Conception de systèmes distribués temps-réel strict tolérants aux fautes avec du matériel « sur étagère »*

Pascal Chevochot — INRIA / IRISA
Isabelle Puaut — INSA / IRISA
Mél : {pchevoch,puaut}@irisa.fr
Tél : 02 99 84 { 25 02, 73 10}
Fax : 02 99 84 25 29
Campus de Beaulieu, 35042 Rennes cedex, France

Mots clés : Tolérance aux fautes, temps-réel strict, matériel « sur étagère », analyse d'ordonnancement, détection d'erreurs, systèmes distribués

Résumé : Depuis plusieurs décennies, différents systèmes embarqués temps-réel sûrs de fonctionnement ont été conçus, chacun répondant aux exigences d'un domaine d'application particulier. Cette spécialisation a souvent mené à des solutions non flexibles et fortement basées sur du matériel dédié, qui sont coûteuses à concevoir et maintenir. Dans cet article nous présentons une synthèse de nos travaux pour exécuter des applications distribuées temps-réel strict en tolérant les fautes, et ce en utilisant uniquement du matériel « sur étagère » (c'est à dire à large diffusion et non conçu pour un domaine d'application particulier). Nos travaux sont présentés sous trois aspects connexes, à savoir les mécanismes de tolérance aux fautes, l'ordonnancement des applications tolérantes aux fautes,

*Ce travail est partiellement supporté par la DGA (#98.34.375.00.470.75.65.) et Dassault-Aviation (#98.C.304.00.31303.01.2). Cet article est paru dans les actes de la 9th International Conference on Real-Time Systems (RTS'01), Paris, mars 2001, pages 209-226.

et l’utilisation de matériel sur étagère.

1 Introduction

Depuis plusieurs décennies, différents systèmes embarqués temps-réel sûrs de fonctionnement ont été conçus, chacun répondant aux exigences d'un domaine applicatif particulier (ferroviaire, spatial, avionique, automobile). Cette spécialisation a souvent mené à des solutions non flexibles et fortement basées sur du matériel dédié, qui certes sont ultra-fiables et rapides, mais qui sont également coûteuses à concevoir et maintenir. C'est pourquoi, pour réduire les coûts, de plus en plus de systèmes utilisent partiellement [14, 17] ou complètement [1] du matériel *sur étagère*, c'est-à-dire du matériel à large diffusion non conçu pour un domaine d'applications particulier.

Notre travail s'inscrit dans cette tendance. Son objectif est de concevoir des mécanismes pour analyser la faisabilité et exécuter des applications distribuées *temps-réel strict* (*i.e.* des applications dont les échéances ne doivent jamais être dépassées) en tolérant les fautes physiques temporaires et permanentes (les fautes de conception sont ici exclues). Ces mécanismes utilisent uniquement du matériel (cartes mères et réseau) sur étagère.

Notre solution repose sur un ensemble de services de base qui sont construits sous forme d'un *intergiciel* (*middleware layer*) au-dessus d'un micro-noyau temps-réel, et sur un outil chargé d'intégrer aux applications distribuées la redondance nécessaire pour éliminer les erreurs. L'ensemble est conçu de manière à ce qu'il soit possible de prouver, avant exécution, que toutes les échéances des applications seront respectées, en tenant compte de tous les coûts de la tolérance aux fautes, y compris dans le pire cas de charge et de fautes. Notre solution a été entièrement implantée dans le prototype HADES (Highly Available Distributed Embedded System), qui est une infrastructure logicielle pour développer et exécuter des applications distribuées temps-réel strict sûres de fonctionnement (voir la figure 1). HADES utilise comme matériel des PCs Pentium reliés par un réseau ATM, et comme système d'exploitation, le micro-noyau CHORUS de SUN.

Dans cet article, nous présentons une synthèse de nos travaux sous trois aspects connexes, à savoir les mécanismes de tolérance aux fautes (§ 2), l'ordonnancement des applications

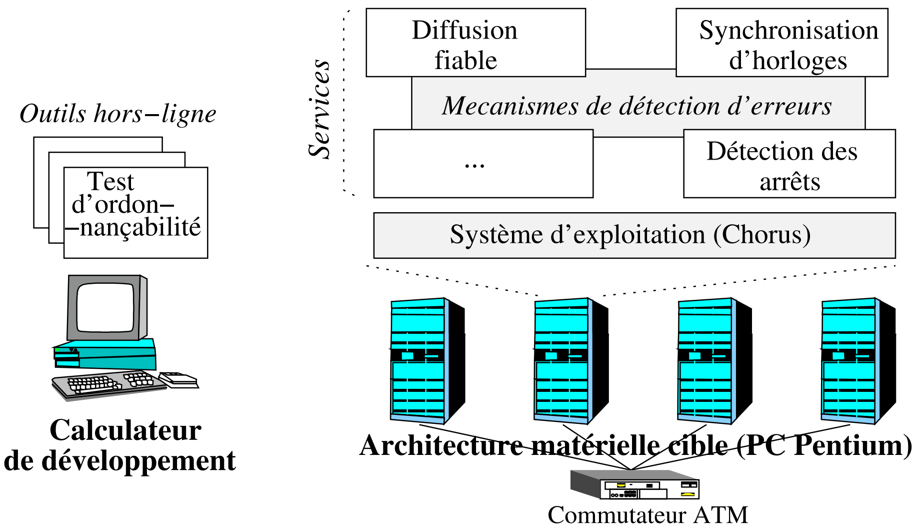


FIG. 1 – *Architecture générale du prototype* HADES

tolérantes aux fautes (§ 3), et l'utilisation de matériel sur étagère (§ 4). De plus amples détails sur nos travaux peuvent être trouvés dans les documents [6, 7, 9, 10, 11].

2 Tolérance aux fautes

La tolérance aux fautes peut être mise en œuvre par différents moyens [3]. Nous distinguons la *détection d'erreurs* qui consiste à identifier un état erroné comme tel, le *recouvrement d'erreurs* qui consiste à substituer une erreur par un état exempt d'erreur, et la *reconfiguration* qui consiste, dans un système qui a défailli, à délivrer un résultat acceptable avec les composants non défaillants.

Pour tolérer les fautes dans les systèmes distribués temps-réel strict, nous proposons d'utiliser un ensemble de services qui s'exécutent au-dessus d'un noyau temps-réel, et un outil hors-ligne (utilisé avant l'exécution du système). Les services sont chargés de fournir des propriétés sur la synchronisation des horloges, sur la détection des calculateurs arrêtés ou non, et sur les communications afin de faciliter la conception d'applications tolérantes aux fautes. Ces applications sont obtenues grâce à l'outil hors-ligne, appelé *outil de réplication*, qui transforme automatiquement une application non-tolérante aux fautes, afin de dupliquer ses traitements et ainsi de pouvoir tolérer les fautes.

Dans la suite, nous présentons le modèle de fautes auquel doit se conformer le matériel pour garantir le bon fonctionnement des services et des applications répliquées (§ 2.1), puis les services (§ 2.2), et enfin la construction d'applications tolérantes aux fautes (§ 2.3).

2.1 Modèle de fautes

Le bon fonctionnement des services et des applications est assuré en présence de fautes temporaires ou permanentes dues à des phénomènes physiques. D'autre part, nous supposons que les hypothèses de défaillances suivantes sont vérifiées par le matériel :

– *Calculateurs à silence sur défaillance*, c'est-à-dire qu'un calculateur produit des résultats corrects dans le domaine des valeurs et temporel, ou s'arrête définitivement de produire des résultats ;
– *Horloges fiables*, c'est-à-dire dont la dérive est bornée ;
– *Réseau à omissions bornées*, c'est-à-dire dont le nombre d'omissions successives (dans une fenêtre bornée) pour un canal (logique) reliant deux calculateurs est borné (par la constante ω).

Ce modèle de fautes a été choisi car il simplifie la conception de services et d'applications tolérantes aux fautes. Il n'impose pas l'utilisation d'un mécanisme particulier pour recouvrir les erreurs. De plus, il n'impose pas l'utilisation de matériel spécifique, ce qui serait nécessaire si des fautes arbitraires (Byzantines) étaient supposées pour les calculateurs et/ou le réseau. Enfin, le niveau de réplication nécessaire pour supporter f fautes lorsque les calculateurs sont à silence sur défaillance est plus faible ($f+1$) que celui nécessaire lorsque des fautes arbitraires sont supposées ($3f+1$), ce qui est important pour les systèmes embarqués dont les ressources sont limitées.

Toutefois, l'hypothèse de silence sur défaillance pour un calculateur sur étagère n'est pas réaliste (par exemple les auteurs de [16] évaluent la couverture du silence sur défaillance dans les systèmes sans mécanisme logiciel de détection d'erreurs et sans redondance matérielle à 54.4 % pour un processeur Z80). C'est pourquoi il est nécessaire d'utiliser des mécanismes de détection d'erreurs intra-calculateurs pour assurer autant que possible cette hypothèse, et d'évaluer sa couverture pour connaître la fiabilité d'un système construit avec notre solution (voir § 4).

2.2 Services de base

Trois services nous semblent indispensables pour exécuter des applications temps-réel distribuées tolérantes aux fautes, à savoir la synchronisation des horloges, la détection des arrêts de calculateurs, et la diffusion fiable. Dans la suite de ce paragraphe, nous présentons succinctement les trois protocoles que nous avons conçus. Des détails sur le comportement temporel au pire cas de ces services peuvent être trouvés dans [2, 6]. Des détails sur leur implantation et sur leur dimensionnement (notamment le choix de la période des tâches implantant les services) peuvent être trouvés dans [11].

Le pré-requis principal pour exécuter ces services est de disposer d'une primitive pour transmettre un message en temps borné (système synchrone). Par ailleurs, ces trois services reposent essentiellement sur des tâches périodiques (pas de tâche déclenchée sur interruptions levées lors de l'arrivée de trames réseau) afin que leurs coûts d'exécution puissent être aisément intégrés dans une analyse d'ordonnancement.

2.2.1 Synchronisation d'horloges

Le protocole de synchronisation d'horloges que nous avons conçu garantit les propriétés d'*accord* et d'*exactitude* sur la valeur des horloges. La première stipule que la différence entre deux horloges est bornée par une constante appelée *précision*, et la seconde que la vitesse de chaque horloge est contenue dans un intervalle borné. Ce protocole est une adaptation du protocole de LUNDELIUS-LYNCH [19].

En plus de sa simplicité, le protocole de LUNDELIUS-LYNCH a été sélectionné car il repose essentiellement sur des activités périodiques. Ce protocole impose d'utiliser $3f + 1$ horloges (calculateurs) pour tolérer la défaillance de f horloges, c'est pourquoi nous supposons que les horloges sont fiables pour pouvoir utiliser moins de 3 calculateurs.

Deux modifications ont été apportées au protocole de LUNDELIUS-LYNCH. D'une part, pour obtenir la meilleure précision possible sans utiliser de matériel spécifique (la précision dépend de la latence réseau maximale), le protocole utilise directement le réseau physique sans passer par le service de diffusion fiable : pour masquer ω omissions successives, chaque calculateur diffuse $\omega+1$ messages contenant la valeur de son horloge. D'autre part, pour éviter que les horloges ne reculent lors des re-synchronisations, au lieu de changer la valeur d'une horloge, nous modifions la durée entre deux incrémentations de sa valeur.

2.2.2 Détection des arrêts de calculateurs

Le service de détection des arrêts (*failure detector*) est chargé de déterminer sur chaque calculateur la liste des calculateurs arrêtés. Une première caractéristique de ce service, commune avec celui de diffusion fiable, est d'être conçu de manière à ne pas dépendre de la précision de synchronisation d'horloges. En effet, puisque la synchronisation d'horloges est entièrement implantée en logiciel, la précision de synchronisation est nécessairement beaucoup plus élevée qu'en utilisant du matériel spécifique. Une seconde caractéristique commune aux deux services est de nécessiter l'utilisation d'un réseau physique où le temps d'accès pire cas au médium est borné, comme par exemple FDDI ou ATM.

Le protocole de détection des arrêts que nous avons conçu offre la propriété de *ponctualité* qui est essentielle dans notre cadre. Cette propriété stipule que la détection de l'arrêt ou du démarrage d'un calculateur est effectuée en temps borné. Ce protocole offre également la propriété de *vivacité* (l'arrêt ou le démarrage d'un calculateur est nécessairement détecté par les autres) et la propriété d'*intégrité* (pas de fausse détection de l'arrêt d'un calculateur).

Pour cela, nous utilisons un protocole très simple, puisqu'il consiste périodiquement à diffuser un message de vie sur le réseau physique et à consulter les messages de vie reçus d'autres calculateurs durant la dernière période. Si le calculateur i est $\omega + 2$ périodes sans recevoir de trames du calculateur j, alors le calculateur j est nécessairement arrêté, et si le calculateur i reçoit une trame du calculateur j qui était arrêté, alors nécessairement, il vient de redémarrer.

2.2.3 Diffusion fiable

Le protocole de diffusion fiable que nous avons conçu offre diverses propriétés dont les plus importantes sont la *ponctualité* (diffusion d'un message en temps borné), la *validité* (si un message est diffusé, et si l'émetteur ne s'arrête pas, alors le message est nécessairement livré aux destinataires non arrêtés), l'*accord* (si un message est adressé à deux destinataires non arrêtés, alors les deux le reçoivent, ou aucun), et la livraison selon l'ordre causal.

Sur chaque calculateur, le protocole est exécuté périodiquement (utilisation d'une tâche périodique qui est chargée d'exécuter le protocole, de lire les messages à émettre dans une file d'attente, et d'appeler les tâches destinataires des messages à livrer). Un message est diffusé $\omega + 1$ fois. Lorsqu'un calculateur destinataire d'un message le reçoit, il diffuse successivement $\omega + 1$ acquittements. Un calculateur destinataire d'un message le livre à l'applicatif s'il reçoit un acquittement pour ce message de tous les calculateurs destinataires et non arrêtés, sinon le message est oublié (pour garantir la propriété d'accord). L'accord étant obtenu en une phase, ce protocole suppose qu'au plus un calculateur s'arrête durant la diffusion d'un message. La causalité est garantie grâce à l'utilisation d'horloges vectorielles.

2.3 Construction d'applications tolérantes aux fautes

Une approche pour exécuter les applications en tolérant les fautes est d'utiliser uniquement des services au niveau du support d'exécution (*e.g.* service de réplication ou de sauvegarde/restauration d'état) qui agissent sans une quelconque participation de l'application. Cette solution, qui a l'avantage de dégager complètement le concepteur d'une application de la tolérance aux fautes, a l'inconvénient de rendre difficile l'évaluation précise des communications, des synchronisations et des surcoûts d'exécution ajoutés pour tolérer les fautes.

Or, pour satisfaire aux exigences des applications temps-réel strict, il est nécessaire de prouver avant l'exécution du système (hors-ligne) que toutes ses échéances seront respectées. Les tests d'ordonnançabilité hors-ligne nécessitent tous de connaître un ensemble de caractéristiques temporelles sur les tâches comme les lois d'arrivée, les temps d'exécution au pire cas [13], les messages transmis, les synchronisations, les temps d'utilisation des ressources, *etc*. Ces informations doivent donc être également connues pour les mécanismes de tolérance aux fautes.

C'est pourquoi, nous proposons une approche alternative à l'utilisation exclusive de services au niveau du support d'exécution (en complément de l'utilisation des services de base présentés dans le paragraphe précédent) qui permet de prendre en compte aisément les mécanismes de tolérance aux fautes dans les tests d'ordonnançabilité hors-ligne. Dans la suite, nous présentons les principes de cette approche, appelée réplication hors-ligne, puis nous présentons le recouvrement d'erreurs (*i.e.* de l'arrêt d'un ou plusieurs calculateurs), puis la reconfiguration d'un système suite au redémarrage d'un calculateur. Pour la détection d'erreurs, la détection distante des arrêts de calculateurs qui est nécessaire au recouvrement de l'arrêt de calculateurs a été présentée avec les services au § 2.2, et la détection des erreurs internes à un calculateur qui est nécessaire pour assurer l'hypothèse de silence sur défaillance sera présentée au § 4 (consacré à l'utilisation de matériel sur étagère).

2.3.1 Principe de la réplication hors-ligne

La réplication hors-ligne consiste à transformer automatiquement, avant l'exécution du système, les tâches d'une application afin d'ajouter la redondance et les blocs de base nécessaires aux différentes phases de la tolérance aux fautes (*e.g.* calcul de consensus, sauvegarde/restauration de l'état des calculs). La stratégie de réplication d'une application doit être définie par son concepteur qui indique pour chaque élément à fiabiliser (tâches ou portions de tâches) la technique de réplication à mettre en œuvre et le placement des copies. Après transformation, l'application tolérante aux fautes obtenue peut être soumise à un test d'ordonnançabilité qui vérifie l'éventuel respect des échéances, comme nous le verrons au § 3. Dans le cas où le besoin en fiabilité des tâches est variable, le concepteur d'une application peut adapter la technique de réplication utilisée (en particulier le niveau de réplication) selon les tâches afin d'économiser les ressources processeurs et réseau.

Pour pouvoir être transformées, les tâches doivent être conformes à un modèle de tâche qui indique comment sont structurées les tâches, les synchronisations possibles entre elles, ainsi qu'un ensemble de caractéristiques qui doivent être connues hors-ligne. Les points essentiels de ce modèle sont :

- Chaque tâche est modélisée en utilisant un *graphe orienté acyclique* (DAG). Un nœud correspond à un calcul sans synchronisation interne, et avec un temps d'exécution au pire cas connu. Un arc correspond à une contrainte de précédence entre deux nœuds, le nœud destination ne pouvant être exécuté qu'après le nœud source. Une contrainte de précédence peut relier deux nœuds placés sur des calculateurs distincts (auquel cas la tâche est distribuée). La transmission de données (d'un message) peut être associée à une contrainte de précédence. Enfin, le placement de chaque nœud sur un calculateur est statique ;
- Un ensemble d'*attributs* concernant les tâches doivent être connus hors-ligne, en particulier des *attributs temporels* (échéance, loi d'arrivée, temps d'exécution au pire cas), des *attributs de tolérance aux fautes* avec lesquels le concepteur d'une application indique les stratégies de réplication à mettre en œuvre, et des *attributs de synchronisation*. Les attributs de synchronisation permettent, entre autres, de définir des contraintes d'exclusion entre nœuds. Pour cela chaque nœud indique les ressources nécessaires à son calcul, et pour chacune, s'il souhaite l'utiliser en *mode exclusif* (pour la modifier) ou en *mode partagé* (uniquement pour la consulter). Chaque *ressource* peut aussi bien modéliser un dispositif matériel (capteur, actionneur) qu'une entité logicielle composée de données et de code pour agir sur ces données. Les données contenues dans les ressources sont les *seules* à persister après l'exécution d'une tâche.

2.3.2 Recouvrement d'erreurs

Nous proposons trois algorithmes de transformation pour recouvrir la défaillance d'un ou plusieurs calculateurs, mettant respectivement en œuvre au sein d'une tâche distribuée la réplication *active* [12], la réplication *semi-active* [4], et la réplication *passive* [5]. La réplication active consiste à exécuter en parallèle les mêmes calculs sur des calculateurs distincts. La réplication semi-active est similaire à la réplication active, excepté que les décisions communes à toutes les copies d'un calcul sont prises par un des calculateurs (appelé primaire), tandis qu'avec la réplication active, elles sont prises par un algorithme de consensus. La réplication passive consiste à exécuter un calcul sur un calculateur appelé primaire, tandis que les autres mémorisent régulièrement son état d'exécution servant à redémarrer en cas d'arrêt du primaire. Une comparaison de ces trois techniques de réplication dans le cadre des systèmes temps-réel strict est présentée dans [6, 8].

Ces trois techniques de réplication existent depuis de nombreuses années. Notre contribution est de les intégrer automatiquement à la structure d'une application en faisant apparaître *statiquement* les surcoûts en processeur, en réseau, et en synchronisations, afin de pouvoir intégrer les coûts de la réplication dans un test d'ordonnançabilité hors-ligne.

Exemple de réplication active : Nous proposons maintenant de concrétiser la réplication hors-ligne sur un exemple de transformation, présenté figure 2, en utilisant la réplication active sur une petite tâche distribuée. Le détail des algorithmes de transformation, des éléments de correction, et les hypothèses de fautes utilisées pour les capteurs et actionneurs selon la technique de réplication utilisée sont présentés dans [6].

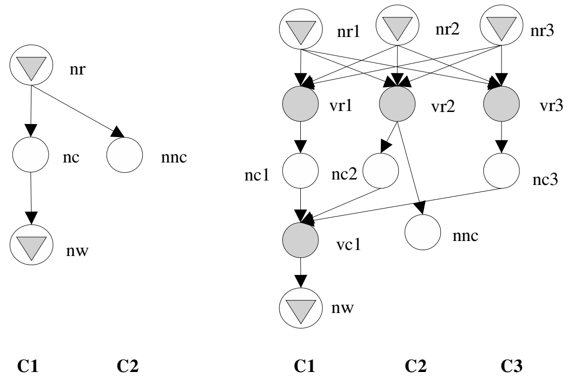


FIG. 2 – *Exemple d'application de la réplication active*

La tâche est composée de quatre nœuds: le nœud nr lit une donnée en utilisant un capteur, le nœud nw transmet le résultat à un actionneur, le nœud nc effectue des traitements critiques et le nœud nnc effectue des traitements non critiques.

Sur l'exemple, les nœuds nr et nc sont répliqués en utilisant la réplication active avec un degré de réplication de 3 sur les calculateurs C_1, C_2 et C_3. Il est ainsi possible de tolérer la défaillance d'un capteur ou de deux calculateurs.

Les copies de ces nœuds (*e.g.* les copies du nœud nr sont nommées $nr1$, $nr2$ et $nr3$ sur la figure) sont créées automatiquement. Pour les capteurs, le concepteur de l'application est en charge de s'assurer que les capteurs physiques sont répliqués, et de notifier l'outil de réplication de l'identité et de la localisation des copies des capteurs (nœuds $nr1$, $nr2$ et $nr3$ sur la figure).

Afin d'atteindre un consensus sur le résultat des copies d'un nœud, l'outil de réplication insère des blocs de base de tolérance aux fautes, nommés *voteurs*. Un voteur ($vr1$, $vr2$, $vr3$ et $vc1$ sur la figure) calcule une seule valeur à partir d'un ensemble de valeurs, chacune provenant d'une copie. Un voteur correspond à la fonction de convergence d'un protocole de consensus. La transmission des valeurs sur lequel l'accord doit être atteint est la responsabilité du service de diffusion fiable, qui est activé lorsque les données sont transmises selon les contraintes de précédence (flèches sur la figure). Tous les voteurs impliqués dans le calcul d'un consensus reçoivent des valeurs identiques grâce à la propriété d'accord de la diffusion fiable. L'hypothèse de silence sur défaillance des calculateurs garantit que toute valeur provenant d'un calculateur non arrêté peut être sélectionnée. C'est pourquoi un voteur *par défaut* retournant n'importe quelle valeur reçue en entrée est inséré automatiquement pour les nœuds effectuant uniquement des calculs (c'est-à-dire n'utilisant pas de capteurs).

Déterminisme : L'objectif du *déterminisme des copies* (d'un calcul) est de s'assurer que les copies non défaillantes restent toujours convergentes. Le déterminisme est un objectif clé pour tous les systèmes basés sur la réplication de tâches.

Avec notre modèle de tâches, deux principales sources d'indéterminisme sont possibles. La première, existant uniquement dans le cadre de la réplication active, est due aux copies de capteurs (nœuds $nr1$ à $nr3$ sur la figure) qui ne fournissent pas nécessairement les mêmes valeurs selon leur emplacement, même si elles sont lues au même instant. Cette source d'indéterminisme est réglée par l'utilisation d'une fonction de convergence adaptée dans les voteurs (le concepteur d'une application peut choisir parmi plusieurs prédéfinies [6]).

La seconde source d'indéterminisme (commune à toutes les techniques de réplication) est l'exécution dans un ordre différent, selon les calculateurs, des copies de plusieurs tâches/nœuds. Par exemple, avec la réplication active, si a_1 et b_1 (copies des nœuds a et b sur le calculateur C_1) ne s'exécutent pas dans le même ordre que a_2 et b_2 (copies des nœuds a et b sur le calculateur C_2), et si a et b modifient une même ressource, alors l'état de cette ressource diverge selon les calculateurs.

Pour éviter cela, une solution est de choisir en ligne (grâce à l'utilisation d'un protocole de synchronisation distribué) un ordre d'exécution identique pour les copies de nœuds effectuant des opérations conflictuelles. Toutefois cet ordre peut être contraire à celui choisi hors-ligne par le test d'ordonnançabilité, et donc mener à des dépassements d'échéances non prévus. C'est pourquoi, nous avons choisi de laisser à l'algorithme d'ordonnancement la responsabilité de calculer un ordre d'exécution non source d'indéterminisme. Pour cela, l'outil de réplication génère, en plus du graphe des tâches répliquées, des contraintes d'ordre qui doivent être respectées par tout algorithme d'ordonnancement en charge des tâches répliquées. Ainsi, un algorithme d'ordonnancement peut être indépendant (excepté concernant ces contraintes d'ordre) de la technique de réplication utilisée pour les tâches. Les contraintes d'ordre sont présentées au § 3.

2.3.3 Reconfiguration (traitement de fautes)

Une fois qu'une erreur a été détectée sur un calculateur, il est arrêté et redémarré immédiatement. Si le mécanisme d'auto-test intégré au moniteur de démarrage (BIOS pour les PC Intel) ne détecte pas d'erreur, alors nous considérons que la faute responsable de ce redémarrage est transitoire, et le calculateur est ré-intégré au système.

La ré-intégration d'un calculateur impose de re-synchroniser son horloge, et de mettre à jour les structures internes du support d'exécution. C'est pourquoi, chaque service gère sa propre ré-intégration en effectuant un traitement adéquat.

La ré-intégration impose également d'obtenir la valeur courante des structures de données applicatives répliquées sur différents calculateurs. Les structures de données qui ne sont pas modifiées durant l'exécution (graphes des tâches, code des nœuds) ne posent pas de difficultées particulières car elles sont liées statiquement avec le micro-noyau, qui est chargé lors du démarrage. Du fait du modèle de tâche que nous utilisons, et comme nous l'avons déjà évoqué, les structures de données qui doivent survivre à l'exécution des nœuds sont contenues dans les ressources. Ré-intégrer un calculateur impose seulement d'obtenir la valeur courante de toutes les ressources utilisées par les nœuds des tâches placées sur ce calculateur. Pour cela, l'outil de réplication génère automatiquement des *tâches de reconfiguration* : pour chaque ressource répliquée R, une tâche distribuée périodique T_R est générée, cette tâche diffusant la valeur courante de R à chaque calculateur l'utilisant. Cette tâche est périodique afin de pouvoir être prise en compte aisément au niveau du test d'ordonnançabilité.

3 Analyse d'ordonnancement

La réplication hors-ligne nous permet de ne pas imposer l'utilisation d'un algorithme d'ordonnancement pour les tâches tolérantes aux fautes [7]. Toutefois, n'importe quel algorithme d'ordonnancement ne peut pas être utilisé, puisqu'il doit satisfaire trois propriétés, à savoir, gestion de la distribution, gestion de contraintes de précédence et d'exclusion, et gestion d'une contrainte d'ordre nommée dans la suite IO (*identical order*) qui impose un ordre série sur l'exécution de nœuds. Cette contrainte est utilisée par l'outil de réplication pour que l'ordre d'exécution calculé par l'algorithme d'ordonnancement ne soit pas source d'indéterminisme.

Différents algorithmes d'ordonnancement distribués satisfont les deux premières exigences, mais à notre connaissance seul l'algorithme d'ordonnancement ORECA [15] et des travaux dérivés satisfont la troisième exigence. Plus précisément, ORECA garantit la propriété de sérialisabilité au niveau des tâches (construites dans un modèle semblable au nôtre), c'est-à-dire que pour deux tâches T_1 et T_2, sur tous les calculateurs où elles s'exécutent, alors elles le font dans le même ordre. ORECA garantit plus que les contraintes d'ordre IO qui imposent seulement un ordre série au niveau de certains nœuds.

Pour montrer que la réplication hors-ligne est ouverte à d'autres algorithmes d'ordonnancement, nous présentons dans la suite un des algorithmes d'ordonnancement à base de plans satisfaisant les trois exigences imposées par la réplication hors-ligne, cet algorithme étant une adaptation de l'algorithme d'ordonnancement de XU et PARNAS [20]. Nous montrons ainsi que certains algorithmes peuvent aisément être modifiés pour prendre en compte la contrainte d'ordre IO nécessaire à la réplication hors-ligne. Notre extension de l'algorithme d'ordonnancement de XU et PARNAS suppose, comme l'algorithme original, suppose que les tâches des applications sont périodiques.

Ensuite, nous présentons comment les coûts d'exécution des services de base (présentés au § 2.2) peuvent être pris en compte dans le test d'ordonnançabilité des applications tolérantes aux fautes.

3.1 Exemple d'algorithme pour les tâches tolérantes aux fautes

L'algorithme d'ordonnancement de XU et PARNAS [20] (XP dans la suite) est de type mono-processeur (non distribué), préemptif (un traitement peut être suspendu au profit d'un autre), hors-ligne (le test d'ordonnançabilité est déroulé avant exécution de l'application) et statique (il calcule un plan d'exécution).

Le modèle de tâches utilisé par l'algorithme est très proche de celui utilisé pour la réplication hors-ligne, excepté que les tâches sont non distribuées et doivent être périodiques. Ce modèle intègre la notion d'exclusion entre nœuds, ce qui permet de prendre en compte les synchronisations dues à l'utilisation de ressources dans notre modèle. Une telle contrainte est notée $a\ EX\ b$, si l'exécution du nœud a (respectivement de b) ne peut être suspendue au profit de l'exécution de b (respectivement a).

Le principe de XP est le suivant : à partir du problème initial (le graphe de chaque tâche, les contraintes de précédence, et les contraintes d'exclusion), l'algorithme calcule une solution initiale en utilisant un algorithme d'ordonnancement de type EDF (Earliest Deadline First). Cette solution est un plan indiquant les dates de début et de fin d'exécution de chaque nœud, ainsi que d'éventuelles dates de préemption pour exécuter des nœuds plus urgents.

Si cette solution ne satisfait pas toutes les échéances, XP calcule un *problème dérivé* issu du problème initial en ajoutant des contraintes de précédence et des contraintes de préemption (pour forcer un nœud à en préempter un autre si les deux sont prêts à s'exécuter). La principale caractéristique d'un problème dérivé est que toute solution à ce problème est solution du problème initial. Si la solution d'un problème dérivé ne satisfait pas toutes les échéances, de nouveaux problèmes dérivés sont calculés, et ce jusqu'à avoir parcouru l'ensemble des problèmes dérivés possibles. XP garantit que les problèmes dérivés sont construits de manière à parcourir toutes les solutions du problème initial (satisfaisant ou non les échéances). L'exploration des solutions est effectuée grâce à un « branch and bound ».

À partir de XP nous avons construit XPext [7, 6] pour prendre en compte les mécanismes de tolérance aux fautes. Plus précisément, XPext *i)* étend les contraintes d'exclusion EX à des groupes de nœuds, *ii)* est distribué, et *iii)* prend en compte la contrainte d'ordre IO. La nouvelle contrainte d'exclusion notée EX_e indique, pour $A\ EX_e\ b$, que l'exécution de l'ensemble des nœuds de l'ensemble A ne doit pas être suspendue au profit de l'exécution de b, et réciproquement.

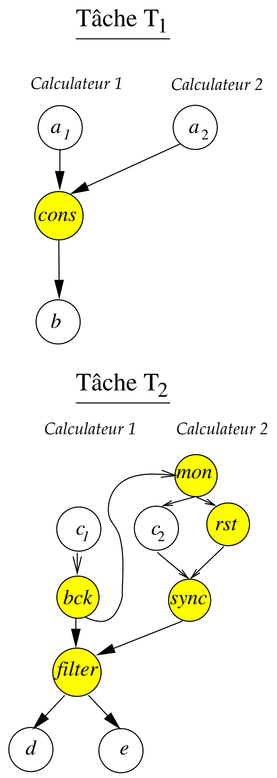
Pour gérer la distribution, XPext calcule grâce à l'algorithme d'ordonnancement EDF un plan par calculateur en distinguant deux temps au pire cas différents pour valider une contrainte de précédence, selon qu'elle soit entre deux nœuds placés sur le même calculateur ou non. Le calcul de ces deux temps est détaillé dans [9].

Pour gérer la contrainte de précédence IO, XPext utilise une version étendue du test d'éligibilité de XP ($Eligible_e(a,t)$ à la place de $Eligible(a,t)$) permettant à l'algorithme d'ordonnancement EDF de savoir si un nœud a peut être exécuté ou non sur le calculateur où il est placé à un instant t. Une contrainte d'ordre IO produite par l'outil de réplication est de la forme $(a,b)\ IO\ (c,d)$, avec a et c deux nœuds placés sur le même calculateur, tout comme b et d. Cette contrainte, indique que a et c doivent s'exécuter dans le même ordre que b et d.

Ainsi, on a :

$$
\begin{aligned}
Eligible_e(a,t) = & Eligible(a,t) \ \wedge \\
& \nexists b, c, d\ tq\ a, b\ IO\ c, d\ \wedge\ begin(c,t) = \infty\ \wedge\ begin(d,t) < begin(b,t)
\end{aligned}
$$

$begin(noeud,t)$ est une fonction qui indique pour la partie du plan déjà construite (*i.e.* avant la date t) la date de début d'exécution de $noeud$, ou bien ∞ si $noeud$ n'est pas commencé d'exécuter. La condition supplémentaire dans $Eligible_e$ signifie donc que le nœud a est éligible s'il n'existe pas de nœud b, c et d tels que *i)* $a, b\ IO\ c, d$, *ii)* l'exécution de c n'est pas commencée, et *iii)* l'exécution de d a commencé avant celle de b.



nœud	r	c	d
a_1	0	10	40
a_2	0	10	40
$cons$	0	1	66
b	38	9	66
c_1	0	10	30
bck	0	2	69
mon	0	1	69
rst	0	2	69
c_2	0	10	30
$sync$	0	1	69
$filter$	0	1	69
d	45	9	69
e	53	9	63

$\{a_1\}$ EX$_e$ c_1
$\{a_1\}$ EX$_e$ bck
$\{c_1, bck\}$ EX$_e$ a_1
$\{a_2\}$ EX$_e$ c_2
$\{c_2\}$ EX$_e$ a_2
$\{a_2\}$ EX$_e$ rst
$\{rst\}$ EX$_e$ a_2
$\{c_2\}$ EX$_e$ rst
$\{rst\}$ EX$_e$ c_2
$\{d\}$ EX$_e$ e
$\{e\}$ EX$_e$ d
(a_1, a_2) IO (bck, rst)

Remarque : Les attributs temporels des blocs de base de tolérance aux fautes ($cons$, bck, ...) sont calculés par l'outil de réplication en fonction des attributs temporels des autres nœuds, exceptés les temps d'exécution au pire cas (colonne **c**) qui doivent être calculés grâce à un outil adapté.

FIG. 3 – *Exemple de deux tâches tolérantes aux fautes*

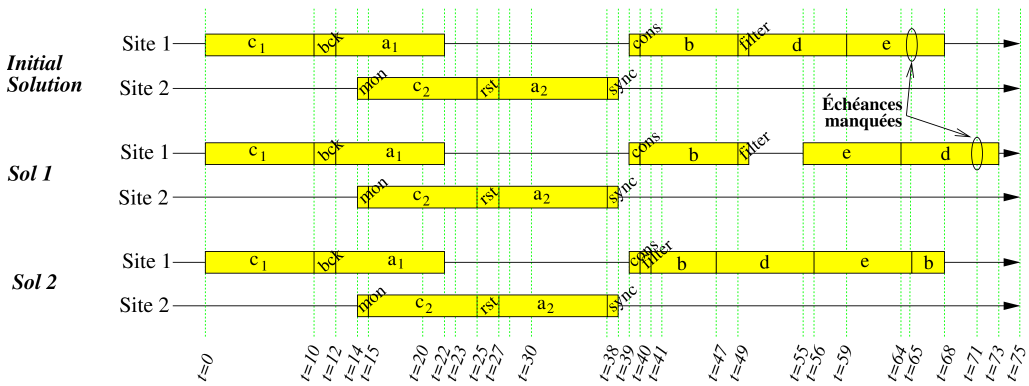


FIG. 4 – *Exemple de solution obtenue grâce à XPext*

Exemple d'utilisation de XPext : La figure 3 présente deux tâches répliquées T_1 et T_2. T_1 a été obtenue par application de la réplication active sur un nœud nommé a. Les copies de a sont a_1 et a_2. T_2 a été obtenue par application de la réplication passive sur un nœud nommé c. Les copies de c sont c_1 et c_2. Avant réplication, les nœuds a et c modifient une même ressource, et donc leurs exécutions doivent s'exclure.

Les nœuds en grisé sur la figure correspondent aux blocs de base de tolérance aux fautes. $cons$ est un voteur, comme présenté au § 2.3.2. bck et rst sont chargés de la sauvegarde de l'état des ressources utilisées par c_1, afin de pouvoir effectuer une reprise arrière.

mon sélectionne l'exécution de c_2 si le calculateur où est placé c_1 est arrêté, ou sinon sélectionne l'exécution de $rst.\ sync$ et $filter$ sont chargés de sélectionner pour d et e le message issu de c_1 ou de c_2 selon que c_1 s'exécute ou non. Pour plus de détails sur la réplication passive, voir le document [6].

Le tableau de la figure 3 indique pour chaque nœud sa date d'arrivée (colonne r), son temps d'exécution au pire cas (colonne c) et son échéance (colonne d). À droite du tableau sont présentées les contraintes d'exclusion entre les nœuds ainsi qu'une contrainte d'ordre. Ces contraintes résultent du fait que les nœuds a et c modifient une même ressource. Pour cet exemple, nous supposons que le temps pour valider une contrainte de précédence entre deux calculateurs est de 2 unités de temps.

La figure 4 présente trois solutions obtenues à partir des tâches T_1 et T_2 et des contraintes associées. La première correspond à la solution initiale, dans laquelle le nœud e ne respecte pas son échéance. C'est pourquoi XPext construit deux problèmes dérivés Prb_1 et Prb_2 dont les solutions sont respectivement Sol_1 et Sol_2. Prb_1 est obtenu en ajoutant une contrainte de précédence de e vers d, tandis que Prb_2 est obtenu en ajoutant deux contraintes de préemption, à savoir e préempte b et d préempte b. Dans la solution Sol_1, d ne respecte pas son échéance, mais par contre dans Sol_2 tous les nœuds respectent la leur. C'est pourquoi Sol_2 peut être retenue comme ordre pour exécuter T_1 et T_2.

3.2 Prise en compte des services de base

La quasi totalité des micro-noyaux ou systèmes d'exploitation embarqués fournissent un algorithme d'ordonnancement à priorités fixes et FIFO (les tâches de même priorité sont ordonnées selon leur ordre d'arrivée). C'est pourquoi, pour des raisons d'efficacité et de simplicité, nous avons conçu les tâches des services pour être ordonnancées avec un tel algorithme. Nous supposons donc l'existence d'un mécanisme pour faire cohabiter deux ordonnanceurs différents pour les services et les applications. La description de ce mécanisme, mis en œuvre sous forme d'un service s'exécutant au-dessus d'un micro-noyau, est présentée dans [7, 6].

Par construction, les tâches des services sont plus prioritaires que les tâches des applications. En appliquant les résultats de TINDELL et CLARK sur l'analyse d'ordonnancement holistique à priorités fixes dans les systèmes distribués [18], il est possible de déterminer le temps au pire cas pour lancer l'exécution d'une tâche applicative, le temps au pire cas pour terminer l'exécution d'un nœud et valider une contrainte de précédence, et le temps d'exécution au pire cas de chacun des nœuds des tâches applicatives en tenant compte de l'exécution des tâches de fond des services [9]. Cette troisième information permet de calculer l'ordre d'exécution des nœuds des tâches applicatives (par exemple grâce à XPext) en faisant abstraction de l'existence des tâches de service.

Pour effectuer l'analyse d'ordonnancement des services, il est nécessaire de connaître la charge provoquée par les applications sur les services. Les résultats de l'analyse d'ordonnancement des services influencent ceux de l'analyse d'ordonnancement des applications, et donc la charge applicative, d'où problème de récursivité. Pour éviter cela, une solution empirique, mais simpliste, est de supposer des coûts au pire pour les services, de calculer en fonction d'eux l'ordre d'exécution des tâches des applications, puis de calculer d'après cet ordre les coûts réels des services. S'ils sont inférieurs à ceux supposés, alors l'ordre des tâches applicatives peut être utilisé tel quel, sinon il doit être recalculé.

4 Hypothèses de défaillances et matériel sur étagère

Au début de ce document, nous avons indiqué les hypothèses de défaillances nécessaires pour la correction de notre solution, à savoir calculateurs à silence sur défaillance, horloges fiables, et réseau à omissions bornées. Si ces hypothèses ne sont pas vérifiées, alors le comportement d'un système basé sur notre solution devient non spécifié.

Dans notre travail, nous nous sommes concentrés dans un premier temps sur la première hypothèse, qui est très certainement la plus difficile à assurer sur du matériel sur étagère (quelques solutions pour assurer les deux autres hypothèses sont présentées dans [6]).

Dans la suite, nous présentons les mécanismes de détection d'erreurs mis en œuvre au niveau des services et des applications pour assurer l'hypothèse de silence sur défaillance en présence de fautes physiques, puis nous présentons l'évaluation de ces mécanismes pour les services en utilisant une méthode logicielle d'injection de fautes.

4.1 Mécanismes de détection d'erreurs

Le processeur de chaque calculateur est partagé par les services et les applications, c'est pourquoi le résultat de l'exécution de chacun d'eux doit être vérifié grâce à des mécanismes de détection d'erreurs. Ces mécanismes sont locaux à chaque calculateur, et donc ne détectent pas d'erreurs à distance, comme le fait le service de détection des arrêts de calculateur (§ 2.2).

Lorsqu'une erreur est détectée, le calculateur fautif est immédiatement arrêté pour éviter la propagation de la faute sur un autre calculateur, ne provoque une violation de l'hypothèse d'arrêt sur défaillance.

4.1.1 Au niveau des services

Au niveau des services, nous utilisons un ensemble de mécanismes de détection d'erreurs adaptés au temps-réel strict, c'est-à-dire dont les surcoûts en temps d'exécution sont bornés et autant que possible réduits. Pour chacun, nous indiquons un nom symbolique qui sera ré-utilisé lors de la présentation des résultats, puis une brève description :

- *Mécanismes à base de redondance* :
  - CODING : ajout d'une somme de contrôle à chaque message transmis par les communications ;
  - CODING_STATIC : ajout d'une somme de contrôle à la structure de données statique décrivant le graphe des tâches. La conformité entre la structure de données et la somme de contrôle est vérifiée quand le support d'exécution est inactif (tâche *idle*) ;

    - ROBUST : vérification de la validité des structures de données (listes et piles) en utilisant des informations redondantes (typiquement un chaînage supplémentaire pour les listes) ;
- *Mécanismes basés sur le comportement* :
    - DEADLINE : vérification du respect des échéances des tâches ;
    - ARRIVAL : vérification du respect de la loi d'arrivée des tâches ;
    - TIMEOUT : vérification du respect des temps d'exécution au pire cas des tâches ;
    - FLOW_CALL_GRAPH : vérification en-ligne de la validité du graphe d'appel des fonctions. Le graphe d'appel est obtenu automatiquement par reconnaissance syntaxique du code des services ;
    - FLOW_AUTOMATON : vérification de la validité des transitions dans les automates utilisés par les protocoles des services (*e.g.* changement de l'état d'exécution des tâches — attente, exécution, terminé —) ;
    - ECI : insertion, dans la mémoire non utilisée, d'instructions qui déclenchent une exception lorsqu'elles sont exécutées ;
    - VMEMORY : utilisation d'un contexte mémoire séparé pour chaque tâche de chaque service ;
    - STRUCTURE : vérification de la validité des structures de données (listes, piles et tableaux) sans ajout de redondance (par exemple vérification des bornes lors des accès aux tableaux) ;
    - SEMAN : utilisation d'assertions dans le code pour détecter la violation d'invariants ;
- *Mécanismes bas-niveaux* :
    - CPU : mécanismes de base de détection d'erreurs inclus dans le processeur (*e.g.* instruction invalide) ;
    - OS : mécanismes de détection d'erreurs inclus dans le micro-noyau sous-jacent (*e.g.* codes d'erreurs retournés par les appels systèmes).

4.1.2 Au niveau des applications

En complément des mécanismes de détection d'erreurs au niveau des services, les applications doivent elles-mêmes vérifier la correction de leur exécution pour détecter les erreurs contenues dans leur état d'exécution. Pour cela, nous utilisons un algorithme de transformation mettant en œuvre la *réplication temporelle*. Cette technique consiste à exécuter deux fois un même traitement (un même nœud) et à comparer le résultat des deux exécutions. L'utilisation de la réplication hors-ligne permet la prise en compte automatique de la réplication temporelle par l'algorithme d'ordonnancement d'une application.

4.2 Résultats obtenus par injection de fautes

Les mécanismes de détection d'erreurs du *support d'exécution* (*i.e.* les services plus le micro-noyau sous-jacent — CHORUS dans notre cas —) ont été évalués sur le prototype

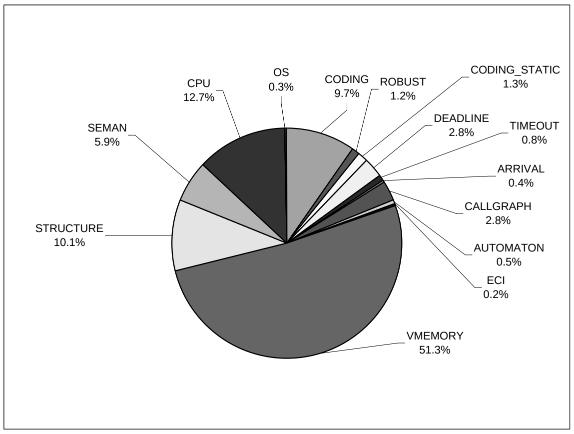


FIG. 5 – *Efficacité des mécanismes de détection d'erreurs*

HADES qui implante l'intégralité de notre solution. Cette évaluation a été effectuée avec une méthode logicielle d'injection de fautes. Cette méthode consiste pour chaque expérience à inverser, à une date aléatoire, un bit pris au hasard dans le code, la pile, ou les données globales du support d'exécution, et à étudier le comportement du système global. Pour cela, une application de charge distribuée (non répliquée) dont les résultats sont prévisibles est utilisée. *Seules* les expériences où l'erreur injectée a été lue par le processeur (erreur dite activée) sont retenues, et les chiffres qui suivent ont été obtenus pour chacun grâce à 3000 expériences pour lesquelles l'erreur a été activée. Entre deux expériences, les calculateurs sont redémarrés.

La couverture de l'hypothèse d'arrêt sur défaillance correspond au pourcentage d'expériences où, soit aucune défaillance (domaine des valeurs et temporel) de l'application n'a été constatée, soit seul le calculateur où la faute a été injectée s'est arrêté (sans transmission de résultats erronés).

Sans l'utilisation de mécanismes de détection d'erreurs, la couverture du silence sur défaillance est de 80,6 %, tandis qu'elle est de 99,1 % lorsque tous sont utilisés. Ce second chiffre est très encourageant, puisque les violations du silence sur défaillance sont divisées par 22 fois grâce à nos mécanismes de détection d'erreurs.

Sur la figure 5, nous présentons l'efficacité de chaque mécanisme de détection d'erreurs en indiquant le pourcentage d'expériences où ce mécanisme a détecté une erreur par rapport au nombre total d'expériences où une erreur a été détectée (une expérience s'arrête dès la première erreur détectée). Sur cette figure, on peut noter que les quatre mécanismes les plus efficaces en terme d'erreurs détectées sont VMEMORY, CPU, STRUCTURE, et CODING.

De plus amples détails sur nos résultats d'injection de fautes dans le support d'exécution sont présentés dans [10].

5 Conclusion

Dans cet article, nous avons présenté la synthèse d'une solution globale pour concilier tolérance aux fautes, contraintes temporelles strictes et utilisation de matériel (cartes mères et réseau) sur étagère. Notre solution repose sur l'utilisation de services exécutés au-dessus d'un micro-noyau, et mettant en œuvre des protocoles essentiels pour la tolérance aux fautes (synchronisation d'horloges, détection des arrêts, diffusion fiable). Ces services sont nécessaires pour l'exécution de tâches répliquées automatiquement par un outil hors-ligne. Ces tâches répliquées font apparaître statiquement tous les surcoûts et synchronisations nécessaires à la tolérance aux fautes, ce qui permet de les prendre en compte dans un test d'ordonnançabilité. Notre solution repose sur l'hypothèse de calculateur à silence sur défaillance, d'où l'utilisation de mécanisme de détection d'erreurs pour assurer cette hypothèse avec du matériel standard.

Jusqu'à présent, nous nous sommes concentrés sur les systèmes statiques où tout est supposé connu. Les travaux futurs concernent les environnements dynamiques où la charge de travail ne peut être anticipée de manière certaine.

Références

[1] T. ABDELZAHER, S. DAWSON, W.-C. FENG, F. JAHANIAN, S. JOHNSON, A. MEHRA, T. MITTON, A. SHAIKH, K. SHIN, Z. WANG, H. ZOU, M. BJORKLAND, et P. MARRON. « ARMADA Middleware and Communication Services ». *Real-Time Systems*, 16(2):127–154, 1999.

[2] E. ANCEAUME, G. CABILLIC, P. CHEVOCHOT, et I. PUAUT. « A Flexible Run-time Support for Distributed Dependable Hard Real-time Applications ». Dans *Proceedings of International Symposium on Object-oriented Real-time distributed Computing*, pages 310–319, St Malo, France, mai 1999.

[3] J. ARLAT, J.-P. BLANQUART, A. COSTES, Y. CROUZET, Y. DESWARTE, J.-C. FABRE, H. GUILLERMAIN, M. KAÂNICHE, K. KANOUN, J.-C. LAPRIE, C. MAZET, D. POWELL, C. RABÉJAC, et P. THÉVENOD. *Guide de la sûreté de fonctionnement*. Cépaduès, 1995.

[4] P.A. BARRETT, A.M. HILBORNE, P. VERÍSSIMO, L. RODRIGUES, P.G. BOND, D.T. SEATON, et N.A. SPEIRS. « The Delta-4 Extra Performance Architecture (XPA) ». Dans *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 481–488, Newcastle, U.K., juin 1990.

[5] N. BUDHIRAJA, K. MARZULLO, F.B. SCHNEIDER, et S. TOUEG. « *Distributed Systems* », Chapitre The Primary-Backup Approach, pages 169–197. Addison–Wesley / ACM Press, 1993.

[6] P. CHEVOCHOT. « *Conception de systèmes distribués temps-réel strict tolérants aux fautes* ». Thèse, IFSIC — Université Rennes I, décembre 1999.

[7] P. CHEVOCHOT et I. PUAUT. « Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies ». Dans *Proceedings of the*

6th International Conference on Real-Time Computing Systems and Applications, pages 356–363, Hong Kong, décembre 1999.

[8] P. CHEVOCHOT et I. PUAUT. « Tolérance aux fautes dans les systèmes répartis temps-réel strict ». *Techniques et Sciences Informatiques*, 18(8), 1999.

[9] P. CHEVOCHOT et I. PUAUT. « Holistic Schedulability Analysis of a Fault-Tolerant Real-Time Distributed Run-time Support ». Dans *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, Cheju Island, Korea, décembre 2000.

[10] P. CHEVOCHOT et I. PUAUT. « Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components ». Dans *Proc. of the International Conference on Dependable Systems and Networks (DSN'01)*, juillet 2001.

[11] P. CHEVOCHOT, I. PUAUT, G. CABILLIC, A. COLIN, D. DECOTIGNY, et M. BANÂTRE. « Hades: a distributed system for dependable hard real-time applications built from COTS components ». Rapport Technique 1357, IRISA, France, octobre 2000. Soumis à IEEETC pour l'issue spéciale Embedded Fault-Tolerant Computer Systems.

[12] M. CHÉRÈQUE, D. POWELL, P. REYNIER, J.-L. RICHIER, et J. VOIRON. « Active Replication in Delta-4 ». Dans *Proceedings of the 22th International Symposium on Fault-Tolerant Computing*, pages 28–37, Boston, Massachusetts, USA, juin 1992.

[13] A. COLIN et I. PUAUT. « Worst case execution time analysis for a processor with branch prediction ». *Real-Time Systems*, 18(2):249–274, avril 2000.

[14] H. KOPETZ et G. GRUNSTEILD. « TTP - A Protocol for Fault-Tolerant Real-Time Systems ». *IEEE Computer*, 27(1):14–23, janvier 1994.

[15] E. LEDINOT et G. LE LANN. « Évaluation d'une méthode de génie système pour l'avionique modulaire ». Dans *Proceedings of Real-Time Systems*, pages 267–275, Paris, France, janvier 1998.

[16] H. MADEIRA et J.G. SILVA. « Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking ». Dans *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pages 350–359, Austin, Texas, USA, juin 1994.

[17] D. POWELL, J. ARLAT, L. BEUS-DUKIC, A. BONDAVALLI, P. COPPOLA, A. FANTECHI, E. JENN, C. RABÉJAC, et A. WELLINGS. « GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems ». *IEEE Transactions on Parallel and Distributed Systems*, 10(6):204–226, juin 1999.

[18] K. TINDELL et J. CLARK. « Holistic Schedulability Analysis for Distributed Hard Real-Time Systems ». *Microprocessors and Microprogramming*, 40:117–134, 1994.

[19] J. Lundelius WELCH et N. LYNCH. « A New Fault-Tolerant Algorithm for Clock Synchronization ». *Information and Computation*, 77:1–36, 1988.

[20] J. XU et D.L. PARNAS. « Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations ». *IEEE Transactions on Software Engineering*, 16(3):360–369, mars 1990.