HAL
open science

Hades: A Middleware Support for Distributed Safety-Critical Real-Time Applications

Emmanuelle Anceaume, Gilbert Cabillic, Pascal Chevochot, Isabelle Puaut

To cite this version:

Emmanuelle Anceaume, Gilbert Cabillic, Pascal Chevochot, Isabelle Puaut. Hades: A Middleware Support for Distributed Safety-Critical Real-Time Applications. [Research Report] RR-3280, INRIA. 1997. inria-00073408

HAL Id: inria-00073408

https://inria.hal.science/inria-00073408v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Hades: A Middleware Support for Distributed Safety-Critical Real-Time Applications

E. Anceaume, G. Cabillic, P. Chevochot, I. Puaut

N° 3280

October 1997

THÈME 1

Rapport de recherche

ISSN 0249-6399

Hades*: A Middleware Support for Distributed Safety-Critical Real-Time Applications

E. Anceaume, G. Cabillic, P. Chevochot, I. Puaut

Thème 1 — Réseaux et systèmes
Projet Solidor

Rapport de recherche n3280 — October 1997 — 30 pages

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Abstract: Most distributed safety-critical real-time systems designed in the past have been specialized to meet the particular requirements of the application domain to which they were targeted. This approach led to specific, inflexible, dedicated and non-reusable solutions, often based on specialized hardware. This report presents an overview of HADES which provides a set of flexible tools built on top of off-the-shelf hardware, and designed to help in the construction of distributed safety-critical real-time applications. In order for HADES to support the execution of the widest range of applications, we have followed a rigorous methodology based on *i)* the separation of services dedicated to a specific application domain (scheduling policy) from services providing a range of robustness properties common to a large spectrum of application domains (e.g. task dispatching, fault detection, clock synchronization, monitoring); *ii)* the provision of a precise cost information induced by all these services in order to increase the accuracy of the application feasibility test.

Key-words: Real-time, Safety-critical, Availability, Distribution

(Résumé : tsvp)

This work is partially supported by the french Department of Defense (DGA/DSP), #96.34.106.00.470.75.65., and by Dassault Aviation, #CA/157/97 convention 197.C.069.00.31.30.30.1.2

* HADES stands for Highly Available Distributed Embedded System (in the Greek mythology, HADES is the King of Hell.).

Hades : Support pour applications temps-réel à sûreté critique

Résumé : La plupart des systèmes temps-réel à sureté critique développés dans le passé sont dédiés à un domaine d'application et une politique d'ordonnancement donnés. Cette approche entraîne souvent le développement de solutions lourdes, rigides et reposant fortement sur le développement de matériel spécifique. Pour réduire de manière significative les coûts d'évolutions de tels systèmes, l'objet du projet HADES est de fournir un ensemble de services permettant de développer des applications distribuées ayant des contraintes temporelles et de disponibilité au-dessus de matériel *sur étagère*. Ce document donne une vue d'ensemble des objectifs et de la méthodologie que nous avons développée pour la construction d'HADES. Cette méthodologie est basée sur *(i)* la fourniture d'une part d'un ensemble de services dédiés à un domaine applicatif donné (i.e., liés à une politique d'ordonnancement donnée) et d'autre part à un ensemble de services exhibant des propriétés de robustesse communes à un large spectre de domaines d'applications (e.g. allocation, détection de fautes, synchronisation d'horloges, surveillance); *(ii)* la fourniture du coût, en terme de durée d'exécution, de tous ces services de façon à accroître la précision du test de faisabilité de l'application.

Mots-clé : Temps-réel, Sûreté critique, Disponibilité, Distribution

1 Introduction

Safety-critical real-time application domains in the last thirty years have expanded from space rockets to flight control systems, nuclear power plants, stock exchange, and medical equipments. Designing systems that support such demanding applications is hard because a triple goal must be reached. First, such systems have to be *highly reliable* since, for example, the maximum acceptable probability of failure typically ranges from 10^{-4} to 10^{-10} per hour [RSL95, KWFT88]. Second, these systems have to meet *strict timeliness* requirements since their typical response times range from $1ms$ to $100ms$, depending on the criticality of the application [MRS+90, RSL95]. Third, these systems must enforce *data consistency* due to the concurrent executions of multiple tasks, especially in a distributed environment. Reaching this triple goal is hard and led so far to solutions that met only the particular requirements of their application domain. Such solutions are thus specific, inflexible and dedicated to a single application domain. Furthermore, they are often based on specialized and costly hardware, making the implemented software seldom reusable in a different context [RSL95, KWFT88].

To significantly decrease the global cost of such solutions, HADES provides a toolkit offering basic services needed by most distributed safety-critical real-time applications. The services offered implement basic functionalities (e.g. scheduling policy, time-bounded and reliable communication) allowing the application programmer to focus only on her/his application, rather than on low-level details. These services are flexible, that is, they support the execution of a wide range of distributed safety-critical real-time applications, and cheap in the sense that they are built over off-the-shelf-components (COTS). To enforce HADES flexibility, services fall into two categories:

- Services that are dedicated to a specific application domain, that is, dependent on application tasks characteristics, and

- Services providing a range of properties shared by a large spectrum of application domains.

Services pertaining to the first category are relevant to scheduling policies. Their design and properties highly rely on application-specific features like task arrival laws, task priorities, preemption policy, and resource access patterns. For example, HADES provides dynamic and static scheduling policies (e.g. Earliest Deadline First (EDF), Rate Monotonic (RM) [LL73]) as well as planning-based scheduling policies [Xu93, RSS90, Agn91]). The second category encompasses a large range of generic services exhibiting robustness properties shared by many safety-critical applications. Example of such services are task dispatching, fault detection, time-bounded reliable multicast and broadcast, clock synchronization, and monitoring services.

Separating application-dedicated from application-independent services gives the key to HADES flexibility. First, the provision of various static and dynamic scheduling policies enables to support a large range of safety-critical applications, each of them having a different degree of criticality in terms of response time. Second, the design of generic services guaranteeing reliability, timeliness and data consistency properties makes transparent to the application designer fault tolerance, concurrency control and monitoring handling.

Another important feature of HADES is that reliability, real-time and distribution properties are studied as a whole. This approach, also followed in [GMM97] and in [Lan91], recommends that the algorithmic aspects involved when designing distributed safety-critical real-time systems should not lead to selecting an algorithm appropriate for each of these properties, but rather to selecting cooperative or compatible algorithms. A trivial example of incompatibility between algorithms is the use of a lock-based concurrency control algorithm together with an EDF scheduling algorithm. These two algorithms may be in conflict when they decide which is the next task to schedule. Providing a toolkit of services that guarantees the compatibility between any composition of the algorithms is a key point of our approach.

The remainder of the paper is organized as follows. Section 2 details the design goals of HADES and the methodology developed to reach them. Section 3 describes the generic task model used for structuring the services proposed by HADES, and details the rules governing task execution. Section 4 characterizes the costs of the proposed services. A complete example of how to utilize HADES for executing applications that use an EDF scheduling policy is given in Section 5. HADES is compared with related projects in Section 6. Finally, we conclude in Section 7.

2 Overview of HADES

HADES intends at supporting a large spectrum of distributed safety-critical real-time applications — from small embedded applications to complex safety-critical applications. This exigency of flexibility covers the objectives presented hereafter.

2.1 Objectives

HADES is a toolkit for building distributed safety-critical real-time applications. Its main objective is to be flexible enough in order to encompass a wide range of robustness services that include guaranteed real-time behavior, high availability through transparent fault-tolerance software, and distribution handling. More precisely, HADES goal is to provide services exhibiting the following properties:

- **Timeliness.** This property states that application deadlines are always met. Since HADES aims at supporting a large spectrum of distributed safety-critical real-time applications, our objective is to support both soft and hard deadlines.

- **Availability.** Guaranteeing to applications high availability through transparent fault-tolerance software in the presence of faults — ranging from crash, omissions and coherent values failures for processors to Byzantine failures for clocks, performance and omission failures for the communication network —

is an objective of HADES. Furthermore, we aim at verifying the correctness of applications execution and thus at guaranteeing a high coverage factor of each assumption, that is a high probability that an assumption is not violated at run-time.

- **Distribution.** Managing consistency of distributed data so as to handle the inherent distribution of components (e.g. CPUs, captors, actuators) and their interaction is the third property to be exhibited by HADES services.

The goal of any distributed safety-critical real-time system is to guarantee each of these properties. An approach consists in selecting or designing one algorithm that is deemed appropriate for each of these properties taken in isolation. It has the advantage of being simple, but has the drawback to lead to potentially incompatible algorithms. Indeed, while each algorithm may be safe in isolation, their combination may exhibit an unsafe behavior because of incompatibilities between each other. HADES approach is to design cooperative and compatible algorithms to provide the sought properties altogether, and thus to avoid unsafe and antagonist applications behavior.

2.2 Methodology

In order to reach these objectives, HADES adopts the following methodology:

- The separation of services dedicated to a specific application domain (scheduling policy) from generic services providing a range of robustness properties common to a large spectrum of application domains (e.g. task dispatching, fault detection, clock synchronization, monitoring),

- The provision of a precise *cost information*, in terms of worst case execution time, for all activities induced by the execution of the provided services (including the cost of the kernel). Such provision allows the scheduling analysis

to consider not only the applications tasks but also the underlying services to check the global feasibility of the applications.

This methodology led us to design HADES as a middleware software layer running on COTS components (see Section 2.2.1). The method used for providing cost information is sketched in Section 2.2.2.

2.2.1 HADES structure

HADES is designed as a *middleware* software layer, depicted in gray in Figure 1. HADES executes on top of a Commercial-Off-The-Shelf (COTS) real-time kernel running on off-the-shelf hardware (network of mono processor machines). The key motivation for using COTS is to cut financial and development costs while using the latest technologies [SAF97]. Any real-time kernel providing standard process management mechanisms (priority-based preemptive scheduling, interprocess synchronization, separate address spaces) and a predictable behavior can be used by HADES as underlying kernel. Our current prototype is being developed on top of the ChorusR3 kernel[1] [Gie95], with an ATM network of Pentium workstations as the underlying hardware architecture.

In HADES, mechanisms dedicated to a specific application domain are isolated in a software component, named *scheduler* in the sequel, implementing a given scheduling policy. A scheduling policy includes a scheduling algorithm which defines the order in which tasks are executed. In Figure 1, two schedulers, among others, are depicted, RM standing for Rate Monotonic, and EDF standing for Earliest Deadline First [LL73]. For example, the RM scheduling algorithm states that a task is assigned a priority that depends on its period. A scheduling policy may also include a scheduling test, analyzing either statically or dynamically whether a set of tasks can meet its timing constraints or not. A scheduler requires tasks to be

[1]Thanks to the modular structure of the ChorusR3 kernel, we have been able to identify and use only the system calls that exhibit a predictable behavior.

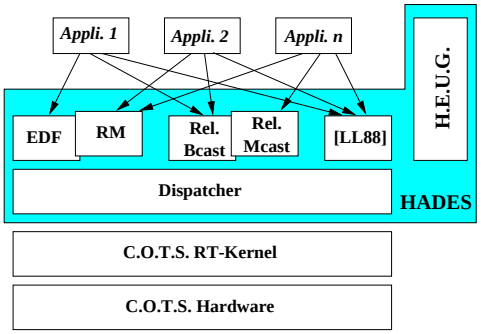

Figure 1: Internal structure of HADES

designed according to a particular *task model* which defines the way tasks are structured (e.g. presence of resource sharing, precedence constraints) and the properties that must be met during their execution (e.g. atomicity, isolation). Any scheduler – dedicated to a specific application domain and assuming a specific task model – can be implemented thanks to the definition in HADES of a generic dispatcher and a set of services exhibiting fault-tolerant and timely behavior.

The intent of the set of services is to provide a wide range of facilities required for executing distributed safety-critical real-time software whatever its timeliness and criticality requirements are. Some of the services provided by HADES, but not necessarily depicted on Figure 1, will encompass *(i)* time-bounded reliable communication primitives (time-bounded point-to-point communication, time-bounded multicast/broadcast) drawn respectively on Figure 1 as *Rel. Bcast* and *Rel. Mcast*; *(ii)* replication services implementing passive, active and semi-active replication [Pol96]; *(iii)* consensus service; *(iv)* persistent storage service; *(v)* dependency tracking service [NMT97]; and *(vi)* clock synchronization service (the clock synchronization algorithm designed in [LL88] is drawn on Figure 1).

The purpose of the *dispatcher* is to provide task dispatching mechanisms and task monitoring services that are generic, in the sense they are not dedicated to a specific application domain and scheduler. The dispatcher allocates resources –

including CPU – to tasks, handles priority conflicts, and monitors tasks execution progress in order to detect the violation of safety, liveness or timeliness properties (e.g. deadline exceeding, deadlocks). It also includes low-level fault-tolerance mechanisms (e.g. state capture, switching of modes of operation in case of failure [Mos94]). Unlike the approaches adopted in [KWFT88, KDK+89], in which the scheduler and the dispatcher form a single software component, the two components have been separated in HADES which makes feasible the support of multiple scheduling policies.

We introduce in HADES a generic task model defining any task (application task, service and scheduler) as a directed acyclic graph, named *HEUG* for *Hades Elementary Unit Graph*, associated to synchronization and timing attributes. Services are directly designed in terms of *HEUGs*, whereas for application tasks HADES provides the *HEUG* model to the application designer, so that she/he can translate them into *HEUGs*. The benefits of defining a uniform task model are twofold. First, all tasks can be integrated into the scheduling test, which leads to a feasibility test the most accurate possible since it encompasses not only application tasks but also all the underlying tasks necessary for the application execution (see Section 4). Second, it eases interactions between tasks and provides an adequate model for the dispatcher to implement monitoring tools and fault-tolerance mechanisms. The structure of *HEUGs* and their use during task execution are detailed in Section 3.

Note that the cohabitation of applications managed by different schedulers requires to take into account resource sharing among these applications. To tackle this problem, one can for instance encompass all these applications into a global scheduling test, or restrict the cohabitation between a single scheduler implementing a feasibility test and any number of *best-effort* schedulers. Note that although the first solution is quite appealing in terms of guaranteed timing properties, it should lead to a rather complex study in terms of scheduling theory, which is circumvented in the second approach.

2.2.2 Predictability and costs

A key design principle in HADES is to provide services with *predictable* behavior. An action is said to be predictable if its results and its duration can be foreseen before it is executed [HWS95]. The provision of predictable services is crucial for the implementation of feasibility tests. For instance a critical application task can miss its deadline and thus have catastrophic consequences only because a low level activity takes more time than expected. Providing predictable services implies that the structure of all software must be known a priori, and that *all* levels of the system must be predictable.

The methodology followed in HADES is to guarantee that any activity (induced by the execution of the scheduler, services and kernel) is predictable. As the designers of COTS real-time kernels generally do not provide any guarantee about predictability of their kernel, access to their source code is required. Predictability of kernel mechanisms in HADES is guaranteed by an analysis of the kernel source code, through a verification of invariants, an analysis of function side effects, and an identification of worst-case execution times.

The provision of predictable activities requires to give a precise cost information — the worst case execution times — engendered at all levels of the architecture. In HADES these costs are classified into two categories according to their dependence on the application tasks (see Section 4). This classification is used to integrate all these costs into the scheduling test. The more precise the information is, the less pessimistic the application feasibility test is. Indeed, due to the complexity of determining cost information, scheduling tests often encompass over-estimated worst case execution time of operating system activities. While this behavior is safe it often leads to a negative answer from the scheduling test, forbidding the execution of the application in spite of its actual feasibility.

3 HADES task model

The execution of any activity in HADES (scheduler, services, application tasks) relies on the introduction of a task model that is generic in the sense that it is independent of any application domain. The definition of a generic task model is delicate, since it must compromise among several issues. It must provide higher level abstractions than those offered by real-time kernels so as to ease the development of schedulers. It must also be generic enough to support different kinds of distributed safety-critical real-time applications requirements. Another important point is that enough information must be maintained at run-time in order to implement monitoring tools. Finally, a task model must have a precise semantic that makes possible the integration of costs induced by system activities and application tasks into any scheduling test. This set of design constraints has led to the task model described in the following paragraphs.

3.1 Task model syntax

The proposed task model relies on the definition of tasks as directly acyclic graphs (*HEUGs*). A *HEUG* as defined hereafter includes synchronization constructions (Section 3.1.1), specifies timing requirements (Section 3.1.2) and exception handling constructions. For space considerations, only the two first points are presented in this section.

Task: A task is defined as a finite set of *Elementary Units (EUs)* connected to each other by precedence constraints. An *EU* can either be a sequence of code (*Code_EU*) or a call to another task (*Inv_EU*). The partial ordering among the *EUs* is modeled as a directed acyclic graph (*HEUG*).

Code elementary unit: A *Code_EU* consists of a sequence of code (*action*), assigned to a statically-defined processor, and accessing a set of resources local to the

processor on which this *Code_EU* executes. Neither synchronization nor resource allocation/release can be required by an action (see Section 3.1.1), and its designer must guarantee that its worst case execution time, named w in the sequel, can be determined.

Invocation elementary unit: An *Inv_EU* of any task *T* is a request to execute *T*. A task invocation may be either synchronous or asynchronous. A synchronous task invocation inv_eu_i of task *T* ends when the invoked task *T* has finished executing. An asynchronous task invocation inv_eu_i immediately ends without waiting for the termination of the invoked task *T*.

Precedence constraint: A precedence constraint from eu_i to eu_j (with eu_i to eu_j any two $Code_EU$ or Inv_EU) means that eu_j is allowed to start its execution if and only if eu_i has finished its own execution. Precedence constraints can carry parameters used to transfer data between *EUs*. A precedence constraint is said to be *local* if eu_i and eu_j are assigned to the same processor, while it is said to be *remote* otherwise. A remote precedence constraint models the invocation of a task $t_{network}$ implementing the communication protocol of a particular hardware and software configuration. Task $t_{network}$ uses a set of resources (i.e., embedded CPUs of the involved network cards, network hardware, DMAs, CPUs) and controls concurrent accesses to the network hardware.

Since network management is modeled as an independent task, applications can be designed independently of the communication protocol and the network hardware actually used. Moreover, the way communications are integrated into the scheduling test is free. For instance,one can choose either to implement an end-to-end scheduling test that integrates application tasks and network management, or use two separate scheduling tests. Finally, modeling the network as an independent task allows task $t_{network}$ to be assigned to parameters specific to a particular communication protocol, as for example, the priority at which the protocol executes.

3.1.1 Synchronization

In addition to precedence constraints, that constitute some kind of synchronization between *EUs*, *EUs* can synchronize themselves using either shared resources or condition variables.

Resource: A resource is defined as any hardware or software component required to execute an action, and is local to a processor. Examples of resources are processors, logical resources (e.g. locks), and peripheral devices (e.g. sensors, actuators). Restrictions may be placed on the use of resources to maintain their consistency. Traditional access modes (*shared* and *exclusive*) are defined to control the simultaneous use of the resource and serve to identify operational constraints for the purpose of resource allocation and scheduling.

Condition variables: A condition variable is a system-wide boolean variable that can be cleared and set. By definition a *Code_EU* can wait for a condition variable to be true only before beginning its execution.

3.1.2 Timing constraints

A given task can be activated multiple times. In HADES, requests to activate a task instance can be triggered by an *Inv_EU*, the expiration of a timer or when an interrupt is triggered. The arrival law of activation requests for one task, used by HADES dispatcher for its monitoring activity, may follow one of the laws enumerated below.

- *Periodic*: two successive activation requests of a given task are separated by a fixed amount of time called the *period*.
- *Sporadic*: a minimum amount of time called the *pseudo-period* separates the arrival of successive activation requests of a given task.
- *Aperiodic*: an arbitrary delay separates two activations of a given task.

Task timing constraints: Each task has a deadline *D* before which the task execution must be completed. This deadline is relative to the date of the task activation request.

EU timing constraints: A *Code_EU* has a set of timing attributes that are used by HADES dispatcher to handle task execution: priority (*prio*), preemption threshold (*pt*), and earliest start time (*earliest*). Other timing attributes such as latest start time (*latest*) and deadline (*D*) are used by HADES dispatcher for its monitoring activity.

The priority of any $code_eu_i$ belongs to the interval $[prio_{min}, prio_{sys}[$. The higher priority level $prio_{sys}$ is reserved for kernel mechanisms. Priority $prio_{min}$ is the lowest priority level among all the application tasks priority levels. Attribute $prio(code_eu_i)$ can be assigned either statically or dynamically according to the used scheduling policy. A static priority assignation can be used to implement static priority-based scheduling algorithms like RM [LL73]. A dynamic priority assignation can serve at implementing dynamic scheduling algorithms. In this case, dispatcher and scheduler cooperate for managing these dynamic priorities (see Section 3.2.2 for an example). Dynamic priority assignation can also be used to avoid priority inversions when defining services. This is done by dynamically setting the priority of services to the one of the actions that invoked them. Attribute $pt(code_eu_i)$ serves at controlling preemption of $code_eu_i$. By definition, actions with a priority higher than $pt(code_eu_i)$ can preempt $code_eu_i$, while the others cannot. By default, during the execution of kernel calls, we have $pt(code_eu_i) = prio_{sys}$, meaning that kernel calls cannot be interrupted by application tasks. Timing attribute $earliest(code_eu_i)$ defines the earliest start time of $code_eu_i$, i.e., the date before which $code_eu_i$ is not allowed to start its execution. Like attribute *prio*, *earliest* can be assigned to a *Code_EU* either statically or dynamically. These two kinds of definitions serve respectively at implementing static and dynamic planning-based scheduling algorithms.

3.2 Task execution

Task execution is managed by HADES dispatcher, whose purpose is detailed in Section 3.2.1. The way the dispatcher interface can be used to implement various scheduling policies is introduced in Section 3.2.2.

3.2.1 Dispatcher

The main purpose of the dispatcher is the allocation of resources for the execution of tasks (both application and HADES tasks). Resource allocation takes into account the synchronization and timing attributes of the tasks (priority, precedence constraints, and resource sharing). The dispatcher uses a distributed set of threads managed by the underlying kernel to execute a task instance, a given thread being dedicated to the execution of one and only one *Code_EU*. The dispatcher uses a priority-ordered queue of threads (*Run Queue*). Thread t_i is said to be *runnable*, and is inserted in the *Run Queue* iff the following four conditions are met:

- the threads that t_i must wait for, due to precedence constraints, have finished their execution,
- all the resources needed by t_i can be granted to t_i,
- all the condition variables t_i must wait for are set,
- the current time is higher than $earliest(t_i)$.

Thread t_i is said to be *running* (i.e., resources (including CPU) are allocated to it) iff:

- t_i is runnable, and
- $prio(t_i)$ is the highest priority among all the *runnable* threads, or for all *runnable* threads t_j with $prio(t_j) > prio(t_i)$, we have $prio(t_i) > pt(t_j)$.

Another important purpose of the dispatcher is to monitor the execution of threads in order to detect the occurrence of certain events. Examples of such events

are *(i)* deadline violation; *(ii)* violation of the arrival law of task activation requests; *(iii)* early thread termination (i.e., the effective execution time of a thread is lower than *w*) and orphan thread execution (these two events can be utilized to re-use released resources); *(iv)* deadlocks; and *(v)* network omission failures based on the observation of remote precedence constraints. Note that at our knowledge no existing real-time environment has implemented all these monitoring activities.

3.2.2 Interaction between scheduler and dispatcher

In HADES, different scheduling policies can be constructed via an interaction between the dispatcher and scheduler. Every scheduler is modeled by a task with a statically-defined priority. For each scheduler this interaction is embodied by a dispatcher primitive and a FIFO queue (shared by the dispatcher and the scheduler). The primitive allows to modify the earliest start time of a thread (if any) and/or its priority, and is called by the scheduler each time a modification of the *Run Queue* is required. The shared FIFO queue is used by the dispatcher to notify the scheduler of events to be taken into account by its scheduling policy. Such notifications concern thread activations, denoted by *Atv*, thread terminations, denoted by *Trm*, and requests to access/release shared resources, denoted respectively by *Rac* and *Rre*[2]. The scheduler blocks until a notification is inserted into the shared FIFO queue and treats it according to its scheduling policy.

An example of interaction between the dispatcher and scheduler is depicted in Figure 2. In the example, the scheduler implements an EDF scheduling policy. The example considers the execution of two threads t_1 and t_2, the scheduler being executed by thread t_{edf} assigned to the highest priority. At the very beginning of the execution, thread t_1 is running. Upon activation of thread t_2, the dispatcher inserts the notification Atv_{t2} into the shared FIFO queue. Thread t_{edf} retrieves Atv_{t2} and

[2] Although not detailed here for space considerations, *Rac* notification allows the implementation of protocols aimed at avoiding multiple priority inversions, such as *Priority Ceiling Protocols* (PCP) [CL90] or *Stack Resource Policy* (SRP) [Bak91].

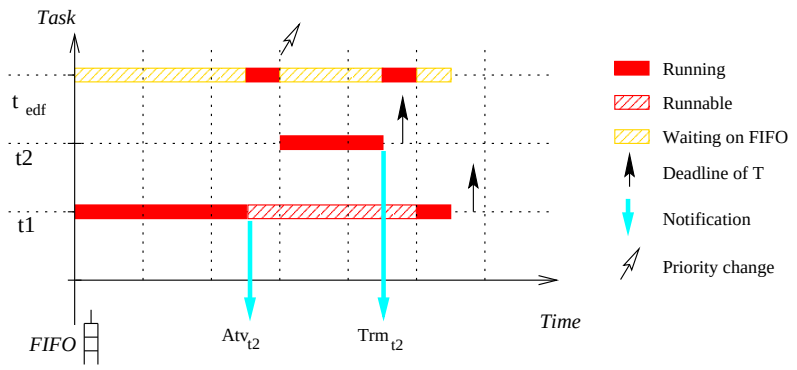

Figure 2: Cooperation between scheduler and dispatcher to implement EDF

treats it according to the EDF scheduling policy. In this example, the deadline of t_2 being lower than the one of t_1, the scheduler sets the highest priority to t_2 and decreases the one of t_1. This is achieved by calling the dispatcher primitive. When t_2 finishes its execution, the scheduler adds the notification Trm_{t2} in the queue, which is ignored by EDF. The thread with the highest priority (in the example t_1) can then continue to execute.

3.3 Discussion

As already mentioned, defining a generic task model for distributed real-time systems is a difficult task. Arguments about the soundness of the proposed task model are given hereafter.

The notions of precedence constraints and shared resources being used in a large number of scheduling policies, their introduction into a generic task model is almost mandatory. However, the use of only these two synchronization tools is not adequate for structuring HADES services. For instance, by solely using a task model with resources and precedence constraints, it is not possible to dynamically activate a task when a particular event occurs, or to implement the remote procedure call

mechanism, or even to built a producer-consumer scheme. This is the reason why task activations and condition variables have been introduced[3].

Although ACID properties of transactions can be required by some schedulers, we have opted no to integrate them since they can be implemented using our low-level synchronization entities (e.g. resources) and a small set of HADES services (e.g. state capture). Moreover, the compatibility of ACID properties with real-time constraints remains an open issue [KS95].

Implementation of a guaranteed scheduling algorithms requires that the scheduler designer exactly knows off-line the duration during which resource access is required, and blocking time due to resource sharing. This feature explains why we forbid the use of synchronizations inside the code of actions. It makes feasible the characterization of worst-case execution time of actions, and thus makes possible the implementation of guaranteed scheduling algorithms.

Finally, some of the attributes of our *HEUG* model are crucial for the implementation of monitoring tools. Precedence constraints allow to build low-cost fault detection and orphan detection. Moreover, deadlines, arrival laws, worst execution times and latest start time attributes enable the detection of timing violations.

The genericity of HADES task model allowed us to develop a large spectrum of scheduling policies. So far, priority-based scheduling policies (RM, EDF [LL73]), planning-based scheduling policies (Spring scheduling policy [RSS90]) and mechanisms aimed at avoiding multiple priority inversions (PCP [CL90] and SRP [Bak91]) have been designed. For space considerations, a full presentation of the design of all these scheduling policies cannot be afforded. A detailed example is given in Section 5.

[3]Note that other low-level synchronization mechanisms like semaphores could have been introduced instead of condition variables.

4 Determination of execution costs

An objective of HADES is to characterize the cost of all activities — induced by the execution of HADES services, dispatcher and underlying kernel — so that they can be integrated into the scheduling test of any scheduler. This analysis should be as precise as possible, so that the integration of these costs into a scheduling analysis tool should not lead to overly pessimistic scheduling tests.

HADES activities (i.e., dispatcher and kernel activities) have been divided into two categories. The first category encompasses all *dispatcher activities*, i.e., all activities engendered directly, explicitly or implicitly from the code of the application task (e.g. *signal* done on a condition variable, task termination, verification of a local precedence constraint, context switch, management of execution queues). The second category consists of all the background *kernel activities*, such as device drivers or clock interrupt handling.

This separation of HADES activities in two categories simplifies the cost determination of these activities. Indeed, beyond the fact that it is difficult to precisely characterize the cost of any kind of activity, it is even more difficult to precisely determine at which frequency these activities are produced. Thus, since dispatcher activities recur with the same frequency as the application tasks they depend on do, our idea is to carry the cost of all dispatcher activities over to the execution cost of application tasks. Accordingly, these activities do not engender new task arrival laws and their integration into a scheduling test is simplified.

4.1 Dispatcher activities

By construction, any dispatcher activity is not distributed, does not contain internal synchronization and recurs with the same frequency as the application task it depends on does. Thus, any dispatcher activity is fully modeled by its worst-case execution time *w* and its priority $prio_{sys}$. Attribute *w* of any dispatcher activity is

determined in HADES either analytically or by running worst-case scenario benchmarks.

A prototype of the dispatcher has been implemented in order to identify all activities and their resulting costs. The identified execution costs are given below. Constant c_{local_dep} is the execution cost of a local precedence constraint; it includes the cost of data copying and the cost of a context switch. Constant c_{remote_dep} is the cost of transmitting data to the communication protocol when executing a remote precedence constraint. It does not include the cost of sending the message, which is not achieved by the dispatcher itself, but by an independent task $t_{network}$ (see Section 3.1). Constants c_{action_beg} and c_{action_end} include the costs of all dispatcher and kernel activities needed to respectively start and end the execution of an action. Finally, constants c_{invoc_deb} and c_{invoc_end} concern the dispatching and kernel costs induced respectively at the beginning and at the end of a task invocation.

The way these costs are used in a scheduling test is straightforward; every constant identified above is simply added to the worst-case execution times *w* of the application actions. For instance, to take into account the cost induced by the starting and termination of an action into a scheduling test, the c_{action_beg} and c_{action_end} costs have to be added respectively at the beginning and at the end of this action. Similarly, each time a local precedence constraint is encountered, c_{local_dep} is added to the cost of the concerned actions.

4.2 Kernel activities

Contrary to dispatcher activities, the arrival law of kernel activities is not linked to the one of any specific application task. Thus, in addition to the priority and worst-case execution time attributes required to model dispatcher activities, the characterization of this second class of activities require the provision of their arrival law. Due to the tedious analysis that would have required a precise characterization of the activation frequencies of all these activities, we have approximated in HADES their arrival law as a sporadic law. Like the first class of activity, the worst-case exe-

cution time of these activities is determined either analytically or by running worst case scenario benchmarks, and their priority is the highest priority $prio_{sys}$. Therefore, to take into account the cost induced by these activities these new sporadic activities have to be integrated into the initial scheduling test.

In order to characterize kernel activities, a study of the ChorusR3 kernel [CS96] adopted in the HADES prototype, has been undertaken. The source code of every background kernel activity has been examined. In the smallest configuration of the kernel we have used, two background kernel activities exist: clock interrupts, used to update a software clock, and the ATM card interrupt handler, triggered at each message receipt. The worst case execution time c_{clock} and c_{atm} as well as their pseudo-period $\frac{1}{\lambda_{clock}}$ and $\frac{1}{\lambda_{atm}}$ of each interrupt handler has been determined.

5 Example

As an illustration of our methodology, we show how HADES can be used to implement a simple scheduler achieving off-line EDF scheduling analysis.

5.1 Overview of the scheduler

For the sake of clarity, we consider here the scheduling of tasks on a single processor. The task model is the one considered in [Spu96] (sporadic tasks with arbitrary deadlines and resource sharing). A *task* consists of an infinite number of instances, whose arrival times are separated by a minimum time called pseudo-period. More precisely, each task i has:

- a worst-case computation time C_i,
- a deadline D_i, measured relative to the arrival time of the task,
- a worst-case inter-arrival time between two successive task activations (pseudo-period) P_i,

– a maximum time cs_i during which task *i* can use resource *s*, a maximum worst-case computation time c_before_i before task *i* uses resource s, and a maximum worst-case computation time c_after_i for task *i* to complete its execution (we have $C_i = c_before_i + cs_i + c_after_i$).
– a worst-case blocking time B_i that task t_i can experience due to resource sharing.

A sufficient condition (see [Spu96] theorem 7.1) for a set of tasks using EDF preemptive scheduling algorithm, and SRP (Stack Resource Policy) to be feasible, is that each deadline d, in the first busy period of the worst-case task arrival pattern, satisfies the following inequation:

$$\sum_{\{i|D_i \leq d\}} \left(1 + \left\lfloor \frac{d - D_i}{P_i} \right\rfloor\right) C_i + B_{k(d)} \leq d,$$

where B_i is the maximum delay task *i* can wait for a resource when using SRP, and $k(d) = max\{k|D_k \leq d\}$.

5.2 Task execution on top of Hades

The translation of Spuri's task model into the *HEUG* task model is depicted in Figure 3.

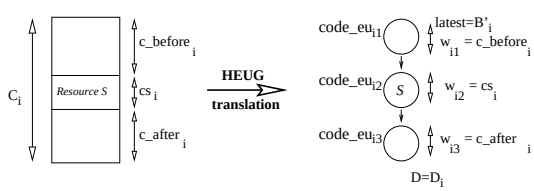

Figure 3: Translation of *Spuri* task model to HEUG task model

Action priorities are dynamically set according to the EDF scheduling policy, i.e., the action with the shorter deadline always has the higher priority. This re-

quires interactions between the HADES dispatcher and the scheduler (see example in Section 3.2.2).

5.3 Modified scheduling test

HADES dispatcher costs are taken into account by replacing C_i by $C_i' = C_i + k(c_{action_beg} + c_{action_end}) + k' c_{local_dep}$, where $k = 3, k' = 2$ when the task accesses a shared resource, and $k = 1, k' = 0$ otherwise, and by replacing B_i by $B_i' = B_i + c_{action_beg} + c_{action_end}$.

Assuming that the scheduler requires at most c_{edf} time units to assign priorities at each new task activation, its global cost during p time units is at most:

$$C_{EDF}(p) = min\left(p, (c_{edf} + c_{action_beg} + c_{action_end}).\sum\left\lfloor\frac{p}{P_i}\right\rfloor\right)$$

In the same manner, the worst-case cost of the kernel mechanisms identified in Section 4 during p time units is at most:

$$C_{Syst}(p) = min\left(p, c_{clock}.\sum\left\lfloor\frac{p}{\lambda_{clock}}\right\rfloor + c_{atm}.\sum\left\lfloor\frac{p}{\lambda_{atm}}\right\rfloor\right)$$

The integration of all these costs leads to the new scheduling test given below:

$$\sum_{\{i|D_i \leq d\}}\left(1 + \left\lfloor\frac{d - D_i}{P_i}\right\rfloor\right) C_i' + B_{k(d)}' \leq d - C_{EDF}(d) - C_{Syst}(d),$$

where the costs of the scheduler and the kernel are withdrawn from the deadline d as they always execute at a higher priority.

Note that a similar work on the integration of execution costs is provided in [BTW95] for the Deadline Monotonic Priority Assignment scheduling test.

6 Related work

We first review existing real-time systems, then examine work in progress, commercial kernels, and finally conclude with standardized middleware architectures. Mars [KDK+89, RSL95] meets the three requirements of real-time, support for fault-tolerance and support for distribution. However, this system is not flexible first, because it is based on a static and periodical scheduling model, and second since it relies on specific hardware-intensive solutions for fault-tolerance and clock synchronization. Maft [KWFT88] 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 [BHV+90] 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 to hard real-time constraints. Real-time systems such as Spring [MRS+90], Maruti [SdSA94], or BASEMENT [HLSL96] propose mechanisms to manage distribution and real-time constraints. However, they do not integrate yet support for safety-critical applications (e.g. fault detection, replication).

Among works in progress, the GUARDS European ESPRIT project [WBDBP96] favors the development of hardware components easing the implementation of fault-tolerance mechanisms.

Numerous commercial real-time operating systems exist. Some products, like EOS [Cas96], QNX [Hil94] and Chorus [Gie95] are based on flexible micro-kernels. These micro-kernels can be used as HADES low-level real-time kernels since they offer a small set of mechanisms suited to our needs (see Section 2.2), and can be configured according to the requirements of our dispatcher.

CORBA [Obj96] 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 [SLM97] is a predictable implementation of CORBA providing facilities to express

real-time requirements. However, our objectives are different from the ones of TAO because TAO focuses on the interconnection of heterogeneous software. Nevertheless, we are convinced that TAO could be easily implemented using the services provided by HADES.

7 Conclusion

In this paper, we have presented HADES, an architecture providing a set of flexible tools mandatory for the execution of a wide range of distributed safety-critical real-time applications. The methodology developed for the design of HADES lies on the separation of services dedicated to a specific application domain (scheduling policy) from a set of services providing a range of robustness properties common to a large spectrum of application domains. This separation is the key of the flexibility exhibited by HADES. So far, HADES has been used to implement priority-based scheduling policies (RM, EDF [LL73]), planning-based scheduling policies (the scheduling policy of Spring [RSS90]) and mechanisms aimed at avoiding multiple priority inversions (PCP [CL90] and SRP [Bak91]). We are currently studying the issues raised by the simultaneous use of different scheduling policies.

The services proposed in HADES have been designed according to a uniform task model (*HEUGs*), which allows to integrate their cost into any scheduling test. As far as cost characterization is concerned, we have studied the predictability of the ChorusR3 kernel, used in the HADES prototype, and we have designed a prototype version of the dispatcher to identify its costs. The work under progress concerns the design of fault-tolerance services.

Given HADES features, i.e., the provision of a range of flexible services deemed at significantly ease and shorten the development of safety-critical real-time applications, a large real-time application from the avionics application domain is planed to be implemented.

References

[Agn91] 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, March 1991.

[Bak91] T. Baker. Stack-based scheduling of realtime processes. *The Journal of Real-Time Systems*, 3(1):67–99, 1991.

[BHV+90] P. Barrett, A. Hilborne, P. Verissimo, L. Rodrigues, P. Bond, D. Seaton, and N. Speirs. The delta-4 extra performance architecture (xpa). In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 481–488, June 1990.

[BTW95] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[Cas96] G. Castelli. Eos : A real-time operating system adapts to application architectures. *IEEE Micro*, pages 41–49, October 1996.

[CL90] M. Chen and J. Lin. Dynamic priority ceilings : A concurrency control protocol for real-time systems. *The Journal of Real-Time Systems*, 2:325–346, 1990.

[CS96] Chorus Systèmes. Chorus/classix r3 technical overview. Technical Report CS/TR-96-119.91, May 1996.

[Gie95] M. Gien. Evolution of the chorus open microkernel architecture : the stream project. In *Proceeding of the 5th International Workshop on Configurable Distributed Systems*, pages 7–13, August 1995.

[GMM97] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerant scheduling on a hard real-time multiprocessor system. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, March 1997.

[Hil94] D. Hildebrand. QNX : Microkernel Technology for Open Systems Handheld Computing. In *Pen and Portable Computing Conference and Exposition*, May 1994.

[HLSL96] H. Hansson, H. Lawson, M. Strömberg, and S. Larsson. Basement: A distributed real-time architecture for vehicle applications. *The Journal of Real-Time Systems*, 11:223–244, 1996.

[HWS95] M. Humphrey, G. Wallace, and J. Stankovic. Kernel-level threads for dynamic, hard real-time environments. In *Real-time Systems Symposium*, August 1995.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Scwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9:25–40, February 1989.

[KS95] Y. Kim and S. Son. *Advances in Real-Time Systems*, chapter Predictability and Consistency in Real-Time Database Systems, pages 509–531. Prentice Hall, 1995.

[KWFT88] R. Kieckhafer, C Walter, A. Finn, and P. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[Lan91] G. Le Lann. Designing real-time dependable distributed systems. Technical Report 1425, INRIA, France, 1991.

[LL73] C.L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LL88] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[Mos94] D. Mossé. Mechanisms for system-level fault tolerance in real-time systems. In *International Conference on Robotics, Vision, and Parallel P rocessing for Industrial Automation*, May 1994.

[MRS+90] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa. Implementing a predictable real-time multiprocessor kernel - the spring kernel. In *"Proceeding of the 7th IEEE Workshop on Real-Time Operating Systems and Software"*, May 1990.

[NMT97] E. Nett, M. Mock, and P. Theisohn. Managing dependencies - a key problem in fault-tolerant distributed algorithms. In *"Proceedings of the 27th International Symposium on Fault-Tolerant Computing"*, pages 2–10, June 1997.

[Obj96] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1996.

[Pol96] S. Poledna. *Fault-Tolerant Real-Time Systems – The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.

[RSL95] 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.

[RSS90] K. Ramamritham, J. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[SAF97] F. Salles, J. Arlat, and J. Fabre. Can we rely on COTS microkernels for building fault-tolerant systems ? In *Workshop on Future Trends of Distributed Computing Systems*, October 1997.

[SdSA94] M. Saksena, J. da Silva, and A. Agrawala. *Principles of Real-Time Systems*, chapter Design and Implementation of Maruti-II, pages 72–102. Sang Son, 1994.

[SLM97] D.C. Schmidt, D.L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications Journal*, 1997.

[Spu96] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, INRIA, France, January 1996.

[WBDBP96] A. Wellings, L. Beus-Dukic, A. Burns, and D. Powell. Genericity and upgradability in ultra-dependable real-time architectures. Technical Report LAAS-R-96417, LAAS, November 1996.

[Xu93] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 19(2):139–154, February 1993.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399