CFSE'1
Rennes, 8 juin - 11 juin 1999
1ère Conférence Française sur les Systèmes d'Exploitation

Un support d'exécution flexible pour applications distribuées temps-réel dur

E. Anceaume	G. Cabillic	P. Chevochot	A. Colin	D. Decotigny	I. Puaut
CNRS/IRISA	INRIA/IRISA	INRIA/IRISA	DGA/IRISA	INRIA/IRISA	INSA/IRISA

{anceaume, cabillic, pchevoch, acolin, ddecotig, puaut}@irisa.fr
IRISA, Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu, 35042 Rennes Cedex, FRANCE.

Résumé

La plupart des systèmes distribués temps-réel tolérants aux fautes développés dans le passé ont été construits pour un domaine d'applications particulier. De telles approches ont abouti à des solutions rigides, non réutilisables dans un domaine d'applications différent, et qui reposent le plus souvent sur des composants matériels spécifiques. Cet article présente un support d'exécution flexible pour applications distribuées temps-réel, qui est construit au dessus de matériel standard. Ce support d'exécution a été conçu en considérant trois aspects fondamentaux et complémentaires : le *temps-réel*, de manière à supporter des applications à contraintes temps-réel dur, la *tolérance aux fautes*, afin que les applications exhibent un niveau de fiabilité élevé en présence de fautes physiques, et la *flexibilité*, de manière à permettre des modifications aisées du support d'exécution pour l'adapter à une large gamme de domaines d'applications, de noyaux temps-réel et de matériel. Cet article se concentre sur les principes de conception qui ont été utilisés pour obtenir un support d'exécution qui soit à la fois temps-réel et flexible.

Abstract

Most distributed dependable real-time systems designed 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 off-the-shelf hardware. Such a support has been designed by considering three fundamental and complementary aspects: *real-time*, to support applications that exhibit hard timing constraints; *fault-tolerance*, to provide a high degree of reliability through transparent fault tolerant software components; and *flexibility*, to allow the modification of parts 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. This paper concentrates on the structuring principles that were exploited to obtain a run-time support that is both predictable and flexible.

Ce travail est financé par la DGA/DSP, #198C4980031303011, et par Dassault Aviation, #197C0690031303012.

1 Introduction

Les applications temps-réel à sûreté critique sont présentes notamment dans les systèmes de contrôle aérien, de production d'énergie nucléaire, de commerce électronique ou encore d'équipements médicaux ou automobiles. La construction de supports d'exécution pour de telles applications est délicate, car un triple objectif doit être atteint. Tout d'abord, ces systèmes doivent être *hautement disponibles*, de manière à assurer un taux de défaillance compris entre 10^{-4} et 10^{-10} défaillances par heure [18, 12] selon le type d'application. Ensuite, ils doivent exhiber des *temps de réponse stricts*, s'échelonnant typiquement de 1 à $100ms$ [15, 18]. Enfin, ils doivent assurer la *cohérence des données* en présence d'exécutions de tâches concurrentes, notamment en environnement distribué.

Atteindre ce triple objectif est complexe et a abouti jusqu'ici à des solutions adaptées uniquement aux contraintes d'un domaine d'application donné (*e.g.* besoins en terme d'ordonnancement des tâches ou de tolérance aux fautes). Nous sommes alors en présence de solutions rigides, souvent basées sur l'utilisation de matériel spécifique, et de ce fait difficilement réutilisables dans un contexte différent [18, 12]. Le projet HADES (Highly Available Distributed Embedded System) [1, 17], développé à l'IRISA, s'attache à la construction d'un environnement pour le développement et l'exécution d'applications distribuées temps-réel à sûreté critique, et plus particulièrement à la construction d'applications du domaine de l'avionique embarquée. L'objectif visé par HADES est de réduire les limitations des systèmes actuels en fournissant un système *flexible* construit au dessus de matériel standard.

Le développement des applications dans HADES repose sur l'utilisation d'un modèle de tâches générique et sur l'utilisation d'outils exécutés hors-ligne. Le modèle de tâches de HADES définit la structure interne des tâches et les contraintes devant être respectées lors de leur exécution (contraintes temporelles, besoins en terme de tolérance aux fautes). Les outils hors-ligne incluent (i) des tests de faisabilité associés à un éventail de politiques d'ordonnancement des tâches, (ii) un outil de dimensionnement de la mémoire nécessaire à l'application, (iii) un outil de réplication automatique des tâches. Ce dernier outil [7] gère la présence de fautes physiques en répliquant les calculs selon les besoins exprimés par le concepteur d'applications (i.e. technique de réplication à utiliser, degré de réplication) de manière transparente à l'application.

L'exécution des applications est gérée par le support d'exécution de HADES [2]. Ce dernier est construit comme une couche logicielle *middleware* intercalée entre le logiciel applicatif et un noyau temps-réel commercial s'exécutant au dessus de matériel standard (à l'heure actuelle, il s'exécute au dessus de Chorus [8] et est émulé sur Solaris). Le support d'exécution est constitué d'un ensemble de services utiles à l'exécution d'une large gamme d'applications distribuées temps-réel. Ces services incluent des politiques d'ordonnancement, un service de communication, un service de gestion de groupe, un service de contrôle des exécutions distribuées, un service de gestion du temps et un service de synchronisation d'horloges. Tous ces services sont à la fois temps-réel et tolérants aux fautes. Le support d'exécution de HADES à été développé en considérant trois aspects complémentaires et fondamentaux :

1. *Temps-réel*: support d'applications exhibant des contraintes temps-réel *dur*, c'est à dire dont le non respect des échéances peut aboutir à des conséquences catastrophiques ;
2. *Tolérance aux fautes*: obtention d'une fiabilité élevée grâce à la fourniture de compo-

sants logiciels pour la tolérance aux fautes physiques, et ce de manière transparente à l'application ;

3. *Flexibilité*, c'est à dire :
   (a) *Extensibilité*: aptitude à modifier des portions du support d'exécution sans avoir à le réécrire complètement ;
   (b) *Adaptabilité*: support d'un large éventail d'applications, de noyaux temps-réel et de matériels.

La prise en compte de l'aspect *temps-réel* repose principalement sur une estimation précise des temps d'exécution au pire cas (WCET, pour *worst-case execution time*) de toutes les activités du système (tâches applicatives, tâches du support d'exécution, noyau temps-réel). L'obtention du WCET des tâches de l'application et du support d'exécution est facilitée par l'utilisation du modèle de tâches de HADES (voir paragraphe 2) ; en particulier, ce modèle de tâches nécessite que toutes les contraintes de synchronisation soient définies avant exécution. Le WCET de la partie centrale du support d'exécution peut être obtenu facilement grâce à sa petite taille qui facilite son analyse. Enfin, les caractéristiques temporelles du noyau temps-réel sont obtenues par analyse de son code source.

L'aspect *tolérance aux fautes* de HADES s'appuie sur l'outil hors-ligne de réplication de tâches (voir [7, 6]) ainsi que sur des mécanismes intégrés au support d'exécution (e.g. détection d'erreurs, services de communication tolérants aux fautes) [7].

Enfin, de manière à être flexible, le support d'exécution de HADES inclut un ensemble de services, qui peuvent évoluer sans avoir à réécrire le support d'exécution (propriété d'*extensibilité*). L'implantation d'un service peut être remplacée par une autre tant qu'elle se conforme à l'interface du service. Ceci nous a permis de développer un éventail de services adaptés aux contraintes spécifiques des applications ou du matériel (propriété d'*adaptabilité*).

Cet article est dédié à une description du support d'exécution de HADES. Plus précisément, il se concentre sur les principes de conception qui ont été utilisés pour obtenir un support d'exécution qui soit à la fois *temps-réel* et *flexible* (voir [7] pour les aspects tolérance aux fautes). Le paragraphe 2 décrit brièvement le modèle de tâches de HADES. Le support d'exécution de HADES est présenté dans le paragraphe 3. Cette description contient une vue d'ensemble de l'interface des services, ce qui illustre la caractéristique d'extensibilité de HADES (paragraphe 3.2). Le paragraphe 3.3 montre pour sa part que l'interface des services est suffisamment riche pour adapter le support d'exécution à une large gamme d'applications (propriété d'*adaptabilité*). Enfin, le support d'exécution de HADES est comparé à des travaux apparentés dans le paragraphe 4.

2 Le modèle de tâches de HADES

Le modèle de tâches de HADES (voir [1] pour plus de détails) décrit chaque tâche comme un graphe orienté acyclique (voir paragraphe 2.1). Un ensemble d'attributs, relatifs aux contraintes temporelles, de synchronisation de distribution ou de tolérance aux fautes des tâches, peuvent être spécifiés (voir paragraphe 2.2). Dans la suite, le terme *tâche* couvre à la fois les tâches de l'application, notées *App_tasks*, et les tâches du support d'exécution, notées *RTS_tasks*.

2.1 Structure des tâches

Chaque tâche est décrite par un graphe orienté acyclique, appelé *HEUG*, pour HADES *Elementary Unit Graph*. Un nœud de ce graphe (également appelé *EU* pour *Elementary Unit*) est soit un nœud contenant du code (*Code_EU*, pour *Code Elementary Unit*), soit un appel au support d'exécution (*Sys_EU*, permettant notamment l'invocation synchrone ou asynchrone de tâches). Un arc du graphe est une contrainte de précédence entre *EUs*. Un arc entre les *EUs* eu_i et eu_j, noté $eu_i \rightarrow eu_j$, signifie que eu_j peut commencer à s'exécuter uniquement lorsque eu_i a terminé.

2.2 Attributs des tâches

Les attributs associés aux tâches peuvent être classés en quatre catégories : les attributs de synchronisation, les attributs temporels, les attributs de distribution et les attributs de tolérance aux fautes. Les attributs de tolérance aux fautes sont principalement utilisés par l'outil hors-ligne de réplication de tâches pour lui indiquer la stratégie de tolérance aux fautes à adopter (voir [7]).

Attributs de synchronisation :

Les *Code_EUs* peuvent se synchroniser entre eux par l'intermédiaire de deux entités : les *variables de condition* (variables booléennes pouvant être positionnées à vrai, faux, et étant utilisées par les *Code_EUs* pour attendre qu'une condition soit vraie), et le *partage de ressources*. Une *ressource* abstrait tout composant matériel ou logiciel utilisé par un *Code_EU*. Tout *Code_EU* ou groupe de *Code_EUs* indique les ressources qu'il doit acquérir avant de débuter son exécution, ainsi que le mode d'accès requis (*partagé* ou *exclusif*).

Attributs temporels :

Des attributs temporels peuvent être associés aux tâches et aux *EUs*. Ils peuvent être utilisés hors-ligne, par exemple par les tests de faisabilité et/ou en-ligne pour l'instrumentation des exécutions ou les politiques d'ordonnancement dynamiques. Les attributs temporels associés aux tâches permettent de spécifier la loi d'arrivée de chaque tâche (périodique, sporadique ou apériodique) ainsi que leur échéance de terminaison. Le concepteur d'une tâche peut également spécifier quel *service d'ordonnancement* (voir paragraphe 3) doit être utilisé pour déterminer l'ordre d'exécution des *EUs* des tâches. Au niveau de chaque *EU*, le concepteur de l'application peut indiquer son échéance de terminaison, sa date de démarrage au plus tôt et au plus tard, ainsi que la priorité qui lui sera affectée à l'exécution.

Attributs de distribution :

Les tâches HADES sont distribuées. Une tâche peut s'exécuter sur un ensemble de *sites*.

Un attribut indique pour chaque *EU* le site sur lequel cet *EU* doit s'exécuter. Une contrainte de précédence $eu_i \rightarrow eu_j$ est dite locale (resp. distante) si eu_i et eu_j sont assignées au même site (resp. à des sites différents). Un autre attribut de distribution permet d'indiquer quel service de gestion de la distribution est chargé de l'exécution des *App_tasks* (voir paragraphe 3).

3 Le support d'exécution de HADES

L'objectif principal recherché lors de la conception du support d'exécution de HADES a été d'obtenir un support d'exécution qui soit à la fois flexible (i.e. extensible et adaptable) et adapté aux applications temps-réel. Ce paragraphe donne une vue d'ensemble de la méthode suivie pour atteindre cet objectif.

3.1 Vue d'ensemble du support d'exécution

Le support d'exécution de HADES s'exécute sur chaque site. Il est constitué d'une couche logicielle *middleware* intercalée entre le logiciel applicatif et un noyau temps-réel standard. Son rôle est de fournir un ensemble flexible de fonctionnalités pour l'exécution d'applications distribuées temps-réel. Le support d'exécution de HADES est divisé en trois couches, nommées respectivement *couche d'adaptation au noyau*, *allocateur*, et *couche de services*. Une liste non exhaustive des fonctionnalités fournies par le support d'exécution est donnée dans la figure 1.

Couche d'adaptation au noyau :

Cette couche a été conçue pour faciliter le portage du support d'exécution de HADES sur différents noyaux temps-réel. Cette couche utilise les appels système du noyau sous-jacent en ce qui concerne l'exécution des processus et la gestion des interruptions.

Allocateur :

L'allocateur est le cœur du support d'exécution de HADES. Il fournit les fonctionnalités de base nécessaires à l'exécution des tâches : gestion de l'*éligibilité* des tâches et *instrumentation* de leur exécution. En ce qui concerne la gestion de l'éligibilité des tâches, l'allocateur gère un ensemble de files d'exécution d'*EUs*, chaque file étant associée à une priorité donnée. Les *App_ tasks* sont insérées dans une des files d'exécution gérée par le *service d'ordonnancement* (voir paragraphe 3.2). Pour leur part, les *RTS_ tasks* sont insérées dans les files d'exécution selon un ordre précalculé. Une *EU*, notée eu_i, est dite *éligible* si elle vérifie les quatre conditions suivantes : *(i)* les *EUs* dont eu_i doit attendre la terminaison pour vérifier ses contraintes de précédence ont terminé leur exécution, *(ii)* toutes les ressources demandées par eu_i sont disponibles, *(iii)* toutes les variables de condition attendues par eu_i sont positionnées, *(iv)* l'heure courante est supérieure à la date de démarrage au plus tôt d'eu_i. L'allocateur, parmis l'ensemble des *EUs* éligibles exécute la plus prioritaire d'entre elles. La fonction d'instrumentation de l'allocateur consiste à vérifier la correction de l'exécution des tâches (e.g. dépassement d'échéance, violation de loi d'arrivée).

Couche de services :

Cette couche comprend un éventail de services exhibant les propriétés de ponctualité et de tolérance aux fautes requises par le domaine d'application visé. Un *service* est défini comme un module accomplissant une fonction donnée, et il est invoqué par une *interface* implantée sous la forme d'un ensemble de tâches. Le support d'exécution de HADES fournit des services d'ordonnancement, de gestion des exécutions distribuées, d'envoi de messages multipoint (multicast) [9], de gestion de groupes [9], de gestion du temps et de synchronisation des horloges (voir figure 1). Contrairement à l'allocateur, l'ensemble des services de la couche de services est *extensible*, i.e. l'implantation d'un service peut être changée sous réserve que cette dernière se conforme à l'interface d'appel du service (voir paragraphe 3.2).

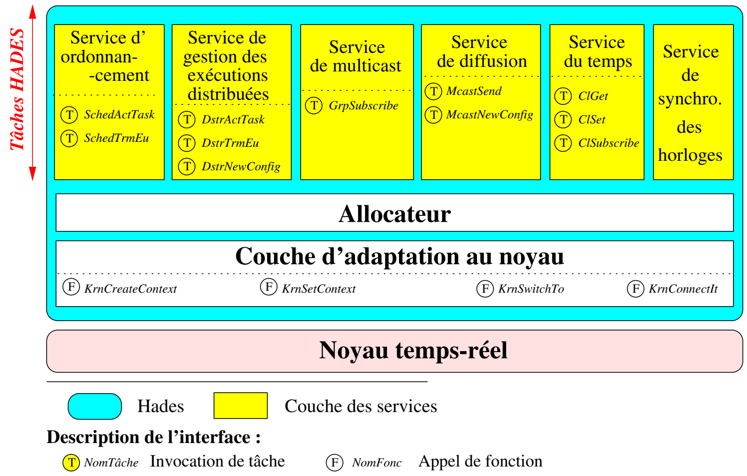

FIG. 1 – *Structure interne du support d'exécution de* HADES

3.2 Propriété d'extensibilité

Nous définissons par *extensibilité* la propriété d'un système lui permettant de faire évoluer ses fonctionnalités à tout moment dans son cycle de vie. Cette propriété est assurée par le support d'exécution de HADES grâce à un découplage entre la couche de services et l'allocateur. Cette séparation permet l'extension des propriétés de tout service sans avoir à se soucier des contraintes de ponctualité, de tolérance aux fautes ou de synchronisation des autres services. Les paragraphes suivants présentent l'interface de deux services : le service d'ordonnancement et le service de gestion des exécutions distribuées.

Service d'ordonnancement :

Le rôle d'un service d'ordonnancement est de calculer l'ordre dans lequel les *EUs* doivent s'exécuter. Les tests de faisabilité, qui analysent si un ensemble de tâches respectent leurs échéances, sont fournis sous la forme d'outils exécutés hors-ligne. Un attribut temporel (voir paragraphe 2) permet d'indiquer le service d'ordonnancement utilisé par chaque *App_task*.

Le service d'ordonnancement est accessible à travers l'invocation de deux *RTS_tasks* appelées respectivement *SchedActTask* et *SchedTrmEu*. La tâche *SchedActTask* est invoquée de manière asynchrone lors de l'activation de chaque *App_task* T_i. Deux paramètres à l'activation identifient la tâche à exécuter (T_i) et les *EUs* de T_i à exécuter sur le site courant. La tâche *SchedTrmEu* est invoquée de manière asynchrone lors de la terminaison de chaque *EU* eu_i, avec eu_i paramètre de l'invocation. Les tâches *SchedActTask* et *SchedTrmEu* font appel à un ensemble de *Sys_EUs* fournis par l'allocateur pour mettre à jour ses files d'exécution (e.g. insertion lors de l'activation d'une tâche, destruction lors de la terminaison d'une *EU*).

L'interface du service d'ordonnancement a été conçue pour implanter un large spectre de politiques d'ordonnancement adaptées à des domaines d'applications différents (voir paragraphe 3.3.1).

Service de gestion des exécutions distribuées :

Le service de gestion des exécutions distribuées est chargé de coordonner l'exécution des tâches à exécuter sur plusieurs sites, de valider leurs contraintes de précédence et de détecter leur terminaison. Un attribut de distribution indique pour chaque tâche T_i quel service de gestion des exécutions distribuées coordonne son exécution (voir paragraphe 2).

L'interface d'un service de gestion des exécutions distribuées est composée de trois *RTS_tasks*, nommées *DstrActTask*, *DstrTrmEu* et *DstrNewConfig*. La tâche *DstrActTask* est invoquée de manière asynchrone lors de l'activation d'une *App_task*. La tâche *DstrTrmEu* est invoquée de manière asynchrone lors de la terminaison d'*EUs*. La tâche *DstrNewConfig* est invoquée lors de la modification de l'ensemble des sites opérationnels (i.e. après une défaillance ou une réinsertion de site). Les paramètres de cette tâche indiquent l'ensemble des sites opérationnels et dans le cas de réinsertion de sites leur date de réinsertion.

Les tâches du service de gestion des exécutions distribuées interagissent avec l'allocateur via un ensemble de *Sys_EUs* (e.g. allocation de flots d'exécution, validation d'un ensemble de contraintes de précédence, terminaison d'une invocation). Ces tâches invoquent également les tâches du service de multicast pour lancer l'exécution des tâches.

L'interface du service de gestion des exécutions distribuées a été conçue pour s'adapter à des modèles de fautes différents (voir [7] pour plus de détails).

3.3 Propriété d'adaptabilité

Les propriétés vérifiées par les implantations des services HADES peuvent être adaptées selon le domaine d'application visé (e.g. service d'ordonnancement), le noyau de système temps-réel utilisé (e.g. couche d'adaptation au noyau), ou encore le modèle de fautes considéré (e.g. service de multicast ou de gestion des exécutions distribuées), et ceci dans la mesure où ces implantations respectent l'interface de ces services (voir paragraphe 3.2). Ce paragraphe illustre cette propriété d'adaptabilité en décrivant pour deux composants du support d'exécution (le service d'ordonnancement et la couche d'adaptation au noyau) deux implantations de ces composants exhibant des propriétés différentes.

3.3.1 Service d'ordonnancement

Le choix d'un service d'ordonnancement est fortement dépendant des caractéristiques de chaque application, à savoir la loi d'arrivée des tâches ou encore le mode de synchronisation entre tâches. Un ensemble de services d'ordonnancement a été implanté dans le support d'exécution de HADES. Nous en présentons deux ci-dessous, qui implantent respectivement une politique d'ordonnancement statique et dynamique.

Politique d'ordonnancement statique :

Nous avons développé dans HADES une politique d'ordonnancement statique qui est une extension au cadre distribué de l'algorithme d'ordonnancement présenté par Xu et Parnas dans [20]. Ce dernier opère sur un ensemble de tâches périodiques avec contraintes de précédence et partage de ressources. Il repose sur un algorithme exécuté hors-ligne (i.e. en dehors du support d'exécution) qui génère un plan d'exécution en indiquant la date de démarrage de chaque portion de tâche (*EU* selon la terminologie HADES). Le plan généré est *préemptif* (une *EU* peut en interrompre une autre).

Un ensemble de files d'exécution est réservé aux *EUs* gérées par ce service d'ordonnancement. L'association d'une priorité à une *EU* dépend de son *niveau de préemption* : si l'*EU* eu_i interrompt l'*EU* eu_j, on lui attribue une priorité supérieure à celle d'eu_j. Le service d'ordonnancement est implanté par deux *RTS_tasks* nommées $t^{Act}_{XuParnas}$ et $t^{Trm}_{XuParnas}$. Quand une *App_task* est activée (i.e. au début de sa période), $t^{Act}_{XuParnas}$ insère ses *EUs* dans les files d'exécution de l'allocateur. Cette tâche utilise pour cela les informations contenues dans le plan d'exécution calculé hors ligne : une *EU* est insérée dans la file d'exécution qui correspond à son niveau de préemption, et dans une file d'exécution donnée les *EUs* sont classées par date de démarrage au plus tôt croissantes. Lors de la terminaison d'une *EU*, la tâche $t^{Trm}_{XuParnas}$ la supprime de la file d'exécution de l'allocateur.

Politique d'ordonnancement dynamique :

Nous avons développé une politique d'ordonnancement dynamique de type EDF (*Earliest Deadline First*) couplé avec la politique SRP (*Stack Resource Policy* [3]) utilisée pour éliminer les inversions de priorité multiples. Le principe de base de SRP est que lorsqu'une tâche demande une ressource, elle est bloquée à son démarrage plutôt qu'ultérieurement lorsqu'elle demande effectivement la ressource, limitant ainsi le nombre de changements de contexte entre tâches.

L'implantation de la politique d'ordonnancement EDF (hors SRP) est triviale. Une file d'exécution unique, dans laquelle les *EUs* sont classés par échéances croissantes, est attribuée aux *App_tasks* ordonnancées selon EDF. Le service d'ordonnancement EDF est implanté par deux *RTS_tasks* nommées t^{Act}_{edf} et t^{Trm}_{edf}. A chaque activation de tâche, t^{Act}_{edf} ajoute les *EUs* de cette tâche dans la file d'exécution EDF selon la valeur croissante de leurs échéances. Lors de la terminaison d'une *EU*, la tâche t^{Trm}_{edf} la supprime de la file d'exécution EDF.

Cette implantation subit quelques modifications lorsque EDF est couplée avec SRP. Le service d'ordonnancement EDF/SRP est implanté par deux *RTS_tasks* nommées $t^{Act}_{edf/srp}$ et $t^{Trm}_{edf/srp}$. A chaque activation d'une tâche t, $t^{Act}_{edf/srp}$ teste si t peut interrompre l'exécution de la tâche en cours d'exécution, notée t_{cur} (voir [3] pour la formule établissant la condition de préemption). Si la condition de préemption est vérifiée, $t^{Act}_{edf/srp}$ insère les *EUs* de t dans l'unique file d'exécution associée à EDF/SRP. Dans le cas contraire, t est placée dans une structure de données contenant les tâches en attente d'exécution. A chaque terminaison de *EU*, la tâche $t^{Trm}_{edf/srp}$ supprime l'*EU* concernée de la file d'exécution EDF/SRP puis examine la liste des exécutions en attente. Si la condition de préemption est vérifiée, la file d'exécution EDF/SRP est modifiée en conséquence.

3.3.2 Couche d'adaptation au noyau

La couche d'adaptation au noyau interagit avec le noyau temps-réel sous-jacent en ce qui concerne la gestion de processus légers (*threads*) et la gestion des interruptions. Pour porter le support d'exécution de HADES au dessus d'un noyau temps-réel, les quatre fonctions suivantes doivent être écrites : *KrnCreateContext* (allocation du contexte d'un processus léger), *KrnSetContext* (initialisation du contexte d'un processus léger sans le lancer), *KrnSwitchTo* (démarrage d'un processus léger, ou redémarrage après une préemption), et *KrnConnectIt* (connexion d'un gestionnaire d'interruptions sur un niveau d'interruption donné).

Par ailleurs, le noyau temps-réel doit permettre de garder les structures de données de l'allocateur cohérentes malgré la présence d'interruptions, et ce sans introduire de blocage au sein de l'allocateur. A ce jour, le support d'exécution de HADES a été implanté au-dessus de Chorus 3.1, et émulé au dessus de Solaris.

Adaptation au noyau Chorus :

Chorus 3.1 [8] fournit un système d'exploitation temps-réel modulaire pouvant être configuré pour s'adapter aux applications embarquées de petite taille aussi bien qu'aux plates-formes distribuées conformes à la norme POSIX. Chorus 3.1 est bâti autour d'un module nommé *core executive*, qui fournit l'ensemble des fonctionnalités indispensables à un exécutif temps-réel. L'environnement HADES a été porté au dessus du *core executive* nommé *micro core* qui offre la notion de processus léger (*thread*), un ordonnancement à priorité fixe, des outils de synchronisation de base, ainsi que la possibilité de connecter des gestionnaires d'interruptions, de trappes et d'exceptions.

Les quatre fonctions répertoriées plus haut ont été implantées de manière triviale en utilisant l'interface de l'exécutif *micro core*. La partie la plus délicate du portage a consisté à gérer la cohérence des structures de données de l'allocateur. Notre solution repose sur la présence dans Chorus 3.1 d'un ordonnanceur à priorité fixe. Elle consiste à attribuer les priorités Chorus de la manière suivante : *(i)* les threads Chorus allouées aux *Sys_EUs* se voient attribuer une priorité Chorus élevée ; *(ii)* les threads Chorus allouées aux gestionnaires d'interruptions, qui sont débloquées quand une interruption est demandée, se voient attribuer une priorité Chorus intermédiaire ; *(iii)* les threads Chorus allouées aux *Code_EU* sont exécutées avec une priorité Chorus faible. Avec une telle attribution des priorités, une interruption peut être demandée pendant que l'allocateur exécute un *Sys_EU*, cependant l'exécution du gestionnaire d'interruption associé est retardée jusqu'à la fin de l'exécution du *Sys_EU*. Cette solution peut être utilisée lors du portage de HADES au-dessus de tout noyau temps-réel possédant un ordonnanceur à priorité fixe.

Adaptation au noyau Solaris :

Le support d'exécution de HADES a été émulé au dessus du système d'exploitation Solaris. L'objectif de ce portage a été de tester les propriétés fonctionnelles des applications, ainsi que leur propriétés temporelles en simulant un scénario temporel d'exécution au pire cas. Les quatre fonctions de la couche d'adaptation au noyau ont été implantées au dessus de la bibliothèque de processus légers de Solaris. Plus précisément, un processus (composé de plusieurs processus légers) est alloué sur chaque site, et exécute à la fois les *App_tasks* et les *RTS_tasks*. Par ailleurs, un processus unique pour tout le système est chargé *(i)* de coordonner tous les sites, et *(ii)* de simuler une architecture distribuée (i.e. simuler la progression du temps-réel, gérer la dérive des horloges locales et le temps de transmission des messages).

3.4 Propriété de prévisibilité

Une propriété centrale du support d'exécution de HADES est sa *prévisibilité*. Une action est dite *prévisible* si ses résultats et son temps d'exécution au pire cas sont connus avant son exécution. Cette propriété de *prévisibilité* est cruciale pour l'implantation de tests de faisabilité associés aux applications temps-réel dur. Fournir un support d'exécution prévisible nécessite que tous ses composants soient prévisibles. Les paragraphes qui suivent montrent la démarche suivie pour assurer la prévisibilité de tous les composants du support d'exécution de HADES (i.e. couche de services, allocateur, noyau temps-réel).

Prévisibilité de la couche de services :

La prévisibilité de la couche de services résulte de deux aspects. D'une part, tous les services sont spécifiés à l'aide du modèle de tâches de HADES. L'utilisation de ce modèle de tâches facilite l'analyse du comportement temporel des tâches grâce à la séparation entre code séquentiel (i.e. *EUs*) et synchronisations. D'autre part, hormis un petit sous-ensemble de tâches de services, qui sont périodiques (tâche du service de multicast, tâches de gestion du temps), toutes les tâches de service suivent la loi d'arrivée de l'*App_task* qui les invoque. Ceci permet de reporter le WCET de ces tâches au WCET des tâches applicatives. Les tâches périodiques sont directement intégrées au test de faisabilité de l'application.

Prévisibilité de l'allocateur :

Les fonctionnalités de l'allocateur étant très réduites, son code source est peu volumineux et de ce fait, est facile à analyser. De plus, il possède un ensemble de caractéristiques le rendant prévisible : *(i)* il ne contient aucune activité asynchrone et aucune tâche indépendante ; *(ii)* il n'a pas recours à l'allocation dynamique de mémoire ; et *(iii)* il ne contient aucune synchronisation, et de ce fait ne se bloque jamais. Ces caractéristiques nous ont permis de caractériser l'allocateur par trois WCETs : $wcet_{term/start}$, pour terminer l'exécution d'un *EU* et démarrer l'exécution de son successeur ; *(ii)* $wcet_{sys}$, pour lancer l'exécution d'un *Sys_EU* ; *(iii)* $wcet_{int}$, pour prendre en compte une interruption et lancer le traitement associé. L'intégration de ces trois WCETs dans le test de faisabilité d'une application est relativement simple. Les termes $wcet_{term/start}$ et $wcet_{sys}$ sont simplement ajoutés au WCET de l'application. Le terme $wcet_{int}$ est intégré sous la forme d'une tâche sporadique de haute priorité.

Prévisibilité du noyau temps-réel :

Puisque les constructeurs de noyaux temps-réel ne fournissent généralement pas de garantie de prévisibilité de leurs noyaux, il est nécessaire pour ce faire d'avoir accès à leur code source. Nous avons par conséquent entrepris une étude du noyau Chorus 3.1, qui est une des deux plates-formes cibles de HADES. Le code source de ce noyau a été analysé (vérification d'invariants, analyse des effets de bord des fonctions) [16]. A ce jour, l'analyse du code est effectuée manuellement.

4 Travaux apparentés

Un certain nombre de systèmes supportant l'exécution d'applications distribuées temps-réel à sûreté critique ont été conçus à ce jour. Mars [13, 18] et Maft [12] sont deux d'entre eux. Mars n'est pas flexible : d'une part, il supporte une unique politique d'ordonnancement statique et limitée aux tâches périodiques, et d'autre part, il repose sur l'utilisation de matériel dédié pour la tolérance aux fautes et la synchronisation des horloges. Au contraire, le support d'exécution de HADES peut s'adapter à plusieurs politiques d'ordonnancement (voir paragraphe 3.3), et utilise du logiciel et du matériel standard. Maft propose des mécanismes pour tolérer les défaillances Byzantines de manière à obtenir un très haut degré de fiabilité, et ce sans sacrifier les performances des applications. Cependant, pour les mêmes raisons que Mars, Maft n'est pas flexible. Delta-4/XPA [4] propose un ensemble flexible de mécanismes de tolérance aux fautes et de protocoles de communication. En revanche, l'environnement d'exécution utilisé pour son implantation n'est pas prévisible et les mécanismes qu'il fournit ne sont pas adaptés aux contraintes temps-réel dur. Parmi les travaux en cours, le projet ESPRIT Européen GUARDS [19] vise des objectifs proches de ceux de HADES. Cependant, contrairement à HADES, GUARDS repose sur l'utilisation de composants matériels spécialisés simplifiant la construction de mécanismes de tolérance aux fautes.

Le support d'exécution de HADES n'est pas directement comparable aux noyaux temps-réel tels que QNX [10] ou Chorus [8]. Bien que ces derniers exhibent pour la plupart un comportement prévisible, ils ne sont généralement pas tolérants aux fautes et ne sont pas tous conçus pour s'adapter aux besoins des applications (e.g. support de plusieurs politiques d'ordonnancement). Ils peuvent néanmoins être utilisés comme noyau temps-réel de base pour le portage du support d'exécution de HADES car la plupart offre les fonctionnalités nécessaires au portage de la couche d'adaptation au noyau de HADES.

Les principes de structuration qui ont été adoptés lors de la construction de HADES pour assurer sa *flexibilité* sont très similaires à ceux proposés par les concepteurs de la première génération de micro noyaux [14]. La différence fondamentale entre les deux provient de la classe de services pouvant être spécialisés. Dans les micro-noyaux, ces services sont des services généraux des systèmes d'exploitation (e.g. spécialisation de la gestion de la mémoire au travers de la construction de serveurs de pagination, spécialisation des services de communication). Dans HADES, les services pouvant être spécialisés sont adaptés aux applications temps-réel à sûreté critique (e.g. service d'ordonnancement temps-réel, service de synchronisation des horloges). Des travaux plus récents ont donné naissance à des noyaux de système hautement spécialisables, tels que Spin [5] ou Exokernel [11], mais ces noyaux ne s'attachent pas au problème de prévisibilité.

5 Conclusion

Nous avons présenté dans cet article un support d'exécution flexible pour applications distribuées temps-réel dur, en insistant sur les principes de conception ayant permis d'assurer de manière conjointe les propriétés de temps-réel et de flexibilité. Nos travaux actuels dans HADES s'articulent autour de trois axes : (i) la validation de l'environnement HADES via le portage d'une application du domaine de l'avionique, développée par Dassault-Aviation ; (ii) l'automatisation de l'analyse temporelle du support d'exécution de HADES via la conception d'un outil d'analyse de temps d'exécution au pire cas de programmes, opérant par analyse de leur code source ; (iii) la prise en compte de conditions opérationnelles non totalement prévues dans les mécanismes de tolérance aux fautes et d'ordonnancement.

Bibliographie

1. E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. Hades: A middleware support for distributed safety-critical real-time applications. In *Proc. of the 18th International Symposium on Distributed Computing Systems (ICDCS'18)*, pages 344–351, mai 1998.
2. E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. A flexible run-time support for distributed dependable hard real-time applications. In *Proc. of the 2nd IEEE Interna-*

tional Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99), mai 1999. Also INRIA Research Report No. 3564, Nov 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, juin 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 and I. Puaut. La réplication hors-ligne comme approche flexible pour la tolérance aux fautes des applications temps-réel dur. In *Actes de la première Conférence Française en Systèmes d'Exploitation (CFSE'1)*, Juin 1999.
7. P. Chevochot, I. Puaut, and G. Cabillic. A flexible environment for automatic replication and execution of distributed safety-critical real-time applications. RR 1183, IRISA, avril 1998.
8. Chorus Systèmes. Chorus/classix R3 technical overview. RR 96-119.91, mai 1996.
9. D. Decotigny. Un protocole de gestion de groupe et un protocole de diffusion atomique sous contraintes de temps-réel, juil. 1998. Stage de DEA, Université de Rennes I.
10. D. Hildebrand. An architectural overview of QNX. In *Proc. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, avril 1992.
11. 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.
12. R. Kieckhafer, C. Walter, A. Finn, and P. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Trans. on Computers*, 37(4):398–405, avril 1988.
13. H. Kopetz et al. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9:25–40, févr. 1989.
14. J. Liedtke. Toward real microkernels. *Comm. of the ACM*, 39(9):70–77, sept. 1996.
15. 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*, mai 1990.
16. Projet Solidor. Choix initiaux de l'architecture matérielle et logicielle retenue, délivrable 1.2 du contrat de recherche numéro 96.106/DSP/STTC. INRIA, nov. 1997.
17. Projet SOLIDOR. Hades: A middleware support for distributed safety-critical real-time applications. WEB Document, 1998. http://www.irisa.fr/solidor/work/hades.html.
18. 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.
19. A. Wellings, L. Beus-Dukic, A. Burns, and D. Powell. Genericity and upgradability in ultra-dependable real-time architectures. RR LAAS-R-96417, LAAS, nov. 1996.
20. 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, mars 1990.