8ème édition du congrès international pluridisciplinaire
QUALITA 2009 – Besançon (France)
Du 18 au 20 mars 2009

SERVICE DE DIAGNOSTIC POUR COMPOSANTS LOGICIELS

BUI Thi Quynh, AKTOUF Oum-El-Kheir, D'ORAZIO Laurent, DANG Michel

LCIS – Grenoble INP
50 rue B. de Laffemas, BP 54, 26902 Valence Cedex 9, France
Phone : (33) (0) 4.75.75.94.46, Fax : (33) (0) 4.75.75.94.50
Email : {Thi-Quynh.Bui, Oum-El-Kheir.Aktouf, Laurent.Dorazio, Michel.Dang}@esisar.inpg.fr

Résumé :

Le développement de logiciels à base de composants permet la construction d'applications complexes en rassemblant des composants existants indépendants, réduisant ainsi les coûts du processus de développement. Les systèmes logiciels actuels deviennent de plus en plus distribués et opèrent dans des environnements hautement dynamiques. La sûreté de fonctionnement des applications à base de composants devient alors un défi important. Dans ce papier, nous proposons un service de diagnostic qui augmente la fiabilité des applications à base de composants en nous intéressant en particulier aux applications embarquées. Ce service de diagnostic permet de détecter et de localiser les composants fautifs durant l'exécution du système. A notre connaissance, il s'agit du premier travail visant à augmenter la fiabilité des applications embarquées à base de composants, en se basant sur une approche par diagnostic.

Abstract:

Component-based software development paradigm enables the construction of complex applications by assembling existing self-contained components. The cost of the software development process can then be sharply reduced. Current software systems are becoming more distributed and operate in highly dynamic environments. Therefore, dependability of component-based applications is an important research issue. In this paper, we propose a diagnosis service that enhances dependability of component-based applications, especially embedded ones. This diagnosis service provides the ability of detection and location of faulty components during runtime. As far as we know it is the first work that applies system-level diagnosis in embedded software component based applications for increasing the reliability of the whole system.

Mots clés : Sûreté de fonctionnement, diagnostic, intergiciel, systèmes embarqués, tolérance aux fautes

Keywords: Dependability, diagnosis, middleware, embedded systems, fault tolerance

1. Introduction

Au cours de ces dernières années, les statistiques ont révélé que 98% des processeurs fabriqués sont destinés aux systèmes embarqués. Les systèmes embarqués sont présents dans divers domaines : l'automobile, l'aéronautique ou encore les télécommunications. Pour ces systèmes, la sûreté de fonctionnement est essentielle, surtout quand des vies humaines sont concernées, comme cela peut être le cas dans les systèmes de transport.

Les technologies à base de composants logiciels sont quant à elles de plus en plus utilisées pour développer des systèmes complexes et améliorer la composition, la réutilisation, la modularité et la configurabilité. En parallèle, les logiciels actuels deviennent de plus en plus distribués et opèrent dans des environnements hautement dynamiques. La sûreté de fonctionnement des applications embarquées à base de composants devient alors un défi important [OCC05, SOM05].

Dans ce contexte, la plupart des approches proposées sont fondées sur la duplication des composants et le masquage de fautes [BUI07a]. Les résultats sont alors garantis corrects bien que certaines fautes puissent corrompre le fonctionnement de quelques composants applicatifs. De telles solutions sont néanmoins

très coûteuses, ce qui est particulièrement problématique pour des applications embarquées où les ressources sont limitées. Une solution alternative est le diagnostic du système qui s'intéresse à la capacité des composants à déterminer l'état global du système. Les avantages du diagnostic sont l'autonomie de l'application du point de vue de la sûreté de fonctionnement, la réduction des coûts liés à la tolérance aux fautes et une meilleure utilisation des ressources du système. Ces aspects sont très importants dans beaucoup de systèmes et particulièrement dans les systèmes embarqués. C'est pourquoi nous nous sommes intéressés aux solutions à base de diagnostic.

Le diagnostic au niveau système, initialement proposé par Preparata *et al.* [PRE67], permet de déterminer l'état du système vis-à-vis des fautes pouvant toucher ses composants. L'idée fondamentale est d'utiliser des tests inter-composants. Un composant est alors utilisé pour tester et déterminer l'état d'un autre composant. Le diagnostic peut être centralisé ou distribué. Dans le premier cas, un composant central supposé fiable contrôle l'exécution des tests et calcule l'état du système à partir des résultats des tests. Dans le second cas, aucun composant central n'est requis. Chaque composant correct détermine de manière autonome l'état global du système. Selon la manière dont les tests sont alloués, les algorithmes de diagnostic peuvent être statiques ou adaptatifs. Dans les algorithmes statiques, les relations de test sont établies avant le processus de diagnostic. Au contraire, elles sont dynamiques dans la stratégie adaptative et dépendent des résultats de tests précédents.

Ce papier se focalise sur la description globale de notre service de diagnostic basé sur les tests inter-composants. Il complète les résultats présentés dans nos précédentes publications [BUI07a, BUI07b] qui détaillent respectivement l'implémentation des tests inter-composants et les fonctions de base du service de diagnostic. Notre approche de diagnostic vise à améliorer la sûreté de fonctionnement d'applications à base de composants, proposant un compromis compétitif entre les performances et les coûts. Pour valider notre approche, nous avons conduit quatre cas d'étude expérimentaux par simulation, dans un système bancaire, un système de climatisation, un système automobile et en considérant un modèle de système générique. Les résultats obtenus constituent les premiers pas vers notre objectif global qui est la construction d'applications tolérantes aux fautes dans les systèmes embarqués.

Ce papier est organisé de la manière suivante : la section 2 présente notre service de diagnostic. La section 3 décrit le fonctionnement du service de diagnostic. Les résultats obtenus sont brièvement présentés dans la section 4. Finalement, la section 5 conclut cet article et donne des perspectives de recherche.

2. Service de diagnostic

Dans cette section, nous citons les problématiques devant être abordées, les solutions possibles et présentons l'architecture de notre service de diagnostic.

2.1. Problèmes soulevés

Le service de diagnostic proposé fournit la possibilité de détecter et de localiser les composants fautifs durant l'exécution du système. Il se base sur des tests inter-composants en ligne. Ceci soulève certains problèmes devant être abordés :

- (1) Implémentation des tests : comment tester un composant ?
- (2) Concurrence : les tests peuvent interférer avec la fonctionnalité normale du composant. Que doit faire le composant si pendant le processus de test, il reçoit une demande de service fonctionnel d'autres composants et réciproquement ?
- (3) Exactitude de l'état du système : le test en ligne est réalisé dans un environnement réel. Si le test manipule et change des données fonctionnelles du composant, il doit fournir des mécanismes de retour arrière, en garantissant que les données modifiées pendant le test retrouvent leur état d'origine pour préserver l'exactitude de l'état du système.
- (4) Efficacité : l'approche de diagnostic doit considérer les ressources disponibles (mémoire,

consommation CPU ou encore bande passante) et s'adapter afin de ne pas perturber le fonctionnement normal du système cible.

Dans la section suivante, nous décrivons les solutions possibles aux problèmes énumérés ci-dessus.

2.2. Les solutions

Nous utiliserons l'exemple d'un lecteur de carte bancaire pour illustrer les concepts introduits pour résoudre les problèmes ci-dessus. Ce système se compose de quatre composants :

- Le composant de validation du code pin vérifie le code pin de la carte bancaire fourni par l'utilisateur. La carte sera bloquée après trois essais incorrects et sera débloquée par le banquier en entrant un code de sécurité.
- Le composant de dépôt est responsable du dépôt d'argent.
- Le composant de retrait gère le retrait d'argent.
- Le composant de consultation fournit le détail du compte de l'utilisateur.

2.2.1. Implémentation des tests

Le premier problème (l'implémentation du test) est étudié dans une de nos précédentes publications [BUI07b]. Le principe de notre approche est d'intégrer une interface de test dans le composant. Cette interface fournit des fonctionnalités de test et les cas de test.

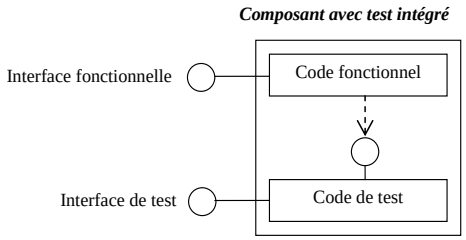


Figure 1. *Composant avec test intégré*

Un composant contient un ensemble d'interfaces fournies et requises. Chaque interface fournie correspond à un ensemble d'opérations que le composant met à disposition d'autres composants. Chaque interface requise est un ensemble d'opérations dont le composant a besoin pour son exécution. D'une manière similaire, les fonctionnalités de test sont d'autres services que le composant fournit à son environnement. Le composant testeur peut donc utiliser cette interface de la même façon que les autres interfaces fonctionnelles (figure 1).

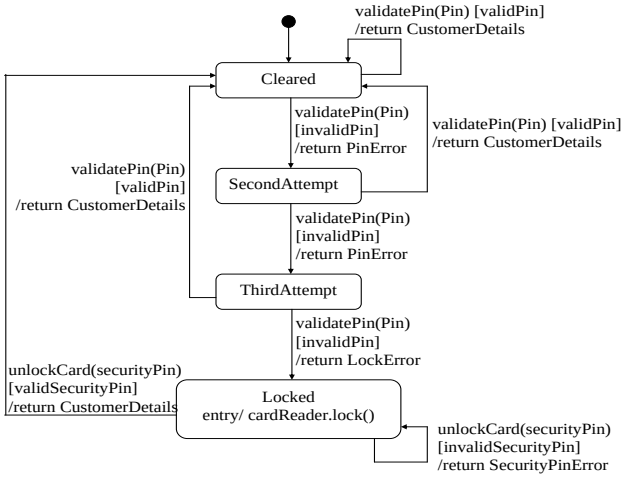


Figure 2. *Machine d'états du composant de validation de code*

Généralement, un composant peut être vu comme une machine d'état et exige le test de transition d'état. La machine d'états du composant de validation de code est illustrée par la figure 2. L'état "*cleared*" indique l'essai réussi, l'état "*locked*" indique que la carte bancaire est bloquée après trois essais incorrects d'introduction de code pin, et sera débloquée par le banquier. "*SecondAttempt*" et

"*ThirdAttempt*" indiquent que l'utilisateur essaie de fournir respectivement un deuxième et un troisième codes pin.

Avant l'exécution d'un test, le composant testé doit être amené à l'état initial requis pour ce test. Après l'exécution d'un test, le testeur doit vérifier que la sortie (si produite) est celle attendue et que le composant testé se trouve dans l'état final attendu. Pour atteindre ce but, nous utilisons une interface de test supplémentaire qui contient des opérations spéciales pour affecter et récupérer l'état interne du composant [GRO02], comme illustré par la Figure 3.

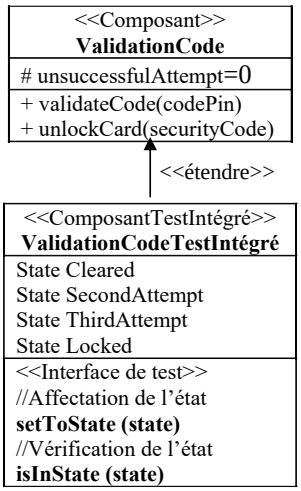


Figure 3. *Conception du composant avec interface de test*

Une interface de test étend la fonctionnalité normale du composant. Elle est exécutée comme une extension autonome du composant, pour que l'implémentation du logiciel de test soit encapsulée et clairement séparée de la fonction normale. Une interface de test comprend des opérations pour modifier et obtenir l'information de l'état interne. Ces opérations sont *setToState(state)* et *isInState(state)*. L'opération *isInState(state)* vérifie si le composant réside dans un état logique passé en paramètre. L'opération *setToState(state)* affecte les attributs internes des composants pour représenter un état distinct logique.

2.2.2. Concurrence et exactitude de l'état du système

La concurrence se produit quand un test et un autre composant appellent une fonction simultanément. Pour assurer l'exactitude du système, la fonctionnalité normale ne devrait pas être perturbée. Dans la littérature, différentes approches sont proposées [SUL06] :

- Le composant est bloqué pendant le processus de test. Au cours de l'exécution d'un test, les demandes de test ou de l'application sont retardées jusqu'à la fin du processus de test. Le problème d'exactitude du système peut être résolu ici en restaurant l'état original du composant après la fin du processus de test.
- Le composant abandonne le processus de test. Quand un autre composant demande le service normal du composant testé, le processus de test est abandonné. Une fois abandonné, l'état du composant doit être restauré. Le test doit être relancé à la fin de l'exécution de la fonctionnalité demandée et le composant testeur doit attendre la terminaison du test demandé.
- Le composant est cloné par l'infrastructure avant le démarrage du test. Le processus de test est alors exécuté sur le clone, et le processus de service est exécuté sur le composant original. Avec cette solution, le fonctionnement normal du système n'est pas perturbé. Cependant, cette option peut être très coûteuse en terme de consommation de ressources.
- L'interface de test de composant fournit des méthodes qui permettent des sessions de test. Ces méthodes garantissent que les données de test et les données réelles ne sont pas mélangées durant le processus de test. Cela pourrait aussi être fait par le clone (conduit par le composant lui-même) ou par des opérations spécifiques. Cette solution résout donc le problème de l'exactitude de l'état du système mais impose une charge supplémentaire pour le développeur de composants.

Il est important de noter qu'aucune solution n'est optimale et que le choix d'une solution plutôt qu'une autre dépend du contexte considéré. Dans le cadre de nos travaux, nous nous intéressons aux systèmes embarqués, pour lesquels les ressources sont limitées, mais sans tenir compte d'une gestion temps réel. C'est pourquoi, la première solution est probablement la plus adaptée. Si les composants ne doivent pas être bloqués, on optera pour la quatrième solution.

2.2.3. Efficacité

Pour améliorer l'efficacité du système, notre approche se base sur un ordonnanceur, qui équilibre l'exécution du test et le traitement de la fonctionnalité normale. Celui-ci emploie les stratégies d'exécution du test pour fournir un compromis approprié entre le test et la fonctionnalité normale de manière intelligente. Ces stratégies sont énumérées ci-dessous :

- Inactivité : si le composant testé est inactif, les tests seront exécutés.
- Seuil de ressources : les tests sont exécutés si la disponibilité de ressources dépasse un certain seuil pour toute ressource individuelle.

2.3. Architecture globale du service de diagnostic

Après l'étude des principaux problèmes soulevés, une architecture globale du service de diagnostic est proposée (figure 4).

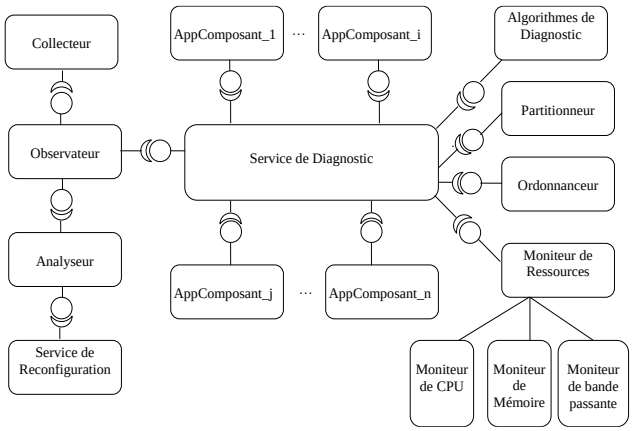


Figure 4. *Architecture du service de diagnostic*

Les composants principaux sont :

- AppComposant_i : sont des composants de l'application fournissant des fonctionnalités de test.
- Partitionneur : divise le système en groupes de diagnostic, où les tests inter-composants sont exécutés suivant une stratégie de test préétablie. L'introduction du partitionnement offre de meilleures performances par rapport aux approches sans partitionnement (qui couvrent le système entier) en réduisant le trafic de messages et le temps d'exécution [BAR93a].
- Algorithmes de Diagnostic : fournissent différents algorithmes de diagnostic, qui peuvent être choisis par l'application selon des critères précis.
- Moniteur de Ressources : observe l'utilisation des ressources du système (CPU, mémoire, bande passante, etc.) afin d'informer le processus de diagnostic des ressources disponibles dans le système.
- Ordonnanceur : il ordonnance les tests inter-composants par rapport aux ressources disponibles et la stratégie de test.
- Observateur : reçoit l'état global du groupe de diagnostic ou du système global.
- Collecteur : a la responsabilité de journaliser les erreurs, pour l'analyse statistique, le rapport d'état du système, etc. Ce composant est utile pour le partitionnement en groupes et le choix d'un algorithme de diagnostic approprié. Par exemple, nous pouvons grouper les composants souvent défaillants dans un groupe et utiliser un algorithme de diagnostic approprié de manière plus efficace, alors qu'un algorithme de diagnostic plus léger peut être utilisé pour un groupe moins défaillant. Ces informations statistiques peuvent aussi être utilisées pour déterminer la bonne période de diagnostic ou la durée du diagnostic.

- Analyseur : reçoit les informations transmises par l'observateur sur le composant défaillant. Ces informations, comme la journalisation des erreurs, ou encore les cas de tests, seront analysées pour déterminer le type de fautes produites par le composant et déclencher le processus de reconfiguration.
- Service de Reconfiguration : fournit différentes stratégies de reconfiguration du système, par exemple, la réparation en ligne, le remplacement de composants ou l'arrêt du système.

3. Procédure de diagnostic

La procédure de diagnostic se compose des étapes suivantes : le déclenchement du diagnostic (sous-section 3.1), l'exécution du diagnostic (sous-section 3.2) et la fin du diagnostic (sous-section 3.3).

3.1. Déclenchement du diagnostic

Le service de diagnostic peut être lancé à différents moments, en fonction d'une des conditions suivantes :

- Changement de la topologie du système : si la topologie change (des composants sont remplacés, enlevés ou ajoutés), le service de diagnostic est déclenché pour vérifier si le système fonctionne correctement.
- Périodicité : le service de diagnostic est exécuté périodiquement selon un instant prédéterminé.
- Demande de l'administrateur du système : l'administrateur peut exécuter le diagnostic pour connaître l'état global du système à n'importe quel instant.
- Contrainte de ressources : quand le système a suffisamment de ressources disponibles, le service de diagnostic peut être lancé.

Ces scénarios peuvent être choisis par l'utilisateur au moment de la configuration du système. Un ou plusieurs scénarios peuvent être combinés.

3.2. Exécution du diagnostic

Une fois déclenché, le service de diagnostic fonctionne comme illustré par la figure 5. Tout d'abord, le service de diagnostic utilise le composant de partitionnement pour diviser l'application en groupes afin de réduire la charge induite par les messages échangés durant le diagnostic et le temps d'exécution (1). Le service de diagnostic installe l'algorithme de diagnostic choisi par l'utilisateur pour chaque groupe ou pour le système entier (2). Le moniteur de ressources observe les ressources et envoie au service de diagnostic les informations sur les ressources disponibles (3). Ensuite, le service de diagnostic utilise l'ordonnanceur pour ordonnancer les tests inter-composants (4) en fonction de la stratégie de test et des ressources disponibles du système (et éventuellement du dernier résultat de test si l'algorithme de diagnostic est adaptatif). La demande de test est envoyée au composant applicatif (5). Ce composant testeur analyse cette demande de test pour savoir quel composant il testera avec quel cas de test.

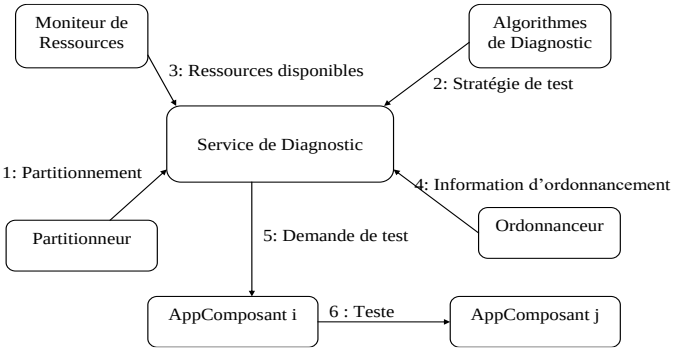


Figure 5. *Processus de diagnostic*

Pour le cas du lecteur de carte bancaire, les interactions entre les membres dans un groupe et le service de diagnostic sont détaillées dans la figure 6. Après réception d'une demande de test, le composant testeur teste des composants et transmet les résultats ou les informations de diagnostic aux composants qui sont définis par le composant de service selon une stratégie de test prédéterminée.

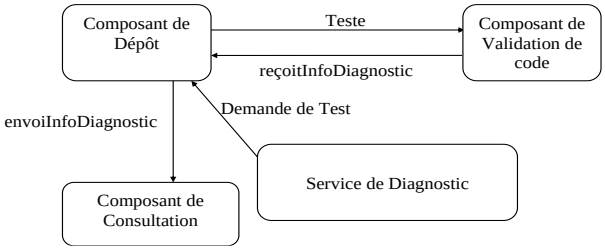


Figure 6. *Interaction entre le composant de diagnostic et les composants membres*

Le composant qui est responsable du processus de diagnostic et qui analyse les résultats des tests inter-composants maintient un tableau de résultats (appelé syndrome). Après analyse du syndrome, ce composant pourra déduire l'état global de son groupe.

3.3. Fin du diagnostic

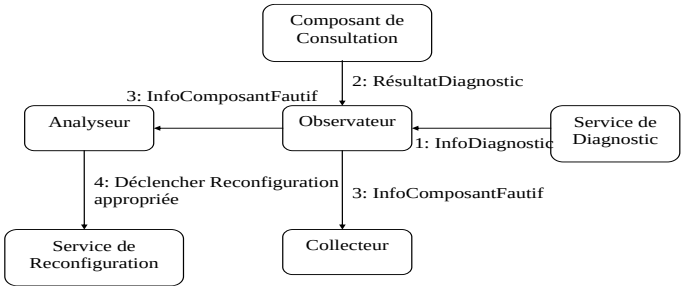


Figure 7. *Fin du service de diagnostic*

La fin du diagnostic est décrite par la figure 7. Le test dépend de l'algorithme de diagnostic choisi par l'utilisateur. Les composants qui assurent le diagnostic (en analysant les résultats des tests inter-composants) sont spécifiés et le service de diagnostic transmet leur identité à l'observateur (1). Après cela, l'observateur se connecte à ces composants pour recevoir l'état de faute du groupe et du système (2). Il informe l'analyseur et le collecteur des composants fautifs. Par exemple le composant de validation de code pin est défaillant avec les cas de test 6 (3). L'analyseur se sert de ces informations pour déclencher le service de reconfiguration (4).

4. Expérimentations

Notre proposition a été expérimentée sur différents exemples d'applications embarquées pour mesurer les surcoûts du diagnostic et analyser les performances relatives du diagnostic centralisé et du diagnostic distribué.

Outre l'exemple du lecteur de carte bancaire présenté ci-dessus, nous avons réalisé trois autres cas d'étude : un système de climatisation, un système automobile et un système générique. Dans ce papier, nous présentons les résultats obtenus pour le système générique. Ce système se compose de 20 composants génériques. Les résultats obtenus pour les autres exemples sont similaires et confirment les observations relatées ci-dessous. L'exemple générique est réalisé avec la technologie OSGi [OSGI] à l'aide de la plate-forme Oscar [OSCAR], celle-ci étant bien connue et très dynamique.

Pour les expérimentations, nous adoptons deux algorithmes de diagnostic de la littérature. Le premier est un algorithme de diagnostic distribué où chaque composant dans le système détermine l'état global du système [BIA91]. Le deuxième est un algorithme centralisé qui se base sur un composant central fiable pour déterminer l'état global du système [PRE67].

Critères	Surcoût de service de diagnostic	Surcoût dans chaque composant Dépend des cas de test (4 cas de test)
Nombre de lignes de code	~1000	~200
Taille des sources	43.1 kB	6.94 kB
Taille des classes compilées	20.9 kB	5.19 kB
Taille de bundle	12.2 kB	2.58 kB

Tableau 1. Surcoût du service de diagnostic

Le tableau 1 montre les résultats obtenus pour les mesures de la mémoire physique nécessaire pour le service de diagnostic. On peut conclure que le diagnostic n'est pas gourmand en mémoire.

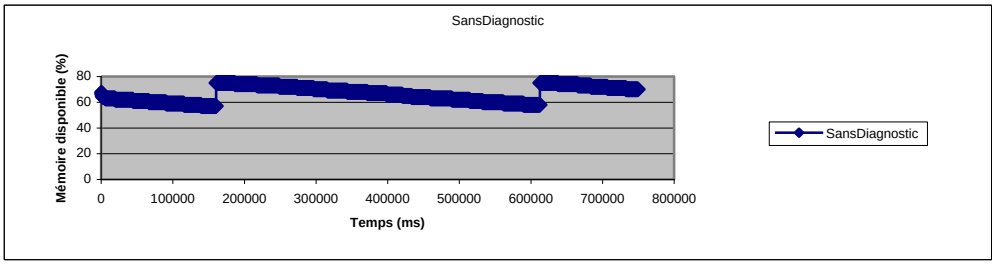
Critères	Diagnostic centralisé	Diagnostic distribué
Nombre de fautes (t)	$9 \geq t$	$19 \geq t$
Temps de diagnostic (ms)	10315	23574
Nombre de tests exécutés	O(n)	O(n)

Tableau 2. Diagnostic centralisé et diagnostic distribué

Le tableau 2 est la comparaison des résultats obtenus pour le diagnostic distribué et le diagnostic centralisé. Pour déterminer le nombre de fautes maximum (t), nous avons utilisé le résultat général présenté dans [BAR93b] qui indique que pour un système avec n composants :

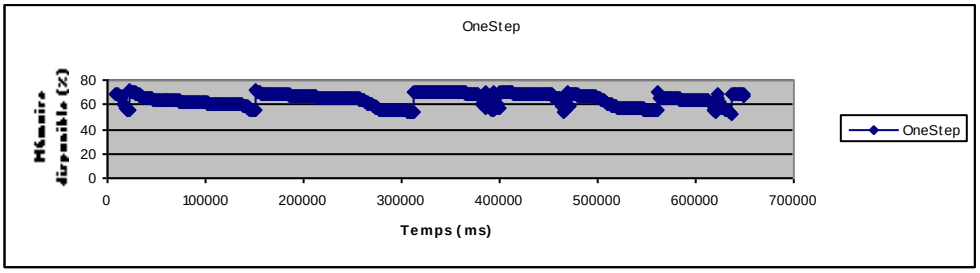
- pour l'approche de diagnostic centralisée, $n \geq 2t+1$,
- pour celle distribuée, $n \geq t+1$.

Avec cette formule, le système générique peut tolérer jusqu'à 19 composants fautifs avec l'algorithme de diagnostic distribué et 9 composants fautifs avec l'algorithme centralisé. Ces résultats montrent qu'il devrait y avoir un compromis dans l'utilisation des ressources entre les méthodes de diagnostic. La méthode de diagnostic distribuée peut tolérer plus de composants défaillants, alors que la méthode de diagnostic centralisé a besoin de moins de temps pour détecter les fautes.



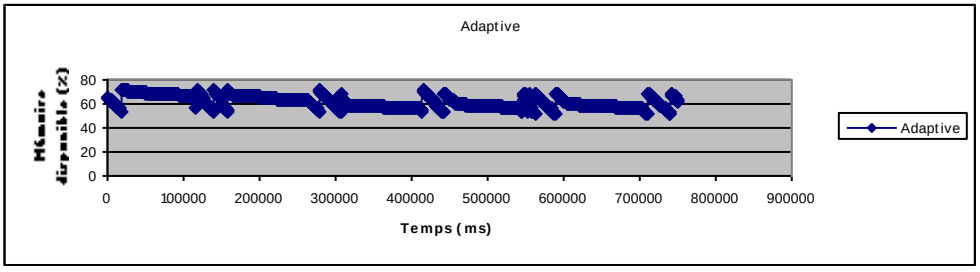

```
Min = 57
Max = 75
Moyenne = 66.436
Ecart type = 5.747
Variance = 33.027
```

(a) Système sans diagnostic




```
Min = 53
Max = 72
Moyenne = 63.98
Ecart type = 4.923
Variance = 24.223
```

(b) Système avec diagnostic centralisé




```
Min = 51
Max = 71
Moyenne = 61.384
Ecart type = 4.931
Variance = 24.315
```

(c) Système avec diagnostic distribué

Figure 8. *Mémoire disponible pour le système générique*

La figure 8 montre les mesures de la mémoire disponible de l'application sans diagnostic (8.a), avec diagnostic centralisé (8.b) et diagnostic distribué (8.c). La mémoire disponible est calculée comme étant la différence entre la mémoire utilisée par l'application et la mémoire totale dédiée à la Machine Virtuelle Java (~5M).

Les résultats obtenus, illustrés par la Figure 8, montrent que les surcoûts liés au service de diagnostic sont très faibles. En effet, la mémoire utilisée par le service est comprise entre 3 et 6 % de la mémoire totale.

5. Conclusion

Nous avons présenté les problèmes principaux liés au développement de l'architecture du service de diagnostic proposé et les solutions pour résoudre ces problèmes. Les approches de test inter-composant et de diagnostic proposées sont des fonctionnalités très intéressantes car elles peuvent améliorer la sûreté de fonctionnement d'applications avec un compromis compétitif entre les performances et les coûts. Les avantages du diagnostic sont l'autonomie de l'application du point de vue de la sûreté de fonctionnement, la réduction des coûts liés à la tolérance aux fautes et une meilleure utilisation des ressources du système. Ces aspects sont très importants dans beaucoup de systèmes et particulièrement dans les systèmes embarqués.

Nous travaillons actuellement sur l'implémentation de l'ordonnancement afin d'utiliser le temps d'inactivité et les ressources disponibles pour exécuter les tests. En parallèle, nous étudions des algorithmes de partitionnement efficaces afin de réduire le temps de détection de fautes et améliorer l'impact de la méthode de diagnostic sur les performances du système. Des résultats plus complets fourniront des indicateurs aux développeurs qui ont l'intention de construire un système tolérant aux fautes basé sur l'approche de diagnostic proposée. Finalement, une extension naturelle de notre travail est la tolérance aux fautes à base de diagnostic. Le processus de diagnostic sera alors suivi par une reconfiguration du système, afin d'assurer la fiabilité du système à long terme.

Références :

[BAR93a] Barborak, M., Makek, M. et Dahbura, A., "*The consensus Problem in Fault-Tolerant Computing",* ACM Computing Surveys, Vol.25, No.2, pp. 171-220, 1993.

[BAR93b] Barborak, M. et Malek, M., *"Partitioning for efficient consensus"*, IEEE Hawaii Intl. Conf. on System Technology Sciences, pp. 438-446, 1993.

[BIA91] Bianchini, R. et Buskens, R. W., *"An adaptative distributed system level diagnosis algorithm and its implementation"*, International IEEE Symp. on Fault-Tolerant Computing, IEEE CS Press, pp. 616-626, Montréal, Canada, 1991.

[BUI07a] Bui, T.Q. et Aktouf, O., *"Diagnosis service for embedded software component based systems"*, Intl. WS on Engineering Fault Tolerant Systems, pp. 14-19, Dubrovnik, Croatie, 2007.

[BUI07b] Bui, T. Q. et Aktouf, O., *"Inter-component testing for system-level diagnosis of embedded component based applications"*, Multidisciplinary Intl. Conf. on Quality and Reliability, pp. 246-254, Tanger, Maroc, Mars 2007.

[GRO02] Groß, H. G., *"Built-in Contract Testing in Component-based Application Engineering"*, CologNet Joint WS on Component-based Software Development and Implementation Technology for Computational Logic, Affiliated with LOPSTR, Madrid, Espagne, 2002.

[OCC05] Occello, A. et Dery-Pinna, A.-M., *"Sûreté de fonctionnement d'applications nomades construites par assemblage de composants"*, French-speaking conf. on Mobility and ubiquity computing, pp. 73–80, Grenoble, France, 2005.

[OSCAR] *Oscar*, DOI= http://www.oscar.objectweb.org.

[OSGI] *OSGi*, DOI= http://osgi.org.

[PRE67] Prerapata, F. P., Metz, G. et Chien, R. T., *"On the connection assignment problem of diagnosticable systems"*, IEEE Transactions on Electronic Computers, vol. EC-16, n°6, pp. 848-854, 1967.

[SOM05] Sommer, N. L. et Roussain, H., *"Une approche pour une continuité de service pour les utilisateurs de terminaux mobiles"*, French-speaking conf. on Mobility and ubiquity computing, pp. 81–88, Grenoble, France, 2005.

[SUL06] Suliman, D., Paech, B., Borner, L., *et al., "The MORABIT approach to runtime component testing",* Annual International Computer Software and Applications Conference, pp. 171-176, Chicago, 2006.