process_Image;
end Image_Analysis;
```

Listing 3-9: Procedure partition_3.adb

```
procedure Partition_3 is
begin
  Partition_3_Configuration_File;
  ... -- Application code
end Partition_3;
```

As was described in Section 3.6, the real-time configuration file for each partition would be generated automatically from the MAST model shown in Figure 3-10. As an example, we will show how to generate the configuration code for CPU-2 and a similar procedure could be followed for CPU-1 and CPU-3. The configuration code for CPU-2 should include the following procedure calls (see Listing 3-12):

Listing 3-10: Procedure partition_2.adb

```
procedure Partition_2 is
begin
  Partition_2_Configuration_File;
  ... -- Application code
end Partition_2;
```

Listing 3-11: Procedure partition_1.adb

```
with Image_Analysis; -- Other dependences
procedure Partition_1 is

  task Take_Image_1 is
    ... -- Set task properties
  end Take_Image_1;

  task Take_Image_2 is
    ... -- Set task properties
  end Take_Image_2;

  task body Take_Image_1 is
  begin
      Set_Event_Id (External1);
      ... -- Application code
     Image_Analysis.Process_Image (Curren_Image);
  end Take_Image_1;

  task body Take_Image_2 is
  begin
      Set_Event_Id (External2);
      ... -- Application code
     Image_Analysis.Process_Image (Curren_Image);
  end Take_Image_2;

begin
  Partition_1_Configuration_File;
  ... -- Application code
end Partition_1;
```

- *Set_Event_Association*: As can be seen in Figure 3-10, CPU-2 receives *e11* or *e21* as incoming events and transforms them to events *e13* or *e23* respectively. Such mapping between events remains registered in the middleware for its management at runtime.
- *Create_Receive_Endpoint* and *Create_Send_Endpoint*: CPU-2 considers four network transmission steps: two of them are for sending messages

Listing 3-12: Procedure partition_2_configuration.adb

```
procedure Partition_2_Configuration is
  -- The definition of variables is omitted
begin

  --  Set event associations
  Set_Event_Association (Input_Event  => e11,
                         Output_Event => e13);
  Set_Event_Association (Input_Event  => e21
                         Output_Event => e23);

  --  Create receive endpoint for shared RPC handler
  Create_Receive_Endpoint (Net      => Default_Network,
                           Port     => Receive_Port,
                           Endpoint => Rcv_Endpoint_Id);

  --  Create RPC Handler and scheduling params associated
  Set_Event_And_Sched_Params_Association
     (Params => Process_Image_Shared_Sched_Params_e13,
      Event  => e13);
  Set_Event_And_Sched_Params_Association
     (Params => Process_Image_Shared_Sched_Params_e23,
      Event  => e23);
  Create_RPC_Handler
     (Default_Params => Process_Image_Shared_Sched_Params,
      Endpoint       => Rcv_Endpoint_Id);

  --  Create two send endpoints
  Create_Send_Endpoint (Param     => Msg_Scheduling_Parameters_e13,
                        Dest_Node => Partition_3,
                        Event     => e13,
                        Net       => Default_Network,
                        Dest_Port => Rcv_Port_Partition_e13,
                        Endpoint  => Snd_Endpoint_Id_e13);
  Create_Send_Endpoint (Param     => Msg_Scheduling_Parameters_e23,
                        Dest_Node => Partition_3,
                        Event     => e23,
                        Net       => Default_Network,
                        Dest_Port => Rcv_Port_Partition_e23,
                        Endpoint  => Snd_Endpoint_Id_e23);
end Partition_2_Configuration;
```

(scheduled by *Order_1* and *Order_2*) and therefore associated with two send endpoints with their specific scheduling information. The other two network steps (scheduled by *Image_1* and *Image_2*) correspond to incoming messages and therefore they must be mapped to receive endpoints. In this case only one receive endpoint is required because both input events (*e11* and *e21*) are processed by the same handler task (which is identified as *Process_Image_Shared* in Figure 3-10).

- *Create_Handler_Task*: The *Process_Image_Shared* task acts as a handler task and therefore must be created explicitly by distribution middleware.
- *Set_Event_And_Sched_Params_Association*: This call allows CPU-2 to be configured in order to have the *Process_Image* operation executed by a single handler task. Thus, both end-to-end flows will share the handler task and this operation can be executed with different scheduling parameters depending on the triggering event after the corresponding transformation (*e13* or *e23*).

3.8 THE ENDPOINTS PATTERN AND THE CONCURRENCY PATTERNS

One of the basic issues in the design of real-time applications lies in bounding the WCRTs. These values do not depend exclusively on the application logic, which is influenced by the concurrent aspects of the system, such as context switches or priority inversions. However, due to the complexity of current software systems, other factors may be involved in the choice of the concurrency pattern for the execution of remote calls (for example, memory constraints in embedded systems or scalability in distributed systems). Therefore, current distribution middleware often relies on different concurrency patterns according to their most critical requirements.

To better handle the configuration of distributed real-time applications, the *endpoints pattern* not only makes visible their schedulable entities, but it also provides complete support for the end-to-end flow model. Furthermore, the *endpoints pattern* is closely related to concurrency patterns as it provides the support required to configure the handler tasks defined by middleware.

As we discussed in Chapter 2, hard real-time systems usually require a concurrency pattern which tends to minimize context switches and avoids the dynamic update of scheduling parameters. Moreover, these requirements can also be useful in other kind of real-time systems such as flexible scheduling environments [ALD06] [FRSH11]. Therefore, this section aims to select the most appropriate concurrency pattern to be used with the *endpoints pattern.* To this end, we briefly review below the most common concurrency patterns used in distribution middleware for processing concurrent remote calls.

In non-tasking environments, applications instantiate one task to process all the remote requests. This task is responsible for I/O monitoring and processing the incoming requests sequentially, so new requests arriving at the system will be blocked until the previous request is completed. This behaviour is illustrated in Figure 3-13-A, in

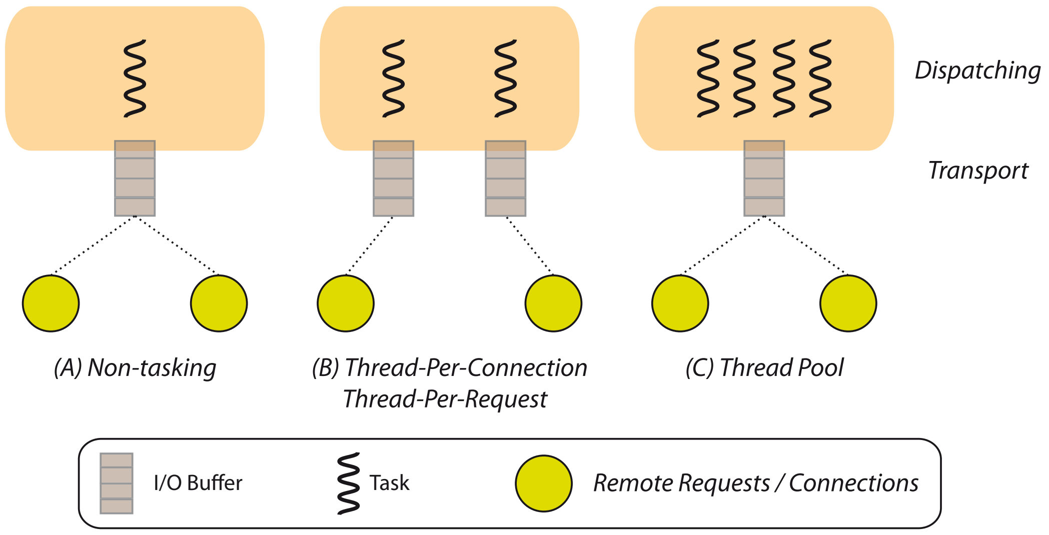

Figure 3-13: Concurrency patterns commonly used in distribution middleware

which all the remote invocations performed by each system node converge at a single communication point.

In multi-tasking environments, applications instantiate multiple tasks to process incoming requests concurrently. Some examples for multi-tasking concurrency patterns include the *Thread-Per-Connection* pattern [SCH98] (see Figure 3-13-B), which creates a handler task for each session or connection request destroys it when the connection is closed, and the *Thread-Per-Request* pattern [SCH98], which creates a handler task for each incoming request and destroys it when the request is completed. Both patterns require the dynamic creation and destruction of tasks per connection or per request, respectively.

The *Threadpool* pattern is represented in Figure 3-13-C and includes all those strategies based on the definition of a pool of tasks in charge of processing incoming requests, such as:

- *Workers* [SCH98]. According to this pattern (see Figure 3-14-A), all tasks from the pool are equal and they monitor the I/O operations and process the incoming requests alternately.
- *Half-Sync / Half-Async* [SCH96][PYA01]. In this case, one specific task monitors the I/O operations and queues the incoming requests, while the tasks from the pool are responsible for processing them. This pattern is shown in Figure 3-14-B.
- *Leader & Followers* [SCH98][PYA01]. Under this pattern, middleware alternately selects a task from the pool to be the leader, thus becoming the

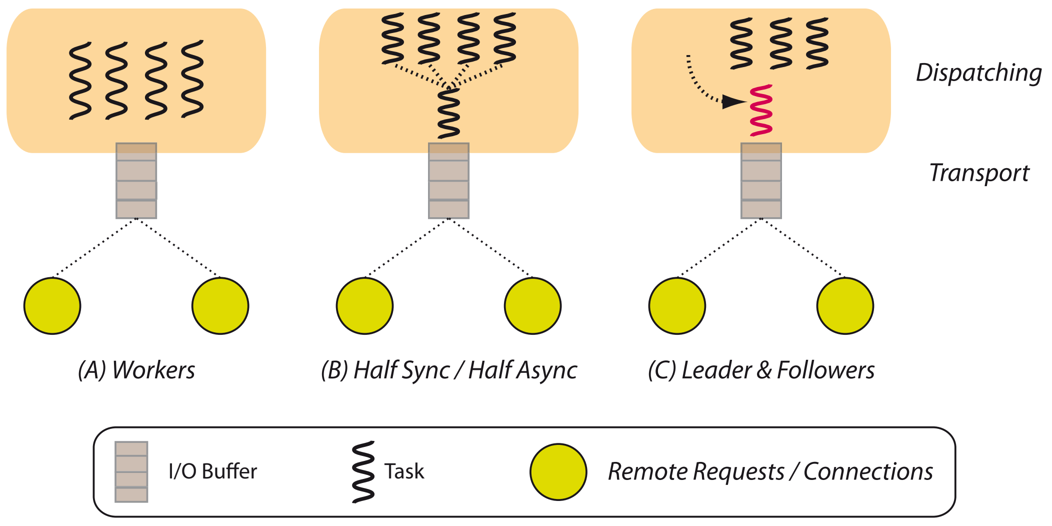

Figure 3-14: Concurrency patterns of type *ThreadPool*

task responsible for waiting, receiving and processing the next incoming request, as is shown in Figure 3-14-C.

None of the concurrency patterns described above can satisfy the requirements of avoiding unnecessary context switches and the dynamic update of scheduling parameters. Therefore, we should define a new concurrency pattern capable of satisfying both requirements and compatible with the *endpoints pattern*.

This new concurrency pattern for multi-tasking environments consists of a pool of statically created, dedicated tasks, that is, each task from the pool is pre-assigned the processing of a set of specific end-to-end flows, thus avoiding unnecessary context

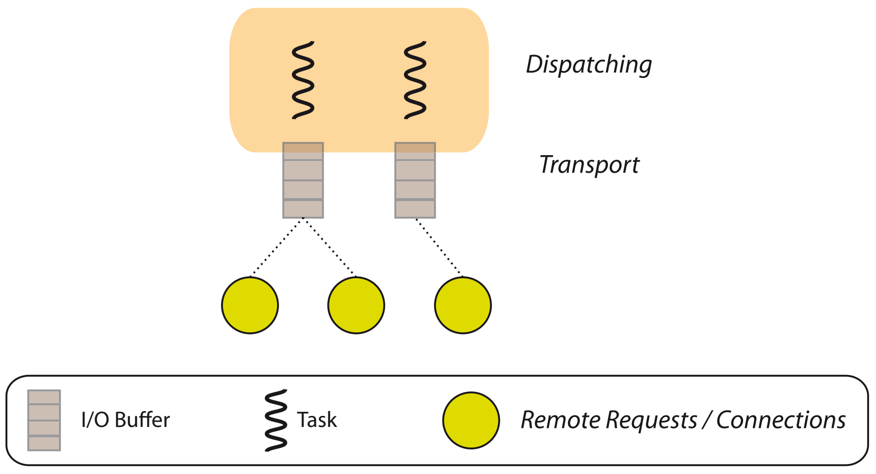

Figure 3-15: Concurrency pattern based on dedicated tasks

switches. In the context of this thesis, this concurrency pattern has been called *Ready To Go* (RATO). RATO supports the use of both static and dynamic scheduling parameters for handler tasks as this is a configurable option. For the former case, developers should define the same number of handler tasks as end-to-end flows that need to be processed with different scheduling parameters. For the latter case, a single handler task is required to process at least two end-to-end flows with different scheduling parameters. Figure 3-15 represents this scenario. In this case, the system is configured to instantiate two handler tasks, but one of them is responsible for handling the arrival of two separate end-to-end flows, and can process both end-to-end flows with the same or different scheduling parameters depending on the real-time configuration.

The scenario described in Figure 3-15, in which a single handler task can await the arrival of several end-to-end flows, has the advantage of being easily scalable. However, as a drawback, these end-to-end flows might be processed sequentially as they share the same handler task.

3.9 CONTRIBUTIONS OF THIS CHAPTER

This chapter has described the process of integrating an analyzable real-time model into middleware. Firstly, we studied the different distribution mechanisms defined in distribution standards (remote procedure call, distributed objects and data-centric) to validate their modelling and analysis by using the end-to-end flow model. Secondly, we defined the entities and mechanisms required to integrate this real-time model into distribution middleware, which resulted in the *endpoints pattern*, whose main characteristics are presented below:

- **Separation of concerns between the logic of the application and the real-time aspects.** Through the proposed API, the application code would only need to include a simple *Set_Event_Id* call to set the initial event that triggers the end-to-end flow. The rest of the end-to-end flow elements can be described as a part of a configuration operation which must be performed during the real-time configuration stage.
- **Support for heterogeneous scheduling policies and parameters.** The proposed interface presents a set of abstract primitives that should be extended for each scheduling policy used.
- **Control in the identification and / or assignment of scheduling parameters.** The proposed interface provides the required mechanisms to create and configure the schedulable entities (handler tasks and endpoints) included in the distributed system.

- **Infrastructure for CASE tools.** In this case, we performed an analysis to integrate the *endpoints pattern* with MAST, a schedulability analysis tool based on the end-to-end flow model. This integration would allow us to validate the proposed model and automatically generate the configuration code required for distributed real-time systems.

The *endpoints pattern* defines a set of interfaces within distribution middleware that allows processors and communication networks to be configured explicitly. In particular, these APIs allow the scheduling parameters of a distributed real-time application to be controlled. Although the configuration of a real-time application using the proposed approach could be seen as a tedious task for developers, it can be performed automatically by applying MDE techniques based on the end-to-end flow model defined in the MARTE standard. In Chapter 5, we will explain how the *endpoints pattern* can be used in a framework for the development of high-integrity distributed applications following an MDE strategy.

The *endpoints pattern* has been implemented for Ada by developing a set of interfaces at two different application levels: developer level, for the creation and configuration of the schedulable entities, and middleware level, for the internal management of the end-to-end flow and its dynamic adaptation. These APIs have been proposed for standardization within the Ada programming language as a solution for the lack of support for development of distributed real-time systems.

Finally, we have defined a new concurrency pattern, which is called RATO, to be used with the *endpoints pattern*. This pattern avoids unnecessary context switches and the dynamic update of scheduling parameters. According to this pattern, developers are responsible for explicitly defining which handler task and which scheduling parameters will be used for each end-to-end flow defined in the distributed system.

4

INTEGRATION AND VALIDATION OF THE REAL-TIME MODEL WITHIN DISTRIBUTION MIDDLEWARE

This chapter is organized as follows. First, in Section 4.1, the general context to integrate the endpoints pattern into a distributed real-time platform is introduced, as well as the basic features of the specific distribution middleware selected for this purpose. Section 4.2 introduces the distributed real-time platform and presents the modifications performed within middleware to optimise its execution in a real-time environment. An example of use is included in Section 4.3. Section 4.4 presents the validation of the proposed approach through two different case studies. Section 4.5 reviews the previous work done to support the end-to-end flow model in distributed Ada according to the DSA, and analyses in detail the differences compared with our approach. Finally, Section 4.6 summarizes the main contributions of this chapter.

4.1 INTRODUCTION

One of the major conclusions drawn from the analysis of distribution standards was the need to include new real-time mechanisms which (1) allow distributed applications to be analysed through schedulability analysis techniques and (2) facilitate the integration of middleware into new model-driven development processes. To fulfil the two conditions, the previous chapter defined the *endpoints pattern*, a set of mechanisms which allows the real-time requirements of a distributed application to be specified. Once these mechanisms have been introduced, this chapter focuses on the implementation and validation of the *endpoints pattern* in a distributed real-time platform.

In addition to integrating the *endpoint pattern* into distribution middleware, the design of this kind of platform requires the integration of other components which may affect the temporal behaviour of the system. In particular, the following components are considered:

- ***Communication middleware***, which is responsible for providing applications with predictable mechanisms when requesting remote services. This kind of mechanisms must also be aware of the underlying communication protocols.
- ***Communication networks***, which are responsible for ordering the nodes' accesses to the communication media and guaranteeing bounded latencies.
- ***Operating system***, which must be able to react to internal or external events in a bounded period of time, as well as to provide a set of basic services for a precise and deterministic execution of the system (e.g. scheduling or timing services).

Besides these components, selecting an appropriate programming language can facilitate the coding of the real-time parameters of applications. Furthermore, this choice is also a key element in terms of efficiency, reliability and maintenance of the system. Although the coding of real-time systems was initially performed by means of assembly language, nowadays the use of high-level programming languages is more and more usual, either sequential (e.g. C) or concurrent languages (such as real-time extensions of Java [BOL00] or Ada [ADA05]). As was commented in Chapter 1, Ada has been designed to develop real-time systems and already includes concurrent and real-time mechanisms in the language itself. Moreover, the endpoints API was implemented for Ada in the previous chapter. Therefore, support for this programming language would also be desirable in the distributed real-time platform.

4.1.1 CHOICE OF THE DISTRIBUTED REAL-TIME PLATFORM

One of the objectives of this thesis is to assist programmers in the development of distributed real-time systems coded in Ada, as was shown in the previous chapter where the endpoints APIs were implemented for Ada. Therefore, the platform selected should provide support for building distributed systems by means of Ada DSA.

From the two DSA implementations that were analysed in Chapter 2, Glade and PolyORB, the latter is the only one which is currently active and still evolving. Additionally, PolyORB addresses the interoperability among distribution models (such as RT-CORBA or DSA), which represents an interesting feature that can be used to validate the proposal with different distribution models while using the same software platform.

As was shown in the analysis included in Chapter 2, PolyORB is oriented to the development of real-time systems as it partially supports the RT-CORBA standard. Although the DSA annex does not provide support for building distributed real-time systems, the DSA personality included in this middleware follows a similar approach to RT-CORBA. However, the analysis concluded that PolyORB requires some extensions

Table 4-1: Classification of real-time communication networks

TECHNOLOGY	BIT RATE	DETERMINISTIC	COMPATIBLE WITH ETHERNET	COST
CAN	1MBPS	YES	INCOMPATIBLE	MIDDLE/LOW
FLEXRAY	20MBPS	YES	INCOMPATIBLE	MIDDLE
PROFIBUS	12MBPS	YES	INCOMPATIBLE	MIDDLE
PROFINET	100MBPS	YES	NON-INTEROPERABLE	MIDDLE
RTPS	10GBPS	NO	INTEROPERABLE	LOW
RTP	10GBPS	NO	INTEROPERABLE	LOW
CSMA/DCR	10GBPS	YES	INCOMPATIBLE	MIDDLE
RT-EP	10GBPS	YES	NON-INTEROPERABLE	LOW
FTT-ETHERNET	10GBPS	YES	NON-INTEROPERABLE	LOW
RTNET	10GBPS	YES	INTEROPERABLE	LOW
SWITCH AFDX	100MBPS	YES	INCOMPATIBLE	VERY HIGH
SWITCH 802.1P	10GBPS	YES[A]	INTEROPERABLE	MIDDLE/HIGH

a. with controlled workload

to optimise its predictability, such as the integration and scheduling of real-time communication networks.

Since the existing version of PolyORB only supports communications based on the IP protocol, one of our objectives will be provide this middleware with real-time communication networks. For this purpose, it is necessary to choose a set of parameters to evaluate the real-time capabilities of each communication network described in Chapter 1. Unfortunately, the large number of scenarios and cases of use makes this evaluation harder. Nevertheless, there are some surveys for specific technological sectors such as that developed by IEC 61784 standard [IEC07] for industrial control systems. Due to the objectives of the platform, the network or networks used should allow the execution of hard real-time applications. Moreover, it should also be based on a costless and accessible technology, such as for example Ethernet. In general, the processing nodes of a hard real-time system communicate through one or several local and controlled networks, and therefore it is not necessary to be interoperable with external systems, for example by using Internet. Table 4-1 summarizes these properties for most of the networks introduced in chapter 1. In particular, the following properties are considered:

- Bit rate or maximum transmission capacity supported by each network link, without considering the overhead introduced by protocols.

- Predictability of message transmission.
- Degree of compatibility with Ethernet standard classified according to:
  - Incompatible, if it does not use Ethernet technology or cannot be implemented without modifying Ethernet hardware/firmware.
  - Non-interoperable, if the mechanisms added to Ethernet cannot operate in the presence of network nodes that do not implement the alterations.
  - Interoperable, if it may coexist with standard Ethernet nodes. However, these solutions may not offer temporal guarantees when standard nodes are present.
- Technology implementation costs.

Of the networks listed in Table 4-1, the RT-EP, FTT-Ethernet and RTnet protocols fulfil the desirable properties of accessibility, predictability and cost. For these solutions not based on Ethernet, the CAN bus can be considered an interesting option as it is mature and accessible technology which is widely used in automotive and control systems. Lastly, it is also important to consider how these protocols are supported by the underlying operating system.

Unlike general-purpose operating systems, which focus on getting good average response times, these services offered by real-time operating systems must be executed with bounded maximum times. The only real-time operating system supported by PolyORB is VxWorks[1], a proprietary operating system for embedded applications whose high cost has ruled out its use in our platform. Therefore, PolyORB should be ported to another real-time operating system. As was the case for the development of distributed systems, it would be desirable that the operating system selected were based on standards, as they provide notable and stable solutions. Among the international standards, the *Portable Operating System Interface* (POSIX) standard is one of the most widely accepted by industry. Its main objective is to define a common interface which allows applications to be executed using different operating systems. This standard defines a set of services for different types of applications, including those related to real-time systems which are collected in a profile named POSIX.13 [POS98]. POSIX.13 has been successfully implemented both in proprietary systems, such as the aforementioned VxWorks or Integrity[2], and in open-source systems, such as for example MaRTE OS [ALD01].

MaRTE OS is a real-time kernel for embedded systems that follows the POSIX.13 profile. This operating system provides the POSIX interfaces for C and Ada programming languages, is open-source and supports hard real-time applications for

1. VxWorks is available at http://www.windriver.com/products/vxworks
2. Integrity is available at http://www.ghs.com/products/rtos/integrity.html

Table 4-2: Scheduling policies supported by MaRTE OS

STANDARD	SUPPORTED POLICIES
POSIX	SCHED_FIFO SCHED_RR SCHED_SPORADIC
ADA	NON_PREEMPTIVE_FIFO_WITHIN_PRIORITIES FIFO_WITHIN_PRIORITIES ROUND_ROBIN_WITHIN_PRIORITIES EDF_ACROSS_PRIORITIES

single processor and distributed systems, as it includes real-time communication protocols like RT-EP or CAN.

Summing up, the distributed real-time platform will consist of several nodes running over MaRTE OS. For the communication services, the distributed platform can use the RT-EP protocol or the CAN bus, which can meet hard real-time requirements and are supported by MaRTE OS. Finally, PolyORB provides us with RT-CORBA and DSA distribution models, two of the most notable standards in the development of distributed real-time systems.

4.1.2 FEATURES OF THE DISTRIBUTED REAL-TIME PLATFORM

After selecting the elements of the distributed real-time platform (middleware, operating system and communication networks), this section aims to review the main features of these elements and it briefly introduces the architecture, configuration options and services provided by each of them. The objective is to provide a general overview of each element, identifying the possible weaknesses that should be modified to meet hard real-time requirements.

4.1.2.1 MaRTE OS overview

As was previously commented in Section 4.1.1, MaRTE OS follows the POSIX.13 profile. This profile includes a minimal set of services to allow embedded applications to be developed efficiently. Among other features, it does not require either implementing a file system or supporting multiple processes. Therefore, the concurrency services (e.g. execution, scheduling or synchronization protocols) are only supported at task level. Besides the concurrency model defined by the POSIX standard, this operating system also provides support for the concurrency model defined by Ada.

The MaRTE OS provides support for the scheduling policies defined in the POSIX and Ada standards, which have been listed in Table 4-2.. Moreover, it also provides a scheduling service, called *Application-Defined Scheduling*, for applications to define their own scheduling algorithms. This service is used, for example, to implement flexible scheduling techniques such as FSF [ALD06] or FRESCOR [FRSH11] but without modifying the existing scheduling service included in MaRTE OS. Therefore, the scheduling architecture defined by the MaRTE OS will allow the *endpoints pattern* to be implemented for different scheduling policies, which represents one of the desirable features for distribution standards and/or their implementations as commented in Chapter 2.

In order to build applications, MaRTE OS uses its own compilation process based on GNAT (i.e. the Ada compiler developed by AdaCore[1]) and provides a compilation script, which is called *mgnatmake,* responsible for compiling and linking with the appropriate kernel's libraries.

Regarding communication networks, the MaRTE OS includes support for the RT-EP protocol and the CAN bus. For the former, MaRTE OS implements the protocol's algorithm and also provides the drivers for several Ethernet cards (sis900, eepro100 and rtl8139). For the latter, the kernel only includes the driver for a single card (Adlink PCI 7841). Finally, the existing version of MaRTE OS does not include any support for networks based on the IP protocol.

4.1.2.2 RT-EP and CAN overview

RT-EP (Real-Time Ethernet Protocol) is a software-based token-passing protocol over a logical ring. This protocol is built upon the Ethernet standard and can handle network messages with MTU (*Maximum Transmission Unit*) up to 1492 bytes in a single packet.

The software architecture of this protocol consists of several reception channels, one per task that may communicate through the network, and only one transmission channel. These channels are implemented as priority queues, and packets with the same priority are stored in FIFO order. There is also one task, the *Main Communications Task,* which is responsible for configuring and managing the logical ring. Furthermore, this task is also in charge of processing I/O events, that is, sending the data stored in the transmission queue and reading the incoming packets from the network to store them into the reception queues.

CAN is a serial communication bus scheduled through fixed priorities, and whose MTU is only 8 bytes. In this case, MaRTE OS only provides the driver for a single card and does not implement any high-level protocol over the CAN bus.

However, none of the available drivers in the MaRTE OS provides message fragmentation services to enable sending larger messages than their MTU (1492 bytes for RT-EP and 8 bytes for the CAN bus). Although 1492 bytes may be enough for some

1. Ada-Core Technologies, The GNAT Pro Company. http://www.gnat.com/

kind of applications, 8 bytes are clearly inadequate for most of the communications used by our platform, such as the CORBA standard.

4.1.2.3 FRESCOR overview

The FRESCOR (*Framework for Real-time Embedded Systems based on COntRacts*) EU project [FRSH11] has the objective of providing engineers with a scheduling framework that represents a high-level abstraction that lets them concentrate on the specification of the application requirements, while the system transparently uses advanced real-time scheduling techniques to meet those requirements. In order to keep the framework independent of specific scheduling schemes, FRESCOR introduces an interface between the applications and the scheduler, called the service *contract*. Application requirements related to a given resource are mapped to a contract, which can be verified at design time by providing off-line guarantees, or can be negotiated at runtime, when it may or may not be admitted. As a result of the negotiation a virtual resource is created, representing a certain resource reservation. The resources managed by the framework are the processors, networks, memory, shared resources, disk bandwidth, and energy; additional resources could be added in the future.

FRESCOR provides support for network contracts through a uniform interface called FNA (*FRSH Network Adaptation layer*), which has been designed to allow network modules to be easily plugged in for the same or different networks. In order to be integrated into the framework, each network must implement this FNA layer. The existing framework provides support for the RT-EP and CAN real-time networks, but their scheduling is based on contracts instead of simple priorities. These are called FRSH-RTEP [FRSH09-A] and FRSH-CAN [FRSH09-B], respectively.

In the FRESCOR framework, support for the *end-to-end flow* model is being built. A tool called the *Distributed Transaction Manager* (DTM) [FRSH09-C] is a distributed application responsible for the negotiation of end-to-end flows in the local and remote processing nodes in a FRESCOR system that implements the contract-scheduling framework. Managing distributed end-to-end flows cannot be done on an individual processing node because it requires dynamic knowledge of the contracts negotiated in the other nodes, leading to a distributed consensus problem. The objective of the Distributed Transaction Manager is to enable the remote management of contracts in distributed systems, including capabilities for remote negotiation and renegotiation, and management of the coherence of the results of these negotiation processes. In this way, FRESCOR provides support for distributed global activities or end-to-end flows consisting of multiple actions executed in processing nodes and synchronized through messages sent across communication networks.

The current implementation of the DTM limits its capabilities to the management of remote contracts, and it is implemented directly over the network services provided by FNA.

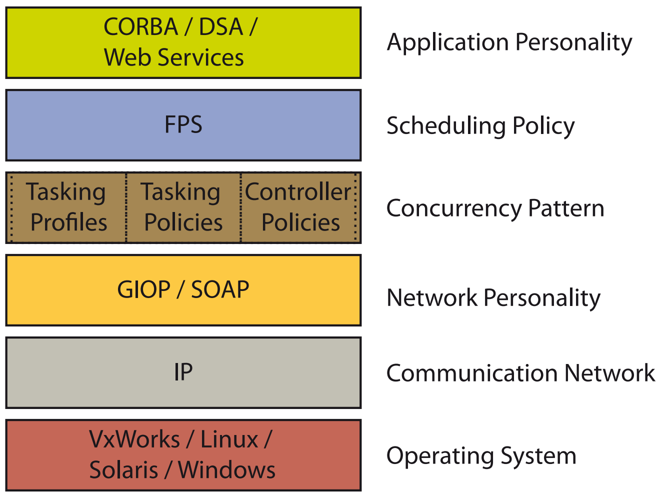

Figure 4-1: General architecture for PolyORB

4.1.2.4 PolyORB overview

As was shown in Chapter 2, PolyORB is a middleware oriented to building distributed real-time systems. However, it requires some extensions to optimise its determinism. Figure 4-1 shows the general architecture for the distributed real-time platform provided by PolyORB and summarizes the configuration options according to six categories: *Application personalities*, *Scheduling policies*, *Concurrency patterns*, *Protocol personalities*, *Communication networks* and *Operating systems*. In the following, this general scheme will be used to identify the modifications included in the proposed distributed real-time platform. Some of the modifications required at the structural level have already been introduced (for example, support for a new real-time operating system, incorporation of scheduling parameters in the communications and integration of new real-time networks); however, it is also necessary to identify other features that may affect the predictability of the approach. In particular, we will focus on the mechanisms available to manage and process remote calls.

Figure 4-2 illustrates how I/O events are managed in PolyORB; this figure represents the three layers defined by PolyORB (application, neutral and protocol layers) and focuses on the *Dispatching* (neutral layer) and *Transport* (protocol layer) services. As is shown in this figure, incoming network events (such as a connection request or the reception of a message) are represented in PolyORB as an object called *Event_Source*, which is a high-level abstraction of the entity responsible for managing the communications for each network (e.g. a socket for IP networks). Therefore, each network has a different type of *Event_Source*. If several *Event_Source* objects share the same type, they are grouped in another entity called *Monitor* (see Figure 4-2). Thanks to this entity, middleware can simultaneously monitor several *Event_Source* with a single

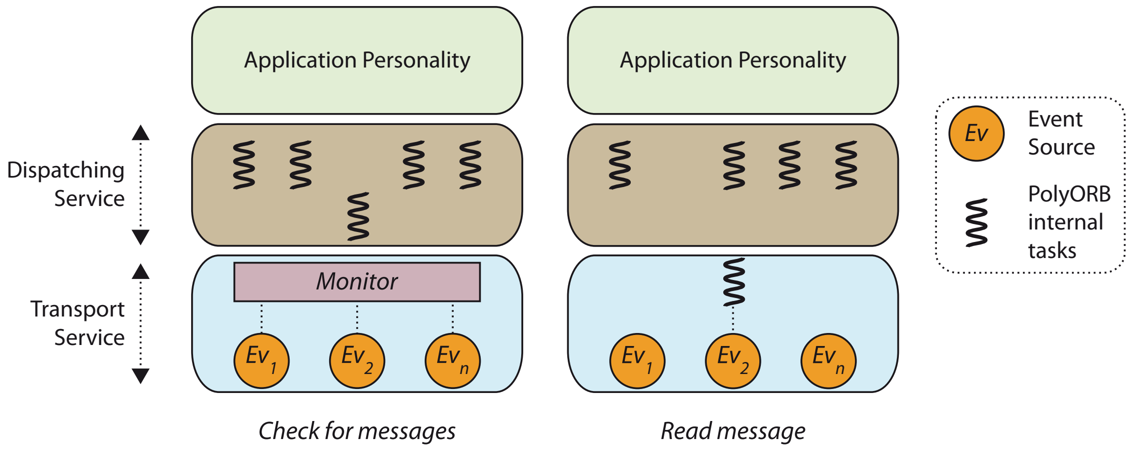

Figure 4-2: I/O request processing in PolyORB

task (for example, by using the *select* call defined for sockets). In PolyORB, the processing of remote calls is divided into two stages as illustrated in Figure 4-2: "*Check for messages*" and "*Read message*". In the first stage, the task designated to monitor I/O events will check the state of each *Event_Source* associated with the same *Monitor.* When a new message has arrived at the system, the second stage starts and middleware will select one or several tasks according to the concurrency pattern to read and process the received message.

Incoming messages are processed depending on the concurrency pattern used. As was described in Chapter 2, PolyORB defines a set of concurrency patterns which control the interaction among the personalities and the distribution micro-kernel. These patterns include (1) tasking policies (*No Tasking, Thread Per Request, Thread Per Session* or *Thread Pool*), (2) ORB controller policies (*No Tasking, Workers, Half Sync/Half Async* or *Leader/Followers*) and (3) tasking profiles (*No Tasking, Full Tasking* or *Ravenscar*). Although these patterns allow the application's concurrent behaviour to be configured, they do not consider the assignment of scheduling parameters for each task. Furthermore, despite PolyORB implementing the RT-CORBA specification and providing support for the management of priorities within the *threadpool*, this management is performed at the application personality level. Therefore, the processing of remote requests within the micro-kernel and protocol personality levels is performed by internal middleware tasks. This is illustrated in Figure 4-3, which shows how this middleware may suffer from the priority inversion problem when any of the predefined concurrency patterns is used. The same problem applies when using the DSA application personality since it also follows the scheduling model proposed by RT-CORBA. In fact, this personality shares several aspects with RT-CORBA, such as the use of GIOP as the communication protocol or the need of a Naming Server [NAM04] to register and locate remote objects.

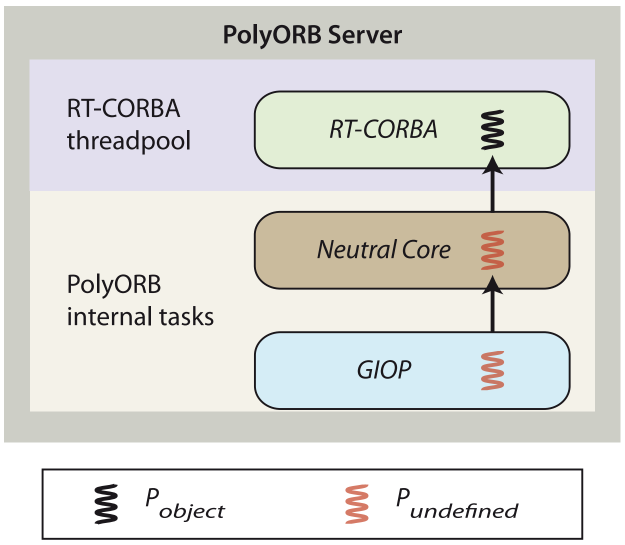

Figure 4-3: RT-CORBA and PolyORB tasking model

However, the compilation process is different for both personalities. While RT-CORBA uses an IDL compiler to statically generate stubs from IDL files and then builds the application with a generic Ada compiler, DSA uses its own compiler, which is called *po_gnatdist*, to generate stubs and build the application. Therefore, the configuration options available for DSA depend on the *po_gnatdist* compiler instead of PolyORB middleware (for example, a PolyORB-DSA application will always set *Workers* as the ORB controller policy).

4.2 THE DISTRIBUTED REAL-TIME PLATFORM AND ITS EXTENSIONS

4.2.1 MODIFICATIONS APPLIED TO THE PLATFORM

One of the major advantages of implementing the *endpoints pattern* relies on separating the logic of the application from the real-time aspects. This can be seen in the implementation performed in PolyORB, where the creation of schedulable entities (*handler tasks* and *endpoints)* and the assignment of their scheduling parameters are carried out by means of the proposed APIs, while events internal to end-to-end flows are handled automatically by middleware.

As was described earlier in Chapter 3, the incorporation of the *end-to-end flow model* and the *endpoints pattern* into distribution middleware requires the definition of a parameter, which is called *Event_Id*, to identify the end-to-end flow being executed. This

Table 4-3: Modifications performed in PolyORB to transmit the Event_id parameter

FILES	PURPOSE
POLYORB-QOS-EVENT_PARAMETERS.ADS POLYORB-QOS-EVENT_PARAMETERS.ADB	MARSHALLING / UNMARSHALLING SUBPRIOGRAMS FOR EVENT_ID PARAMETER
POLYORB-QOS-SERVICE_CONTEXTS.ADS	TAG DEFINITION FOR SERVICE CONTEXT
POLYORB-QOS.ADS	SUPPORTED QOS PARAMETERS

parameter is sent through the network and the appropriate scheduling parameters will be used depending on it. In PolyORB, the *Event_Id* is transmitted as part of the GIOP message in a data field called *Service_Context* [COR03]. This data field allows middleware to specify service-specific context information for requests and replies, such as the priority or the encoding used, as this information is passed implicitly with each message. Unlike other solutions such as appending the *Event_Id* to a raw network message, the use of the *Service_Context* data field is still compatible with the CORBA standard and so it maintains the interoperability among different ORBs. Nevertheless, this solution introduces some overhead because messages must be partially processed before getting the *Event_Id* and updating the scheduling parameters whenever it is required. The amount of overhead is not fixed and depends on each implementation since the standard does not define any specific processing order when several *Service_Contexts* are used [COR03]. Table 4-3 summarizes the new files and modifications performed in PolyORB to implement the transmission of the *Event_Id* parameter.

As was commented in Section 4.1.2.4, the management of priorities included in PolyORB follows the same approach as the RT-CORBA specification. In this case, the implementation of the *endpoints pattern* has a set of advantages over the RT-CORBA approach, such as the use of interchangeable scheduling policies or the free assignment of scheduling parameters.

The endpoints API has been developed and completed in PolyORB for three scheduling policies: FPS, EDF and flexible scheduling. Abstract interfaces have been extended for each policy and therefore they use their corresponding scheduling parameters, that is, priorities, deadlines and FRESCOR contracts, respectively. The former two scheduling policies are fully supported by Ada so their integration was straightforward. However, the flexible scheduling policy requires (1) the creation of the Ada *bindings* to the FRESCOR framework (see Annex B) and (2) integrating the creation and negotiation of contracts for internal middleware tasks into distribution middleware.

Furthermore, the mechanisms included in the original version of PolyORB are insufficient to optimise the timing behaviour of distributed applications, as is clear from the analysis in Chapter 2, which has led to the incorporation of a number of improvements and extensions aimed at providing greater predictability.

As we said earlier in Chapter 2 and Section 4.1.2.4, PolyORB requires some extensions to optimise the real-time behaviour of distributed applications. This has motivated the incorporation of a set of enhancements which are described below:

- Support for a new real-time operating system
- New ORB controller and tasking policies that avoid unnecessary context switches and the dynamic update of scheduling parameters.
- Definition of a new protocol personality for PolyORB to include appropriate communication networks for hard real-time systems
- New tasking profiles adapted to the requirements set by the *endpoints pattern* and the flexible scheduling policy.

4.2.1.1 Support for a new real-time operating system

PolyORB has been ported to MaRTE OS which provides support for the architectures based on fixed priorities, EDF and contracts that have been implemented in middleware. The following changes were applied:

- Removal of packages whose subprograms are not supported by MaRTE OS, such as those packages related to sockets or requiring a persistent file system.
- Fixing the compilation process by adding a set of files to *libmgnat,* a general-purpose library included in MaRTE OS.
- Development of a new two-stage compilation process: firstly, these development tools and services included in PolyORB are compiled using the Ada compiler (GNAT), as they must be executed in the host computer in order to enable the generation of stubs or the configuration of PolyORB parameters; secondly, real-time applications are built by compiling and linking source code and PolyORB libraries by means of the aforementioned *mgnatmake* script.
- Adaptation of the *po_gnatdist* compiler to the MaRTE OS environment.

4.2.1.2 New ORB controller and tasking policies

One of the conclusions drawn from the analysis of distribution middleware for real-time systems was the direct connection between concurrency patterns and the unbounded priority inversion problem. Regardless of the concurrency pattern used, the

Table 4-4: Modifications performed in PolyORB to create new tasking and orb_controller policies

FILES	PURPOSE
POLYORB-ORB_CONTROLLER-READY_TO_GO.ADS POLYORB-ORB_CONTROLLER-READY_TO_GO.ADB	NEW ORB_CONTROLLER TASKING POLICY
POLYORB-ORB-THREAD_PER_TARGET.ADS POLYORB-ORB-THREAD_PER_TARGET.ADB	NEW ORB TASKING POLICY

determinism of applications can only be guaranteed through an absolute control of the scheduling parameters associated not only with application tasks but also with internal middleware tasks.

Returning to the discussion dealt with in Chapter 3, hard real-time systems usually select a concurrency pattern capable of avoiding unnecessary context switches and the dynamic update of scheduling parameters. This may also be applicable to these environments where the cost of the dynamic modification of the scheduling parameters can be computationally high, as is the case of the flexible scheduling policy based on contracts [ALD06] [FRSH11]. Since none of the concurrency patterns defined in PolyORB can satisfy these requirements, a new concurrency pattern must be integrated into middleware. In addition to satisfying both requirements, this new pattern should be implemented together with the *endpoints pattern*.

The RATO concurrency pattern was introduced in Chapter 3 and illustrated in Figure 3-15. This pattern consisted of a pool of statically created, dedicated tasks. According to the nomenclature included in PolyORB, RATO can be considered as a new *ORB Controller Policy*. Specifically, it manages the distribution kernel’s main loop, the I/O events and the processing of requests by means of dedicated tasks that wait directly on the network.

PolyORB also defines the *Tasking Policies* to control the creation/removal of internal middleware tasks. To this end, a new tasking policy named *Thread Per Target* (TPT) has been defined as a complementary policy to RATO. This policy provides mechanisms to create tasks explicitly with the appropriate scheduling information. As a result, this policy can guarantee that tasks intended to process incoming remote calls are usually created in the *real-time configuration stage* (see Section 3.4.4 in Chapter 3). Table 4-4 summarizes the new files and modifications performed in PolyORB to implement the new ORB controller and tasking policies.

4.2.1.3 New protocol personality

Protocol personalities handle the mapping of network events onto middleware data structures. In our case, the RT-EP protocol and the FNA FRESCOR communication layer have been adapted to PolyORB in order to add network messages as new

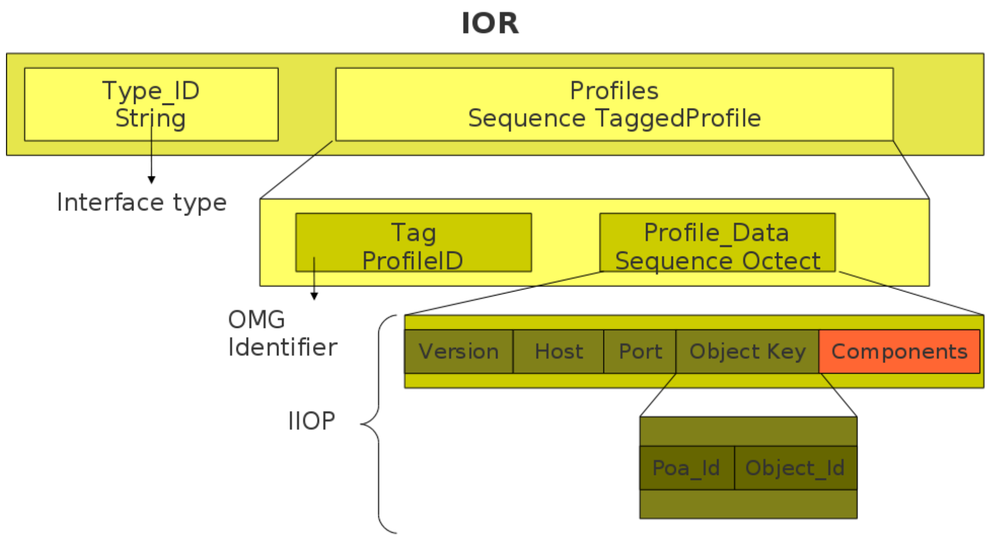

Figure 4-4: ***Interoperable Object Reference structure***

schedulable entities. These network protocols have been implemented within the GIOP communication layer. Not only does this layer maintain the interoperability among different ORBs which implement any of these protocols, but it also provides a message fragmentation layer at the transport level. In our case, this fragmentation service is particularly important as the network services included in MaRTE OS do not provide this functionality.

This deployment has required the use of two RT-CORBA standard elements: the IOR and the *Service_Context*. The former is a data structure that stores the required information to locate remote objects. As can be seen in Figure 4-4, the IOR consists of a list of supported profiles, each of them with a specific identifier associated. In our case, a new profile has been created for each protocol (i.e. RT-EP and FNA) which includes the version of the protocol, the location of the remote object encoded as the station and channel parameters, an object identifier to locate it within the remote node (*Object Key*) and an extendable data field called *Components*. This data field contains additional information supporting optional protocol features, such as the use of encrypted communications or priorities depending on the scheme defined for each protocol (e.g. RT-EP or CAN).

Once the processing nodes know the location of the remote objects and how to contact them through the communication network, it is necessary to have mechanisms to specify the execution priority for the requested services and the priority of the reply messages in the case of RPCs. As was stated at the beginning of Section 4.2.1, the transmission of the *Event_Id* parameter within the *Service_Context* data field allows contextual information to be included with each request and response. This data structure is transmitted with each request using a standard GIOP message, as illustrated in Figure 4-5.

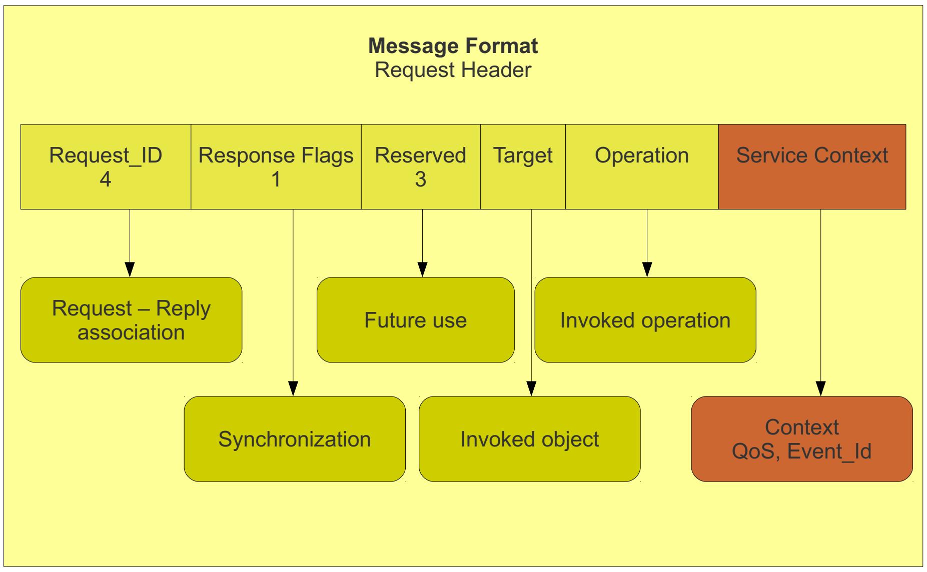

Figure 4-5: Message format for GIOP requests

After setting the scheduling parameters for both network messages and remote objects, the next issue to consider is the management of these parameters by middleware. As was discussed in Section 4.1.2.4, PolyORB defines a set of policies for managing concurrent remote calls that are suitable for the priority inversion problem, both in the management of the I/O events and in the processing of messages within the ORB (see Figure 4-3). To solve this problem, a new software layer has been added within the protocol personality level. This layer takes advantage of the features provided by RATO and TPT policies in the management of network messages and the controlled use of task and message scheduling parameters.

This software layer is common for both networks added (RT-EP and FNA) but it provides specific operations for sending or receiving data, as well as for the specific type of scheduling parameters handled by each of the protocols. Unlike the original version of PolyORB, this modified version is designed to wait directly on the network (i.e. the stage *Check for messages*, which was shown in Figure 4-2, was removed) and assign scheduling parameters based on the value of the *Event_Id* parameter. Consequently, this operation scheme based on the *Event_Id* does not use the RT-CORBA policies since real-time requirements are managed independently at end-to-end flow level through the mechanisms defined by the *endpoints pattern*.

Finally, Table 4-5 summarizes the most notable differences among the three protocol personalities: IIOP, RT-EP and FNA. While the former was originally included in PolyORB and is intended for IP-based networks, the latter two correspond to real-time networks and have been developed as part of this thesis.

Table 4-5: Differences among the new protocol personalities

	IIOP	RT-EP	FNA
SUPPORTED PROTOCOLS	TCP-IP	RT-EP	FRSH-RTEP / FRSH-CAN
SCHEDULING PARAMETERS	SENT THROUGH THE NETWORK	STATIC CONFIGURATION	STATIC CONFIGURATION
TASKING POLICIES	THREAD-PER-SESSION / THREAD-PER-REQUEST / THREADPOOL	TPT	TPT
ORB CONTROLLER POLICIES	WORKERS /HS-HA / LF	RATO	RATO
I/O MONITORING	SINGLE TASK[a]	TASKS ON DEMAND	TASKS ON DEMAND
UNFORCED CONTEXT SWITCHES FOR HIGHEST PRIORITY TASK	ACCORDING TO POLICY	NONE	NONE

a. *Select* call will be used to monitor multiple receive endpoints

4.2.1.4 New tasking profiles

As was introduced in Chapter 2, PolyORB defines a series of tasking profiles (*No Tasking, Full Tasking, Ravenscar*) to establish the restrictions that system tasks must fulfil. In the case of the *endpoints pattern*, it is necessary to define (1) a way to establish the scheduling parameters upon creating tasks and (2) a data structure that associates each task with the specific *Event_Id* that is currently being processed, as well as with other characteristic parameters (for example, a *receive endpoint* is associated with a specific *handler task*). For our purposes, a data structure called *Notes*, which is already defined in PolyORB, will be used. This data structure allows contextual information to be associated with certain entities, such as for example tasks. For this particular case, *Notes* is implemented as a tasks Ada attribute [ADA05]. This behaviour is collected by means of a new tasking profile called *Full Tasking Endpoints*.

In addition to the conditions set by the *endpoints pattern*, the use of flexible scheduling policies (i.e. the FRESCOR framework) adds several restrictions in the creation / deletion of tasks that should be included in a new profile. Specifically, each task executed within the framework must satisfy two conditions: firstly, it must be associated with a contract that authorizes the use of system resources; secondly, it must be restricted to possible changes in its scheduling parameters (e.g. priority) through operations outside the framework. Therefore, a new profile has been added to PolyORB. This new profile is called *Full Tasking Frsh* and it enables the creation of tasks associated with a previously negotiated contract.

Due to the special features of tasks being executed within the FRESCOR framework, specific support for the creation and integration of tasks into the framework through middleware has been also included. To this end, two new subprograms have been added to PolyORB, which are detailed in Listing 4-1. The first subprogram, which

Listing 4-1: Processing_Node_Scheduling API extensions for the FRESCOR framework

```
procedure Create_FRSH_Thread
    (Params          : Frsh_Task_Scheduling_Parameters_Ref;
     New_Thread_Code : Frsh_Ada_Types.FRSH_Thread_Code);

procedure Negotiate_Contract_For_External_Thread
    (Params      : Frsh_Task_Scheduling_Parameters_Ref);
```

is called *Create_FRSH_Thread*, enables the creation of application tasks within the framework. This subprogram takes the scheduling parameters and the code to be executed by the new task (i.e. *New_Thread_Code* in Listing 4-1) as input parameters.

As Ada supports the creation of tasks by means of the language itself, it may be of interest that middleware can provide a second subprogram to integrate these tasks into the FRESCOR framework. To this end, the *Processing_Node_Scheduling* API for FRESCOR contracts adds *Negotiate_Contract_For_External_Thread* as a new subprogram, whose specification is also defined in Listing 4-1. By invoking this subprogram, tasks will be integrated into the FRESCOR framework after negotiating a new contract with the input scheduling parameters.

Finally, the new files and modifications performed in PolyORB to implement our two new tasking profiles (*Full Tasking Endpoints* and *Full Tasking Frsh*) are summarized in Table 4-6.

Table 4-6: Modifications performed in PolyORB to create new tasking profiles

FILES	PURPOSE
POLYORB-SETUP-TASKING-FULL_TASKING_ENDPOINTS.ADB POLYORB-SETUP-TASKING-FULL_TASKING_ENDPOINTS.ADS	SETUP PACKAGE FOR FULL TASKING ENDPOINTS PROFILE
POLYORB-TASKING-PROFILES-FULL_TASKING-ENDPOINTS_THREADS.ADS POLYORB-TASKING-PROFILES-FULL_TASKING-ENDPOINTS_THREADS.ADB	NEW TASKING PROFILE TO ASSOCIATE SHCEDULING PARAMETERS AND ENDPOINTS WITH HANDLER TASKS
POLYORB-TASKING-PROFILES-FULL_TASKING-ENDPOINTS_THREADS-ANNOTATIONS.ADS POLYORB-TASKING-PROFILES-FULL_TASKING-ENDPOINTS_THREADS-ANNOTATIONS.ADB	DATA STRUCTURES ASSOCIATED TO TASKS
POLYORB-SETUP-TASKING-FULL_TASKING_FRSH.ADB POLYORB-SETUP-TASKING-FULL_TASKING_FRSH.ADS	SETUP PACKAGE FOR FULL TASKING FRSH PROFILE
POLYORB-TASKING-PROFILES-FULL_TASKING-FRSH_THREADS.ADS POLYORB-TASKING-PROFILES-FULL_TASKING-FRSH_THREADS.ADB	NEW TASKING PROFILE TO INTEGRATE HANDLER TASKS INTO FRESCOR FRAMEWORK
POLYORB-TASKING-PROFILES-FULL_TASKING-FRSH_THREADS-ANNOTATIONS.ADS POLYORB-TASKING-PROFILES-FULL_TASKING-FRSH_THREADS-ANNOTATIONS.ADB	DATA STRUCTURES ASSOCIATED TO TASKS

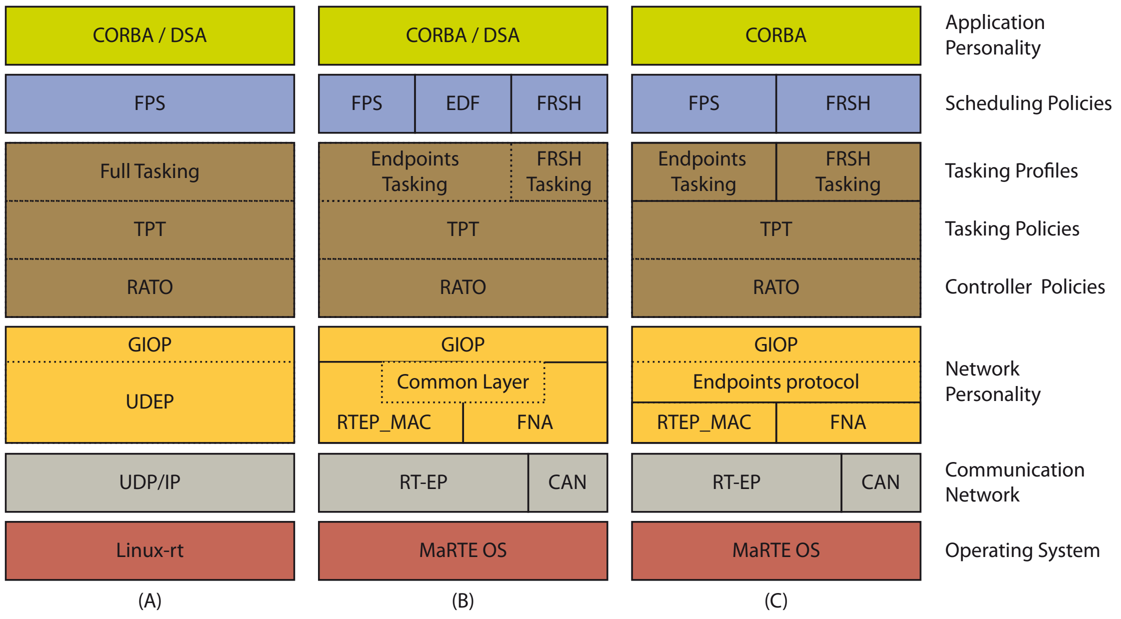

Figure 4-6: General architecture for the distributed real-time platform

4.2.2 DISTRIBUTED REAL-TIME PLATFORM IMPLEMENTATIONS

The work included in this thesis has been in evolution until the final proposal for the *endpoints pattern* was made. This has led to different implementations in which models and functionalities have been validated. These implementations are illustrated in Figure 4-6. Although the main objective is to validate the use of the *endpoints pattern* within distribution middleware, each implementation provides a set of different features which are described below.

Figure 4-6-A shows the implementation performed on a Linux operating system with the *CONFIG_PREEMPT_RT*[1] patch. Although this operating system is not able to satisfy hard real-time requirements, it facilitates the use of debugging tools and so it has also facilitated the implementation of RATO and TPT policies. Therefore, it can be considered as a first step in the development of our distributed real-time platform. Furthermore, a new protocol personality for IP networks has been developed. Unlike the IP-based protocol personalities already included in PolyORB, this new personality supports the use of the *endpoints pattern* and includes the use of priorities in communication networks through the *Diffserv* data field [RFC2474] (that is, a solution similar to that proposed by TAO).

1. CONFIG_PREEMPT_RT patch is available at https://rt.wiki.kernel.org/index.html

Table 4-7: Files required to develop a new protocol personality in PolyORB

FILES	PURPOSE
POLYORB-BINDING_DATA.ADS	NEW TAGS FOR THE PROTOCOL PERSONALITY
POLYORB-BUFFERS.ADB POLYORB-BUFFERS.ADS	NEW FUNCTIONS TO ADAPT RECEIVING MESSAGES TO MIDDLEWARE FORMAT
POLYORB-TRANSPORT-ENDPOINTS.ADB POLYORB-TRANSPORT-ENDPOINTS.ADS	BASIC STRUCTURES ASSOCIATED TO ENDPOINTS
POLYORB-TRANSPORT-ENDPOINTS-COMMON_LAYER.ADB POLYORB-TRANSPORT-ENDPOINTS-COMMON_LAYER.ADS	COMMON LAYER ASSOCIATE DTO PROTOCOLS BASED ON ENDPOINTS
POLYORB-BINDING_DATA-GIOP-ENDPOINTS.ADB POLYORB-BINDING_DATA-GIOP-ENDPOINTS.ADS	COMMON UTILITIES FOR GIOP INSTANCES THAT RELY ON ENDPOINTS ADDRESSES
POLYORB-BINDING_DATA-GIOP-EP.ADB POLYORB-BINDING_DATA-GIOP-EP.ADS	NEW PROFILE FOR THE PROTOCOL PERSONALITY
POLYORB-GIOP_P-TRANSPORT_MECHANISMS-EP.ADB POLYORB-GIOP_P-TRANSPORT_MECHANISMS-EP.ADS	NEW TRANSPORT MECHANISMS FOR THE PROTOCOL PERSONALITY
POLYORB-PROTOCOLS-GIOP-EP.ADB POLYORB-PROTOCOLS-GIOP-EP.ADS	FILES REQUIRED TO CREATE A NEW GIOP INSTANCE FOR THE PROTOCOL PERSONALITY

Figure 4-6-B shows the first implementation performed on the distributed real-time platform. It uses MaRTE OS as the hard real-time kernel and CAN and RT-EP as the communication networks. The deployed version integrates processor and network scheduling in a single Ada package and so cannot use different scheduling policies for both resources. This restriction limits the implementation of mixed systems scheduled by fixed priorities and, for example, TDMA communications. However, this implementation allows the *endpoints pattern* to be validated in different distribution models such as RT-CORBA and DSA, and under different scheduling policies such as FPS, flexible scheduling and EDF. Currently, this implementation defines an API corresponding to an earlier version of the *endpoints pattern* described in Chapter 3.

This preliminary version of the *endpoints pattern* does not include the event transformation service either, and each end-to-end flow is identified by means of a single *Event_Id*. This shortcoming limits the system whenever a remote object is invoked several times during the same end-to-end flow but using different scheduling parameters. Under this scenario, the use of a different *Event_Id* per remote call is required.

Figure 4-6-C illustrates the implementation corresponding to the final proposal of the *endpoints pattern* included in this thesis. In this case, given that this work is not included in the main development line of PolyORB, it was decided to rewrite the API and the proposed extensions completely in order to facilitate the maintenance of the proposal in future versions of middleware. Nevertheless, this API is currently implemented only for the RT-CORBA distribution model and the fixed-priority and

Table 4-8: RT-EP and FNA adaptation files to the protocol personality

FILES	PURPOSE
POLYORB-TRANSPORT-ENDPOINTS-COMMON_LAYER-RTEP_MAC.ADB POLYORB-TRANSPORT-ENDPOINTS-COMMON_LAYER-RTEP_MAC.ADS	WRITE / READ OPERATIONS FOR RT-EP PROTOCOL
POLYORB-SETUP-ACCESS_POINTS-RTEP_MAC.ADB POLYORB-SETUP-ACCESS_POINTS-RTEP_MAC.ADS	SETUP OPERATIONS FOR RT-EPPROTOCOL
POLYORB-SETUP-RTEP_MAC.ADB POLYORB-SETUP-RTEP_MAC.ADS	GIOP VERSIONS ENABLED AND INITIALIZATION OPERATIONS FOR RT-EP PROTOCOL
POLYORB-TRANSPORT-ENDPOINTS-COMMON_LAYER-FNA.ADB POLYORB-TRANSPORT-ENDPOINTS-COMMON_LAYER-FNA.ADS	WRITE / READ OPERATIONS FOR FNAPROTOCOL
POLYORB-SETUP-ACCESS_POINTS-FNA.ADB POLYORB-SETUP-ACCESS_POINTS-FNA.ADS	SETUP OPERATIONS FOR FNA PROTOCOL
POLYORB-SETUP-FNA.ADB POLYORB-SETUP-FNA.ADS	GIOP VERSIONS ENABLED AND INITIALIZATION OPERATIONS FOR FNA PROTOCOL

FRESCOR contracts scheduling policies, since our main objective is to verify the functionality of the new features added to the *endpoints pattern*.

Unlike the other implementations of the *endpoints pattern*, this development includes a common software layer shared by each real-time protocol and which has been integrated as a new protocol personality (see Figure 4-6-C). This personality, which is called the *Endpoints protocol personality,* is responsible for the management and processing of I/O events of any real-time protocol based on the *endpoints pattern*. Table 4-7 summarizes the files required to implement the *Endpoints protocol* personality, and Table 4-8 shows the adaptation files for each specific network protocol used (i.e. the RT-EP protocol and the FRESCOR FNA communication layer).

4.3 EXAMPLE OF USAGE

The objective of this example is to describe the usage of the *endpoints pattern* proposed in Chapter 3 and to introduce a simple application which executes over our distributed real-time platform. In this case, the application consists of a linear and synchronous end-to-end flow which executes remote calls through two processors and one communication network. The application is implemented over the distributed real-time platform defined in Figure 4-6-C, which implements all the features described by the *endpoints pattern*.

Figure 4-7 shows the Ada packages used in this example. These packages can be classified into three different groups of files: (1) the PolyORB configuration files, (2) the end-to-end flow or real-time configuration files and (3) the application code.

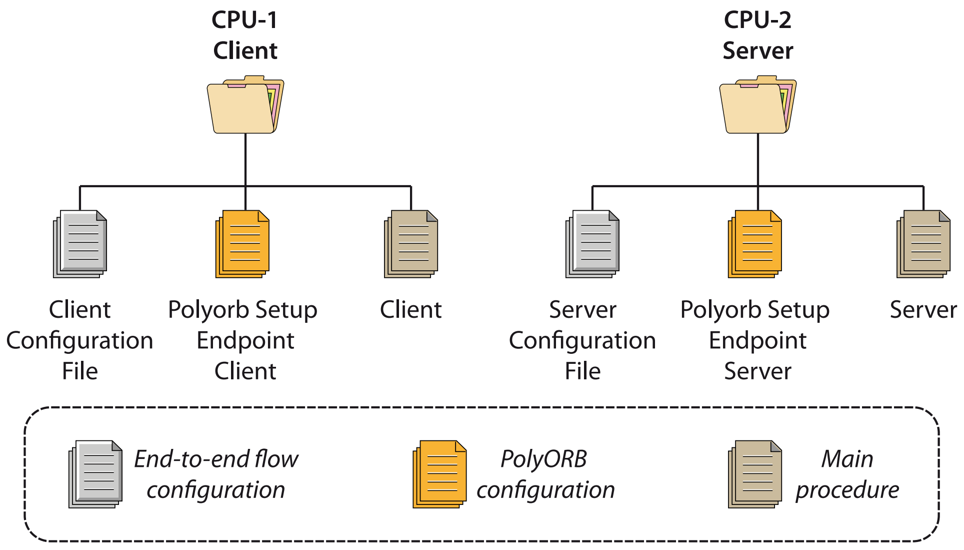
CPU-1
Client
CPU-2
Server
Client Configuration File
Polyorb Setup Endpoint Client
Client
Server Configuration File
Polyorb Setup Endpoint Server
Server
End-to-end flow configuration
PolyORB configuration
Main procedure

Figure 4-7: Package structure for the PolyORB example

Middleware configuration files allow a specific architecture within our platform to be selected. In PolyORB, these files are different depending on the role of each CPU (i.e. client or server) so two Ada packages have been developed called *polyorb-setup-endpoint_server* (see Listing 4-2) and *polyorb-setup-endpoint_client* (see Listing 4-3).

Listing 4-2: Package polyorb-setup-endpoint_server.adb

```
-- Basic configuration of PolyORB
with PolyORB.Setup.Base;
-- CORBA POA Configuration
with PolyORB.Setup.OA.Basic_POA;

-- Tasking profile
with PolyORB.Setup.Tasking.Full_Tasking_Endpoints;
-- Tasking policy
with PolyORB.ORB.Thread_Per_Target;
-- ORB Controller policy
with PolyORB.ORB_Controller.Ready_To_Go;

--  Transmission of Event Id parameter
with PolyORB.QoS.Event_Parameters;

--  Personalities setup
with PolyORB.Setup.RTEP_MAC;
with PolyORB.Setup.Access_Points.RTEP_MAC;

package body PolyORB.Setup.Endpoint_Server is
end PolyORB.Setup.Endpoint_Server;
```

Listing 4-3: Package polyorb-setup-endpoint_client.adb

```
-- Basic configuration of PolyORB
with PolyORB.Setup.Base;

-- Tasking profile
with PolyORB.Setup.Tasking.Full_Tasking_Endpoints;
-- Tasking policy
with PolyORB.ORB.Thread_Per_Target;
-- ORB Controller policy
with PolyORB.ORB_Controller.Ready_To_Go;

--  Transmission of Event Id parameter
with PolyORB.QoS.Event_Parameters;

--  Personalities setup
with PolyORB.Setup.RTEP_MAC;
with PolyORB.Binding_Data.GIOP.EP;
with PolyORB.Binding_Data.GIOP.Endpoints;
with PolyORB.Binding_Data.GIOP;

package body PolyORB.Setup.Endpoint_Client is
end PolyORB.Setup.Endpoint_Client;
```

Listing 4-4: client_configuration.adb

```
procedure Client_Configuration is
  ROP_Snd_Endpoint   : Send_Endpoint_Id;
  ROP_Reply_Endpoint : Receive_Endpoint_Id;
  ROP_Params         : Priorities_Message_Scheduling_Parameters_Ref :=
                          new Priorities_Message_Scheduling_Parameters;
begin
  ROP_Params.Message_Priority := 15;
  --  Create one send endpoint and one reply receive endpoint
  Create_Send_Endpoint (Param     => ROP_Params,
                        Dest_Node => Server_Node,
                        Event     => 1,
                        Net       => RTEP_Network,
                        Dest_Port => 5,
                        Endpoint  => ROP_Snd_Endpoint);

  Create_Reply_Receive_Endpoint (Net        => RTEP_Network,
                                 Event_Sent => 1,
                                 Port => 5,
                                 Endpoint  => ROP_Reply_Endpoint);
end Client_Configuration;
```

For the server side, the application uses the CORBA distribution model (Listing 4-2, lines 2 to 4), the RATO and TPT policies (Listing 4-2, lines 7 to 11) for the management of remote calls and the RT-EP protocol (Listing 4-2, lines 17 to 18). The scheduling policy used for the processor is FPS, although this feature is selected through the endpoints API. Lastly, it is also necessary to specify that the *Event_Id* parameter will be transmitted as part of the *Service Context* field (Listing 4-2, line 14).

For the client side, the PolyORB configuration file is similar to the file already described for the server. However, in this case, PolyORB requires the inclusion of three additional packages to select the RT-EP protocol personality (Listing 4-3, lines 16 to 18).

After setting up the distribution middleware architecture, our next step deals with the real-time configuration for the synchronous end-to-end flow. According to the *endpoints pattern*, each element of the end-to-end flow can be described as a part of a configuration operation executed during the *real-time configuration stage* (see Section 3.4.4). Listing 4-4 and Listing 4-5 show these real-time configuration files for the client and server nodes, respectively. The former requires a send endpoint to be created to

Listing 4-5: server_configuration.adb

```
procedure Server_Configuration is
  ROP_Snd_Reply_Endpoint    : Send_Endpoint_Id;
  ROP_Receive_Endpoint      : Receive_Endpoint_Id;
  ROP_Handler               : Handler_Id;
  ROP_Handler_Params        : Priorities_Task_Scheduling_Parameters_Ref :=
                                 new Priorities_Task_Scheduling_Parameters;
  ROP_Reply_Params          : Priorities_Message_Scheduling_Parameters_Ref :=
                                 new Priorities_Message_Scheduling_Parameters;
begin
  -- Create one receive endpoint and one reply send endpoint
  Create_Receive_Endpoint (Net      => RTEP_Network,
                           Port     => 5,
                           Endpoint => ROP_Receive_Endpoint);

  -- Create handler task and scheduling params associated
ROP_Handler_Params.Handler_Priority := 50;
  Create_Handler_Task (Params => ROP_Handler_Params,
                       Endpoint      => ROP_Receive_Endpoint
                       Handler_Task => ROP_Handler);

  ROP_Reply_Params.Message_Priority := 15;
  Create_Reply_Send_Endpoint (Param   => ROP_Reply_Params,
                              Dest_Node => Client_Node,
                              Event     => 2,
                              Net       => RTEP_Network,
                              Dest_Port => 5,
                              Endpoint  => ROP_Snd_Endpoint);

  Set_Event_Association (Input_Event  => 1
                         Output_Event => 2);
end Server_Configuration;
```

execute the remote request and a receive reply endpoint to wait for its result (see Listing 4-4). The latter requires a handler task to be created and a receive endpoint to process the incoming request and a send reply endpoint to transmit the result. Furthermore, the event mapping must also be included in the real-time configuration file for the server node (see Listing 4-5).

Finally, the application code for the main server and client procedures are shown in Listing 4-6 and Listing 4-7, respectively. Those lists deliberately omit the configuration corresponding to a CORBA system, but include calls to configure the end-to-end flow (line 6 for Listing 4-6 and Listing 4-7) and the setting of the initial *Event_Id* (see Listing 4-6, line 9).

Listing 4-6: server.adb

```
 1: procedure Server is
 2: begin
 3:   --  Set up new CORBA object
 4:   ...
 5:   -- Real-time configuration
 6:   Server_Configuration_File;
 7:
 8: --  Launch the server
 9:   ....
10: end Server;
```

Listing 4-7: client.adb

```
 1: procedure Client is
 2: begin
 3:   --  Getting the CORBA object
 4:   ...
 5:   --  Real-time configuration
 6:   Clie nt_Configuration_File;
 7:
 8: --  Set Event_Id
 9:   Set_Event_Id (Event);
10:
11:   loop
12:     Result := ROP (ROP_Ref, Input_Params);
13:   end loop;
14:
15: end Client;
```

4.4 CASE STUDIES

This section deals with the validation of the *endpoints pattern* by defining two case studies which allow the most outstanding features of our proposal to be evaluated:

- *Applying the endpoints pattern in a real and complex system*, whose main goal is to evaluate the complexity in the development of distributed real-time applications running over our platform.
- *Adapting and using the endpoints pattern in dynamic systems.* This case study permits the evaluation of the capabilities of our platform to be adapted to dynamic changes in the system workload.

4.4.1 APPLYING THE ENDPOINTS PATTERN IN A REAL AND COMPLEX SYSTEM

We have evaluated the impact of migrating a real application to the proposed approach. The test platform consists of an industrial robotic arm (BTM) controlled by a man-machine interface. In this case, the objective is to evaluate the complexity introduced by applying the *endpoints pattern* to the controller software of a fully operative robot.

The BTM is a robotic arm composed of six independent axes driven by servo motors. The controller software was developed previously so the effort has mainly focused on adapting the source code to the distribution middleware developed in this thesis. The main features of the platform are described below and in Figure 4-8:

- Hardware architecture. The platform shows a distributed architecture connected through a 100 Mbps Ethernet network and is composed of two processing nodes: the *local robot controller*, which manages the electrical connections with the remotely manipulated arm, and the *man-machine interface*, which sends the orders to the controller and periodically supervises the system status.
- Software architecture. The robot software is divided into several software modules, which are represented in Figure 4-8 and briefly described next. On the one hand, the man-machine interface node includes the modules named *Control Manager*, which manages the commands that the operator introduces through the interface board, *Trajectory Planner*, for calculating the trajectories that the robot must follow, and *Reporter,* to collect, display and send information about the state of the system. On the other hand, the

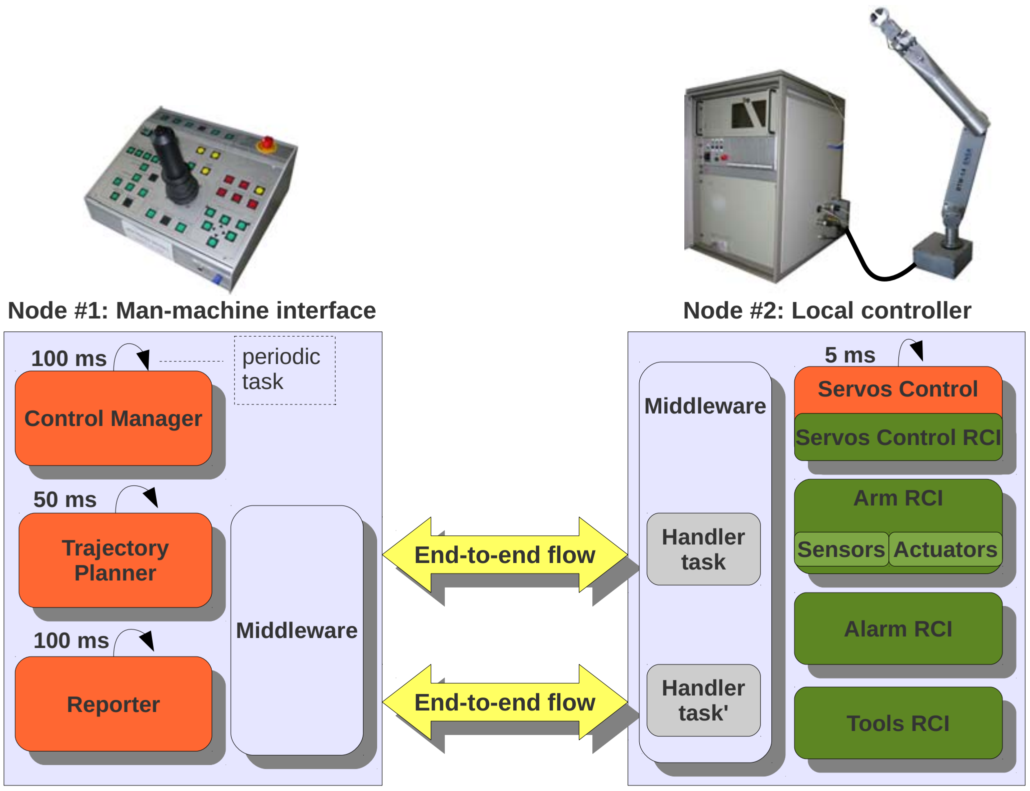

Figure 4-8: Distributed architecture of the BTM

local controller implements four modules which provide different remote services: *Servo Control*, which implements the servo motors' digital control algorithm, *Arm*, which defines the basic elements of the remotely operated arm grouped into two modules (sensors and actuators), *Alarm*, to manage the alarms state for the overall control system, and *Tools*, which includes the operations associated with the tools that the robot may have.

- End-to-end flow model. Observing the software architecture shown in Figure 4-8, we can see that the BTM application defines four periodic tasks which are included in *Control Manager, Trajectory Planner, Reporter* and *Servo Control* modules. In this case, we have four end-to-end flows, each corresponding to one of the periodic tasks, but only two of them (the *Trajectory Planner* and the *Reporter* end-to-end flows) are distributed as is shown in Figure 4-8.
- Timing requirements. Hard Real-Time deadlines for control operations.

The distributed real-time platform developed in this chapter is responsible for providing strict timing guarantees for the processing and the communications involved in the BTM. In particular, the tests have been executed over a platform with the MaRTE

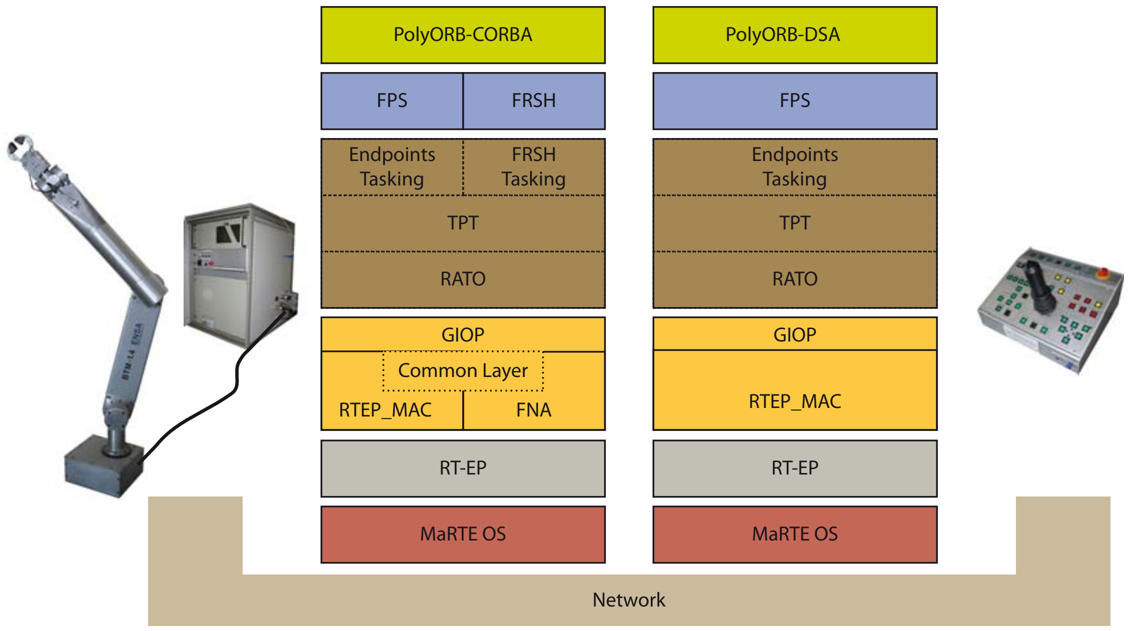

Figure 4-9: Distributed real-time platform for the BTM

v1.9 operating system, the modified version of PolyORB 2.4 and the FRESCOR Framework v. January 2009.

Figure 4-9 summarizes the platform configurations tested with the distributed controller, which uses the version of the *endpoints pattern* described in Figure 4-6-B. As is shown in Figure 4-9, the software has been adapted to use both FPS and FRESCOR contract scheduling policies. To avoid unnecessary context switches, middleware has been configured to use the RATO and TPT policies for the processing of remote calls. Furthermore, the RT-EP protocol has been selected as the communication network, using both fixed-priority and contract-based implementations.

The distribution has been performed following both RT-CORBA and Ada DSA standards. In the first case, the Ada packages involved in the remote services have been re-written using the IDL language. Some difficulties have arisen in this step since the Ada mapping to IDL specification [ALM01] does not cover all the aspects of the Ada programming language. Some restrictions such as Ada ranges or subtypes do not have an equivalent item in the IDL language, and moreover, predefined types (e.g. digits) or arrays indexed by enumeration types lead to a minor modification of the application code.

On the other hand, a version has also been developed that uses Ada DSA and FPS scheduling policy. As was introduced in Section 4.1.2.4, the DSA application personality of PolyORB requires a naming server to be executed so a third node has been added to the platform for this case. The evaluation results with the DSA personality were

Table 4-9: Complexity evaluation for the BTM (SLOCs)

DISTRIBUTION STANDARD	SCHEDULING POLICY	ENVIRONMENT INITIALIZATION	REAL-TIME CONFIGURATION	TOTAL
RT-CORBA	PRIORITIES	38	23	61
RT-CORBA	FRSH	38	38	76
DSA	PRIORITIES	6	37	43

not satisfactory due to a bug in the management of the dynamic memory. This bug caused the middleware not to deallocate the memory previously allocated per request, running out the available memory after processing a certain number of remote calls. Because of this, the implementation based on FRESCOR contracts has been discarded.

The evaluation of the complexity in the integration of both distribution standards within the distributed application has been measured computing the number of extra source lines of code (SLOC) required to use the *endpoints pattern* (see Table 4-9). This table shows the SLOC necessary for the environment initialization (e.g., selecting tasking and controller policies, CORBA or DSA initialization operations, etc), the real-time configuration (e.g., creating communication endpoints and handler tasks) and the total. Three different versions of distribution middleware have been evaluated: RT-CORBA with both FPS and FRSH scheduling, and DSA with FPS scheduling. As can be observed, with the policy based on fixed priorities, the number of SLOC used by RT-CORBA for the real-time configuration (23) is lower than for DSA (37). That slight difference is caused by the new end-to-end flows in charge of the naming server operations, that is, the registration and retrieval of the remote services location. When the application is configured to use RT-CORBA with FRESCOR contracts, the SLOC added is 38. That increase is due to the number of scheduling parameters included in a contract and the negotiation process for external Ada tasks (i.e. not created by FRESCOR framework). As final results, the RT-CORBA version required the addition of 61 SLOC in total while DSA required only 43 SLOC over a project of more than 13,000 SLOC. This comparison has not computed the necessary SLOC to *with'ed* new packages and the overhead introduced by the data type mapping in the RT-CORBA case.

The adaptation of a complex real-time system, such as the BTM, to the proposed platform has allowed us to validate the main features of the *endpoints pattern*: minimal impact on the application code (less than 1%), full support for heterogeneous scheduling policies and parameters (tested with FPS and FRESCOR contracts) and full control over the creation and configuration of schedulable entities (by applying the RATO and TPT policies).

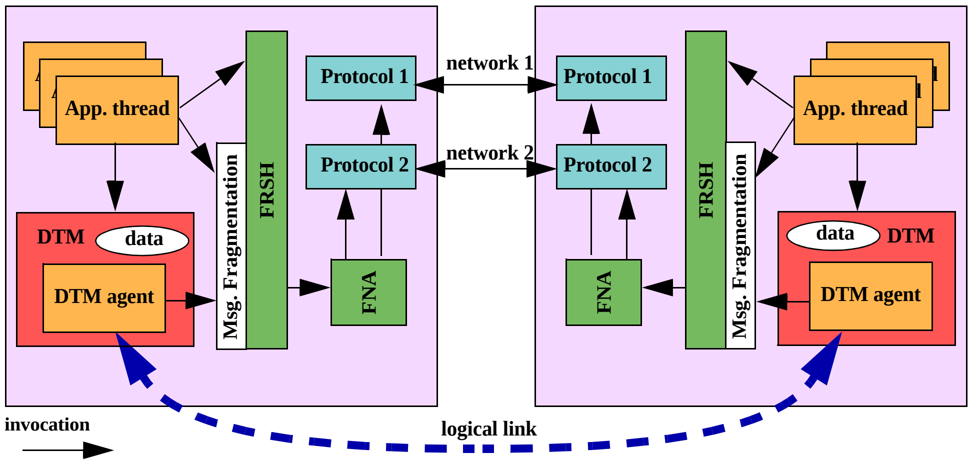

Figure 4-10: DTM implementation and FRESCOR framework

4.4.2 ADAPTING AND USING THE ENDPOINTS PATTERN IN DYNAMIC SYSTEMS

The FRESCOR project had the objective of providing a flexible scheduling framework able to apply advanced real-time scheduling techniques transparently. These techniques may be applied either to systems with static workload, characterized by having a fixed number of end-to-end flows, or those systems with dynamic workload and, as a consequence, a variable number of end-to-end flows. In the latter, the application requirements related to a given resource are mapped to a contract which is negotiated at runtime and may or may not be admitted in the system.

This last case will enable us to check the use of the *endpoints pattern* in dynamic systems. According to the end-to-end flow model, this dynamic workload is interpreted as new end-to-end flows arriving at the system which may or may not be executed, depending on the number of resources available. In a distributed system, accepting a new end-to-end flow requires local and remote information from each node involved in the negotiation process. As was previously commented, this complex process is efficiently managed though a high-level tool called *Distributed Transaction Manager* (DTM) [FRSH09-C]. Therefore, the first objective of this section is adapting the DTM to our distributed real-time platform.

Figure 4-10 shows the overall DTM architecture as viewed from the application. The DTM has been designed as a layer between the application and the FRESCOR API, called FRSH, in order to avoid increasing the complexity of the FRESCOR framework. The existing version of the transaction manager limits its capabilities to the management of remote contracts only. Under this implementation, the DTM contains an agent in every node, which listens for messages either from the local node or from remote nodes, performs the requested actions, and sends back the replies (see Figure 4-10). In every node there is also a DTM data structure with the information used by the corresponding agent. Part of this information is shared with the DTM services invoked locally from the application threads. As was described before, this architecture was initially implemented directly over the network communication primitives provided by FNA as shown in Figure 4-10, but could alternatively be implemented using different distribution standards (e.g. CORBA, DDS, or Ada DSA), thus simplifying the complexity of the communication among agents, as well as enabling the use of all the basic services provided by middleware (initialization, fragmentation, data codification, etc).

Furthermore, this architecture could benefit from the presence of a distribution middleware based on the *endpoints pattern* by providing full support for the end-to-end flow model. To this end, the DTM should be provided with the following services:

- Specification of the full end-to-end flow with identification of its activities, remote services and events, and contracts for the different resources (processors and networks).
- Automatic deployment of the end-to-end flow in middleware. This would require:
  - choosing unused *Event_Id*s for the end-to-end flow events.
  - choosing unused ports in the nodes involved, for the communications.
  - creating send endpoints for the client-side of the communications, using the desired contracts and networks.
  - creating receive endpoints for the reception of the reply on the client-side of the communications, using the desired networks, ports, and Event_Ids.
  - creating the necessary handler tasks with their corresponding contracts.
  - creating the receive endpoints on the server-side for the communications using the desired contracts and networks.
  - creating the send endpoints on the server-side for the communication using the desired contracts and networks.

All this deployment would be done automatically by the DTM using the configuration information of the end-to-end flow. After the *real-time configuration stage*, the end-to-end flow would start executing, its remote operations would be invoked and middleware would automatically direct them through the appropriate endpoints and handler tasks almost transparently. We would only specify the appropriate Event_Ids.

The DTM is essentially a distributed application that, in this case, will be implemented using the CORBA personality. Thus, the remote data types and services provided by the DTM must be described through the IDL language [COR03]. In particular, this has required the definition of:

- **DTM internals**
  - *Send_Message*. A remote call interface used to exchange the different kinds of messages between the DTM agents.
  - *Idl_Dtm_Message*. A data record used to store a generic message unit. It consists of a header, indicating the message type, and a body, storing the specific data associated with a particular message type.
  - *Exchanged data types*: Each kind of message has different data structures which will be exchanged among the agents. All of them must be described in IDL, including those consisting of complex types such as the contracts.
- **DTM services**
  - *Initialization.* This service assures that every agent in the distributed system is ready to send and receive requests. In this case, the initialization process is based on the CORBA Naming Service [NAM04]. Under this approach, each agent should register its IOR on the *name server* when it becomes ready to start or accept requests, and obtain the IOR from the remaining agents to assert that they are also in the ready state.
  - *Routing*. In this context, routing means the capacity to interconnect different networks in systems where nodes are not connected directly. This service should be included as part of distribution middleware to limit the overhead that would be incurred by crossing all the implementation layers up to the application level so it has been implemented in PolyORB within the GIOP protocol (see Figure 4-11). Our approach focuses on routing through the *Event_Id*. Once the routing node has identified the event parameter and thus the ongoing end-to-end flow, it asks middleware whether the invoked object is located in the local node or not. If not, the implementation will route the incoming message through the pre-configured send endpoint.

Figure 4-11 represents a general overview of the DTM integrated into PolyORB. The current version uses PolyORB-CORBA, FRSH contracts for the scheduling policy, and RATO + TPT policies (which support the explicit creation of handler tasks and their association with endpoints). Since the DTM and its integration

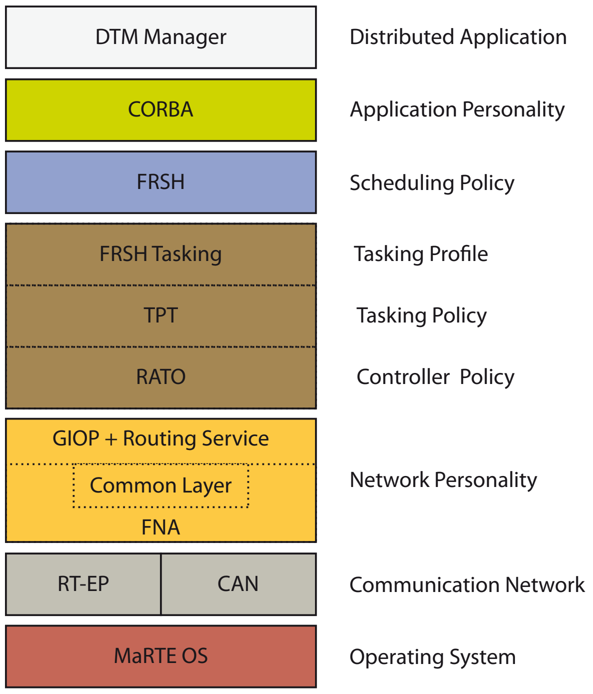

Figure 4-11: The integration of the DTM into distribution middleware

into distribution middleware is a complex development, any interested reader may consult further details in [SAN10].

The last step is validating the use of the *endpoints pattern* in dynamic systems by obtaining some time metrics for the negotiation and admission of new end-to-end flows into the system. We have evaluated the DTM over a platform consisting of four 800 Mhz embedded nodes connected through a 100 Mbps Ethernet with GNAT GPL 2008, MaRTE OS 1.9 and the modified version of PolyORB 2.4. Three tests have been run for two, three and four processing nodes. The tests consist of 10 independent asynchronous end-to-end flows, each one negotiating one contract for each processing node and one contract for each link over the network. Table 4-10 shows the times measured for these three tests. As can be seen, the distributed negotiation process takes less than 20 ms for two processing nodes. In the cases of three and four processing nodes, this time is not increased significantly despite the higher overhead in the network and the extra negotiations of contracts. In spite of the fact that the DTM specification might seem complex and laborious, we have found that the system can obtain good performance when negotiating new end-to-end flows.

4.5 ADVANCES OVER RELATED WORK

As was introduced in Section 3.3.1, the integration of the end-to-end flow model into distribution middleware has already been considered in a previous work. This work led to the development of middleware technology called RT-GLADE, a modification of GLADE which adds a set of extensions to optimise its real-time behaviour. There are two versions of RT-GLADE: in the first one [LOP04], free assignment of priorities in remote calls is allowed (both in the processors and in the communication networks); the second version [LOP06] proposes a way of incorporating distributed transactions or end-to-end flows into the DSA and providing support to different scheduling policies in a distributed system.

Although the basic differences between the two proposals have been briefly introduced, the implementation of the distributed real-time platform with PolyORB adds several new features as shown in Figure 4-12. Each of these differences represents a step forward in this previous work and is described next:

- **Separate interfaces for processing resources**

  RT-GLADE proposes a single interface to configure both processing resources. However, our proposal defines different interfaces for *handler tasks* and *communication endpoints*, which eases the integration of new, possibly different, scheduling policies both in processing nodes and networks.

- **Shared use of handler tasks among different end-to-end flows**

  As a result of defining different interfaces for processors and networks, the initial restriction of having a *handler task* associated with a particular Event_Id can be removed, thus allowing the same handler task to be shared by several end-to-end flows.

Table 4-10: DTM metrics to negotiate dynamic end-to-end flows using middleware (times in μs)

NUM. OF NODES	MAX	MED	MIN	STD. DEVIATION
TWO NODES	19508	19383	19133	111
THREE NODES	25546	25219	24470	330
FOUR NODES	33255	32713	32565	199

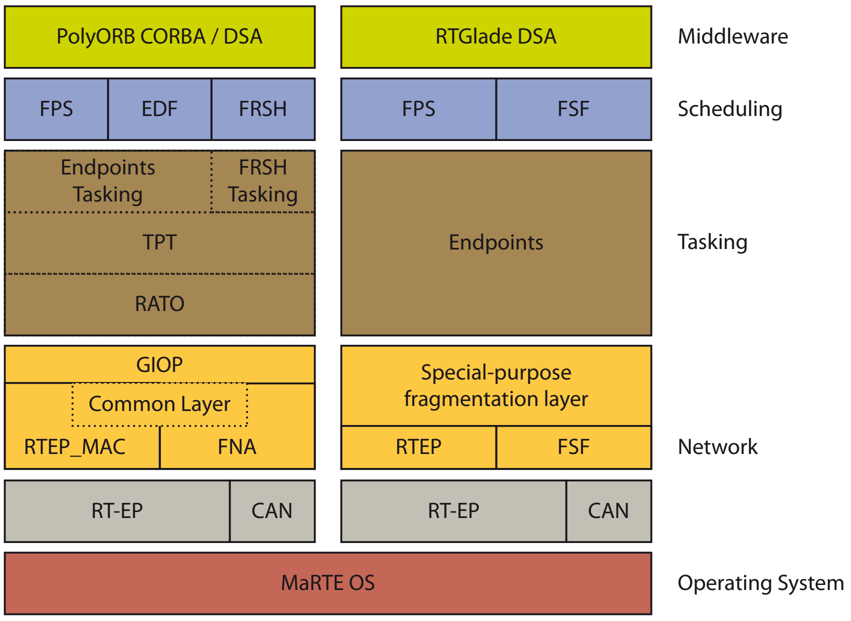

Figure 4-12: Architecture for the distributed real-time platform

- **Automatic management of events**

  Our proposal supports a flexible end-to-end flow model with event transformations to avoid the management of real-time aspects within the application code. This management is done at runtime, but using the configuration information defined at initialization time. It also supports complex event patterns which are different from linear ones.

- **API designed to ease the use of CASE tools**

  The proposed interfaces facilitate the automatic generation of the real-time configuration code through CASE tools, especially with those based on the end-to-end flow model such as MAST. As was described in Section 3.6, the transition between the MAST model and the real-time model integrated into middleware is almost straightforward.

- **Support for different distribution models**

  As is illustrated in Figure 4-12, the interface defined for RT-GLADE was only validated for Ada DSA, while our proposal has been validated in both CORBA and DSA distribution models.

- **Support for standardized communications**

  The proposed approach relies on the standard GIOP protocol for communications. This protocol enables the interoperability with other

CORBA implementations. Furthermore, it also provides a standard fragmentation service, while RT-GLADE implements a specific fragmentation layer for its implementation.

- **Validation of the proposal in dynamic systems**

  The use of the FSF framework in RT-GLADE does not include a high-level manager to handle the remote negotiation and renegotiation of contracts and the coherence of the results of these processes. Therefore, FSF contracts must be negotiated locally and so new end-to-end flows must be scheduled offline. However, the FRESCOR framework adds support for the distributed negotiation of contracts through the DTM distributed tool and thus our approach can be applied to distributed real-time systems with variable workload. In order to provide support for this kind of scenario, the DTM has been adapted to run over our distributed real-time platform by developing it as a CORBA distributed application and also integrating it into distribution middleware.

- **Adding new schedulers**

  The middleware technology included in RT-GLADE was only validated for static systems using FPS and FSF contracts [ALD06] scheduling policies. Our development, besides the static systems scheduled with FPS and FRESCOR contracts, adds support for these kinds of systems scheduled through the EDF policy (see Figure 4-12). This case has only been tested using simple examples with synthetic workload in order to check whether this policy can be used with the *endpoints pattern*.

4.6 CONTRIBUTIONS OF THIS CHAPTER

This chapter has addressed the validation of the *endpoints pattern* by implementing a distributed real-time platform capable of providing strict temporal guarantees. Among other features, this platform has been developed to provide the end-to-end flow model with support within distribution middleware, interchangeable scheduling policies such as FPS, FRESCOR contracts and EDF, and a set of extensions to enhance the system predictability. These improvements include (1) porting to a new real-time operating system (MaRTE OS), (2) a new communication layer to use different real-time protocols such as RT-EP or CAN, and (3) a new internal management of middleware tasks by defining new tasking and controller policies and profiles.

One major contribution of this chapter is the use of the *endpoints pattern* with different distribution models, such as RPCs for Ada DSA or distributed objects for RT-CORBA. Figure 4-6 summarizes the architecture of our distributed real-time platform and shows the supported distribution models, tasking and scheduling policies, and network protocols integrated into PolyORB.

After the development of the distributed real-time platform, two case studies have been proposed to validate the *endpoints pattern* in different scenarios. In a static system, where there are a predefined number of end-to-end flows, the complexity of adapting an existing real-time system to the proposed model was evaluated, and for dynamic systems, the tests carried out demonstrated the capability of our approach to be adapted to environments whose workload is variable.

In the static case, the test results were satisfactory and allowed several design goals to be evaluated: (1) separation of concerns between the logic of the application and the real-time configuration code, (2) use of heterogeneous scheduling policies and parameters, and (3) control in the creation and configuration of schedulable entities. Likewise, the tests executed for dynamic systems showed that our approach can provide temporal guarantees even when new end-to-end flows are being negotiated in the system.

5 ADAPTATION OF THE ENDPOINTS PATTERN TO HIGH-INTEGRITY DISTRIBUTED REAL-TIME SYSTEMS DEVELOPED IN ADA

This chapter focuses on the adaptation of the proposed endpoints pattern to the restrictions imposed by high-integrity systems. The first section introduces the basic concepts and the Ravenscar profile, a subset of the Ada programming language that facilitates the development of safety-critical applications. Section 5.2 describes the necessary modifications to the original API in order to be compatible with Ravenscar. The new endpoints API for Ada is proposed in Section 5.3, while Section 5.4 shows an example of use. Section 5.5 validates the proposal by implementing it over a high-integrity distributed real-time platform. Section 5.6 introduces an environment for the development of high-integrity distributed real-time applications where the proposal can be integrated. Section 5.7 lists the minimum set of requirements that should be implemented to support the end-to-end flow model in those systems that must be configured at compilation time. A review of the related work is included in Section 5.8. Finally, Section 5.9 summarizes the contributions of the chapter.

5.1 HIGH-INTEGRITY SYSTEMS AND ADA

The correct operation of a current complex computer system can depend on a wide range of factors. For example, as noted in Chapter 1, a real-time application not only depends on its logical result, but also on the time at which the result has been produced. Furthermore, other real-time systems may even consider safety factors. For example, 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, etc.

The main objective for high-integrity systems lies in the development of reliable applications where the simplicity of the system is seen as an essential property. One of the major advantages of Ada over other programming languages is the definition of profiles that restrict the use of certain aspects of the programming language to favour the simplicity of the analysis. In particular, Ada provides the following facilities to develop safety-critical systems:

- **ISO/IEC TR 1594** [GAHI00]. *Guide for the use of the Ada programming language in high-integrity systems* (GA-HI). This document transfers the general restrictions on the development of a high-integrity system to the specific field of Ada.
- **SPARK** [SPA10]. It is a subset of Ada which restricts the use of certain features to facilitate the static analysis, focusing exclusively on the sequential part of this programming language. However, not only does SPARK restrict Ada but it also adds formal annotations within the source code to perform data flow analysis automatically.
- **Pragma Restrictions** [ADA05]. It is a compiler directive that allows the developer to select the restrictions that must be applied to software.
- **Ravenscar profile** [ADA05]. It is a profile that defines a safe and analyzable subset of Ada concurrency facilities. Therefore, it aims to define a deterministic concurrency model for Ada. It consists of a set of restrictions.

Since the *endpoints pattern* facilitates the application of schedulability analysis, its use in high-integrity systems is mainly limited by the restrictions defined by the Ravenscar profile. Therefore, this chapter focuses on the adaptation of the proposed real-time model to Ravenscar, but it also identifies the potential conflicts that may arise with the other aforementioned Ada tools.

5.2 ADAPTING THE ENDPOINTS PATTERN TO THE RAVENSCAR PROFILE

The Ravenscar profile defines a set of restrictions in the system concurrency model, thus imposing a review of the *endpoints pattern*. The complete list of restrictions included in the profile can be found in [ADA05], although only a subset of them may affect the proposal. In particular, the following restrictions must be considered:

- **Only the FIFO_Within_Priorities dispatching policy is allowed**.

  The proposed approach is independent of the selected policy. However, this flexibility does not violate this restriction because the choice of available scheduling policies remains implementation-defined. Furthermore, keeping this flexibility enables future profile extensions [WHIT10].

- **The set of tasks in the system is fixed and created at library level.**

  The *Processing_Node_Scheduling* interface provides operations to create handler tasks at the real-time configuration stage, which is not Ravenscar compliant. Therefore, it requires a revision of the proposal in order to create handler tasks at library level.

- **Tasks have static scheduling parameters.**

  Although the current approach allows handler tasks to update their scheduling parameters at runtime according to the retrieved Event_Id, this feature is not compatible with Ravenscar and must be disabled. However, the transmission of the Event_Id parameter remains necessary in order to allow the same handler task to be shared among multiple end-to-end flows, if necessary.

Although the *endpoints pattern* does not violate any further Ravenscar restrictions, there are some other aspects that middleware implementations should take into account:

- **Prevent the use of task attributes.**

  As was said earlier in Chapter 4, the implementation of the *endpoints pattern* for PolyORB [PER08] uses task attributes to store and retrieve the Event_Id parameter.

- **All tasks are non-terminating.**

  The *Processing_Node_Scheduling* interface provides operations to destroy handler tasks and, therefore, these operations must be disabled.

The Ravenscar profile has mainly been applied for the static analysis of applications running within a single node or multiple nodes but without considering the communication networks in the analysis. However, the real-time distributed model proposed in this work includes the endpoints as schedulable entities representing the communication points within the network. The ARINC 653 specification [ARINC06] follows a similar approach through the definition of *ports*, entities that enable the inter-partition and intra-partition communication and behave as the input points to a real-time network (e.g., AFDX [AFDX09]).

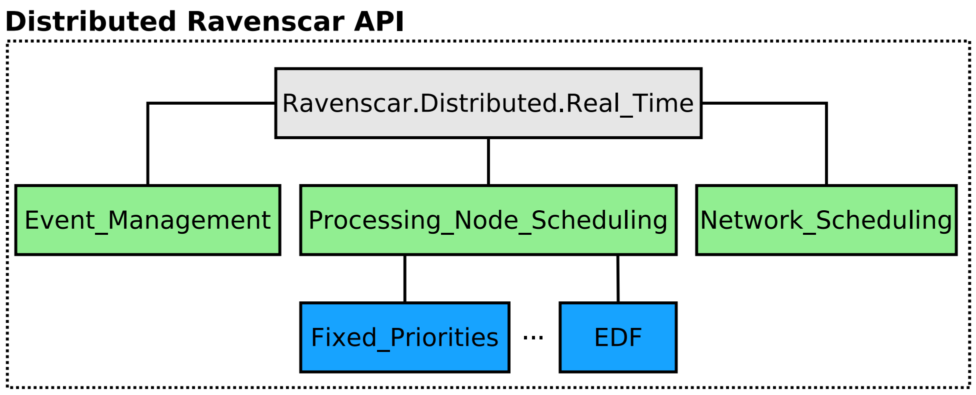
Distributed Ravenscar API
Ravenscar.Distributed.Real_Time
Event_Management
Processing_Node_Scheduling
Network_Scheduling
Fixed_Priorities
...
EDF

Figure 5-1: Package hierarchy for end-to-end flow Ravenscar systems

5.3 THE ENDPOINTS API FOR HIGH-INTEGRITY SYSTEMS

Each of these aforementioned considerations must be addressed within the set of interfaces introduced in Chapter 3. Figure 5-1 represents the Ada package hierarchy for the new proposal for integrating the *endpoints pattern* into high-integrity systems. The modifications proposed over the original API are detailed in the following subsections.

Listing 5-1: Package r-distributed-real_time-event_management.ads

```
package Ravenscar.Distributed.Event_Management is

  procedure Set_Event_Association
   (Input_Event        : Event_Id;
    Output_Event       : Event_Id);

  function Get_Event_Association
   (Input_Event        : Event_Id) return Event_Id;

  procedure Set_Event_Id (New_Event : Event_Id);
  function Get_Event_Id return Event_Id;

end Ravenscar.Distributed.Event_Management;
```

5.3.1 EVENT MANAGEMENT INTERFACE

Real-time developers should configure the sequence of events within an end-to-end flow, and the middleware will be in charge of automatically setting the appropriate event at the transformation points of the remote call as shown in Chapter 3. This interface is Ravenscar compliant and therefore it does not require any modification from the original (see Listing 5-1).

5.3.2 NETWORK SCHEDULING INTERFACE

The overall response time of a distributed system is strongly influenced by the underlying networks and therefore networks are required to be scheduled with appropriate techniques. This API addresses this aspect by making the communication endpoints visible, and by associating scheduling parameters to the messages sent through them. The approach defined in Chapter 3 is already Ravenscar compliant and thus it could remain unaltered. However, the use of class-wide types and operations is prohibited in high-integrity Ada systems [GAHI00] [SPA10]. As a result, a new Network Scheduling API has been defined which is shown in Listing 5-2.

The operations provided to destroy endpoints (*Destroy_Receive_Endpoint* and *Destroy_Send_Endpoint*) are not strictly necessary in this kind of static systems and can be removed. However, in order to keep similar interfaces for full and restricted Ada, both operations have been included.

Extensions of the *Message_Scheduling_Parameters* tagged type will contain the specific network scheduling parameters that must be associated with a specific send endpoint. Furthermore, each scheduling policy must implement operations to map its own scheduling parameters (e.g., priorities) onto extensions of this private type. However, the use of abstract types in high-integrity systems can be controversial, as they are forbidden by SPARK but allowed by GA-HI.

5.3.3 PROCESSING NODE SCHEDULING INTERFACE

Handler tasks are responsible for awaiting remote requests and processing them. The proposal included in Chapter 3 to create and manage handler tasks relied on the dynamic creation of tasks within the real-time configuration stage, which is forbidden in Ravenscar systems (i.e. all tasks must be created at library level). The new API uses a set of Ada packages instead: a *Processing_Node_Scheduling* package to perform the registration and identification of tasks in the system, and a set of child packages to create tasks with the appropriate scheduling parameters. The contents of the *Processing_Node_Scheduling* package are implementation-defined, but an example of this specification is shown in Listing 5-3.

Listing 5-2: Package r-distributed-network_scheduling.ads

```
package Ravenscar.Distributed.Network_Scheduling is
  type Message_Scheduling_Parameters is abstract tagged private;

  procedure Create_Receive_Endpoint
   (Net        : Network_Id;
    Port       : Port_Id;
    Endpoint   : out Receive_Endpoint_Id) is abstract;

  procedure Create_Send_Endpoint
   (Param      : Message_Scheduling_Parameters;
    Dest_Node  : Node_Id;
    Event      : Event_Id;
    Net        : Network_Id;
    Dest_Port  : Port_Id;
    Endpoint   : out Send_Endpoint_Id) is abstract;

  procedure Create_Reply_Receive_Endpoint
   (Net        : Network_Id;
   Event_Sent  : Event_Id;
   Port        : Port_Id;
   Endpoint    : out Receive_Endpoint_Id) is abstract;

  procedure Create_Reply_Send_Endpoint
   (Param      : Message_Scheduling_Parameters;
    Dest_Node  : Node_Id;
    Event      : Event_Id;
    Net        : Network_Id;
    Dest_Port  : Port_Id;
    Endpoint   : out Send_Endpoint_Id) is abstract;

  procedure Destroy_Receive_Endpoint
   (Endpoint   : Receive_Endpoint_Id) is abstract;
  procedure Destroy_Send_Endpoint
   (Endpoint   : Send_Endpoint_Id) is abstract;
private
  type Message_Scheduling_Parameters is abstract tagged ...;
end Ravenscar.Distributed.Network_Scheduling;
```

Under this proposal for high-integrity systems, implementations should define one child package per scheduling policy supported and allow the scheduling parameters to be assigned statically (for example, using a pragma). Since handler tasks must be created explicitly at library level, the new API considers the creation of tasks through a generic package which has been demonstrated to be a suitable approach [BORD07]. This generic package includes the following parameters and operations:

Listing 5-3: Package r-distributed-processing_node_scheduling.ads

```
package Ravenscar.Distributed.Real_Time.Processing_Node_Scheduling is

  function Get_Handler_Id
    (The_Task : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Handler_Id;

  function Register_Task
    (The_Task : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Handler_Id;

private
  ...
end Ravenscar.Distributed.Real_Time.Processing_Node_Scheduling;
```

- *Task scheduling parameters*. The scheduling parameters are set statically via a pragma.
- *Handler_Task_Callback*. Procedure that will be used by handler tasks as callback for each middleware implementation.
- *Create_Handler_Task_Endpoint*. This function returns the receive endpoint where the calling handler task will wait for incoming requests. Since the communication endpoints are created through the API (i.e. during the real-time configuration stage), and handler tasks are created at library level, the latter requires a way to create the associated receive endpoint before performing the I/O.

Furthermore, this generic package could be completed by including several optional subprograms and parameters; for instance, to execute the basic initialization operations required within each middleware implementation, to execute recovery procedures when an error is detected or to specify basic properties associated with a task (e.g., the stack size).

Finally, the use of generics, although common in many safety-critical Ada research projects [BORD07] [HUG08], is debatable, as it is forbidden by SPARK but allowed by GA-HI. As an example, the package shown in Listing 5-4 represents the generic unit used for fixed priorities scheduling.

Although Ravenscar only considers fixed-priority based scheduling, future extensions of this profile may include the use of other scheduling policies (e.g., EDF as shown in Figure 5-1 and proposed by [WHIT10]), which would also be supported by means of the proposed API.

Listing 5-4: Package r-distributed-processing_node_scheduling.fixed_priorities.ads

```
-- Dependences are omitted
generic
 Handler_Task_Priority : System.Priority;
 with procedure Handler_Task_Callback
            (Endpoint : Receive_Endpoint_Id);
 with function Create_Handler_Task_Endpoint return
            Receive_Endpoint_Id;

package Ravenscar.Distributed.Processing_Node_Scheduling.Fixed_Priorities is

 task FP_Handler_Task is
   pragma Priority (Handler_Task_Priority);
 end FP_Handler_Task;

end Ravenscar.Distributed.Processing_Node_Scheduling.Fixed_Priorities;
```

5.4 EXAMPLE OF USE

Once the endpoints API has been defined for high-integrity systems, this section describes how to perform the real-time configuration for a simple example. To this end, we consider the example introduced in Chapter 1 and depicted in Figure 5-2. We will show how to generate the configuration code for CPU-2, but a similar procedure could be followed for CPU-1 and CPU-3.

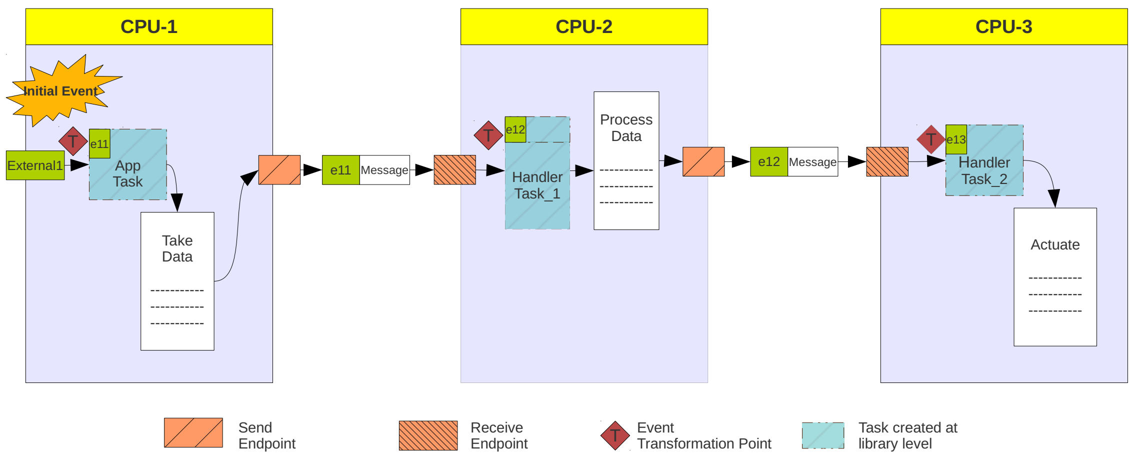

Figure 5-2: Example of use for the endpoints API in high-integrity systems

Listing 5-5: Package partition_2_configuration.ads

```
package Partition_2_Configuration is

 package Handler_Task_1 is new
  Ravenscar.Distributed.Processing_Node_Scheduling.Fixed_Priorities
   (Handler_Task_Priority        => Handler_Task_1_Priority,
    Create_Handler_Task_Endpoint =>
    Partition_2_Configuration_File.New_Receive_Endpoint,
    Handler_Task_Callback        => MW_Implementation.Main_Loop);
...
 procedure Set_Partition_Configuration;
end Partition_2_Configuration;
```

The real-time configuration shall include the code required to create and configure each end-to-end flow defined in the system. Under this approach, the real-time configuration is included in an Ada package (instead of a simple procedure) named *partition_2_configuration*. For the example illustrated in Figure 5-2, there is only one linear and asynchronous end-to-end flow that requires the definition of the following elements (see Listing 5-5):

- *Handler_Task_1* package: It is responsible for the creation of a handler task, the creation/retrieval of the associated receive endpoint and the callback to the main loop defined in the middleware implementation.
- *Set_Partition_Configuration* procedure: It performs the configuration of the rest of the end-to-end flow elements (communication endpoints, scheduling parameters and event association).

Then, the package body for *partition_2_configuration* should define the real-time configuration code for CPU-2. In particular, Listing 5-6 shows the *Set_Partition_Configuration* procedure that calls the following subprograms:

- *Set_Event_Association*. As can be seen in Figure 5-2, CPU-2 receives *e12* as an incoming event and transforms it to event *e13*. This mapping between events remains registered in middleware. Although the event transformation service does not have any effect in this specific example, this service is still required in order to allow the same handler task to be shared among multiple end-to-end flows.
- *Create_Send_Endpoint*. According to Figure 5-2, this partition considers two network transmission steps: one of them is for sending messages and so it is associated with one send endpoint, with its specific scheduling information. The other network step corresponds to the incoming messages

Listing 5-6: Package partition_2_configuration.adb

```
package body Partition_2_Configuration is

 procedure Set_Partition_Configuration is
 -- The definition of variables is omitted
 begin
   -- Set event associations
   Set_Event_Assocation (Input_Event => e2, Output_Event => e3);

   -- Create one send endpoint
   Create_Send_Endpoint
    (Param     => Msg_Scheduling_Parameters_e3,
     Dest_Node => CPU-3,
     Event     => e3,
     Net       => Default_Network,
     Dest_Port => Rcv_Port_Partition_e3,
     Endpoint  => Snd_Endpoint_Id_e3);
 end Set_Partition_Configuration;
end Partition_2_Configuration;
```

and therefore it must be mapped to a receive endpoint. However, according to the endpoints proposal for high-integrity systems, this receive endpoint is now created by the associated handler task to avoid the complexity of synchronization issues during the creation of endpoints (i.e. operation performed within the real-time configuration stage) and handler tasks (i.e. operation performed at library level).

5.5 INTEGRATION AND VALIDATION OF THE ENDPOINTS PATTERN IN A HIGH-INTEGRITY DISTRIBUTED REAL-TIME PLATFORM

5.5.1 OVERVIEW OF THE HIGH-INTEGRITY DISTRIBUTED REAL-TIME PLATFORM

This section describes how the endpoints pattern has been integrated into a high-integrity platform composed of Ocarina and PolyORB-HI tools [HUG08]. This set of tools provides an appropriate framework to develop high-integrity distributed real-time applications (HDRT) through the automatic generation of source code from high-level system models.

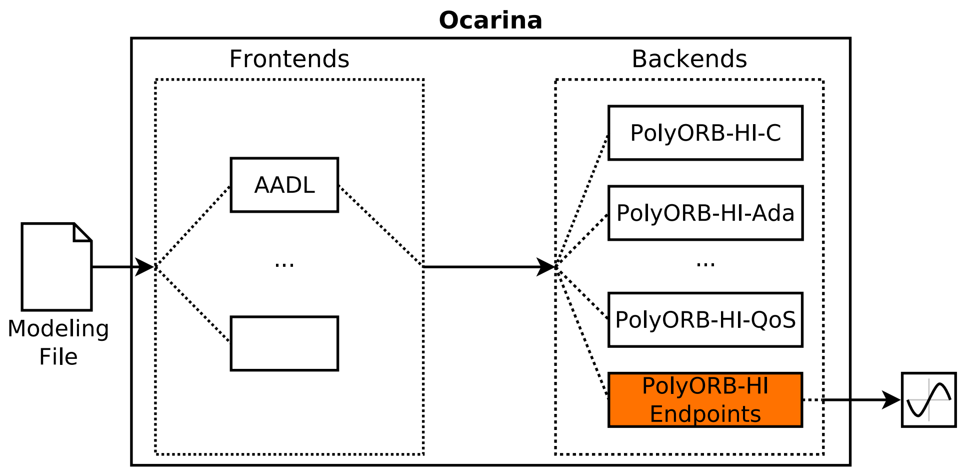

Figure 5-3: Architecture overview for Ocarina

Figure 5-3 shows the architecture of Ocarina which comprises two different parts: a *frontend*, which processes the system model described in the input file, and a *backend*, which implements the strategies to generate the source code for different targets. The current version supports the AADL modelling language [SAE09] as input and several targets, such as those based on the PolyORB-HI middleware, as output.

PolyORB-HI is a lightweight distribution middleware compatible with the restrictions specified by the Ravenscar profile. It is distributed with the Ocarina tool as an AADL runtime that provides all the required resources (i.e. stubs, skeletons, marshallers and concurrent structures) to build high-integrity distributed systems. Additionally, it also provides a communication layer which takes care of the physical communication over the network. The current software release provides three runtimes depending on the target system: PolyORB-HI-C, PolyORB-HI-QoS and PolyORB-HI-Ada.

PolyORB-HI-Ada supports both native (e.g. Linux or Solaris for testing purposes) and high-integrity platforms (e.g. ERC32 [ATMEL05] or LEON bare board [LEON05] targets). Communication between nodes can be enabled, with current support for TCP/IP sockets on native platforms, and the SpaceWire fieldbus [SPW08] on LEON boards.

The process of producing a working application from an AADL model in Ocarina uses three main entities: (1) an AADL model and application logic (i.e. user code), (2) a code generator or *backend* and (3) a minimal middleware known as PolyORB-HI. The process starts with the description of the system through an AADL model. Then, the code generator automatically produces Ada code from this model by mapping the AADL constructs onto the middleware primitives, which represent an abstraction layer on top of OS concurrency primitives and communication stacks. Finally, the code generated is compiled with the middleware and the user code to create the executable.

Basically, the code generated relies on middleware which provides it with basic services (i.e. tasking and communication facilities, data types or device drivers). To interface with PolyORB-HI, the *backend* automatically creates a set of files which are described below:

- *PolyORB_HI_Generated.Types*, a package that includes the data types used in a given node.
- *PolyORB_HI_Generated.Activity*, a package that includes the protected objects and tasks defined for a given node.
- *PolyORB_HI_Generated.Subprograms*. The code generator implements each of the AADL subprograms defined for a given node in this package.
- *PolyORB_HI_Generated.Marshallers*. This package provides the basic mechanisms of data representation, including the subprograms for marshalling and unmarshalling data.
- *PolyORB_HI_Generated.Naming*, a package that provides a naming service through static tables for each of the supported transport protocols.
- *PolyORB_HI_Generated.Transport*, a package that provides high level operations for exchanging messages among local or remote entities. It provides subprograms to send and receive messages through the correct low transport layer based on the source or the destination of a message.
- *PolyORB_HI_Generated.Deployment*. This package provides information about the topology of the distributed application.
- *Main procedure*. For each node of the distributed application, the code generator produces a main procedure that includes the remaining components not included in the other packages and the initialization procedures.

The middleware layer defined for PolyORB-HI may be regarded as a passive entity (i.e. it does not implement any task). It provides a set of generic packages which defines different types of tasks that must be instantiated during the code generation process. Let us briefly review them in some coarse detail:

- *Periodic*, a type of task which periodically executes a job with a predefined priority.
- *Aperiodic*, a type of task which waits for a triggering event to execute a job with a predefined priority.
- *Sporadic*, a type of task which waits for a triggering event to execute a job with a predefined priority but also guarantees that a minimal inter-arrival time between event processing has elapsed.

- *Background*, a type of task to execute a single job with a predefined priority.
- *Hybrid*, a type of task which implements both periodic and sporadic behaviour. The periodic behaviour is driven by another type of task, called hybrid task driver.
- *Hybrid driver*, a type of task which sends periodic events to hybrid tasks.

Although PolyORB-HI is mainly a passive entity, its communications layer is designed to be an active entity. In PolyORB-HI, network devices are managed by one or more protocols instances. Each of these instances creates one internal task to decouple the I/O events arriving at one specific receiving port. Then, incoming network messages are stored in a protected object where a task instance (e.g. a sporadic task) will be waiting to process it.

5.5.2 MODIFICATIONS AND EXTENSIONS APPLIED TO THE HIGH-INTEGRITY DISTRIBUTED REAL-TIME PLATFORM

In order to validate the proposed *endpoints pattern* for high-integrity systems, both Ocarina and PolyORB-HI have been extended to provide a new backend or code generation strategy called *PolyORB-HI-Endpoints* (see Figure 5-3). Table 5-1 summarizes the files required to build this new backend, and also describes the purpose of each new file. As can be seen, the generation of this new backend involves defining the rules for building all the *PolyORB_HI_Generated* files described in the previous section, as well as a set of facilities to help in the development of automatically generated packages. Likewise, the *ocarina-backends* root file has also been modified to integrate and register the PolyORB-HI-Endpoints as a new Ocarina backend.

In relation to high-integrity distribution middleware, a set of modifications has been performed to allow the endpoints API to be accessed by PolyORB-HI. Furthermore, this middleware has been extended to provide two additional functionalities: firstly, the marshalling and unmarshalling primitives required to transmit the Event_Id parameter through communication networks (*polyorb_hi-event_id_marshallers* package); secondly, the operations responsible for the internal management of events by middleware (i.e., those subprograms defined by the *Event_Management* interface).

Furthermore, a new network service has been implemented and integrated into PolyORB-HI. This service adds two functionalities: firstly, it allows multiple receive ports to be used in a single protocol instance (for example, for connectionless networks); secondly, it allows handler tasks to wait directly in the network, thus avoiding decoupled communications.

Table 5-1: Modifications performed in Ocarina to create a new backend

FILES	PURPOSE
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-ACTIVITY.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-ACTIVITY.ADS	OPERATIONS TO BUILD THE POLYORB_HI_GENERATED.ACTIVITY PACKAGE THAT CONTAINS THE MAPPING OF TASKS AND PROTECTED OBJECTS
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS.ADS	ROOT UNIT FOR THE CODE GENERATOR FOR POLYORB-HI MIDDLEWARE
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-DEPLOYMENT.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-DEPLOYMENT.ADS	OPERATIONS TO BUILD THE POLYORB_HI_GENERATED.DEPLOYMENT PACKAGE THAT CONTAINS DEPLOYMENT INFORMATION ON THE DISTRIBUTED APPLICATION
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-MAIN.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-MAIN.ADS	OPERATIONS TO BUILD THE MAIN SUBPROGRAM CORRESPONDING TO EACH NODE
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-MAPPING.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-MAPPING.ADS	OPERATIONS TO MAP AADL ENTITIES INTO POLYORB-HI ENTITIES
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-MARSHALLERS.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-MARSHALLERS.ADS	OPERATIONS TO BUILD THE POLYORB_HI_GENERATED.MARSHALLERS PACKAGE THAT CONTAINS THE DATA MARSHALLERS AND UNMARSHALLERS
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-NAMING.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-NAMING.ADS	OPERATIONS TO BUILD THE POLYORB_HI_GENERATED.NAMING PACKAGE THAT CONTAINS THE INFORMATION CONTACT OF A GIVEN NODE
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-RUNTIME.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-RUNTIME.ADS	FACILITIES TO HANDLE THE ADA ENTITIES
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-SUBPROGRAMS.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-SUBPROGRAMS.ADS	OPERATIONS TO BUILD THE POLYORB_HI_GENERATED.SUBPROGRAM PACKAGE THAT CONTAINS THE MAPPING SUBPROGRAMS
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-TRANSPORT.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-TRANSPORT.ADS	OPERATIONS TO BUILD THE POLYORB_HI_GENERATED.TRANSPORT PACKAGE THAT CONTAINS THE MAPPING OF THE TRANSPORT FACILITIES
OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-TYPES.ADB OCARINA-BACKENDS-PO_HI_ADA_ENDPOINTS-TYPES.ADS	OPERATIONS TO BUILD THE PACKAGE POLYORB_HI_GENERATED.TYPES THAT CONTAINS THE MAPPING OF USER DATA

Finally, these modifications were incorporated into a prototype. This prototype implements the PolyORB-HI-Endpoints backend on a x86 architecture and a UDP-based network, as is illustrated in Figure 5-4. Although this platform is not appropriate to build high-integrity systems, here the objective is to develop a distributed real-time platform on which the endpoints pattern can be conceptually validated. As we said earlier, this prototype uses fixed-priority scheduling policies for both schedulable resources: processors and networks. The network uses the 802.1p specification [VBLAN06] to prioritize different message streams. Lastly, the real-time configuration file is currently generated by hand, although the automatic generation of the real-time configuration code can be performed by integrating the approach within a MDE development framework for high-integrity systems. This issue is dealt with next.

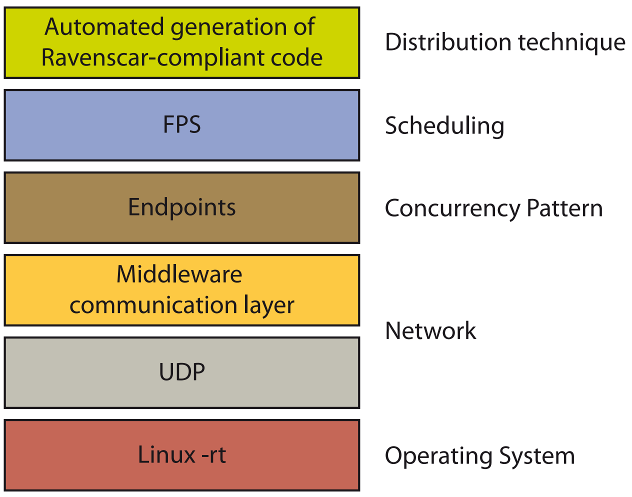

Figure 5-4: General architecture for the high-integrity distributed real-time platform based on PolyORB-HI

5.6 INTEGRATING THE ENDPOINTS PATTERN INTO A MODEL-DRIVEN DEVELOPMENT FRAMEWORK

A development framework for real-time systems must fulfil several conditions, such as the use of a predictable execution platform suitable for calculating the WCET accurately, an analyzable concurrency model and communication middleware that guarantees bounded transmission times. In the case of high-integrity systems, the first condition implies the use of specific processor architectures, such as the ERC32 [ATMEL05] or the LEON [LEON05] processors, and operating systems such as INTEGRITY-178B[1] or the open source kernel ORK+ [PUE00]. The Ravenscar profile and its proposed concurrency model have been designed to meet the second requirement, while the third condition can be fulfilled by any of the distribution models discussed in Chapter 2 or through the automatic generation of distribution code, as described in the previous section.

Schedulability analysis is a crucial activity in the development of high-integrity real-time systems. It allows software engineers to detect potential timing problems in early development phases, and take corrective actions on the system architecture in order to guarantee that the implementation will provide the required

1. INTEGRITY-178B is available at http://www.ghs.com/products/safety_critical

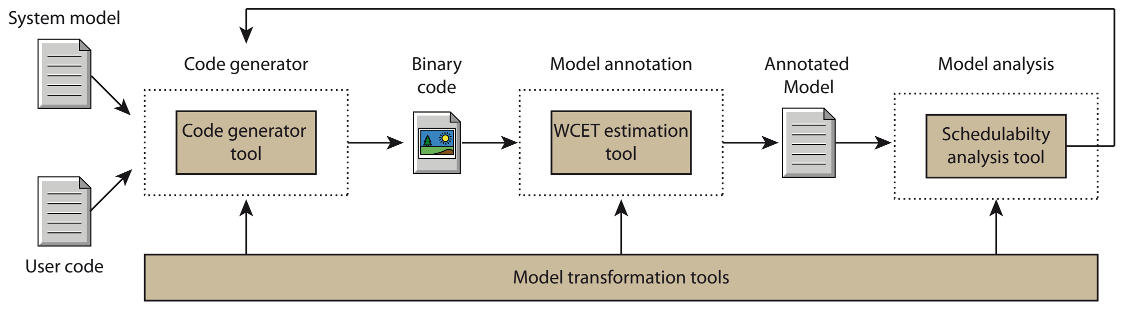

Figure 5-5: Development framework architecture for high-integrity distributed real-time systems

temporal behaviour. When an MDE approach is used, several properties can be verified by applying the corresponding analysis techniques over system models. However, the verification of temporal properties is more complex as it strongly depends on the execution platform. For this reason, the development framework should be flexible enough to allow the system to be reconfigured through an iterative process that may modify the system model according to the analysis results.

A model-driven development process for high-integrity systems can be represented by the architecture shown in Figure 5-5. This figure focuses exclusively on the timing analysis tools, and omits other kinds of analysis. In this case, the development process has been divided into three stages: code generator, model annotation and model analysis. First, the binary code is auto-generated from the model and the application code. In a second stage, the tools for WCET calculations are applied on the binary code and the result completes the timing view of our model (annotated model in Figure 5-5). Finally, the third stage can generate the real-time model on which to apply the schedulability analysis and obtain the scheduling parameters as shown in Figure 5-5. This step completes the system model and then the framework can automatically generate the source code and the real-time configuration for the application.

As we said earlier in Chapter 1, the project ASSERT (*Automated proof-based System and Software Engineering for Real-Time systems*) defines different model-driven strategies for the development of high-integrity systems [MAZ09] [PERR10]. The latter work includes a set of tools, called TASTE, which provides support for the ASSERT development process. Specifically, TASTE uses RapiTime[1] and Bound-T[2] tools for WCET calculations, Ocarina and a set of scripts for model transformations, the PolyORB-HI backend for automatic generation of source code, and MAST and Cheddar [SIN04] for modelling and schedulability analysis.

1. RapiTime is available at http://www.rapitasystems.com/rapitime
2. Bound-T is available at http://www.bound-t.com

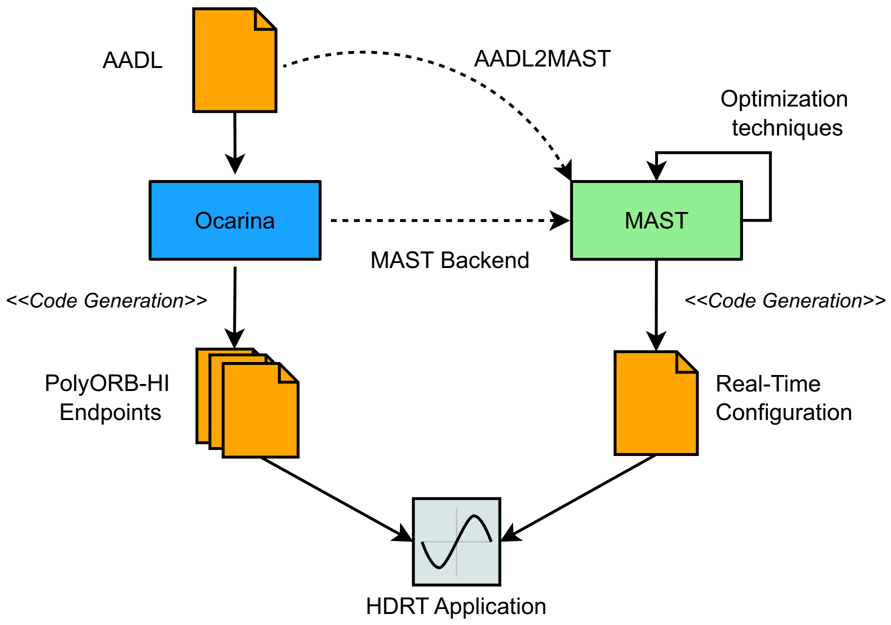

Figure 5-6: Ada toolchain for the development of high-integrity distributed real-time systems

Section 1.5 already discussed how model-driven strategies may benefit from adapting distribution middleware to the real-time model used. This would facilitate the development and integration of the different timing tools. Therefore, the set of tools included in TASTE seems an appropriate framework to incorporate the end-to-end flow model, using the PolyORB-HI-Endpoints backend for the automatic generation of source code and MAST for modelling and schedulability analysis

The proposed high-integrity distributed real-time platform can be extended to include the automatic generation of the real-time configuration code. This feature can be integrated into TASTE as illustrated in Figure 5-6. Under this approach, the MAST model can be auto generated from the annotated system model (for instance, with the WCET calculations from the RapiTime tool) through a new Ocarina backend or an external AADL2MAST conversion tool [PER11] and thus it provides the necessary support to perform an offline verification of the end-to-end deadlines. As a result of the analysis, a real-time configuration file is created containing all the parameters required. Finally, the application is generated by compiling the source code generated from the PolyORB-HI-Endpoints backend and the real-time configuration file. A more detailed description of this framework can be found in [PER11].

5.7 THE ENDPOINTS PATTERN AND STATIC DISTRIBUTED ADA

Section 5.2 proposed the adaptation to Ravenscar of the endpoints API designed to support the timing analysis of distributed real-time systems in full Ada. Although the changes suggested are compatible with the Ravenscar profile, the new approach may not be acceptable for static high-integrity systems due to the following issues:

- **The use of abstract types and generic units.** Although both Ada features are used in some research projects oriented to the development of high-integrity systems, neither of them are included in SPARK.
- **The use of a configuration interface.** While the Ravenscar profile seems to fit better into an absolutely static system, the use of the endpoints API provides a more dynamic nature: the system is static but only after the real-time configuration stage.

One of the most important advantages of adapting the endpoints API is that it represents a quite similar solution for full and restricted Ada. However, another kind of solution could fit better into this kind of systems. The next list briefly summarizes the minimum requirements for developing high-integrity distributed real-time systems following the end-to-end flow model. As a consequence, the configuration of the application can be divided into three stages:

1. **Configuring the partition of the program**, as required by the Ada DSA. The definition and implementation of this stage is implementation-defined.
2. **Identification and configuration of the schedulable entities**. This step should contain the following semantics:
   - A distributed real-time application defines two kinds of schedulable entities: the tasks for the processor, and the messages for the networks.
   - The implementation shall provide means for explicitly creating and configuring each schedulable entity with the appropriate scheduling parameters and associating the handler tasks to the appropriate receive endpoint.
3. **Identification and configuration of the different end-to-end flows.** This step should contain the following semantics:

- The implementation shall provide means for explicitly setting the initial event for application tasks. This can be achieved by defining a new pragma associated with tasks that identify the starting end-to-end flow.
- The implementation shall provide means for explicitly performing the event mapping.
- The implementation shall include the Event_Id parameter as part of the network message if a single handler task may process several requests matching different end-to-end flows.

Another possible approach could be to incorporate support for initialization-level configuration to Ada. This may include the definition of APIs which can only be used at certain stages (for example, during system startup and system shutdown), thus avoiding exposing the API to erroneous usage.

5.8 RELATED WORK

One of the main objectives of this chapter was to address how to build HDRT systems compatible with Ravenscar. This objective is of great interest to the community, as can be seen by the research works arising in the same line and sharing the same objective. According to the distribution model, those works can be classified as follows:

- **Ravenscar and Ada DSA**. These works are mainly focused on the adaptation of the Ada DSA to be Ravenscar compliant as discussed in [AUD01] and [URU11]. The main disadvantage of this option is the lack of a standard real-time distributed framework for Ada.
- **Ravenscar and a custom mechanism to perform the distribution.** Under this approach the HDRT systems are built by automatically generating source code from architectural descriptions (i.e. system models). A representative example is the tool suite Ocarina, software that has already been introduced in this chapter as the base platform to validate the proposed API for high-integrity systems.

Comparing the proposal in this thesis and these works, the most notable difference is that our approach does not only rely on the Ada DSA to perform the distribution, but it can be applied to different distribution models [PER09]. Nevertheless, it can be considered as a complementary work to [AUD01] and [URU11] since the end-to-end flow model provides a feasible architecture for performing a static timing analysis in which the timing requirements of the applications can be validated. Our approach does not deal with other important features required in high-integrity systems and detailed in these references, such as the coordinated elaboration of partitions or the bounded size of network messages. However, our real-time distributed model still provides support for some of the restrictions discussed in these works:

1. **The use of synchronous remote calls (RPCs).** According to [AUD01] and [URU11], the use of synchronous RPC complicates the schedulability analysis and must be prohibited for high-integrity systems. However, both the synchronous and asynchronous remote procedure calls provided by the DSA are predictable (given predictable low-level communications). In the case of RPCs, this kind of communication is amenable to schedulability analysis and current techniques have actually reduced the pessimism introduced in a wide range of scenarios. However, there are typical situations in distributed systems whose analysis can still be improved; for instance, the existence of simultaneous activations owning the same end-to-end flow (i.e. a linear end-to-end flow with the response time greater than the period). The endpoints pattern supports the use of RPCs by defining the reply endpoints.
2. **The use of concurrent remote calls.** According to [URU11], a remote operation should not be called while processing a past invocation of the same remote subprogram. This avoids the implementation of wait queues for each remote operation and thus facilitates the schedulability analysis. However, the buffering of incoming requests usually relies on the services provided by communication networks, such as in AFDX [AFDX09] or SpaceWire [SPW08]. Furthermore, the problem of dimensioning a wait queue to hold the incoming requests is already considered by the timing analysis techniques. Therefore our approach does not preclude the reception of concurrent remote calls.
3. **The use of nested RPCs.** [URU11] argues that the schedulability analysis can be simplified if a synchronous remote subprogram cannot perform another (blocking) remote call before returning to the caller. However, the end-to-end flow model is able to compute the waiting times associated with the nested remote calls and thus, from the real-time perspective, the response time analysis can be performed except when the response time of the end-to-end flow is greater than the period.
4. **Assignment of priorities for remote calls.** [URU11] differentiates how to configure the real-time aspects between Ada remote procedure calls and remote objects. For the latter, the author proposes three design strategies to specify their execution priority: *priority per object*,

priority per remote reference and *priority per tagged type*. In contrast to this approach, the *endpoints pattern* provides a uniform solution for both distribution paradigms, as they are equally configured through the proposed API.

Finally, HDRT systems usually require support for other safety-critical facilities that could also be adapted to the Ravenscar profile. For instance, the authors in [PIN02] and [PIN02-B] propose a framework for the development of fault-tolerant applications conforming to the Ravenscar profile. Although it is an interesting field for research, this kind of facilities falls beyond the scope of this thesis.

5.9 CONTRIBUTIONS OF THIS CHAPTER

Building high-integrity systems with timing requirements is a hard and complex task whose development is restricted by several specific standards. To simplify this process, the Ada programming language has defined the Ravenscar profile as a set of guidelines that restricts the concurrent part of the real-time application for single-processor systems. Furthermore, Chapter 3 defined an API that implements the *endpoints pattern* for Ada and thus it integrated the end-to-end flow model within the language for developing distributed real-time systems in Ada. Based on both aspects, this chapter has aimed to validate the use of the endpoints pattern over a high-integrity platform. To this end, the following actions have been performed:

- **Adaptation of the *endpoints pattern* to the Ravenscar profile.** Since the endpoints API proposed in Chapter 3 is not compatible with the Ravenscar profile, the adaptation has involved a set of modifications mainly related to the way in which handler tasks are created and assigned their scheduling parameters.
- **Proposals for integrating the *endpoints pattern* into restricted Ada.** We have defined two different proposals, both compatible with the Ravenscar profile. The first, which is based on a configuration interface, represents a homogeneous solution for full and restricted Ada; however, this solution must be applied during the real-time configuration stage. The second represents an even more static approach and lists the minimum requirements that must be implemented to support the end-to-end flow model, leaving the internal details open to implementations. This solution can be applied at compilation time and thus it can fit better into the development of high-integrity distributed real-time systems.

- **Implementation and validation of the proposed interface on a high-integrity platform.** To validate the use of the *endpoints pattern* with the Ravenscar profile, the restricted interface has been integrated into the Ocarina toolsuite. To this end, we have developed a prototype that implements an endpoints API compatible with Ravenscar which, together with the new PolyORB-HI-Endpoints backend and the modifications performed to middleware, have the following features:
    - *Automatic generation of source code.* The proposed approach has been seamlessly integrated within the Ocarina architecture by developing a new backend to automatically generate the source code based on the real-time end-to-end flow model
    - *Automatic management of events.* It comprises an extension of PolyORB-HI to provide marshalling and unmarshalling primitives for the Event_Id parameter and thus enable the use of shared handler tasks among several end-to-end flows.
    - *Fixed-priority scheduling for the processors.* According to the Ravenscar profile, it includes an implementation of the *Processing_Node_Scheduling* interface that has been developed for fixed-priority policy
    - *Fixed-priority scheduling for a new network service.* A new network service has been developed and integrated in PolyORB-HI to use the handler tasks to directly wait on the network for incoming requests, thus avoiding I/O decoupling. Furthermore, a fixed priority version has been implemented for the *Network_Scheduling* interface.
    - *Adaptation code to integrate the middleware internal characteristics into the proposed model.* Middleware built on top of the endpoints pattern requires some glue code to handle and map its internal structures consistently and integrate the management and utilization of the communication endpoints.
    - *Real-time configuration code.* As we discussed earlier in Chapter 3, the initialization code for end-to-end flows (for example, the creation of endpoints and handler tasks or the event mapping) may be specified through a simple configuration operation.

- **Description of a development framework for high-integrity distributed real-time systems.** Due to the complexity associated with the development of this kind of systems, the TASTE toolset has been explored to provide support for the development of distributed real-time systems compatible with the Ravenscar profile.

- **Study and analysis of the related work.** We reviewed the other solutions proposed to extend the use of the Ravenscar profile to distributed systems. Since our objective is to provide an analyzable real-time model, most of these solutions have features that complement our work and should be taken into account in the development of a Ravenscar profile for distributed systems.

5 ADAPTATION OF THE ENDPOINTS PATTERN TO HIGH-INTEGRITY DISTRIBUTED REAL-TIME SYSTEMS DEVELOPED IN ADA

CONCLUSIONS 6

This chapter summarizes the main contributions of this thesis and proposes the future work. Section 6.1 outlines the main actions undertaken throughout this thesis, while Section 6.2 details the specific contributions. Finally, Section 6.3 describes the possible lines of future work which arise from the results obtained.

6.1 THESIS OVERVIEW

This thesis aimed to make contributions in the field of distributed real-time systems. Current trends in the development of distributed systems include the use of middleware technology as a stand-alone tool or as a part of a MDE strategy. However, the latter option becomes particularly important in distributed real-time systems in which new requirements apply and the integration of middleware into an MDE strategy can enable the automatic configuration of certain critical parameters for real-time applications.

One of the major objectives of this thesis is the integration of a real-time model suitable for schedulability analysis into distribution middleware. This integration becomes particularly important in model-driven based approaches, as it facilitates the development process and the integration of the different timing tools. To this end, our approach has focused on how to adapt the end-to-end flow model, which is taken from scheduling theory and also from the MARTE modelling standard, to current distribution standards.

Firstly, we reviewed the main distribution standards oriented to real-time systems in order to identify the real-time facilities initially included by them for the management of processors and networks. In particular, we dealt with the analysis of the RT-CORBA, Ada DSA and DDS specifications, and some particular implementations (TAO, PolyORB and RTI-DDS, respectively). In practice, their real-time facilities were grouped in five categories *(scheduling policies, concurrency patterns, controlled access to shared resources, setting of scheduling parameters* and *support for the end-to-end flow model*) which motivated the identification of a set of features required for the development of analyzable applications regardless of the distribution model and / or standard used.

Taking this set of features as the basis, we presented a proposal to integrate the end-to-end flow model into distribution middleware called the *endpoints pattern*. It consists of a set of interfaces within distribution middleware that allows processors and communication networks to be configured explicitly. Then, we also addressed how to develop analyzable distributed real-time applications in Ada by proposing a specific implementation of the *endpoints pattern* for this programming language. Furthermore, a first step towards the integration of this pattern into model-driven development processes was performed by considering the automatic generation of the real-time configuration through the MAST modelling tool.

We have also focused on the implementation and validation of the *endpoints pattern* in a distributed real-time platform. To this end, the platform integrated several software components such as (1) PolyORB as a distribution middleware, (2) MaRTE OS as an operating system and (3) RT-EP and FRSH communication layer as communication networks. Two case studies were proposed to validate the *endpoints pattern* with different distribution models in static and dynamic scenarios, and these studies showed satisfactory results.

Finally, we explored the flexibility and generality of the *endpoints pattern* for use in high-integrity systems. For this purpose, we defined an endpoints API which was compatible with the Ravenscar profile. We also proposed a specific implementation of the endpoints API for restricted Ada. Furthermore, this deployment was integrated into a model-driven development framework for high-integrity systems.

6.2 CONTRIBUTIONS OF THE WORK

Our first step in this thesis was to report an analysis to determine whether the real-time mechanisms, included in distribution standards and their implementations, were enough to ensure determinism over the whole application. Based on this analysis, we isolated a set of features and objectives that all distribution standards for real-time systems and / or their implementations should incorporate:

- Control of remote calls by selecting an appropriate concurrency pattern and providing explicit mechanisms to assign the scheduling parameters.
- Enabling free assignment of scheduling parameters throughout the chain of entities that compose the end-to-end flow.
- Support for different scheduling policies to be easily adaptable to multiple scenarios.

- Bounding of the effect of priority inversion through the appropriate configuration of synchronization protocols and ensuring a maximum size for network messages.
- Documentation of the overhead introduced by implementations, even those entities created internally by middleware.
- Support for the end-to-end flow model to facilitate the application of CASE tools for analysis and optimization, as well as its integration into MDE strategies.
- Enabling schedulability analysis of the complete application by supporting the configuration of both processors and communications networks.

The process of integrating an analyzable real-time model into distribution middleware has been built upon this set of features. As a result, we have defined a set of entities and mechanisms required to integrate the end-to-end flow model into distribution middleware which was called the *endpoints pattern*. This pattern defines a set of interfaces within distribution middleware that allows processors and communication networks to be configured explicitly, supporting heterogeneous scheduling policies and parameters and providing the required mechanism to configure them.

One of the major features of the *endpoints pattern* is that it enables the logic of the application to be separated from the real-time aspects. Under our approach, the only requirement is simply to set the initial event that triggers the end-to-end flow (i.e. by including a *Set_Event_Id* call within the application code), while the rest of the end-to-end flow elements can be described as a part of a configuration operation. Although the configuration of a real-time application using the *endpoints pattern* could be seen as a complex process, it can be performed automatically by applying MDE techniques based on the end-to-end flow model. For this purpose, this thesis presented an analysis to integrate the *endpoints pattern* with MAST. This analysis identified the additional properties that should be added to MAST to enable the integration, as well as the set of rules required to generate the real-time configuration code automatically.

As a complementary facility, this thesis was considered which concurrency pattern should be selected to be used with the *endpoints pattern*. Our analysis showed that none of the most common concurrency patterns used in distribution middleware can satisfy some of the usual requirements for hard real-time systems, so this motivated the definition of a new concurrency pattern called RATO. The benefits of applying RATO include avoiding unnecessary context switches and the dynamic update of scheduling parameters, while it is still compatible with the *endpoints pattern.*

In order to validate the features of the *endpoints pattern* and its usability, a distributed real-time platform capable of providing strict temporal guarantees was implemented. Among other features, this platform was developed to provide support for the end-to-end flow model within distribution middleware, interchangeable scheduling policies such as FPS, FRESCOR contracts and EDF, and a set of extensions to enhance the system predictability. These extensions include (1) porting to a new real-time operating system (MaRTE OS), (2) a new communication layer to use different real-time protocols such as RT-EP or CAN, and (3) a new internal management of middleware tasks by defining new tasking and controller policies and profiles.

The proposal was evaluated in two case studies. The first study dealt with the migration of an industrial robotic distributed system to our platform using the *endpoints pattern*. This distributed real-time system was implemented using different distribution models (RPCs and distributed objects) to evaluate the complexity of using our approach, resulting in a minimal impact in the application code (less than 1%). The second case study implemented a software tool responsible for the distributed negotiation of new end-to-end flows, thus checking the capability of our approach to be adapted to environments whose workload is variable. In this case, the study showed that our approach can provide temporal guarantees even when new end-to-end flows are being negotiated in the system.

Another important aspect of this thesis is the adaptation of the *endpoints pattern* to high-integrity systems. In particular, we modified the *endpoints pattern* to make it compatible with the Ravenscar profile by reviewing how handler tasks are created and how their scheduling parameters are assigned. Furthermore, this adaptation also considered the restrictions imposed by other high-integrity facilities, such as SPARK or GA-HI. As a result, a new endpoints API was defined for use with this kind of systems. The resulting approach has been validated by implementing a high-integrity distributed real-time platform. The Ocarina toolsuite was taken as a starting point for supporting the approach. This toolsuite has been extended with a new backend called PolyORB-HI-Endpoints and new network services. Due to these modifications, the platform developed was able to integrate the end-to-end flow model with the automatic generation of Ravenscar-compliant source code and distribution middleware.

For the high-integrity distributed real-time platform, the endpoints configuration may also be specified through a simple configuration operation. In this case, this thesis explored the use of the TASTE toolset to provide complete support for a MDE strategy for high-integrity systems, using the PolyORB-HI-Endpoints backend for the automatic generation of source code and MAST for modelling and schedulability analysis.

Finally, the last aspect we want to highlight is the development of a specific implementation of the *endpoints pattern* for Ada as a solution to the lack of support for the development of distributed real-time systems using this programming language. Furthermore, this thesis has also defined two different proposals for integrating the *endpoints pattern* into restricted Ada. The first, which is based on a configuration interface, represents a homogeneous solution for full and restricted Ada; however, this solution must be applied during the real-time configuration stage. The second represents an even more static approach and lists the minimum requirements that must be implemented to support the end-to-end flow model, leaving the internal details open to implementations. This solution can be applied at compilation time and thus it can fit better into the development of high-integrity distributed real-time systems.

6.3 FUTURE WORK

The *endpoints pattern* has been developed in response to the need of add a real-time model in distribution middleware, thus facilitating the integration of both into model-driven development processes. This pattern represents one of the steps towards the definition of a complete development framework supporting the verification of temporal properties over system models. However, there are still open research lines in this field. These are briefly described below.

Although distribution standards play a central role in the current development of distributed real-time systems, they usually provide little or no support for the end-to-end flow model. Based on the analysis presented in this thesis, we can conclude that the distribution models proposed by DSA, RT-CORBA and DDS are appropriate to integrate the concept of end-to-end flow. However, these standards also support advanced configurations that require further analysis. For example, both RT-CORBA and DDS support different mechanisms to provide a wide range of QoS guarantees, whose representation using the end-to-end flow model should be further investigated. In the case of DDS, there are still several aspects to be analysed that may affect the determinism of the distributed system, for instance, the influence of the built-in entities, dynamic systems or the discovery process, as has been indicated in this thesis.

Furthermore, the *endpoints pattern* can be extended in a number of ways. Although it defines a real-time configuration stage to create and configure the schedulable entities explicitly, it is necessary to deal with other relevant aspects of distributed systems such as their coordinated initialization during the start-up. This is not strictly part of the real-time model and therefore it was beyond the scope of this thesis, but this aspect represents a key element of these systems and should be further investigated.

Since the integration of our distributed real-time platform into MDE strategies has been developed as a proof of concept, other plans for the near future include completing the platform with those aspects that were outside the initial prototype. In this case, this would require:

- Developing or adopting a platform to build distributed real-time systems using MDE strategies based on the MARTE standard in order to integrate the middleware technology developed.

- Obtaining the complete real-time model of distribution middleware. This real-time model would allow an *a priori* schedulability analysis to be performed that is suitable for any application using this distribution middleware. For instance, this could imply obtaining the MAST model of PolyORB and PolyORB-HI.
- Completing the integration of timing analysis tools (e.g., for schedulability analysis and WCET estimation) into the MDE-based platform to provide the necessary support to perform the verification of end-to-end deadlines.
- Providing support for the automatic generation of the real-time configuration. In the case of MAST, this would require incorporating the changes proposed in this thesis (e.g., the identification of the nature of tasks and messages) in order to generate the endpoints configuration.

Additionally, if the MDE representation strategy is not based on the MARTE standard, the MDE platform should provide model transformation tools to generate the end-to-end flow real-time model. For example, the MAST model could be directly auto generated from an AADL textual representation.

Regarding high-integrity systems, traditionally this kind of system does not consider the use of distribution standards, because it involves the use of a middleware layer which makes the certification process more complex. However, there are some efforts in this direction; for example, a DDS profile for safety-critical systems is being discussed at current meetings of the OMG. This opens an important research field to apply communication middleware based on distribution standards over, for example, ARINC partitioned systems.

Although several approaches have been proposed to solve the problem of developing distributed real-time systems compatible with the Ravenscar profile, the definition of a generic solution for full and restricted Ada would be desirable and should be further investigated. For example, this could be done by exploring how to incorporate support for initialization-level configuration into Ada.

Finally, the integration of the Distributed Systems Annex and the Real-Time Annex of Ada is an interesting task that can be addressed by using our proposal as a starting point. In the same line, the definition of a Distributed Ravenscar profile could be another important objective for future work.

A RELATED RESEARCH PROJECTS

The research work included in this thesis has been funded by the IST Programme of the European Commission under project FP6/2005/IST/5-034026 (FRESCOR), by the Spanish CDTI under project CEN2009-1048 (ENERGOS) and by the Spanish Ministry of Science and Technology under grants number TIC2005-08665-C03-02 (THREAD), TIN2008-06766-C03-03 (RT-MODEL), and TIN2011-28567-C03-02 (HI-PARTES). This annex introduces these research projects and their relationship with this thesis.

1 THREAD PROJECT

THREAD (Integral support for embedded, distributed open real-time systems) is a research project funded by the Spanish Ministry of Science and Technology (2005 - 2008). The major objective of this project was to provide integral support for the development of distributed real-time embedded systems which will include a family of interoperable platforms, their connection mechanisms, the applicable architecture and design methodologies, and the application domains of the new generation of this kind of systems. This integral support will deal with all the levels from the operating system and the networks, through the communications and quality of service management middleware, up to the application level.

Within the context of this project, this thesis has contributed to the development of communication middleware for distributed real-time embedded systems with capabilities to provide deterministic communications between subsystems, even when they are being executed in different environments and have been developed using different methodologies, communication protocols and programming languages.

2 FRESCOR PROJECT

The FRESCOR (Framework for Real-time Embedded Systems based on COntRacts) EU project, which is funded by the European Union’s Sixth Framework

Programme (2006 - 2009), had the objective of providing engineers with a scheduling framework that represents a high-level abstraction that lets them concentrate on the specification of the application requirements, while the system transparently uses advanced real-time scheduling techniques to meet those requirements.

The approach to achieve this main objective was to integrate advanced flexible scheduling techniques directly into an embedded systems design methodology, covering all the levels involved in the implementation, from the operating system primitives, through the middleware, up to the application level. This was achieved by creating a contract model that specifies the application requirements with respect to the flexible use of the processing resources in the system, and also the resources that must be guaranteed if the component is to be installed in the system. Moreover, it explains how the system can distribute any spare capacity that it has, to achieve the highest usage of the available resources.

Contributions of this thesis include the development of a distributed real-time platform for testing purposes and the integration of middleware technology into the framework in order to provide distribution services within a flexible scheduling environment.

3 RT-MODEL PROJECT

RT-MODEL (Real-time platforms for model-driven design of embedded systems) is a research project funded by the Spanish Ministry of Science and Technology (2009 - 2011). This project was aimed at extending model-driven software development methods (MDA/MDE) to computer systems with non-functional requirements, such as real-time, quality of service and high-integrity. Such systems were assumed to execute on distributed, heterogeneous embedded platforms with standard middleware and underlying services.

Within the context of this project, this thesis has contributed to the development of a specific middleware and platform for high-integrity distributed real-time systems. These technologies must be able to provide services and resources that support the execution of applications built from high-level system models, as well as mechanisms to simplify the configuration of these platforms using CASE tools. Furthermore, this thesis also explored the real-time modelling of data-centric middleware.

4 ENERGOS PROJECT

The ENERGOS (Technologies for the automatic and intelligent management of future distribution networks) project, which is funded in part by the CENIT Fifth Framework Programme (2009 - 2012), has the objective of developing technologies and knowledge to move towards the implementation of future electrical distribution networks (*smart grid*). This network is much more complex than the present distributed networks due to distributed energy generation (based on renewable energy sources) and the automation required to optimise energy consumption.

Within the context of this project, this thesis has contributed to the analysis and adaptation of middleware technologies capable of providing real-time guarantees for new smart grids.

5 HI-PARTES PROJECT

HI-PARTES (High-integrity partitioned embedded systems) is a research project funded by the Spanish Ministry of Science and Innovation (2012 - 2015). The main aim of this project is to contribute to the improvement of technology for the development and execution of high-integrity embedded systems, which currently show a continuous increase in their complexity. The development of high-integrity systems requires a number of additional activities, such as system partitioning, partition configuration, or global response time analysis. Model-driven architecture (MDA) is a suitable basis for providing integrated support for these activities. This technology allows the abstraction level of development languages and tools to be raised, and the information and processing logic to be isolated from the aspects related to the implementation technology and the execution platform.

From the experience and results obtained in this thesis, our work will contribute to this project with the study and characterization of communication middleware based on distribution standards for partitioned systems.

FRESCOR ADA BINDINGS

B

This annex introduces the Ada bindings for the FRESCOR framework. These Ada packages provide thin, direct Ada bindings to the most important data types and subprograms of the FRESCOR API.

1 FRESCOR DATA TYPES

The following package, which is called *frsh_ada_types.ads*, defines the bindings for Ada of most of the data types used in the FRESCOR framework, including those related to the definition of contracts and schedulable entities.

Listing B-1: Package frsh_ada_types.ads

```
--  -----------------      FRSH Mapping      ----------------------- --
----------------------------------------------------------------------------
--                      Copyright (C) 2006-2009
--                    Universidad de Cantabria, SPAIN
--                       http://www.ctr.unican.es/

--  This package contains the Ada interface to main procedures and functions
--  used in FRESCOR Framework

with Interfaces.C;
with MaRTE.Timespec;
with System;

package Frsh_Ada_Types is

   package Time_MaRTE_C_Types renames MaRTE.Timespec;
   package C_Types            renames Interfaces.C;

   --  FRESCOR TYPES  --

   --  Kind of schedulable entities
   type Resource_Type is (FRSH_RT_PROCESSOR, FRSH_RT_NETWORK, FRSH_RT_MEMORY,
                     FRSH_RT_DISK);
   --  Must follow the same value
   for Resource_Type use
     (FRSH_RT_PROCESSOR => 0, FRSH_RT_NETWORK => 1,
      FRSH_RT_MEMORY => 2, FRSH_RT_DISK => 3);

   --  Must follow C enumeration size (integer)
   pragma Convention (C, Resource_Type);

   --  Virtual resource identification
   type VRES_ID is new C_Types.unsigned;

   --  Endpoint identification
   subtype Endpoint_Id is C_Types.int;

   --  Pointers to message buffer
   subtype Pvoid is System.Address;

   --  Initial code to be executed by a newly created thread
   type Initial_Thread_Code is not null access procedure (Arg : in Pvoid);
   pragma Convention (C, Initial_Thread_Code);

   --  Parameterless procedure to follow PolyORB restrictions
   type FRSH_Thread_Code is not null access procedure;

   --  Timespec definition compatible with posix one.
   subtype Timespec is Time_MaRTE_C_Types.Timespec;

   --  Preemption level priority definition
   subtype Preemption_Level is C_Types.unsigned_long;

   --  Instead of mapping all contract parameters, we introduce a identificator
   --  and we manage them in C wrapper
   type Contract_Label is new C_Types.char_array (1 .. 16);

   --  Mapping of contract variable
   --     /** Utilization (C, T, and D) **/
   type Frsh_Utilization_T is record
      Budget   : Timespec;    -- Execution time
      Period   : Timespec;    -- Period
      Deadline : Timespec;    -- Deadline
   end record;
   pragma Convention (C, Frsh_Utilization_T);

   --  /**
   --   * Maximum number of utilization values (pairs of budget and period)
   --   * that can be stored in a contract parameter object
   --   **/
   --  Configuration parameter in frsh
```

```
FRSH_MAX_N_UTILIZATION_VALUES : constant := 5;

--  /** List of utilization values **/
type Frsh_Utilization_T_Array is array
  (0 .. FRSH_MAX_N_UTILIZATION_VALUES) of Frsh_Utilization_T;
type Frsh_Utilization_Set_T is record
   Size          : C_Types.int; -- = 0
   Utilizations  : Frsh_Utilization_T_Array;
end record;
pragma Convention (C, Frsh_Utilization_Set_T);

--     /** Kind of workload expected in vres: bounded or indeterminate **/
type Frsh_Workload_T is (FRSH_WT_BOUNDED,
                         FRSH_WT_INDETERMINATE,
                         FRSH_WT_SYNCHRONIZED);
--  Must follow the same value
for Frsh_Workload_T use
  (FRSH_WT_BOUNDED       => 0,
   FRSH_WT_INDETERMINATE => 1,
   FRSH_WT_SYNCHRONIZED  => 2);

--  Must follow C enumeration size (integer)
pragma Convention (C, Frsh_Workload_T);

--  /** Kind of contract: regular, background or dummy **/
type Frsh_Contract_Type_T is (FRSH_CT_REGULAR,
                              FRSH_CT_BACKGROUND,
                              FRSH_CT_DUMMY);
--  Must follow the same value
for Frsh_Contract_Type_T use
  (FRSH_CT_REGULAR       => 0,
   FRSH_CT_BACKGROUND    => 1,
   FRSH_CT_DUMMY         => 2);

--  Must follow C enumeration size (integer)
pragma Convention (C, Frsh_Contract_Type_T);

--  Mapping of a C union
type Signal_Access is (Value, Pointer);
type Frsh_Signal_Info_T (Option : Signal_Access := Value) is
   record
      case Option is
         when Value =>
            Sival_Int : C_Types.int;
         when Pointer =>
            Sival_Ptr : Pvoid;
      end case;
   end record;

```

```
pragma Unchecked_Union (Frsh_Signal_Info_T);
pragma Convention (C, Frsh_Signal_Info_T);

-- /** Granularity of spare capacity requirements: continuous or discrete **/
type Frsh_Granularity_T is (FRSH_GR_CONTINUOUS, FRSH_GR_DISCRETE);
-- Must follow the same value
-- Values not defined in FRSH. Take default one -- TO_BE_REVISED
for Frsh_Granularity_T use
  (FRSH_GR_CONTINUOUS      => 0,
   FRSH_GR_DISCRETE        => 1);
-- Must follow C enumeration size (integer)
pragma Convention (C, Frsh_Granularity_T);

-- /**
--  * Critical section data
--  * - comon parameters
--  *    op_kind;     // kind of operation (READ or WRITE)
--  *    obj_handle;  // handle to shared object
--  *    wcet;        // Execution time
--  *    blocking;    // Blocking time (execution time + protection overheads)
--  * - attributes used only for protected shared objects
--  *    op;          // pointer to the operation
--  * - attributes used only for protected write operations
--  *    areas;       // memory areas to be protected
--  *
--  **/

-- /**
--  * Kind of protected operation: read, write or unchecked
--  **/
type Frsh_Csect_Op_Kind_T is (FRSH_CSOK_UNCHECKED, FRSH_CSOK_READ,
                              FRSH_CSOK_WRITE);
   -- Must follow the same value
for Frsh_Csect_Op_Kind_T use
  (FRSH_CSOK_UNCHECKED     => 0,
   FRSH_CSOK_READ          => 1,
   FRSH_CSOK_WRITE         => 2);

-- Must follow C enumeration size (integer)
pragma Convention (C, Frsh_Csect_Op_Kind_T);

subtype Frsh_Sharedobj_Handle_T is C_Types.int;
--  /**
--   *  Pointer to protected operation, which takes a pointer to
--   *  the input parameters, and a pointer to the output
--   *  parameters; the user is responsible for not exceeding the
--   *  sizes of the respective input and output parameters data structures
--   */
type Frsh_Csect_Op_T is access procedure (Input_Arg  : in Pvoid;
                                Output_Arg : out Pvoid);
pragma Convention (C, Frsh_Csect_Op_T);

--  /**
--   *  A memory area
--   */
type Frsh_Memory_Area_Data_T is record
   Size : C_Types.int;  --  Size_t
   Area : Pvoid;
end record;
pragma Convention (C, Frsh_Memory_Area_Data_T);

-- /**
--   * Maximum number of memory areas that can be specified for a
--   * write operation in a critical section
--   **/
-- Defined in frsh configuration parameters
FRSH_MAX_N_MEMORY_AREAS  : constant := 4;

-- /**
--   * Memory areas container
--   **/
type Frsh_Memory_Area_Data_T_Array is array
  (0 .. FRSH_MAX_N_MEMORY_AREAS) of Frsh_Memory_Area_Data_T;
type Frsh_Memory_Areas_T is record
   Size         : C_types.int; -- = 0
   Memory_Areas : Frsh_Memory_Area_Data_T_Array;
end record;
pragma Convention (C, Frsh_Memory_Areas_T);

type Frsh_Csect_T is record
   Op_Kind            : Frsh_Csect_Op_Kind_T;
   Obj_Handle         : Frsh_Sharedobj_Handle_T;
   Wcet               : Timespec;
   Blocking           : Timespec;
   Op                 : Frsh_Csect_Op_T;
   Areas              : Frsh_Memory_Areas_T;
   Storage            : Frsh_Memory_Areas_T;
end record;
pragma Convention (C, Frsh_Csect_T);

-- /**
--   * Maximum number of critical sections that can be stored in a
--   * contract parameter object
--   **/
-- Defined in FRSH Configuration parameters
FRSH_MAX_N_CRITICAL_SECTIONS : constant := 10;

-- /**
--   * Container of a group of critical sections, up to a maximum size
--   **/
type Frsh_Csect_T_Array is array
  (0 .. FRSH_MAX_N_CRITICAL_SECTIONS) of Frsh_Csect_T;
type  Frsh_Csects_Group_T is record
   Size   : C_Types.int;        --  size of the group; initially=0
   Csects : Frsh_Csect_T_Array; --  array of csect
end record;

--     /** Scheduling policies **/
type Frsh_Sched_Policy_T is (FRSH_FP, FRSH_EDF, FRSH_TABLE_DRIVEN,
                             FRSH_RR, FRSH_NONE);
for Frsh_Sched_Policy_T use
  (FRSH_FP => 0, FRSH_EDF => 1, FRSH_TABLE_DRIVEN => 2,
   FRSH_RR => 3, FRSH_NONE => 4);
--  Must follow C enumeration size (integer)
pragma Convention (C, Frsh_Sched_Policy_T);

--   /**
--   * Extra protocol dependent opaque information for the application.
--   * It can be used in different places: contract negotiation, extra
--   * endpoint info, extra status info...
--   **/
type Frsh_Protocol_Info_T is record
   Body_Ptr : PVoid;
   Size     : C_Types.int;
end record;

--  /**
--   * Algorithm used when the queue is full to choose the message to reject
--   **/
type Frsh_Queue_Rejection_Policy_T is
  (
   --     /** A new message is admitted rejecting the oldest message in the
   --         queue to make room for the newcomer **/
   FRSH_QRP_OLDEST,
   --     /** Incoming messages are rejected if the queue is full **/
   FRSH_QRP_NEWCOMER);

--  Must follow the same value
--  Values not defined in FRSH. Take default one --  TO_BE_REVISED
for Frsh_Queue_Rejection_Policy_T use
  (FRSH_QRP_OLDEST       => 0,
   FRSH_QRP_NEWCOMER        => 1);

--  Must follow C enumeration size (integer)
pragma Convention (C, Frsh_Queue_Rejection_Policy_T);

--  /**
--  * Queing information for endpoints
--  **/
type Frsh_Endpoint_Queueing_Info_T is record
   Queue_Size : C_Types.int;  --  /** Size 0 means that there is no queue **/
   Queue_Policy : Frsh_Queue_Rejection_Policy_T;
end record;

type Contract is record
   --    /** frsh_contract_parameters_t **/
   --   /** Processor Id or Network Id **/                    '
   Resource_Id             : C_Types.unsigned_short;
   --        /** Whether processor or network **/                    '

   --   /** Maximum period that the system system can sustain **/   '
   Period_Max              : Timespec;

   --   /** Signal parameters for the case of
   --       attempting to use too much budget       **/            '

   Budget_Overrun_Siginfo    : Frsh_Signal_Info_T;

   --   /** Signal parameters for the case a deadline
   --      is missed **/                                 '
   Deadline_Miss_Siginfo     : Frsh_Signal_Info_T;

   Queueing_Info            : Frsh_Endpoint_Queueing_Info_T;

   --   /** Maximum loss rate
   --       Percentage of packet loss in the network that is
   --       tolerated by the application **/                   '
   Max_Loss_Rate            : C_Types.int;
end record;
pragma Convention (C, Contract);

end Frsh_Ada_Types;
```

2 FRESCOR PROGRAMMING INTERFACE

The following package, which is called *frsh_ada.ads*, defines the bindings for Ada of most of the API operations defined in the FRESCOR framework, including those related to the creation and management of contracts.

Listing B-2: Package frsh_ada.ads

```
--  -----------------       FRSH Mapping      ------------------------ --
---------------------------------------------------------------------------
--                        Copyright (C) 2006-2009
--                    Universidad de Cantabria, SPAIN
--                         http://www.ctr.unican.es/

--  This package contains the Ada interface to main procedures and functions
--  used in FRESCOR Framework

with Interfaces.C;

with MaRTE.Timespec;       --  MaRTE definition and functions of type "timespec"
with MaRTE.Integer_Types;  --  MaRTE definiton of C Integer types

with Interfaces.C.Strings;
with Ada.Streams;
with System;

with Frsh_Ada_Types; use Frsh_Ada_Types;

package FRSH_Ada is

   package MaRTE_C_Types      renames MaRTE.Integer_Types;
   package C_Types            renames Interfaces.C;
   package String_C_Types     renames Interfaces.C.Strings;
   package Time_MaRTE_C_Types renames MaRTE.Timespec;

   ---------------------------------
   -- FRSH_Initialize --
   ---------------------------------
   --  FRESCOR Framework init function

   procedure FRSH_Initialize (Status : out C_Types.int);
   pragma Import (C, FRSH_Initialize, "frsh_init");
   pragma Import_Valued_Procedure
     (Internal        => FRSH_Initialize,
      External        => "frsh_init",
      Mechanism       => (Value));

   --  FUNCTIONS  --

   -------------------------------
   -- Contract_Init --
   -------------------------------

   procedure Contract_Init (Status      : out C_Types.int;
                            My_Contract : in System.Address);
   pragma Import (C, Contract_Init, "frsh_contract_init");
   pragma Import_Valued_Procedure
     (Internal        => Contract_Init,
      External        => "frsh_contract_init",
      Mechanism       => (Value));

   -----------------------------------------------
   -- Contract_Set_Resource_And_Label --
   -----------------------------------------------

   procedure Contract_Set_Resource_And_Label
     (Status      : out C_Types.int;
      My_Contract : in System.Address;
      Res_Type    : in Resource_Type;
      CPU         : in Integer;  --  C_TYpes.Int mejor, no??? TO DO
      Label       : in Contract_Label);

   pragma Import (C, Contract_Set_Resource_And_Label,
                  "frsh_contract_set_resource_and_label");
   pragma Import_Valued_Procedure
     (Internal        => Contract_Set_Resource_And_Label,
      External        => "frsh_contract_set_resource_and_label",
      Mechanism       => (Value));

   ------------------------------------------
   -- Contract_Set_Basic_Params --
   ------------------------------------------

   procedure Contract_Set_Basic_Params
     (Status        : out C_Types.int;
      My_Contract   : in System.Address;
      B_Min         : in Timespec;
      P_Max         : in Timespec;
      Workload_Type : in Frsh_Workload_T := FRSH_WT_INDETERMINATE;
      Contract_Type : in Frsh_Contract_Type_T := FRSH_CT_REGULAR);

   pragma Import (C, Contract_Set_Basic_Params,
                  "frsh_contract_set_basic_params");
   pragma Import_Valued_Procedure
     (Internal        => Contract_Set_Basic_Params,
      External        => "frsh_contract_set_basic_params",
      Mechanism       => (Value));

   ---------------------------------------------
   -- Contract_Set_Preemption_Level --
   ---------------------------------------------

   procedure Contract_Set_Preemption_Level
     (Status      : out C_Types.int;
      My_Contract : in System.Address;
```

```
     Priority    : in Preemption_Level);
   pragma Import (C, Contract_Set_Preemption_Level,
                  "frsh_contract_set_preemption_level");
   pragma Import_Valued_Procedure
     (Internal        => Contract_Set_Preemption_Level,
      External        => "frsh_contract_set_preemption_level",
      Mechanism       => (Value));

   -----------------------------------
   -- Contract_Negotiate --
   -----------------------------------

   procedure Contract_Negotiate
     (Status      : out C_Types.int;
      My_Contract : in  System.Address;
      VRes        : out VRES_ID);

   pragma Import (C, Contract_Negotiate, "frsh_contract_negotiate");
   pragma Import_Valued_Procedure
     (Internal        => Contract_Negotiate,
      External        => "frsh_contract_negotiate",
      Mechanism       => (Value));

   ------------------------------------------
   -- Contract_Renegotiate_Sync --
   ------------------------------------------

   --  The operation renegotiates a contract for an existing vres

   procedure Contract_Renegotiate_Sync
     (Status      : out C_Types.int;
      My_Contract : in System.Address;
      VRes        : in VRES_ID);
   pragma Import (C, Contract_Renegotiate_Sync,
                  "frsh_contract_renegotiate_sync");
   pragma Import_Valued_Procedure
     (Internal        => Contract_Renegotiate_Sync,
      External        => "frsh_contract_renegotiate_sync",
      Mechanism       => (Value));

   ---------------------------------------
   -- Thread_Create_And_Bind --
   ---------------------------------------

   --  Create FRSH Thread and associate it with a Vres and an execution code
   --  This operation creates a thread and binds it to an existing vres.
   --  This is the preferred way to add threads to the application because
   --  we make sure that the thread won't become unbound.
   --
```

```
--  TO DO
--  Thread Attributes not mapped
--  MaRTE issue: Thread new code termination not supported by OS by now.

--    procedure Thread_Create_And_Bind
--      (Status      : out C_Types.int;
--       VRES        : in VRES_ID;
--       Thread_ID   : out System.Address;
--       Thread_Code : in Initial_Thread_Code;
--       Arg         : in Pvoid);
--
--    pragma Import (C, Thread_Create_And_Bind, "frsh_thread_create_and_bind");
--    pragma Import_Valued_Procedure
--      (Internal        => Thread_Create_And_Bind,
--       External        => "frsh_thread_create_and_bind",
--       Mechanism       => (Value));

procedure Thread_Create_And_Bind
  (VRES        : in VRES_ID;
   Thread_ID   : out System.Address;
   Thread_Code : in Initial_Thread_Code;
   Arg         : in Pvoid);

pragma Import (C, Thread_Create_And_Bind, "frsh_thread_create_and_bind_c");

-------------
-- Network --
-------------

----------------
-- Send_Async --
----------------

--  This operation sends a message stored in msg and of length size through
--  the given endpoint.
--  The operation is non-blocking and returns immediately.

procedure Send_Async
  (My_Endpoint  : in MaRTE_C_Types.Int;
   Message      : in Ada.Streams.Stream_Element_Array;
   Message_Size : in C_Types.size_t);

pragma Import (C, Send_Async, "frsh_send_async_c");

pragma Import_Procedure
  (Internal        => Send_Async,
   External        => "frsh_send_async_c",
   Mechanism       => (Value, Reference, Value));

```

```
------------------------------
-- Receive_Sync --
------------------------------
--  If there are no messages available in the specified receive endpoint
--  this operation blocks the calling thread waiting for a message to be
--  received.
--  When a message is available, if its size is less than or equal to the
--  buffer_size, the function stores it in the variable pointed to by buffer
--  and puts the number of bytes received in the variable pointed to by
--  message size.

procedure Receive_Sync
  (Station      : out C_Types.unsigned_short;
   My_Endpoint  : in C_Types.int;
   Message      : out Ada.Streams.Stream_Element_Array;
   Buffer_Size  : in C_Types.size_t;
   Message_Size : out C_Types.size_t);

pragma Import (C, Receive_Sync, "frsh_receive_sync_c");
--  This pragma assures a valid exchange of parameters. Target function must
--  return the first parameter.
pragma Import_Valued_Procedure
  (Internal        =>  Receive_Sync,
   External        => "frsh_receive_sync_c",
   Mechanism       => (Value,
                       Value,
                       Reference,
                       Value,
                       Reference));

----------------------------------------
-- Network_Bytes_To_Budget --
----------------------------------------

--  This operation converts a number of bytes into a temporal budget for a
--  specific network. Network overheads are not included here but are
--  considered internally when negotiating a specific contract.

procedure Network_Bytes_To_Budget
  (Network_ID : in C_Types.int;
   Bytes      : in C_Types.int;
   Budget     : out Timespec);

-------------------------------------
-- Send_Endpoint_Create --
-------------------------------------

--  This operation creates a unidirectional stream input endpoint through
--  which, after the corresponding binding, it is possible to send data to a
--  unicast or multicast destinations.
--
--  TO DO
--  Protocol info not mapped

procedure Send_Endpoint_Create
  (Network_ID : C_Types.int;
   To_Station : C_Types.int;
   To_Channel : C_Types.int;
   Endpoint   : out C_Types.int);

-----------------------------------------
-- Receive_Endpoint_Create --
-----------------------------------------

--  This operation creates a receive endpoint associated with a
--  undirectional
--  stream within a network interface of the node.
--  Receiving endpoints are not bound to any network vres, this is because
--  they don't originate any traffic.
--
--  TO DO
--  Protocol info not mapped
--  Queueing info not mapped

procedure Receive_Endpoint_Create
  (Network_ID : C_Types.int;
   Channel    : C_Types.int;
   Endpoint   : out C_Types.int);

------------------------------------
-- Send_Endpoint_Bind --
------------------------------------

--  This operation associates a send endpoint with a network vres, which
--  means that messages sent through this endpoint will consume the vres's
--  reserved bandwidth and its packets will be sent according to the
--  contract established for that vres.

procedure Send_Endpoint_Bind
  (VRES     : VRES_ID;
   Endpoint : C_Types.int);

pragma Import (C, Send_Endpoint_Bind, "frsh_send_endpoint_bind_c");

------------------------------------------
-- Thread_Join_In_Background --
------------------------------------------

```

```
--  This function creates a "background contract" that does not need to
--  be negotiated, associating the calling task with this contract and binding it
--  to the new vres.

procedure Thread_Join_In_Background
  (Resource_Id : in Integer;
   Res_Type    : in Resource_Type;
   Label       : in Contract_Label;
   VRES        : out VRES_ID);

-------------------------------
-- Get_My_CPU_Id --
-------------------------------
--  Recover FRSH_CPU_ID_DEFAULT Variable

procedure Get_My_CPU_Id (Id : out C_Types.int);
pragma Import (C, Get_My_CPU_Id, "get_my_cpu_id");
pragma Import_Valued_Procedure
  (Internal        => Get_My_CPU_Id,
   External        => "get_my_cpu_id",
   Mechanism       => (Value));

private
---------------------------------------------
-- Thread_Join_In_Background_C --
---------------------------------------------

function Thread_Join_In_Background_C
  (Resource_Id : in Integer;
   Res_Type    : in Resource_Type;
   Label       : in Contract_Label) return VRES_ID;

pragma Import (C, Thread_Join_In_Background_C,
               "frsh_thread_join_in_background_c");

-------------------------------------------
-- Receive_Endpoint_Create_C --
-------------------------------------------

function Receive_Endpoint_Create_C
  (Network_ID : C_Types.int;
   Channel    : C_Types.int) return C_Types.int;

pragma Import (C, Receive_Endpoint_Create_C,
               "frsh_receive_endpoint_create_c");

---------------------------------------
-- Send_Endpoint_Create_C     --
---------------------------------------
```

```

   function Send_Endpoint_Create_C
     (Network_ID : C_Types.int;
      To_Station : C_Types.int;
      To_Channel : C_Types.int) return C_Types.int;

   pragma Import (C, Send_Endpoint_Create_C, "frsh_send_endpoint_create_c");

   -------------------------------------------
   -- Network_Bytes_To_Budget_C  --
   -------------------------------------------

   function Network_Bytes_To_Budget_C
     (Network_ID : in C_Types.int;
      Bytes      : in C_Types.int) return Timespec;

   pragma Import (C, Network_Bytes_To_Budget_C,
                  "frsh_network_bytes_to_budget_c");

end FRSH_Ada;
```

REFERENCES

[ADA05] Taft, S. T., Duff, R. A., Brukardt, R., Ploedereder, E. and Leroy, P. Ada 2005 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1, Vol. 4348, Springer, 2006.

[AFDX09] “Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network”, ARINC Specification 664P7, Airlines Electronic Engineering Committee, Aeronautical Radio INC., 2009.

[ALD01] Aldea, M. and González Harbour, M. “MaRTE OS: An Ada Kernel for Real-Time Embedded Applications”, Proceedings of the 6th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2001, pp. 305-316.

[ALD06] Aldea, M., Bernat, G., Broster, I., Burns, A., Dobrin, R., Drake, J. M., Fohler, G., Gai, P., González Harbour, M., Guidi, G., Gutiérrez, J. J., Lennvall, T., Lipari, G., Martínez, J. M., Medina Pasaje, J., Palencia J. C. and Trimarchi, M. “FSF: A Real-Time Scheduling Architecture Framework”, in 'IEEE Real Time Technology and Applications Symposium', 2006, pp. 113-124.

[ALM01] Ada Language Mapping Specification (v1.2), Object Management Group, OMG Document, 2001.

[ARINC06] ARINC. “Avionics Application Software Standard Interface”. ARINC Specification 653-1, 2006.

[ARINC99] “ARINC Specification 629”, Aeronautical Radio, Incorporated (ARINC), 1999.

[ATMEL05] "Rad-Hard 32 bit SPARC V8 Processor — AT697E". Available on http://www.atmel.com/dyn/resources/prod_documents/doc4226.pdf, Atmel, 2005.

[AUD01] Audsley, N. and Wellings, A. "Issues with using Ravenscar and the Ada distributed systems annex for high-integrity systems", Ada Letters (XXI), 2001, pp. 33-39.

[AUD93] Audsley, N., Burns, A., Richardson, M., Tindell, K. and Wellings, A. "Applying new scheduling theory to static priority pre-emptive scheduling", Software Engineering Journal (8:5), 1993, pp. 284 -292.

[BAK91] Baker, T. P. "Stack-based scheduling for realtime processes", Real-Time Systems (3), 1991, pp. 67-99.

[BAR90] Baruah, S. K., Rosier, L. E. and Howell, R. R. "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", Real-Time Systems (2), 1990, pp. 301-324.

[BAS07] Basanta-Val, P. and García Valls, M. (Thesis supervisor) "Técnicas y extensiones para Java de Tiempo Real distribuido", Doctoral Dissertation , Universidad Carlos III de Madrid, 2007.

[BOHN07] Bohnenkamp, H., Hermanns, H. and Katoen, J.-P. "MOTOR: the MODEST tool environment", in Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 500-504.

[BOL00] Bollella, G. and Gosling, J. "The Real-Time Specification for Java,", IEEE Computer (33:6), 2000, pp. 47-54.

[BORD07] Bordin, M. and Vardanega, T. "Correctness by construction for high-integrity real-time systems: a metamodel-driven approach", in Proceedings of the 12th international conference on Reliable software technologies, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 114-127.

[BUR09] Burns, A. and Wellings, A. J. "Real-Time Systems and Programming Languages: ADA 2005, Real-Time Java, and Real-Time POSIX", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2009.

[BUR91] Burns, A. “Scheduling hard real-time systems: a review”, Software Engineering Journal (6), 1991, pp. 116-128.

[BUR94] Burns, A. and Wellings, A. J. “HRT-HOOD: A structured design method for hard real-time systems”, Real-Time Systems (6), 1994, pp. 73-114.

[CAN91] “CAN Specification”(v 2.0), Bosch, Postfach 50, D-700 Stuttgart 1, 1991.

[CHO95] Choi, J.-Y., Lee, I. and Xie, H.-L. “The Specification and Schedulability Analysis of Real-Time Systems using ACSR”, in Proceedings of the 16th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 1995, pp. 266-275.

[COR03] “CORBA Core Specification” (v3.0), Object Management Group, OMG Document formal/02-06-01, 2003.

[DCE97] “DCE 1.2: Remote Procedure Calls”, The Open Group, 1997.

[DDS07] “Data Distribution Service for Real-time Systems”, (v1.2), Object Management Group, OMG Document formal/07-01-01, 2007.

[DEC05] Decotignie, J.D. “Ethernet-Based Real-Time and Industrial Communications”, in Proceedings of the IEEE (93:6), 2005, pp. 1102 - 1117.

[DRTSJ00] “Distributed real-time specification”, JSR-50, http://www.jcp.org/en/jsr/detail?id=50, 2000.

[ETF04] “Extensible Transport Framework”, (v 1.0), Object Management Group, OMG Document formal/04-03-03, 2004.

[FEL01] Felser, M. “Ethernet TCP/IP in automation: a short introduction to real-time requirements”, in 8th IEEE International Conference on Emerging Technologies and Factory Automation, 2001, pp. 501 -504 vol.2.

[FLEX05] “FlexRay Communications System Protocol Specification”, (v 2.1), 2005.

[FLOWC11] “IEEE 802.1Qbb “Priority based flow-control”, (v 1.0), The Institute of Electrical and Electronics Engineers (IEEE), 2011.

[FOH02] Fohler, G. and Buttazzo, G. C. “Introduction to the Special Issue on Flexible Scheduling”, Real-Time Systems, Vol. 22, Springer Netherlands, 2002.

[FOS05] Foster, A. and Aslam-Mir, S. “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 Proceeding of the SDR Technical Conference and Product Exposition, 2005.

[FRSH09-A] Vila-Carbó, J., López, D. S., Orallo, E. H. and Smolík, P. “General Purpose Networks. Deliverable (D-ND2) - FRESCOR Framework”, 2009.

[FRSH09-B] Sangorrín, D. and González Harbour, M. “Fieldbus Systems. Deliverable (D-ND1) - FRESCOR Framework”, 2009.

[FRSH09-C] Sangorrín, D. and González Harbour, M. “Distributed Transaction Manager - Proof of Concepts. Deliverable (D-ND5) - FRESCOR Framework”, 2009.

[FRSH11] FRESCOR project web page: http://frescor.org. Last access in April, 2011.

[GAHI00] “Guide for the Use of the Ada Programming Language in high integrity Systems”, ISO/IEC TR 15942, 2000.

[GOM84] Gomaa, H. “A software design method for real-time systems”, Communications of the ACM (27), 1984, pp. 938-949.

[GUT01] Guitiérrez, J. J. and González Harbour, M. “Towards a real-time distributed systems annex in Ada”, Ada Letters (XXI), 2001, pp. 62-66.

[GUT02] Gutiérrez, J. J., Drake, J. M., González Harbour, M. and Medina Pasaje, J. “Modeling and schedulability analysis in the development of real-time distributed Ada systems”, Ada Letters (XXII), 2002, pp. 58-65.

[GUT96] Gutiérrez, J. J. and González Harbour, M. “Minimizing the effects of jitter in distributed hard real-time systems”, Journal of Systems Architecture (42), 1996, pp. 431-447.

[GUT99] Gutiérrez, J. J. and González Harbour, M. “Prioritizing remote procedure calls in Ada distributed systems”, Ada Letters (XIX), 1999, pp. 67-72.

[HAM04] Hamann, A., Jersak, M., Richter, K. and Ernst, R. “Design Space Exploration and System Optimization with SymTA/S - Symbolic Timing Analysis for Systems”, in Proceedings of the 25th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 2004, pp. 469-478.

[HAR01] González Harbour, M., Gutiérrez J. J., Palencia J. C. and Drake J. M. “MAST: Modeling and Analysis Suite for Real Time Applications”, in Proceedings of the 13th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, 2001, pp. 125-134.

[HAR12] González Harbour, M. Gutiérrez, J. J., Drake, J. M., López Martínez, P. and Palencia, J. C. “Modeling distributed real-time systems with MAST 2”, Journal of Systems Architecture, http://dx.doi.org/10.1016/j.sysarc.2012.02.001, 2012.

[HEN06] Hendriks, M. and Verhoef, M. “Timed automata based analysis of embedded system architectures”, in 20th International Parallel and Distributed Processing Symposium (IPDPS), 2006, pp. 179-187.

[HRI05] Hristu-Varsakelis, D., Levine, W. S., Alur, R., Arzen, K.-E., Baillieul, J. and Henzinger, T. A. Handbook of Networked and Embedded Control Systems (Control Engineering), Birkhauser, 2005.

[HUG08] Hugues, J., Zalila, B., Pautet, L. and Kordon, F. “From the prototype to the final embedded system using the Ocarina AADL tool suite”, ACM Transactions on Embedded Computing Systems (7), 2008, pp. 1-25.

[IEC00] “IEC International Standard 61158: Fieldbus standard for use in industrial control systems”, IEC International Electrotechnical Comittee, IEC Document, 2000.

[IEC04] “IEC PAS 62030: Digital data communications for measurement and control”, (First edition), IEC International Electrotechnical Commission., IEC Document, 2004.

[IEC07] “IEC International Standard 61784-1 and -2: Industrial communication networks.”, IEC International Electrotechnical Comittee, IEC Document, 2007.

[JAH86] Jahanian, F. and Mok, A. K. “Safety analysis of timing properties in real-time systems”, IEEE Transactions on Software Engineering (12), 1986, pp. 890-904.

[JOS86] Joseph, M. and Pandya, P. “Finding Response Times in a Real-Time System”, The Computer Journal (29:5), 1986, pp. 390-395.

[KER99] Kermarrec, Y. “CORBA vs. Ada 95 DSA: a programmer's view”, Ada Letters (XIX), 1999, pp. 39-46.

[KISZ05] Kiszka, J. and Wagner, B. “RTnet - a flexible hard real-time networking framework”, in 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2005, pp. 456-464.

[KLE93] Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and González Harbour, M. “A practitioner's handbook for real-time analysis”, Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[KOP93] Kopetz, H. and Grunsteidl, G. “TTP - A time-triggered protocol for fault-tolerant real-time systems”, in The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS), 1993, pp. 524 -533.

[KOP11] Kopetz, H. “Real-Time Systems: Design Principles for Distributed Embedded Applications”, Springer, 2011.

[KRAH01] Krahl, D. “Extend: the Extend Simulation Environment”, in Proceedings of the 33rd Winter Simulation Conference, IEEE Computer Society, Washington, DC, USA, 2001, pp. 217-225.

[LAN02] Lankes, S., Reke, M. and Jabs, A. “A Time-Triggered Ethernet Protocol for Real-Time CORBA”, in Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE Computer Society, Washington, DC, USA, 2002, pp. 215-222.

[LAN03] Lankes, S., Jabs, A. and Bemmerl, T. “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, IEEE Computer Society, Washington, DC, USA, 2003, pp. 121-129.

[LEE98] Lee J-Y., Moon H-J., Kwon W. H., Less S. W. and Park I. S. “Token-Passing bus access method on the IEEE 802.3 physical layer for distributed control networks”, Distributed Computer Control Systems (DCCS), Elsevier Science, 1998, pp. 31-36.

[LEH89] Lehoczky, J., Sha, L. and Ding, Y. “The rate monotonic scheduling algorithm: exact characterization and average case behavior”, in Real Time Systems Symposium, 1989, pp. 166 -171.

[LELAN93] LeLan, G. Rivierre, N. “Report RR1863: Real-Time Communications over Broadcast Networks: the CSMA-DCR and the DOD-CSMA-CD Protocols”, Technical report, INRIA, 1993.

[LEON05] “LEON2 Processor User’s Manual”, Gaisler Research, 2005.

[LI04] Li, P., Ravindran, B., Cho, H. and Jensen, E. D. “Scheduling Distributable Real-Time Threads in Tempus Middleware”, in Proceedings of the Tenth International Conference on Parallel and Distributed Systems, IEEE Computer Society, Washington, DC, USA, 2004, pp. 187-194.

[LIU00] Liu, J. W. S. “Real-Time Systems”, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[LIU73] Liu, C. L. and Layland, J. W. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, J. ACM (20), 1973, pp. 46-61.

[LOP04] Campos, J. L., Gutiérrez, J. J. and González Harbour, M. “The Chance for Ada to Support Distribution and Real-Time in Embedded Systems”, in Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2004, pp. 91-105.

[LOP06] Campos, J. L., Gutiérrez, J. J. and González Harbour, M. “Interchangeable Scheduling Policies in Real-Time Middleware for Distribution”, in Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2006, pp. 227-240.

[LOPE08] López, J. M., Díaz, J. L., Entrialgo, J. and García, D. “Stochastic analysis of real-time systems under preemptive priority-driven scheduling”, Real-Time Systems (40), 2008, pp. 180-207.

[LOPZ04] López Martínez, P., Medina Pasaje, J. and Drake, J. M. “Sim_MAST: Simulador de Sistemas Distribuidos de Tiempo Real”, in XII Jornadas de Concurrencia y Sistemas Distribuidos, 2004.

[LOPZ10] López Martínez, P. and Drake, J.M. (Thesis Supervisor) “Desarrollo de sistemas de tiempo real basados en componentes utilizando modelos de comportamiento reactivos”, Doctoral Dissertation, 2010.

[LOS04] Losert, T., Huber, W., Hendling, K. and Jandl, M. “An extensible transport framework for CORBA with emphasis on real-time capabilities”, in Second IEEE International Conference on Computational Cybernetics (ICCC), 2004, pp. 155-161.

[LU11] Lu, Y., Nolte, T., Cucu-Grosjean, L. and Bate, I. “RapidRT: A Tool For Statistical Response-Time Analysis of Complex Industrial Real-Time Embedded Systems”, in Real-Time SystemS @ Work, the Open Demo Session of Real-Time Techniques and Technologies of the 32nd IEEE Real-Time Systems Symposium (RTSS'11), 2011.

[MACB04] “IEEE Std 802.1D-2004. “Media Access Control (MAC) Bridges”, The Institute of Electrical and Electronics Engineers (IEEE), 2004.

[MAH04] Mahmoud, Q. Middleware for communications, J. Wiley & Sons Ltd, 2004.

[MAKI08] Mäki-Turja, J. and Nolin, M. “Efficient implementation of tight response-times for tasks with offsets”, Real-Time Systems (40), 2008, pp. 77-116.

[MAR05] Martínez, J. M. and González Harbour, M. “RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet”, in Proceedings of the 10th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2005, pp. 180-195.

[MAR08] “A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems”, Object Management Group, OMG Document ptc/2008-06-09, 2008.

[MAZ09] Mazzini, S., Puri, S. and Vardanega, T. “An MDE methodology for the development of high-integrity real-time systems”, in Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2009, pp. 1154-1159.

[MED05] Medina Pasaje, J. and Drake, J.M. (Thesis Supervisor) “Metodología y herramientas UML para el modelado y análisis de sistemas de tiempo real orientados a objetos”, Doctoral Dissertation, 2005.

[MED11] Medina Pasaje, J. and García Cuesta, A. “Model-based analysis and design of real-time distributed systems with Ada and the UML profile for MARTE”, in Proceedings of the 16th Ada-Europe international conference on Reliable software technologies, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 89-102.

[MIS04] “Guidelines for the Use of the C Language in Critical Systems”, Technical report, MISRA Consortium, 2004.

[MOK78] Mok, A. and Dertouzos, M. “Multiprocessor scheduling in a hard real-time environment”, in Proceedings of the Seventh Texas Conference on Computing Systems, 1978.

[MOK96] Mok, A. K., Tsou, D.-C. and de Rooij, R. C. M. “The MSP.RTL real-time scheduler synthesis tool”, in Proceedings of the 17th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 1996, pp. 118-128.

[NAM04] “Naming Service Specification”, (v 1.3), Object Management Group, OMG Document formal/04-10-03, 2004.

[PAL05] Palencia, J. C. and González Harbour, M. “Response time analysis of EDF distributed real-time systems”, Journal of Embedded Computing (1), 2005, pp. 225-237.

[PAL97] Palencia, J. C., Gutiérrez J. J. and González Harbour, M. “On the schedulability analysis for distributed hard real-time systems”, Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, 1997, pp. 136-143.

[PAL99] Palencia, J. C. and González Harbour, M. “Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems” in Proceedings of the 20th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 1999, pp. 328-339.

[PAU00] Pautet, L. and Tardieu, S. “GLADE: A Framework for Building Large Object-Oriented Real-Time Distributed Systems”, ISORC, 2000, pp. 244-251.

[PED02] Pedreiras, P., Almeida, L. and Gai, P. “The FTT-Ethernet Protocol: Merging Flexibility,Timeliness and Efficiency”, in Proceedings of the 14th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, 2002, pp. 134-142.

[PED03] Pedreiras, P., Leite, R. and Almeida, L. “Characterizing the Real-Time Behavior of Prioritized Switched-Ethernet”, in Proceedings of the 2nd Workshop on Real-Time LAN's in the Internet Age (RTLIA), 2003.

[PER08] Pérez, H., Gutiérrez, J. J., Sangorrín, D. and González Harbour, M. “Real-Time Distribution Middleware from the Ada Perspective”, in Proceedings of the 13th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2008, pp. 268-281.

[PER09] Pérez, H. and Gutiérrez, J. J. “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, NY, USA, 2009, pp. 73-78.

[PER10] Pérez, H., Gutiérrez, J. J. and González Harbour, M. “Support for a real-time transactional model in distributed Ada”, Ada Letters (XXX), 2010, pp. 91-103.

[PER11] Pérez, H., Gutiérrez, J. J., Asensio, E., Zamorano, J. and de la Puente, J. A. “Model-Driven Development of High-Integrity Distributed Real-Time Systems Using the End-to-End Flow Model” in Proceedings of the 37th Euromicro Conference on Software Engineering and Advanced Applications, Oulu, Finland, 2011, pp. 209-216.

[PER12] Pérez, H. and Gutiérrez, J. J. “On the schedulability of a data-centric real-time distribution middleware”, Journal of Computer Standards & Interfaces, Volume 34, Issue 1, 2012, pp. 203-211.

[PERA07] Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Henia, R., Racu, R., Ernst, R. and González Harbour, M. “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, ACM, New York, NY, USA, 2007, pp. 193-202.

[PERR10] Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T. and Hugues, J. “The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software”, in Proceedings of Embedded Real Time Software and Systems, Toulouse, France, 2010.

[PIN02] Pinho, L. M. and Vasques, F. “Transparent Environment for Replicated Ravenscar Applications”, in Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies, Springer-Verlag, London, UK, 2002, pp. 297-308.

[PIN02-B] Pinho, L. M. and Vasques, F. “Using Ravenscar to support fault-tolerant real-time applications”, Ada Letters (XXII), 2002, pp. 47-52.

[PLA08] Merle, A. P. P. and Seinturier, L. “A Real-Time Java Component Model”, in IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, (0), 2008, pp. 281-288.

[POS98] “POSIX.13 IEEE Std. 1003.13-1998. Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)”, The Institute of Electrical and Electronics Engineers (IEEE), 1998.

[PUE00] de la Puente, J. A., Ruiz, J. and Zamorano, J. “An Open Ravenscar Real-Time Kernel for GNAT”, in Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies, 2000, Springer Berlin / Heidelberg, pp. 5-15.

[PYA01] Pyarali, I., Spivak, M., Cytron, R. and Schmidt, D. C. “Evaluating and Optimizing Thread Pool Strategies for Real-Time CORBA”, LCTES/OM, ACM, 2001, pp. 214-222.

[RAJ89] Rajkumar, R. “Task synchronization in real-time systems”, Doctoral Dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, AAI9016357, 1989.

[RAP03] “Rapid-RMA: The Art of Modeling Real-Time Systems”, Technical report, Tri-Pacific, 2003.

[RED04] Redell, O. “Analysis of tree-shaped transactions in distributed real time systems”, in Proceedings of the 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004, 2004, pp. 239-248.

[REK03] Rekik, R. and Hasnaoui, S. “Application of a CAN BUS transport for DDS middleware”, in Second International Conference on the Applications of Digital Information and Web Technologies (ICADIWT), 2009, pp. 766 -771.

[RFC2474] Nichols, K., Blake, S., Baker, F. and Black, D. “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers”, RFC Editor, United States, 1998.

[RFC3550] “RFC 3550: RTP: A Transport Protocol for Real-Time Applications”, The Internet Society, 2003.

[RIV10] Rivas, J., Gutiérrez, J. J., Palencia, J. C. and González Harbour, M. "Optimized Deadline Assignment and Schedulability Analysis for Distributed Real-Time Systems with Local EDF Scheduling", 8th International Conference on Embedded Systems and Applications (ESA), 2010.

[RIV11] Rivas, J., Gutiérrez, J. J., Palencia, J. C. and González Harbour, M. "Schedulability Analysis and Optimization of Heterogeneous EDF and FP Distributed Real-Time Systems", 23rd Euromicro Conference on Real-Time Systems (ECRTS), 2011, pp. 195-204.

[RMI04] "Java Remote Method Invocation (RMI)", Sun Microsystems, http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf, 2004.

[RTC05] "Realtime CORBA Specification", (v1.2), Object Management Group, OMG Document formal/05-01-04, 2005.

[RTPS09] "The Real-time Publish-Subscribe Wire Protocol. DDS Interoperability Wire Protocol Specification", Object Management Group, 2009.

[SAE09] "Architecture Analysis and Design Language (AADL) - AS5506A", SAE, 2009.

[SAN10] Sangorrín, D., González Harbour, M., Pérez, H. and Gutiérrez, J. J. "Managing Transactions in Flexible Distributed Real-Time Systems", in Proceedings of the 15th Ada-Europe International Conference on Reliable Software Technologies, Valencia, Spain, Springer, 2010, pp. 251-264.

[SCH01] Schmidt, D. C., Mungee, S., Flores-Gaitan, S. and Gokhale, A. "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers", Real-Time Systems (21), 2001, pp. 77-125.

[SCH05] Schmidt, D. C. "TAO Developer's Guide. Building a standard in performance", Object Computing, Inc., 2005.

[SCH96] Schmidt, D. C. and Cranor, C. D. "Pattern languages of program design 2", Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996, pp. 437-459.

[SCH98] Schmidt, D. C. “Evaluating Architectures for Multithreaded Object Request Brokers”, Communications of the ACM (41:10), 1998, pp. 54-60.

[SCH98-2] Schmidt, D. C., Levine, D. L. and Mungee, S. “The design of the TAO real-time object request broker”, Computer Communications (21:4), 1998, pp. 294-324.

[SHA90] Sha, L., Rajkumar, R. and Lehoczky, J. P. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, IEEE Transactions on Computers (39), 1990, pp. 1175-1185.

[SIN04] Singhoff, F., Legrand, J., Nana, L. and Marcé, L. “Cheddar: a flexible real time scheduling framework”, in Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies, ACM, New York, NY, USA, 2004, pp. 1-8.

[SPA10] “SPARK - The Spade Ada Kernel”, SPARK LRM, 2010.

[SPU96] Spuri, M. “Analysis of Deadline Scheduled Real-Time Systems”, Technical report, Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), 1996.

[SPU96-2] Spuri, M. “Holistic Analysis of Deadline Scheduled Real-Time Distributed Systems”, Technical report, Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), 1996.

[SPW08] “SpaceWire - links, nodes, routers and networks”, European Corporation for Space Standardization (ECSS), 2008.

[STA92] Stankovic, J. “Distributed real-time computing: the next generation”, Technical report, Dept. of Computer and Information, University of Massachusetts, 1992.

[TEJ07] Tejera, D., Alonso, A. and de Miguel, M. A. “RMI-HRT: remote method invocation - hard real time”, in Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, ACM, New York, NY, USA, 2007, pp. 113-120.

[THI00] Thiele, L., Chakraborty, S. and Naedele, M. “Real-time calculus for scheduling hard real-time systems”, in Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), 2000, pp. 101 -104 vol.4.

[TIM02] “Using TimeWiz to Understand System Timing before you Build or Buy. White paper”, Technical report, TimeSys Corporation, 2002.

[TIN94A] Tindell, K. W., Burns, A. and Wellings, A. J. “An extendible approach for analyzing fixed priority hard real-time tasks”, Real-Time Systems (6), 1994, pp. 133-151.

[TIN94B] Tindell, K. “Adding Time-Offsets to Schedulability Analysis”, Technical report, Technical report, University of York, 1994.

[TIN94C] Tindell, K. and Clark, J. “Holistic schedulability analysis for distributed hard real-time systems”, Microprocessors and Microprogramming Journal (40), 1994, pp. 117-134.

[URU11] Urueña, S. and Zamorano, J. (Thesis Supervisor) “Arquitectura Software De Comunicaciones Para Sistemas Distribuidos Críticos Con Requisitos De Tiempo Real Estrictos”, Doctoral Dissertation, Universidad Politécnica de Madrid, 2011.

[VBLAN06] “IEEE Std 802.1Q. “Virtual Bridged Local Area Networks”, Annex G, The Institute of Electrical and Electronics Engineers (IEEE), 2006.

[VER04] Vergnaud, T., Hugues, J., Pautet, L. and Kordon, F. “PolyORB: A Schizophrenic Middleware to Build Versatile Reliable Distributed Applications”, in Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies, Springer, 2004, pp. 106-119.

[VILA08] Vila-Carbó, J., Tur-Masanet, J. and Hernández-Orallo, E. “An evaluation of switched ethernet and linux traffic control for real-time transmission”, ETFA, 2008, pp. 400-407.

[VILA08-B] Vila-Carbó, J. and Hernández-Orallo, E. “An analysis method for variable execution time tasks based on histograms”, Real-Time Systems (38), 2008, pp. 1-37.

[WHIT10] White, R. “Providing additional real-time capability and flexibility for Ada 2005”, Ada Letters. (30), 2010, pp. 135-146.

[WIL97] Wilner, D. “What really happened on Mars?”, Keynote talk at the 18th IEEE Real-Time Systems Symposium (RTSS), IEEE Computer Society, 1997.

[XIO03] Xiong, M., Parsons, J., Edmondson, J., Nguyen, H. and Schmidt, D. “Evaluating Technologies for Tactical Information Management in Net-Centric Systems”, in Proceedings of the Defense Transformation and Net-Centric Systems Conference, 2007.

[XU93] Xu, J. and Parnas, D. L. “On Satisfying Timing Constraints in Hard-Real-Time Systems”, IEEE Transactions on Software Engineering (19), 1993, pp. 70-84.

[ZHA01] Zhang, C. and Tsaoussidis, V. “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, ACM, New York, NY, USA, 2001, pp. 189-198.