# A Flexible Run-time Support for Distributed Dependable Hard Real-time Applications*


| Emmanuelle Anceaume | Gilbert Cabillic | Pascal Chevochot | Isabelle Puaut |
|---|---|---|---|
| CNRS/IRISA | INRIA/IRISA | INRIA/IRISA | INSA/IRISA † |
| anceaume@irisa.fr | cabillic@irisa.fr | pchevoch@irisa.fr | puaut@irisa.fr |



## Abstract

*Typically, most distributed, dependable, real-time systems designed in the past can only meet the particular requirements of the application domain to which they were targeted. This approach led to specific, non-flexible, dedicated and non-reusable solutions, often based on specialized hardware. This paper presents an alternative approach where a flexible run-time support for distributed dependable hard real-time applications is built on top of off-the-shelf hardware. This support has been designed by considering three fundamental and complementary aspects:* real-time*, to support applications that exhibit strict timing constraints;* fault-tolerance*, to provide a high degree of reliability through the transparent provision of fault tolerant mechanisms; and* flexibility*, to allow the modifications of components of the run-time support without having to rewrite it entirely, and to support a large range of application domains, real-time kernels and hardware.*


## 1. Introduction

Dependable hard real-time applications in the last thirty years have been spread out in space rockets and flight control systems, nuclear power plants, stock exchange, medical and automotive equipments. Designing systems supporting such demanding applications is difficult because they must reach a triple goal. First, these systems have to be *highly reliable* to meet the maximum acceptable probability of failure ranging from $10^{-4}$ to $10^{-10}$ failures per hour [22, 14]. Second, they have to ensure *strict timeliness* requirements to guarantee response times typically ranging from 1 to $100ms$ [18, 22]. Third, they must enforce *data consistency* to handle concurrent executions of multiple tasks, especially in a distributed environment.

Reaching this triple goal is complex and led so far to solutions that met only the particular requirements of an application domain. By particular requirements we refer to the constraints imposed by the applications in terms of resources consumption, specific scheduling needs, or fault-tolerance needs. Existing solutions are therefore dedicated to a single application domain, and often based on specialized (and costly) hardware, making the implemented software seldom reusable in a different context [22, 14].

The HADES project (Highly Available Distributed Embedded System) [2, 20] developed at IRISA addresses these issues and provides an environment for the development and the execution of distributed dependable hard real-time applications.

Application development in HADES relies on the use of an original task model and a set of off-line tools. The HADES task model defines the way application tasks are structured and the constraints that must be handled during their execution. The off-line tools encompass feasibility tests associated with a panel of scheduling policies, an automatic replication tool, and a tool that determines the right amount of memory to be allocated for the application execution. The replication tool [6] automatically replicates application tasks according to the requirements of the application designer (e.g., tasks or the portions of tasks to be replicated, replication strategy to be used such as active, passive or semi-active replication).

Application execution is managed by the HADES run-time support. This run-time support is built as a middleware software layer running on top of off-the-shelf real-time kernels (it has been ported on CHORUS [7] and emulated on SOLARIS). It consists in a set of services mandatory for the execution of a large panel of dependable distributed real-time tasks. Examples of such services are scheduling policies, communication primitives, group membership management, distributed execution control, and clock synchronization. All these services exhibit timeliness and depend-


*Proc. of the 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing, may 2-5 1999, Saint Malo, France. This work is partially supported by the French Department of Defense (DGA/DSP), #198C4980031303011, and by Dassault Aviation, #197C0690031303012



†IRISA, Institut de Recherche en Informatique et Systèmes Aléatoires, Campus de Beaulieu, 35042 Rennes Cedex, FRANCE

ability properties.

The HADES run-time support has been designed to guarantee three fundamental and complementary aspects:

1. *Real-time*: support of applications that exhibit strict timing constraints;
2. *Fault-tolerance*: achievement of a high degree of reliability thanks to fault-tolerance mechanisms which are transparent to the application designer;
3. *Flexibility*:
   (a) *Extensibility*: ability to change parts of the run-time support without having to rewrite it entirely;
   (b) *Adaptability*: support for a large range of application domains, real-time kernels and hardware.

The achievement of the real-time aspect mainly relies on an accurate estimation of the worst-case execution times (WCETs) for all the activities to be executed in the system (application tasks, run-time support tasks, real-time kernel). The computation of WCETs of the application and run-time support tasks is eased by the HADES task model (see Section 2); in particular, it requires that all synchronization constraints (e.g., precedence between tasks, resource sharing) be statically specified. Concerning the core of the run-time support, it has been kept small enough to be easily analyzed. Lastly, the predictability of the real-time kernel has been verified through the analysis of the CHORUS kernel source code.

The fault-tolerance aspect of the run-time support relies on the automatic task replication tool, and on fault-detection and exception handling mechanisms offered by the run-time support. For space consideration, this aspect is not detailed in this paper, but the interested reader is invited to read [6].

To achieve flexibility, we have specified in HADES a set of services with a well-defined interface. These services can evolve without having to rewrite the run-time support (*extensibility* property). The implementation of any service can be replaced by an other one with different properties as far as it conforms to the service interface. This allowed us to develop a range of service implementations adapted to the specific constraints of the applications, the real-time kernel and the hardware (*adaptability* property).

The remainder of this paper is devoted to the description of the HADES run-time support. It concentrates on the main structuring principles that were exploited to obtain a run-time support exhibiting a timely behavior, while maintaining its flexibility. It is organized as follows. Section 2 briefly presents the HADES task model. The HADES run-time support is described in Section 3. This description encompasses an overview of the services interface illustrating the extensibility of HADES (Section 3.2). Section 3.3 points out the completeness of this interface to demonstrate the adaptability of HADES. Finally, the HADES run-time support is compared with related work in Section 4.

## 2. The Hades task model

The HADES task model describes every task as a directed acyclic graph (see Section 2.1). A set of timing, synchronization, distribution and fault-tolerance attributes can be specified for each task (see Section 2.2). The fault model assumed when designing these tasks is given in Section 2.3. Hereafter, the term *task* covers both the application tasks, denoted by *App_tasks*, and the tasks implementing run-time support services, denoted by *RTS_tasks*. In the remainder of this section, we give only a brief description of the task model (see [2] for more details).

### 2.1. Structure of tasks

Every task is described as a direct acyclic graph, called *HEUG*, for HADES *Elementary Unit Graph*. A node (also called *EU* for *Elementary Unit*) models a computation local to a site. An *EU* can be either a *Code Elementary Unit* (*Code_EU*), i.e., a sequence of code without synchronization, or a *System Call Elementary Unit* (*Sys_EU*). Tasks can use an *invocation Sys_EU* to request the synchronous or asynchronous execution of other tasks, or to interact with the run-time support. Contrary to *Code_EUs*, *Sys_EUs* are not preemptible, that is their execution can not be suspended by any other computation. An edge is a precedence constraint between nodes. An edge from an *Elementary Unit* $eu_i$ to an *Elementary Unit* $eu_j$ states that $eu_j$ can start its execution only after $eu_i$ has finished its execution.

### 2.2. Task attributes

Task attributes can be classified into synchronization attributes, timing attributes, distribution attributes and fault-tolerance attributes.

**Synchronization attributes:** The *Code_EUs* can be synchronized by using either condition variables (i.e., boolean variables that can be cleared, set and waited for by *Code_EUs*) or shared resources. A *resource* abstracts any hardware or software component required to execute a *Code_EU*. A resource can be associated with persistent data, i.e., data that survives beyond the execution of a *Code_EU*. A *Code_EU* or a group of *Code_EUs* must specify which resources to acquire before beginning its execution, together with the requested access modes (*shared* or *exclusive*).

**Timing attributes:** Timing attributes express temporal properties at the task level and at the elementary unit level. They can be used off-line, for instance by static feasibility tests, and/or on-line for the purpose of execution monitoring, or dynamic scheduling policies. At the task level, attributes express either the expected task arrival law (i.e., periodic, sporadic, aperiodic) or its relative deadline (i.e., the date at which the task execution must be completed, relative to the task activation date). In addition, the scheduling service to be used to compute the execution order of *App_task EUs* can be chosen by the application designer (see Section 3). At the *EU* level, the application designer can specify the *EU* deadline, earliest start time, latest start time, and execution priority.

**Distribution attributes:** A task may execute on a set of sites. A site has a processor, a memory, and a set of local hardware devices (peripheral devices, sensors, actuators). Distribution attributes specify for each *Code_EU* and *Sys_EU* the site on which it executes. A precedence constraint between $eu_i$ and $eu_j$ is said to be *local* (resp. *remote*) if $eu_i$ and $eu_j$ are assigned to the same site (resp. different sites). The application designer can specify which execution service is in charge of managing the execution of the *App_tasks* (see Section 3).

**Fault-tolerance attributes:** Fault-tolerance attributes specify for each *HEUG* or portions of them (i.e., *HEUG* subgraphs) which fault-tolerance strategy is to be applied (active, passive, semi-active, temporal replication) and the requested replication degree. These attributes are used by the HADES off-line replication tool [6]. The application designer can also specify for each *Code_EU* a handler that catches an exception whenever it is raised. An HADES exception is defined as a synchronous event triggered by the run-time support to notify the *Code_EU* of errors related to its execution (e.g. processor exception, attempt to execute an operation on a failed resource). This enables the designer to define specific fault-tolerance strategies.

### 2.3. Fault model

HADES has been designed to support omissions, value and timing faults. We have defined mechanisms to simulate *fail-silent sites*, i.e., sites for which omission, value and timing faults affecting its processor, memory or some other hardware device causes the site to stop before propagating the erroneous state to another site. Mechanisms (see [6]) have been defined at the task construction level (e.g., temporal replication), and at the run-time support level (e.g., fault-detection) to approximate the fail-silent behavior. The communication network is subject to omission failures. To tolerate these failures, we assume that the number of successive omissions is bounded.

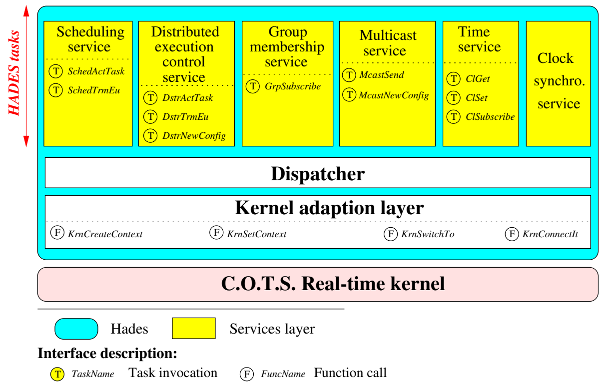


**Figure 1. Internal structure of HADES run-time support**

## 3. Description of the Hades run-time support

A key concern when designing the HADES run-time support was to guarantee its timeliness while retaining its flexibility. This section gives an overview of the structuring principles that were followed during HADES construction to reach these objectives.

### 3.1. Overall functionalities of the run-time support

The HADES run-time support executes on every site of the distributed system as a middleware software layer based on a Commercial-Off-The-Shelf (COTS) real-time kernel. Its purpose is to provide a range of functionalities to execute dependable distributed real-time applications. The HADES run-time support has been divided into three software layers, namely the *dispatcher*, the *kernel adaptation layer* and the *services layer* (see Figure 1).

**Dispatcher:** The dispatcher is the core of the run-time support. It provides the very basic functionalities needed to execute a task, namely *task eligibility management* and *task monitoring*. It interacts with both the services layer and the kernel adaption layer.

Concerning eligibility management, the dispatcher implements multilevel scheduling. It maintains a priority-ordered set of run-queues. Upon activation of a given task (*App_task* or *RTS_task*), the task *EUs* are inserted into the appropriate run-queue by their scheduling algorithm. Each scheduling policy is implemented with *RTS_tasks* of the services layer, called *scheduling tasks*. To order *App_tasks EUs*, HADES provides a set of scheduling algorithms, ranging from static to dynamic policies; a temporal attribute (see

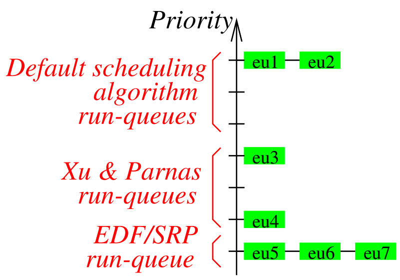


**Figure 2. Example of run-queues division**

Section 2.1) specifies the scheduling algorithm in charge of each *App_task*. On the other hand, all the *RTS_tasks* (including the *scheduling tasks*) are scheduled by a default scheduling algorithm (fixed priority scheduling algorithm [25]), directly provided by the dispatcher. To each scheduling policy is associated a subset of the priority-ordered set of run-queues (see figure 2).

An *EU* $eu_i$ is *runnable* if it meets the following four eligibility conditions : *(i)* the *EUs* that $eu_i$ must wait for, due to precedence constraints, have finished their execution, *(ii)* all the resources needed by $eu_i$ can be granted to $eu_i$, *(iii)* all the condition variables that $eu_i$ must wait for are set, *(iv)* the current time is higher than $eu_i$ earliest start time. Among runnable *EUs*, the dispatcher executes the one with the highest priority. The cohabitation of different scheduling algorithms has been made feasible in HADES thanks to the introduction of preemptible *Code_EUs*.

The monitoring functionality of the dispatcher consists in checking tasks execution progress in order to detect the violation of safety, liveness and timeliness properties (e.g., missed deadline, violation of tasks arrival laws).

**Kernel adaption layer:** To ease the port the HADES run-time support on a different real-time kernel, a *kernel adaption layer* has been designed. This layer interacts with the underlying real-time kernel for activities related to thread and interrupt management. The adaptability of this kernel adaption layer is shown in Section 3.3.4.

**Services layer:** The services layer consists in a panel of services exhibiting fundamental properties (i.e., availability, dependability, timeliness) meeting the requirements of distributed dependable real-time applications. A *service* is defined as a module accomplishing a given function, and is called through an interface made of *task invocations*.

Services pertaining to the services layer are relevant to scheduling policies, distributed execution control, message multicast, group membership, time management, and clock synchronization (see Figure 1). In contrast to the dispatcher functionalities, these services are specified in terms of *HEUGs* (e.g. the *scheduling RTS_task*) and are extensible, i.e., new services can be added and existing services can be modified by the application designer as long as their implementation conforms to the service interface (see Section 3.2).

## 3.2. Support for extensibility

Extensibility is the property of systems in which the evolution of existing functionalities or the design of new ones is possible at any time during the life of the system. In order to reach extensibility, the dispatcher is decoupled from the services layer. This enables the application designer to extend the properties of any given service without having to cope with timing, synchronization, fault-tolerance and distribution constraints that are inherent to any other services. The following sections present the interface of four services, namely the scheduling service, the distributed execution control service, the multicast service, and the group membership service.

**Scheduling service:** The scheduling service is in charge of computing the order in which *App_task EUs* must execute to meet their timing requirements (e.g., deadline, priority). The feasibility tests, analyzing whether a set of tasks can meet its timing constraints or not, are provided by off-line tools. In order to bind a given scheduling policy with a given *App_task* $T_i$, one indicates in the timing attribute of $T_i$'s *HEUG* (see Section 2) which scheduling policy is to be used. The scheduling service interface is composed of two *RTS_tasks* invocations, named *SchedActTask* and *SchedTrmEu*. The *SchedActTask* task is asynchronously invoked by the dispatcher upon activation of any *App_task* $T_i$. Two parameters identifying the task to be scheduled (i.e., $T_i$) and the set of $T_i$'s *EUs* that must be executed on the current site are provided. The *SchedTrmEu* task is asynchronously invoked by the dispatcher upon termination of any *EU*. The *EU* is given as argument.

The *SchedActTask* and *SchedTrmEu* tasks interact with the dispatcher via a set of *Sys_EUs* that manage the priority-ordered set of run-queues (i.e., insert first, insert last, insert after, and delete *Sys_EUs*).

The scheduling service interface has been designed to provide different scheduling policies, each being adapted to a different application domain (see Section 3.3.1).

**Distributed execution control service:** The distributed execution control service is in charge of launching the execution of distributed tasks on the relevant sites, validating their precedence constraints and detecting their termination. The distributed execution control service to be used by a task $T_i$ is specified in the distribution attribute of $T_i$'s *HEUG* (see Section 2). The execution control service is called through the invocations of three *RTS_tasks*, named *DstrActTask*, *DstrTrmEu* and *DstrNewConfig*. Upon activation of a distributed *App_task* $T_i$, the dispatcher asynchronously invokes the *DstrActTask* task which is in charge of launching the distributed execution of $T_i$. The *DstrTrmEu* task is asynchronously invoked by the dispatcher upon termination of an *EU*. Finally, upon modification of the group membership, the *DstrNewConfig* task is asynchronously invoked by the group membership service, with two parameters indicating the new membership group view, and the time at which reinserted sites are bound to be operational.

The execution control tasks interact with the dispatcher via a set of *Sys_EUs* designed to interact with the dispatcher (e.g., allocation of an execution flow, validation of a set of precedence constraints, completion of the execution of an *invocation Sys_EU*). These tasks also interact with the multicast service to launch the task executions on the appropriate sites.

The execution control service interface has been designed to allow the implementation of execution control policies relying on different fault models (see [6] for further details).

**Multicast service:** The HADES multicast service is concerned with the transmission of messages to a set of sites. The HADES multicast service interface is made of two *RTS_tasks* invocations, named *McastSend* and *McastNewConfig*. Upon request to multicast a message to a group of sites, the requesting task asynchronously invokes the *McastSend* task which is in charge of multicasting the message. Three arguments indicate the message body, the destination sites and the task in charge of handling the message on each destination site. The *McastNewConfig* task is asynchronously invoked upon modification of the group membership. Two arguments indicating the new membership group view and the time at which reinserted sites are bound to be operational are provided.

Upon receipt of a message, the multicast service invokes the appropriate task in charge of handling the received message, with the message as an argument. The invocation of this task can be delayed to ensure that the delivery ordering properties of the multicast service are met (i.e., causal order, total, or temporal order).

The multicast service interface has been designed to allow the implementation of multicast protocols with different reliability and delivery order properties (see Section 3.3.2).

**Group membership service:** The HADES group membership service is in charge of detecting failures, allowing operational sites to agree on the set of failed and operational sites. Its interface is composed of a *RTS_task* invocation named *GrpSubscribe* allowing a HADES service to be notified upon group membership changes. More precisely, upon request to be notified of group membership modifications, the requesting service invokes the *GrpSubscribe* task by indicating its identifier.

Upon group membership change, the group membership service provides the new membership group view as well as the time at which reinserted sites are bound to be operational to subscriber services.

Notice that the respective interfaces of the multicast and group membership services do not impose the role and number of tasks used to implement these services. For performance reasons, the service designer can implement these two services jointly by using the same task to implement both the multicast and group membership protocols (see sections 3.3.2 and 3.3.3), while preserving their flexibility.

## 3.3. Support for adaptability

HADES specifies the interface of a set of run-time support services, so that the properties they exhibit can be adapted to the target application domain, to the hardware, or to the fault hypothesis, as long as they conform to the service interface (see Section 3.2). An interface, made of function calls, is equally specified for the kernel adaptation layer. This section illustrates this flexibility by giving for three services of the services layer and the kernel adaption layer, two different implementations according to the scheduling requirements (Scheduling service), the message delivery order (Multicast service), the group membership properties (Group membership service), and the underlying kernel features (Kernel adaption layer).

### 3.3.1 Scheduling service

The choice of a scheduling service highly relies on application-specific features such as the task arrival laws, the task priorities, the preemption policy, and the resource access patterns. Several dynamic and static scheduling algorithms (e.g. Earliest Deadline First (EDF) [17, 24]), Rate Monotonic (RM) [17]) as well as planning-based scheduling policies [27, 21, 1]) exist and some of them have been specified and implemented in the HADES run-time support. Two scheduling policies that conform to the interface of the scheduling service are presented hereafter : a *static planning-based scheduling* policy and a *dynamic scheduling* policy. The first implementation is intended for applications whose required resources must be preallocated so that deadlines can be *a priori* guaranteed. The motivation for the second implementation is to support sporadic tasks.

**Static planning-based scheduling policy:** The static planning-based scheduling policy we have specified and implemented is based on the Xu and Parnas algorithm [27], which acts on a set of periodic tasks with resources and precedence constraints. The original algorithm has been extended to handle distributed tasks. This scheduling policy relies on a feasibility test run off-line, which generates a scheduling plan, containing for every *EU* its start time relative to the beginning of the task period. A set of HADES run-queues is reserved for *EUs* scheduled by the extended Xu and Parnas scheduler. Each *EU* is mapped onto one of this run-queue depending on its preemption level (i.e. if *EU* $eu_i$ preempts *EU* $eu_j$, it is assigned to a run-queue with a higher priority). The execution of the Xu and Parnas scheduling policy is implemented by two *RTS_tasks* named $t^{Act}_{XuParnas}$ and $t^{Trm}_{XuParnas}$. Upon activation of an *App_task*, that is at the beginning of the task period, task $t^{Act}_{XuParnas}$ inserts each *EU* of this task into the dispatcher run-queues. It uses the information given in the scheduling plan to insert the *EUs* in the run-queue corresponding to their preemption level and for a given run-queue, to sort them according to their increasing earliest start times. Upon *EU* termination, task $t^{Trm}_{XuParnas}$ removes the *EU* from the run-queue it belongs to.

**Dynamic scheduling policy:** The dynamic scheduling policy we have implemented is the Earliest Deadline First (EDF) scheduling policy coupled with the Stack Resource Policy (SRP) [3] to avoid multiple priority inversions. The key idea of SRP is that when a task needs a resource, it is blocked at start time rather than later when it actually requests the shared resource. The main motivation for this early blocking is to save unnecessary context switches. Briefly, the SRP statically assigns a *preemption level*, noted $\pi(t)$ to each task $t$. Preemption levels are defined so that they are ordered inversely with respect to the order of relative deadlines.

The implementation of the EDF scheduling policy without the SRP is straightforward. The *App_tasks* scheduled according to the EDF policy are assigned to a single HADES run-queue, into which tasks are sorted by increasing deadlines. The EDF scheduling policy is managed by two *RTS_tasks* named $t^{Act}_{edf}$ and $t^{Trm}_{edf}$. Each time a task is activated, task $t^{Act}_{edf}$ adds the *EUs* of this task to the EDF run-queue according to their deadlines. Each time an *EU* terminates, task $t^{Trm}_{edf}$ removes it from the EDF run-queue. This implementation undergoes some modifications when the EDF scheduling policy is coupled with the SRP.

The EDF/SRP scheduling policy is managed by two *RTS_tasks* named $t^{Act}_{edf/srp}$ and $t^{Trm}_{edf/srp}$. Each time a task $t$ is activated, task $t^{Act}_{edf/srp}$ checks if $t$ can preempt the currently executing task $t_{cur}$. According to SRP, $t$ can preempt $t_{cur}$ if the following preemption condition is met : $deadline(t) < deadline(t_{cur})\ and\ \overline{\pi} < \pi(t)$, where $\overline{\pi}$ is the maximum of the current ceilings of all resources (see [3] for a detailed description of the SRP protocol). If the preemption condition is met, task $t^{Act}_{edf/srp}$ inserts the *EUs* belonging to $t$ at the beginning of the EDF/SRP run-queue; otherwise, $t$ is kept in a data structure containing pending task execution requests. Each time an *EU* terminates, task $t^{Trm}_{edf/srp}$ removes it from the EDF/SRP run-queue and scans the list of pending task execution requests to test the preemption condition and to modify the EDF/SRP run-queue if necessary.

### 3.3.2 Time-bounded reliable multicast

The problem of sending messages in a failure-prone environment has been subject to intensive investigation, yet relatively little research seems to address the problem of designing reliable multicast that are amenable to schedulability analysis. The HADES run-time support provides three time-bounded multicast protocols. The first one, called *time-bounded basic multicast* protocol, has been designed for a failure-free environment, whereas the others rely on the fault hypothesis given in Section 2.3. These two fault tolerant multicast protocols, described hereafter and named respectively *time-bounded atomic multicast* protocol and *time-bounded causal multicast* protocol ensure that any message sent by a correct site will be delivered by all correct destination sites by guaranteeing the *Validity*, *Integrity*, *Agreement* and *real-time* $\Delta$*-Timeliness* properties [10]. They only differ by the order in which messages are delivered : *temporal order* for the first one and *causal order* for the second protocol. The causal order ensures that messages are delivered in accordance with their precedence relationship. The temporal order of events in a distributed system is defined in terms of an omniscient external observer. This observer possesses a single reference clock and timestamps each event by this clock. So event $E_1$ temporally precedes event $E_2$ if timestamp($E_1$) < timestamp($E_2$). In simpler terms, if $E_1$ occurs before $E_2$, then $E_1$ temporally precedes $E_2$.

**Time-bounded atomic multicast protocol:** The HADES time-bounded atomic multicast protocol is based on the Time-Division Multiple Access (TDMA) strategy, and relies on the clock synchronization service to get locals clocks synchronized within $\gamma$ of each other. Time is divided into rounds, a round being composed of a succession of time slots. Each site $i$ gets $s_i$ slots per round to send its messages. The assignment of slots to sites is done statically; it is never modified even in case of group membership changes. Upon message submission, task *McastSend* timestamps the message with a unique identifier based on the local time of the site, and stores it in a sending buffer. Concurrently, a periodic *RTS_task* called $t_{send/recv}$ is activated during the time slots granted to the site. This task is in charge of broadcasting each message of the sending buffer on the channel, and receiving each broadcast message. Note that received messages are piggybacked in broadcast messages until all messages have been broadcast $\omega = f_{send} + 1$ times to tolerate up to $f_{send}$ successive omissions. For each message $m$, the $t_{send/recv}$ task invokes the task in charge of handling $m$ as soon as $m$'s timestamp satisfies the delivery conditions (respect of the temporal order). This protocol ensures that if a message $m$ is multicast at real-time $t$ (by invoking task *McastSend*), then no correct site invokes the destination task in charge of handling $m$ after real-time $t + \Delta$. The value of $\Delta$ is equal to $3((\omega+2).(2P+2\gamma+r)+\Delta_{trans}+2\gamma)$, with $P$ the period of task $t_{send/recv}$, $\Delta_{trans}$ the maximum network latency in a failure-free environment, $\gamma$ the precision of the clock synchronization algorithm, and $r$ the maximum response time of task $t_{send/recv}$. By taking, $\Delta_{trans} = 1ms$, $P = 10ms$, $r = 1ms$, $\gamma = 10ms$, and $\omega = 2$, we get $\Delta \simeq 555ms$.

**Time-bounded causal multicast protocol:** The HADES time-bounded causal multicast protocol ensures that messages are causally delivered. Compared to the time-bounded atomic multicast protocol, this protocol does not rely on synchronized clock, which offers better performance. This protocol is implemented using a single periodic *RTS_task* named $t_{send/recv}$. This task is periodically activated, and broadcasts each message of the sending buffer $\omega$ times. Upon receipt of a message $m$, $t_{send/recv}$ invokes the task in charge of handling $m$ when delivery conditions are satisfied (respect of causal order). To avoid infinite message collisions on the communication medium, a mechanism providing a unique access to the medium is required (e.g., in HADES, the ATM switch is used; in case where Ethernet were used instead of ATM, the medium access protocol CSMA/CD would guarantee a deterministic access [11]). The value of $\Delta$ is for this protocol is equal to $(3.\omega + 1).P + \Delta_{trans} + 2.r$. By taking the above numerical values, we get $\Delta \simeq 73ms$.

#### 3.3.3 Group membership service

The challenge when designing a group membership protocol for a hard real-time system is to fulfill the three following requirements : *(i)* to have a small and bounded failure detection latency, *(ii)* to have a small and bounded number of messages exchanged and *(iii)* to have a protocol that can be easily integrated into feasibility tests. The HADES run-time support comes with two different implementations of the group membership service that meet such constraints. The first implementation, called *strong* group membership, guarantees the *Vivacity*, *Uniqueness*, *Consistency* and *Real-time $\Delta$-Timeliness* properties [8]. For performance reasons, the second one, called *weak* group membership service, guarantees only the *Vivacity*, and *Real-time $\Delta$-Timeliness* properties. Both implementations periodically piggybacks heartbeat messages on application messages (in the current implementation, the $t_{send/recv}$ task is used to implement both the multicast and the group membership services). For space considerations, the following is devoted to the description of only the strong group membership protocol.

The strong group membership service is based on the same principle as the time-bounded atomic multicast service, i.e., a TDMA approach. Each site $p$ piggybacks on each application message the view it has of the sites group, by means of two vectors named $recv_p$ and $join_p$. Vector $recv_p$ is used to implement the heartbeat: $recv_p[i]$ is the latest clock value that $p$ knows that site $i$ sent. Vector $join_p$ is used to agree on when a site has restarted : when the current local time is at least $join_p[i]$, then $i$ has joined the group of operational sites. Upon receipt of a message on site $p$, vector $recv_p$ is updated and the message is forwarded to the group. This protocol guarantees that if at local time $t_p$, a site $p$ operational for more than $\Delta$ time units that has not received any heartbeat message from site $q$ for $\Delta$ time units, then $p$ excludes $q$ from its group view at local time $t_p$. The value of $\Delta$ is equal to the one exhibited by the time-bounded atomic multicast protocol (see Section 3.3.2). Thanks to the clock synchronization service, all correct sites exclude $q$ from their group view at time $t_q = t_p$. Similarly, a site $p$ initially crashed, and wishing to be reinserted in the group is guaranteed to be reinserted in the new group view by all correct sites at time $t_p + \Delta$, with $t_p$ the time at which $p$ broadcast its first heartbeat message ($join_p[p] = t_p + \Delta$). Upon configuration change, tasks that need to be notified of the configuration are invoked with the new group view as argument. Note that the time-bounded atomic multicast service altogether with the group membership service ensure the virtual synchrony property, i.e., the guarantee that the delivery of messages are totally ordered with respect to changes in the group's membership.

#### 3.3.4 Kernel adaption layer

The kernel adaption layer interacts with the underlying real-time kernel for activities related to the management of threads and interrupts. To port the HADES run-time support on a real-time kernel, one has to develop a kernel adaption layer that conforms to an interface made of four function-calls directly invoked by the dispatcher:

| Name | Description |
|---|---|
| KrnCreateContext | Allocates a context for a thread. |
| KrnSetContext | Sets the thread context (e.g., initializes the thread starting address) but does not start the thread yet. |
| KrnSwitchTo | Starts a thread or restarts it after a preemption. |
| KrnConnectIt | Connects an interrupt handler on a given interrupt level. |

The first three function calls allow the dispatcher to assign separate threads of control to concurrent *EUs* within a *HEUG*. The fourth function call allows the dispatcher to asynchronously invoke a *RTS_Task* upon occurrence of an interrupt (e.g. the wakeup *RTS_Task* of the time service).

Such an interface requires that the real-time kernel offers enough support *(i)* to implement a multithreading interface; *(ii)* to connect user-defined interrupt handlers. In addition the real-time kernel must provide enough support to guarantee the consistency of the dispatcher data structures in presence of interrupts, and this without any blocking. This last point is detailed hereafter. Up to now, the HADES kernel adaption service has been ported to CHORUS 3.1 and has been emulated on SOLARIS.

**Chorus kernel adaption layer:** CHORUS 3.1 [7] provides a real-time operating system that can scale down to small embedded platforms and scale up to distributed POSIX compliant platforms. It is built around a real-time executive module which provides the basic services mandatory for a real-time executive. The HADES environment has been ported on the *micro core* executive module. This module provides support for multithreaded applications, FIFO fixed-priority scheduling, basic synchronization and user-provided management of interrupts, traps and exceptions. Note that the *core* module, although it offers more functionalities than *micro core* (e.g., memory protection, dynamic memory management) has not been selected for predictability reasons (its code was far too large to be guaranteed predictable through manual code analysis). The HADES kernel adaption layer interface has directly been implemented above the *micro core* API.

The most difficult part was to guarantee the consistency of the dispatcher data structures on top of the *micro core* executive. One trivial solution could have been to simply mask interrupts (using *ix86* instructions *cli* and *sti*) during dispatcher activities. This solution was rejected since some dispatcher activities include CHORUS system calls at the end of which interrupts are unmasked. Instead, our solution relies on the *micro core* fixed-priority scheduler. It consists in assigning CHORUS priorities to thread execution according to the following policy : *(i)* threads allocated to dispatcher activities (i.e., *Sys_EUs*) are executed with the highest CHORUS priority; *(ii)* threads allocated to interrupt handlers are executed with a CHORUS intermediate priority. These threads are unblocked whenever an interrupt is triggered; *(iii)* threads allocated to *Code_EU* are executed with a low CHORUS priority. With such a priority assignment, an interrupt can be triggered while a *Sys_EU* is being executed, but its interrupt handler is executed only when the *Sys_EU* is terminated. This solution can be implemented on top of any real-time kernel exhibiting a fixed-priority scheduling policy.

**Solaris kernel adaption layer:** The HADES kernel adaption service has been ported on the SOLARIS operating system. This port aimed at validating the system temporal properties by simulating a worst-case execution scenario, as well as its functional properties. The HADES kernel adaption layer interface has directly been implemented above the SOLARIS thread library. More precisely, per each site is allocated one SOLARIS multi-thread process in charge of executing all the *App_tasks* and *RTS_tasks*. For the global system is allocated one SOLARIS multi-thread process in charge of *(i)* coordinating all the sites, *(ii)* simulating the global time, *(iii)* handling the local clocks drift and the messages transmission time, *(iv)* simulating failures.

### 3.4. Support for predictability

A key design principle in HADES was to provide a run-time system with a *predictable* behavior. An action is said to be predictable if its results and its worst case execution time (WCET) are known before it is executed. Predictability is crucial for the implementation of feasibility tests. Moreover, the more precise the WCET estimation, the less pessimistic the application feasibility test. Indeed, due to the complexity of determining worst case cost information, scheduling tests often use over-estimated WCETs of run-time support activities. While this behavior is safe, it often leads to a negative answer of the scheduling test, forbidding the execution of applications despite their actual feasibility. Providing a predictable run-time support requires that all its components be predictable. A WCET analyzer is currently being developed to partially automate kernel and *Code_EUs* WCET analysis. The following sections show how the design of all components of the HADES environment (i.e., services layer, dispatcher and real-time kernel) makes HADES predictable.

**Services layer predictability:** The predictable behavior of the service layer results from two aspects. The first one comes from the specification of all run-time support services according to the HADES task model. This model eases the determination of the temporal behavior of these services by requiring that sequential codes (i.e., *EUs*) be isolated from any synchronization, or distribution constraints. This makes possible a precise evaluation of *EUs WCET*, and thus an exact knowledge of any blocking time. The second aspect is related to the arrival law of these services. Except for a small set of *RTS_tasks* which are periodical (the $t_{send/recv}$ task of the multicast service, the clock synchronization and the time tasks), all the services are invoked explicitly or implicitly by the application code. This enables to carry their WCETs over to the execution cost of the application tasks. The periodic laws of the aforementioned *RTS_tasks* are directly integrated into the application feasibility test.

**Dispatcher predictability:** The dispatcher has been designed to handle very basic functionalities, which make its code small and easy to analyze. Beyond this feature, the dispatcher exhibits a number of characteristics that makes it predictable: *(i)* it does not contain any asynchronous or periodic activity; *(ii)* it does not use dynamic memory allocation (tasks arrival laws, resources, condition variables are known off-line); and *(iii)* it never blocks (see Section 3.3.4).

Such features enable to fully characterize the dispatcher with four WCETs : *(i)* $wcet_{term/start}$ to terminate an *EU* execution and to start the execution of the following ones; *(ii)* $wcet_{sys}$ to execute a *Sys_EU*; *(iii)* $wcet_{int}$ to take into account an interrupt and to start the execution of the appropriate interrupt handler; *(iv)* $wcet_{except}$ to substitute the execution of the current *Code_EU* by the exception handler. These four WCETs have been identified in our prototype and correspond to treatments that cannot be preempted. The integration of these costs into the application feasibility test is relatively easy. The $wcet_{term/start}$ and $wcet_{sys}$ costs are directly added to the application tasks costs, while the $wcet_{int}$ and $wcet_{except}$ are integrated as high priority sporadic tasks.

**Kernel predictability:** As the designers of real-time kernels generally do not provide any guarantee about predictability of their kernel, access to their source code is required. Thus, a study of the Chorus R3 kernel adopted in the HADES prototype, has been done. The ChorusR3 kernel source code has been analyzed through a verification of invariants, an analysis of function side effects, and an identification of WCETs. So far, code analysis has been done manually.

## 4. Related work

A number of systems have been designed to support dependable real-time distributed applications. MARS [15, 22] and MAFT [14] are two of them. However, MARS is not flexible: first it uses a single scheduling policy (static and periodical), and second it relies on specific hardware-intensive solutions for fault-tolerance and clock synchronization. In contrast, the HADES run-time support can be adapted to support multiple scheduling policies (see Section 3.3) and uses COTS hardware and software. MAFT proposes mechanisms to tolerate Byzantine failures in order to provide extremely reliable computations, and this, without sacrificing performances. However, for the same reasons as MARS, MAFT is not flexible. DELTA-4/XPA [4] proposes flexible fault-tolerance and communication protocols. Unfortunately, the execution environment used for its implementation is not predictable and the proposed fault-tolerance real-time protocols are not suited for strict real-time constraints. Among works in progress, the GUARDS European ESPRIT project [26] has objectives close to those of HADES. However, while HADES is more concerned with distributed applications and flexibility, GUARDS focuses on dependability of embedded systems. Furthermore, in GUARDS, hardware components are mainly developed for ensuring a safe interaction with controlled systems (e.g. train, aircraft, or submarine).

The HADES run-time support cannot be directly compared with real-time kernels, like QNX [12] and CHORUS [7]. While some of them exhibit a predictable behavior, they are generally not designed to support failures or to adapt to application needs (e.g., support for multiple scheduling policies). However, they can be used as HADES base real-time kernel so long as they offer enough functionalities to implement the HADES kernel adaption layer.

The structuring principles that where adopted in HADES to meet the *flexibility* property are very similar to the ones proposed by micro kernel designers to specialize their kernels [16]. The difference comes mainly from the class of services that can be specialized. In micro kernels, services that can be specialized are general-purpose operating system services (e.g., memory management specialization through paging services, communication services), while in HADES, such services are representative of services found in run-time supports for dependable applications (e.g., real-time scheduling, time-bounded communications, clock synchronization). More recent work on kernel specialization gave birth to highly customizable operating systems kernels, such as SPIN [5] or EXOKERNEL [13], but these kernels do not address predictability issues.

CORBA [19] is a standardized middleware architecture for distributed object computing on heterogeneous environments. It eases the development of distributed services by providing features to interconnect applications and services. TAO [23] is a predictable implementation of CORBA, providing facilities to support hard real-time requirements. HADES, like TAO relies on a predictable middleware layer. However the two platforms differ by the fact that HADES does not rely on an object-based approach to develop applications, by the fact that HADES does not conform to any standard like CORBA and by the fact that to our knowledge, TAO is not fault-tolerant.

## 5. Conclusion

In this paper, we have presented a flexible run-time support for distributed dependable real-time applications. Its flexibility comes from the specification of a set of services with a well-defined interface. These services can evolve without having to rewrite the run-time support. The implementation of any service can be replaced by a service with different properties as far as it conforms to the service interface (*extensibility* property). This allowed us to develop a range of service implementations adapted to the specific constraints of the applications and the underlying hardware (*adaptability* property). So far, we have designed (i) a panel of scheduling policies suited to different applications requirements (a static planning-based scheduling policy and a dynamic scheduling policy); (ii) a set of multicast services suited to different fault models and ensuring different ordering properties; (iii) membership services with different properties; (iv) two kernel adaption layer to port the HADES run-time support on CHORUS and SOLARIS.

The HADES run-time support described in this paper has been implemented. The prototype with CHORUS uses four PCs based on PENTIUM 90 processors and an ATM switch. A small demonstration of the HADES prototype can be found on [20].

Future work concerns the adaptation of the HADES run-time support (e.g. replication strategy, scheduling policy) to dynamic systems where workload is variable [9]. In addition, one objective is to automate the run-time support predictability analysis through the construction of a WCET analyser.

## References


[1] R. Agne. Global cyclic scheduling : A method to guarantee the timing behavior of distributed real-time systems. *The Journal of Real-Time Systems*, 3(1):45–66, Mar. 1991.

[2] E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. Hades: A middleware support for distributed safety-critical real-time applications. In *Proc. of ICDCS-18*, pages 344–351, May 1998.

[3] T. Baker. Stack-based scheduling of real-time processes. *The Journal of Real-Time Systems*, 3(1):67–99, 1991.

[4] P. Barrett, A. Hilborne, P. Verissimo, L. Rodrigues, P. Bond, D. Seaton, and N. Speirs. The delta-4 extra performance architecture (xpa). In *Proc. of FTCS-20*, pages 481–488, June 1990.

[5] B. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *Proc. of SOSP-15*, volume 29(5) of *OSR*, pages 267–284, Oct. 1995.

[6] P. Chevochot, I. Puaut, and G. Cabillic. A flexible environment for automatic replication and execution of distributed safety-critical real-time applications. TR 1183, IRISA, Apr. 1998.

[7] Chorus Systèmes. Chorus/classix R3 technical overview. TR 96-119.91, May 1996.

[8] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, 1991.

[9] O. Gonzàles, H. Shrikumar, J. Stankovic, and K. Ramamritham. Adaptive and fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proc. of 18th RTSS*, San-Francisco, USA, 1997.

[10] V. Hadzilacos and S. Toueg. *Distributed systems, fault tolerant broadcasts and related problems*. Addison-Wesley / ACM Press, 1993.

[11] J. Hermant, G. L. Lann, and N. Rivierre. A general approach to real-time messages scheduling over distributed broadcast channels. In *IEEE Conf. on Emerging Technologies and Factory Automation*, Oct. 1995.

[12] D. Hildebrand. An architectural overview of QNX. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Apr. 1992.

[13] M. Kaashoek et al. Application performance and flexibility on exokernel systems. In *Proc. of SOSP-16*, volume 31(5) of *OSR*, pages 52–65, Oct. 1997.

[14] R. Kieckhafer, C. Walter, A. Finn, and P. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Trans. on Computers*, 37(4):398–405, Apr. 1988.

[15] H. Kopetz et al. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9:25–40, Feb. 1989.

[16] J. Liedtke. Toward real microkernels. *Comm. of the ACM*, 39(9):70–77, Sept. 1996.

[17] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environmene. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[18] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa. Implementing a predictable real-time multiprocessor kernel - the Spring kernel. In *Proc. of the 7th IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.

[19] OMG. *The Common Object Request Broker: Architecture and Specification*, July 1996.

[20] Projet SOLIDOR. Hades: A middleware support for distributed safety-critical real-time applications. WEB Document, 1998. http://www.irisa.fr/solidor/work/hades.html.

[21] K. Ramamritham, J. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. on Parallel and Dist. Systems*, 1(2):184–194, Apr. 1990.

[22] J. Reisinger, A. Steininger, and G. Leber. *Predictably Dependable Computing Systems*, chapter The PDCS Implementation of MARS Hardware and Software, pages 209–224. Springer-Verlag, 1995.

[23] D. Schmidt, D. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Comm. Journal*, 1997.

[24] M. Spuri. Holistic analysis for deadline scheduled real-time distributed systems. Technical Report RR-2873, INRIA, France, Jan. 1996.

[25] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessors and Microprogramming*, 40:117–134, 1994.

[26] A. Wellings, L. Beus-Dukic, A. Burns, and D. Powell. Genericity and upgradability in ultra-dependable real-time architectures. TR LAAS-R-96417, LAAS, Nov. 1996.

[27] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3):360–369, Mar. 1990.