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

La réplication hors-ligne comme approche flexible pour la tolérance aux fautes des systèmes temps-réel dur

Pascal Chevochot — INRIA / IRISA
Isabelle Puaut — INSA / IRISA
Campus de Beaulieu, F-35042 Rennes cedex
{pchevoch,puaut}@irisa.fr

Résumé

Nous décrivons dans cet article un outil flexible (Hydra) pour tolérer les fautes dans les systèmes distribués temps-réel dur qui est transparent aux applications. Notre approche est flexible dans le sens où elle est extensible, indépendante d'un mécanisme particulier du support d'exécution ou du matériel, et adaptable aux besoins applicatifs. Elle est fondée sur la réplication hors-ligne qui consiste à modifier automatiquement la structure des tâches applicatives grâce à des schémas de transformation. Ces schémas implantent la réplication active, passive, et semi-active afin de traiter les erreurs des sites, et implantent la réplication temporelle afin de détecter les erreurs. La réplication hors-ligne simplifie l'intégration de la tolérance aux fautes dans les algorithmes d'ordonnancement et le test qui vérifie la correction du comportement temporel des systèmes.

Abstract

We describe in this paper a flexible tool (Hydra) for fault tolerance in hard real-time distributed systems, which is transparent to applications. Our approach is flexible in the sense that it is extensible, independent of a particular run-time support mechanism or hardware, and adaptable to applications requirements. It is based on off-line replication which consists in automatically modifying the structure of the application tasks thanks to given transformation schemes. These schemes implement active, passive, and semi-active replication in order to treat site errors and to implement temporal replication in order to detect errors. Off-line replication simplifies the integration of fault-tolerance into scheduling algorithms and into the test which verifies the correctness of the system's temporal behavior.

Ce travail est financé par la DGA/DSP, #198C4980031303011.

1 Introduction

Un *système temps-réel dur* est un système ayant des contraintes temporelles strictes (*e.g.* pilote automatique d'avion, chaîne de montage). Ces contraintes rendent difficile la conception de mécanismes de tolérance aux fautes et mènent à des solutions qui sont spécifiques à une application (*e.g.* mécanismes de tolérance aux fautes intégrés dans le code applicatif ou dépendants de la sémantique des données utilisées), qui dépendent de mécanismes du support d'exécution (*e.g.* protocole réseau, algorithme d'ordonnancement), ou qui dépendent de matériel spécifique (*e.g.* mémoire stable, bus répliqué). De telles solutions sont difficilement réutilisables, ce qui provoque des coûts de développement élevés. Par ailleurs, certains systèmes embarqués imposent de limiter les ressources matérielles utilisées afin de réduire les coûts matériels des plates-formes d'exécution et leur consommation en énergie. Il est donc souhaitable de fiabiliser uniquement les portions critiques de ces applications.

Un système temps-réel dur doit utiliser un algorithme d'ordonnancement adapté à ses besoins. Cet algorithme calcule un ordre dans lequel les tâches doivent être exécutées, et utilise un *test de faisabilité* pour vérifier que l'ordre d'exécution calculé respecte toutes les contraintes temporelles des tâches, y compris en présence du plus important pic de charge et du pire scénario de faute. La présence de contraintes temps-réel dur impose que le test de faisabilité soit vérifié hors-ligne (avant exécution) pour prouver la correction du système. Un algorithme d'ordonnancement est basé sur un *modèle de tâche* qui définit la manière dont les tâches sont structurées (*e.g.* utilisation de ressources, présence de contraintes de précédence), et inclut assez d'informations pour que leur comportement temporel soit prévisible (*e.g.* période des tâches).

Pour être adaptés aux systèmes temps-réel dur, les mécanismes de tolérance aux fautes (ainsi que tous ceux du support d'exécution) doivent être *prévisibles*, *i.e.* leurs temps d'exécution au pire cas doivent pouvoir être déterminés hors-ligne afin d'être intégrés dans les tests de faisabilité. De plus, si les tâches sont répliquées, l'ordre d'exécution des copies de chaque tâche doit pouvoir être calculé par les algorithmes d'ordonnancement. Il existe donc nécessairement des dépendances entre les mécanismes de tolérance aux fautes et les algorithmes d'ordonnancement dans les systèmes temps-réel dur.

Nous décrivons dans cet article un outil, appelé HYDRA, pour traiter automatiquement les erreurs dans les systèmes distribués temps-réel dur. Ce travail fait parti du projet HADES (Highly Available Distributed Embedded System) [1] qui a pour objectif la spécification d'un environnement flexible pour la construction de systèmes distribués fiables temps-réel.

L'approche utilisée par HYDRA consiste à transformer hors-ligne les tâches non tolérantes aux fautes en tâches tolérantes aux fautes, en modifiant automatiquement leur structure afin de répliquer leurs calculs. HYDRA est *flexible*, *i.e.* :

- *extensible :* de nouvelles stratégies de tolérance aux fautes peuvent être ajoutées ou des stratégies existantes peuvent être modifiées sans répercution sur le reste du système ;
- *adaptable aux besoins applicatifs :* HYDRA offre différentes stratégies de tolérance aux fautes, et il permet de choisir différentes stratégies selon la portion d'application considérée en fonction de leur besoin en sûreté de fonctionnement et de la fiabilité des ressources utilisées ;
- *générique :* HYDRA n'impose pas d'utiliser un mécanisme particulier pour le support d'exécution (*e.g.* protocole réseau TDMA, algorithme d'ordonnancement à priorités fixes) ou du matériel spécifique (*e.g.* mémoire stable, bus répliqué). Les dépendances entre HYDRA, le matériel et le support d'exécution sont faibles et clairement identifiées : le support d'exécution et le matériel doivent être conformes à un modèle de faute, le support d'exécution doit respecter quatre propriétés en cas de faute, et enfin les algorithmes d'ordonnancement doivent respecter des contraintes d'ordre indiquées par HYDRA pour chaque tâche.

Le reste de cet article est organisé comme suit : la section 2 présente le modèle de système sur lequel notre proposition est basée. La section 3 présente l'outil de réplication automatique des tâches. Enfin, la section 4 situe notre travail par rapport aux travaux existants. Nous concluons dans la section 5.

2 Modèle du système

Pour construire une application tolérante aux fautes, il est nécessaire de s'appuyer sur un modèle architectural (§ 2.1) et un modèle de faute (§ 2.2). Les tâches sont construites grâce à un modèle de tâche générique appelé dans la suite *modèle de tâche* HADES (§ 2.3).

2.1 Modèle architectural

Les calculs sont distribués sur n sites connectés par un réseau. Chaque site est également connecté à des capteurs et des actionneurs pour respectivement observer et agir sur l'environnement extérieur. Un site est composé d'un processeur, d'une mémoire, et d'une horloge.

Une *ressource* modélise tout composant matériel ou toute entité logicielle qui peut être partagée entre plusieurs calculs (*e.g.* segment de mémoire, verrou). Une ressource est locale à un site. Nous appelons *état d'une ressource* une zone de mémoire qui est associée à cette ressource. L'état d'une ressource persiste tant que son site n'est pas arrêté.

2.2 Modèle de faute

Dans ce travail, nous considérons les fautes permanentes et transitoires qui sont la conséquence de phénomènes physiques. Pour les sites, nous supposons que chacun d'eux est à *silence sur défaillance*, *i.e.* chaque site envoie des résultats corrects dans le domaine valué et le domaine temporel aux autres sites et aux actionneurs, ou reste définitivement silencieux (*i.e.* est arrêté). Pour le réseau physique, nous supposons qu'il n'est jamais partitionné et qu'au plus ω omissions successives peuvent se produire. Pour les actionneurs, nous supposons qu'ils sont fiables (des dispositifs matériels permettent de construire un actionneur tolérant aux fautes grâce à la réplication matérielle). Enfin, les capteurs sont sujets à des erreurs valuées et temporelles.

2.3 Modèle de tâche Hades

La structure et les attributs de chaque tâche construite dans le modèle de tâche HADES et qui sont nécessaires dans le cadre de cet article sont présentés dans la suite.

2.3.1 Structure des tâches

Chaque tâche est décrite par un graphe acyclique direct appelé *HEUG* (*HADES Elementary Unit Graph*), dont les nœuds modélisent les calculs, et les arcs les contraintes de précédence entre eux.

Un nœud (également appelé *EU* pour *Elementary Unit*) correspond à une séquence de code sans synchronisation. Un *EU* utilise un ensemble de variables d'entrée et calcule un ensemble de variables de sortie.

Un arc modélise une *contrainte de précédence* entre deux nœuds. Une contrainte de précédence de eu_i vers eu_j (notée $eu_i \rightarrow eu_j$) indique que eu_j peut commencer son exécution seulement lorsque l'exécution de eu_i est terminée. Une variable de sortie *out* d'un *EU* eu_i

peut être copiée dans la variable d'entrée *in* d'un *EU* qui est destination d'une contrainte de précédence issue de eu_i. Nous notons cette copie $in := out$.

Un *HEUG* peut également inclure des *constructions conditionnelles*. Une construction conditionnelle permet de demander dynamiquement l'exécution de p sous-graphes parmi n possibles ($p \leq n$).

2.3.2 Attributs des tâches

Un ensemble d'attributs de synchronisation, temporels, de distribution et de tolérance aux fautes est spécifié pour chaque tâche.

Les *EUs* peuvent se synchroniser entre eux en utilisant des ressources partagées. Un *EU* eu_i peut acquérir une ressource dans deux modes différents. Dans le *mode exclusif*, aucun autre *EU* ne peut l'utiliser jusqu'à ce que eu_i la libère. Dans le *mode partagé*, d'autres *EUs* peuvent l'acquérir dans le même mode. Les attributs de synchronisation spécifient pour chaque *EU* les ressources qu'il utilise et leur mode d'accès. Un attribut associé avec chaque contrainte de précédence ($eu_i \rightarrow eu_j$) indique les ressources qui ne doivent pas être libérées lorsque eu_i termine son exécution. Ces ressources sont dites *héritées* de eu_i par eu_j.

Les attributs temporels indiquent pour chaque tâche sa loi d'arrivée (*i.e.* périodique, sporadique, apériodique) et son échéance (*i.e.* la date à laquelle l'exécution doit être terminée, relativement à la date de demande d'exécution). Pour chaque *EU*, ils indiquent son échéance et son temps d'exécution au pire cas.

Les attributs de distribution définissent pour chaque *EU* le site où il doit être exécuté (le site d'exécution d'un *EU* est connu statiquement).

Les attributs de tolérance aux fautes spécifient pour chaque *HEUG* ou portion de *HEUG* (*i.e.* un sous-graphe) quelle stratégie de réplication doit être appliquée, et le degré de réplication nécessaire.

3 Outil de réplication automatique des tâches (Hydra)

L'approche suivie pour tolérer les fautes affectant une tâche construite dans le modèle de tâche HADES est de transformer hors-ligne (avant exécution) cette tâche en modifiant automatiquement sa structure afin de répliquer ses calculs. Les différentes transformations possibles sont spécifiées grâce à des *schémas de transformation*, chaque schéma de transformation implantant une stratégie de réplication particulière. Les schémas de transformations sont appliqués par HYDRA. La réplication hors-ligne des tâches est possible car les synchronisations et les chemins de communication des tâches sont identifiés statiquement dans le modèle de tâche HADES.

Dans la suite, nous présentons brièvement trois stratégies de réplication utilisées par HYDRA pour traiter les erreurs, et une utilisée pour faire respecter l'hypothèse de silence sur défaillance. Puis, nous présentons la transformation automatique des tâches qui est réalisée par HYDRA (§ 3.2). Ensuite, nous présentons les bénéfices de notre approche pour la construction de systèmes temps-réel dur (§ 3.3). À la fin de ce paragraphe, nous décrivons brièvement l'implantation d'HYDRA(§ 3.4).

3.1 Stratégies de réplication

Pour traiter les erreurs (arrêts de sites ou erreurs de capteur), HYDRA inclut trois schémas de transformation implantant respectivement la *réplication active* [5], la *réplication passive* [4] et la *réplication semi-active* [2]. La réplication active consiste à exécuter le même calcul en parallèle sur différents sites. La réplication passive consiste à exécuter un calcul sur un site appelé *primaire*, tandis que des *sites de sauvegarde* mémorisent l'état du calcul du site primaire. La réplication semi-active est similaire à la réplication active, excepté que les décisions communes à toutes les répliques sont prises par un site alors qu'avec la réplication active, elles sont prises grâce à un protocole de consensus distribué.

Chacune de ces stratégies de réplication exhibe différents avantages : la réplication passive consomme moins de ressource processeur que la réplication active car, sur les sites de sauvegarde, le coût processeur pour mémoriser l'état d'un calcul est généralement moins important que le coût nécessaire à son exécution. Ainsi, les sites de sauvegarde disposent de plus de ressources disponibles pour exécuter d'autres tâches (*e.g.* des tâches sans contraintes temporelles strictes). La réplication active réduit les temps d'exécution au pire cas en comparaison de la réplication passive à cause du temps nécessaire pour effectuer un recouvrement arrière et continuer les calculs sur un des sites de sauvegarde. De plus, la réplication active peut traiter les capteurs fautifs, contrairement à la réplication passive ou semi-active. La réplication semi-active exhibe une consommation processeur et un temps d'exécution au pire cas compris entre la réplication active et la réplication passive.

Les mécanismes de détection d'erreur d'un site inclus dans un support d'exécution ne peuvent pas faire respecter une couverture de 100% de l'hypothèse de silence sur défaillance, essentiellement parce qu'ils ne peuvent pas tester la correction des résultats des tâches applicatives. Aussi, nous proposons un schéma de transformation implantant la *réplication temporelle* [8] pour détecter les erreurs valuées dans les variables de sortie des *EUs* et dans l'état des ressources qu'ils utilisent. La réplication temporelle consiste à exécuter le même calcul plusieurs fois (généralement 2 fois) en utilisant les même ressources et en comparant les résultats obtenus. Entre chaque exécution, les états des ressources utilisées sont réinitialisés avec les valeurs qu'ils avaient avant la première exécution. Si les variables de sortie ou les états des ressources divergent, le site fautif est arrêté.

3.2 Transformation des tâches

Les attributs de tolérance aux fautes du modèle de tâche HADES permettent au concepteur d'une application de spécifier un ensemble de sous-graphes distincts qui doivent être répliqués (chacun de ces sous-graphes est appelé *motif* dans la suite). Un motif est la granularité à laquelle les mécanismes de tolérance aux fautes sont appliqués. Chaque motif est composé d'*EUs* assignés au même site, possède un seul nœud initial et un seul nœud final. Deux motifs distincts appartenant au même *HEUG* peuvent être placés sur des sites différents.

Pour chaque motif gr, un attribut spécifie le schéma de transformation à utiliser pour traiter les erreurs, le nombre de copies de gr qui sont nécessaires, et les sites où elles doivent être placées. Un attribut spécifie également si le schéma pour la réplication temporelle doit être utilisé pour détecter les erreurs. Si c'est le cas, il doit être appliqué avant le schéma de réplication active, passive ou semi-active.

Un schéma de transformation remplace un motif gr par n copies de gr, avec n le *degré* *de réplication* (*i.e.* le nombre de copies à utiliser), et connecte chaque copie avec des *EUs* qui implantent des blocs de base de tolérance aux fautes (*e.g.* protocole de consensus). Les copies d'un motif conservent ses attributs temporels (*e.g.* échéance).

Pour assurer la généricité, notre approche n'impose pas au support d'exécution un mécanisme particulier, mais elle impose que le support d'exécution respecte quatre propriétés d'exécution (§ 3.2.1) qui sont nécessaires pour la correction des schémas de transformation. Bien que tous les schémas de transformation mentionnés précédemment aient été spécifiés et implantés, pour des considérations de place, nous présentons seulement dans la suite les schémas pour la réplication active (§ 3.2.2) et passive (§ 3.2.3). Ensuite, nous montrons comment le déterminisme des tâches répliquées est assuré *i.e.* comment les copies d'un motif répliqué calculent les même valeurs (§ 3.2.4).

3.2.1 Propriétés d'exécution pour la réplication des tâches

La correction des applications répliquées repose sur quatre propriétés qui doivent être respectées par le support d'exécution. Ces propriétés sont assurées par le service d'exécution distribué du support d'exécution de HADES [1] en s'appuyant sur un service de diffusion fiable des messages.

Soit une contrainte de précédence $(eu_i \rightarrow eu_j)$ d'une tâche T. Par définition, $(eu_i \rightarrow eu_j)$ est *validée* lorsqu'elle n'empêche pas eu_j de commencer son exécution. Par définition, la validation de $(eu_i \rightarrow eu_j)$ est *correcte* si et seulement si le support d'exécution en charge de l'exécution de eu_j a été notifié par le support d'exécution en charge de eu_i de la terminaison de eu_i. La validation de $(eu_i \rightarrow eu_j)$ est donc incorrecte si le site de eu_i est arrêté lors de l'exécution de T.

Propriété 1 (Vivacité) *Si une tâche T commence son exécution au temps t, alors tout site non arrêté participe à l'exécution de T s'il est en charge d'un de ses* EUs*, même si d'autres sites impliqués dans l'exécution de la tâche T sont arrêtés au temps t.*

Propriété 2 (Ponctualité) *Soit $(eu_i \rightarrow eu_j)$ une contrainte de précédence et soit s le site où est placé eu_i. La contrainte de précédence est validée au plus tard δ_{VPS} après la terminaison de eu_i (si s ne s'arrête pas avant la terminaison de eu_i) ou après l'arrêt de s (si s s'arrête avant la terminaison de eu_i). δ_{VPS} dépend de l'implantation du support d'exécution.*

Propriété 3 (Correction) *Soit $(eu_i \rightarrow eu_j)$ une contrainte de précédence. Si le site de eu_i termine l'exécution de eu_i au temps t, et si les sites de eu_i et eu_j ne s'arrêtent pas avant le temps $t + \delta_{VPS}$, alors la validation de $(eu_i \rightarrow eu_j)$ est correcte.*

Propriété 4 (Accord) *Pour chaque* EU *eu_i, tous les sites non arrêtés où sont placés un* EU *eu_j tel que $(eu_i \rightarrow eu_j)$ s'accordent sur la correction de la validation de la contrainte de précédence $(eu_i \rightarrow eu_j)$.*

3.2.2 Schéma pour la réplication active

La figure 1 présente le schéma de transformation pour la réplication active de degré n d'un motif *gr* dont les variables de sortie sont nommées Out_i, $1 \leq i \leq p$.

Ce schéma transforme le motif *gr* en n copies de *gr* appelées gr_k, $1 \leq k \leq n$, qui sont exécutées en parallèle sur n sites différents. Toutes les ressources utilisées par *gr* sont également répliquées sur tous ces sites. Le schéma de transformation connecte les n copies

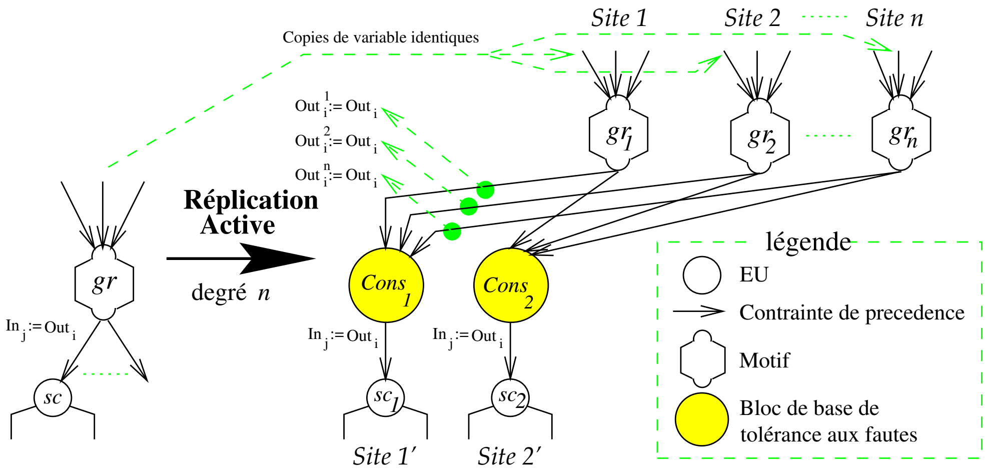

FIG. 1 – *Schéma de transformation pour la réplication active de degré n*

de gr avec m blocs de base de tolérance aux fautes appelés $Cons_k$, $1 \leq k \leq m$, chaque $Cons_k$ étant la destination d'une contrainte de précédence issue des n copies de gr. Le nombre m d'*EUs* $Cons_k$ est égal au nombre de sites différents sur lesquels sont placés un successeur sc de gr (ou une copie de sc si sc est également répliqué). Par exemple, sur la figure 1, $m = 2$ car sc est répliqué sur deux sites. Les *EU* $Cons_k$ $(1 \leq k \leq m)$ sont en charge de déterminer une valeur de consensus pour chaque variable de sortie Out_i de gr $(1 \leq i \leq p)$, *i.e.* chaque $Cons_k$ doit calculer la même valeur correcte pour chaque variable Out_i à partir des valeurs de Out_i de toutes les copies de gr, même en présence de sites arrêtés ou de capteurs défaillants. Pour cela, si $Cons_k$ doit déterminer une valeur de consensus pour la variable de sortie Out_i de gr, alors Out_i de la $j^{\text{ème}}$ copie de gr est copiée dans la variable d'entrée Out_i^j de $Cons_k$. La valeur de consensus calculée est affectée à la variable de sortie Out_i de $Cons_k$.

Si les *EUs* de gr n'utilisent pas de capteurs, chaque $Cons_k$ utilise la fonction *quelconque* qui prend la valeur de Out_i de n'importe qu'elle copie de gr sur un site non arrêté. La fonction *quelconque* permet de masquer $n-1$ sites arrêtés. Dans le cas contraire, si les *EUs* de gr utilisent des capteurs, chaque $Cons_k$ utilise la fonction spécifiée dans un attribut du *HEUG* associé à gr qui est spécifique à la réplication active. En effet, les valeurs produites par chaque copie non fautive d'un capteur répliqué peuvent varier selon les sites et la date lorsqu'un *EU* les lit. Les fonctions les plus utilisées sont la fonction *f-point médian* et *f-moyenne* [6] qui permettent de masquer f capteurs fautifs, avec $f \leq \lfloor \frac{n-1}{2} \rfloor$.

Correction : La propriété de *Vivacité* est essentielle afin de garantir que tous les sites non arrêtés impliqués dans l'exécution de la tâche répliquée exécuteront les copies de gr et les *EUs* $Cons_k$. L'accord sur les valeurs des variables de sortie des copies de gr peut être implanté en seulement une phase grâce à l'hypothèse de silence sur défaillance des sites (qui assure que la valeur d'une variable de sortie d'un *EU* ne peut être incorrecte que si cet *EU* accède à un capteur erroné), et grâce aux propriétés de *Correction* et d'*Accord*. Enfin, la propriété *Ponctualité* assure que le délai pour atteindre l'accord est borné.

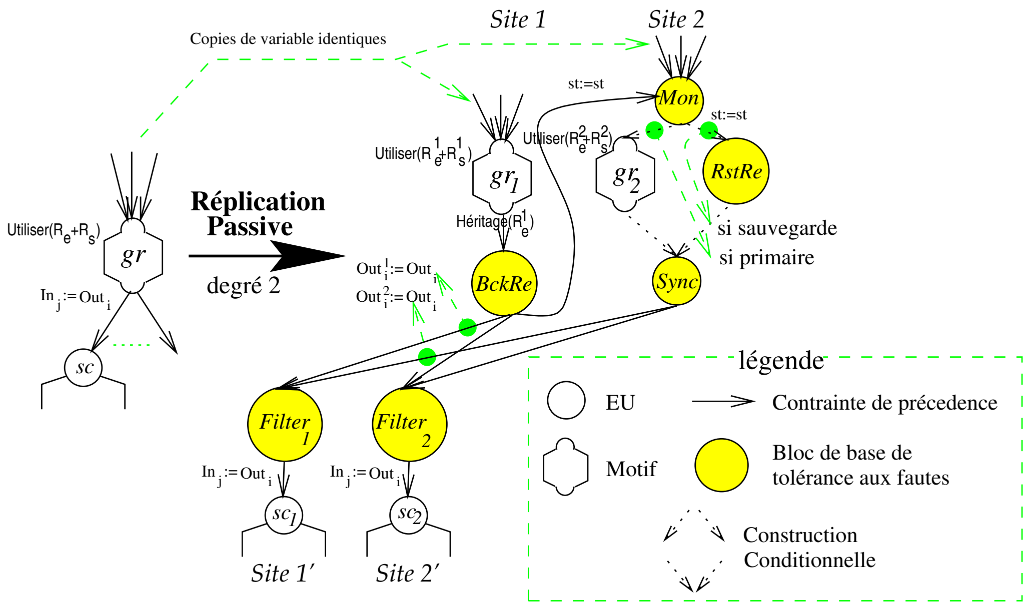

FIG. 2 – *Schéma de transformation pour la réplication passive de degré 2*

3.2.3 Schéma pour la réplication passive

La figure 2 présente le schéma de transformation pour la réplication passive de degré 2 d'un motif *gr* dont les variables de sortie sont nommées Out_i, $1 \leq i \leq p$. Soit R_e l'ensemble des ressources utilisées par *gr* en mode exclusif, et R_s l'ensemble de celles utilisées en mode partagé. La réplication passive impose que chaque ressource de R_e soit sauvegardable, *i.e.* que son état peut être sauvegardé et restauré.

Ce schéma de transformation remplace *gr* par deux copies de *gr* localisées sur deux sites différents. Un des deux sites (site 1 sur la figure) est le site primaire, tandis que l'autre site (site 2) est le site de sauvegarde sauf si le site 1 est arrêté. Les ressources utilisées par *gr* sont répliquées sur les sites 1 et 2, et sont notées R_e^1 et R_s^1 sur le site 1, et R_e^2 et R_s^2 sur le site 2.

Sur le site 1, le bloc de base de tolérance aux fautes *BckRe* est en charge de sauvegarder l'état de l'ensemble des ressources de R_e^1 (*i.e.* de copier leur état dans la variable de sortie *st*). La variable de sortie *st* est copiée dans la variable d'entrée *st* du bloc de base de tolérance aux fautes *Mon* exécuté sur le site 2 qui est un successeur de *BckRe*. L'*EU Mon* décide si le site 2 est site de sauvegarde ou site primaire selon la correction de la validation de la contrainte de précédence (*BckRe* $\rightarrow$ *Mon*).

Si le site 2 est sauvegarde (*i.e.* si le site 1 n'est pas arrêté), l'*EU Mon* sélectionne, grâce à une construction conditionnelle, l'exécution du bloc de construction de tolérance aux fautes *RstRe* qui restaure l'état de l'ensemble des ressources de R_e^1 (*i.e.* il copie sa variable d'entrée *st* dans l'état des ressources de R_e^2). Si le site 2 est primaire (*i.e.* si le site 1 est arrêté) l'exécution est reprise sur ce site, *i.e.* *Mon* sélectionne l'exécution de gr_2. Le bloc de construction de tolérance aux fautes *Sync* est un artifice de construction pour terminer la construction conditionnelle sur le site 2.

m blocs de construction de tolérance aux fautes *Filter*$_i$ sont en charge de collecter les valeurs des variables de sortie des copies de gr situées sur les m sites où un successeur sc de gr est placé (ou une des copies de sc si sc est répliqué).

Le schéma de transformation pour la réplication passive de degré deux permet de masquer un site arrêté et suppose des capteurs fiables.

Correction : La propriété *Vivacité* garantit que l'*EU Mon* est toujours exécuté même si le site 1 est arrêté. La propriété de *Correction* garantit que le site 2 est primaire si et seulement si le site 1 est arrêté, et cela en un temps borné grâce à la propriété *Ponctualité*. Sur le site 1, l'héritage des ressources de R_e^1 entre gr et *BckRe* assure que l'état des ressources de R_e^1 n'est pas modifié par l'exécution concurrente d'un *EU* avant que cet état ne soit sauvegardé par *BckRe*. Dans le cas contraire, si l'exécution est reprise sur le site 2, l'état des ressources de R_e^2 peut être incohérent.

3.2.4 Maintien du déterminisme des tâches répliquées

Soit $Rep(A)$ l'ensemble de motifs répliqués d'une application A grâce à Hydra. L'application A est *déterministe* [13] si, étant donné le même état initial pour toutes les copies de chaque ressource répliquée, après chaque exécution sans faute de $g \in Rep(A)$, chaque copie d'un *EU* de g a les même valeurs pour ses variables de sortie, et chaque copie d'une ressource répliquée utilisée par un *EU* de g a le même état.

Deux causes d'indéterminisme sont possibles. La première, qui apparaît seulement avec la réplication active, est due aux capteurs répliqués. Le schéma de réplication active résout cette source d'indéterminisme grâce au protocole de consensus implanté par les blocs de tolérance aux fautes $Cons_i$ (voir § 3.2.2).

La seconde source d'indéterminisme est l'exécution dans un ordre incompatible d'*EUs* appartenant à des motifs répliqués différents et qui modifient des ressources communes. Illustrons cette source d'indéterminisme avec la réplication passive : soit la ressource R qui est modifiée par les *EUs* de deux motifs gr_a et gr_b. Ces deux motifs sont transformés par le schéma de réplication passive en les copiant sur les sites 1 et 2. Les modifications de la copie de R effectuées par gr_a (respectivement gr_b) sur le site 1 sont sauvegardées par l'*EU* $BckRe_a$ (respectivement $BckRe_b$), et sont restaurées sur le site 2 par l'*EU* $RstRe_a$ (respectivement $RstRe_b$). $BckRe_a$ et $BckRe_b$ doivent être exécutés dans le même ordre que $RstRe_a$ et $RstRe_b$ pour maintenir l'état des copies de R cohérent. Dans le cas contraire, si l'exécution est reprise sur le site 2, des modifications de l'état de R effectuées sur le site 1 peuvent être perdues sur le site 2.

Nous pensons que cette seconde source d'indéterminisme doit être résolue par les algorithmes d'ordonnancement. Dans le cas contraire, si cette source d'indéterminisme est résolue par un mécanisme de synchronisation indépendant, l'ordre d'exécution des tâches peut différer de celui calculé par leur algorithme d'ordonnancement. L'exécution ne reflète alors pas l'analyse effectuée par le test de faisabilité, ce qui mène à un comportement incorrect de tout le système.

Pour assurer la généricité, nous n'imposons pas un algorithme d'ordonnancement spécifique pour gérer les applications répliquées. Hydra indique pour chaque application répliquée A, grâce à un attribut du modèle de tâche Hades, un ensemble de contraintes d'ordre entre les *EUs* des motifs répliqués de A. Ces contraintes d'ordre doivent être respectées par l'algorithme d'ordonnancement du système, et sont les seules dépendances entre la réplication et les algorithmes d'ordonnancement.

Dans le projet Hades, nous utilisons plusieurs algorithmes d'ordonnancement. L'un d'eux est une extension de l'algorithme de Xu & Parnas [15] aux systèmes distribués. L'intégration de la gestion des contraintes d'ordre dans cet algorithme d'ordonnancement a été effectuée sans difficulté. Pour cela, nous avons dû essentiellement étendre le test qui détermine si un *EU* peut être exécuté à une date fixée.

3.3 Avantages de la réplication hors-ligne

La réplication hors-ligne permet de définir avant exécution, dans le modèle de tâche Hades, les copies des *EUs*, les blocs de base de tolérance aux fautes, les contraintes de précédence et les synchronisations nécessaires pour l'exécution d'une tâche répliquée. Ceci est une caractéristique déterminante car elle permet à un algorithme d'ordonnancement et à son test de faisabilité de traiter les tâches d'une application et les mécanismes de tolérance aux fautes de manière uniforme et indépendante des techniques de réplication utilisées. De plus, la réplication hors-ligne permet de limiter les mécanismes nécessaires dans le support d'exécution, ce qui simplifie la prise en compte de ses coûts d'exécution dans le test de faisabilité. Enfin, elle permet d'ajouter de nouvelles techniques de réplication sans avoir à modifier le support d'exécution et donc à analyser de nouveau son comportement temporel.

Mais la réplication hors-ligne impose l'utilisation d'un modèle de tâche comme le modèle de tâche Hades, ce qui exige des efforts de conception pour porter les applications existantes. Toutefois, l'utilisation d'un tel modèle de tâche est une contrainte commune à n'importe quel test de faisabilité pour système temps-réel dur, qu'il utilise ou non l'approche décrite dans cet article.

3.4 Implantation de Hydra

Chaque application est implantée en deux parties : la première, appelée *initialisation*, est chargée de construire en mémoire, grâce à une librairie, les *HEUGs*. L'autre partie contient la séquence de code de chaque *EU* des tâches de l'application. Ces séquences de code utilisent les fonctions d'une librairie pour accéder aux variables d'entrée et de sortie de leur *EU* et pour demander l'exécution d'un sous-graphe d'une construction conditionnelle. Les applications sont actuellement écrites dans le langage C, et l'initialisation des tâches est contenue dans un fichier distinct.

Chaque schéma de transformation est implanté dans Hydra sous forme d'une *fonction de réplication*. Cette implantation est aisée grâce à l'utilisation d'une bibliothèque bien adaptée pour manipuler des portions de *HEUG*.

Pour transformer une application, Hydra exécute son initialisation pour construire en mémoire les *HEUGs* des tâches, puis les réplique. Ensuite, il génère un nouveau fichier d'initialisation qui reflète les transformations des *HEUGs*. Hydra génère également le code source des blocs de base de tolérance aux fautes qui sont nécessaires aux tâches répliquées.

Pour chaque ressource répliquée R, Hydra crée une nouvelle tâche chargée d'initialiser chaque copie de R lorsque son site démarre grâce aux états des copies de R sur les autres sites. Ces tâches sont nécessaires lorsqu'un site démarre après une faute transitoire.

4 Travaux existants

À notre connaissance, seul [12] décrit une solution pour répliquer les tâches hors-ligne indépendamment d'un algorithme d'ordonnancement en s'appuyant sur un modèle de tâche générique. Mais ce travail est limité à la réplication active et ne prend pas en compte les problèmes de déterminisme liés aux décisions d'ordonnancement.

Différents travaux traitent de l'ordonnancement des copies de tâches répliquées (*e.g.* [9, 3]), mais ces travaux sont spécifiques à une implantation particulière d'une technique de réplication et ne considèrent pas l'ordonnancement des blocs de base de tolérance aux fautes.

Différents environnements comprenant un support d'exécution, un algorithme d'ordonnancement et un test de faisabilité hors-ligne permettent de construire des systèmes temps-réel dur tolérants aux fautes, mais à notre connaissance aucun d'eux ne satisfait les propriétés d'extensibilité, d'adaptabilité et de généricité énoncées au § 1. Par exemple, les environnements Mars [11], Maft [10, 7] et Guards [14] utilisent uniquement la réplication active et fournissent un seul algorithme d'ordonnancement, excepté Guards qui en fournit trois. Chacun de ces algorithmes d'ordonnancement et leur test de faisabilité s'appuient sur des spécificités de la technique de réplication utilisé par leur environnement. En particulier, ces trois environnements divisent les sites d'un système en sous-ensembles disjoints, chaque site d'un ensemble exécutant les mêmes copies de tâches. Cette répartition des copies de tâches sur les sites permet de masquer en partie la réplication des tâches aux algorithmes d'ordonnancement.

5 Conclusion

Dans cet article, nous avons décrit l'outil Hydra qui est extensible, adaptable et générique pour traiter automatiquement les erreurs dans les applications distribuées temps-réel dur. Notre approche consiste à transformer hors-ligne la structure des tâches applicatives non tolérantes aux fautes en des tâches tolérantes aux fautes. Ces transformations sont effectuées selon des schémas de transformation qui réalisent différentes stratégies de réplication. Les schémas de transformation assurent l'extensibilité, car il est aisé de rajouter un nouveau schéma pour le traitement ou la détection des erreurs. L'adaptation aux besoins applicatifs est assurée car le concepteur d'une application peut choisir, à la granularité d'une portion de tâche distribuée, entre la réplication active, passive ou semi-active pour traiter les défaillances, et il peut choisir d'utiliser la réplication temporelle pour détecter les erreurs. Finalement, la généricité est assurée car les dépendances entre Hydra, le matériel et le support d'exécution sont faibles et clairement identifiées, et n'impose pas d'utiliser du matériel spécifique.

L'apport principal de la réplication hors-ligne pour les systèmes temps-réel dur est de permettre à un algorithme d'ordonnancement et à son test de faisabilité de traiter les tâches d'une application et les mécanismes de tolérance aux fautes de manière uniforme et indépendante des techniques de réplication utilisées.

Les travaux en cours concernent la détection des erreurs d'un site grâce au support d'exécution afin d'offrir la meilleure couverture possible pour l'hypothèse de silence sur défaillance des sites.

Démonstration

Une démonstration de Hydra avec le support d'exécution de Hades peut être trouvée dans le document Web `http://www.irisa.fr/solidor/work/hades.html`

Bibliographie

1. Anceaume (E.), Cabillic (G.), Chevochot (P.) et Puaut (I.). – A flexible run-time support for distributed dependable hard real-time applications. *In : Proceedings of International Symposium on Object-oriented Real-time distributed Computing.* – St Malo, France, mai 1999.
2. Barrett (P.A.), Hilborne (A.M.), Veríssimo (P.), Rodrigues (L.), Bond (P.G.), Seaton (D.T.) et Speirs (N.A.). – The Delta-4 extra performance architecture (XPA). *In : Proc. FTCS-20*, pp. 481–488. – Newcastle, U.K., juin 1990.
3. Bertossi (A.), Fusiello (A.) et Mancini (L.). – Fault-tolerant deadline-monotonic algorithm for scheduling hard-real-time tasks. *In : Proceedings of the 11th International Parallel Processing Symposium.* – Geneva, Switzerland, avril 1997.
4. Budhiraja (N.), Marzullo (K.), Scheneider (F.B.) et Toueg (S.). – *Distributed Systems*, chap. The Primary-Backup Approach, pp. 169–197. – Addison–Wesley, 1993.
5. Chérèque (M.), Powell (D.), Reynier (P.), Richier (J.-L.) et Voiron (J.). – Active replication in Delta-4. *In : Proc. FTCS-22*, pp. 28–37. – Boston, USA, juin 1992.
6. Dolev (D.), Lynch (N.A.), Pinter (S.S.), Stark (E.W.) et Weihl (W.E.). – Reaching approximate agreement in the presence of faults. *In : Proc. of the Third Symp. on Reliability in Dist. Software and Database Systems*, pp. 145–154. – Clearwater Beach, USA, octobre 1983.
7. Elvany (M.C. Mac) et Hugue (E.). – Guaranteed task deadlines for fault-tolerant workloads with conditional branches. *Real-Time Systems*, vol. 3, n 3, 1991, pp. 275–305.
8. Fuchs (E.). – An evaluation of the error detection mechanisms in MARS using software-implemented fault injection. *In : 2nd European Dependable Computing Conf.*, pp. 73–90. – Taormina, Italy, octobre 1996.
9. Ghosh (S.), Melhem (R.) et Mossé (D.). – Fault-tolerant scheduling on a hard real-time multiprocessor system. *IEEE Trans. on Parallel and Dist. Systems*, vol. 8, n 3, mars 1997, pp. 272–284.
10. Kieckhafer (R.M.), Walter (C.J.), Finn (A.M.) et Thambidurai (P.M.). – The MAFT architecture for distributed fault tolerance. *IEEE Trans. on Computers*, vol. 37, n 4, avril 1988, pp. 398–405.
11. Kopetz (H.), Damm (A.), Koza (C.), Mulazzani (M.), Scwabl (W.), Senft (C.) et Zainlinger (R.). – Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, vol. 9, février 1989, pp. 25–40.
12. Mossé (D.). – *Design, Development, and Deployment of Fault-Tolerant Applications for Distributed Real-Time Systems.* – Thèse de PhD, University of Maryland, 1993.
13. Poledna (S.). – *Fault-Tolerant Real-Time Systems – The Problem of Replica Determinism.* – Kluwer Academic Publishers, 1996.
14. Wellings (A.J.), Beus-Dukic (Lj.) et Powell (D.). – Real-time scheduling in a generic fault-tolerant architecture. *In : Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 390–398. – Madrid, Spain, décembre 1998.
15. Xu (J.) et Parnas (D.L.). – Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, vol. 16, n3, mars 1990, pp. 360–369.