Adaptation en ligne de mécanismes de tolérance aux fautes par une approche à composants ouverts

Thomas Pareaud

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Institut National Polytechnique de Toulouse (INP-T)*
Discipline ou spécialité : *Systèmes Informatiques*

Présentée et soutenue par *Thomas Pareaud*[1]
Le *27 Janvier 2009*

Titre :

Adaptation en ligne de mécanismes de tolérance aux fautes par une approche à composants ouverts

---

JURY

Daniel Hagimont (Prof. INP-ENSEEIHT / IRIT)
Laurence Duchien (Prof. Univ. Lille / INRIA, Rapporteur)
Michel Raynal (Prof. Univ. Rennes / IRISA, Rapporteur)
François Taïani (MdC. Univ. Lancaster, Grande Bretagne)
Jean-Paul Blanquart (Ingénieur EADS-Astrium, Toulouse)
Marc-Olivier Killijian (CR CNRS, LAAS, Co-Directeur de thèse)
Jean-Charles Fabre (Prof. INP-ENSEEIHT / LAAS, Directeur de thèse)

École doctorale : *EDSYS*
Unité de recherche : *Laas-Cnrs, Groupe TSF*
Directeur(s) de Thèse : *Jean-Charles Fabre, Marc-Olivier Killijian*

---

[1] Cette thèse a été financée par une bourse DGA-CNRS ainsi que le réseau d'excellence européen ReSIST (Resilience for Survivability in IST).

AVANT-PROPOS

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS). Je remercie Messieurs Malik Ghallab et Raja Chatila, qui ont assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire.

Je souhaite grandement remercier la Délégation Générale de l'Armement (DGA) et le Réseau d'Excellence Européen Resilience for Survivability in IST (ReSIST) pour avoir financé ces travaux de thèses. Je remercie en particulier Messieurs Sarrazin et Blanc-Talon, responsables du domaine scientifique « Ingénierie de l'Information » de la Mission pour la Recherche et l'Innovation Scientifique (DGA/MRIS) qui ont su porter un regard extérieur positif sur ces travaux.

Je remercie également Monsieur Jean Arlat et Madame Karama Kanoun, Directeurs de Recherche CNRS, responsables successifs du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe, et pour travailler d'arrache pied à fournir un environnement propice au travail et aux échanges tous deux nécessaires pour effectuer une thèse dans les meilleures conditions.

J'exprime ma très sincère reconnaissance à Messieurs Jean-Charles Fabres, Directeur CNRS et Professeur à l'Institut National Polytechnique de Toulouse, et Marc-Olivier Killijian, Chargé de Recherche CNRS, pour m'avoir encadré, soutenu et encouragé tout au long de cette thèse. Ces travaux sont le fruit de débats et discutions souvent enflammées que nous avons eu ensemble. Ils ont su canaliser mes ardeurs, me permettant de prendre du recul sur les travaux présentés ici. Ils ont su rendre inoubliables les années passées en leur compagnie. Pour tout cela, je les en remercie.

Je remercie Monsieur Daniel Hagimont, Professeur à l'Institut National Polytechnique de Toulouse, pour l'honneur qu'il me fait en présidant mon jury de thèse, ainsi que :

- Madame Laurence Duchien, Professeur à l'Université de Lille
- Monsieur Michel Raynal, Professeur à l'Université de Rennes1
- Monsieur François Taïani, Professeur Assistant à l'Université de Lancaster
- Monsieur Jean-Paul Blanquart, Ingénieur à EADS-ASTRIUM et Responsable d'études avancées en sûreté de fonctionnement et autonomie des systèmes spatiaux

pour l'honneur qu'ils me font en participant à mon jury de thèse. Je remercie particulièrement Madame Laurence Duchien et Monsieur Michel Raynal qui ont accepté la charge d'être rapporteurs, et pour leurs encouragements qu'ils m'ont fait.

Je tiens à remercier Thomas Robert (BoB) et Nicolas Salatge pour leur amitié et leur soutien. Thomas a su partager avec moi de longues et fréquentes pauses café propices aux formalisations les plus importantes de ces travaux de thèse. Nicolas a su montrer de l'intérêt à mes travaux, et a su garder ma motivation intacte en dépit des difficultés rencontrées.

Je remercie également l'ensemble des membres du groupe TSF, permanents, doctorants, post-doctorants et stagiaires pour leur constante disponibilité et leur gentillesse. Je pense tout particulièrement à Yves Crouzet pour tous les services (informatiques et multimédia) qu'il a grandement contribués à mettre en place au LAAS-CNRS.

Je souhaite remercier ceux qui sont toujours à mes côtés. Je veux bien entendu parler de ma famille. Je remercie, ma femme, Lydie, mes parents, et mes beaux-parents, pour être toujours à mes côtés, pour m'apporter leur soutien, leur confiance.

Enfin, je souhaite remercier toutes les personnes de l'ombre qui contribuent activement à la vie de laboratoire, et donc à la réussite de tout un chacun. Je veux bien entendu parler des membres du service informatique, du service du personnel, du service financier, du personnel de la cantine, et tous les autres services sans lesquels le LAAS-CNRS ne pourrait fonctionner.

SOMMAIRE

INTRODUCTION GENERALE

La prise en compte de l'évolution des systèmes en fonctionnement du point de vue de leurs mécanismes de sûreté est un impératif dans de nombreux domaines applicatifs, qu'ils soient critiques ou non. Cela concerne tout particulièrement les systèmes embarqués critiques mais aussi l'informatique ambiante, les systèmes autonomes ou encore les systèmes mobiles. Ces systèmes évoluent dans un contexte opérationnel constitué de l'environnement, des ressources et de besoins utilisateurs qui vont changer au cours du temps. Il existe deux principales raisons pour qu'un système adapte ses mécanismes de tolérance aux fautes au contexte opérationnel. La première concerne l'évolution du modèle de faute : des modèles de fautes différents doivent être pris en compte selon le contexte opérationnel courant du système (nouveaux besoins utilisateurs, environnement plus agressif, changement de phase d'une mission pour une phase plus ou moins critique que la précédente). La deuxième concerne l'utilisation des ressources du système : la tolérance aux fautes consomme des ressources du système en l'absence de faute, donc il est possible de devoir dégrader la tolérance aux fautes pour assurer la survie du système et prévenir une pénurie fatale des ressources.

De façon générale, l'adaptabilité des systèmes est une propriété incontournable pour des raisons d'évolutivité et de flexibilité. Cette problématique a souvent été étudiée au travers de méthodes architecturales ou de génie logiciel, permettant de faciliter l'adaptation des systèmes hors-ligne. L'adaptation en-ligne permet idéalement de pratiquer des modifications en minimisant le temps d'interruption du service. Les nouvelles technologies à composant permettent la modification en-ligne du logiciel conçu sous la forme d'un graphe de composants. Au-delà de l'aspect mécanique de l'assemblage et de désassemblage de composants, la prise en compte de la dynamique lors de l'adaptation est essentielle pour maintenir une certaine cohérence des modifications effectuées.

Dans cette thèse, nous nous intéressons à l'adaptation en-ligne du logiciel dans un système critique qui est donc équipé de mécanismes assurant des propriétés de sûreté de fonctionnement. Nous illustrons ici les besoins d'adaptation d'un système par un exemple concret : une sonde spatiale. La construction d'un engin spatial est soumise à deux contraintes : le poids et l'énergie. Plus l'engin est lourd, plus il est difficile de l'envoyer dans l'espace et de l'y faire voyager. Une sonde spatiale possède des ressources informatiques et énergétiques embarquées limitées qui sont dimensionnées dans l'objectif de réaliser une mission. Cette mission est découpée en phases pendant lesquelles le système doit fournir certains services sous certaines contraintes de performances et de sûreté de fonctionnement. Considérons deux phases : le voyage de la Terre à la planète cible et la phase de mise en orbite autour de la planète cible. Le modèle de faute considéré n'est alors pas identique pendant ces deux phases. Lorsque la sonde est en transit vers sa destination, les durées sont relativement longues, ce qui offre beaucoup de temps pour traiter une erreur. De plus, pendant cette phase, les concepteurs souhaitent minimiser les ressources énergétiques consommées. Ainsi, les erreurs peuvent être traitées en utilisant des resets matériels du système. En phase d'approche, ces contraintes sont radicalement différentes. Une défaillance du système met en échec la mission de la sonde. De plus, les temps disponibles pour traiter les erreurs sont beaucoup plus faibles. Une implémentation répliquée de certains services peut s'avérer nécessaire.

Les drones militaires sont un autre exemple nécessitant l'adaptation. En effet, ces drones évoluent dans le but de réaliser une mission. Toutefois, leur contexte opérationnel peut éventuellement changer en cours de mission. Par exemple, un drone de reconnaissance peut être attaqué pendant sa mission. Dans le contexte nominal de sa mission (i.e. en dehors d'agression), la sûreté de fonctionnement des appareils de vol, de mesures, d'observations et de communications prime sur celle de l'armement. Les ressources du système sont alors employées à garantir la tolérance aux fautes de ces services. Lorsque le drone entre dans un

environnement hostile, la seule défense permettant sa survie est l'affrontement. L'armement et le vol sont alors les services les plus critiques du système. Il peut alors être nécessaire de récupérer des ressources utilisées pour la tolérance aux fautes des services moins critiques afin de renforcer celle des services les plus critiques, et donc de renforcer la capacité de survie du système. Le changement de contexte opérationnel nécessite donc une adaptation en ligne de la configuration du logiciel de tolérance aux fautes du système.

Dans ces exemples, les besoins en termes de tolérance aux fautes évoluent au cours de la vie opérationnelle du système. Ainsi, la tolérance aux fautes doit donc être *adaptée au contexte opérationnel* du système. L'adaptation possède deux dimensions. La première est de déterminer l'instant à partir duquel la configuration du système doit être modifiée pour répondre à des exigences non-fonctionnelles ainsi que la nouvelle configuration à atteindre pour respecter ces exigences. Cette dimension est appelée *décision pour l'adaptation*. La deuxième dimension est de modifier le logiciel qui compose le système pendant son fonctionnement, afin de le placer dans une configuration souhaitée. Cette dimension est appelée *adaptation du logiciel*.

Les travaux originaux présentés dans ce manuscrit traitent de l'*adaptation du logiciel* de tolérance aux fautes d'un système critique. Ces travaux proposent de coupler les modèles à composants et la réflexivité pour séparer la tolérance aux fautes du logiciel fonctionnel et réaliser l'adaptation du logiciel de tolérance aux fautes au cours de l'exécution du système.

Le chapitre I montre l'intérêt de l'adaptation pour la survie des systèmes, et brosse un éventail des travaux réalisés dans le domaine de la tolérance aux fautes vis-à-vis de la modification du logiciel. Nous montrerons que les approches existantes manquent de flexibilité en termes des possibilités de stratégies de tolérance aux fautes qui peuvent être utilisées. Ainsi, nos travaux se focaliseront sur l'amélioration de cet aspect en montrant comment laisser le concepteur du système libre de choisir et de concevoir les mécanismes de tolérance aux fautes nécessaires à son système.

Le chapitre II présente une approche et des techniques permettant de traiter la problématique de l'adaptation du logiciel de tolérance aux fautes. L'approche utilisée repose sur l'utilisation d'une architecture réflexive permettant de séparer les fonctionnalités, la tolérance aux fautes et l'adaptation dans des couches différentes du système. Cette séparation est nécessaire pour rendre indépendant la tolérance aux fautes et les traitements de son adaptation. De plus, nous montrerons comment les modèles à composant peuvent être utilisés pour permettre la modification du logiciel pendant l'exécution.

Dans le chapitre III, nous détaillerons les modèles réflexifs utilisés pendant l'adaptation. Nous introduirons une représentation du comportement du système du point de vue de son exécution. Nous montrerons comment définir les états permettant l'adaptation à partir de cette modélisation et nous en déduirons des politiques de contrôle de l'exécution pour permettre de faire converger le système et le maintenir dans un état permettant l'adaptation. L'utilisation de ces modèles s'inscrit dans un processus de développement du logiciel pour son adaptation.

Nous appliquons ces principes de conception dans le chapitre IV au logiciel de tolérance aux fautes, appliqué à un système de contrôle-commande automatique. Nous présenterons les outils mis en place pour aider à la conception du logiciel, afin d'automatiser le processus d'adaptation. Ce manuscrit se terminera par une conclusion générale sur ces travaux et une analyse des efforts à réaliser dans le domaine de l'adaptation.

Chapitre I

SURETE DE FONCTIONNEMENT, TOLERANCE AUX FAUTES ET ADAPTATION

I.1 INTRODUCTION

Les systèmes critiques répondent à des domaines d'utilisation très variés comme les transports, l'exploration spatiale ou encore les systèmes bancaires. Ils possèdent des durées opérationnelles variables pouvant aller de quelques heures pour un trajet en voiture à plusieurs années pour l'exploration spatiale. Ils sont de plus en plus autonomes dans le sens où ils ne nécessitent pas l'intervention d'êtres humains pour fonctionner, de plus en plus mobiles et de plus en plus communicants.

La conception de ces systèmes est souvent réalisée sous l'hypothèse simplificatrice d'un environnement connu a priori dans son intégralité et donc considéré comme constant au cours de la vie opérationnelle. De notre point de vue, lorsque l'environnement est complexe et possède une grande disparité, il est plus facile de morceler cet environnement en sous-environnements plus simples, pour lesquels des solutions connues peuvent s'appliquer, et de décrire comment ces sous-environnements s'agencent au cours de la vie opérationnelle du système. Ces sous-environnements et les solutions appliquées forment les contextes opérationnels du système. Il est alors nécessaire de pouvoir modifier le système pendant sa vie opérationnelle lorsque le contexte change. C'est ce que l'on appelle l'adaptabilité d'un système à son contexte opérationnel, i.e. la capacité à répondre à un changement de contexte opérationnel en modifiant les services, la manière dont ils sont rendus et les propriétés assurées pendant la délivrance de ce service.

Dans un système autonome, l'adaptation du matériel est souvent très limitée dans le sens où il est « impossible » de créer un nouveau composant matériel. Elle s'applique surtout à la modification d'algorithmes (protocoles) codés dans des FPGAs et n'en demeure pas moins compliquée. Ces travaux se focalise sur la partie logicielle du système.

L'adaptation du logiciel d'un système est souvent appliquée à ses fonctionnalités. Un système peut développer des fonctionnalités en fonction de l'environnement dans lequel il évolue. Ceci est particulièrement sensé dans le domaine de la domotique. Ainsi, à la maison, un simple téléphone portable peut se transformer en une commande pour les lumières ou la télévision, alors qu'au travail, celui-ci se transforme en un badge d'accès ou en billet électronique dans un aéroport. En ce qui nous concerne, nous voyons dans l'adaptation du logiciel une opportunité pour sa résilience et sa survie. Il est facile de voir les bénéfices de l'adaptation dans les systèmes phasés critiques fortement contraints en termes de ressources sur sa tolérance aux fautes. Ainsi, chaque phase peut être considérée comme un contexte. Elle peut posséder des contraintes ou objectifs différents, nécessitant alors l'utilisation de stratégies de tolérance aux fautes propres à chaque phase.

Pour faire la lumière sur les bénéfices d'une tolérance aux fautes adaptables, nous introduisons tout d'abord quelques notions de sûreté de fonctionnement, et plus précisément de tolérance aux fautes, puisqu'il s'agit de l'aspect ciblé par l'adaptation. Nous reviendrons ensuite sur la notion d'adaptation, et notamment sur la manière dont elle s'applique dans un système critique tolérant les fautes. Nous nous intéresserons à caractériser l'adaptation, et déterminer les opportunités et les manques dans ce domaine. Ces deux aspects nous amèneront à préciser les termes de notre problématique en termes d'adaptation de la tolérance aux fautes. La problématique de l'adaptation du logiciel est un sujet de recherche très actuel, celui de l'adaptation des mécanismes de tolérance aux fautes en ligne est plus original.

I.2 SURETE DE FONCTIONNEMENT ET TOLERANCE AUX FAUTES

Ces travaux s'intéressent aux systèmes informatiques. Bien que la sûreté de fonctionnement ne soit pas spécifique à ce domaine, nous nous focaliserons à son application aux systèmes informatiques.

La sûreté de fonctionnement des systèmes vise à pouvoir placer une confiance justifiée dans le service qu'ils délivrent [1]. La sûreté de fonctionnement s'intéresse entre autre à améliorer la survivance des systèmes, c'est-à-dire leur capacité à continuer de fonctionner pendant et après une perturbation rencontrée durant leur vie opérationnelle. Cette perturbation peut provenir de l'environnement du système (ce qui lui est extérieur) ou du système lui-même (ressources, fautes, ce qui lui est propre). De manière générale, nous ne distinguerons pas ces deux sources de perturbation et nous parlerons du contexte opérationnel du système.

La sûreté de fonctionnement met à disposition quatre moyens pour assurer cette confiance :

- La *prévention des fautes* est intégrée au développement du système. L'objectif est d'éviter l'introduction de fautes de développement tant au niveau logiciel qu'au niveau matériel.
- La *suppression des fautes* consiste à vérifier la présence de fautes lors du processus de développement et à réaliser de la maintenance corrective et préventive sur le système en cours d'utilisation.
- L'*estimation des fautes* est réalisée par une évaluation du comportement du système en présence de faute.
- La *tolérance aux fautes* est intégrée au système et agit en ligne en détectant et corrigeant les erreurs du système.

L'adaptation est un processus qui a lieu pendant le fonctionnement du système. Pour améliorer la sûreté de fonctionnement du système, nous proposons d'appliquer cette technologie à l'unique moyen que la sûreté de fonctionnement met en place durant la vie opérationnelle du système : la tolérance aux fautes. Nous proposons ici de détailler ce qu'est la tolérance aux fautes afin d'appréhender ce qui peut être adapté.

I.2.1 Chaîne de causalité fautes, erreurs et défaillances

Un système défaille lorsque le service qu'il délivre diffère du service attendu. La tolérance aux fautes cherche à éviter ces défaillances, ou au moins, à prévenir les plus catastrophiques. Ainsi, on préfèrera arrêter un train sur les rails, retardant ainsi l'arrivée des passagers, que leur faire risquer une collision.

Pour empêcher l'occurrence de défaillances, la sûreté de fonctionnement s'intéresse à leurs causes. Elle introduit deux notions : les fautes et les erreurs. Une faute est la cause d'une défaillance. La notion de faute est très subjective. Ainsi, si un système soumis à des perturbations magnétiques défaille, la faute est-elle d'origine matérielle dans le sens où le matériel n'a pas toléré les perturbations, ou d'origine conceptuelle, dans le sens où le concepteur du système n'a pas dimensionné le blindage correctement ? L'activation d'une faute du système provoque la propagation d'erreurs dans le système. Une défaillance a lieu lorsqu'une erreur est observable à la frontière du système. Faute, erreur et défaillance forment donc une chaîne de causalité.

Dans un souci de gestion de la complexité, les systèmes sont souvent appréhendés comme une composition de sous-systèmes appelés composants. Ainsi, un système est le résultat de la composition de plusieurs composants qui sont eux-mêmes des systèmes. Ainsi, pour un certain niveau d'abstraction (celui du système considéré), les composants sont les entités élémentaires. Leur défaillance est alors une source d'erreur pour le système, dans le sens où elle peut occasionner un comportement anormal du système : il s'agit donc d'une faute pour le système. La chaîne faute, erreur, défaillance devient alors récursive.

La tolérance aux fautes a pour objectif d'empêcher la défaillance du système, ou du moins maîtriser ses conséquences. Dans ce but, la tolérance aux fautes cherche à casser la chaîne de causalité, afin de protéger l'utilisateur final du système d'une défaillance. Cette protection peut prendre plusieurs formes selon que l'on cherche à masquer la faute à l'utilisateur, ou le protéger en plaçant le système dans un état sûr (train à l'arrêt par exemple). Pour réaliser cette protection, la tolérance aux fautes utilise un certain nombre de moyens (cf. section I.2.3) implémentés sous la forme de mécanismes de tolérance aux fautes (section I.2.4). Une *stratégie de tolérance aux fautes* est une combinaison de ces moyens permettant de couvrir le modèle de fautes souhaité, avec les performances (temps de recouvrement, indisponibilité) et les propriétés (disponibilité, fiabilité, robustesse, testabilité, maintenabilité, etc.) désirées.

I.2.2 Modèles de fautes

Le modèle de fautes d'un système définit la nature des fautes présentes dans le système. Il s'agit d'une étape préliminaire à la définition des mécanismes de tolérance aux fautes à mettre en place dans le système. En effet, la tolérance aux fautes devra permettre de tolérer les fautes définies dans le modèle de fautes en utilisant de la redondance et de la diversité (cf. I.2.3). Ainsi, si une faute présente dans le système n'est pas définie dans le modèle de faute, la tolérance aux fautes ne permettra pas de la tolérer. La notion de faute est alors récursive : la définition d'un modèle de faute qui diffère de la réalité est une faute de conception de la tolérance aux fautes du système. L'exemple emblématique est celui d'Ariane 5.

Les fautes peuvent être classées selon différents critères : le moment où elles apparaissent dans le cycle de vie du système (faute de conception, faute opérationnelle), leur origine par rapport aux frontières du système (faute interne, faute externe), leur origine phénoménologique (faute naturelle, faute humaine), de leur dimension (faute logicielle, faute matérielle), de leur objectif (faute malicieuse, faute non-malicieuse), de leur intention (faute intentionnelle, faute non-intentionnelle), de la compétence des utilisateurs (faute d'incompétence, faute accidentelle) ou de leur persistance (faute persistante, faute transitoire). Chacune de ces classes de fautes nécessite l'introduction de moyens palliatifs appropriés. Par exemple, les fautes d'incompétence peuvent être évitées par une formation adéquate des utilisateurs, mais aussi une conception ergonomique et des vérifications de la cohérence des ordres des utilisateurs.

Dans un système compositionnel, les fautes considérées sont les défaillances des composants. Ces défaillances peuvent être classées en quatre modes :

- *La défaillance par crash :* le composant cesse toute interaction avec les autres composants du système. La notion d'interaction dépend du modèle de système considéré (appel de procédure, envoi de message, écriture dans une mémoire partagée) ;
- *La défaillance temporelle :* le composant interagit avec les autres composants du système en dehors des fenêtres temporelles attendues. Cela concerne aussi bien des interactions qui ont lieu trop tard (échéance manquée) ou trop tôt. La défaillance par crash est une

défaillance temporelle, dans le sens où toutes les interactions seront réalisées trop tard (jamais en l'occurrence).

- *La défaillance en valeur :* le composant interagit avec les autres composants du système en utilisant des valeurs incorrectes.
- *La défaillance byzantine :* le composant défaille de manière arbitraire. Ce mode de défaillance regroupe la défaillance par crash, la défaillance temporelle, la défaillance en valeur, et toute combinaison de ces modes de défaillance (valeur erronée trop tard par exemple).

Lorsqu'un composant ne défaille que par crash, il est dit « à silence sur défaillance ». Cette hypothèse est l'une des plus fréquemment admise. Toutefois, elle n'est que rarement vérifiée.

I.2.3 Redondance et diversité

La tolérance aux fautes repose principalement sur deux principes : la redondance et la diversité. La redondance consiste à multiplier les sources d'informations du système. Elle peut être de plusieurs natures : redondance des données (code correcteur d'erreur, enregistrement sur plusieurs supports) et redondance d'exécution (exécution multiple, plusieurs instances du même service). La diversité consiste à s'assurer de l'indépendance des défaillances des services redondées. La diversité joue un rôle essentiel pour tolérer les fautes de mode commun qui existent lors de l'utilisation de répliques identiques.

La réplication active est un exemple typique de l'utilisation de la redondance pour masquer les défaillances en valeurs. Plusieurs instances déterministes du même service (plus de trois) évoluent en parallèle. Les mêmes entrées leur sont fournis, et les mêmes résultats sont attendus. Ainsi, il est possible de masquer une faute transitoire de ce service amenant à une défaillance en valeur en comparant les résultats de chacune de ces instances. Si toutes les instances fournissent le même résultat, il n'y a pas de faute transitoire. Si une minorité d'instances diffèrent de la majorité, elles ont alors défailli. Toutefois, la réplication active ne suffit pas si le modèle de faute considère les fautes persistantes, car dans ce cas, même la majorité des répliques peuvent avoir tort. Ainsi, l'utilisation de la diversification [2] permet de se prémunir des fautes de mode commun. En introduisant une diversité dans la réplication active, la stratégie de tolérance aux fautes permet de masquer les fautes transitoires mais aussi les fautes logicielles persistantes d'origine conceptuelle amenant à une défaillance en valeur.

Redondance et diversité ne se limitent pas uniquement au logiciel. Ces deux principes s'appliquent à l'intégralité du système, matériel inclus.

I.2.4 Mécanismes de tolérance aux fautes

Cette section n'a pas pour objectif d'énumérer tous les mécanismes de tolérance aux fautes existants. Cela est impossible du fait que souvent la tolérance aux fautes est étroitement liée au service auquel elle s'applique. L'objectif est de comprendre quels sont les constituants de la tolérance aux fautes pour, d'une part comprendre pourquoi l'adaptation est une opportunité, et d'autre part, comment la tolérance peut être modifiée à l'exécution. La tolérance aux fautes s'appliquant au système dans son ensemble, la notion de composant s'applique aussi bien au logiciel qu'au matériel.

Pour tolérer le modèle de faute d'un système, il est nécessaire de mettre en place une stratégie. La notion de stratégie est une notion de haut niveau : une stratégie consiste à décider comment mettre à profit la redondance et la diversité pour tolérer le modèle de fautes désiré.

L'implémentation d'une stratégie de tolérance aux fautes s'articule autour de mécanismes élémentaires de la tolérance aux fautes. Ces mécanismes sont de deux ordres :

- Les *mécanismes de détection d'erreurs* : ils permettent de détecter la présence d'erreurs dans le système après activation d'une faute. Il existe deux catégories de mécanismes de détection d'erreurs :
  - La *détection préventive* : elle a lieu pendant des phases particulières de l'activité du système, comme les phases de démarrage ou de maintenance par exemple. Couramment, elle consiste à vérifier des propriétés à l'aide de test. Par exemple, au démarrage d'une voiture, le calculateur subit une batterie de tests permettant de vérifier son fonctionnement vis-à-vis d'une certain modèle de faute (potentiel des branches du processeur, temps de réponse, etc.) ;
  - La *détection concurrente* : elle a lieu pendant l'exécution normale du système. Elle a pour objectif de détecter la présence d'une erreur afin d'y remédier. Dans un système compositionnel, les erreurs détectées sont le plus souvent les défaillances des composants constitutifs du système.
- Les *mécanismes de recouvrement d'erreur* : ils transforment l'état du système qui contient des erreurs et des fautes dans un état où aucune erreur n'est détectée, et où les fautes ne peuvent être réactivées. Ces mécanismes se classent en deux catégories :
  - Les *mécanismes traitant les erreurs* qui éliminent les erreurs de l'état du système ;
  - Les *mécanismes traitant les fautes* qui empêchent les fautes d'être réactivées.

Il y a trois manières d'éliminer les erreurs de l'état d'un système :

- La reprise (*rollback*) consiste à ramener le système dans un état précédent l'erreur afin qu'il continue son fonctionnement. Par exemple, on redémarre son logiciel de traitement de texte qui s'est arrêté inopinément et on ouvre le dernier enregistrement de son document avant la défaillance du logiciel.
- La poursuite (*rollforward*) consiste à placer le système dans un état sûr, exempt d'erreur connue, à partir duquel le système peut reprendre son service. Par exemple, le train est mis à l'arrêt.
- La compensation consiste à utiliser la redondance de l'état pour masquer l'occurrence d'erreur. Par exemple, le RAID [3] compense les erreurs en utilisant la redondance des données.

Trois mécanismes permettent d'empêcher la faute d'être réactivée :

- Le *diagnostic* cherche à caractériser la faute qui a amené l'erreur dans le système en termes de localisation (quels composants sont à l'origine de l'erreur), de nature (transitoire ou permanente), etc.
- L'*isolation* permet d'empêcher un composant contenant une faute persistante d'être de nouveau sollicité par le système en réalisant une exclusion logique ou matérielle de ce composant pour empêcher sa participation future dans la délivrance du service. En d'autres termes, la faute devient dormante.
- La *reconfiguration* consiste à changer le composant fournissant le service, ou à réassigner les tâches sur d'autres composants.

– La *réinitialisation* vérifie et met à jour les tables du système pour tenir compte de la reconfiguration effectuée. Cette mise à jour peut aller jusqu'au redémarrage du système (reset).

On remarque que la tolérance aux fautes introduit une notion relativement proche de l'adaptation en termes de problématique : la reconfiguration. Les mécanismes de reconfiguration et de réinitialisation permettent la mise en place de modes dégradés du fonctionnel. Un mode dégradé est un mode dans lequel le système fournit un service partiel à la fois en termes de fonctionnalités (nombre de services diminué), mais aussi en termes de propriétés (implémentation plus simple, moins propice à contenir des fautes, mais fournissant un service de moins bonne qualité).

La reconfiguration réalisée par le traitement de la faute diffère donc de l'adaptation par deux aspects. Tout d'abord, la reconfiguration est un mécanisme de traitement de la faute qui modifie le fonctionnel pour répondre à certaines fautes diagnostiquées, alors que l'adaptation vise à modifier le système (logiciel fonctionnel ou non-fonctionnel) pour répondre à des changements de contexte opérationnel (le système dans son environnement). Ensuite, la reconfiguration est un processus « irréversible » en tolérance aux fautes, dans le sens où le système ne fait que dégrader le service. Au contraire, l'adaptation peut mettre en place un service « mis à jour » dans le sens où plus de fonctionnalités/propriétés peuvent être fournies dans certains contextes opérationnels. Par exemple, dans les systèmes ouverts, lorsque de nouvelles ressources sont découvertes, le nombre de répliques peut être augmenté.

Il est intéressant de voir comment les notions de redondance et de diversité sont utilisées pour mettre en place les mécanismes de tolérance aux fautes. Une redondance d'exécution permet de faire de la compensation. Lorsque les instances redondées sont identiques, une comparaison des résultats de chaque instance permet de détecter une erreur transitoire. Si elles sont diversifiées, une comparaison des résultats de chaque instance permet de détecter des erreurs permanentes liées à des fautes de conception/développement.

Ainsi, l'implémentation d'une stratégie de tolérance aux fautes se compose d'un ensemble de mécanismes de tolérance aux fautes, qui, utilisés conjointement, permettent de couvrir le modèle de fautes spécifié en obtenant les propriétés voulues. Il existe deux grandes familles de stratégies de tolérance aux fautes, selon les mécanismes de recouvrement utilisés, et la manière dont ils sont agencés :

– Le masquage et recouvrement où l'erreur est naturellement compensée par la redondance d'état. La détection d'erreur est réalisée indépendamment de la compensation et provoque le traitement de la faute. C'est le cas par exemple des mécanismes de réplication active ;
– La détection et recouvrement qui une fois l'erreur détectée, traitent l'erreur puis la faute. Le traitement d'erreur peut être indifféremment réalisé par :
    - Reprise, on parle alors de recouvrement arrière dont l'exemple emblématique est la stratégie *Primary Backup Replication* ;
    - Poursuite qui donne lieu à un recouvrement avant partiel. Le mécanisme de traitement de l'erreur est très dépendant de l'applicatif car il nécessite de connaître un état sûr ou une manière d'en reconstruire un ;
    - Compensation qui donne lieu à un recouvrement avant total, ce qui est le cas des stratégies *Leader Follower Replication* et *Triple Modular Replication*.

I.2.5 Distribution et tolérance aux fautes

La réplication consiste à redonder des services en utilisant plusieurs instances du même service appelées les répliques. La distribution permet de rendre indépendant les modes de défaillance de ces répliques. Par exemple, la distribution permet de localiser les répliques dans des lieux géographiques différents, rendant alors indépendantes les défaillances liées à des fautes d'origine naturel (incendie, tremblement de terre, inondation).

Distribuer des répliques consiste à les faire s'exécuter sur des nœuds différents du système. Afin de pouvoir recouvrer une erreur, ces exécutions doivent être coordonnées. Un protocole de synchronisation permet d'assurer cette propriété. Ce protocole repose d'une part sur un médium de communication, et donc de ses propriétés intrinsèques, mais aussi de propriétés intrinsèques de la réplique exécutée (déterminisme, granularité, robustesse). Les architectures rencontrées vont ainsi d'une architecture à base de processeurs synchronisés sur la même horloge dont les sorties sont comparées, à des systèmes utilisant des réseaux locaux à jetons dont l'exemple emblématique est Delta-4 [4], en passant par les liaisons point à point rencontrées par exemple dans l'avionique civile.

Ainsi, ces architectures peuvent être regroupées en deux catégories : les architectures distribuées synchrones et asynchrones. Dans une architecture synchrone, tout échange de message peut être borné dans le temps. Inversement, dans une architecture asynchrone, aucune hypothèse ne peut être réalisée. Ainsi, il est impossible de déterminer un message perdu, d'un message qui met beaucoup de temps à être reçu. Autrement dit, il est impossible dans un système asynchrone de différencier un processus lent d'un processus qui a défailli par arrêt [5]. Toutefois, contrairement aux architectures synchrones, les architectures asynchrones sont plus flexibles et moins lourdes à déployer.

Ces deux catégories sont les extrêmes d'un panel de systèmes existant.

I.2.6 Intégration de la tolérance aux fautes

L'adaptation de la tolérance aux fautes nécessite de concevoir la tolérance aux fautes pour être capable de la modifier. Nous nous intéressons ici aux manières dont la tolérance aux fautes est traditionnellement intégrée dans un système.

Il existe de nombreuses approches pour intégrer la tolérance aux fautes dans un système. On distingue :

- Les approches bas-niveau (système) : ce sont les approches où les mécanismes de tolérance aux fautes sont intégrés dans les couches les plus basses du système.
- Les approches intermédiaires (intégration, interception, réflexivité) : elles consistent à introduire une couche supplémentaire afin de fournir des abstractions de la tolérance aux fautes.
- Les approches du niveau applicatif (appel de services, librairie, héritage, réflexivité langage) : l'application a la responsabilité de sa tolérance aux fautes

I.2.6.1 Approches bas-niveau

Les approches bas-niveau visent à masquer la tolérance aux fautes au niveau applicatif. Elles proposent d'intégrer le logiciel de tolérance aux fautes dans les couches basses du système. Ces mécanismes ont alors accès à des informations de bas niveau. Toutefois, ces mécanismes ne profitent pas des abstractions plus évoluées du niveau applicatif.

Le système Delta-4 [4] est un exemple d'un système complet tolérant les fautes qui repose sur l'utilisation des contrôleurs réseaux à silence sur défaillance (NAC), matériel auto-testable avec des mécanismes d'exclusion mémoire. Les services applicatifs sont considérés comme des composants sur étagères (COTS). Les mécanismes de tolérance aux fautes permettent de répliquer les composants applicatifs, en utilisant un protocole de communication de groupe à jeton. Les protocoles inter-répliques développés ont permis de mettre en place de la réplication passive, semi-active et active avec vote à la source et vote à la destination. Ces protocoles sont alors intégrés dans la couche de communication.

Le choix du type de réplication utilisé est réalisé hors ligne, lors de la configuration du système, et est indépendant du niveau applicatif. La tolérance aux fautes est alors transparente pour l'utilisateur du système.

I.2.6.2 Approches intermédiaires

Les approches intermédiaires consistent en l'introduction d'une couche entre le niveau applicatif et les couches les plus basses du système. Cette couche est communément appelée intergiciel. Elle masque intégralement le système au niveau applicatif et offre de nouvelles abstractions à la couche applicative. Par exemple, CORBA introduit les objets distribués, la notion d'annuaire, de stub ou encore de skeleton.

Il existe trois manières de coupler la tolérance aux fautes à un intergiciel :

- L'intégration : cette approche est une alternative aux approches de bas niveau. Elle consiste à intégrer les mécanismes de tolérance aux fautes statiquement dans un intergiciel. Toutefois elle diffère en deux points de l'approche système. Le premier est qu'elle ne permet pas d'accéder aux informations bas-niveau, ce qui était le cas dans une approche système. Le deuxième point est qu'elle permet de profiter d'abstractions plus évoluées, plus proches du niveau applicatif, comme la notion d'objet distribué par exemple. C'est le choix réalisé dans ELECTRA [6]. Comme dans Delta-4, ELECTRA repose sur une couche de communication de groupe permettant la réplication d'objet CORBA. Les mécanismes de tolérance aux fautes sont toutefois dissociés de la communication. Ainsi, il est possible de changer la couche de communication de groupe en utilisant un adaptateur. Cette propriété permet de porter cet intergiciel sous plusieurs architectures système.
- L'interception : cette approche repose sur des mécanismes d'interception d'un intergiciel standard, par exemple CORBA, permettant ainsi de rajouter des mécanismes de tolérance aux fautes à l'intergiciel, sans pour autant le modifier. C'est l'approche utilisée dans ETERNAL [7]. Ainsi, ETERNAL peut être appliqué à n'importe quel ORB du commerce. L'approche repose sur l'utilisation du protocole Internet-Inter-ORB-Protocole (IIOP) qui fait parti du standard CORBA. Ce protocole repose sur la couche TCP-IP du système. Les messages TCP-IP sont interceptés par ETERNAL, et sont déroutés vers un système de communication de groupe. Les mécanismes d'interception sont toutefois très dépendants du système d'exploitation, limitant la portabilité d'ETERNAL. Toutefois, les mécanismes d'interception ne permettent pas d'accéder à l'état de l'application. Les protocoles de réplications sont donc réduits à de la réplication (semi) active. Plus récemment cette approche a été utilisée dans MEAD [8] pour faire coexister les contraintes de tolérance aux fautes et les contraintes temporelles. MEAD s'inscrit dans l'optique de fournir une sûreté de fonctionnement flexible [9] à l'aide d'une tolérance aux fautes proactive. Il sera détaillé plus loin dans ce chapitre.
- La réflexivité : la réflexivité [10] a maintenant fait ses preuves pour intégrer la tolérance aux fautes dans un système [11-13]. Elle permet de séparer les traitements applicatifs des

traitements non-fonctionnels dans des couches différentes du système. Plusieurs systèmes utilisent cette approche. MAUD [14] est basé sur le support d'exécution réflexif GARF [15] utilisant la réflexivité fourni au niveau du langage SmallTalk et applique aux acteurs applicatifs des stratégies de tolérance aux fautes reposant sur de la réplication à l'aide d'une librairie de communication de groupe. FRIENDS [16, 17] utilisent une réflexivité à la compilation à l'aide du compilateur ouvert OpenC++[18], afin de munir les objets CORBA de sondes et d'actionneur pour leur appliquer des mécanismes de tolérance aux fautes. L'objet muni de ses mécanismes réflexifs est alors dynamiquement associé à un méta-objet en charge de sa tolérance aux fautes. Plusieurs mécanismes de réplication ont été réalisés de cette manière.

I.2.6.3 Approches applicatives

L'approche applicative consiste à déléguer au concepteur de l'application le soin de mettre en place sa tolérance aux fautes. Bien que le concepteur puisse concevoir les mécanismes de tolérance aux fautes dans son intégralité, plusieurs approches existent pour favoriser la réutilisation de mécanismes de tolérance aux fautes déjà existants. L'utilisation de librairie en est un bon exemple. Elle peut se traduire de deux manières différentes : l'utilisation de services et l'héritage.

Les services sont fournis par les couches inférieures du système. Il peut s'agir du système d'exploitation, d'un intergiciel ou d'une librairie indépendante. Ces services sont assez similaires à ceux existant dans une approche système. Toutefois, l'appel à ces services est à la charge du niveau applicatif. Contrairement à une approche système, cette approche ne fournit aucune transparence pour le niveau applicatif. OPENDREAM [19] utilise cette approche dans un intergiciel de type CORBA. Des services de communication de groupe sont rendus disponibles à la couche applicative. On peut aussi citer des librairies indépendantes, fournissant des mécanismes de sauvegarde d'état (libckp [20]) sur support stable.

ARJUNA [21] propose une approche par héritage. La librairie est constituée d'un ensemble de classes fournissant des mécanismes de tolérance aux fautes. Ainsi, lorsque la classe fonctionnelle hérite d'une classe de la librairie, elle hérite des mécanismes de tolérance aux fautes correspondant.

I.2.6.4 Évaluation des approches d'intégrations

Afin de pouvoir modifier les mécanismes de tolérance aux fautes pendant l'exécution du système fonctionnel, nous cherchons à caractériser les approches d'intégration proposées selon les critères de transparence et séparation vis-à-vis de l'applicatif, de configurabilité, de composabilité, de réutilisabilité et de portabilité.

- *Transparence* et *séparation* : la transparence vis-à-vis de l'applicatif est le fait que la tolérance aux fautes est invisible pour l'applicatif. La séparation est l'indépendance existante entre le logiciel de tolérance aux fautes et le logiciel applicatif. Ces deux caractéristiques sont importantes pour limiter et maîtriser l'impact des modifications à l'exécution du logiciel de tolérance aux fautes sur l'applicatif. Les approches du niveau applicatif à base de service offrent un très fort couplage et une transparence inexistante entre l'applicatif et le fonctionnel. Les approches par héritage ou intégration fournissent une meilleure séparation, mais une transparence toujours très faible. Les approches par interception ou réflexives et les approches système fournissent une transparence et une séparation tout à fait satisfaisante.
- *Configurabilité* : la configurabilité est la propriété de pouvoir choisir les mécanismes de tolérance aux fautes à mettre en place pour un logiciel applicatif. Cette propriété permet

de rendre visible les mécanismes de tolérance aux fautes à un utilisateur de la plateforme tolérant les fautes. C'est une propriété importante pour pouvoir modifier le logiciel de tolérance aux fautes. La configurabilité peut être obtenue hors ligne ou en ligne. Une configurabilité en ligne implique une configurabilité hors ligne. Lorsqu'elle est obtenue en ligne, la tolérance aux fautes est alors adaptable, dans le sens où elle peut être modifiée pendant le fonctionnement du système. Les approches proposées fournissent toutes un degré de configurabilité hors ligne satisfaisant. Toutefois, aucun moyen n'est disponible pour modifier les mécanismes de tolérance aux fautes en cours de fonctionnement.

- *Composabilité* : la composabilité est la propriété de pouvoir composer plusieurs mécanismes de tolérance aux fautes de natures différentes. Si les mécanismes de tolérance aux fautes ne sont pas composables entre eux, il est difficile de réutiliser les mécanismes existants dans une nouvelle stratégie de tolérance aux fautes mise en place lors de l'adaptation. Dans les approches possédant une séparation faible, la composition est implicite, et repose sur le concepteur du fonctionnel. C'est aussi le cas des approches par intégration. Les approches à base d'interception ou système ne semblent pas prendre en compte cette composition. L'approche réflexive permet une bonne composabilité grâce à son protocole réflexif qui offre une certaine souplesse dans la réalisation des mécanismes de tolérance aux fautes.
- *Réutilisabilité* : la réutilisabilité est la propriété de pouvoir réaliser du nouveau à partir d'ancien. La réutilisabilité est importante pour les mécanismes de tolérance aux fautes, mais surtout pour les stratégies de tolérance aux fautes basées sur ces mécanismes. Ainsi, lors de l'adaptation d'une stratégie de tolérance aux fautes à une autre, il est intéressant de pouvoir conserver les mécanismes présents dans l'ancienne stratégie pour les utiliser dans la nouvelle stratégie à mettre en place. Lorsque les mécanismes et stratégies sont réalisés de manière statique dans le système, il est très difficile de les réutiliser. Lorsqu'au contraire, ils sont dynamiquement présents dans le système, leur réutilisation est alors simplifiée. C'est le cas des approches réflexives et à base d'interception. L'héritage fournit toutefois une réutilisabilité du code, intrinsèque à la technique utilisée.
- *Portabilité* : la portabilité est la propriété d'indépendance par rapport aux couches inférieures du système (matériel, noyau, intergiciel). Cette indépendance est importante en tolérance aux fautes, surtout lorsque les principes de diversité sont utilisés sur ces couches. L'approche système proposée ainsi que l'approche par interception sont très dépendantes des couches inférieures du système. La portabilité des autres techniques dépend de la manière dont elles sont implémentées. Par construction, OPENDREAM est la solution la plus portable, parmi toutes les solutions proposées, car elle repose sur la norme CORBA, et peut être couplée à tout intergiciel respectant cette norme.

I.3 ADAPTATION ET TOLERANCE AUX FAUTES

Nous allons tenter dans cette section de caractériser les approches de l'adaptation. Nous nous intéresserons ensuite à l'état de l'art concernant l'application de la notion d'adaptation vis-à-vis de la tolérance aux fautes.

I.3.1 Acception de l'adaptation

Les termes « *adaptable* » et « *adaptation* » inondent le monde de la recherche depuis maintenant plusieurs années. Il est parfois très difficile de comprendre ce qui se cache derrière ces termes. Nous proposons ici de caractériser les notions qui se cachent derrière ces termes afin de clarifier l'objectif des travaux réalisés dans cette thèse. Pour cela, nous étudions plusieurs aspects : l'instant où l'adaptation est réalisée, le centre décisionnel de l'adaptation, la cible de l'adaptation c'est-à-dire ce qui est effectivement modifié dans le système et enfin les techniques utilisées pour ce faire.

I.3.1.1 Instant de l'adaptation

L'adaptation peut être caractérisée selon l'instant où les modifications sont appliquées. On distingue :

- L'adaptation hors-ligne, où le système n'est pas en fonctionnement nominal concerne :
  - La configuration : la configuration est appliquée hors ligne et est prise en compte lors du démarrage du système. La reconfiguration est le fait de changer la configuration du système et de le redémarrer (dans sa totalité ou partiellement) pour prendre en compte la nouvelle configuration.
  - La maintenance : la maintenance est proche de la configuration, à ceci prêt que le système se trouve généralement dans un mode particulier dans lequel il ne fournit pas de service (ou des services très dégradés) qui n'est pas nécessairement l'arrêt, et que les causes de l'adaptation sont différentes : l'objectif de la maintenance est strictement de réaliser des actions préventives pour maintenir le système en fonctionnement.
  - La conception : l'adaptation se focalise sur la réutilisation de composants existants (COTS) [22]. Lorsque ces composants ne répondent pas tout à fait aux besoins du système (interfaces, service rendu ou manière dont le service est rendu), il est nécessaire de modifier son code. Les approches reposent alors sur un suivi des versions des composants à l'aide de modèles permettant de définir quelle partie du code doit être modifiée, et de quelle manière.
- L'adaptation en ligne : l'adaptation en ligne est la modification autonome du système pendant son fonctionnement normal pour répondre à un changement de contexte opérationnel. Autonome signifie que le système est capable de déterminer et d'appliquer les modifications sans intervention d'un tiers. De plus, nous le verrons dans la section suivante (cf. I.3.1.2), pour prévoir ou détecter le changement opérationnel, il est nécessaire que le système ait la capacité d'observer son environnement ainsi que son propre fonctionnement. Ainsi, lors de l'adaptation, le système continue à fournir le service, en dépit des modifications qu'il subit, et n'est pas intégralement redémarré.

Dans ces travaux, nous nous focalisons sur l'adaptation en ligne du logiciel, et tout particulièrement, le logiciel de tolérance aux fautes.

I.3.1.2 Centre décisionnel

Dans une adaptation en ligne, il existe nécessairement un déclencheur de l'adaptation. Ce déclencheur est celui qui va entraîner les modifications sur le système pendant son fonctionnement pour tenir compte d'un changement de contexte opérationnel. Ce déclencheur est appelé centre décisionnel de l'adaptation.

Dans un *système adaptable*, le centre décisionnel est considéré comme ne faisant pas partie du système considéré. Ainsi, le système reçoit un ordre d'adaptation de l'extérieur. En réponse à cet ordre, le système se modifie. Un système adaptable est un système modifiable.

Dans un *système adaptatif*, ou auto-adaptable, le centre décisionnel de l'adaptation fait partie intégrante du système. Ainsi, le système est autonome vis-à-vis de son adaptation. C'est un système adaptable qui prend lui-même les décisions d'adaptation.

Dans un système adaptatif, deux cas d'adaptation peuvent être différenciés. Dans le cas où l'adaptation est réalisée en réponse à un changement, elle est qualifiée de *réactive*. Il est facile de réagir à un contexte opérationnel du fait même que celui-ci est observable au moment où l'on décide de l'adaptation. Ce type d'adaptation est celui le plus souvent mis en place. Toutefois, il n'est pas sans inconvénients. En effet, par construction, la modification du système intervient après observation du changement de contexte. Il existe donc une phase de transition où le système ne correspond pas au contexte opérationnel dans lequel il se trouve. De même, dans un système possédant des contraintes temporelles, le terme « après » peut être synonyme de « trop tard ». Pour palier à ce genre de problème, un autre type d'adaptation existe. Si l'adaptation est réalisée de manière préventive, pour pallier une situation future, l'adaptation est qualifiée de *proactive*. Faire de l'adaptation proactive est limitée par la capacité à prédire les phénomènes opérationnels du système. Toutefois, ce type d'adaptation est réellement intéressant, car il permet d'anticiper les problèmes afin de les éviter. L'adaptation proactive a été utilisée dans MEAD [8] et dans [23] pour concilier des contraintes de tolérance aux fautes et des contraintes temporelles.

I.3.1.3 Cible de l'adaptation

Nous considérons le logiciel comme une implémentation d'un algorithme au sens mathématique qui vise à fournir un service. Un algorithme est une notion compositionnelle. En effet, un algorithme peut être décomposé en sous-algorithmes. Un algorithme possède deux types de données : les paramètres et les variables. Les variables sont manipulées par l'algorithme et évoluent au fur et à mesure de son exécution. Les paramètres sont des constantes pour l'algorithme et déterminent son résultat ou la manière dont ce résultat est obtenu.

Ainsi, l'adaptation peut avoir lieu à trois niveaux :

- l'algorithme : lorsque l'algorithme est modifié, le service n'est alors plus rendu de la même manière et les variables manipulées par le nouvel algorithme doivent être initialisées dans un état cohérent avec le passé du système. Il s'agit alors d'un transfert de l'état du système avant adaptation vers le système après adaptation. Un changement d'algorithme est appelé un *basculement*.
- les paramètres : lorsque les paramètres de l'algorithme sont modifiés, le fonctionnement intrinsèque de l'algorithme n'est pas changé. Un changement de paramètres est appelé un *paramétrage*.
- les interfaces : pour rendre compatibles des algorithmes entre eux, il est parfois nécessaire de modifier leurs interfaces. Par exemple, lorsqu'un service web veut utiliser une entité

CORBA sur Internet, il est nécessaire de rendre compatibles les interactions entre ces entités. Ce problème peut être résolu par l'introduction d'un bus de données générique sur lequel des connecteurs logiciels sont connectés et font la passerelle entre le bus et les entités à relier. C'est le cas de PEtALS [24] qui repose sur les technologies JBI [25].

Il existe un cas particulier de l'adaptation du logiciel où le paramètre d'un algorithme permet de spécifier un sous-algorithme à utiliser. En effet ce cas est un paramétrage dans le sens où la modification sur le système est un changement de paramètre. Mais il peut être considéré comme un basculement dans le sens où un nouvel algorithme (au sens compositionnel) est utilisé. De notre point de vue, il s'agit d'un paramétrage dans le sens où, ce basculement est prévu dans la spécification de l'algorithme, et que les deux algorithmes existent conjointement à l'exécution. Il existe donc une volonté sous-jacente du basculement : minimiser l'empreinte mémoire du logiciel en ne chargeant en mémoire que le logiciel strictement nécessaire à un instant donné de la vie opérationnelle du système.

Illustrons ces notions par en exemple. Prenons l'exemple d'un logiciel de contrôle de processus continu. Ce logiciel contient un algorithme de calcul d'intégrale de signal échantillonné. Il existe plusieurs manières de calculer cette intégrale : l'intégrale par la méthode des rectangles avec retard qui est rapide mais peu précise, et l'intégrale par la méthode des trapèzes avec retard qui est plus précise mais un peu plus complexe. Dans le calcul de cette intégrale, la période d'échantillonnage est un paramètre de l'algorithme. Les variables sont le flux du signal d'entrée, la valeur de l'intégrale avant l'instant courant, et, dans le cas du calcul par trapèzes, la dernière valeur du signal d'entrée. Le paramétrage de la période d'échantillonnage ne nécessite pas de mise à jour des variables de l'algorithme. Le basculement de la méthode d'intégration des rectangles à la méthode d'intégration par trapèzes nécessite l'initialisation des variables pour le nouvel algorithme. Le signal d'entrée utilisé par l'ancienne méthode doit être fourni à la nouvelle. La valeur de l'intégrale doit être initialisée à la valeur de la dernière valeur calculée par l'ancienne méthode. Le dernier échantillon du signal est nécessaire à la nouvelle méthode, mais n'est pas une variable pour l'ancienne : il ne fait donc pas partie de son état. Dans ce cas, il peut être envisagé de l'initialiser avec la valeur courante du signal d'entrée. Dans le meilleur cas, cette valeur correspond à la valeur du pas précédent, et dans le pire cas, l'algorithme des trapèzes se comportera alors comme l'algorithme des rectangles lors du prochain pas de calcul.

I.3.1.4 Modification du logiciel

La modification du logiciel, i.e. de son code et de son état, peut être mise en place de plusieurs manières. Dans [26], les auteurs référencent différentes approches à cette modification. De notre point de vue, ces approches peuvent être classées selon deux catégories :

- Approche où l'adaptation repose sur des capacités des couches inférieures du système. Par exemple un système d'exploitation permet de mettre en pause l'exécution de processus et de dynamiquement changer le contenu de leur mémoire, ou bien permet de modifier dynamiquement les librairies logicielles associées [27]. Toutefois ces approches reposent sur des abstractions de très bas niveau qui sont peut intrusives mais dont l'impact sur le système est difficile à maîtriser ;
- Approche nécessitant l'instrumentation de l'application : le logiciel est muni de mécanismes permettant sa modification comme par exemple, l'utilisation de proxys pour dynamiquement changer des classes C++ dans un programme en cours de fonctionnement [28]. Ces approches sont relativement intrusives, mais fournissent des abstractions de haut-niveau faciles à manipuler.

Situé à la frontière de ces deux approches, les modèles à composant introduisent le concept de composant logiciel. Un composant logiciel est une boîte qui encapsule du logiciel. Il possède des interfaces au travers desquelles sont fournis des services. Ces interfaces peuvent être fournies par un composant qui joue le rôle de prestataire du service, ou requis par un composant qui joue alors le rôle de client. Un modèle à composant repose sur l'utilisation d'une plateforme logicielle spécifique en charge de manipuler des composants logiciels, mais aussi des moyens d'instrumentation des composants logiciels pour permettre leur modification. Nous observons aujourd'hui un réel engouement pour ce type de technologie, qui est largement utilisée [29-31].

I.3.2 Systèmes adaptables tolérants aux fautes

I.3.2.1 Modification du nombre de répliques

Deux projets se sont focalisés sur l'adaptation du nombre de répliques au contexte opérationnel : AQuA et MEAD

AQuA

AQuA [32, 33] est un projet se focalisant sur la qualité de service (QoS) pouvant être dynamiquement adaptée. Plus précisément, les besoins non fonctionnels des clients peuvent être différents, et même, évoluer au cours du temps. Il est intéressant de fournir uniquement la qualité de service désirée par ceux qui utilisent le service. Le but d'AQuA est donc de fournir un intergiciel pourvu de mécanismes permettant de faire évoluer la qualité du service rendue en fonction des besoins des utilisateurs. Ceci est réalisé grâce à la réplication dynamique d'objets. Ces objets répliqués forment alors un groupe de répliques.

Ce travail explore comment le nombre de répliques dans une stratégie reposant sur de la réplication peut permettre de résoudre les problématiques suivantes :

- *Tolérer un nombre variable de crash* [33, 34] : la solution proposée consiste à faire varier le nombre de répliques dans une réplication passive ou active. Le problème majeur consiste à synchroniser l'état de la nouvelle réplique avec l'état des répliques existantes et de déterminer à partir de quel instant, la nouvelle réplique est opérationnelle et doit être prise en compte ;
- *Tolérer un nombre variable de défaillance en valeur* [33] : les défaillances en valeur sont tolérées par l'utilisation de la réplication active et de la mise en place d'un vote. Le problème reste assez similaire au précédent, à ceci près que l'entité en charge du vote doit être elle aussi mise à jour pour prendre en compte la nouvelle réplique ;
- *Garantir les temps d'exécution* [35] : cette technique ne vaut que pour des services sans mémoire et déterministes. Les temps réseau, dans les files d'attentes et de traitement étant dynamiquement mesurés, un nombre de répliques peut être choisi pour atteindre la probabilité souhaitée d'obtenir le résultat dans le temps imparti. Le souci majeur est d'obtenir les informations concernant les temps d'attente et de choisir les répliques en conséquence ;

AQuA met en place une passerelle (Figure 1) qui est un frontal pour les applications objets clientes. Ce frontal permet à chaque client de s'adresser à un unique objet de type CORBA, alors qu'en réalité, toutes les répliques du service sont sollicitées.

Cette passerelle possède une interface standard IIOP pour échanger les données avec le monde CORBA. Elle possède aussi un ensemble de primitives permettant de communiquer de manière sûre avec toutes les répliques d'un même groupe.

Il est intéressant de remarquer que les processus prenant part aux décisions concernant la QoS sont ici externes à la passerelle, et s'exécutent dans le monde CORBA. La gestion de la sûreté est réalisée de manière centralisée et redondée. AQuA utilise de plus des objets de qualité baptisés *Quality Object* (QuO). Ils permettent à un utilisateur de spécifier ses attentes non-fonctionnelles. Un nouveau contrat de QoS est alors édité entre le gestionnaire de qualité et le QuO. Nous ne rentrerons pas dans les détails de ces objets, faute de place. Plus de détails sont fournis dans [36].

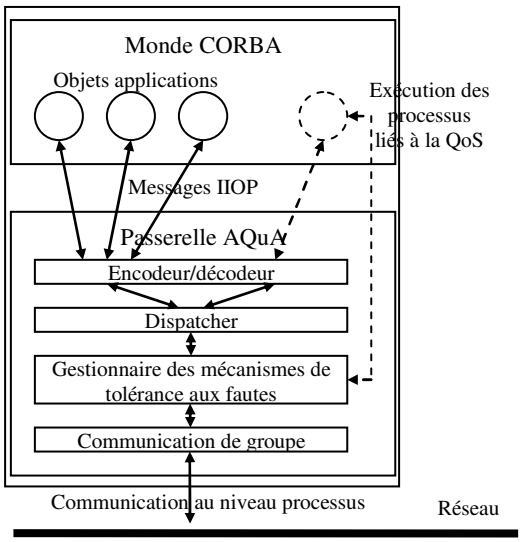


Figure 1 **Architecture d'AQuA.**

La tolérance aux fautes est ici adaptative. Les mécanismes de tolérance aux fautes sont intégrés dans une passerelle. Le paramétrage du mécanisme en place est la seule action réalisable. Ce paramétrage consiste principalement à modifier le nombre de répliques à prendre en compte. Aucun changement de mécanisme, i.e. de stratégie, à l'exécution n'est envisagé. La supervision/décision est réalisée de manière centralisée. Les sources d'informations sont variables. Il s'agit dans un cas des ressources et nœuds disponibles, et dans un autre des temps de traitement des répliques.

MEAD

MEAD [8] est un intergiciel introduisant le concept de tolérance aux fautes versatile [9] pour des applications temps-réel CORBA. L'idée de la sûreté de fonctionnement versatile est qu'il n'existe pas d'intergiciel répondant à la fois aux problématiques du temps-réel et de la tolérance aux fautes pour les applications CORBA, tout simplement parce que souvent les exigences sont contradictoires. Ainsi, les auteurs proposent d'introduire la notion de compromis entre temps réel et tolérance aux fautes. Ces compromis induisent des modifications quand aux propriétés de la tolérance aux fautes pour assurer un comportement temporel souhaité. La tolérance aux fautes est alors qualifiée de versatile, c'est-à-dire qu'elle est susceptible de changer.

MEAD complète les spécifications RT-CORBA en introduisant les nouvelles fonctionnalités suivantes :

- des mécanismes de tolérance aux fautes dynamiquement paramétrables (nombre de répliques, temps d'attente, fréquence de la sauvegarde d'état, etc.) ;
- la possibilité de réaliser des actions préventives qui consistent à anticiper la défaillance d'une réplique due à un défaut de ressources par l'ajout de répliques sur d'autres sites [37] ;
- la prise en compte des ressources disponibles avant de réaliser toute action liée à la tolérance aux fautes par exemple, la création de répliques ne peut s'effectuer que si certaines ressources sont disponibles.

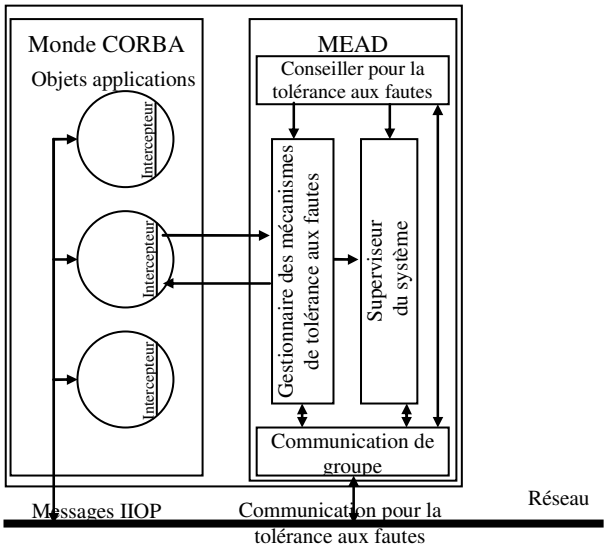


Figure 2 **Architecture de MEAD.**

Pour introduire de tels mécanismes dans l'intergiciel RT-CORBA, des mécanismes d'interception sont utilisés (cf. figure 2). Il s'agit de rediriger l'édition de lien dynamique des librairies standards CORBA vers des librairies spécialisées, contenant les mécanismes de MEAD.

Le gestionnaire de tolérance aux fautes tient compte de la fréquence des fautes, de l'utilisation des ressources, des bornes temporelles tolérées pour le recouvrement pour paramétrer les mécanismes de détection et recouvrement en conséquence pour chacun des objets. Cette supervision est réalisée de manière totalement distribuée. Chaque nœud possède un gestionnaire pour la tolérance aux fautes qui participe au choix de mécanisme/configuration en fournissant des informations sur son propre nœud, comme les ressources disponibles ou les temps d'exécution, au travers d'une couche de communication de groupe sûre et garantissant un ordre total sur les messages.

D'autres problèmes conceptuels sont traités. Le principal est celui du déterminisme des applications. Dans le domaine du temps-réel, cette notion fait référence au fait que les tâches des applications sont bornées dans le temps. Dans le domaine de la tolérance aux fautes, le déterminisme fait référence à la reproductibilité d'un comportement, i.e. un même mécanisme est sensé produire le même résultat à partir du même état initial. Il est alors nécessaire de s'assurer que l'application conçue pour le temps réel réponde bien à la problématique du déterminisme pour la tolérance aux fautes. L'approche utilisée est l'analyse statique du code, et l'introduction à l'exécution de synchronisations pour les sources de non-déterminisme entre les répliques.

En conclusion, MEAD utilise l'interception pour introduire de la tolérance aux fautes adaptative dans un intergiciel temps-réel. Les mécanismes de tolérance aux fautes utilisés sont la réplication passive et semi-active. Le déterminisme des répliques est assuré. L'adaptation consiste à modifier à l'exécution, d'une part, certains paramètres (nombre de répliques, temps d'attente, fréquence de sauvegarde de l'état, etc.), d'autre part, les mécanismes de synchronisation de l'état des répliques (i.e. basculer de la réplication passive à la réplication semi-active). Le basculement est effectué entre deux traitements lors d'un gel du service. L'objectif de l'adaptation est exprimé dans un contrat de sûreté qui stipule, par exemple, le nombre de fautes supportées et le temps de recouvrement toléré (garanties temporelles). Lorsque les ressources ne permettent plus de maintenir une configuration, l'adaptation est alors déclenchée. Les mécanismes de tolérance aux fautes possèdent un grand nombre de paramètres, tant au niveau de la détection (fréquence des *Heart Beat*) qu'au niveau du recouvrement (fréquence de l'envoi des points de reprise). Les informations concernant les ressources et les fautes rencontrées sont à la base de la supervision qui est distribuée.

I.3.2.2 Changement de stratégie

Plusieurs projets se sont focalisés sur le changement de stratégie pendant l'exécution : ROAFTS et AFT-CCM.

ROAFTS

Les Recovery Blocks (RB) [38] sont une technique de tolérance aux fautes du logiciel basée sur la diversification de fonctions comme dans la programmation diversifiée appelée *N-Version Programming* [2]. Chaque bloc est constitué de plusieurs variantes de la procédure fonctionnelle ainsi que d'un test d'acceptation (AT) en charge de valider le résultat fourni par chacune des variantes.

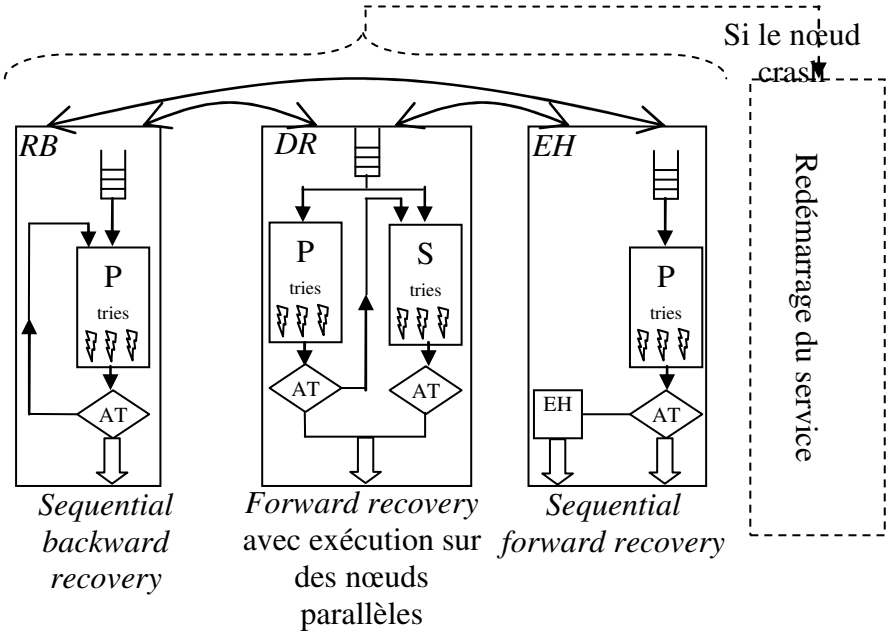


Figure 3 Un Adaptive Recovery Block.

Cette notion a été élargie en *Distributed Recovery Blocks* (DRB) [39, 40] pour tolérer les fautes matérielles et logicielles dans les applications temps réel distribuées en utilisant l'exécution sur des nœuds distants de deux RB (cf. figure 3). Les variantes sont exécutées

dans l'ordre de temps d'exécution décroissant sur le nœud primaire (P) et inverse sur le secondaire (S). La défaillance temporelle du nœud P peut être recouvrée par le nœud S qui a exécuté des variantes plus rapides. L'échec des deux RB provoque la défaillance du DRB.

Les ARB [40] permettent de basculer à l'exécution les mécanismes de tolérance aux fautes moyennant le gel du système. Le choix est effectué entre le recouvrement arrière (RB), le recouvrement avant (DRB) ou gestion d'erreur (EH) et n'est pas extensible. Ces mécanismes de tolérance aux fautes ne sont pas paramétrables.

Les changements de mode non-fonctionnel sont effectués entre deux traitements pour faciliter le maintien de la cohérence de l'état de la réplique principale. En effet, il est plus simple de s'assurer du bon résultat ou de recouvrer une erreur, avant d'entamer les modifications sur le mode non-fonctionnel. De plus, l'état du secondaire doit être synchronisé avec celui du primaire dans le cas du passage du mode RB (ou EH) à DRB.

ROAFTS [41] est un intergiciel qui introduit une capacité de supervision nommé *Network Surveillance and Reconfiguration* pour commander l'adaptation des ARB. Il permet donc de rendre adaptative la tolérance aux fautes vis-à-vis des mécanismes proposés par les ARB. Cette supervision est réalisée de manière semi-centralisée par l'utilisation de superviseurs chargés de la surveillance au travers du réseau. Une autre entité appelée *Adaptive Fault Tolerance Manager* est l'actionneur en charge d'effectuer les modifications relatives aux ARB dans le système.

Une autre application des ARB a été réalisée pour une mission spatiale [42] et ressemble beaucoup au travail proposé dans ROAFTS. La supervision consiste alors à repérer les changements de phase de mission. En effet, chaque phase met en place des mécanismes de tolérance aux fautes spécifiques, alors que les applications fonctionnelles restent les mêmes. La philosophie est identique, nous ne détaillerons pas plus cet exemple pratique d'application.

AFT-CCM

AFT-CCM [43], mis pour *Adaptive Fault Tolerant Corba Component Model*, est un projet visant à introduire les concepts de tolérance aux fautes adaptable [44] dans l'intergiciel CORBA Component Model [45] (CCM) qui fournit un modèle à composant distribué à base d'appels de procédure distants. AFT-CCM permet à un concepteur de spécifier la qualité de service requise pour son application cliente en termes de niveau de tolérance aux fautes. AFT-CCM introduit plusieurs entités au niveau du modèle à composant CCM :

- Entités de tolérance aux fautes :
  - *Replication Coordinator* : ce composant à en charge d'assurer le protocole de réplication entre les différentes répliques du service. Par exemple, ce composant réalise un protocole de réplication passive, en utilisant des mécanismes de transfert d'état des composants fonctionnels. L'adaptation consiste à remplacer ce composant par un autre, assurant un protocole de réplication différent.
  - *Connector* : un connecteur fait la liaison entre le composant client et le composant serveur. Il intercepte les appels et les redirige vers le *Replication Coordinator* en charge de la réplique primaire.
  - *Fault Detection Agent (FD-Agent)* : ce composant assure le mécanisme de détection du crash d'un composant fonctionnel et du *Replication Coordinator* qui lui est associé.
- Entités d'adaptation :

- *Fault-Tolerance Manager (FT-Manager)* : chaque composant fonctionnel avec des contraintes de tolérance aux fautes se voit attribué un composant FT-Manager. Un FT-Manager définit la configuration de la tolérance aux fautes déduite des besoins en qualité de service attendue pour le composant applicatif. La reconfiguration (remplacement des *Replication Coordinators*) a lieu lorsque les données collectées par le FT-Manager montre que la configuration actuelle ne satisfait pas les besoins en termes de tolérance aux fautes.

Ces travaux visent donc à créer des systèmes adaptatifs, permettant le déploiement d'une stratégie de tolérance aux fautes quelconque. On remarque dans ces travaux une volonté de généraliser l'adaptation du logiciel de tolérance aux fautes par l'utilisation de paradigme propices à la manipulation du logiciel (les composants) et une séparation plus ou moins claire du logiciel de tolérance aux fautes de celui en charge des fonctionnalités. Toutefois, le comportement du logiciel pendant l'adaptation n'est pas explicite.

I.3.2.3 Cas du temps réel dur

Les FERTs (*Fault-tolerant Entity for Real Time)* tentent de répondre au problème de l'adaptation dans le domaine du temps-réel. Les tâches considérées sont des tâches souvent périodiques possédant une échéance temporelle. Ces tâches sont ici considérées sans état et déterministes.

Un FERT (Figure 4) est un composant constitué d'un ensemble de modules d'application pour la tâche qu'il représente, notés AM_i. Ces modules d'application peuvent être des algorithmes fonctionnels ou des adjudications comme des algorithmes de vote par exemple et peuvent être exécutés sur des nœuds différents. Une partie contrôle sert de glue entre tous les modules d'application en indiquant, d'une part, leurs interactions et, d'autre part, leur agencement dans le temps.

Ce contrôleur est couplé avec un ordonnanceur possédant deux niveaux d'ordonnancement (cf. figure 4). Le premier ordonnance les tâches de manière statique. L'ordre des tâches est calculé hors ligne pour garantir leurs échéances temporelles. Le deuxième niveau d'ordonnancement récupère le temps non-utilisé par une tâche (le *slack time*) pour le mettre à la disposition de la tâche suivante. Ce *slack time* est la pierre angulaire de ce composant. En effet, la tâche suivante peut profiter de ce temps supplémentaire pour redonder son exécution. Deux sortes de redondance sont considérées :

– la *redondance séquentielle* : des variantes d'algorithmes sont exécutées sur le même nœud, les unes à la suite des autres.

– la *redondance spatiale* : des variantes d'algorithmes sont exécutées sur des nœuds différents en parallèle et nécessitent un vote.

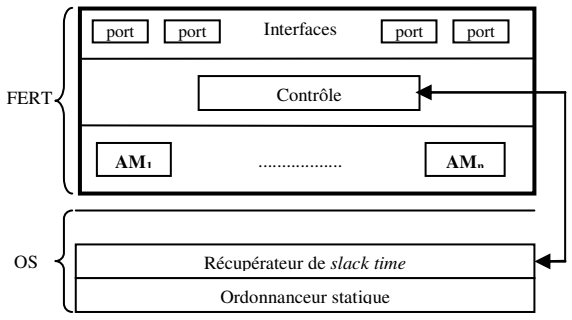


Figure 4 Un FERT et le noyau système à double niveau d'ordonnancement.

Les interfaces permettent à la partie contrôle de synchroniser les informations et les actions entre FERT.

Une sorte de langage « métier » est utilisé pour décrire le fonctionnement d'un FERT, caractérisant ainsi les politiques de redondance qui peuvent être mises en place, et leur temps d'exécution. L'ordonnancement hors ligne garantit l'exécution de la politique minimale. Le *slack time* permet d'exécuter des politiques avec plus de redondance.

L'adaptation fournie par les FERTs réside dans la capacité de choisir, avant chaque exécution de tâche, une des politiques de redondances prédéfinies hors-ligne. Ces politiques sont des configurations agençant les mécanismes de tolérance aux fautes. Ainsi, elles correspondent à des stratégies de tolérance aux fautes. Les mécanismes de tolérance aux fautes sont limités et reposent sur des techniques de redondance spatiale et séquentielle. La supervision est réalisée de manière distribuée par les contrôleurs qui collectent les données temporelles fournies par l'ordonnanceur et celles relatives à l'état des autres répliques. Cette technique respecte les échéances temporelles des tâches temps-réel. L'adaptation consiste à choisir la politique de redondance maximale à utiliser selon une table des correspondances entre temps et ressources disponibles.

I.3.2.4 Approches génériques

CHAMELEON [46] est un projet reposant sur l'utilisation d'une librairie de classes fournissant une infrastructure pour la tolérance aux fautes et sa modification à l'exécution. Ces classes donnent lieu à des objets appelés ARMOR qui sont les briques de l'infrastructure du système.

Il existe deux types d'objets. Les objets réalisant des mécanismes de tolérance aux fautes (vote, détection, réplication), et ceux en charge de la reconfiguration à l'exécution appelés les *Surrogate Managers* qui, d'une part, sont responsables de l'exécution d'une application et de la tolérance aux fautes qui lui est associée, et d'autre part, ont la charge d'installer et désinstaller les objets ARMOR de tolérance aux fautes de l'application. Ainsi, une application hérite d'une classe *Application*. Toutefois, il ne s'agit que d'une infrastructure, l'implémentation est laissée au jugement du concepteur du système.

I.4 PRINCIPES DE L'APPROCHE PROPOSEE

I.4.1 Limites de l'existant

Une question récurrente à propos de ces travaux concerne l'assurance que le système fonctionne correctement en dépit des modifications qu'il subit. En effet, quelles sont les garanties que les modifications ne vont pas entraîner des erreurs dans le système ? La modification du logiciel pendant son fonctionnement soulève trois problèmes :

- Isoler l'exécution des parties du logiciel qui va subir les modifications : pendant l'adaptation en ligne, il existe un accès concurrent sur le logiciel entre les tâches du logiciel qui rendent le service pour lequel le logiciel est conçu, et les tâches en charge de la modification du logiciel.
- Modifier le logiciel : le changement de réplique comme réalisé dans AQuA ne nécessite pas la modification du logiciel, mais uniquement son paramétrage. Toutefois, lorsque l'on souhaite changer de stratégie de tolérance aux fautes (changement du protocole de réplication par exemple) le logiciel doit être modifié. Il s'agit donc, à partir d'un logiciel déployé, de garantir que les modifications vont amener le système dans la configuration souhaitée.
- Mettre le logiciel dans un état de reprise : lorsque les modifications ont été réalisées, l'état des données doit permettre une reprise de l'exécution des tâches sans introduire des erreurs issues d'une divergence de l'état entre les parties du logiciel qui n'ont pas été modifiées et les parties du logiciel qui ont été modifiées. En d'autres termes, le logiciel doit être placé dans un état de reprise tenant compte du passé du système.

Les différents travaux portant sur l'adaptation de la tolérance aux fautes à l'exécution peuvent être regroupées en deux catégories :

- Les approches génériques : ces approches reposent essentiellement sur l'utilisation d'architecture ou une structuration du logiciel permettant de manipuler le logiciel à l'exécution, et donc un changement radical de stratégie de tolérance aux fautes à l'exécution. Elles laissent alors à la charge du concepteur le choix d'implémenter les mécanismes de tolérance aux fautes souhaités. Ces solutions reposent sur l'attribution de rôles dans le processus d'adaptation sans expliquer comment s'assurer du bon fonctionnement du système pendant et après l'application des modifications. C'est le cas d'AFT-CCM ou de CHAMELEON.
- Les approches spécifiques : les mécanismes de tolérance aux fautes sont déterminés par l'approche, et il n'est pas possible d'introduire de nouveaux mécanismes. Toutefois, l'adaptation est très bien maîtrisée, et amène à un fonctionnement correct du système. C'est le cas de tous les autres travaux.

On remarque alors que ces problèmes sont peu détaillés, et lorsqu'ils le sont, la solution est souvent ad-hoc, dépendant souvent, à la fois de l'applicatif et des mécanismes de tolérance aux fautes proposés. Ainsi, il est difficile pour un concepteur de système de réaliser ses propres mécanismes de tolérance aux fautes et de les adapter à l'exécution du système avec les solutions existantes.

I.4.2 Problématique

Dans ce manuscrit nous allons tenter de réconcilier les deux approches, i.e. les solutions génériques et spécifiques. Ces travaux se focalisent sur la modification d'un logiciel critique (le logiciel de tolérance aux fautes) et n'abordent pas les problèmes liés à la décision de l'adaptation. Ainsi, la décision peut être elle aussi spécifiée par le concepteur du système puisqu'elle est rendue indépendante du processus d'adaptation. Cela soulève un problème supplémentaire : lorsqu'une configuration est déployée, les configurations futures sont inconnues.

Dans l'état de l'art, nous avons pu constater que l'adaptation de la tolérance aux fautes nécessite une séparation entre la tolérance aux fautes et les fonctionnalités du système. Cette séparation est essentielle pour permettre de maîtriser le logiciel de tolérance aux fautes pendant l'adaptation et minimiser l'impact de ces modifications sur le fonctionnel. De plus, la modification du logiciel de manière générique nécessite d'être capable de modifier l'algorithmique du logiciel de tolérance aux fautes. Cela signifie qu'il est nécessaire d'utiliser des techniques permettant la modification du code du logiciel. Enfin, nous apporterons lors de modifications l'assurance que le logiciel modifié est placé dans un état permettant une poursuite correcte du système.

De part la complexité des problèmes posés, nous proposons de laisser de côté les aspects temps-réel dans ces travaux.

I.4.3 Approche

L'intégration du logiciel de tolérance aux fautes au travers de la réflexivité offre des propriétés de transparence, de séparation, de configurabilité, de composabilité mais aussi de réutilisabilité. Ces propriétés sont particulièrement intéressantes du point de vue de l'adaptation. La séparation et la transparence sont nécessaires à l'adaptation. La configurabilité démontre la facilité de spécifier des configurations du logiciel de tolérance aux fautes, la composabilité permet de facilement ajouter ou retirer des mécanismes de tolérance aux fautes en les composants et favorise donc l'adaptation. Enfin, la réutilisabilité favorisera l'application d'une même stratégie de tolérance aux fautes à plusieurs entités fonctionnelles. De plus, les propriétés apportées par l'utilisation de la réflexivité sont intéressantes pour l'adaptation elle-même, car elle permet une certaine généralisation de l'approche de l'adaptation.

La modification du logiciel nécessite l'utilisation d'abstractions permettant sa manipulation. De notre point de vue, les modèles à composant sont des intergiciels fournissant des modèles réflexifs d'un logiciel et permettant leur manipulation au travers des abstractions que sont les architectures à composants. Ils permettent donc d'intégrer l'adaptation à un méta-niveau.

Ainsi, notre approche repose sur une architecture réflexive permettant une séparation claire entre le logiciel de tolérance aux fautes et le logiciel fonctionnel, et permettant d'insérer le traitement de l'adaptation comme une couche à part du système. Cette approche sera renforcée par l'utilisation d'un modèle à composant, permettant la manipulation du logiciel pendant son exécution.

Nous détaillerons cette architecture réflexive, les problèmes liés à la conception de la tolérance aux fautes pour son adaptation, les problèmes de la modification d'un logiciel distribué et son impact sur la sûreté de fonctionnement du système.

Ainsi, ces travaux permettront à des concepteurs de systèmes tolérants les fautes, ayant pour objectif d'adapter le logiciel de tolérance aux fautes, de mettre en place une solution leur garantissant la cohérence des modifications du logiciel pendant son exécution.

Chapitre II

REFLEXIVITE ET MODELES A COMPOSANT

II.1 INTRODUCTION

Sur le plan architectural des systèmes, l'approche classique consiste à réaliser un intergiciel spécialisé qui fournit des mécanismes de tolérance aux fautes. L'objectif de nos travaux est de réaliser un intergiciel permettant la modification de tolérance aux fautes en cours d'exécution. Dans ce but, nous nous appuyons sur une approche réflexive afin de séparer le logiciel de tolérance aux fautes, les fonctionnalités du système et le traitement de la modification du logiciel, et les modèles à composants qui permettre la manipulation du logiciel de tolérance aux fautes pendant son fonctionnement.

Dans un premier temps, nous rappelons les principes de la réflexivité en informatique ainsi que les travaux introduisant son utilisation pour la tolérance aux fautes. Nous proposons d'utiliser la réflexivité de manière récursive pour construire une architecture logicielle permettant d'une part la réalisation d'applications tolérantes aux fautes, et d'autre part, la modification en ligne des mécanismes de tolérance aux fautes. Ensuite, nous présentons la notion de modèle à composant et de composants « ouverts ». Les modèles à composant fournissent des propriétés intéressantes pour la manipulation du logiciel pendant l'exécution. Nous montrons comment un modèle à composant peut être utilisé pour implémenter l'architecture réflexive proposée.

Il a été montré dans [16, 47] que la réflexivité [10, 48] est un très bon moyen pour intégrer la tolérance aux fautes dans un système, car elle fournit des propriétés de transparence, de séparation et de composabilité. Le grand intérêt de cette approche est de séparer conceptuellement (*separation of concerns* en anglais) les aspects fonctionnels et le traitement des aspects non-fonctionnels.

Nous postulons que cette approche réflexive permet de séparer la problématique de l'adaptation des autres problématiques au même titre qu'elle le permet pour la tolérance aux fautes. Nous sommes donc à la recherche d'une architecture réflexive qui satisfasse les besoins de la couche adaptation au même titre que ceux de la couche tolérance aux fautes.

Dans cette section nous présentons ce qu'est la réflexivité, et une extension de cette technologie à un système complexe multicouches. Nous détaillerons ensuite le problème architectural auquel nous sommes confrontés et étudierons les solutions à base de réflexivité qu'il est possible de mettre en place. Nous conclurons ensuite sur l'architecture réflexive choisie pour traiter le problème de l'adaptation en ligne.

II.2 REFLEXIVITE ET ARCHITECTURE LOGICIELLE

II.2.1 Principes

La réflexivité est la propriété d'un système qui est capable de raisonner à propos de lui-même [10]. Cette définition élémentaire introduit la notion de moyens d'analyse, de modification ou d'extension a posteriori du comportement initial d'un système informatique.

II.2.1.1 Historique et définition

La réflexivité est un concept né de la philosophie. Est alors qualifié de **réflexif** [Littré] ce qui est *« Propre à la réflexion, au retour de la pensée, de la conscience sur elle-même »*.

Par la suite, la réflexivité est introduite en informatique comme un moyen d'étendre des fonctionnalités d'un système par l'enrichissement de son modèle grâce à l'ajout d'un méta-modèle. Ce méta-modèle contient les abstractions nécessaires au traitement d'une problématique transverse à celle du système de base, par exemple, la tolérance aux fautes.

La réflexivité a été adoptée comme un outil puissant et général des langages de programmation. On peut en donner la définition suivante [10] :

« Un système informatique est dit réflexif lorsqu'il fait lui-même partie de son propre domaine. Plus précisément cela implique que (i) le système a une représentation interne de lui-même, et (ii) le système alterne parfois entre calcul «normal» à propos de son domaine externe et calcul «réflexif» à propos de lui-même. »

Ainsi, un système réflexif contient une représentation qui décrit la connaissance qu'il a de lui-même. Cette représentation est appelée le **méta-modèle**. Un lien causal existe entre le système réflexif et son méta-modèle : toute modification que l'un des deux subit est répercutée sur l'autre. Ainsi en modifiant son modèle, le système agit indirectement sur lui-même et peut ainsi modifier son état (exécution, données). Le méta-modèle étant lui-même un modèle, nous parlerons de manière générale de modèle.

En pratique (fig. 5), un système réflexif est composé d'un niveau de base où sont effectués les traitements liés à la fonction première du système et d'un **méta-niveau** qui utilise le modèle du système pour traiter les problématiques transverses à sa fonction première. Ce modèle est causalement connecté au système par des mécanismes d'observation qui peuvent être des mécanismes de réification (observation spontanée) et d'introspection (observation à la demande). Le modèle (et donc par causalité le système) peut ensuite être modifié au travers de mécanismes d'intercession structurelle et comportementale. Ces mécanismes d'observation et d'intercession sont appelés les *mécanismes réflexifs* du système. Ils peuvent être considérés comme les sondes et les actionneurs du système.

Figure 5 **Concept de réflexivité**

Dans le monde objet, ces mécanismes réflexifs fournissent ce que l'on appelle le **protocole à méta-objet**. En effet, ces mécanismes définissent un protocole d'interaction entre un objet et un méta-objet (son modèle).

L'**empreinte réflexive** d'une famille de mécanismes est définie comme l'ensemble des mécanismes réflexifs nécessaires à la réalisation d'un mécanisme de cette famille à un méta-niveau [49]. Ainsi, il est possible de définir l'empreinte réflexive pour les mécanismes de tolérance aux fautes, permettant ainsi de mettre en place tous les mécanismes de tolérance aux fautes souhaités. La définition de l'empreinte réflexive pour les mécanismes de tolérance aux fautes à base de réplication a été définie dans FRIENDS [11]. Nous la détaillerons dans la suite de ce manuscrit afin de mettre en place le modèle pour la tolérance aux fautes nécessaire dans ces travaux.

II.2.1.2 Réflexivité Multi-Niveaux

Plus récemment, la réflexivité multi-niveaux [49] a été introduite comme une extension de la réflexivité dans les systèmes complexes en couches (fig. 6). Elle consiste à introduire le modèle de chaque couche du système. Elle permet de résoudre des problèmes complexes liées aux abstractions des couches les plus hautes et nécessitant les moyens d'actions des couches les plus basses.

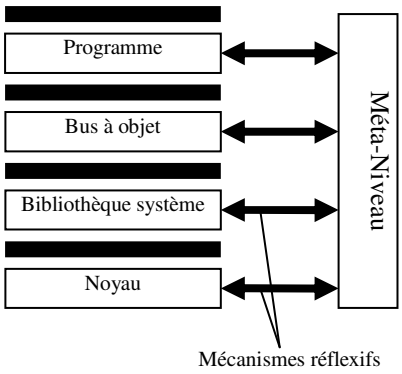


Figure 6 Réflexivité multi-niveaux

Cette technique a été employée pour traiter le problème du déterminisme. Le déterminisme d'un système est défini comme :

« La propriété de reproduire toujours le même résultat à une interaction à partir d'un état initial donné. »

Cette problématique est récurrente à la tolérance aux fautes et plus précisément lors de la réplication. Elle est majoritairement issue des problèmes d'ordonnancement des threads. Cependant, tous les threads n'introduisent pas du non-déterminisme. La réflexivité multi-niveaux permet alors de déterminer à l'exécution les ordonnancements susceptibles d'introduire du non-déterminisme pour les synchroniser lors de l'utilisation de la réplication active.

II.2.2 Réflexivité et implémentation

Il existe plusieurs sortes de réflexivité qui peuvent être classées selon l'étape du cycle de vie auquel la réflexivité intervient. Nous distinguons :

– La réflexivité au niveau des langages de programmation

– La réflexivité pendant l'exécution

Nous allons maintenant détailler chacun de ces points.

II.2.2.1 Langage réflexif

LISP [50] a été l'un des premier langage de programmation à introduire les concepts de réflexivité. Dans un langage réflexif, un logiciel peut être entendu comme un programme qui s'auto-modifie au fur et à mesure qu'il génère de nouvelles règles dictant le comportement que l'on attend de lui. On peut aujourd'hui classer les langages de programmation réflexifs selon qu'ils sont interprétés ou compilés.

Un programme écrit dans un langage réflexif interprété (ex : LISP, SmallTalk [51] ou Java [52] avec la librairie java.lang.reflect ou AspectJ [53] qui introduit la notion d'aspects pour le langage Java) s'exécute sur une plateforme d'exécution (un intergiciel) appelée l'interpréteur. La réflexivité du langage permet à l'interpréteur d'évaluer l'exécution du programme en réalisant des observations, pour ensuite exécuter son propre code machine. La réflexivité du langage permet donc, via l'interpréteur d'auto-modifier le programme source à l'exécution, favorisant son adaptabilité vis-à-vis de l'environnement. Toutefois, la réflexivité est alors obtenue au prix d'une certaine complexité.

Un programme écrit dans un langage réflexif compilé (ex. C++) est traduit en langage machine avant son exécution. Le programme en charge de cette transformation est un compilateur. Un langage de programmation réflexif compilé s'appuie sur un compilateur réflexif autrement appelé compilateur « ouvert ». Un exemple type est OpenC++ [18]. Ce compilateur réflexif possède une connaissance du langage qui lui permet de raisonner sur l'objet de la compilation (une classe), et d'interpréter cette compilation selon un méta-objet de compilation (une méta-classe). C'est la technique utilisé dans FRIENDS [12]. Le méta-niveau, composé des méta-classes, est alors un méta-programme.

II.2.2.2 Support à l'exécution

La réflexivité peut être introduite par l'environnement d'exécution. Cet environnement d'exécution est réflexif s'il fournit des mécanismes réflexifs pour contrôler son comportement à un méta-niveau.

FRIENDS [11] est un exemple complet d'utilisation de la réflexivité à la compilation avec un support à l'exécution. La réflexivité est utilisée de deux manières. Tout d'abord, à la compilation, elle permet de modifier le programme fonctionnel afin d'introduire des mécanismes réflexifs pour un support de la réflexivité à l'exécution. Ensuite, à l'exécution, lorsque la classe est instanciée en objets, les mécanismes réflexifs introduits permettent à un intergiciel de dynamiquement munir ces objets de méta-objets qui fournissent des services de tolérance aux fautes à l'exécution.

II.2.3 Architecture réflexive pour l'adaptation de la tolérance aux fautes

Dans cette section, nous nous intéressons à organiser le logiciel pour traiter le problème de l'adaptation de la tolérance aux fautes. Nous montrons comment la réflexivité peut être mise en place afin de séparer le traitement des différentes problématiques dans des couches différentes du système.

II.2.3.1 Dépendance des problématiques

Notre architecture réflexive doit permettre de séparer les traitements de trois problématiques distinctes qui sont, le traitement fonctionnel pour lequel le système est conçu, la tolérance aux fautes à appliquer au fonctionnel, l'adaptation de la tolérance aux fautes à l'exécution. L'objectif de l'architecture réflexive pour l'adaptation est donc d'isoler les traitements des différentes problématiques dans des méta-niveaux distincts (les couches de notre architecture).

Ces problématiques possèdent des dépendances. Ces dépendances sont illustrées par la figure 7. La tolérance aux fautes doit permettre de tolérer le modèle de fautes de l'application fonctionnelle. De plus, la tolérance aux fautes est directement concernée par notre problématique d'adaptation. Il est donc primordial pour maîtriser l'impact de l'adaptation de la tolérance aux fautes sur le fonctionnel de séparer ces deux problématiques dans des couches disjointes du système. Pour cela, nous utilisons une première fois la réflexivité où le niveau de base est le niveau fonctionnel, et le méta-niveau contient les traitements pour la tolérance aux fautes.

L'adaptation doit permettre de modifier les mécanismes de tolérance aux fautes. Toutefois, il ne peut être fait totalement abstraction du fonctionnel pendant l'adaptation. En effet, la tolérance aux fautes s'applique au fonctionnel. De ce fait, elle modifie son état et son exécution. Ainsi, pendant l'adaptation, le fonctionnel peut nécessiter certaines mise-à-jour, afin de pouvoir changer d'une stratégie à une autre. Nous souhaitons introduire le traitement de la problématique d'adaptation en utilisant la réflexivité. Le niveau de base est donc le système tolérant aux fautes qui est composé du fonctionnel et de la tolérance aux fautes. Le méta-niveau aura en charge la problématique de l'adaptation de la tolérance aux fautes.

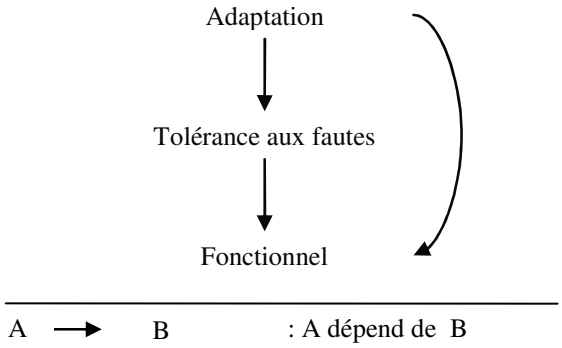


Figure 7 Dépendances entre les problématiques

II.2.3.2 Modèles récursifs

Pour tenir compte des dépendances entre les différentes problématiques du système, et permettre de séparer leur traitement, nous proposons d'utiliser la réflexivité de manière récursive.

Nous distinguons deux modèles récursifs organisés par couches ordonnées :

- Le modèle en couches superposées ;
- Le modèle en couches imbriquées.

Ces deux modèles sont détaillés ci-dessous.

Couches superposées

Dans le modèle en couches superposées (Figure 8), chaque méta-niveau utilise un modèle du méta-niveau inférieur. Ainsi, le méta-niveau *i* utilise le modèle du niveau *i-1*. Cette architecture limite les dépendances à la couche directement inférieure. Elle ne permet donc pas de respecter les dépendances entre les problématiques fonctionnelles, de tolérance aux fautes et de l'adaptation, comme explicité dans la section II.2.3.1.

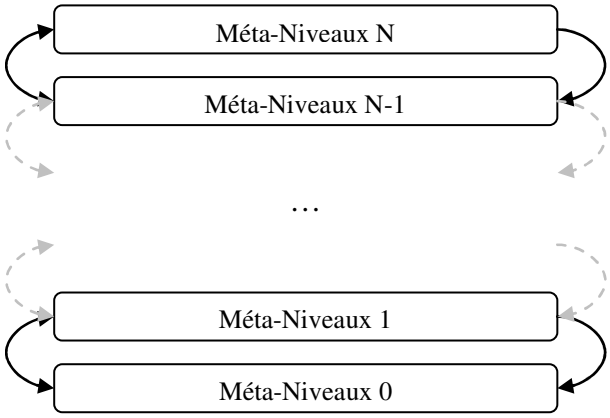


Figure 8 Récursivité en couches superposées

Couches imbriquées

Dans le modèle en couches imbriquées, chaque méta-niveau peut utiliser un modèle de tous les méta-niveaux inférieurs. Ainsi le méta-niveau *i* peut utiliser un modèle des méta-niveaux *i-1* à *0*. Ainsi, chaque méta-niveau peu dépendre de tous les méta-niveaux qui lui sont inférieurs.

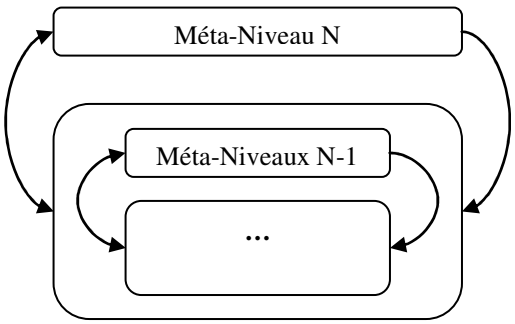


Figure 9 Récursivité en couches imbriquées

Ce modèle permet de respecter les dépendances entre le fonctionnel, la tolérance aux fautes et l'adaptation. Ce sera donc le modèle choisi pour notre architecture.

Un parallèle peut être fait entre la réflexivité multi-niveaux et la réflexivité utilisée de manière récursive avec le modèle en couches imbriquées. Dans la réflexivité multi-niveaux, le méta-niveau repose sur l'utilisation de modèles de toutes les couches du système, le rendant ainsi transverse au système dans sa globalité. Dans le modèle en couches imbriquées, le méta-niveau *i* est transverse à tous les méta-niveaux inférieurs. Ce méta-niveau repose donc sur des

modèles des méta-niveaux qui lui sont inférieurs. Ainsi, le modèle en couches imbriqués est équivalent à appliquer la réflexivité multi-niveaux aux couches réflexives d'un système.

II.2.3.3 Choix de l'architecture

L'architecture réflexive que nous utilisons pour traiter l'adaptation de la tolérance aux fautes est donc une architecture utilisant de la réflexivité de manière récursive. Nous utilisons le modèle en couches imbriquées qui permet de respecter les dépendances entre les différentes problématiques du système.

Cette architecture (Figure 10) possède donc trois méta-niveaux :

- Le niveau de base traite la problématique fonctionnelle ;
- Le méta-niveau 1 permet d'appliquer des mécanismes de tolérance aux fautes au système situé dans le niveau de base ;
- Le méta-niveau 2 traite l'adaptation de la tolérance aux fautes, et utilise donc une représentation du niveau de base et du méta-niveau 1.

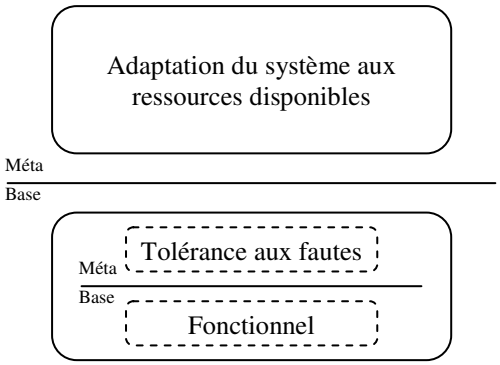


Figure 10 Architecture réflexive utilisée

Cette architecture permet de séparer les problématiques dans des couches distinctes du système. Toutefois, il est nécessaire de pouvoir modifier le logiciel contenu dans la couche de tolérance aux fautes et la couche fonctionnelle. Pour cela, nous proposons d'utiliser les technologies à composants. Nous allons voir dans la section suivante comment les outils apportés par les modèles à composant vont permettre de définir un modèle partiel du système tolérant les fautes pour l'adaptation.

II.3 MODELE A COMPOSANT ET CONCEPTION OUVERTE

II.3.1 Principes

Les composants facilitent la réutilisation et l'ingénierie des architectures dirigées par les modèles en proposant de découper le logiciel en entités s'échangeant des services. Ces entités sont appelées des composants. Un composant est constitué de :

- Un contenu qui est constitué de variables et d'instructions, donc en d'autres termes, il s'agit là d'un algorithme au sens informatique du terme ;
- Des interfaces qui permettent d'abstraire le contenu du composant aux services qu'il rend ou dont il dépend. Les interfaces peuvent être de natures différentes : appel de procédures, évènements, communication. Ces interfaces sont de deux types :
  - Fournies si le composant rend le service ;
  - Requises si le composant nécessite le service pour rendre un service correct. Dans certain modèle (par exemple OpenCOM [54]), la nomenclature UML[2] est utilisée. On parle alors de réceptacle ;
- D'autres composants. Le composant est alors qualifié de composite.

Le modèle à composant fournit donc une abstraction du logiciel comme une composition de boîtes, interconnectées. De notre point de vue, cette abstraction simplifie la modification du logiciel pour deux raisons. Premièrement, cette abstraction permet de confiner un algorithme et ses données qui sont susceptibles d'être modifiés dans un composant. Deuxièmement, elle permet de spécifier des interactions entre les composants aux travers de connexions à des services. La description de ces services étant indépendante de leur implémentation, il est possible de remplacer l'implémentation d'un service par une autre, pour répondre à des critères non-fonctionnels de ce service comme par exemple, sa consommation des ressources du système.

Pour fonctionner, un composant est créé, ses interfaces requises sont connectées aux interfaces fournies de composants existants dans le système, ce composant est ensuite démarré et finalement, d'autres composants sont connectés aux interfaces qu'il fournit. Ce même composant peut être retiré du système par les étapes inverses : les composants sont déconnectés des interfaces qu'il fournit, le composant est ensuite arrêté, puis ces interfaces requises sont débranchées, et enfin, le composant est détruit.

II.3.2 Réflexivité dans les modèles à composant

Un modèle à composant repose sur l'utilisation d'un support à l'exécution qui permet de réaliser les opérations de création et de destruction de composants ou de connexions. De notre point de vue, un modèle à composant est réflexif dans le sens où il fournit à la fois des mécanismes d'observations (introspection des composants et des connexions existantes), et des mécanismes d'intercession (ajout/suppression de composants et de connexions). Le modèle à composant fournit donc l'opportunité à un méta-niveau de manipuler le logiciel.

Nous allons maintenant tenter de détailler les mécanismes réflexifs fournis par les modèles à composant. Il existe cependant de nombreux modèles à composant qui proposent des

[2] Unified Modeling Language (http://www.uml.org/)

enrichissements du modèle basique, rajoutant ainsi des capacités réflexives. Citons par exemple les solutions industrielles, JavaBeans[3] qui introduit des composants sous Java et CCM[4] qui propose la notion de composant pour CORBA, ainsi que les solutions plus académiques OpenCOM [54] qui propose un modèle à composant embarqué extensible et Fractal [55] qui propose un modèle à composant très riche et extensible. Il est donc impossible de faire une liste exhaustive des mécanismes réflexifs fournis par les différents supports d'exécution des modèles à composant. Toutefois, nous allons détailler les principaux mécanismes réflexifs en essayant de montrer leurs atouts pour l'adaptation du logiciel.

Beaucoup de modèles à composant n'utilisent pas la notion de composants composites. De manière générale, les modèles à composants fournissent les mécanismes réflexifs élémentaires pour manipuler les composants. Ces mécanismes sont détaillés dans le tableau 1.

Tableau 1 **Mécanismes réflexifs communs aux modèles à composants**

Mécanisme	*Type de mécanisme réflexif*	*Détails*
Création	Intercession structurelle	Crée un composant logiciel en le chargeant en mémoire
Destruction	Intercession structurelle	Détruit un composant et le retire de la mémoire
Interfaces	Introspection	Identifie les interfaces (fournies ou requises) pour un composant donné
Connexion	Intercession structurelle	Relie une interface requise d'un composant préalablement créé à une interface fournie d'un composant existant
Déconnexion	Intercession structurelle	Supprime le lien entre les interfaces fournie et requise de deux composants
Démarrage	Intercession comportementale	Démarre le composant et le rend opérationnel
Arrêt	Intercession comportementale	Arrête un composant, et le rend non-opérationnel
État opérationnel	Introspection	Fournit l'état opérationnel d'un composant (démarré ou arrêté)
Architecture	Introspection	Identifie les composants présents et la manière dont ils sont connectés

Ces mécanismes réflexifs permettent de construire un modèle architectural pour la manipulation du logiciel. Ainsi, il est possible de dynamiquement créer ou supprimer des composants d'un programme. Toutefois, la modification du logiciel ne se résume pas à modifier une architecture de composants. En effet, les modifications peuvent ne pas être à tout moment réalisables. Pour ce faire, il est nécessaire d'obtenir des informations concernant leur exécution. Ces informations ne sont pas toujours fournies par les modèles à composants du commerce. Les modèles plus académiques, et plus particulièrement lorsqu'ils sont extensibles, permettent d'introduire des mécanismes réflexifs supplémentaires, qui pourront être utilisés lors de l'adaptation. Ces mécanismes sont détaillés dans le tableau 2.

Tableau 2 **Mécanismes réflexifs pour l'exécution**

Mécanisme	*Type de mécanisme réflexif*	*Détails*
Appel sur interface	Intercession comportementale	Permet d'appeler une méthode sur une interface fournie d'un composant
Interception d'appel	Réification	Informe d'un appel sur une interface fournie de manière spontanée (dans certains modèles comme fractal, les appels à partir d'interfaces requises sont aussi interceptés)

II.3.3 Extension du modèle à composant

Certains modèles à composant comme Fractal ou OpenCOM peuvent voir leur capacité réflexive étendue par l'utilisateur. Ainsi, il est possible d'enrichir le modèle en rajoutant des

[3] http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp
[4] Corba Component Model http://www.corba.org/

mécanismes réflexifs. Il existe deux manières d'étendre le modèle d'un modèle à composant : l'utilisation de composant dits « ouverts » et l'utilisation de composants composites [56].

II.3.3.1 Composants « ouverts »

Un composant « ouvert » est un composant qui rend visible une partie de son contenu à l'aide de mécanismes réflexifs. Ainsi, un composant ouvert peut fournir des informations sur son fonctionnement interne, qui ne sont pas directement observables dans le modèle à composant. Un composant ouvert introduit donc un modèle qui lui est propre. De ce fait, le composant n'est plus considéré comme une boîte noire. Les capacités réflexives dont il est muni en fait une boîte grise, dans le sens où une partie du contenu du composant est rendue visible à l'extérieur. Ainsi, dans le modèle à composant Fractal, les composants peuvent implémenter des interfaces de contrôle, permettant l'accès à des contrôleurs situés à l'intérieur du composant. Ces contrôleurs implémentent alors un modèle propre à un composant.

Ainsi, il est possible d'étendre le modèle d'un composant pour traiter des problématiques qui n'ont pas été directement intégrées au modèle initial.

II.3.3.2 Composants Composites

Un composant composite contient un sous ensemble de l'architecture à composant du logiciel. Ainsi, il peut introduire des modèles architecturaux ou comportementaux de ce sous ensemble. De ce fait, il permet d'introduire des modèles d'un sous ensemble de l'architecture.

Lorsque ces modèles introduisent des composants composites, des mécanismes réflexifs sont alors ajoutés (tableau 3).

Tableau 3 **Mécanismes réflexifs pour les composants composites**

Mécanisme	*Type de mécanisme réflexif*	*Détails*
Composition	Intercession structurelle	Ajoute un composant dans un composant composite
Décomposition	Intercession structurelle	Retire un composant d'un composant composite
Architecture	Introspection	Identifie et liste les composants contenus dans un composant composite

L'utilisation de ces composants composites est un moyen d'introduire un modèle dédié à un sous-ensemble de composants.

II.3.4 Étude de cas

Nous proposons de choisir un modèle à composant qui répond à trois critères.

Tout d'abord, ce modèle à composant doit fournir les capacités réflexives nécessaires pour le traitement de la problématique de l'adaptation. S'il ne possède pas directement ces capacités réflexives, il est nécessaire de pouvoir les étendre. Certains modèles à composant offrent l'opportunité de modifier l'architecture à composant d'un logiciel, pendant son exécution. Toutefois, afin de pouvoir appliquer les modifications sur le système dans un état adéquat de l'exécution, il est nécessaire d'avoir des mécanismes permettant d'observer et modifier cette exécution.

Ensuite, ce modèle à composant doit être compatible avec le modèle d'architecture choisie. Donc, il doit permettre d'introduire de la réflexivité de manière récursive suivant le modèle en couches imbriquées. Cela impose qu'il soit possible de réaliser des mécanismes de tolérance aux fautes à un méta-niveau. Donc le modèle doit permettre l'ajout de l'empreinte réflexive pour la tolérance aux fautes.

Enfin, le modèle à composant doit avoir un surcoût négligeable. En effet, une stratégie de tolérance aux fautes possède une taille logicielle réduite. Cela signifie que la granularité des composants est relativement fine, d'autant plus que ce logiciel est susceptible d'être morcelé pour faciliter sa manipulation à l'exécution. Un surcoût important introduit par le modèle à composant provoque alors une chute des performances du logiciel de tolérance aux fautes et n'est pas souhaité.

Deux modèles à composant possèdent le potentiel pour répondre à nos critères :

- Fractal [55]
- OpenCOM [54]

Nous les détaillons maintenant.

II.3.4.1 OpenCOM

OpenCOM est un modèle à composant dérivé du modèle à composant Microsoft COM[5] qui a été pensé pour le monde de l'embarqué. Il en existe deux versions. La deuxième version est une extension de la première qui introduit des concepts facilitant le déploiement d'une application sur un système hétérogène, avec des composants de nature différente. Nous présentons ici la version 2 de ce modèle qui est la plus récente. Il est applicable sur un vaste panel d'environnements matériels. Il ne dépend pas d'un domaine spécifique (multimédia, temps réel, …). Il possède des mécanismes réflexifs permettant la modification de l'architecture à composant du logiciel pendant son exécution ainsi que des moyens d'intercepter les appels sur les interfaces des composants. Il masque de manière sélective les spécificités de l'environnement de déploiement. Il fournit des mécanismes pour séparer les problématiques (programmation, déploiement, …). Il offre de bonnes performances pour des composants de taille très réduite.

L'approche architecturale employée consiste à définir un modèle à composant générique pour l'exécution, et de l'enrichir avec des notions de canevas autrement appelés *Component Framework* (CF). Un CF est un composant composite. Il permet de réaliser un modèle réflexif.

Le modèle à composant d'OpenCOM (Figure 11) est constitué de :

- *Composants* qui sont eux même constitués de :
  - *Interfaces* qui fournissent un service.
  - *Réceptacles* qui sont les interfaces requises d'un composant. Il existe cinq sortes de réceptacles :
    - simple : une seule interface connectée ;
    - multiple : plusieurs interfaces de même signature connectées et mises à la disposition du composant au travers d'une liste ;
    - multiple parallèle : plusieurs interfaces de même signature connectées et appelées en parallèle. L'appel est non bloquant et le type de retour des méthodes doit être muet (void). Le composant ne perçoit qu'une seule interface.
    - multiple contextuel : un contexte est ajouté au réceptacle à un méta-niveau. Lors d'un appel, le méta-niveau décide selon le contexte quelle est l'interface à appeler.

[5] http://www.microsoft.com/com/

- multiple contextuel parallèle : un mixte entre réceptacle parallèle et réceptacle avec contexte.

– *Capsules* correspond à l'intégralité de la plateforme attachée à une instance du noyau OpenCOM. Une capsule contient alors au moins un caplet, correspondant à celle contenant l'instance du noyau.

– *Caplets* qui sont des CFs et regroupent les composants traitant des problématiques proches

– *Loaders* qui fournissent différentes manières de charger les composants dans les caplets

– *Binders* qui fournissent plusieurs manières de connecter une interface à un réceptacle, à la fois au sein et au travers de différents types de caplets et différentes instances

Ces caplets, binders et loaders sont implémentés sous la forme de composant qui sont dynamiquement chargés dans des canevas facilitant le déploiement et le rendant transparent aux technologies matérielles et logicielles employées.

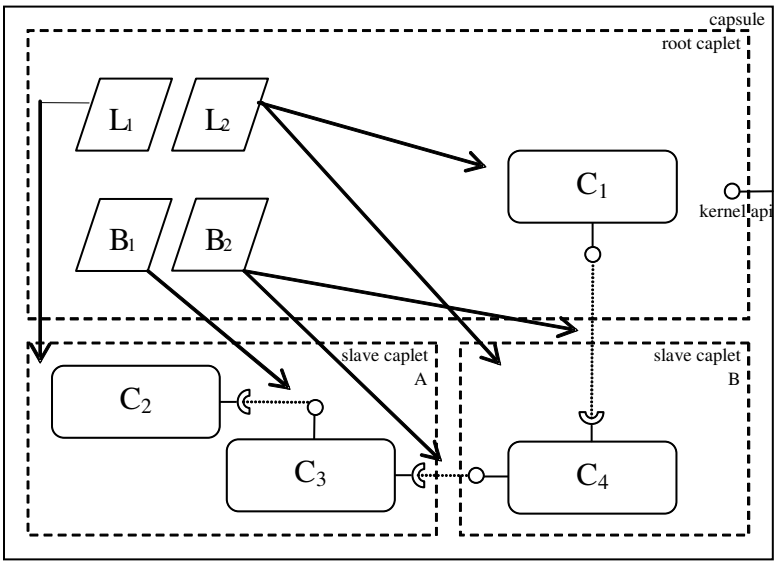


Figure 11 Modèle à composant OpenCOM

De manière graphique, les CFs sont représentés sous la forme de parallélogrammes, les caplets sont représentés par un rectangle en pointillés, les composants par un rectangle arrondi, les interfaces par des boules et les réceptacles par des arcs de cercle. La Figure 11 montre une capsule contenant trois caplets. Le caplet racine (*root caplet*) contient les *binders* et les *loaders*. Ce caplet est le premier créé dans l'architecture logicielle. Il possède un accès à l'interface du noyau d'OpenCOM. Dans notre exemple, le *loader* L_1 est utilisé pour charger les composants du caplet A et le *loader* L_2 ceux du caplet racine et B. Les connexions entre interfaces et les réceptacles sont réalisées par le *binder* B_1 à l'intérieur du caplet de gauche et par le *binder* B_2 entre les différents caplets.

Chaque composant est représenté par une interface *IUnknown* possédant une unique méthode *QueryInterface* permettant de récupérer un pointeur vers l'interface de ce composant. De plus, chaque composant implémente l'interface *IMetaInterface* permettant de connaître toutes les interfaces implémentées, et tous les réceptacles publics d'un composant. Un composant possédant des réceptacles implémente l'interface *IConnections* permettant la connexion et la déconnexion de ses réceptacles aux interfaces d'autres composants du système. L'interface

noyau d'OpenCOM est appelée *IOpenCOM* et possède, entre autres, les méthodes de création/destruction de composant et de connexion/déconnexion.

L'intergiciel OpenCOM fournit plusieurs modèles permettant de mettre en place l'adaptation d'un logiciel [57] :

- Méta Architecture : ce modèle représente l'architecture à composant sous la forme d'un graphe permettant de connaître l'agencement des composants et le modifier ;
- Méta-Interface : ce modèle fournit des informations sur les interfaces et réceptacles des composants. Il permet d'attacher des méta-informations aux interfaces, qui seront prise en compte pour déterminer le contexte lors d'un appel à partir d'un réceptacle contextuel.
- Méta-Interception : ce modèle permet d'être réifié des appels entrants sur un composant, en utilisant des mécanismes d'interception reposant sur des proxys d'interface.

La version 1 que nous avons utilisée n'introduit pas les concepts de *loaders*, *binders* ni *caplets*. Ces extensions du modèle ont été réalisées pour répondre au problème de déploiement des composants sur différentes plateformes matérielles et permettre la cohabitation de plusieurs technologies logicielles.

Ce modèle à composant permet d'ouvrir les composants. Les mécanismes réflexifs sont alors fournis par des interfaces implémentées par le composant. De plus, il est possible d'utiliser les mécanismes réflexifs sur les méta-interfaces d'un composant ouvert, ce qui permet d'introduire une certaine récursivité de la réflexivité.

Un exemple de réalisation d'un modèle à l'aide d'un composant composite (CF) est illustré à la figure 12. Un CF encapsule une partie de l'architecture à composant du logiciel. Il est possible de fournir un modèle spécifique de cette sous architecture pour capturer un fonctionnement spécifique connu a priori. Ainsi, ce composant composite est considéré comme un composant à part entière par le méta-niveau. Il peut alors fournir des services réflexifs au travers d'interfaces pour les composants de ce méta-niveau.

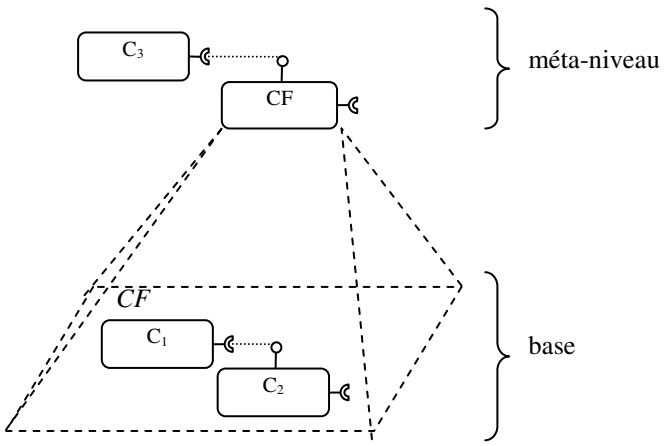


Figure 12 Réalisation d'un méta-niveau sous OpenCOM

II.3.4.2 Fractal

Fractal est un modèle à composant réflexif générique. Comme illustré à la figure 13, il se compose :

- de *composants* qui possèdent :
  - des *interfaces* de service. Une interface peut être :

- *serveur* (ou fournie) si le composant réalise le service décrit par cette interface
- *cliente* (ou requise) si le composant utilise le service décrit par cette interface et réalisé par un autre composant
- *externe* si elle est accessible de l'extérieur du composant
- *interne* si elle est accessible de l'intérieur du composant

• des *interfaces de contrôle* qui correspondent aux mécanismes réflexifs

• de *membranes* qui sont des composants composites. Ces membranes contiennent :

- les mécanismes de contrôles appelés *contrôleurs* qui utilisent les interfaces de contrôle fournies par les composants
- au moins un composant fonctionnel. D'un point de vue architectural, une membrane qui contient plusieurs composants est appelée *composant composite*.

– de *fabriques* en charge de créer les composants. Ces fabriques peuvent être :

• *standards* lorsqu'elles fabriquent une seule sorte de composant. Certaines fabriques standards sont appelées fabriques à *gabarit* (*templates*). Ces fabriques créent uniquement des composants qui répondent à un gabarit. Ce gabarit détaille les interfaces qui doivent être implémentées par le composant.

• *génériques* lorsqu'elles fabriquent plusieurs sortes de composant. La première fabrique générique créée et toujours disponible est appelée *bootstrap*. Cette fabrique doit être capable de créer les autres pour fabriquer les composants de l'application

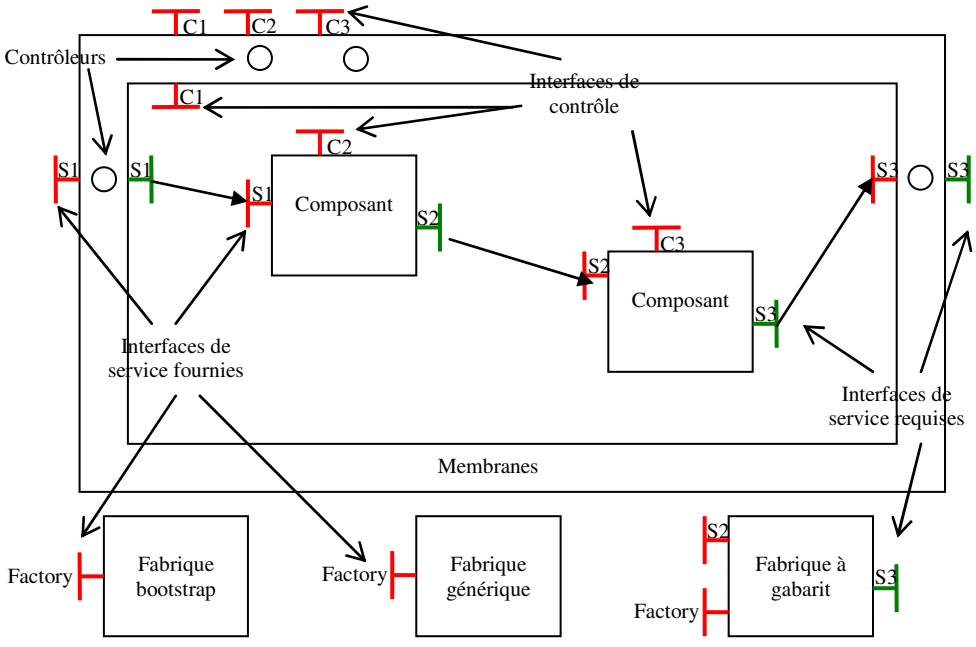


Figure 13 Modèle à composant Fractal

Fractal offre la possibilité d'étendre les mécanismes réflexifs par ajout de contrôleurs au niveau des membranes. De plus, il est possible d'ouvrir les composants par ajout d'interface de contrôle. En effet, toute interface d'un composant portant un nom se terminant par "-*controller*" est une interface de contrôle. Ainsi, il est possible d'ajouter les mécanismes

réflexifs que l'on souhaite voir présents. De ce fait, Fractal permet de réaliser un modèle pour la tolérance aux fautes comme pour l'adaptation.

Fractal ne définit pas la réflexivité de manière récursive dans ses spécifications. En effet, il n'est pas spécifié comment manipuler les contrôleurs. Dans AOKell [58] (Figure 14) qui est une implémentation de Fractal, les contrôleurs situés dans la membrane (i.e., l'implémentation du méta-niveau) sont des composants qui sont eux-mêmes dans une nouvelle membrane. Ces composants-contrôleurs sont agrégés aux composants de base par l'utilisation d'aspects (glue). Nous obtenons alors un modèle réflexif récursif avec couches superposées. Chaque couche réflexive est alors une couche de contrôle pour la couche directement inférieure. Pour passer à un modèle en couches imbriquées, il faut pouvoir créer une nouvelle membrane contenant les *N* méta-niveaux inférieurs, et réalisant les traitements de niveaux *N+1*.

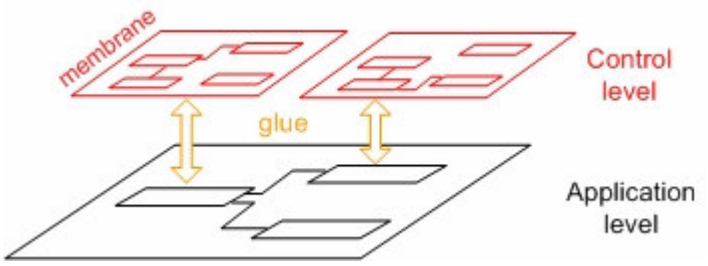


Figure 14 **Réflexivité récursive dans AOKell**

II.3.4.3 Conclusion

Ces deux modèles à composant introduisent des capacités réflexives dans le but de manipuler les composants en opération. Ces capacités permettent d'une part d'introspecter et modifier l'architecture à composant, et d'autre part, d'intercepter des interactions entre les composants. Ces modèles possèdent des concepts relativement similaires qui comprennent les composants ouverts, les modèles architecturaux, la gestion du cycle de vie. L'extension des capacités réflexives est possible dans chaque cas. Enfin, ils permettent tous deux de réaliser l'architecture réflexive imbriquée. Dans ces travaux, les aspects liés à l'implémentation seront réalisés à l'aide du modèle OpenCOM. Toutefois, il est tout à fait envisageables de les réaliser avec le modèle Fractal.

II.4 ARCHITECTURE REFLEXIVE POUR L'ADAPTATION

Les mécanismes réflexifs nécessaires à l'introduction de la tolérance aux fautes à un méta-niveau ont été dans un premier temps proposés dans les modèles à objets distribués de type ORB [11], puis complétés dans un deuxième temps dans les systèmes [49]. Dans un souci de simplification, nous utiliserons l'empreinte réflexive pour la tolérance aux fautes proposée dans [11], bien que les travaux sur les systèmes complexes ont montrés leur efficacité. Ainsi, nous nous focaliserons sur la couche applicative[6].

Dans [11], le modèle proposé pour la tolérance aux fautes rassemble les mécanismes réflexifs suivants :

- Intercession :
  - Appel de méthodes d'un objet de base
  - Sauvegarde de l'état d'un objet de base
- Introspection :
  - Chargement de l'état d'un objet de base
- Réification :
  - Réification des appels d'une méthode de l'objet de base
  - Réification des appels de méthodes par un objet de base sur un autre objet distant

Pour pouvoir adapter les mécanismes de tolérance aux fautes pendant l'exécution en utilisant les modèles à composants, ce modèle doit être lui-même un composant. Il fournit les mécanismes réflexifs nécessaires à la mise en œuvre des mécanismes de tolérance aux fautes à un méta-niveau. Ces mécanismes sont fournis au travers d'interfaces auxquelles les composants de tolérance aux fautes seront connectés. Nous appellerons ce composant l'*ApplicationController*. Comme nous le verrons par la suite, l'*ApplicationController* est un composant composite qui permet de wrapper les composants qu'il contient, et fournir un modèle comportemental aux composants de méta-niveau en charge de la tolérance aux fautes des composants wrappés.

II.4.1 Capture de l'état

La gestion de l'état est nécessaire pour beaucoup de stratégies de tolérance aux fautes. Sauvegarder un état correct pour un composant avant qu'une défaillance n'ait lieu est essentiel pour un recouvrement ultérieur, et peut être réalisé en utilisant des techniques à base de points de reprise. Le problème des points de reprise [59-61] est un problème réellement difficile dans des architectures logicielles complexes. Lorsque les points de reprise ne sont pas bien conçus, cela peut entraîner de sérieux problèmes et empêcher un recouvrement correct. Ce problème est connu sous le nom d'effet domino [62]. De plus, l'état d'un composant logiciel est une notion très complexe. L'état d'un composant est constitué de l'état de ses données internes, de l'état de la pile d'exécution, mais aussi de l'état des canaux ouverts, par exemple les fichiers ou les sockets. Donc une partie de cet état dépend des couches inférieures

[6]Nous considérons dans ce cas précis les couches intergicielles et système d'exploitation comme faisant partie du niveau applicatif.

du système (intergiciel, noyau). La capture d'un tel état peut être réalisée en utilisant des technologies réflexives multi-niveau [49] par exemple.

La gestion de l'état peut être réalisée de plusieurs manières. Une première manière consiste à déléguer les opérations de sauvegarde et de restauration de l'état à une tierce partie [20] qui peut être la plateforme d'exécution par exemple. Une seconde manière de procéder est d'utiliser l'héritage [21] et de déléguer le soin d'implémenter les opérations de sauvegarde et de restauration au concepteur de l'application. Une troisième manière consiste d'utiliser la réflexivité des langages (OpenC++ [63], AspectJ [53] ou la librairie java.lang.reflect de Java) pour insérer les opérations dans le logiciel. Dans notre cas, nous proposons d'utiliser la deuxième manière de procéder. Ainsi, tous les composants fonctionnels se voient équipés d'une interface *IState*, qui décrit les méthodes de sauvegarde et de restauration de l'état. L'implémentation de cette interface est alors de la responsabilité du concepteur du composant fonctionnel. Toutefois, cette solution n'est pas en contradiction avec l'utilisation des autres méthodes. Ainsi, le concepteur peut utiliser la réflexivité du langage pour implémenter les mécanismes de sauvegarde et restauration de l'état au niveau application.

II.4.2 Capture du comportement

La capture du comportement d'un composant fonctionnel est essentielle pour synchroniser les mécanismes de tolérance aux fautes avec le niveau de base de l'architecture réflexive. Ce comportement est observable au travers des interactions des interfaces et réceptacles des composants. Nous considérons deux sortes d'interactions. La première correspond à un appel sur l'interface d'un composant. Ces interactions sont appelées les *interactions entrantes*. La seconde correspond à un appel à partir d'un réceptacle d'un composant, vers l'interface d'un composant qui y est connectée. Ces interactions sont appelées les *interactions sortantes*. Dans l'objectif de capturer le comportement et d'appliquer les traitements relatifs à la tolérance aux fautes, ces interactions doivent être réifiées de manière synchrone à la couche de tolérance aux fautes.

Tableau 4 **Mécanismes réflexifs, définition des interfaces**

Nom de l'interface	*Méthodes*	*Détails*
IReifyIncomingInteractions	Object reifItfCall(String itfName, String mthName, Object[] args) throws Throwable	Réification d'une interaction entrante correspondant à l'appel de la méthode mthName sur l'interface itfName avec les arguments args. Cette méthode retourne le résultat de l'appel entrant ou une exception.
IReifyOutgoingInteractions	Object reifRcpCall(String itfName, String mthName, Object[] args) throws Throwable	Réification d'une interaction sortante correspondant à l'appel de la méthode mthName de l'interface itfName avec les arguments args. La méthode retourne le résultat de cet appel, ou une exception.
IIntercessionIncomingInteractions	Object interItfCall(String itfName, String mthName, Object[] args) throws Throwable	Intercession d'un appel de la méthode mthName sur l'interface itfName avec les arguments args. Cette méthode retourne le résultat de l'appel de la méthode sur le composant fonctionnel ou une exception.
IIntercessionOutgoingInteractions	Object interRcpCall(String rcpName, String mthName, Object[] args) throws Throwable	Intercession d'un appel de la méthode mthName à partir du réceptacle connecté à l'interface itfName avec les arguments args. Cette méthode retourne le résultat de l'appel de la méthode sur le composant fonctionnel ou une exception.

Nous avons introduit un composant composite appelé *ApplicationController* qui capture le comportement du composant fonctionnel qu'il contient. Ce composant composite joue le rôle

d'un *wrapper* (cf. figure 15). Il intercepte les interactions entrantes et sortantes de ce composant, et les réifie à la couche de tolérance aux fautes. Ainsi, un composant en charge de la tolérance aux fautes de C_2 peut être connecté aux interfaces et réceptacles de l'*ApplicationController* qui correspondent aux mécanismes réflexifs pour la tolérance aux fautes. Ces interfaces sont détaillées dans le tableau 4.

La figure 15 illustre ces mécanismes réflexifs et leur utilisation par un composant en charge de la tolérance aux fautes. Trois composants fonctionnels sont connectés. Nous nous focalisons sur l'application de mécanismes de tolérance aux fautes au composant C_2. Les interactions entrantes (1) sont réifiées au méta-niveau (2) puis traitées par le composant de tolérance aux fautes (3). Le composant de tolérance aux fautes peut alors appeler la méthode d'origine sur le composant fonctionnel (4) qui va réaliser les traitements relatifs à cet appel (5). Lors de ces traitements, des appels sortants peuvent être passés (6). Ils sont alors réifiés au méta niveau par l'*ApplicationController*. Le composant de tolérance aux fautes traite cet appel (7). Il peut alors demander le résultat de cet appel au niveau de base (8), ce qui provoque l'interaction sortante (9). Cette figure ne détaille pas le retour de ces appels.

Nous avons implémenté l'*ApplicationController* en utilisant OpenCOM sous Java.

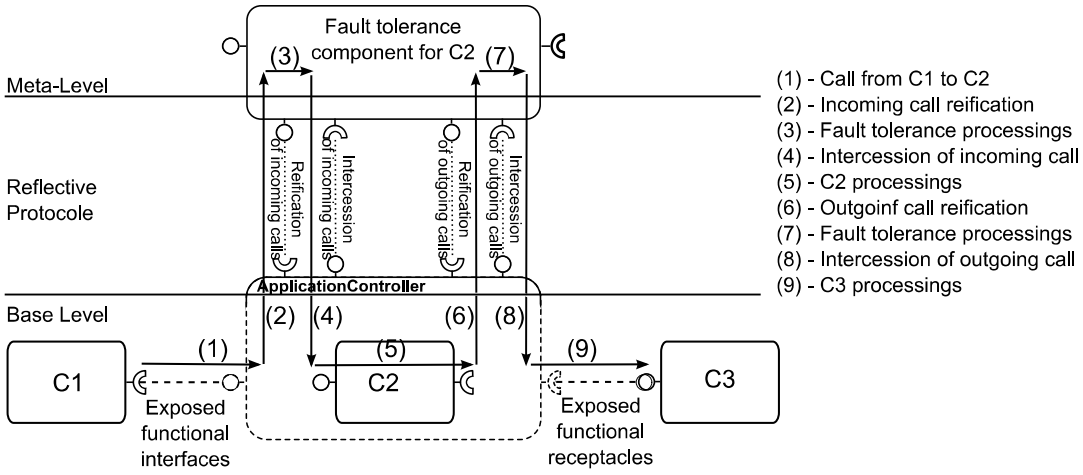


Figure 15 Chemin d'exécution d'un appel dans le modèle pour la tolérance aux fautes

II.5 CONCLUSION

Ce chapitre introduit les méthodes et les outils qui sont utilisés pour concevoir et réaliser l'architecture logicielle du système adaptable. Ces outils sont d'une part la réflexivité qui permet d'efficacement séparer des problématiques dans un système complexe et les modèles à composant qui fournissent des mécanismes permettant de modifier le code d'un logiciel, grâce à la manipulation de composants pendant son fonctionnement.

Parmi les modèles à composant, certains offrent des capacités réflexives qui facilitent la mise en œuvre du processus d'adaptation. Nous en distinguons deux : les modèles architecturaux qui permettent d'introspecter et modifier l'architecture à composant constitutive du logiciel et les modèles d'interception qui permettent d'intercepter les appels sur les composants et donc de mettre en place un modèle comportemental du logiciel pour son adaptation. Nous avons choisi d'utiliser OpenCOM pour implémenter notre prototype.

L'architecture mise en place repose sur trois couches logicielles distinctes implémentant les différents méta-niveaux du système. Ces couches reposent sur l'utilisation récursive de la réflexivité de manière imbriquée. La première couche contient les fonctionnalités du système, c'est-à-dire l'application. La deuxième couche permet de rendre cette application tolérante aux fautes. Elle nécessite de mettre en place un modèle de l'application pour la tolérance aux fautes. Ce modèle doit respecter une contrainte essentielle à l'utilisation des modèles à composants pour l'adaptation : le modèle doit être lui-même un composant. Ce modèle est mis en place au travers de l'utilisation d'un composant composite utilisé comme wrapper des composants applicatifs. Ce wrapper capture la dynamique des composants applicatifs au travers des interactions qu'ils réalisent avec les autres composants applicatifs du système. De plus, la capture de l'état du composant applicatif est introduite par une interface. Cela est rendu possible par l'utilisation de composants ouverts.

Cette architecture est une première étape au traitement de la problématique de l'adaptation de la tolérance aux fautes. Elle permet d'ores et déjà la manipulation du logiciel à l'exécution. Toutefois, aucune garantie n'est actuellement fournie sur le résultat de cette manipulation. Le prochain chapitre traitera plus particulièrement de cette problématique, ainsi que de l'automatisation des modifications du logiciel à partir d'une description haut-niveau.

Chapitre III

ADAPTATION DU LOGICIEL DE TOLERANCE AUX FAUTES

III.1 INTRODUCTION

Ce chapitre traite de la reconfiguration à l'exécution d'un logiciel distribué sur plusieurs nœuds fournissant un service de tolérance aux fautes pour un logiciel fonctionnel. Ce problème possède plusieurs dimensions. Tout d'abord, il est nécessaire de déterminer les modifications que le système doit subir pour atteindre la configuration souhaitée à partir d'une configuration déployée. En effet, il ne s'agit pas de faire table rase du système mais de modifier le système existant. Nous verrons que la conception orientée composant et l'utilisation d'un modèle à composant réflexif se prête assez bien à cet exercice. Ensuite, ces modifications doivent être appliquées au système alors même qu'il fournit son service. Le processus de reconfiguration va donc nécessairement avoir des conséquences sur l'exécution du système. Nous verrons comment maîtriser ces conséquences en apportant certaines garanties quant au respect des spécifications des configurations avant et après la reconfiguration. Enfin, après reconfiguration, le système contient des éléments réutilisés, et des éléments nouvellement introduits. L'état des éléments réutilisés a évolué pendant la vie opérationnelle du système. Il est donc nécessaire d'initialiser les composants nouvellement insérés avec un état qui tient compte du vécu des autres éléments du système.

Nous proposons une solution reposant à la fois sur une méthode de conception et sur des outils génériques permettant de modifier le logiciel courant afin d'obtenir la configuration spécifiée lorsque la décision d'adapter le système est prise. Ainsi, lors de l'adaptation, chaque réplique est susceptible d'être modifiée. Nous étudierons dans ce chapitre les problèmes que ces modifications entraînent, et nous proposerons une solution exprimée sous la forme de modèles, qui seront la base de la définition du modèle pour l'adaptation.

Ce chapitre est organisé de la manière suivante. La première section détaille les propriétés souhaitées pendant l'adaptation et leur expression en termes de problèmes liés à la modification du logiciel distribué. La deuxième section explique comment déterminer automatiquement les modifications que doit subir le logiciel afin d'être placé dans la configuration souhaitée, à partir d'une configuration a priori quelconque. La troisième section introduit un modèle comportemental de l'application distribuée permettant de placer le système dans un état de l'exécution permettant l'application des modifications déterminées dans la section précédente.

III.2 ADAPTATION D'UN LOGICIEL DISTRIBUE

Considérer le logiciel de tolérance aux fautes de manière réaliste consiste à le considérer comme un logiciel distribué, multithreadé, possédant des exécutions parallèles pouvant d'une part s'échanger des messages, et d'autre part nécessiter des sections critiques.

Nous cherchons à modifier le logiciel de tolérance aux fautes d'un système distribué critique alors que ce système fonctionne, et fournit un service. Cela nécessite de garantir que le processus d'adaptation respecte des propriétés de sûreté de fonctionnement. Ainsi, l'adaptation doit être sûre dans le sens où soit elle aboutit à un fonctionnement correct sans introduire d'erreur, soit elle n'est pas réalisée. Nous supposons dans ces travaux que la configuration future n'est pas connue en avance, c'est-à-dire, que le système ne sait a priori pas vers quelle configuration il doit être adapté avant qu'une décision ne soit explicitement prise. Lorsque cette décision est prise, une gâchette déclenche l'adaptation vers la configuration spécifiée par la prise de décision.

Ce problème possède trois dimensions :

- La capture du contexte consiste à superviser le système et son environnement pour pouvoir déterminer le contexte opérationnel du système.
- La prise de décision est réalisée à partir du contexte opérationnel. Son objectif est de déterminer si la configuration courante est justifiée dans le contexte courant, ou s'il est nécessaire de changer de configuration. Dans ce dernier cas, le résultat d'une prise de décision est une gâchette déclenchant l'adaptation du logiciel.
- La reconfiguration est le cœur du processus d'adaptation. Elle consiste à répondre à un ordre d'adaptation, représenté par la gâchette, en modifiant le logiciel courant pour le placer dans la configuration spécifiée.

Dans les travaux présentés ici, nous nous focalisons sur le troisième aspect du problème, c'est-à-dire la reconfiguration du logiciel de tolérance aux fautes pendant son exécution.

L'adaptation d'un unique composant ou paramètre est le cas le plus souvent traité dans la littérature. Par exemple, le remplacement d'un plugin par un autre a été très utilisé dans le monde des protocoles et des flux vidéo et audio. Il permet d'adapter l'algorithme utilisé aux ressources réseau d'un système distribué ou alors de l'infrastructure de ce réseau [64]. L'adaptation du composant nécessite de placer le composant dans un état particulier appelé état de quiescence (cf. III.2.2.4). Dans le cas où l'adaptation impacte plusieurs composants, ces composants sont alors souvent considérés comme indépendants durant le processus d'adaptation. Ainsi, le problème peut être ramené au cas précédent.

Nous voyons deux limitations à cette manière de procéder. La première concerne l'hypothèse d'isolation. Il est entendu qu'une condition nécessaire à la modification d'un composant est son isolation. Toutefois, nous sommes en droit de nous demander si celle-ci est suffisante. En effet, quelles sont les garanties que la modification d'un composant sous l'hypothèse d'isolation n'introduise pas d'erreur dans le système. La deuxième concerne l'hypothèse d'indépendance. D'une part, les composants interagissent dans le système. Ainsi, leurs états respectifs dépendent de ces interactions. La modification des composants sans comprendre comment elle impacte l'état et le comportement du système semble relativement hasardeuse. Ainsi, quelle confiance peut-on placer dans le système après adaptation ? Cela est d'autant plus vrai que ces composants fournissent des services de tolérance aux fautes, ou des services fonctionnels critiques. D'autre part, une configuration spécifie les architectures à composants

déployées et les interactions entre ces composants. L'hypothèse d'indépendance pendant l'adaptation peut amener à appliquer au système des configurations non spécifiées. Dans un système critique, nous sommes en droit de nous poser la question de la confiance que l'on peut placer dans ces configurations transitoires.

Dans les travaux de cette thèse ciblant l'adaptation de mécanismes de tolérance aux fautes, il est important d'assurer des propriétés de correction du processus de reconfiguration. De notre point de vue, la correction du processus d'adaptation est le fait que le processus d'adaptation permet de transformer un système respectant une spécification relative à la configuration déployée en un système respectant une nouvelle spécification relative à la configuration à mettre en place lors de l'adaptation sans introduire de comportements imprévus par les spécifications. La reconfiguration englobe des modifications paramétriques et architecturales (au sens des modèles à composant) du système. La modification de l'architecture à composant consiste à supprimer et créer un nombre quelconque de composants ou de connexions.

Bien souvent, les travaux sur l'adaptation se limitent aux spécifications architecturales. Du point de vue de la sûreté de fonctionnement, il semble que cela soit trop limitatif. Au travers des connexions entre les composants s'expriment d'une part les dépendances architecturales, mais aussi des dépendances comportementales qui se traduisent au niveau du flux de l'exécution et de l'évolution de l'état. Donc, de notre point de vue, une configuration possède trois niveaux de description : l'architecture, l'état et l'exécution. Au niveau de l'état des composants, deux composants peuvent posséder des données internes dont les valeurs sont liées par une relation logique. Ainsi, les relations logiques de la configuration à mettre en place doivent être vérifiées à la fin du processus de reconfiguration. Au niveau de l'exécution du système, certains traitements doivent être réalisés de manière transactionnelle pour maintenir leur cohérence pendant l'adaptation. Donc, afin de garantir le processus d'adaptation, plusieurs problèmes doivent être traités :

- Déterminer les modifications qu'une architecture à composant existante doit subir pour obtenir l'architecture à composant correspondante à la configuration souhaitée ;
- Placer le système dans un état où les modifications peuvent être appliquées sans introduire d'erreurs d'exécution ;
- Mettre les composants fonctionnels et de tolérance aux fautes dans un état de reprise après adaptation, permettant de respecter les relations logiques reliant l'état des différents composants de l'architecture ;
- Tolérer les fautes pendant le processus d'adaptation, ce qui nécessite de les définir, de les détecter et de les recouvrer.

Ces quatre problèmes vont maintenant être détaillés.

III.2.1 Détermination des modifications logicielles

Le premier pas vers l'adaptation du logiciel est de déterminer quelles modifications doivent être effectuées pour atteindre la configuration attendue. Nous nous plaçons dans le monde des composants logiciels. Ces composants sont rendus manipulables au travers d'un modèle à composant fournissant une représentation de l'architecture pendant l'exécution. Le logiciel est un ensemble d'algorithmes et de variables, découpé en composants qui sont interconnectés. Ces variables peuvent être classées en deux catégories qui sont d'une part les données qui sont manipulées par l'algorithme, et d'autre part les paramètres qui sont des constantes pour l'algorithme et qui déterminent pour un cas précis le comportement de l'algorithme.

La configuration du logiciel possède alors deux dimensions. La première dimension d'une configuration est l'agencement des composants entre eux. Cet agencement est appelé une architecture à composants. Elle représente l'ensemble des services disponibles dans le logiciel, leur implémentation ainsi que leurs dépendances. La deuxième dimension concerne les paramètres de chacun des composants logiciels. Ainsi, chaque composant possède un ensemble de paramètres. Ces paramètres prennent leur valeur dans un espace défini de valeurs correctes. Un paramétrage d'un composant est donc représentable par un vecteur dans l'espace des valeurs correctes des paramètres.

Changer de configuration nécessite d'introduire une description de sa spécification qui soit interprétable au méta-niveau adaptation. À partir de cette description, le système doit être capable de déterminer les modifications à réaliser sur l'architecture à composants, ainsi que sur les paramètres des composants. Cette description se décompose donc en une partie relative à l'architecture à composants et une partie paramètres des composants.

III.2.1.1 Modification d'une architecture à composants

Nous considérons que le modèle architectural fournit les services décrits dans le tableau 1 et le tableau 3. Ainsi, à tout moment, l'architecture à composant est connue. De plus, il est possible de la modifier, en ajoutant, supprimant, insérant, ôtant des composants ou en les connectant et déconnectant. Nous considérons que ces opérations sont les opérations élémentaires qui peuvent être appliquées pour modifier une architecture à composants.

Nous cherchons à exprimer la modification de l'architecture à composant sous la forme d'opérations élémentaires à appliquer, via le modèle, à l'architecture. Ces opérations sont partiellement ordonnées. En effet, il est nécessaire de déconnecter un composant avant de le supprimer de l'architecture, de même qu'il est obligatoire de déconnecter un réceptacle d'un composant avant de le reconnecter, surtout si ce réceptacle n'admet qu'une seul connexion.

Le problème de la modification de l'architecture à composant peut s'énoncer de la manière suivante : étant donné la description d'une architecture à composant dans laquelle doit être placé le système, quelles sont les modifications, et leur ordre d'application, à réaliser au travers du modèle à composant pour transformer l'architecture courante, et obtenir l'architecture spécifiée.

Il est donc nécessaire d'introduire une méthodologie permettant de déterminer dynamiquement les modifications à appliquer sur le système. Nous le verrons par la suite, la méthode que nous proposons repose sur deux outils. Le premier est un langage de description de l'architecture à composant, i.e. un ADL (*Architecture Description Language*). Le deuxième est l'utilisation de règles de transformation entre la description d'une architecture déployée et la description d'une architecture à mettre en place. Ces règles permettent de déterminer les opérations qui doivent être appliquées sur le modèle et leur ordre afin d'obtenir l'architecture à composant souhaitée.

III.2.1.2 Paramétrage d'un composant

Un **paramètre** est défini comme : *« une* variable susceptible de recevoir une valeur constante pour un cas déterminé et qui désigne certains coefficients ou certaines quantités en fonction desquels on veut exprimer une proposition ou les solutions d'un système d'équations » (d'apr. Bouvier-George Math. 1979).

L'utilisation de paramètres dans un modèle à composant permet de capitaliser l'algorithmique d'un composant pour le faire correspondre à des besoins identifiés lors d'une phase de configuration appelée le paramétrage.

Ainsi, un composant de communication sur Ethernet n'est pas uniquement dédié à envoyer ou recevoir et envoyer des données d'une unique machine sur un port prédéterminé. Le composant est paramétrable pour permettre l'échange de données avec n'importe quelle machine identifiée par les paramètres correspondants (le nom et le port de la machine cible par exemple).

OpenCOM n'introduit pas la notion de paramètres pour les composants de manière générique. En effet, les composants ne sont pas paramétrables dans le sens où aucun mécanisme ne permet la modification de variables internes au composant. Dans Fractal, le paramétrage est rendu possible au travers d'une interface de contrôle définissant les méthodes pour lire et modifier la valeur de chacun des paramètres dont le contrôleur a la charge.

Nous nous intéressons à l'analyse de ce problème. Le but est de mettre en place un modèle du composant permettant son paramétrage. Le processus d'adaptation peut nécessiter deux types de mécanismes :

- l'introspection des paramètres : lors de l'adaptation, il est nécessaire de connaître la configuration actuelle pour savoir dans quelle mesure celle-ci va être modifiée. Ce mécanisme doit donc permettre d'obtenir dynamiquement une connaissance des paramètres du système ;
- l'intercession des paramètres : le but du paramétrage est de modifier la valeur des paramètres. Il est donc nécessaire de permettre cette modification.

Nous avons identifié trois problèmes majeurs à la modification des paramètres à l'exécution.

Le problème de la corrélation : L'ensemble des vecteurs de paramètres étant prédéfini dans la spécification du composant, la modification d'un unique paramètre peut, de manière transitoire, amener à un vecteur de paramètres invalide vis-à-vis des spécifications, entraînant un comportement erroné du composant. La figure 16 illustre un scénario de reconfiguration par paramétrage d'un composant. L'objectif est de modifier la configuration 1 pour placer le composant dans la configuration 2. On remarque alors que la modification des paramètres amène le composant dans des configurations transitoires. Dans cet exemple, la configuration sort de l'ensemble des configurations correctes. C'est le problème de **cohérence** qui découle de la corrélation pendant l'application des valeurs aux paramètres. Le problème de corrélation est étroitement lié à la conception du composant et des configurations correctes.

Une manière de pallier cette situation est de réaliser le changement de paramètres de manière **atomique**. Ainsi, les deux paramètres ne sont plus modifiés indépendamment. Le paramétrage consiste alors à prendre en compte un nouveau vecteur de paramètres dans son intégralité.

Le problème d'isolation : Un paramétrage ne peut voir aucun autre paramétrage en cours. En effet, l'accès concurrent aux différents paramètres peut amener le composant à se trouver dans une configuration incorrecte. En d'autres termes, il est nécessaire que les modifications apportées par un paramétrage soient visibles uniquement lorsque toutes les opérations liées à ce paramétrage sont effectuées et validées.

Le problème de durabilité : Le paramétrage d'un composant peut ne pas être réalisable car amenant à une configuration incorrecte du composant. Il est alors nécessaire de placer le composant dans une configuration connue et correcte, ceci afin de maîtriser toute défaillance issue d'une mauvaise configuration du composant vis-à-vis de la spécification du composant. C'est la propriété de durabilité. Lorsqu'un paramétrage est achevé, le composant est dans un état stable durable, qui est la nouvelle configuration lorsque le paramétrage réussit ou l'ancienne configuration en cas d'échec du paramétrage. Cette propriété est intimement liée à la propriété d'intégrité du composant.

On remarque donc que le paramétrage d'un composant s'apparente à une transaction, car nécessite les règles ACID caractéristiques des transactions. On peut remarquer que Fractal ne tient actuellement pas compte de l'aspect transactionnel dans son modèle pour le paramétrage.

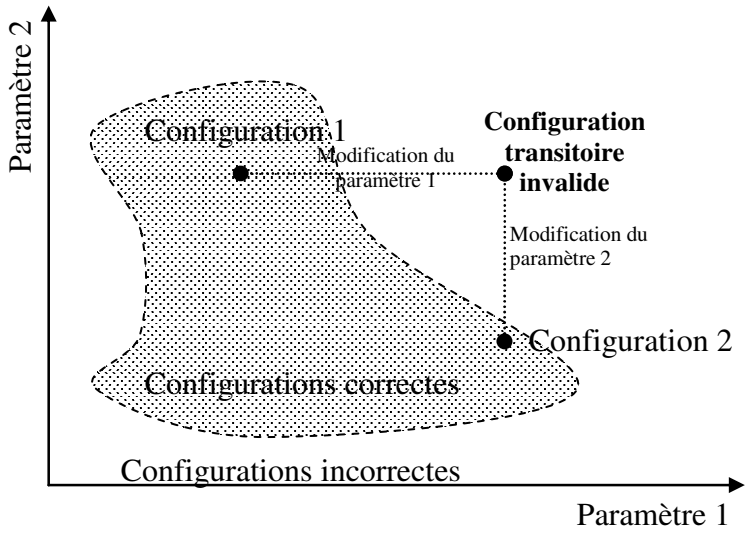


Figure 16 Paramétrage et corrélation

III.2.2 Exécution et adaptation

L'adaptation du système a lieu pendant son exécution, alors même que celui-ci délivre son service. Il est donc nécessaire de s'intéresser à l'impact de l'exécution du système sur l'adaptation et vice-versa. Nous allons montrer les contraintes imposées par l'exécution qui doivent être respectées pendant la modification du logiciel.

Lorsque l'on souhaite tenir compte de l'exécution du système pendant la reconfiguration, nous devons résoudre trois problèmes :

- L'observabilité des interactions entre les composants du système, pour pouvoir prendre en compte l'exécution du système pendant l'adaptation ;
- L'isolation des parties modifiées du système. Nous avons vu dans l'introduction que c'était une condition nécessaire pour pouvoir modifier le système. Nous allons en expliquer les raisons ;
- Le respect des spécifications. Une configuration spécifie des comportements attendus en termes d'exécution. Pendant l'adaptation, nous souhaitons garantir qu'aucun comportement non spécifié ne sera introduit.

L'état de l'exécution du système permettant l'adaptation est appelé état de quiescence ou état adaptable. Il garantit les propriétés d'isolation et la mise en place de la nouvelle configuration dans le respect des spécifications des configurations avant et après adaptation.

Nous allons maintenant détailler ces trois problèmes et détailler la notion d'état de quiescence.

III.2.2.1 Problème d'observabilité

Dans le monde des composants, les interactions entre les composants sont sensées être réalisées au travers des interfaces de ceux-ci. Toutefois, lorsque ce monde à composant repose sur des couches inférieures du système, on est en droit de remettre en question cette assertion.

Ainsi, chaque composant peut librement accéder à des services des couches inférieures du système qui sont non-observables dans le monde des composants. Dans la figure 17, le composant C1 peut appeler une méthode au travers de la connexion sur l'interface I1 du composant C2. Cet appel est réalisé dans le monde des composants et est donc observable au travers du modèle à composant. On parle alors de **dépendance explicite** des composants. Toutefois, C1 peut utiliser des services du système pour interagir avec C2, en envoyant des messages, des évènements, des signaux ou en partageant de la mémoire. Ces interactions sortent du cadre du modèle à composant, et de ce fait, ne sont pas observables dans le monde des composants. On parle alors de **dépendance implicite**. Afin de pouvoir maîtriser l'impact du processus d'adaptation sur l'exécution du système, il est nécessaire de rendre observable ces interactions.

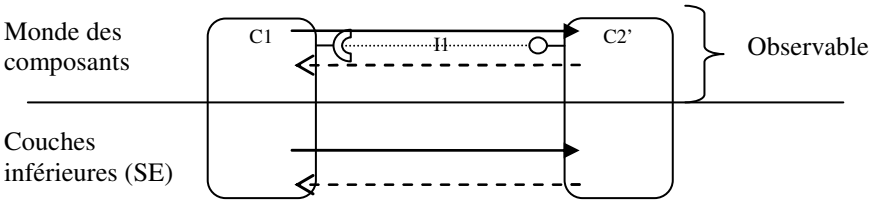


Figure 17 Observabilité dans le monde des composants

III.2.2.2 Isolation et accès concurrent pendant l'adaptation

L'ensemble des connexions et composants qui doivent être modifiés pendant l'adaptation forme une sous-architecture à composant dans l'architecture globale du système. Le processus de reconfiguration du logiciel se trouve en compétition avec les tâches du système pour accéder à la sous architecture à modifier.

Cette sous-architecture doit donc être considérée comme une ressource partagée. Il est nécessaire de mettre en œuvre une section critique pour l'adaptation, permettant de gérer l'accès concurrent du processus d'adaptation et des tâches du système à cette ressource partagée. Ainsi, cette section critique garantit l'exclusion mutuelle des tâches d'adaptation et des tâches du système au niveau de la sous-architecture pour son adaptation.

III.2.2.3 Respect des spécifications

De notre point de vue, il existe deux niveaux de description pour une configuration. Le premier niveau est statique. Il correspond à la description de l'architecture à composants en termes de composants présents et connexions entre ces composants. Le deuxième niveau est dynamique. Il correspond au comportement attendu du système, en termes d'interactions entre les composants et leurs agencements dans le temps, si la configuration est déployée sur un système vierge, i.e. sans aucune configuration précédemment déployée. En d'autres termes, il s'agit d'une description de l'exécution attendue de l'architecture à composants. Lors de l'adaptation, les changements apportés à l'architecture à composant modifient son exécution. Dans les systèmes critiques, il est nécessaire de maîtriser ces modifications comportementales, afin de ne pas introduire de comportements dangereux, susceptibles d'introduire des erreurs dans le système. Avant adaptation, le système est supposé se comporter selon les spécifications de la configuration déployée. Après adaptation, le système doit se comporter selon les spécifications de la nouvelle configuration qui a été mise en place. La difficulté réside alors dans la définition d'un comportement attendu pendant le processus de reconfiguration, i.e. lorsque le système est en transit de l'ancienne vers la nouvelle configuration.

L'état courant de l'exécution a été atteint après l'exécution dans une autre configuration. Ainsi, le passé de l'exécution dépend des traitements effectués dans la configuration avant adaptation. Après adaptation, le système poursuit l'exécution à partir de l'état atteint dans l'ancienne configuration. Cela pose plusieurs problèmes :

- Est-ce que cet état existe dans la nouvelle configuration ? En effet, si cet état d'exécution n'existe pas dans la nouvelle configuration, il est impossible de poursuivre l'exécution.
- Est-ce que cet état permet une poursuite de l'exécution correcte ? Si cet état existe, faut il encore que la poursuite de l'exécution soit correcte. Par exemple, dans une stratégie de tolérance aux fautes, lors du remplacement des composants assurant la synchronisation des répliques, l'état de l'exécution de ces répliques peut être désynchronisé si un message est en transit. Cet état peut exister dans la configuration avant et après adaptation. De ce point de vue, l'état d'exécution existe. Toutefois, la poursuite de l'exécution dans la nouvelle configuration est la plupart du temps incorrecte, car les messages envoyés dans l'ancienne configuration sont incohérents dans la nouvelle configuration.

Dans ces travaux nous choisissons de nous intéresser à un logiciel contenant un nombre de tâches fini. Chaque tâche est bornée dans le temps, et peut être exécutée une ou plusieurs fois. Nous considérons que les spécifications de l'exécution d'une tâche sont décrites par un flot de contrôle. Ce flot de contrôle détermine les traitements qui seront réalisés par la tâche lors de chaque cycle, ainsi que l'ordre dans lequel ces traitements seront réalisés. Ces traitements correspondent d'une part aux interactions réalisées dans le monde des composants, et d'autre part, à celles qui ont lieu en dehors de ce monde, par exemple, l'échange de signaux ou messages au travers des services du noyau (cf. III.2.2.1). Ainsi, sur un cycle, une tâche respecte ses spécifications si et seulement si la trace de son exécution, composée des observations des traitements que la tâche a réalisés, respecte le flot de contrôle spécifié dans les spécifications de la configuration.

Ce modèle de tâche possède deux intérêts majeurs :

- Il existe toujours au moins un état d'exécution atteignable compatible avec une configuration future (i.e. l'état d'arrêt)
- La spécification décrit le comportement sur un cycle, et ce comportement est répétable. Ainsi, si deux configurations réalisent les mêmes traitements au début du cycle, les états d'exécution entre ces traitements sont des états compatibles pour l'adaptation, du moins localement à la tâche.

Nous considérons un logiciel distribué complexe. Le système possède plusieurs tâches. Ces tâches sont soumises à des dépendances liées à des ressources partagées et à des communications. Ces dépendances possèdent deux effets sur l'exécution :

- **L'exclusion mutuelle :** lorsque deux tâches ne peuvent exécuter simultanément une partie de leurs traitements, c'est qu'elles sont en exclusion mutuelle. Cette technique est utilisée pour définir des sections critiques pour l'accès aux ressources partagées, i.e. des zones du code qu'au plus une tâche peut parcourir à la fois.
- **L'antériorité :** L'antériorité entre deux tâches est le fait que, par conception, un évènement sera observé sur une tâche avant qu'un autre évènement le soit. Cette antériorité existe par exemple lors de l'échange de messages, ou de signaux entre les tâches.

Lors de l'adaptation d'un tel système, nous ne souhaitons pas introduire de comportements non-prévus a priori. Ainsi, nous nous imposons de toujours respecter au moins une

spécification pour le comportement du système. C'est la propriété de **validité du comportement**. Ainsi, nous souhaitons garantir que :

- Avant adaptation, le comportement du système respecte les spécifications de la configuration avant adaptation
- Après adaptation, le comportement du système respecte les spécifications de la configuration après adaptation
- Pendant l'adaptation, le comportement du système respecte, soit les spécifications de la configuration avant adaptation, soit celles de la configuration après adaptation, mais n'introduit pas de comportements non spécifiés.

S'il y a antériorité entre deux évènements de deux tâches, c'est qu'il peut exister un lien causal entre ces deux évènements. Ce lien causal fait partie des spécifications. Sans informations supplémentaires sur la causalité, il est nécessaire d'appliquer le principe de précaution. Ainsi, lors de la reconfiguration, si l'observation du premier évènement a lieu dans une configuration, alors, l'observation et les traitements du deuxième doivent avoir lieu dans la même configuration, sous peine d'entraîner des incohérences au niveau de l'état et des données, ou même, au niveau du flux de contrôle. Ce problème consiste alors à rechercher les cycles d'exécution des tâches compatibles avec l'adaptation souhaitée.

III.2.2.4 État de quiescence

La notion d'état de quiescence a été introduite dans [65] dans le but de modéliser le comportement du système où le comportement d'un nœud était spécifié par un invariant quiescent : un état stable caractérisé par un invariant local respecté par le nœud. Ces travaux ont donné l'idée d'utiliser cet état de quiescence pour réaliser des opération de maintenance à chaud dans un système distribué de type base de données [66]. La notion d'état de quiescence est alors introduite de la manière suivante. Un nœud est dit quiescent si :

- Il n'est pas dans une transaction qu'il a commencée,
- Il n'a plus d'autres transactions à réaliser
- Il ne réalise lui-même aucune transaction
- Les autres nœuds ne commenceront pas de nouvelles transactions impliquant le service fourni par ce nœud.

Ces travaux ont été la base d'une réflexion et d'une extension de la notion de quiescence pour un composant, en conservant le fait que l'invariant considéré est local au composant. Dans un état de quiescence, un composant est à la fois dans un état cohérent et gelé. Il peut donc être modifié.

Toutefois, nous nous plaçons dans le cas où plusieurs composants doivent être modifiés, et ceux de manière distribuée. Ces composants possèdent des dépendances d'une part architecturales, d'autre part d'exécution. Ainsi cette notion d'état de quiescence doit être étendue à un état de quiescence pour une architecture complexe qui doit être modifiée, et donc, à un invariant global du système. Dans ces travaux, nous préférons parler d'état adaptable du système, c'est-à-dire état qui permet la modification d'une sous architecture à composants du système en respectant un invariant global. Dans nos travaux, l'invariant global est le respect des spécifications du système.

Un état adaptable d'un système dans une configuration *C*, en vue de son adaptation vers une configuration *C'*, est l'état de l'exécution respectant les deux points précédents, à savoir un état respectant la propriété d'exclusion mutuelle entre les tâches du système et les tâches

d'adaptation, et un état permettant de respecter les spécifications de l'exécution lors de la modification architecturale.

Il est donc nécessaire d'introduire des mécanismes permettant la recherche de cet état adaptable de l'architecture dans le but de la modifier. L'état adaptable peut alors être soit déterminé en ligne, soit calculé hors ligne et fourni au processus d'adaptation.

III.2.3 État de reprise

Comme nous l'avons détaillé dans la section II.4.1, la notion d'état est complexe. Dans une architecture à composants, nous considérons que chaque composant possède son état propre. La connaissance de cet état est partagée lors des interactions qui ont lieu entre les composants au travers des connexions entre interfaces et réceptacles. De plus, ces interactions font évoluer la valeur de cet état. Il peut donc exister des relations logiques entre l'état de plusieurs composants.

Lors de la modification d'une architecture à composants, certains composants sont insérés dans l'architecture, et d'autres sont retirés. Le problème consiste à déterminer l'état dans lequel les nouveaux composants doivent être placés, afin d'interagir correctement avec les composants qui ont été conservés lors de l'adaptation. Il s'agit de respecter la cohérence de l'état des composants vis-à-vis des relations logiques existantes dans le système.

Dans ces travaux, nous nous intéressons plus particulièrement à l'adaptation de mécanismes de tolérance aux fautes situés au méta-niveau du fonctionnel. La réplication des composants fonctionnels est réalisée à partir de certaines hypothèses initiales. L'une de ces hypothèses concerne l'état des composants fonctionnels, et plus particulièrement le fait que cet état soit synchronisé ou non, c'est-à-dire, que l'état initial du composant fonctionnel répliqué doit être identique pour chaque réplique. Ainsi, lorsque l'adaptation conduit à un changement de protocole de réplication, ou l'ajout d'une réplique, l'hypothèse initiale sur l'état doit être respectée.

Prenons un exemple. La réplication passive consiste à réaliser une sauvegarde de l'état d'une unique réplique active (i.e. réalisant les traitements), et à stocker cet état sur un support stable. Lorsque la réplique active est défaillante (crash), l'état est chargé dans une des répliques passives, puis cette réplique est démarrée afin de devenir la nouvelle réplique active. L'hypothèse de départ de cette stratégie de tolérance aux fautes est que la réplique initialement active possède un état à jour vis-à-vis du système. La réplication semi-active repose sur l'utilisation de plusieurs répliques qui exécutent les traitements en parallèle. Une seule de ces répliques interagit avec le reste du système. Cette réplique est dite active. Les autres répliques utilisent les requêtes et réponses fournies à la réplique active à des fins de synchronisation, et donc n'interagissent pas avec le reste du système. Ces répliques sont dites semi-actives. Lorsque la réplique active est défaillante, une de ces répliques devient la réplique active, et se met à interagir avec le reste du système. L'hypothèse de départ de cette stratégie est que toutes les répliques possèdent un état à jour avec le reste du système, donc identique d'une réplique à l'autre. Lors de l'adaptation de la réplication passive à la réplication semi-active, il est nécessaire de copier l'état de l'unique réplique active vers les autres répliques afin d'assurer l'hypothèse sur l'état nécessaire à la réplication semi-active.

Ainsi, le problème de l'état (cf. figure 18) peut être décomposé en :

- Une resynchronisation de l'état de certains composants fonctionnels si nécessaire ;
- Une reconstruction de l'état des composants de tolérance aux fautes nouvellement inséré, permettant de respecter les relations logiques reliant l'état des différents composants.

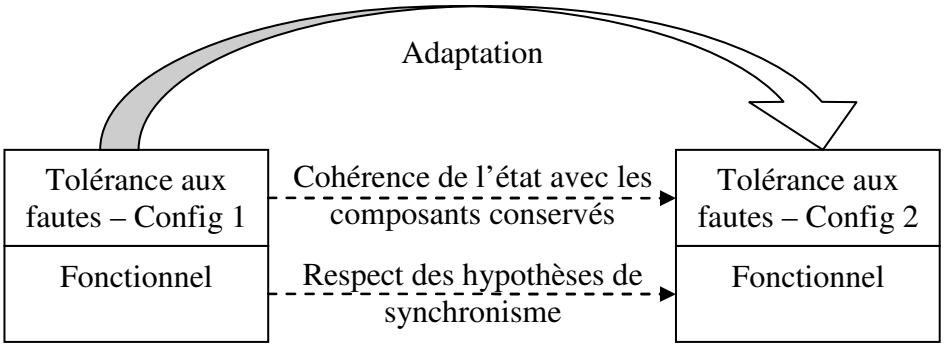


Figure 18 Problème de l'état lors de l'adaptation

Les travaux réalisés dans cette thèse se focalisent sur l'aspect comportemental pendant l'adaptation. Dans ces travaux, nous ferons l'hypothèse que le concepteur du système est capable d'écrire les règles de transfert d'état entre les composants qu'ils soient fonctionnels ou non-fonctionnel. Toutefois, il serait intéressant de trouver une méthode permettant l'écriture de ces règles de transfert d'état, ou même, essayer de construire un modèle permettant de généraliser ce traitement pendant l'exécution. Le principal obstacle concerne la sémantique des variables constitutives de l'état qui nécessite probablement d'explorer la piste des ontologies.

III.2.4 Sûreté de fonctionnement du processus d'adaptation

Les mécanismes de tolérance aux fautes contribuant à la sûreté de fonctionnement du système, leur modification peut directement impacter leur capacité de traiter les erreurs. Il est impératif que le processus d'adaptation ne remette pas en cause la sûreté de fonctionnement du système. L'adaptation doit donc fournir des garanties en termes de résultat, mais aussi en termes de comportement en présence de fautes.

Nous pensons que certaines garanties doivent donc être apportées au processus d'adaptation. Tout d'abord, le processus d'adaptation doit assurer que, si l'adaptation a lieu, elle est justifiée et amène le système dans un état correct, c'est-à-dire respectant ses spécifications. Ensuite, le processus d'adaptation doit être tolérant aux fautes, c'est-à-dire que son modèle de fautes doit être étudié, ses modes de défaillances évalués et des mécanismes de tolérance aux fautes doivent contribuer à renforcer sa robustesse vis-à-vis de son modèle de fautes.

Dans ces travaux de thèse, nous nous intéressons à réaliser une adaptation correcte, et à fournir des mécanismes permettant son recouvrement en cas de crash d'un nœud du système. Le premier point a été souligné dans chacune des problématiques traitées précédemment dans cette section. Il consiste à trouver une méthode de transformation du logiciel permettant de respecter ses spécifications avant, pendant et après l'adaptation du logiciel. Le deuxième point consiste à permettre d'annuler le processus d'adaptation en replaçant le système dans la configuration avant que les modifications ne soient appliquées. Ainsi, lorsque le processus d'adaptation rencontre une erreur (crash d'un nœud par exemple), l'adaptation est annulée, et le système est replacé dans la configuration qui précédait l'ordre d'adaptation.

III.3 MODELES DU LOGICIEL

Pour permettre de traiter l'adaptation, il est nécessaire d'avoir en notre possession des représentations du logiciel pendant son fonctionnement. Ces représentations sont appelées des modèles du logiciel. Nous en distinguons trois :

- Modèle de l'architecture
- Modèle d'interception
- Modèle d'exécution

Nous allons maintenant les détailler.

III.3.1 Modèle de l'Architecture

Dans une architecture à composant, il existe deux types de liens entre les composants : la composition et les connexions. La composition est introduite par l'utilisation de composants composites appelés aussi conteneur, qui peuvent contenir un ou plusieurs composants. Une connexion est le lien qui relie un réceptacle à une interface, afin que deux composants puissent interagir.

Nous proposons de représenter chaque type de lien par un graphe (cf. figure 19), où un composant est représenté par un nœud, et le lien (composition ou connexion) par un arc orienté. Dans le graphe de composition, l'arc relie le conteneur au composant qu'il contient. Dans le graphe des connexions, l'arc relie le composant client possédant le réceptacle au composant prestataire implémentant l'interface vers laquelle le réceptacle est connecté. Cet arc est labélisé avec le nom de l'interface du composant prestataire.

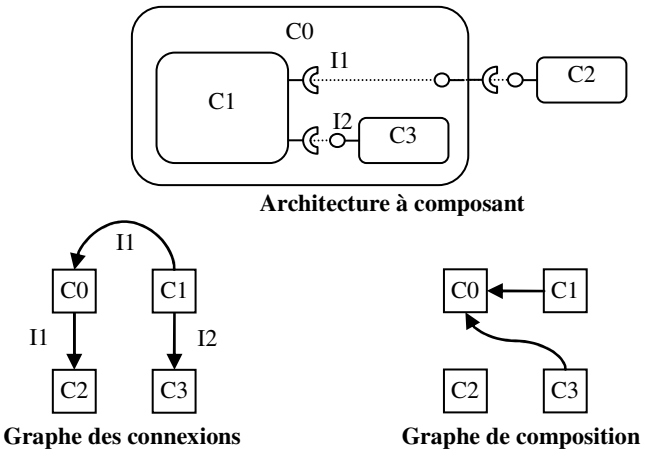


Figure 19 Représentation d'une architecture à composants

III.3.2 Modèle d'interception

Pour mettre en place le modèle de l'exécution, il est nécessaire de pouvoir observer l'exécution du système. De notre point de vue, l'exécution peut être observée à deux niveaux :

- Interception aux interfaces du composant : les appels sur une interface sont systématiquement interceptés. Toutefois, plusieurs composants peuvent être connectés à cette interface, et donc, être à l'origine de cet appel. En ne considérant l'interception qu'au

niveau des interfaces, aucune information n'est disponible sur le composant à l'origine d'un appel.

– Interception sur les connexions entre composant : les appels entre deux composants sont interceptés, il y a connaissance de l'origine de l'appel, cependant, les appels passés en dehors des connexions supervisées ne sont pas interceptés.

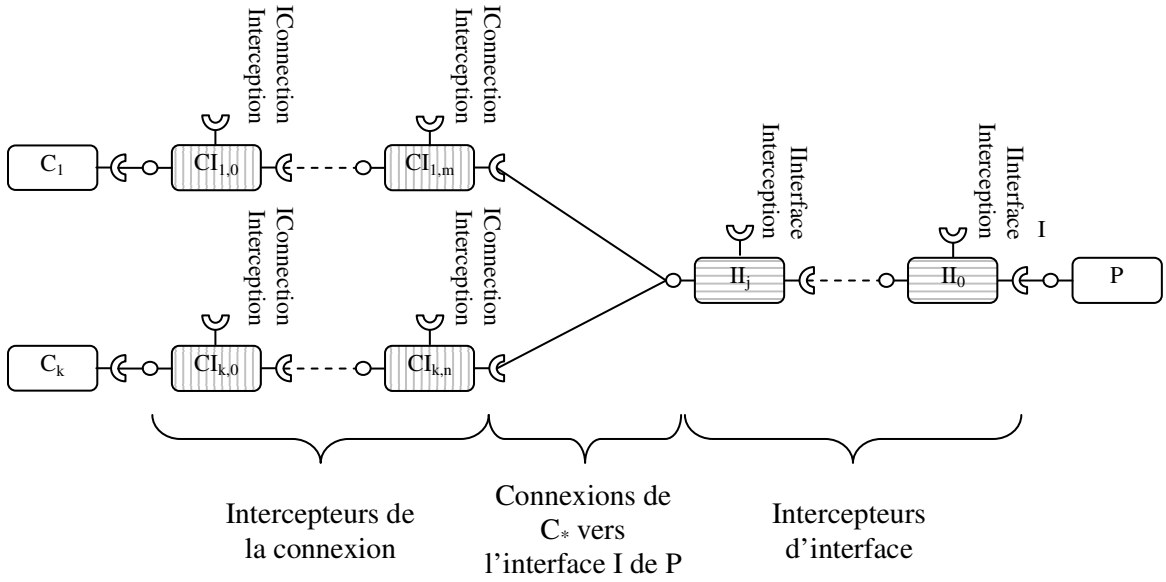


Figure 20 Utilisation des composants d'interception

Ces deux types d'interception fournissent des propriétés différentes, qui peuvent être intéressantes pour l'adaptation. En effet, l'isolation d'un composant vis-à-vis des appels extérieurs (autres composants de l'architecture) nécessite d'intercepter l'ensemble des appels, peu importe leur provenance, alors que la supervision de l'exécution nécessite de connaître l'origine des appels.

Tableau 5 **Détail des interfaces pour l'interception**

Nom de l'interface	*Méthodes*	*Détails*
IDelegateInterface	Object delegateInterfaceCall(String iid, Object interface, Method m, Object[] args) throws Throwable	Délégation de l'appel de la méthode m sur l'interface identifiée par iid avec les arguments args. Si un appel doit être passé au composant, il doit l'être au travers de l'interface qui pointe vers le prochain délégateur d'interface, ou le composant si le délégateur est le dernier.
IDelegateConnection	Object delegateConnectionCall (Long connID, Object interface, Method m, Object[] args) throws Throwable	Délégation de l'appel de la méthode m au travers de la connexion identifiée par connID avec les arguments args. Si un appel doit être passé au composant, il doit l'être au travers de interface qui pointe vers le prochain délégateur de connexion, ou sur le premier délégateur d'interface s'il en existe un, et sinon, vers le composant cible.
	void informComponentDisconnection(Long connID, String iid)	Informe que la connexion entre les composants a été rompue.

Pour mettre en place ces deux types d'interception, nous introduisons des composants appelés intercepteurs. Ces composants implémentent une interface et un réceptacle compatibles avec le service intercepté. Ces composants possèdent un réceptacle qui est connecté au composant

non-fonctionnel nécessitant l'interception. Les intercepteurs d'interface sont placés au niveau de l'interface, et les intercepteurs de connexion sont placés au niveau du réceptacle. Ainsi, lorsqu'une connexion est effectuée entre deux composants, le dernier intercepteur de connexion est connecté avec le dernier intercepteur d'interface (cf. Figure 20).

Cette manière de réaliser l'interception est particulièrement adaptée à nos problématiques d'adaptation. En effet, l'utilisation des intercepteurs de connexion permet, lors d'une déconnexion, de fournir certaines garanties selon le type de connexion utilisée comme le blocage des appels vers la connexion déconnectée par exemple.

III.3.3 Modèle de l'Exécution

Ce modèle est l'une des contributions majeures de ces travaux de thèse. Nous proposons de construire un modèle comportemental du logiciel représentant les tâches du système, leurs dépendances, et leurs chemins d'exécution à l'intérieur de l'architecture à composants en utilisant les réseaux de Petri. Ce modèle permet, à l'exécution, de connaître l'état de chaque tâche, et de contrôler l'exécution de ces tâches. Il a pour objectif de permettre le contrôle de l'exécution du système pour l'amener vers un état adaptable et l'y maintenir.

Notre approche nécessite d'identifier les chemins d'exécution internes à chaque composant. Le modèle représente alors les chemins d'exécution sous la forme d'un réseau de Petri qui existe pendant le fonctionnement du système. À l'exécution, ce modèle est animé par l'observation du logiciel. Ces observations peuvent reposer sur les mécanismes d'interception du modèle à composant si les interactions existent dans le monde des composants. Sinon, ces observations doivent être réifiées par les composants. Cela est possible du fait que les composants sont ouverts, ce qui permet d'insérer des sondes permettant d'observer ces évènements.

Nous sommes en présence d'un système qui possède plusieurs tâches liées par des dépendances et des contraintes (cf. section III.2.2) entre elles. Il est donc nécessaire de faire apparaître ces contraintes dans le modèle pour calculer et atteindre un état adaptable du système. Nous considérons que les appels de procédures au travers des connexions de l'architecture à composants sont bornés dans le temps et donc qu'en l'absence de défaillance tout appel commencé se terminera.

Nous souhaitons que ce modèle conserve la propriété de composition introduite par le modèle à composant. Ainsi, à partir d'une connaissance locale à chaque composant, nous souhaitons obtenir, par composition d'une connaissance locale du comportement des composants de l'architecture à composants, le modèle comportemental global de cette architecture.

Dans un premier temps, nous introduisons des motifs d'exécution élémentaires décrits sous la forme de réseau de Petri. Ensuite, nous montrerons comment combiner ces motifs pour décrire le comportement d'un unique composant. Puis, nous montrerons comment composer les modèles des composants pour obtenir un modèle de l'architecture déployée.

III.3.3.1 Représentation d'un état d'exécution

L'objectif du modèle est de caractériser tous les états d'exécution du système selon qu'ils permettent l'adaptation ou non. L'utilisation des réseaux de Petri nécessite de faire un choix de représentation important pour la réalisation du modèle. En effet, il existe deux manières d'encoder l'état de l'exécution :

- Un marquage du réseau de Petri représente un état de l'exécution : le réseau de Petri est alors équivalent à son graphe de marquage. Cette représentation ne permet pas de profiter

du caractère compact des réseaux de Petri, et risque d'entraîner une explosion combinatoire du nombre de places et de transitions dans le modèle ;

- La séquence de transitions depuis le démarrage du système représente un état de l'exécution : le réseau de Petri utilisé est alors plus compact. Toutefois, la complexité est transférée au niveau de l'encodage de l'état qui nécessite une mémoire des transitions franchies, ce qui peut être non négligeable en termes de taille.

De plus, nous devons définir les notions de passé et de futur d'une exécution. Ces notions sont définies de la manière suivante :

- Le **passé** est la séquence de transitions franchies depuis l'état initial.
- Un **futur** d'une tâche est une séquence de transitions qui peut être franchies à partir de l'état d'exécution courant. Il existe plusieurs futurs possibles pour une tâche dans un état d'exécution donné.

L'utilisation de tâches cycliques dans le système permet un *phénomène d'oubli du passé* propice à l'adaptation. En effet, lorsque la tâche s'arrête, on suppose qu'elle est placée dans son état initial du point de vue du flot de contrôle. De ce fait, le passé de l'exécution devient vide, ce qui est comparable à un oubli du passé. Dans l'état initial, l'adaptation est toujours possible. Cela montre l'intérêt d'utiliser la notion de tâches cycliques dans les systèmes adaptables.

Nous remarquons que, dans la deuxième représentation, l'état d'exécution est caractérisé par son passé. Dans la première représentation, puisque le réseau de Petri est équivalent à son graphe de marquage, il existe une bijection entre les séquences de transitions et le marquage du réseau. Donc passé et marquage son équivalent.

De notre point de vue, la deuxième solution semble plus cohérente avec l'utilisation des réseaux de Petri et permet de représenter l'exécution de manière plus compacte. Nous choisirons de mettre en œuvre cette solution.

Enfin, pour assurer l'existence d'une représentation sous la forme d'un réseau de Petri, il est nécessaire d'assurer que ce réseau est fini. La récursivité va à l'encontre de cette hypothèse. Nous supposons donc que le système ne possède pas d'appel récursif entre les différents composants de l'architecture.

III.3.3.2 Motifs élémentaires

Les tâches

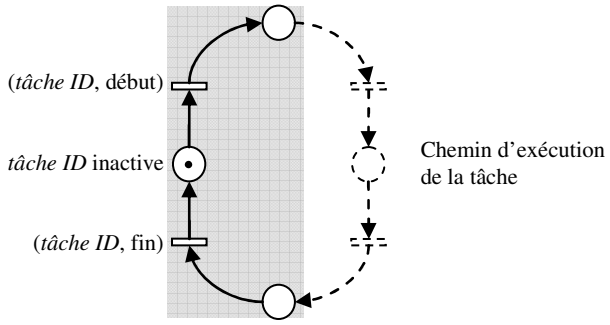


Figure 21 **Motif d'une tâche**

Nous avons fait l'hypothèse que les tâches du système sont bornées dans le temps et qu'elles pouvaient être exécutées plusieurs fois de manière cyclique. Nous supposons que chaque tâche du système peut être identifiée de manière unique au sein d'un composant. Ainsi, pour obtenir une identification unique au sein de l'architecture à composant, on peut ajouter à cet identifiant, l'identifiant unique du composant contenant cette tâche. Lors de la création d'un composant, toutes les tâches qu'il contient sont inactives. En opération, la tâche peut être démarrée (transition début). Ensuite, la tâche parcourt son chemin d'exécution, puis se termine (transition fin) et redevient inactive. Elle peut alors être démarrée à nouveau. La notion de tâche séquentielle début-fin peut se ramener à cette représentation (Figure 21), en considérant en fait que toute tâche s'appuie sur un mode cyclique de fonctionnement avec l'assurance d'un bouclage en un temps fini. Ainsi, une tâche est caractérisée par le composant qui la déclare. Une tâche t_i appartient au composant C_j si et seulement si, le composant C_j déclare la tâche t_i.

Les appels de procédure

Nous considérons que chaque appel peut être identifié de manière unique, par exemple, par la signature de la méthode. Un appel de procédure est l'exécution d'une méthode. L'appel commence par l'observation de l'évènement « début de l'appel de la méthode » et termine par l'observation de l'évènement « fin de l'appel de la méthode ». Le début de l'appel est réalisé à un seul endroit du chemin d'exécution, dans le sens où, si la méthode est appelée à un autre endroit, il s'agit d'un autre appel. Par contre, la fin de l'appel peut avoir lieu à plusieurs endroits du chemin de l'exécution de cette méthode. Il existe donc potentiellement plusieurs transitions de fin d'appel. Toutes les fins d'un même appel ramènent l'exécution à la même position du chemin d'exécution. Toutefois, peu importe comment l'appel sortant de la méthode termine, l'exécution reviendra dans le même état du chemin d'exécution. Ainsi, un appel sortant est caractérisé par un état avant l'appel, une transition de début d'appel, plusieurs (au moins une) transitions de fin d'appel, et un état après l'appel (cf. figure 22).

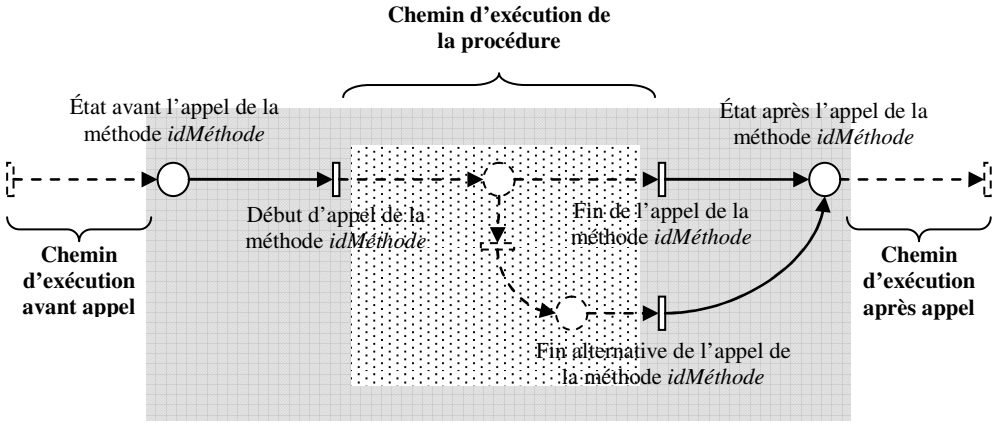


Figure 22 **Motifs des appels de procédure**

Les messages et évènements

L'échange de messages est caractérisé par deux évènements qui sont l'envoi et la réception du message. L'envoi et la réception sont soumis à une contrainte d'antériorité (cf. section antériorité). Ainsi, l'observation de la réception ne peut se faire avant l'observation de l'émission du message. Il en va de même pour l'échange d'évènements. Dans la suite de ces travaux, échange d'évènements et de message seront confondus.

Les structures de contrôles

Un chemin d'exécution est structuré. En effet, dans les langages de programmation de haut niveau, des structures de contrôle ont été introduites. Nous distinguons trois principaux cas :

– Pas de structure de contrôle. En l'absence de structure de contrôle, les instructions sont exécutées séquentiellement.

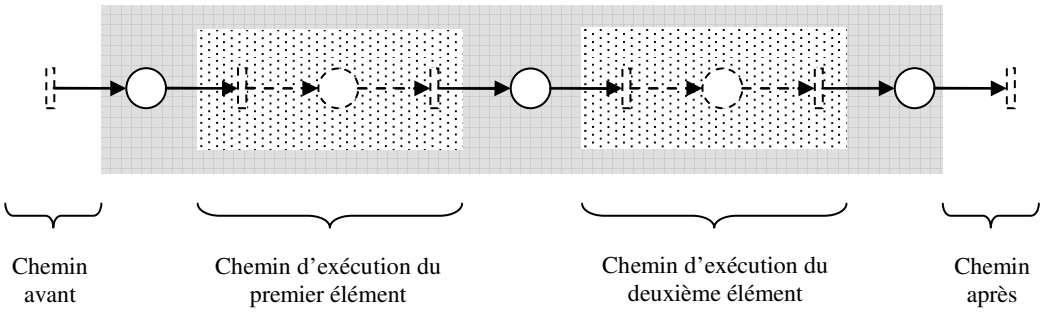


Figure 23 Représentation d'une séquence

– Alternatives : dans le cas de l'utilisation de *If, then else* et toutes les dérivées comme par exemple *elseif* ou *switch*, selon qu'une condition est vérifiée, l'exécution se branche sur une partie du code du programme. En d'autres termes, cela représente l'alternative entre plusieurs branchements de l'exécution (cf. figure 24).

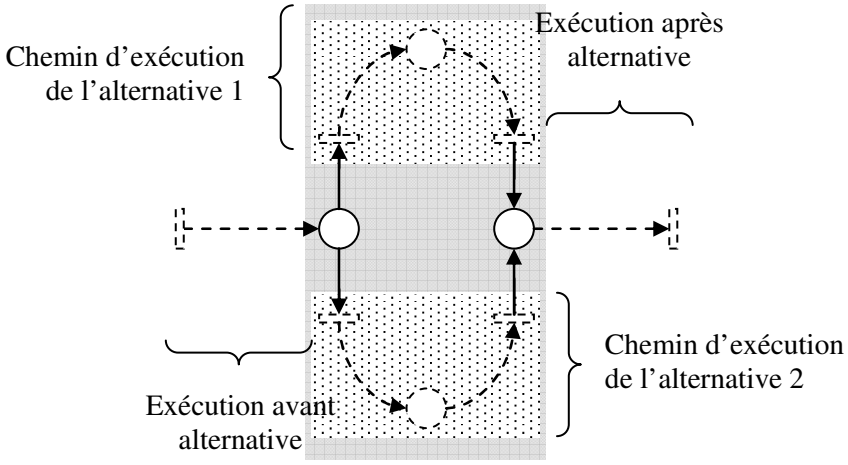


Figure 24 Représentation d'une alternative

– Boucles : lors de l'utilisation de boucles (*for*, *while*), les instructions contenues dans la boucle sont exécutées zéro, une ou plusieurs fois. Nous avons préalablement émis l'hypothèse d'une exécution temporellement bornée pour chacune des tâches, ce qui implique que le nombre d'itérations est borné, et que chaque itération est bornée dans le temps (cf. figure 25).

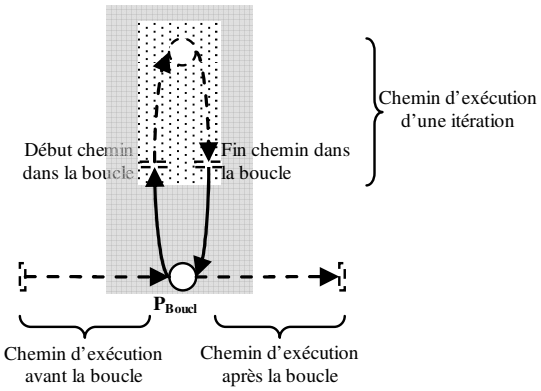


Figure 25 Représentation d'une boucle

Nous tenons à faire remarquer au lecteur que les boucles concernées sont celles provoquant des interactions entre les composants. Les boucles internes à un composant qui ne provoquent pas d'appels extérieurs, ni d'envoi ou réception de messages ou d'évènements ne sont pas concernées.

De plus, nous verrons par la suite que la définition de l'état adaptable du système peut rendre atomique l'ensemble des itérations vis-à-vis de l'adaptation. Dans le cas où l'adaptation peut être réalisée entre deux itérations, la solution n'est pas optimale. Toutefois, cette boucle est compatible avec le modèle de tâche cyclique introduit ici. Il est donc possible de permettre l'adaptation entre deux itérations en implémentant la boucle sous la forme d'une tâche cyclique. De cette manière, chaque cycle est rendu indépendant vis-à-vis de l'adaptation qui peut alors être réalisée au milieu des itérations.

Antériorité

Deux évènements peuvent être ordonnés dans le temps. Par exemple, la réception d'un message ne peut être observée avant l'envoi de ce message. Pour cela, nous proposons d'introduire un motif représentant cette antériorité. Ce motif est représenté à la figure 26. Il introduit une place représentant un message ou un évènement en transit.

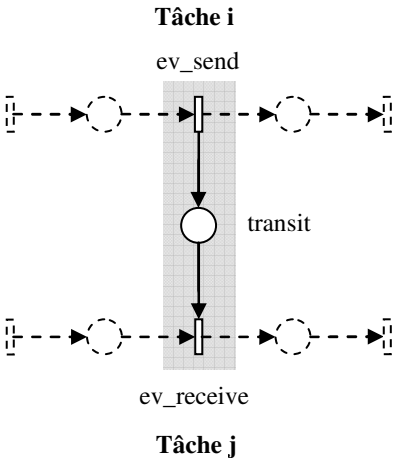


Figure 26 Motif d'antériorité

Synchronisation

Il existe plusieurs mécanismes de synchronisation. Ces mécanismes sont utilisés pour assurer l'exclusion mutuelle entre plusieurs tâches. Ces mécanismes fournissent des comportements relativement différents :

- Le mutex : c'est une primitive de synchronisation qui garantit l'unicité de l'exécution dans une section critique ;
- Le sémaphore : c'est une variable protégée qui garantit sa lecture et sa modification dans un cycle insécable. Son utilisation permet de résoudre des problèmes d'accès concurrents tels que le problème de lecture/écriture sur une ressource partagée où plusieurs exécutions lisent la ressource et aucune ne la modifie, ou bien une seule exécution modifie la ressource, et aucune ne la lit ;
- Les moniteurs et les variables conditions : un moniteur est un outil de synchronisation plus évolué qu'un sémaphore. Il consiste à encapsuler une ressource partagée dans un objet dont les méthodes permettant de manipuler la ressource s'exécutent en exclusion mutuelle. La notion de variable condition permet de représenter l'état d'une ressource sur laquelle les méthodes se synchronisent. Elle permet alors de préciser les règles d'accès à la ressource partagée par l'utilisation de deux primitives *wait* et *notify* qui provoque respectivement l'endormissement d'une tâche provoquant le relâchement de l'accès au moniteur, et son réveil.

Les mécanismes de synchronisation peuvent avoir un impact important sur le contrôle de l'exécution. En effet, contrôler l'exécution sans connaître les mécanismes de synchronisation peut amener à introduire des interblocages.

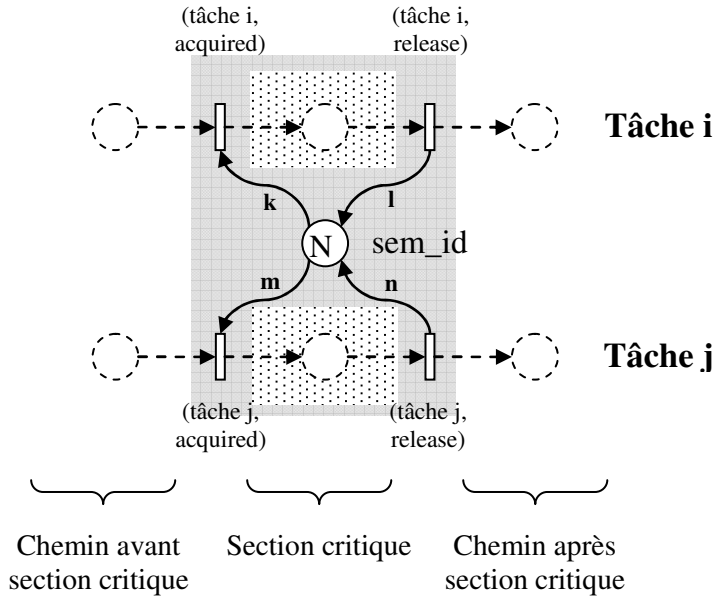


Figure 27 Motif d'un sémaphore

Dans ces travaux, nous proposons de représenter le mutex et le sémaphore. Le mécanisme de synchronisation nécessite d'introduire une place dans le réseau de Petri et deux évènements *acquired* et *release*. Le premier est observable dès la prise de jetons et le deuxième est observable avant de relâcher des jetons. Le nombre de jetons contenus dans cette place représente alors les jetons du mécanisme de synchronisation. Dans le cas du mutex, la place contient un seul jeton au maximum. Les arcs partant de cette place représentent la prise de jeton. Le poids de ces arcs représente le nombre de jeton pris. Les arcs allant vers cette place

représentent le relâchement de jetons. Leur poids représente le nombre de jetons rendus. Dans le cas du mutex, le poids est nécessairement de un.

Lorsqu'aucune tâche n'est pas à la fois consommatrice et productrice, le mécanisme de synchronisation permet d'implémenter de l'antériorité. Cette utilisation permet de résoudre des problèmes de production-consommation, où la tâche qui produit « signale » la présence d'une donnée en libérant le sémaphore, tandis que la tâche consommatrice consomme la donnée après avoir acquis le sémaphore. Dans ce cas, le sémaphore implémente un mécanisme d'antériorité.

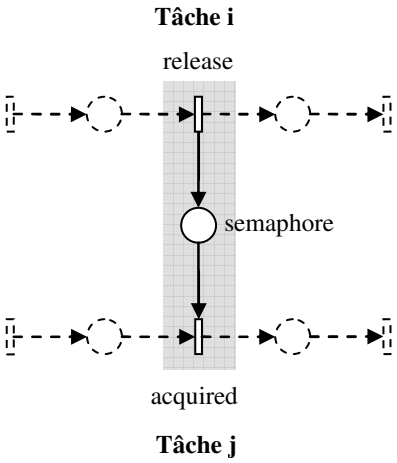


Figure 28 Synchronisation et antériorité

III.3.3.3 Composition des motifs pour la description d'un composant

En combinant les motifs, nous pouvons décrire le chemin d'exécution interne d'un composant. Prenons l'exemple d'un composant de communication. Ce composant possède :

- une interface *ISend* contenant une méthode *send* permettant d'envoyer un message ;
- une interface *IReceive* contenant une méthode *receive* qui permet de recevoir les messages ;
- une interface de type réseau permettant d'échanger des messages avec un client ;
- un réceptacle connecté à une interface *IListener* contenant une méthode *received* appelée lorsqu'un message est arrivé ;
- une tâche interne *task1*.

Un appel de *send* provoque l'envoi d'un message vers le serveur. Le composant possède deux modes : soit lorsqu'un message est reçu, la tâche appelle la méthode *received* pour délivrer le message reçu, soit elle le stocke dans un buffer et lorsqu'un appel de *receive* est passé, un message reçu est consommé. L'appel de la *receive* est rendu bloquant par l'utilisation d'un sémaphore *sem_1*. Dans le cas où le composant délivre automatiquement les messages, un appel à la méthode *receive* retourne une erreur.

À partir de ces descriptions, nous sommes capables de construire les chemins d'exécution au sein d'un unique composant (Figure 29). Ces chemins sont alors partiels. Les chemins d'exécution des appels à partir des réceptacles ne sont connus que lorsque le réceptacle est connecté à un autre composant, en d'autres termes, lorsque le composant est inséré dans une architecture à composant.

Le problème est assez similaire pour les mécanismes de synchronisation et d'antériorité qui n'ont pas de signification dans la modélisation locale d'un composant, puisque les tâches parcourant le composant ne sont pas connues et le seront uniquement lorsque ce composant sera inséré dans l'architecture à composant et connecté aux autres composants. Ainsi, dans cette représentation, les places représentant les messages/évènements en transit et les mécanismes de synchronisation ne sont pas représentées.

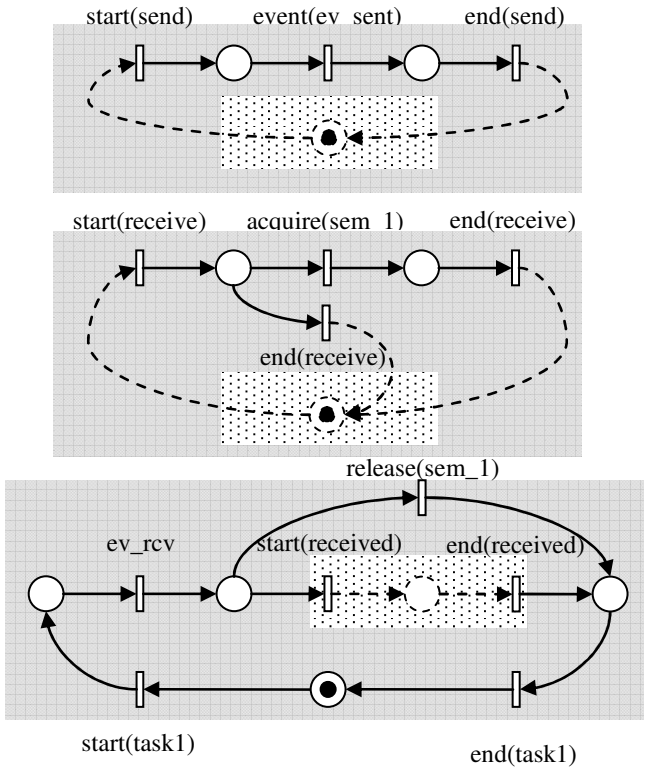


Figure 29 **Modèle du composant**

Les transitions de ce modèle sont associées à des évènements observés dans l'architecture à composant. La combinaison de ces motifs soulève un problème d'incertitude sur l'état courant. Cette incertitude peut s'exprimer de la manière suivante : il existe une séquence d'évènements observés tels que, à partir d'un marquage du réseau de Petri connu, plusieurs marquages peuvent être obtenus. La cause de cette incertitude est le non-déterminisme, c'est-à-dire le fait qu'un même évènement puisse amener dans deux marquages différents à partir d'un même marquage initial.

Prenons par exemple une alternative entre deux séquences. Les deux séquences débutent par le même évènement : le début d'un appel de la méthode m1 de l'interface I1 sur le composant C1. Toutefois, la première séquence est suivie d'un appel d'une méthode m2 alors que la deuxième séquence est suivie d'un appel sur une méthode m3. Ainsi, lorsque le début de l'appel sur m1 est observé, il est impossible de savoir si cet évènement correspond à la première séquence d'évènement ou à la deuxième. En effet, l'évènement correspondant à deux transitions franchissables, il est impossible de savoir quelle est la transition effectivement franchie. Il faudrait attendre l'appel de m2 ou m3 pour le déterminer. Ce non-

déterminisme est intrinsèque à la structure du réseau de Petri qui est couplée à l'observation du système afin d'animer le modèle à l'exécution.

Pour résoudre cette incertitude, nous pouvons observer que chaque morceau de chemin d'exécution est un réseau de Petri borné, avec un seul jeton. Ce réseau de Petri est alors équivalent à une machine à états finis. Le problème du non-déterminisme consiste alors à trouver pour cette machine, une machine équivalente minimale déterministe permettant de lire le même langage que la machine originale. C'est le problème de minimisation de machine à état finie [67].

III.3.3.4 Construction du modèle d'exécution de l'architecture

Un composant possède des interfaces qui peuvent être de deux types : synchrone et asynchrone. Les interfaces synchrones fournissent des services accessibles au travers d'appels de procédures, quelles soit locales ou distantes. Ces services synchrones sont caractérisés par un début et une fin. Les interfaces asynchrones fournissent des services du type échange de messages ou d'évènements. La connexion d'interface synchrone consiste à compléter le chemin d'exécution incomplet d'une tâche. La connexion d'interfaces asynchrone consiste à ajouter un motif d'antériorité.

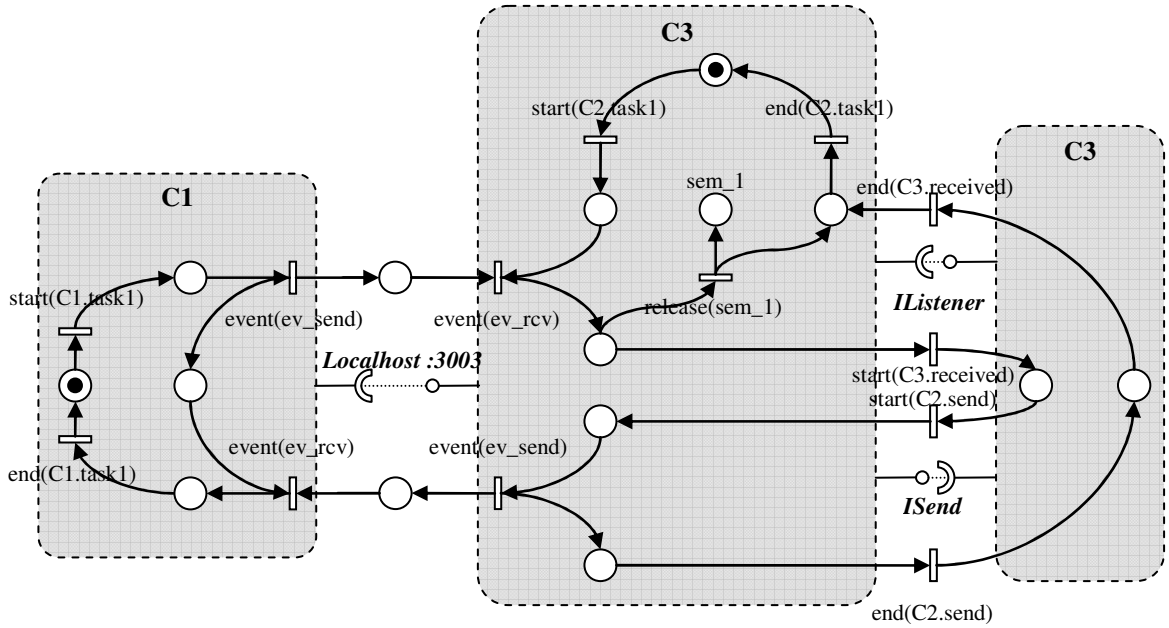


Figure 30 Modèle de l'exécution pour une architecture

Ainsi, nous pouvons construire le modèle de l'exécution d'une architecture (Figure 30). Dans cet exemple, l'architecture est composée de quatre composants :

- C1 est le client d'un service
- C2 est le composant de communication précédemment décrit
- C3 est le fournisseur du service requis par C1. Pour fournir ce service à C1, C3 est connecté à C2.

C1 est donc connecté sur l'interface réseau de C2. C3 est connecté à l'interface *ISend* de C2 et C2 est connecté à l'interface *IListener* de C3. C2 est configuré pour délivrer directement les messages reçus via l'interface *IListener*.

La figure 30 illustre le réseau de Petri de cette configuration logicielle. Il est alors construit à partir de la connaissance de chaque composant.

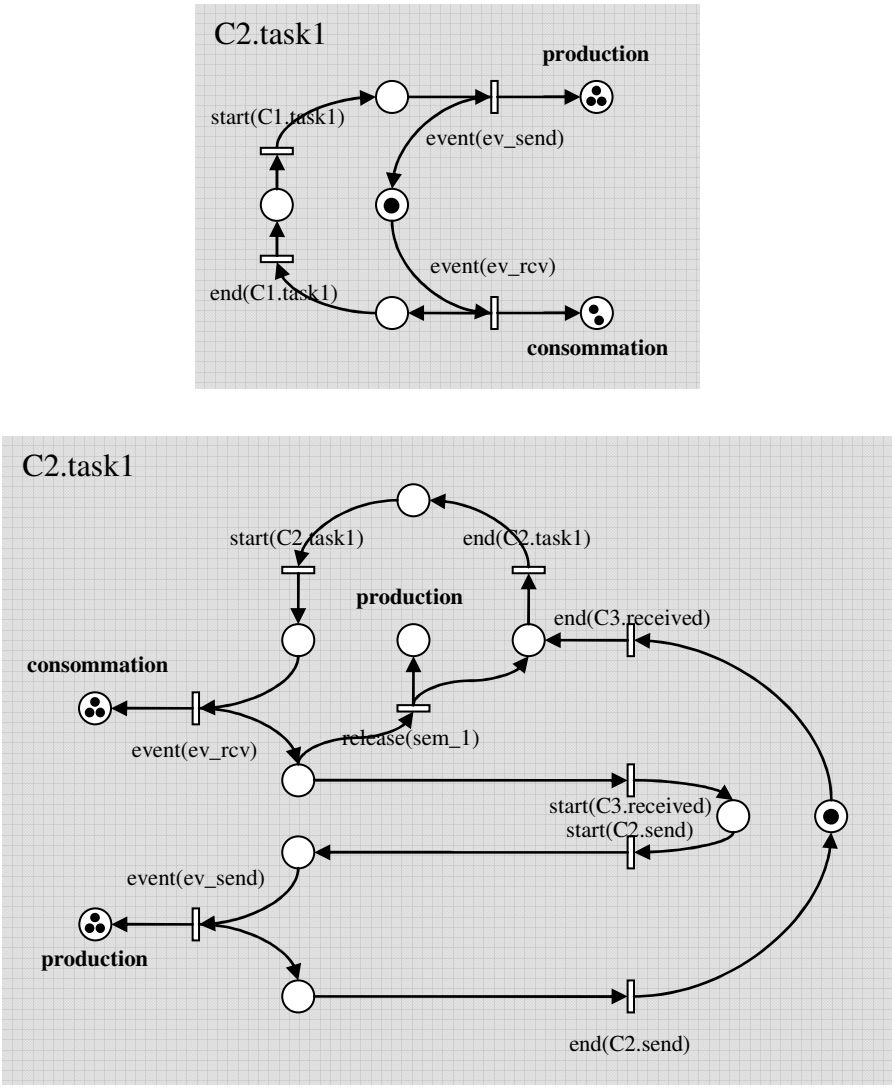


Figure 31 Implémentation du modèle

III.3.3.5 Implémentation du modèle de l'exécution

Dans un système distribué, il existe trois manières d'implémenter ce modèle :

- Centralisée : le modèle est situé sur un nœud du système, et toutes les observations réalisées sont envoyées à ce nœud. Toutefois, dans l'objectif d'utiliser ce modèle pour contrôler l'exécution des tâches du système, il est nécessaire de synchroniser l'observation de l'évènement et l'animation du modèle. Ainsi, chaque observation nécessiterait au minimum deux messages échangés entre le nœud de la tâche et le nœud du modèle. Le coût de cette implémentation semble alors trop important pour être acceptable.
- Décentralisée : chaque nœud possède une connaissance du modèle global qui doit être synchronisée avec les exécutions des différents nœuds du système. Le coût est encore plus important que dans le cas précédent.

– Distribuée : chaque nœud possède une connaissance des tâches qui lui sont locales, et partage uniquement les informations qui sont communes à plusieurs nœuds (messages), ce qui réduit considérablement l'impact sur les performances de l'exécution.

Nous proposons de mettre en place la troisième solution qui paraît plus adaptée à la distribution du système. Cette implémentation (Figure 31) est constituée d'une représentation de chaque tâche du système appelée méta-tâche. Chaque tâche est alors supervisée de manière indépendante au travers de la méta-tâche qui la représente. Toutefois, il est nécessaire de découpler les représentations de l'exécution de chaque tâche du système. Pour cela, nous proposons de découper le réseau de Petri du système en remplaçant les dépendances entre les tâches (synchronisation et antériorité) par des places artificielle. Ces places représentent alors la production ou la consommation de jeton pour les mécanismes de synchronisation ou d'antériorité considérés. Lors de l'adaptation, il est nécessaire de connaître le nombre de messages en transit par exemple. Pour cela, il suffit de faire la différence entre le nombre de jetons contenu par places représentant la production et le nombre de jetons contenu par les places représentant la consommation. Par exemple, pour connaître le nombre de messages en transit envoyés par la tâche C1.task1 vers la tâche C2.task1 dans l'architecture précédente, il suffit de faire la différence entre la place représentant la production et la place représentant la consommation (ici 3-2=1 message en transit). La tâche C2.task1 est donc en attente d'un message qui est actuellement en transit.

Ainsi, pour construire le modèle de l'exécution, nous interceptons la création/destruction des composants, leur connexion/déconnexion. Lorsqu'un composant est créé, nous recherchons les tâches que ce composant possède, et nous créons l'ensemble des méta-tâches correspondantes. Lorsque le composant est détruit, les méta-tâches sont elles aussi supprimées du système. Sur connexion de deux composants, chaque tâche de l'architecture complète les chemins d'exécution à partir des différents morceaux correspondant aux appels de méthodes du composant prestataire. Sur déconnexion de deux composants, les morceaux de chemin d'exécution passant par cette connexion sont supprimés.

Nous obtenons alors l'ensemble des méta-tâches du système qui constituent le modèle de l'exécution disponible en-ligne.

III.4 ADAPTATION EN-LIGNE ET CONTROLE

Nous avons introduit un modèle de l'exécution d'un logiciel mis sous la forme d'une architecture à composant. Nous allons maintenant montrer comment traiter le problème de sa modification en opération en tenant compte de son exécution.

Pour réaliser l'adaptation, il est nécessaire de résoudre plusieurs problèmes :

- Atteindre et maintenir le système dans un état adaptable. Ce problème peut être décomposé en :
  - Trouver pour chaque tâche, un état permettant d'assurer les spécifications, et contrôler son exécution pour l'amener dans cet état ;
  - Trouver pour l'ensemble des tâches les cycles d'exécution permettant d'assurer la cohérence de l'exécution des différentes tâches, et contrôler l'exécution pour maintenir cette cohérence pendant l'adaptation ;
- Automatiser la reconfiguration à partir d'une description de haut niveau ;
- Assurer la tolérance aux fautes du processus d'adaptation. Ce problème se décompose en :
  - Assurer le recouvrement du processus d'adaptation ;
  - Détecter les défaillances de ce processus et les traiter. Dans notre cas, seul le crash d'un nœud est pris en compte.

III.4.1 Tâches localement adaptables

Les tâches sont considérées indépendantes. Les mécanismes d'antériorité et de synchronisation ne sont alors pas pris en compte dans cette section. Cette indépendance permet de considérer l'**état adaptable de l'ensemble des tâches du système comme le produit cartésien des états adaptables de chaque tâche du système**. Ces dépendances sont introduites et prises en compte par le niveau en charge de la synchronisation des tâches. Nous allons donc étudier l'état adaptable pour une unique tâche.

Pour illustrer cette section, nous proposons de considérer l'architecture à composant et le modèle de l'exécution de la figure 32. Cette architecture est composée de deux composants *C1* et *C2*. *C1* est connecté à une interface de *C2*. Le système contient trois tâches :

- *C1.Tâche1* : cette tâche réalise deux appels à la suite sur le composant *C2*
- *C1.Tâche2* : cette tâche ne réalise pas d'appel sur *C2*
- *C2.Tâche1* : cette tâche appartient à *C2*

L'adaptation considérée dans cet exemple consiste à remplacer *C2* par un autre composant *C2'*.

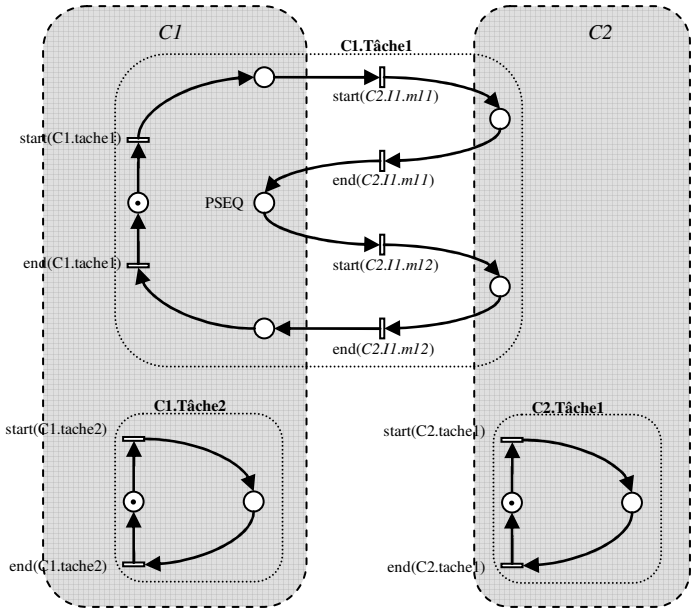


Figure 32 Cas d'étude pour le calcul de l'état adaptable dans le cas de tâches indépendantes

Nous avons vu dans la section III.2.2.2 que la première propriété que doit respecter l'état adaptable du système est l'isolation des composants/connexions à modifier des exécutions du système.

Deux cas peuvent alors se présenter :

- La tâche appartient à un composant qui va être modifié : cette tâche doit dont être arrêtée, ce qui correspond au fait qu'un jeton est présent dans la place représentant l'arrêt de la tâche (entre les transitions de début et de fin de la tâche) ;
- La tâche appartient à un composant qui va être conservé sans aucune modification : cette tâche ne doit pas avoir d'appel initié sur un composant qui doit être modifié. Ainsi, si une transition représentant un début d'appel de méthode vers un composant à modifier est franchie, une transition représentant la fin de cet appel doit être aussi franchie.

La deuxième propriété que doit respecter l'état adaptable pour une tâche est que si l'on applique les modifications sur le système à partir de cet état, alors lorsque la tâche aura fini son exécution, **elle aura respecté soit les spécifications de la configuration avant les modifications, soit les spécifications de la configuration après les modifications**.

Nous proposons d'étendre la notion du futur de l'exécution à la notion de futur dans un même cycle. Ainsi, nous considérons ici uniquement les futurs ne franchissant pas la transition de démarrage de la tâche.

Les états d'exécution d'une tâche peuvent être regroupés en trois ensembles disjoints :

- *E1* : le passé d'un état de *E1* ne contient pas de transitions représentant un appel vers un composant/connexion qui doit être modifié ;
- *E2* : aucun futur dans un même cycle à partir d'un état de *E2* ne contient de transition représentant un retour d'appel vers un composant/connexion qui doit être modifié ;
- *E3* : les états qui ne sont contenus ni dans *E1* ni dans *E2*.

Les états de *E1* et de *E2* respectent la propriété d'isolation. En effet, il n'y a pas d'appel en cours vers un composant qui doit être modifié.

E1 représente les états d'exécution qui sont communs aux spécifications avant et après adaptation. Ainsi, si les modifications sont appliquées dans un état appartenant à *E1*, alors, la poursuite de l'exécution se fera dans la configuration après adaptation. Puisque le début de l'exécution de la tâche a été réalisé dans un état commun aux deux spécifications, sur le cycle entier, la tâche aura globalement respecté les spécifications de la configuration après adaptation, et donc aura eu une exécution correcte.

E2 représente les états à partir desquels les chemins d'exécution ne sont pas modifiés par les modifications de l'architecture à composant. Ainsi, si les modifications sont appliquées dans un état appartenant à *E2*, la poursuite de l'exécution ne sera pas impactée par les modifications. Puisque le début de l'exécution de la tâche a été réalisé dans des états respectant les spécifications de la configuration avant adaptation et que la fin n'est pas impactée par les modifications architecturales, alors, à la fin de l'exécution du cycle, cette tâche aura respecté les spécifications de la configuration avant adaptation et lors de son prochain cycle d'exécution, elle respectera les spécifications de la configuration après adaptation.

Lorsque la tâche est dans un état de l'ensemble *E3* aucune garantie ne peut être apportée ni en termes d'isolation, ni concernant le respect des spécifications. Ainsi, l'adaptation dans un état de *E3* conduirait potentiellement à une défaillance de l'application et/ou de ses mécanismes de tolérance aux fautes.

E1 et *E2* ne sont pas nécessairement disjoints, et ce, dans deux cas. Le premier est celui où le chemin d'exécution de la tâche ne parcourt jamais de connexions/composants à modifier. Le deuxième est lorsqu'il existe des branches du chemin d'exécution qui ne parcourt pas de composants/connexions à modifier.

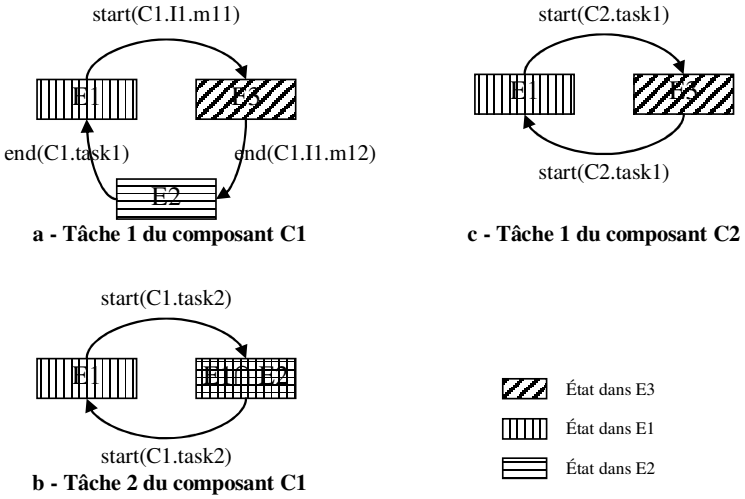


Figure 33 Propriétés d'adaptation d'une tâche

Pour un cycle d'une tâche, une séquence d'état d'exécution parcourt dans l'ordre (cf. figure 33) :

- des états de *E1* : si l'adaptation est réalisée dans cet état, au cours du cycle entamé, la tâche respectera les spécifications de la configuration après adaptation ;
- des états de *E3* : états ou aucune propriété n'est garantie (isolation, spécifications) ;

– des états de *E2* : si l'adaptation est réalisée dans cet état, au cours du cycle entamé, la tâche respectera les spécifications de la configuration avant adaptation.

On remarque que selon les chemins d'exécution des tâches du système, les ensembles E2 et E3 peuvent être vides. E2 est vide dans le cas où le composant implémentant la tâche doit être supprimé. Ainsi, aucun état de cette tâche, n'appartient à la spécification après modification, puisque la tâche n'existe pas dans cette configuration. C'est le cas de la tâche 1 du composant C2 (figure 33-c). *E3* est vide dans le cas où le chemin d'exécution de la tâche n'est pas impacté par les modifications architecturales du logiciel. C'est le cas de la tâche 2 du composant C1 (figure 33-b).

Les ensembles E1, E2 et E3 peuvent être déterminés hors ligne par l'étude des chemins d'exécution sur un cycle de la tâche.

Pour assurer la propriété d'isolation, il est nécessaire de contrôler l'exécution du système afin d'empêcher que la tâche n'évolue d'un état adaptable vers un état non-adaptable pendant le processus de reconfiguration. Pour cela, nous proposons d'introduire deux mécanismes de contrôle de l'exécution : le *lock* qui pose un verrou sur une transition du système et le *unlock* qui retire un verrou existant.

Les tâches étant indépendantes, le contrôle de leur exécution est lui aussi indépendant. Ainsi, lorsque toutes les tâches sont dans un état adaptable, alors, le système est globalement adaptable et les modifications peuvent être effectuées. Localement à chaque tâche, le contrôle assure la propriété d'isolation, une fois que la tâche a atteint un état adaptable (cf. figure 34). Après application des modifications, les transitions verrouillées sont de nouveau autorisées.

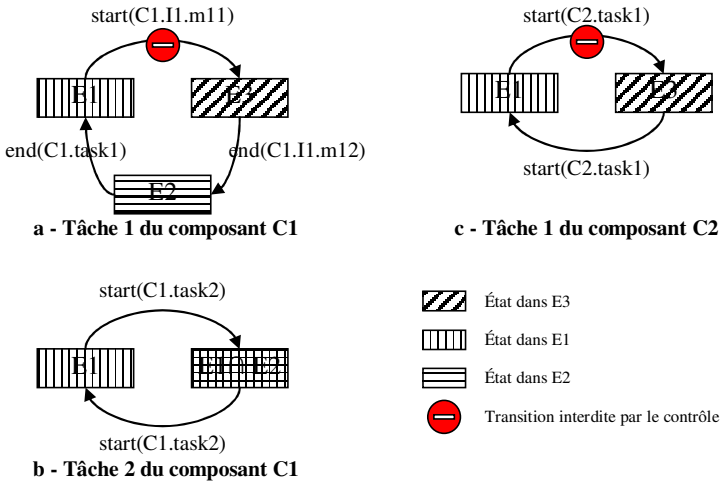


Figure 34 Influence des mécanismes de synchronisation sur le contrôle

III.4.2 Tâches avec dépendances

Nous abordons maintenant le cas de tâches dépendantes, en considérant les mécanismes de synchronisation et l'antériorité introduite lors de l'échange de messages ou d'évènements.

Dans un premier temps, nous étudions l'influence de ces phénomènes sur l'exécution du système et la cohérence de l'état vis-à-vis de l'adaptation. Ensuite, nous proposons un contrôle de l'exécution de ces tâches pour amener le système dans un état adaptable global au système distribué multitâches.

III.4.2.1 Influence des mécanismes de synchronisation

Nous considérons ici que les mécanismes de synchronisation sont utilisés pour réaliser l'exclusion mutuelle de plusieurs tâches sur une section critique du code. Il y a deux effets possibles de la présence de mécanismes de synchronisation. Le premier concerne la modification de cet état adaptable, et le deuxième concerne l'inaccessibilité de certains marquages appartenant à l'état adaptable du système.

Il est intéressant de remarquer que la section critique n'est pas toujours observable. Par exemple, si une méthode d'un composant contient une section critique, et que cette méthode ne réalise pas d'appel sortant, alors, il n'existe pas d'évènement observable au sein de la section critique. Ainsi, la section critique n'est pas observable. Cette propriété n'est pas réellement problématique dans ce cas bien précis. En effet, lorsqu'une section critique contenue dans une unique méthode n'est pas observable, il est impossible de bloquer l'exécution du système à l'intérieur de la section critique. Donc, il n'est pas possible de créer un inter-blocage.

Prenons maintenant le cas d'un mutex utilisé dans plusieurs méthodes. Comme dans le cas précédent, la section critique peut ne pas être directement observable. Toutefois, contrairement au cas précédent, cela peut amener à des inter-blocages. Prenons l'exemple de deux méthodes. Dans la première, la section critique n'est pas observable. Dans la deuxième, il existe un appel sortant dans la section critique, la rendant ainsi observable. Ainsi, deux tâches du système évoluent. La deuxième est entrée dans la section critique de la deuxième méthode, et donc, est en possession du mutex. La première est entrée dans la première méthode et est en attente de ce mutex. Toutefois, cette information n'est pas observable. Supposons que l'état d'adaptation nécessite d'une part d'attendre la fin de l'appel de la première tâche sur la première méthode, et d'autre part, empêcher la deuxième tâche de réaliser l'appel contenu dans la section critique. Ainsi, en bloquant cet appel, la deuxième tâche ne relâchera jamais le mutex, et la deuxième tâche ne pourra jamais l'obtenir. Elle ne terminera donc jamais l'appel de la première méthode. Il y a donc inter-blocage.

État non-adaptables

Nous nous intéressons ici au cas où la prise et la restitution du ou des jetons correspondant au mécanisme de synchronisation ne sont pas réalisés au cours d'un même appel. Généralement, les mutex et les moniteurs sont utilisés pour réaliser une section critique au sein d'une méthode. Seuls les sémaphores sont susceptibles d'être utilisés de cette manière.

Lorsque le composant qui possède le mécanisme de synchronisation doit être modifié, il est nécessaire de se demander dans quel état le mécanisme de synchronisation doit se trouver. Cela peut être résolu de deux manières : soit en spécifiant cet état comme une contrainte sur l'état adaptable du système, soit en transférant cet état vers le composant qui prendra sa place après modification du système.

De notre point de vue, une telle utilisation d'un sémaphore masque des interactions complexes entre les tâches du système, modifiant fortement le comportement global des tâches. Nous proposons donc d'introduire une contrainte supplémentaire sur l'état adaptable : **le sémaphore doit se trouver dans son état initial pour pouvoir modifier le composant qui le possède**. Ainsi, si le sémaphore est initialisé avec N jetons, l'état adaptable est une combinaison des états adaptables des tâches, telle que le sémaphore contienne N jetons.

Toutefois, le contrôle de l'exécution peut être non trivial en dehors du cas de l'antériorité et des sections critiques. Dans le cas de l'adaptation, nous pensons nécessaire de rajouter une contrainte supplémentaire sur la conception des composants et l'utilisation des mécanismes de

synchronisation. Cette contrainte consiste à n'utiliser les mécanismes de synchronisation uniquement pour réaliser des sections critiques et forcer l'antériorité. Tout autre utilisation nécessite de faire une étude plus approfondie du contrôle nécessaire pour placer le système dans un état adaptable en assurant la vivacité de l'exécution du système.

Dans tous les cas, il apparaît alors nécessaire de rendre le mécanisme de synchronisation visible et contrôlable par le méta-niveau.

Inaccessibilité des marquages de l'état adaptable

Dans le cas de tâches indépendantes, l'état adaptable global est le produit cartésien des états adaptables des tâches. L'existence des mécanismes de synchronisation rend certains états de l'exécution inaccessibles. L'utilisation des mécanismes de contrôle de l'exécution introduits dans la section III.4.1 peut amener à des situations d'interblocage du système.

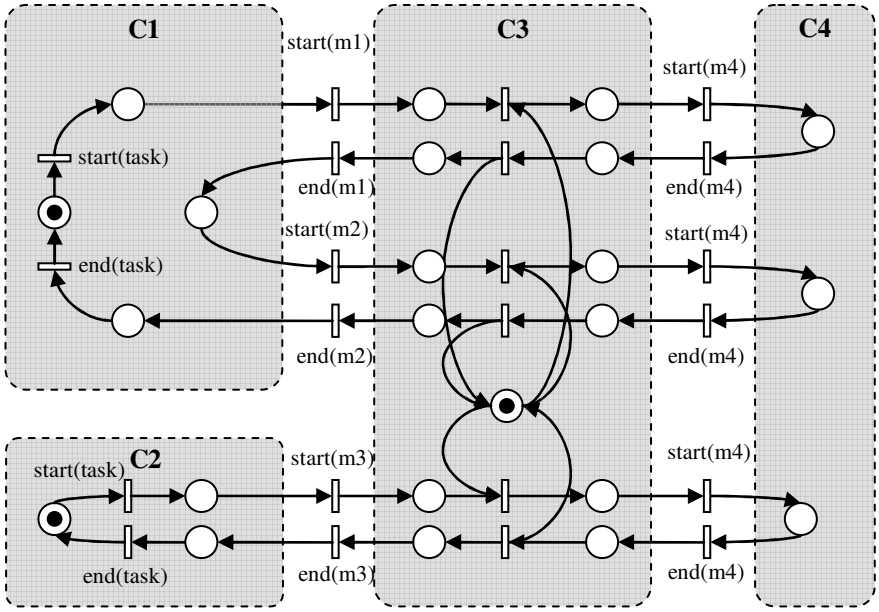


Figure 35 Exemple d'exécution avec sections critiques

Prenons par exemple l'exécution décrite à la figure 35. Le système possède deux tâches qui réalisent des appels vers des méthodes partageant des sections critiques (*m1*, *m2* et *m3*). Nous souhaitons remplacer le composant C4 par un nouveau composant C4'.

Pour chaque tâche du système, nous pouvons abstraire le réseau de Petri à la machine à état représentant les états d'adaptation de la tâche. De plus, nous proposons d'éclater les états dans le but de faire apparaître les transitions en relation avec le mécanisme de synchronisation (Figure 36).

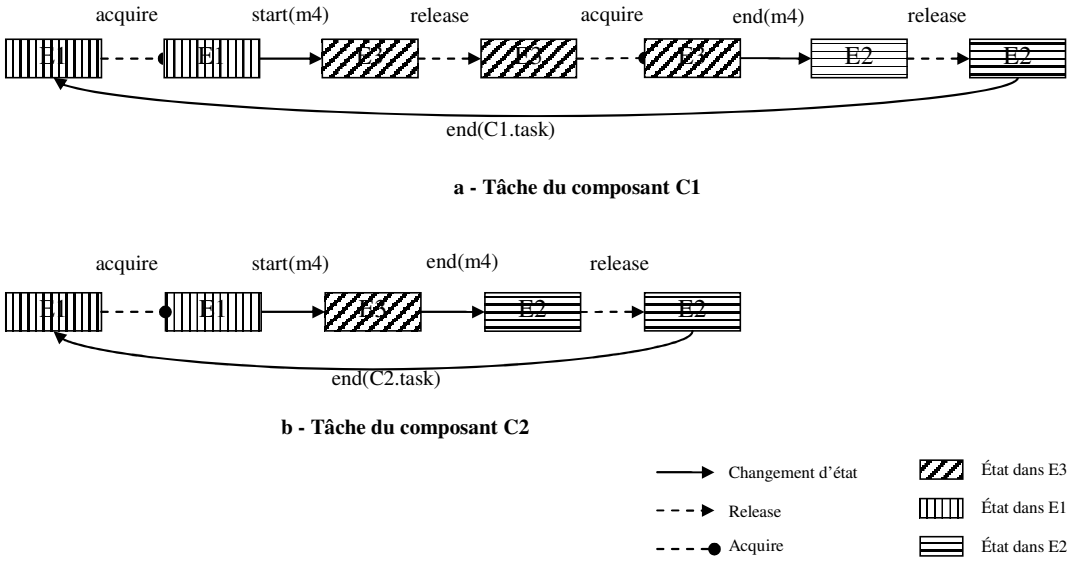


Figure 36 États adaptables des tâches avec section critique

Prises indépendamment, ces deux tâches parcourent cycliquement E1 puis E3 et enfin E2. Toutefois, la présence des sections critiques supprime certaines transitions qui existaient dans le cas de tâches indépendantes. Ainsi, les règles de contrôle de l'exécution qui s'appliquaient dans le cas de tâches indépendantes peuvent créer des situations d'interblocage entre les tâches du système. Dans l'exemple, cela arrive lorsque la tâche du composant C2 est dans la section critique, et est bloquée par le contrôle avant l'appel de m4 et que la tâche C1 est en attente d'entrée dans la section critique pour réaliser le deuxième appel vers le composant C4. La figure 37 représente les états du système, qui est donc un produit des états de chaque tâche. Ainsi, on remarque que des transitions entre deux états du système qui existaient dans le cas de tâches indépendantes sont rendues impossibles. L'application de la même politique de contrôle peut amener le système dans des états puits non-adaptables, où aucune transition n'est franchissable. Il y a donc interblocage dans un état non-adaptable.

Deux solutions peuvent alors être appliquées :

- Rendre conditionnel le verrou : si une tâche est en attente d'un sémaphore dans un état non-adaptable (le sémaphore est nécessaire à sa sortie de E3) et qu'une tâche qui doit être bloquée (entrée dans E3) possède un futur permettant de rendre ce sémaphore, alors, le verrou n'est pas posé, et la tâche entre dans son état non-adaptable (i.e. E3) ;
- Étendre l'état non-adaptable à la prise du verrou : sur un chemin d'exécution d'une tâche alternant prise de sémaphore, état non-adaptable et relâchement du sémaphore, l'état non-adaptable est étendu aux états de l'exécution entre la prise de sémaphore et l'entrée dans l'état non-adaptable.

Ces deux solutions possèdent des avantages et des inconvénients. Un verrou conditionnel peut amener le système à ne jamais converger vers un état adaptable. Par exemple, si la tâche sur lequel le verrou conditionnel s'applique préempte toujours la tâche en attente du sémaphore, alors le système n'atteint jamais un état adaptable. Certes, la probabilité avec deux tâches est relativement faible, mais lorsque plusieurs tâches sont en concurrence pour le verrou, ce comportement ne peut plus être négligé. L'extension de l'état non-adaptable peut amener à considérer un grand nombre d'états comme des états non-adaptables alors qu'en réalité, ces états permettent l'adaptation. Toutefois, cette méthode offre la garantie de convergence.

Nous proposons de mettre en place une combinaison des deux méthodes permettant à la fois de résoudre les interblocages de limiter les états considérés comme non-adaptables et garantissant la convergence du système. Il s'agit alors d'introduire des verrous temporaires sur l'entrée dans la section critique en plus des verrous conditionnels. Ces verrous temporaires sont relâchés une fois l'état adaptable atteint. Ainsi, les verrous conditionnels garantissent l'absence d'interblocage, et les verrous temporaires garantissent la convergence en maximisant le nombre d'états adaptables.

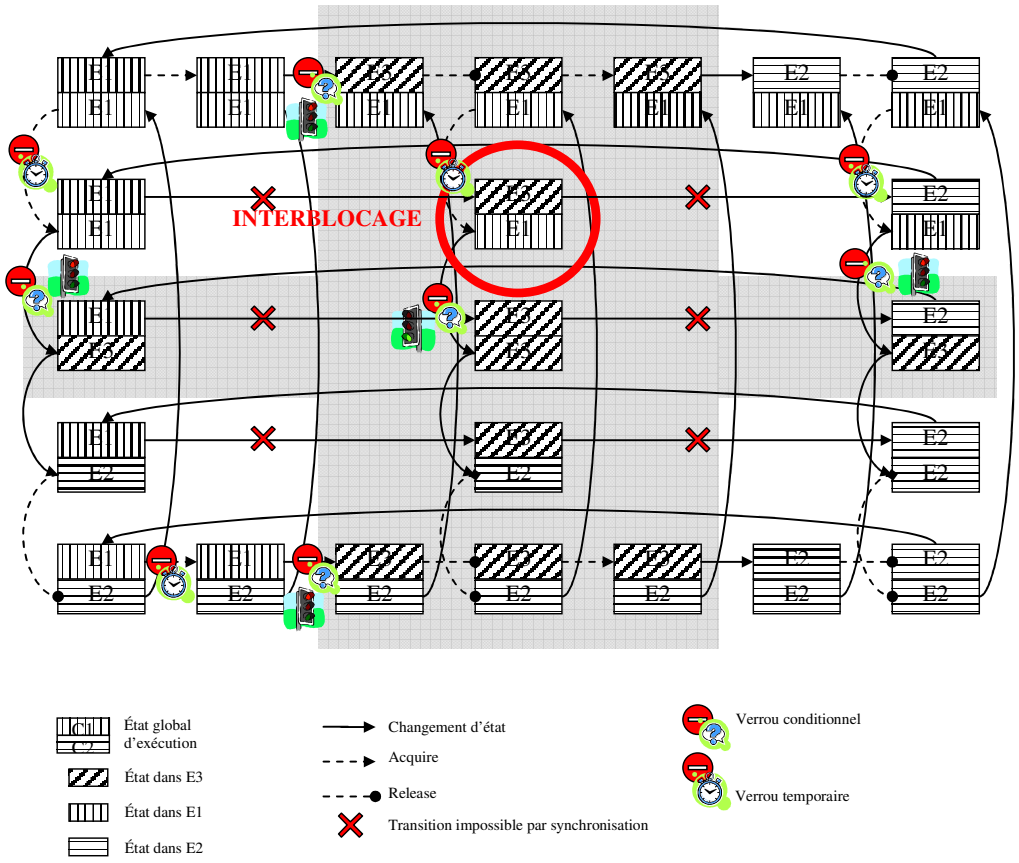


Figure 37 Combinaison des états adaptables avec section critique

Ainsi, la figure 37 représente les états du système comme le produit des états des deux tâches. L'exécution de la tâche du composant C1 suit les flèches horizontales alors que l'exécution de la tâche du composant C3 suit les flèches verticales. Lorsqu'une tâche n'est pas dans un état adaptable, le système est globalement non-adaptable. Ces états sont grisés sur la figure. En plus des transitions faisant passer le système d'un état Ei à un état Ej, nous rendons visibles les évènements qui correspondent à la synchronisation ainsi que les transitions qui existaient dans le cas de tâches indépendantes et qui sont rendus inaccessibles par l'utilisation d'une section critique. La propriété de vivacité est le fait qu'un état possède toujours au moins une transition vers un autre état. Ainsi, on remarque que l'utilisation d'un verrou inconditionnel sur le passage de l'état E1 à l'état E3 pour la tâche C2.task1 provoque un état puits (E3,E1) qui ne possède plus de transitions amenant à d'autres états du système. Cette place amène le système dans un état non vivace. Il y a donc interblocage (place encerclée sur la figure). L'utilisation d'un verrou conditionnel permet d'éviter ce phénomène. Si le système est dans l'état (E3,E1), une tâche est dans l'attente du sémaphore pour sortir de l'état non-adaptable. Ce sémaphore est relâché dans le futur de l'exécution de la tâche à bloquer. La transition est alors permise. La tâche entre alors dans son état non-adaptable afin de relâcher le jeton

nécessaire à la tâche en attente. Si le système est dans les états (E1,E1) ou (E2,E1), aucune tâche n'est dans un état non-adaptable. Le verrou empêche alors de rentrer dans un état non-adaptable, i.e. (E1, E3) ou (E2, E3). L'utilisation du verrou temporaire permet de pousser l'exécution de la tâche C1.task1 en dehors de son état E3 en empêchant la tâche C2.task1 de réacquérir le sémaphore. Une fois l'état adaptable atteint, ce verrou est relâché. Le système peut donc continuer à s'exécuter dans son état adaptable, puisque les verrous conditionnels l'empêcheront de rentrer dans un état non-adaptable.

Ainsi, une fois un état adaptable atteint, le contrôle empêche toute évolution vers un état non-adaptable. L'adaptation du logiciel, c'est-à-dire l'application des modifications (composants, paramètres, état) peut être effectuée en étant assuré que les spécifications seront respectées sur le cycle en cours pour toutes les tâches du système.

III.4.2.2 Influence de l'antériorité

L'antériorité d'évènements comme lors de l'échange de messages entre les tâches du système contraint considérablement l'état d'adaptable global du système. Nous allons montrer que toute la difficulté repose sur l'appréhension du futur de l'exécution globale du système, et donc, que l'état adaptable ne peut plus être déterminé à partir de l'évaluation d'un invariant à partir du modèle. Ainsi, prendre en compte l'antériorité dans la détermination de l'état adaptable du système nécessite d'introduire des mécanismes de contrôle de l'exécution sur le système, pour maîtriser son futur, et donc le faire converger dans son état adaptable.

Pour illustrer cette section, nous proposons d'étudier un exemple simple faisant apparaître une contrainte d'antériorité. Cet exemple est illustré par la figure 38. Deux tâches partagent une contrainte d'antériorité. Nous cherchons à remplacer le composant C4 par un composant C4'.

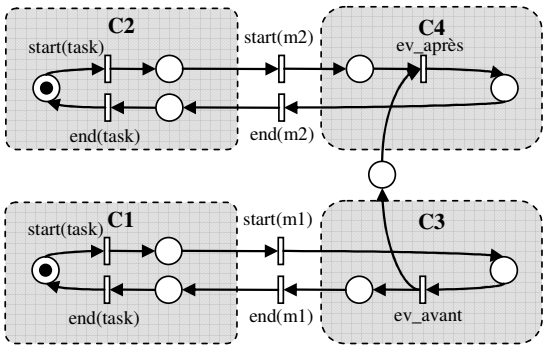


Figure 38 Exemple d'exécution avec antériorité

Le graphe des états adaptables locaux des tâches de cette architecture pour le remplacement de C4 par C4' est représenté à la figure 39.

Il existe deux phénomènes induits par l'antériorité :

- **Perte des messages/évènements :** lorsque l'on applique les modifications sur le système, nous supposons que tout message en transit vers un composant à modifier est susceptible d'être perdu. Cette perte peut être envisageable, ou non. En l'absence d'informations, nous pensons nécessaire de ne pas perdre de messages.
- **Corrélation des spécifications entre tâche émettrice et tâche réceptrice :** lorsqu'un message est échangé entre deux tâches du système dans une configuration, le contenu du message dépend de ce qui a été exécuté par la tâche émettrice avant l'envoi et donc des spécifications qu'elle a respectées. De plus, le futur de cette tâche est conditionné par le

message qui a été envoyé. La tâche réceptrice attend un contenu particulier dans configuration, par exemple en termes de format du message. La suite de son exécution est conditionnée par les données véhiculées par le message et donc, indirectement par les spécifications respectées par la tâche émettrice. Ainsi, nous considérons que :

- La tâche émettrice est contrainte de respecter les mêmes spécifications avant et après l'envoi d'un message.
- La tâche réceptrice est contrainte de respecter les mêmes spécifications que la tâche émettrice.

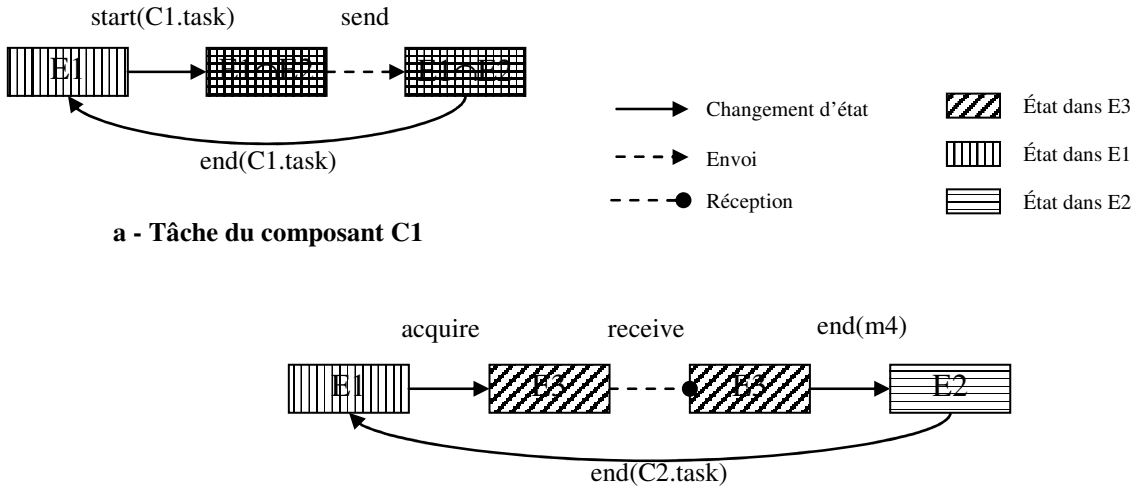


Figure 39 États adaptables des tâches avec antériorité

Il est nécessaire de vider la place représentant l'antériorité. Lorsque cette place n'est pas vide et que la consommation du jeton est réalisée dans un état non-adaptable d'une tâche, cette tâche doit nécessairement consommer ce jeton, et donc rentrer dans cet état non-adaptable. De plus, si une tâche réceptrice est en attente d'un message dans son état non-adaptable, ce message doit être envoyé, sous peine de créer un interblocage.

Ainsi, nous proposons d'étendre les techniques de contrôles introduites dans la section précédente pour les sémaphores, en introduisant la notion de verrou temporaire conditionnel. Ainsi les mécanismes utilisés sont :

– Les verrous conditionnels : ils bloquent l'exécution d'une tâche t_i si
  - aucune tâche dans un état non-adaptable n'est ou ne sera en attente de consommer un jeton (transit) émis dans le future de t_i
  - il n'y a aucun message en transit consommé par un composant à modifier
– Les verrous temporaires conditions : identiques aux verrous conditionnels, ils sont cependant relâchés lorsque l'état adaptable est atteint s'ils respectent une condition de relâchement qui est : pas d'envoi de message vers un composant à modifier. Ils sont utilisés pour agrandir temporairement l'ensemble des états non adaptables en incorporant les états entre production/consommation d'éléments en transit et un état non adaptable. Ils permettent de garantir la convergence vers un état adaptable en empêchant la production de messages qui nécessiterait que d'autres tâches entrent dans leur état non-adaptable.

Le contrôle se fait alors en deux temps :

- La synchronisation des cycles des tâches : pas de message en transit vers ou en provenance d'une tâche impactée par la reconfiguration. Ce problème est un problème bien connu et facilité par la connaissance de la présence de messages/évènements en transit fournie par le modèle ;
- La synchronisation des états d'adaptation : chaque tâche est amenée dans un état d'exécution permettant l'adaptation (E1 ou E2) et cohérent avec les états d'exécution des tâches avec lesquelles elle communique.

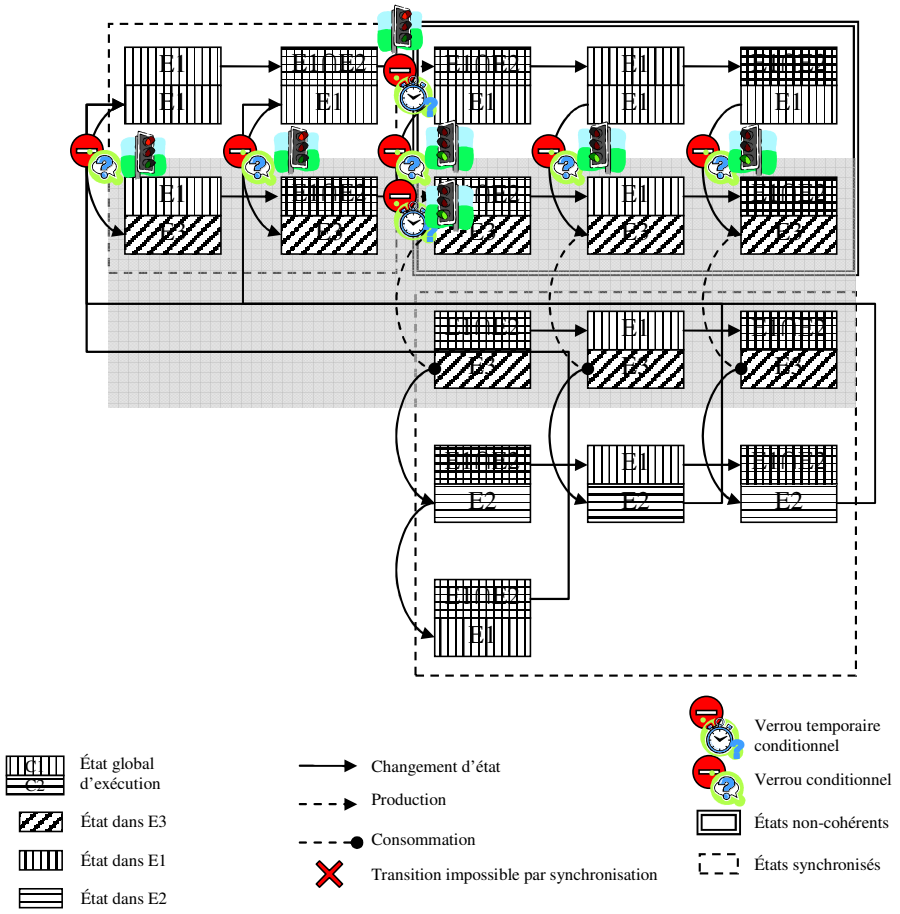


Figure 40 **Contrôle de l'exécution, perte de messages et causalité**

Ce deuxième point est illustré à la figure 40. La synchronisation amène le système dans un état parmi les états globalement synchronisés. Ces états sont encadrés par des pointillés. Dans de tels états, il n'y a pas de messages en transit. Toutefois, ces états ne sont pas nécessairement des états adaptables. Dans ce cas d'étude, nous remarquons que si après synchronisation, le système est dans un état adaptable, alors un état adaptable global est atteint. Cela est principalement dû au fait que la tâche du composant C1 n'est pas modifiée pendant l'adaptation. Cette tâche respecte alors les spécifications E1∩E2, et permet de conserver la causalité dans chacun de ses états adaptables.

Dans le cas où après synchronisation, le système n'est pas dans un état adaptable, les verrous permettent de faire converger le système vers un état globalement adaptable. Par exemple, si le système est dans (E1∩E2, E3) après synchronisation (i.e. à l'intérieur d'une zone en pointillée), alors le verrou conditionnel temporisé laisse évoluer le système. En effet, dans cet

état, la tâche C2.task1 est en attente d'un message en provenance de la tâche C1.task1. La tâche doit envoyer ce message. La tâche C2.task1 peut alors boucler et amener le système dans l'état (E1∩E2, E3). Toutefois, cet état est alors non-synchronisé. Le verrou empêche alors l'émission du message, puisque le message attendu par la tâche C1.task1 est déjà en transit. C'est pourquoi la figure 40 ne fait pas apparaître de transition correspondant à l'envoi d'un message à partir de cet état, car la transition correspondante est rendue non-franchissable par le verrou. Le système n'a d'autre choix que d'évoluer vers un état adaptable synchronisé. On remarque alors que les états adaptables non synchronisés ne sont pas accessibles par l'exécution dans ce cas précis. En effet, le composant C4 qui doit être adapté est le destinataire des messages. Afin de ne pas perdre de message, le verrou conditionnel temporaire empêchant l'envoi de messages vers C4 à partir d'un état globalement adaptable n'est jamais relâché. Lorsque qu'un état adaptable est atteint, les modifications peuvent être réalisées sur le système sous la garantie de respecter localement à chaque tâche les spécifications comportementales, mais aussi en respectant la causalité introduite par les messages.

III.4.3 Reconfiguration transactionnelle et tolérance au crash

Dans le but de permettre le recouvrement du processus de reconfiguration, nous proposons de le rendre transactionnelle. Ce niveau de reconfiguration utilise les deux niveaux précédemment introduits qui permettent de placer le système dans un état adaptable et d'y maintenir le système.

Ainsi, la reconfiguration est initiée (*init*), puis toutes les modifications à réaliser sur le système sont spécifiées, et enfin, la reconfiguration est appliquée (*commit*). Lorsque la reconfiguration est appliquée, le système est placé dans un état adaptable (tâche localement adaptable respectant les contraintes de synchronisation) qui tient compte des modifications qui doivent lui être appliquées (composants et connexions modifiés). Lorsqu'un état permettant l'adaptation est atteint, l'état des composants à modifier est alors sauvegardé sur un support de stockage fiable. Les modifications sont ensuite appliquées sur le système. Si une erreur est détectée (impossibilité de créer ou supprimer des composants et déconnecter ou connecter des composants ou crash d'un nœud impliqué dans le processus de reconfiguration), les modifications effectuées sont rembobinées, et l'état des composants est rechargé à partir du support de stockage fiable. Une fois les modifications (ou leur abandon) réalisées, le contrôle de l'exécution est relâché. La reconfiguration se termine alors soit par un échec entrainant ainsi une poursuite du fonctionnement du système dans l'ancienne configuration, soit par un succès entrainant une poursuite du fonctionnement du système dans la nouvelle configuration.

III.4.4 Automatisation de la reconfiguration

Nous avons vu dans la section III.2.1 qu'il est nécessaire de décrire la configuration du système afin de pouvoir déduire ce qui doit être modifié dans le système. De plus, il est nécessaire d'introduire des mécanismes réflexifs pour paramétrer les composants, ces mécanismes n'étant pas disponibles sous le modèle à composant OpenCOM, nous détaillerons leur implémentation.

Cette section introduit une description de ce qu'est une configuration d'un point de vue statique. À partir de cette description, nous montrons comment en déduire les modifications à réaliser sur l'architecture à composant, afin de mettre en place cette configuration à partir d'une configuration déployée. Finalement, nous détaillerons les mécanismes de paramétrage d'un composant que nous avons mis en place.

Une architecture à composants peut être représentée par un ensemble de graphes. Le premier graphe représente la composition au sens des connexions entre les composants. Le deuxième graphe représente la composition au sens du contenu des composants composite. À partir de ces représentations, il est alors possible de comparer l'architecture déployée avec la nouvelle configuration à mettre en place pour en déduire les actions à réaliser sur le modèle architectural pour obtenir la nouvelle configuration à partir de celle actuellement déployée.

III.4.4.1 Description d'une configuration

La configuration du système est une architecture à composant répartie sur un ensemble de nœuds du système, ainsi qu'un ensemble de paramètres pour chacun des composants. Ainsi, la description de la configuration doit d'une part refléter cette architecture à composant, et d'autre part, spécifier les paramètres des différents composants de cette architecture.

Plusieurs travaux se sont intéressés à développer des langages de description pour des architectures à composants. Ces langages de description ont par exemple permis d'automatiser le déploiement d'architectures à composants sous OpenCOM, au travers d'une plateforme appelée Plastik [68]. Le langage de description de l'architecture (ADL[7]) à composants utilisés dans cette plateforme est ACME [69]. Il introduit un langage reposant sur les ontologies qui permet de jongler entre diversité des architectures et l'universalité d'une description. D'autres modèles à composant ont introduit leur propre ADL comme par exemple dans Fractal [70] à partir duquel une architecture à composant est automatiquement déployée.

Nous proposons d'utiliser les mêmes techniques, non pas uniquement pour le déploiement d'architectures à composants, mais pour déterminer leur modification. Ainsi, à partir de cette description, ainsi que de la représentation de l'architecture à composants du système, obtenue par le modèle, nous proposons de calculer les modifications à effectuer sur le système.

Nous avons choisi de créer notre propre ADL adapté à nos besoins. Ce choix est motivé par le fait que nous pouvons étendre la description de l'architecture comme nous le souhaitons, et plus particulièrement, en introduisant la description des chemins d'exécution internes à un composant. Ainsi, comme nous le verrons par la suite, cet ADL nous permet de décrire la configuration d'un point de vue purement statique, mais aussi du point de vue dynamique.

Dans cette section, nous nous focalisons uniquement sur les aspects statiques de la description. Ces aspects se décomposent en une description des composants et de l'architecture. Ces descriptions sont contenues dans un fichier XML, permettant leur manipulation pendant l'exécution du système. Nous présenterons ensuite un ensemble d'outils que nous avons mis en place dans le but de générer et manipuler ces documents.

Description d'un composant

La description statique d'un composant passe par la connaissance de ses interfaces, ses réceptacles et leur type, ainsi que ses paramètres. Sachant que la version utilisée repose sur Java et ses concepts Orientés Objets, nous proposons de conserver la notion d'héritage dans la description. Ainsi, un composant héritant d'un autre composant hérite de ses interfaces, de ses réceptacles et de ses paramètres.

Le document XML permettant de décrire un composant est constitué des nœuds détaillés dans le tableau 6.

[7] Architecture Description Language

Tableau 6 **Format de l'ADL pour la description d'un composant**

Noeuds	*Nœuds parents*	*Attributs*	*Description*
definition	∅	clsid	Type du composant
		composite (optionel)	Booléen mis à vrai si le composant est composite
		super	Type du composant dont ce composant hérite
interfaces	definition	∅	Déclaration des interfaces du composant
interface	interfaces	iid	Identifiant de l'interface
receptacles	definition	∅	Déclaration des réceptacles du composant
receptacle	receptacles	iid	Identifiant du réceptacle
		type	Type du réceptacle. Cela permet par exemple de savoir si ce réceptacle admet plusieurs connexions
parameters	definition	∅	Déclaration des paramètres du composant
parameter	parameters	name	Nom du paramètre
		type	Type du paramètre

Ainsi, le document de la figure 41 décrit le type de composant *package.NomDeLaClasse* qui n'est pas un composant composite et qui hérite du composant de type *package.NomDeLaClasseParente*. Ce composant possède deux interfaces. La première correspond à la classe *package.Interface1* et la deuxième à la classe *package.Interface2*. Ce composant possède un réceptacle de type *OCM_SingleReceptacle* qui n'admet qu'une seule connexion et qui peut être connecté à une interface de type *package.Interface3*. Ce composant est paramétrable grâce à un unique paramètre appelé *param1* de type *package.NomDeLaClasseDuParametre*.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definition clsid="package.NomDeLaClasse" composite="false"
super="package.NomDeLaClasseParente">
   <interfaces>
      <interface iid="package.Interface1"/>
      <interface iid="package.Interface2"/>
   </interfaces>
   <receptacles>
      <receptacle iid="package.Interface3" type="single"/>
   </receptacles>
   <parameters>
      <parameter name="param1" type="package.NomDeLaClasseDuParametre"/>
   </parameters>
</definition>
```

Figure 41 Exemple de description de composant

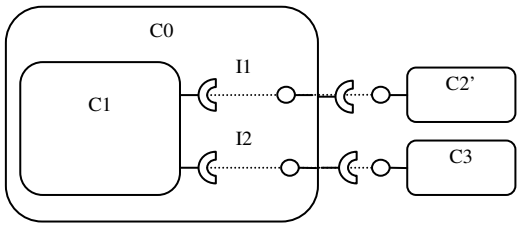
Description d'une architecture

La description d'une architecture à composant doit permettre de spécifier d'une part l'arborescence des composants en termes de contenant et contenu et d'autre part leur agencement en termes de connexions. Le document XML spécifiant l'architecture à composants est formaté selon les nœuds XML détaillés dans le tableau 7.

Tableau 7 **Format de l'ADL pour la description d'une architecture**

Nœuds	*Nœuds parents*	*Attributs*	*Description*
architecture	∅	name	Nom de l'architecture
components	architecture	∅	Ensemble des composants externes à tout composant composite
bindings	architecture	∅	Ensemble des connexions externes à tout composant composite
component	components	name	Nom du composant
		started	État du composant (*true* le composant est démarré *false* le composant est arrêté)
		type	Type du composant
components	component	∅	Ensemble des sous composants contenu dans un composant composite
bindings	component	∅	Ensemble des connexions à l'intérieur du composant composite
parameters	component	∅	Déclaration des paramètres du composant
exposed_interfaces	component	∅	Interfaces exposées
exposed_receptacles	component	∅	Réceptacles exposés
binding	bindings	iid	Identifiant de l'interface à connecter
		sink	Nom du composant implémentant l'interface
		source	Nom du composant possédant le réceptacle
parameter	parameters	name	Nom du paramètre
		value	Nom du fichier contenant la valeur du paramètre
exposed_interface	exposed_interfaces	iid	Identifiant de l'interface à exposer
exposed_receptacle	exposed_receptacles	iid	Identifiant du réceptacle à exposer
		type	Type du réceptacle

Ainsi, la figure 42 montre la description d'une architecture à composant à l'aide de cet ADL. Cette architecture est composée de quatre composants. C0 est un composant composite qui expose deux réceptacles correspondants aux interfaces *Interface1* et *Interface2*. Ce composant contient le composant C1 dont les deux réceptacles sont connectés aux interfaces internes du composant composite qui correspondent aux réceptacles exposés. Les réceptacles exposés de C0 sont connectés à deux composants *C2'* et C3, implémentant respectivement l'interface *Interface1* et *Interface2*.




```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<architecture name="Console-Distributed">
   <components>
      <component name="C0" started="true" type="opencom.components.CFMetaInterface">
         <parameters/>
         <exposed_interfaces/>
         <exposed_receptacles>
            <exposed_receptacle iid="I1"
               type="opencom.receptacles.OCM_SingleReceptacle"/>
            <exposed_receptacle iid="I2"
               type="opencom.receptacles.OCM_SingleReceptacle"/>
         </exposed_receptacles>
         <components>
            <component name="C1" started="true" type="components.Component1">
               <parameters/>
            </component>
         </components>
         <bindings>
            <binding iid="I1" sink="C0" source="C1"/>
            <binding iid="I2" sink="C0" source="C1"/>
         </bindings>
      </component>
         <component name="C2'" started="true" type="components.Component2bis">
      </component>
         <component name="C3" started="true" type=" components.Component3">
      </component>
   </components>
   <bindings>
      <binding iid="I1" sink="C2'" source="C0"/>
      <binding iid="I2" sink="C3" source="C0"/>
   </bindings>
</architecture>
```

Figure 42 Exemple de description d'une configuration

Outils d'ingénierie

Ces langages de description doivent être intégrés dans le processus de conception du système. Pour faciliter leur utilisation, des outils peuvent être mis en place. Dans Fractal par exemple, un certains nombre d'outils logiciels ont été réalisés pour faciliter la conception d'un système à partir de composant Fractal. Citons par exemple Fractal ADL (langage ADL dédié à Fractal), les Fraclets (annotations java pour les composants), Fractal Explorer (pour la visualisation et la modification d'architecture à composant de manière graphique), FPath et FScript (langages de parcours d'ADL et de modification d'architecture à composant), et FractalGUI (pour l'édition graphique d'architecture à composants Fractal).

Ces outils n'existant pas pour le modèle à composant OpenCOM, nous en avons développé deux qui seront détaillés dans le chapitre IV. Il s'agit d'annotations Java pour la fabrication des fichiers descriptifs des composants, et d'un outil graphique pour la fabrication d'architecture à composants OpenCOM.

III.4.4.2 Construction des graphes

Dans chacun des deux graphes, une place représente un unique composant. Cette place est identifiée par le couple (type, nom) du composant qu'elle représente. Un arc relie toujours deux places.

Dans le **graphe des connexions**, un arc représente une connexion entre deux composants. L'origine de l'arc est la place représentant le composant qui détient le réceptacle, et la cible de l'arc est la place représentant le composant qui implémente l'interface attaché. Un label est associé à cet arc et représente le type de l'interface impliquée dans la connexion.

Dans le **graphe de composition**, un arc représente l'inclusion d'un composant dans un autre. L'origine de l'arc est le composant contenu, et la flèche pointe vers le composant composite contenant.

Ainsi, les graphes représentant l'architecture à composant de la figure 42 sont décrits dans la figure 43. Pour simplifier la figure, nous avons représenté les places du graphe uniquement par le nom du composant.

Figure 43 Graphes de composition et de connexion

Une représentation plus compacte consiste à fusionner les deux graphes en un seul en différenciant les arcs représentant les connexions des arcs représentant la composition. Cette représentation est plus compacte, et a été choisie dans l'implémentation. Toutefois, pour des soucis de lisibilité, nous garderons une représentation séparée des deux graphes.

III.4.4.3 Détermination des modifications architecturales

Les actions à réaliser sur le modèle architectural sont déduites de la comparaison des graphes entre la configuration déployée et la configuration à mettre en place. Ainsi, la disparition d'un arc dans le graphe des connexions représente une déconnexion, l'apparition d'un arc représente une nouvelle connexion. La disparition d'un arc dans le graphe de composition représente le fait qu'un composant est retiré d'un composant composite, et l'apparition d'un arc représente le fait qu'un composant est inséré dans un composant composite. L'apparition d'une place représente la création d'un nouveau composant. La disparition d'une place représente la suppression d'un composant.

III.4.4.4 Ordre partiel des actions de reconfiguration

Les actions de reconfiguration doivent être ordonnées. En effet, un composant ne peut être supprimé avant que toutes les connexions vers ou à partir de ce composant ne soient supprimées.

Dans le processus de modification de l'architecture à composant, les actions sont réalisées dans l'ordre suivant :

1. Déconnexion
2. Retrait des composants des composites les contenant
3. Suppression des composants
4. Création des composants
5. Insertion des composants
6. Mise à jour des interfaces et réceptacles exposés
7. Connexion
8. Paramétrage
9. Démarrage

Création et insertion des composants sont quelque fois confondues dans les modèles à composant, tel que dans Fractal. Toutefois, d'autres modèles proposent de conceptuellement séparer ces deux actions.

III.4.4.5 Exemple

Dans cet exemple illustré à la figure 44, nous proposons de modifier une architecture à composants déployée pour obtenir la nouvelle configuration décrite à la figure 42.

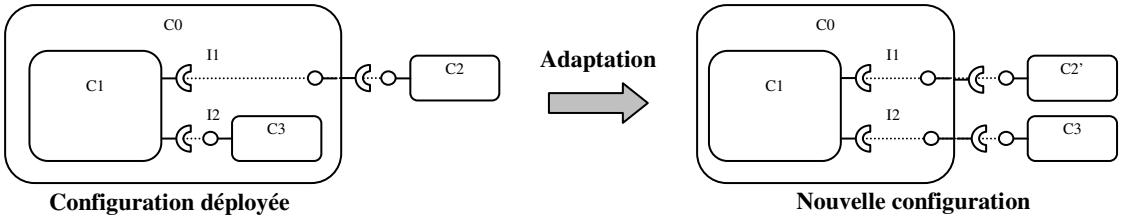


Figure 44 Exemple d'adaptation

Les graphes correspondant à la configuration à mettre en place sont ceux de la figure 43. Il est nécessaire de construire les graphes représentant la configuration courante. Ces graphes sont ceux de la figure 45.

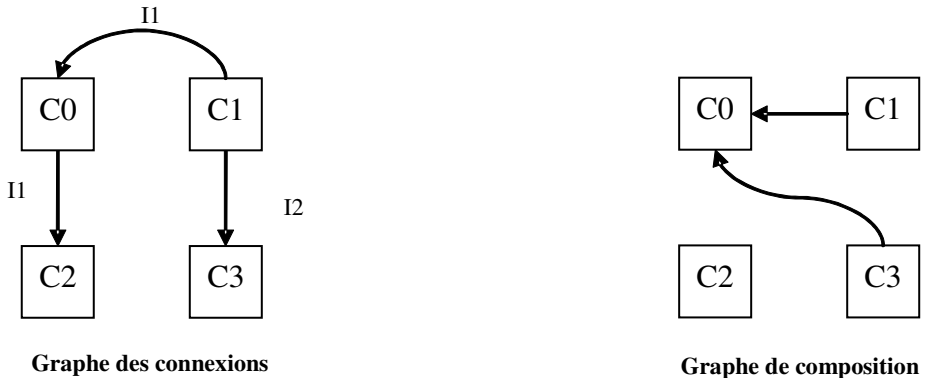


Figure 45 Graphes de composition et de connexion de la configuration déployée

Ainsi, en comparant la configuration courante avec la configuration à mettre en place, nous en déduisons les actions suivantes :

1. Déconnexions : la connexion de C1 vers C3 sur l'interface *I2* doit être supprimée ainsi que la connexion de C0 vers C2 sur l'interface *I1*
2. Retrait : le composant C3 doit être retiré du composant C0
3. Suppression : le composant C2 doit être supprimé
4. Création : le composant C2' doit être créé
5. Insertion : aucun composant ne doit être inséré
6. Exposition : le réceptacle de type *opencom.receptacles.OCM_SingleReceptacle* sur l'interface *I2* doit être exposé.
7. Connexions : le composant C1 doit être connecté à l'interface *I2* du composant C0, le composant C0 doit être connecté respectivement à l'interface *I1* et *I2* des composants C2' et C3.
8. Paramétrage : pas de paramétrage
9. Démarrage : le composant C2' doit être démarré, les autres l'étant déjà.

III.4.4.6 Reconfiguration et scripts

FPath est un langage similaire à XPath dans les documents XML. Il permet de parcourir une architecture à composant en utilisant des requêtes à base de chemin. Ainsi, un composant *C1* situé dans un composant composite *C0*, lui-même situé à la racine de l'architecture, peut être atteint en spécifiant le chemin *'/$C0/$C1'*. Ces requêtes permettent d'obtenir des informations concernant les composants, comme par exemple « Quels sont les sous-composants de *C* qui implémentent l'interface *I ?* » (⇔ *$C/child::*[./interface::$I[server(.)]]*), ou « Quels sous-composants de *C* sont partagés » (⇔ *$C/descendant-or-self ::*[size(./parent::*)>1]*).

FScript est un langage dédié à la manipulation des méta-interfaces Fractal (Controller) pour écrire des scripts de reconfiguration. Il permet de garantir la 'bonne utilisation' de ces interfaces et faciliter l'écriture d'une reconfiguration qui se faisait jusqu'à présent sous la forme d'un programme Java. Ainsi, il y a une meilleure maîtrise de la reconfiguration, car celle-ci est validée à la compilation par la vérification de propriétés à partir des documents ADL décrivant les architectures déployées et à mettre en place, et dynamiquement par la détection d'incohérence entre les propriétés des composants attendues (contrôleurs implémentés) et celles réellement observées. De plus, la reconfiguration est appréhendée

comme une transaction ce qui améliore sa fiabilité. FScript fournit alors un langage de haut niveau, plus concis que le langage Java, traditionnellement utilisé pour reconfigurer les architectures à composants, et donc améliorant la fiabilité de la reconfiguration.

Toutefois, aucun moyen n'est donné pour garantir que l'état dans lequel se trouve un ensemble de composants permet effectivement de réaliser l'adaptation. De plus, aucun moyen n'est donné pour automatiser l'écriture des scripts à partir de plusieurs descriptions.

En conclusion, ce langage est une opportunité de renforcer le processus d'adaptation, par une analyse structurelle des architectures à composants.

III.4.5 Paramétrage

Un paramètre est identifié par un nom unique. Il possède un type particulier. Ainsi, pour représenter un paramètre, nous avons créé une interface *IParameter*. Cette interface permet de manipuler un paramètre. Elle est décrite dans le tableau 8.

Tableau 8 **Détail de l'interface *IParameter***

Méthodes	*Description*
String getName()	Méthode de lecture du nom du paramètre
String getType()	Méthode retournant le type du paramètre sous la forme d'une chaîne de caractères
Object getValue()	Méthode retournant la valeur du paramètre

Nous avons vu dans la section III.2.1.2 que le paramétrage doit être réalisé de manière transactionnelle, c'est-à-dire qu'un ensemble de paramètres doit être modifié de manière transactionnelle. Nous proposons donc d'introduire une méta-interface appelée *ISetting* dans ce but. Cette méta-interface permet d'introspecter et de modifier de manière atomique les paramètres d'un composant. Le détail de cette interface est donné dans le tableau 9.

Tableau 9 **Détail de l'interface *ISettings***

Méthodes	*Description*
boolean initSettings()	Initialise une transaction sur le paramétrage. La méthode retourne false si une transaction est en cours.
boolean commitSettings()	Application du paramétrage du composant. Cette méthode sauvegarde la configuration actuelle en appelant checkpointSettings(). Si le paramétrage est correctement appliqué, le checkpoint est supprimé par l'appel de la méthode removeSettingsCheckpoint(). Le résultat de la méthode est alors true. Dans le cas contraire, la méthode retourne false et le composant est placé dans l'état avant le paramétrage par l'appel de la méthode rollbackSettings().
void declareParameter(IParameter param)	Méthode utilisée dans le constructeur pour déclarer un paramètre.
void checkpointSettings()	Méthode appelée au début d'un commit pour réaliser une sauvegarde des paramètres courants.
void removeSettingsCheckpoint()	Méthode appelée en cas de succès du commit permettant d'oublier la sauvegarde des anciens paramètres.
void rollbackSettings()	Méthode appelée en cas d'échec du commit permettant de recharger les anciens paramètres.
boolean setParameter(IParameter param)	Méthode permettant de spécifier la valeur d'un paramètre. Cette valeur est stockée dans une table qui pourra être lue lors du commit. Elle ne modifie pas le paramètre courant. Cette méthode retourne false dans le cas où le paramètre est incorrect (mauvais type ou mauvais nom) ou si aucune transaction n'est en cours.
boolean setParameters(IParameter[] params)	Méthode permettant de modifier un ensemble de paramètres. La méthode retourne false si un paramètre est incorrect (mauvais type ou mauvais nom)
IParameter getParameter(String name)	Méthode permettant de récupérer la valeur d'un paramètre.
IParameter[] getAllParameters()	Méthode permettant de récupérer tous les paramètres du composant

Le concepteur d'un composant doit alors surcharger les méthodes *commitSettings*, *checkpointSettings*, *rollbackSettings* et *getParameter* dans le but de réaliser les traitements spécifiques à son composant.

III.5 CONCLUSION

L'adaptation comme nous souhaitons l'aborder est un problème complexe. Elle met en jeu des problèmes de restructuration du logiciel (modification de l'architecture à composant) et de modification de son état (paramétrage, transfert d'état) sous des contraintes d'exécution complexes.

Nous nous sommes intéressés à définir ce que pouvait être une adaptation correcte d'un point de vue générique pour un type de système particulier composé uniquement de tâches cycliques dont l'exécution est bornée dans le temps. Cette correction découle directement du respect des spécifications dynamiques du système, en termes de trace d'exécution. Ainsi, une adaptation est dite correcte si aucun comportement non-prévu par les spécifications des configurations avant et après adaptation n'est introduit. Nous avons montré que cette définition influence directement le choix d'un état d'exécution pour réaliser l'adaptation. Un état permettant l'adaptation est alors appelé état adaptable du système. Dans un système réaliste où échanges de messages et synchronisation ont lieu, nous avons justifié les contraintes existantes sur les états adaptables, en termes de causalité par exemple. Nous avons montré qu'il est possible de définir des mécanismes de contrôle de l'exécution des tâches permettant d'amener le système dans un état adaptable en conservant des propriétés de causalité et de vivacité, et de l'y maintenir le temps de l'adaptation.

Dans ce chapitre, nous avons laissé de côté les problèmes liés au transfert de l'état qui pourraient à eux seuls constituer le sujet de travaux de recherche intéressants pour nous focaliser sur l'automatisation des modifications architecturales et paramétrique du logiciel. Cette automatisation est réalisée au travers d'une comparaison du logiciel déployé et d'une description haut-niveau de la configuration souhaitée. À partir de cette comparaison, nous avons montré comment déduire les modifications à apporter au logiciel.

Nous proposons dans le chapitre suivant d'appliquer ces travaux au logiciel de tolérance aux fautes. Cela nécessite de concevoir le logiciel de tolérance aux fautes sous la forme de composants logiciels respectant le modèle de tâche introduit dans ce chapitre. Nous détaillerons dans le prochain chapitre les étapes en termes de conception qui permettent d'utiliser ces résultats pour adapter un logiciel critique : le logiciel de tolérance aux fautes.

Chapitre IV

ÉTUDE DE CAS : DEVELOPPEMENT D'UN SYSTEME ADAPTATIF TOLERANT LES FAUTES

IV.1 INTRODUCTION

Dans ce chapitre, nous proposons d'appliquer nos travaux à un cas d'étude. Dans un premier temps, nous montrons comment le logiciel de tolérance aux fautes peut être conçu en utilisant les composants (CBSE[8]). Dans un deuxième temps, nous détaillerons les outils logiciels d'aide à la conception que l'on a mis en place. Ces outils permettent de générer les documents qui décrivent les composants et les architectures. Ces documents permettent de décrire une configuration logicielle du système. Enfin, nous illustrerons ensuite les travaux de thèse en montrant comment l'adaptation est mise en place à l'exécution. Cela sera réalisé au travers d'un cas d'étude qui a été employé au cours d'un mini-projet appelé ASAP (ASsessment based AdaPtation) dans le cadre du réseau d'excellence européen ReSIST (Resilience for Suvavility in IST). Ce cas d'étude consiste à mettre à jour un logiciel fonctionnel en introduisant une phase d'évaluation des nouvelles versions dans leur environnement. Une fois cette évaluation effectuée, soit le composant a été jugé suffisamment sûr de fonctionnement et est déployé à la place de l'ancien, soit le composant contient trop de fautes résiduelles et est rejeté. Pour optimiser les performances et la consommation des ressources, le logiciel de tolérance aux fautes est alors automatiquement adapté à la nouvelle version.

[8] Component Based Software Engineering

IV.2 CONCEPTION DU LOGICIEL DE TOLERANCE AUX FAUTES

IV.2.1 Problématique de l'adaptation et conception

Notre objectif est de guider le processus de décomposition du logiciel de tolérance aux fautes dans le but de faciliter sa modification en ligne. Lors de la reconfiguration à l'exécution, deux modifications peuvent être réalisées : le changement d'un paramètre, et le changement d'un algorithme. La conception orientée composant permet de rendre modulaire les algorithmes utilisés, et donc faciliter la gestion de leur modification pendant l'exécution.

Lors de la reconfiguration, le plus petit bloc de l'algorithmique du logiciel qui peut être modifié est un composant. Dans le modèle à composant, les frontières d'un composant sont définies par ses interfaces et ses réceptacles. Ainsi, concevoir la tolérance aux fautes sous la forme de composant consiste à définir les interfaces des algorithmes susceptibles d'être modifiées au cours de la vie opérationnelle du système. Cette conception est fortement connectée aux besoins d'adaptation du système et de sa tolérance aux fautes. Ainsi, si l'adaptation consiste à changer la manière dont les points de reprise sont sauvegardés, la tolérance aux fautes sera monolithique. Seul l'algorithme de prise de points de reprise sera un composant à adapter. Toutefois, il doit être souligné que, par la suite, il sera extrêmement difficile d'intégrer de nouveaux besoins d'adaptation. Nous allons donc proposer une décomposition générique de la tolérance aux fautes pour permettre l'adaptation de n'importe quel constituant du logiciel. Cela a deux intérêts majeurs : tout d'abord, il est possible de définir a postériori de nouveau besoins d'adaptation et ensuite, l'adaptation peut être utilisée pour faire de la maintenance à chaud du logiciel, par exemple, pour remplacer une partie du logiciel dont les bugs ont été corrigés.

De plus, reconfigurer dynamiquement du logiciel nécessite d'être capable d'une part de contrôler l'exécution du système pour le placer dans un état qui permet la reconfiguration et d'autre part de transférer l'état des composants supprimés vers les composants nouvellement insérés. Ces problèmes sont détaillés dans le chapitre 3. Le même service peut être rendu par un ensemble de composants qui coopèrent ou un unique composant plus gros. Il s'agit en fait d'un choix de finesse de décomposition. De ce fait, lorsque l'on est en présence de plusieurs composants très petits, la supervision de l'exécution devient plus complexe, au profit d'un état souvent très simple. Inversement, lorsque l'on utilise des composants plus gros, la supervision de l'exécution peut être plus simple, au détriment d'une complexité accrue de l'état. Ainsi, il est nécessaire de trouver un compromis au niveau de la finesse de la décomposition. Toutefois, il est aussi nécessaire de garder à l'esprit que les composants sont les briques élémentaires de l'adaptation, et donc qu'une décomposition grossière empêche une adaptation fine du logiciel.

Lorsque la tolérance aux fautes est implémentée à l'aide de composants, l'état de la tolérance aux fautes est réparti dans chaque composant impliqué dans la stratégie de tolérance aux fautes. Cela concerne aussi les composants fonctionnels wrappés par l'*ApplicationController*. Pendant la reconfiguration du logiciel de tolérance aux fautes, l'état du logiciel implémentant la stratégie en cours d'exécution doit être transféré à la nouvelle stratégie de tolérance aux fautes mise en place. Une grosse partie de cet état est persistant pendant l'adaptation. En effet, il est conservé à l'identique. Notre idée est de déterminer cet état, dans le but de l'isoler dans des composants qui ne seront donc pas modifiés pendant la reconfiguration. Ainsi, l'état de ces composants sera naturellement transféré d'une configuration à l'autre, puisqu'il sera

contenu dans des composants qui seront conservés. Cette approche au niveau de la conception, évite une gestion trop lourde de la gestion de l'état pendant le processus de reconfiguration. Cet état qui est conservé est constitué des données internes de l'application sauvegardées sur un support de stockage, des messages en transit sur les canaux de communication ou des alarmes par exemple.

IV.2.2 Analyse de la tolérance aux fautes pour sa décomposition

Le logiciel de tolérance aux fautes repose sur des algorithmes qui font son cœur de métier, et d'autres qui sont plus générique. Notre décomposition doit exhiber les mécanismes du domaine de la tolérance aux fautes (détection, recouvrement) ainsi que les mécanismes plus génériques (communication, horloge, etc.). Les mécanismes de tolérance aux fautes utilisent les services génériques. Il existe donc des dépendances (architecturales et fonctionnelles) entre ces deux types de composants.

Lors de l'adaptation, les modifications vont cibler les mécanismes de détection et de recouvrement, et la manière dont ils sont utilisés. Ces modifications vont impacter les services génériques et les abstractions de tolérance aux fautes dont les mécanismes dépendent. Deux types de modifications vont alors être opérés : soit ces composants vont être arrêtés et retirés du système s'il n'existe plus de dépendance avec des composants existants, soit ils vont être conservés. Lorsqu'ils sont conservés, l'état qu'ils contiennent est dans de nombreux cas suffisant pour résoudre les problématiques de transfert d'état. L'état des composants nouvellement insérés peut être initialisé avec une valeur par défaut.

Faire apparaître les services génériques est intéressant du point de vue de l'adaptation. Ils permettent d'une part de diminuer la taille du code chargé en mémoire et d'autre part de minimiser l'impact des modifications effectuées pendant une adaptation.

IV.2.3 Proposition de décomposition

IV.2.3.1 Les services génériques

De notre point de vue, une grande partie des services génériques servent à utiliser les ressources du système. Nous référençons trois familles de services génériques liés aux ressources du système :

- **les services de communication**. Nous considérons que, dans un système embarqué, la communication est réalisée au travers de canaux de communication qui ne sont pas modifiés en cours de fonctionnement autrement que par le processus d'adaptation. Ainsi, les services rendus par ces canaux de communication consistent à envoyer et recevoir des messages, soit de manière synchrone (attente bloquante) soit de manière asynchrone (notification de nouveau message). Nous introduisons donc trois interfaces : *IMessageSend* pour l'envoi de message, *IMessageReceive* pour la réception bloquante de message, et *IMessageListen* pour la réception par notification de messages. Ces canaux de communication peuvent fournir de la communication de groupe ou point à point. Une interface *IGroup* permet la consultation des informations sur les membres d'un groupe, et une interface *IGroupListen* permet d'être spontanément averti des modifications d'un groupe.
- **les services de stockage**. La tolérance aux fautes utilise des services de stockage pour sauvegarder certaines données nécessaires au recouvrement (par exemple, les points de reprise) ou à la maintenance (les logs). Les services de stockage peuvent être introduits à plusieurs niveaux d'abstraction : niveau support où le service permet de gérer le médium

utilisé, niveau système de fichier où le service permet de gérer les fichiers écrits sur le médium, niveau fichier où le service permet de gérer les données inscrites dans un fichier. Ces trois niveaux d'abstraction sont hiérarchisés. Le fichier nécessite un gestionnaire de fichier pour exister et repose sur l'utilisation d'un support de stockage, et le gestionnaire de fichier nécessite un support sur lequel gérer les fichiers. Ces trois niveaux d'abstraction peuvent être introduits dans la décomposition. Ainsi, l'adaptation peut avoir lieu au niveau d'un fichier, de la manière dont ceux-ci sont gérés sur le support de stockage ou sur le support de stockage lui même. Toutefois, l'adaptation de l'un des services impose une modification de l'état des deux autres relativement lourde, nécessitant la copie de nombreuses données d'un medium à l'autre ou leur conversion par exemple. Dans nos travaux, nous choisissons une approche alternative qui consiste à regrouper ces trois services en un seul, plus restrictif représentant des flux de données vers un support de stockage et représenté par l'interface *IStorage*. Ces flux de données sont représentés par un identifiant (le fichier) sur lequel deux actions sont possibles (l'écriture et la lecture). L'implémentation de ce service par un composant masque alors les propriétés du support de stockage au composant client. Ainsi, le composant le service *IStorage* peut masquer l'utilisation d'un support de stockage stable, ou d'un support stable distribué, non fiable.

– **les services de gestion du temps**. Le temps est une ressource importante du système. En tolérance aux fautes, le temps peut être utilisé pour estampiller des données (de journalisation par exemple), pour mesurer l'écoulement du temps, ou encore pour effectuer des actions dans un intervalle de temps déterminé (*watchdog*). Ainsi, il existe trois services fondamentaux concernant le temps : la consultation du temps réel *IClock*, la consultation d'un temps logique *ILogicalClock*, et les alarmes *IAlarm*.

À partir de ces services de base, il est possible de construire des services plus complexes. Ainsi, nous avons introduit un service d'élection *IElection*, un service de journal *ILog* qui permet de gérer des enregistrements sur un support de stockage et des *stubs* et *skeletons* qui utilisent les services de communication afin de réaliser des appels de procédures distantes.

Un *stub* est la représentation locale d'un service rendu par un composant distant. Le skeleton est le dual du *stub*. Il représente localement à un fournisseur de service un ou plusieurs clients distants. Le stub implémente alors la même interface que le composant serveur, et le skeleton implémente un réceptacle compatible avec l'interface du composant serveur. Lorsqu'un client est connecté à un stub, il réalise des appels en direction du stub comme s'il s'agissait du composant serveur. Le stub formate l'appel sous la forme d'une requête en sérialisant les arguments de l'appel. Ensuite, il envoie cette requête via le composant de communication à un skeleton (ou plusieurs si le composant de communication implémente une communication de groupe). Le skeleton est notifié de la requête, la transforme en appel de procédure sur le composant serveur. Ce dernier retourne le résultat qui est à son tour formaté sous la forme d'une réponse, et envoyé au stub. Le stub retourne ainsi la réponse du composant serveur au composant client.

IV.2.3.2 Les abstractions issues de la taxonomie

Nous proposons d'utiliser la taxonomie de la sûreté de fonctionnement, et plus particulièrement celle de la tolérance aux fautes introduite dans [1] pour guider la décomposition. Cette taxonomie introduit les concepts de stratégies et mécanismes de tolérance aux fautes. Cette décomposition nous semble intéressante, car elle permet de faire apparaître les cœurs de métier de la tolérance aux fautes. Une stratégie de tolérance aux fautes vise à couvrir un certain modèle de faute du système, afin d'éviter que le système ne défaille. Une stratégie repose sur l'utilisation de mécanismes de tolérance aux fautes. Ces mécanismes

sont les briques élémentaires de la tolérance aux fautes. Ils permettent de détecter et recouvrer une erreur dans le système. Ils peuvent être classés de la manière suivante :

- La *détection des erreurs* : ces mécanismes permettent de détecter la présence d'erreur dans le système. Il en existe de deux types : la détection concurrente lorsqu'elle a lieu pendant le fonctionnement du système, et la détection préventive qui prend place dans des phases particulières du fonctionnement du système (démarrage, phase de maintenance). Nous nous intéressons à l'adaptation en cours d'exécution. Seule la détection concurrente qui est réalisée pendant le fonctionnement normal du service nous intéresse ici. Les mécanismes de détection concurrente sont par exemple la détection du *crash* par *heartbeat*, ou la détection temporelle par *watchdog* ;
- Le *recouvrement* : les mécanismes de recouvrement transforment l'état du système qui contient une ou plusieurs erreurs ou fautes dans un état exempt d'erreur ou de fautes susceptibles d'être réactivées. Ces mécanismes permettent :
  - Le *traitement de l'erreur* : il contribue à l'élimination des erreurs de l'état du système. Trois types de recouvrement d'erreur existent : le *rollback* qui consiste à placer le système dans un état précédent l'activation de la faute, le *rollforward* qui consiste à placer le système dans un état sûr prédéfini ou reconstruit et la compensation qui consiste à utiliser la redondance de l'état inhérente à la stratégie pour masquer l'occurrence de l'erreur à l'utilisateur du système.
  - Le *traitement de la faute* : il vise à éviter la réactivation des fautes du système. Ces mécanismes permettent le diagnostic de l'erreur pour en déterminer la cause, l'isolation pour empêcher la faute d'être réactivée, la reconfiguration pour réassigner les tâches sur des composants sains, et la réinitialisation pour mettre à jour les tables du système afin de prendre en compte les modifications réalisées par l'isolation et la reconfiguration. Il est à noter que la reconfiguration en tant que traitement de la faute diffère de la reconfiguration en termes d'adaptation en deux points :
    - D'un point de vue décisionnel, la reconfiguration est réalisée suite à la détection d'une erreur dans le cas du traitement de la faute alors que dans le cas de l'adaptation, la raison est un changement de contexte opérationnel (ressources, besoins, modèle de faute) qui peut alors impacter les mécanismes de traitements de la faute.
    - Ensuite, la reconfiguration s'applique dans un cas au fonctionnel (traitement de la faute) et dans l'autre, au système tolérant aux fautes. Toutefois, il est important de constater que les raisons qui nous ont conduis à introduire l'adaptation de la tolérance aux fautes (et donc sa reconfiguration) est l'amélioration de la sûreté de fonctionnement du système car elle permet de tenir compte d'un nouveau type de faute lié à l'évolution contextuelle du système.

À partir de ces mécanismes de tolérance aux fautes, il est possible de réaliser deux familles de stratégie, selon que la détection a lieu avant ou après le traitement de l'erreur :

- Détection et recouvrement (*Detection and Recovery*) : la détection d'une erreur provoque le recouvrement de celle-ci. On discerne :
  - Le recouvrement arrière (*Backward Recovery*) reposant sur le traitement d'erreur par recouvrement arrière ;
  - Le recouvrement avant complet (*Full Forward Recovery*) qui repose sur la compensation des erreurs

- Le recouvrement avant partiel (*Partial Forward Recovery*) reposant sur le mécanisme de *rollforward* (état reconstruit)

– Masquage et recouvrement (*Masking and Recovery*) : le système compense l'erreur. La détection provoque le traitement de la faute.

On remarque que l'agencement de ces mécanismes de tolérance aux fautes varie peu d'un type à l'autre et qu'une entité orchestre les mécanismes de tolérance aux fautes.

Dans une stratégie reposant sur la réplication de type détection puis recouvrement, avant recouvrement, une élection peut être nécessaire. L'élection permet de sélectionner un membre au sein d'un groupe de répliques pour déterminer la réplique principale après observation d'une défaillance par exemple.

Ainsi, nous distinguons trois types de cœur de stratégie, qui peuvent être étendus aux besoins :

– Détection, traitement de l'erreur, traitement de la faute (D-E-F) utilisée pour réaliser une stratégie de type détection et recouvrement ne reposant pas sur de la réplication.

– Détection, élection, traitement de l'erreur, traitement de la faute (D-EL-E-F) utilisée pour réaliser une stratégie de type détection et recouvrement reposant sur la réplication du composant fonctionnel, comme par exemple la réplication passive (*Primary Backup Replication*) ou la réplication semi-active (*Leader Follower Replication*).

– Détection, traitement de la faute (D-F) utilisée dans des stratégies de type masquage et recouvrement comme par exemple dans la réplication active (*Triple Modular Redundancy*)

L'utilisation du mécanisme de recouvrement arrière suppose l'existence d'un état sain préalablement sauvegardé. Nous introduisons deux services qui contribuent à la sauvegarde et la restitution de l'état d'une réplique : la journalisation qui permet de sauvegarder les appels passés aux interfaces des composants et les réponses fournies lors des appels sur réceptacle, et la sauvegarde de point de reprise.

Notre décomposition fait donc apparaître les interfaces représentant les services fournis par ces différents mécanismes de tolérance aux fautes. Ces interfaces sont répertoriées dans le tableau 10.

Tableau 10 **Interfaces issues de la décomposition de la tolérance aux fautes**

Interfaces	*Description*
IErrorNotify	Notification d'une erreur après détection
IErrorHandlings	Traitement d'une erreur
IDiagnose	Diagnostic d'une erreur afin de trouver la faute qui en est l'origine
IIsolate	Isolation d'un composant fonctionnel défaillant
IReconfiguration	Assignation de tâches aux composants fonctionnels non-défaillants
IReinitialisation	Mise à jour des données
ICheckpointing	Sauvegarde d'un point de reprise
ILogging	Sauvegarde de données de log

IV.3 OUTILS D'INGENIERIE

IV.3.1 Modification du mécanisme d'exposition sous OpenCOM

OpenCOM fournit un mécanisme d'exposition d'interfaces et de réceptacles pour le composant composite *ComponentFramework*. Du point de vue des interfaces, ce mécanisme consiste à copier la référence de l'interface d'un composant et de la déclarer comme interface appartenant au composant composite. Ainsi, lorsqu'un composant est connecté à l'interface exposée, il est en réalité connecté à un *proxy* implicitement connecté au composant interne dont l'interface a été exposée (Figure 46-a). Le problème est que cette interface n'apparaît pas explicitement dans la représentation architecturale. Il est donc difficile de généraliser et d'automatiser la mise en place du modèle de l'exécution à partir de ce mécanisme d'exposition. Il en va de même pour l'exposition des réceptacles.

Pour mettre en place notre modèle de l'exécution, il est donc nécessaire de modifier ce mécanisme d'exposition. Nous proposons de modifier le mécanisme d'exposition et d'introduire les interfaces/réceptacles internes, duales des interfaces exposées, à la manière de Fractal (Figure 46-b). Ainsi, lors de l'exposition d'une interface, le composant composite arbore une interface externe et un réceptacle interne. Ainsi, il est possible de connecter un composant à l'interface externe (l'interface exposée) et de connecter le composant composite à un composant interne à partir du réceptacle interne correspondant à l'interface exposée.

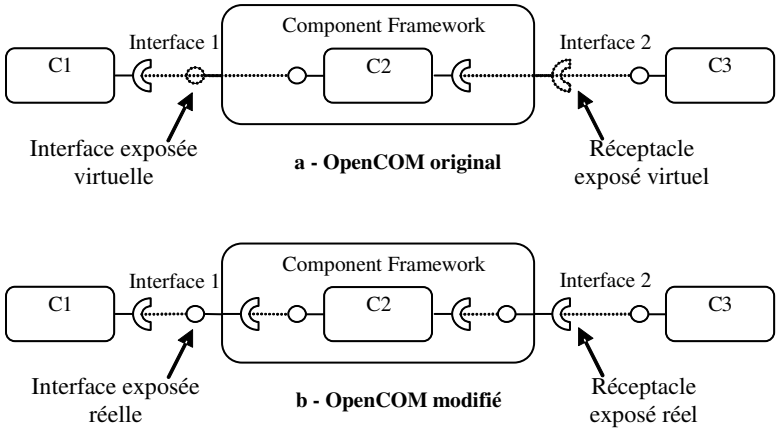


Figure 46 Mécanisme d'exposition sous OpenCOM

IV.3.2 Exécution et langage de description

Dans le but de construire le modèle de l'exécution, nous proposons d'introduire un langage dédié. Ce langage est permet de décrire les chemins d'exécution des tâches et des méthodes des différentes interfaces.

Ce langage est encore au stade de la conception. Il ne prend pas encore en compte toutes les subtilités que nécessite le modèle. Entre autres, nous ne prenons pas en compte aujourd'hui les connexions multiples vers les interfaces de communication et les connexions multiples à partir d'un composant.

Nous montrerons comment la description du composant a été étendue pour contenir la description de la dynamique du composant. La compilation de ce langage permet de

construire les réseaux de Petri qui correspondent aux chemins d'exécution partiels de chaque composant du logiciel. Il est alors possible de combiner ces réseaux de Petri pour former le réseau de Petri de l'architecture.

Nous allons présenter ce langage qui est assez intuitif en détaillant les opérandes et les opérateurs permettant de décrire les chemins d'exécution.

IV.3.2.1 Type d'interactions

Appel de procédure

Du fait de l'utilisation de composant composite, il est nécessaire de différencier les appels à partir d'un réceptacle normal, les appels à partir d'un réceptacle exposé et les appels à partir d'un réceptacle dual d'une interface exposée.

Nous introduisons ces trois types d'appel de procédure au travers des mots clés :

- call(*interface*, *méthode*) : appels à partir d'un réceptacle normal vers un composant externe ;
- call_exp_out(*interface*, *méthode*) : appels à partir d'un réceptacle exposé vers un composant externe ;
- call_exp_in(*interface*, *méthode*) : appels à partir d'un réceptacle interne dual d'une interface exposée (cf. IV.3.1) vers un composant interne.

Les paramètres *interface* et *méthode* correspondent à la méthode appelée sur l'interface qui est connectée au réceptacle. Dans le cas de l'exposition, la valeur des paramètres *interface* et *méthode* peut être remplacées par la valeur « * ». Cette valeur signifie qu'un appel peut potentiellement être passé sur n'importe quelle méthode de n'importe quel composant connecté à partir d'un réceptacle. Toutefois, nous assumons que lorsqu'un appel est passé à une interface et un réceptacle exposés, l'appel sortant est cohérent avec l'appel entrant et prend donc les mêmes valeurs que ce dernier. Cela évite l'explosion combinatoire du modèle, qui nécessiterait alors de considérer tous les appels possibles, à partir de tous les réceptacles internes. Ce point est valable aussi pour les réceptacles exposés.

De plus, nous avons introduit le mot clé *return* qui permet de forcer le retour d'un appel dans le chemin d'exécution.

Évènements

Nous considérons que les évènements sont traités soit dans le monde à composant (donc observables par interception au niveau des connexions) soit dans des couches inférieures du système. Dans ce deuxième cas, le composant doit être *ouvert* afin de rendre observables ces évènements. Ce deuxième cas est toujours vrai dans le cas des mécanismes de synchronisation qui font partie intégrante du code des composants.

Nous introduisons deux types d'évènement au travers des mots clés :

- event_ant(*type*, *id*) : évènement d'antériorité identifiée par le paramètre *id* ;
- event_synch(*type*, *id*): évènement de synchronisation correspondant au mécanisme de synchronisation identifié par *id*.

Le paramètre *type* peut prendre deux valeurs qui sont *production* et *consumption*. La production correspond à l'envoi d'un message ou évènement pour l'antériorité et au relâchement d'un sémaphore ou mutex pour les mécanismes de synchronisation. Inversement, la consommation correspond à la réception d'un message ou d'un évènement pour

l'antériorité et à l'acquisition d'un sémaphore ou mutex pour les mécanismes de synchronisation.

Le paramètre *id* peut correspondre à un paramètre interne du système. Lorsque c'est le cas, sa valeur commence par le caractère *$* suivi du nom du paramètre.

Dans ce langage, nous ne prenons pas en compte l'utilisation de mécanismes de synchronisation où plus d'un jeton est consommé/produit.

IV.3.2.2 Structures de contrôle

Il existe trois structures de contrôle dans notre modèle de l'exécution :

- La séquence représentée par le mot clé *AND* : *path1 AND path2*
- L'alternative représentée par le mot clé OR : *path1 OR path2*
- La boucle représentée par le mot clé *FOR* : *FOR(path)*

Dans notre langage OR prime sur AND qui prime sur FOR. Les parenthèses peuvent être utilisées pour surcharger ces priorités.

IV.3.2.3 Extension de la description d'un composant

La description d'un composant introduit dans III.4.4.1 permet de connaître la structure du composant. Nous proposons d'étendre cette description pour introduire la dynamique du composant (cf. Tableau 11). Nous introduisons alors des nœuds enfants au nœud *definition* nommés *calls* et *tasks* qui contiennent respectivement la liste des chemins des différents appels de méthode (*call*) et la liste des chemins d'exécution des tâches du composant (*task*).

Tableau 11 **Extension de la description d'un composant**

Noeuds	*Nœuds parents*	*Attributs*	*Description*
calls	definition	∅	Déclarations des chemins d'exécution des appels sur les interfaces d'un composant
call	calls	type	Type de l'appel qui prend trois valeurs *regular* pour les appels vers une interface standard, *exp_i* pour les appels vers une interface exposée, *exp_r* pour les appels vers une interface interne duale d'un réceptacle exposé.
		iid	Type de l'interface
		method	Nom de la méthode appelée
		path	Chemin d'exécution correspondant à cet appel
tasks	definition	∅	Déclarations des chemins d'exécution des tâches d'un composant
task	tasks	id	Identifiant de la tâche
		path	Chemin d'exécution de la tâche

Les attributs *path* contiennent la description du chemin d'exécution en utilisant le langage introduit ici.

IV.3.2.4 Exemple complet

Nous proposons dans cette section de décrire le comportement d'un composant introduit précédemment dans la section III.3.3.3. Ce composant est un composant de communication.

Ce composant contient trois interfaces dont une de type réseau et un réceptacle. Le composant contient une tâche interne qui attend la réception d'un message sur l'interface réseau, puis soit signale cette réception en appelant le composant connecté à partir du réceptacle *IListener* soit relâche un sémaphore. Deux méthodes existent dans ce composant. La méthode *send* permet

d'envoyer un message vers le composant connecté à l'interface réseau, et la méthode *receive* bloque jusqu'à la réception d'un message par la tâche 1.

La figure 47 représente les chemins d'exécution de ce composant. Les réseaux de Petri partiels correspondants à ce composant sont ceux de la figure 29.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definition clsid="components.communication.CCommunication" composite="false"
super="opencom.MetaComponent">
   <interfaces>
      <interface iid="tcp://localhost:3003"/>
      <interface iid="components.communication.api.ISend"/>
      <interface iid="components.communication.api.IReceive"/>
   </interfaces>
   <receptacles>
      <receptacle iid="components.communication.api.IListener" type="single"/>
   </receptacles>
   <parameters/>
   <tasks>
      <task id="1" path="event_ant(consumption, tcp://localhost:3003) AND
(event_ant(production,sem_1) OR call(components.communication.api.IListener, received))" />
   </tasks>
   <calls>
      <call type="regular" iid="components.communication.api.IReceive" method="receive"
path="return OR event_ant(consumption, sem_1)" />
      <call type="regular" iid="components.communication.api.ISend" method="send" path="
event_ant(production,tcp://localhost:3003)" />
   </calls>
</definition>
```

Figure 47 Exemple de description de la dynamique d'un composant

IV.3.3 Génération de la description des composants à partir d'annotations

Java 1.5 introduit la notion d'annotation qui permet de réaliser des traitements supplémentaires lors de la compilation. Ces traitements sont réalisés grâce à un outil de compilation comme l'outil *Annotation Processing Tool*[9] de *Sun Microsystems*. Dans Fractal les annotations ont été utilisées afin d'automatiser l'écriture des documents décrivant les composants à partir de Fraclet [71]. Nous proposons de nous inspirer de cet engouement pour les annotations afin d'automatiser l'écriture des descriptions des composants.

La génération de la description n'étant pas le cœur des travaux de thèse, les annotations introduites sont relativement sommaires, mais permettent tout de même de générer les documents de description. Nous introduisons donc les annotations détaillées dans le tableau 12.

Tableau 12 **Annotations pour la génération des documents de description des composants**

Annotation	*Localisation*	*Description*
@OpenCOMComponent	Class	Déclaration d'un type de composant
@OpenCOMInterface	Interface	Déclaration d'une interface
@OpenCOMReceptacle	Field	Déclaration d'un réceptacle
@OpenCOMParameter	Field	Déclaration d'un paramètre
@OpenCOMCall	Method	Déclaration d'un chemin d'exécution pour une méthode
@OpenCOMTask	Class	Déclaration d'un chemin d'exécution pour une tâche

Ces annotations possèdent des paramètres détaillés dans le tableau 13 permettant de décrire les différentes annotations.

[9] http://java.sun.com/j2se/1.5.0/docs/guide/apt/

Tableau 13 **Paramètres des annotations**

Annotation	*Paramètres*	*Description*
@OpenCOMComponent	@OpenCOMInterface[] interfaces	Interfaces explicites du composant
	@OpenCOMReceptacle[] receptacles	Réceptacles explicites du composant
	@OpenCOMParameter [] parameters	Paramètres explicites du composant
	@OpenCOMTask [] tasks	Les tâches du composants
	@OpenCOMCall[] calls	Les chemins d'exécution explicites du composant
@OpenCOMInterface	String iid	Le nom de l'interface
@OpenCOMReceptacle	String iid	Le nom du réceptacle
	String type	Le type du réceptacle
@OpenCOMParameter	String type	Le type du paramètre
	String name	Le nom du paramètre
@OpenCOMCall	String method	Le nom de la méthode
	String iid	Le nom de l'interface correspondante
	String type	Le type de l'appel parmi *regular*, *exp_i* et *exp_r*
	String path	La description du chemin d'exécution de la méthode
@OpenCOMTask	int id	L'identifiant de la tâche
	String path	La description du chemin d'exécution de la tâche

IV.3.4 Outils graphique pour construire les descriptions des configurations

L'écriture de la description du document représentant la configuration d'un logiciel est une opération longue, fastidieuse et source de nombreuses erreurs. Pour faciliter son écriture, et éviter les erreurs, nous introduisons un outil permettant de générer le document XML décrivant une configuration. Cet outil est appelé « OpenCOM ADL Factory ».

Cet outil repose sur l'utilisation d'un répertoire dans lequel toutes les données de configurations sont stockées. Ce répertoire est organisé de la manière suivante :

- composants : contient les documents de description des composants connus et utilisés ;
- architectures : contient les documents de description de toutes les architectures à composants réalisées. Ces documents représentent les configurations disponibles pendant l'adaptation ;
- paramètres : les valeurs des paramètres à fournir lors du déploiement d'une configuration.

Ce répertoire est ensuite directement utilisable par notre plateforme. Notre plateforme consulte alors les configurations disponibles dans ce répertoire et récupère leurs descriptions pour les déployer. De plus, le modèle de l'exécution récupère la description des composants et construit alors dynamiquement les chemins d'exécution du logiciel.

L'interface graphique de l'outil (Figure 48) est constituée de trois panneaux principaux :

- Navigateur : il permet de navigation des types de composant disponibles et les configurations existantes.
- Visualisation : il permet de visualiser le contenu d'une architecture à composant et fait apparaître les composants de cette architecture et les connexions reliant les composants.
- Erreurs et conflits : il permet de visualiser les dépendances non satisfaites (i.e. réceptacles non-connectés) afin d'aider l'architecte logiciel à construire son application.

Pour insérer un composant dans l'architecture, il suffit de faire un glisser-déplacer du type correspondant au composant à créer, vers le panneau de visualisation représentant l'endroit où le composant doit être inséré. Une fenêtre apparaît alors et demande de spécifier un nom pour

le composant. De plus, cette fenêtre possède un onglet qui permet de fixer la valeur d'un paramètre. Ce paramètre est alors compilé et sérialisé dans un fichier dans le répertoire *paramètres*. Le nom du fichier contenant le paramètre est alors ajouté à la description de la configuration.

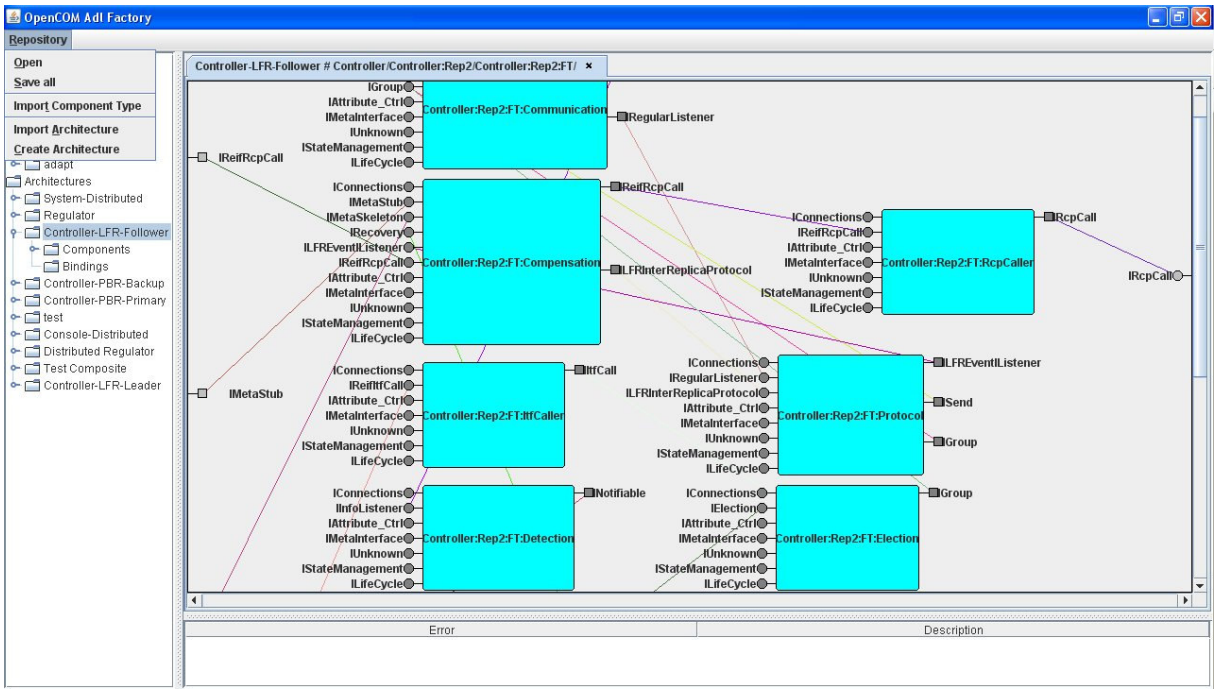


Figure 48 Interface de l'outils OpenCOM ADL Factory

À partir de ces descriptions, il est possible de générer les chemins d'exécution sous la forme d'un réseau de Petri correspondant au modèle. Ainsi, nous pouvons spécifier des adaptations logicielles en précisant la configuration avant adaptation et la configuration après adaptation, pour déterminer hors ligne le graphe des états d'adaptation et la position des verrous pour le contrôle de l'exécution. Ces informations sont ensuite sauvegardées pour être utilisées en ligne afin de réaliser l'adaptation.

IV.4 MISE EN ŒUVRE DE L'APPROCHE

Pour illustrer le travail effectué dans ces travaux de thèse, nous proposons d'appliquer les principes d'adaptation à un système de contrôle commande automatique. Dans cette section, nous détaillons le système utilisé. La section suivante traitera de l'adaptation de ce système en ligne.

Ce système est composé d'une partie physique et d'un contrôleur en charge de la régulation du système automatique. Ce contrôleur est critique dans le sens où sa défaillance provoque la perte du service, et une possible détérioration du matériel. Il est donc nécessaire de mettre en place des mécanismes de tolérance aux fautes.

Dans notre cas d'étude, n'étant pas munis du système physique, ce dernier est simulé par du logiciel. Nous pouvons donc évacuer les aspects temps réel dans une première approche de l'adaptation.

IV.4.1 Système physique

Le système physique à contrôler est constitué d'un chariot motorisé possédant un axe permettant la rotation d'un pendule (figure 49). Le chariot dont le mouvement est piloté par un moteur électrique est fixé sur un rail. Le chariot possède d'une masse *M*. Le pendule possède une tige de longueur *l* et de masse considérée négligeable par rapport au disque situé à son extrémité. Ce disque possède une masse *m*.

Ce système possède deux capteurs permettant respectivement de connaître l'inclinaison du pendule, noté θ et la position du chariot *x*. De plus, le moteur peut être commandé par une valeur *U* représentant son couple instantané (proportionnel au courant d'alimentation).

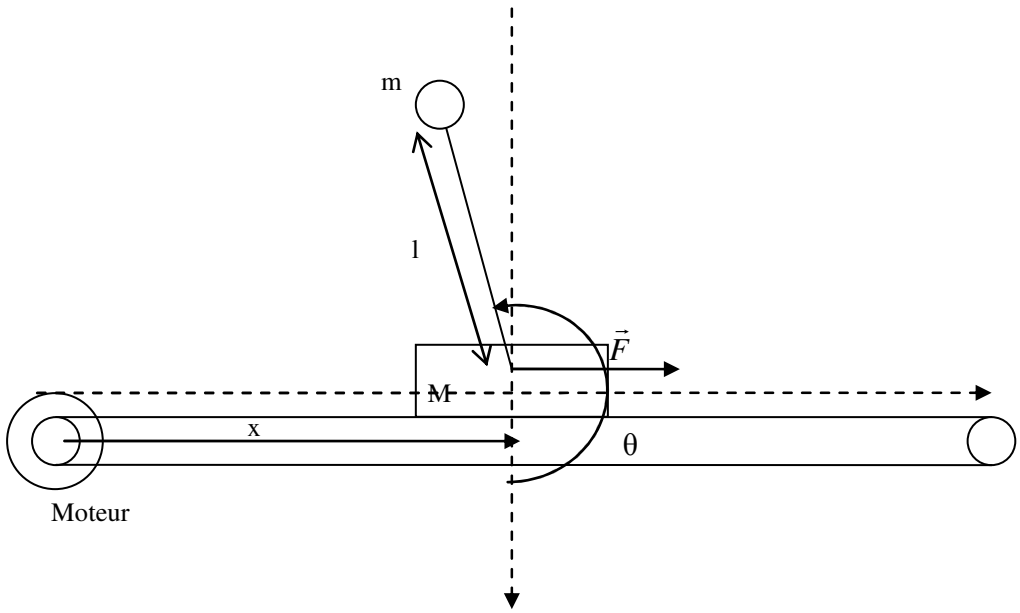


Figure 49 Pendule inversé

Les équations différentielles du mouvement obtenues à partir de l'expression du Lagrangien, s'écrivent :

$$(M+m)\ddot{x} + ml\cos(\theta)\ddot{\theta}^2 = GU - f_c\dot{x}$$

$$\cos(\theta)\ddot{x} + l\ddot{\theta} - g\sin(\theta) = 0$$

Avec :

- M, la masse du chariot (M=2,4kg),
- m, la masse du pendule (m=0.23kg),
- l, la longueur du pendule (l=0.36m),
- G, le facteur de conversion de la commande (G=20N/V),
- f_c, le coefficient de frottement visqueux du chariot sur les rails (f_c=11N.s/m),
- g, le coefficient de gravité (g=9.81m.s^{-2}),
- U, la grandeur de la commande du moteur bornée entre ±0.8.

Certains frottements et autres phénomènes physiques ont été négligés dans cette étude.

Ce système physique est couplé avec un ordinateur connecté à un réseau au travers duquel il peut communiquer avec les autres nœuds présents (cf. figure 50).

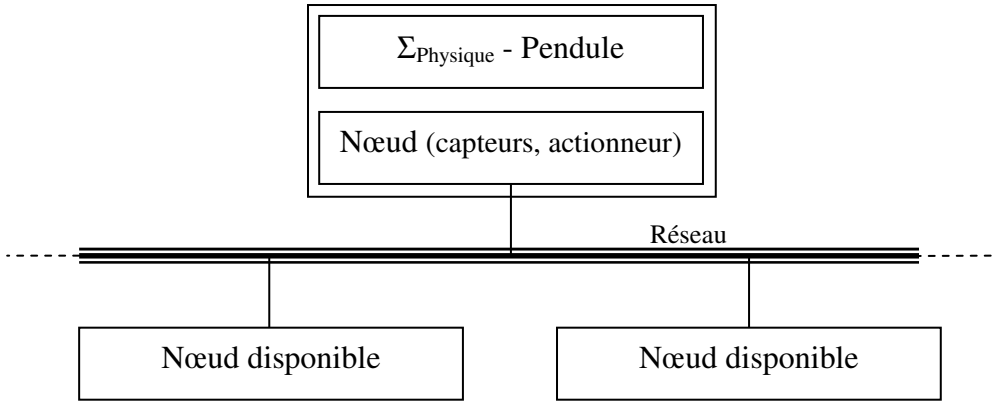


Figure 50 Nœuds et réseau

IV.4.2 Contrôle

Le but du contrôle est déplacer le chariot à un point spécifié par un utilisateur noté *xref* tout en maintenant le pendule en position d'équilibre. Le pendule ne doit pas tomber pendant le déplacement du chariot. Les équations du système peuvent être linéarisées autour du point d'équilibre $\theta = \pi + \alpha$ en utilisant le développement limité du sinus et cosinus. Les équations linéarisées du système deviennent alors :

$$\ddot{x} = -\frac{m}{M} g\alpha - \frac{f_c}{M}\dot{x} + \frac{G}{M}U$$

$$\ddot{\alpha} = -\frac{m+M}{Ml} g\alpha - \frac{f_c}{Ml}\dot{x} + \frac{G}{Ml}U$$

Sur ce dispositif, on désire contrôler deux variables d'états (x et α) avec une seule variable d'entrée (U). Pour cela, on peut utiliser plusieurs structures de contrôle, notamment, un double PID, ou un retour d'état avec action intégrale. Nous avons utilisé la deuxième solution (cf. figure 51). Les équations du système peuvent alors s'exprimer :

$$\dot{X} = AX + BU$$
$$Y = CX + DU$$

Avec :

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{m}{M} g & -\frac{f_c}{M} & 0 \\ 0 & -\frac{M+m}{Ml} g & -\frac{f_c}{Ml} & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ 0 \\ \frac{G}{M} \\ \frac{G}{Ml} \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

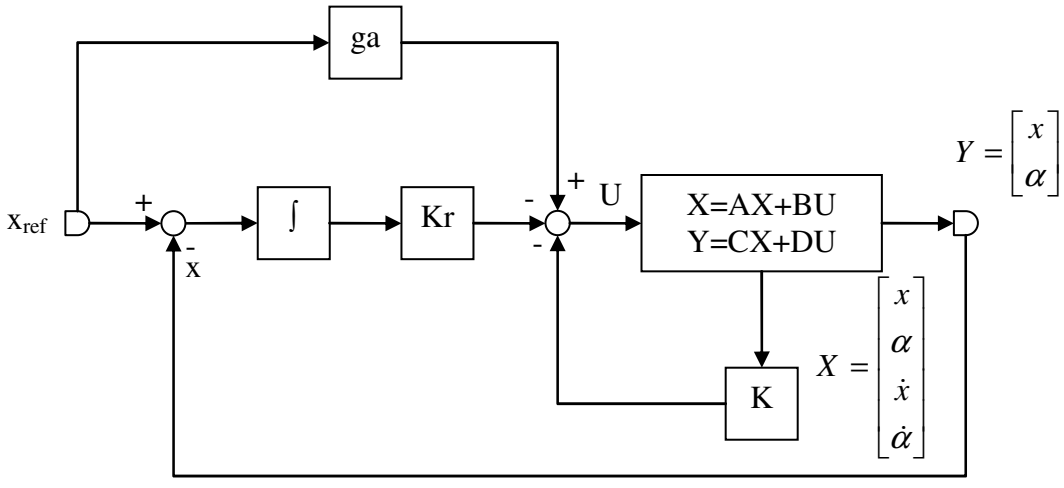


Figure 51 Retour d'état gain avec action intégrale

Le paramètre ga intervient au numérateur de la fonction de transfert équivalente en boucle fermée. Il peut être déterminé soit pour annuler la grandeur x_r en régime permanent soit pour

compenser un pôle en boucle fermée. Nous avons choisi la première solution. Le gain ga a alors pour valeur :

$$ga = -\frac{1}{C.\left(A - B.K^{t}\right)^{-1}.B}$$

Le placement de pôle choisi, consiste en un pôle d'ordre 5 en -4. L'échantillonnage du système est de 0.05 secondes. L'étude de la stabilité du système sort du cadre de ces travaux et ne sera détaillée ici.

IV.4.3 Implémentation

Le système physique est ici simulé. Le système de contrôle-commande est implémenté à l'aide de trois composants logiciel :

- le système fournissant les interfaces *ICapteur* et *IActionneur* permettant respectivement de lire les capteurs (*Y*) et commander le moteur (*U*) ;
- le contrôleur qui possède une interface *IReference* permettant de lui fournir la position de référence $x_{réf}$ vers laquelle le chariot doit être amené, et deux réceptacles connectés aux interfaces *ICapteur* et *IActionneur* du système afin de pouvoir calculer la commande du moteur à partir des mesures des capteurs ;
- une console qui est connectée au contrôleur, et qui à partir du clavier, lit la position de référence et la fournit au contrôleur.

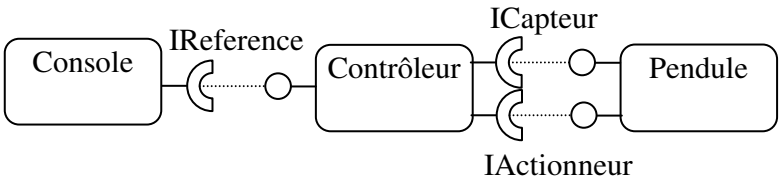


Figure 52 Conception du système de contrôle commande

La figure 53 détaille les interactions entre ces composants à l'aide d'un diagramme de séquence UML[10]. La console possède une tâche cyclique. Lorsqu'une saisie clavier est effectuée, la console vérifie qu'il s'agit d'une position valide et l'envoie au contrôleur. Le contrôleur possède une tâche périodique. Cette tâche débute par la lecture des capteurs. S'en suit le calcul de la commande du moteur, qui est ensuite fournie au système. La période de cette tâche est de 50ms et correspond à la fréquence d'échantillonnage du notre système physique. Le contrôleur possède une section critique (notée *region* en UML) : le calcul de U et la modification de la position de référence x_{ref} ne peuvent être faits simultanément.

[10] Unified Modelling Language (http://www.uml.org/)

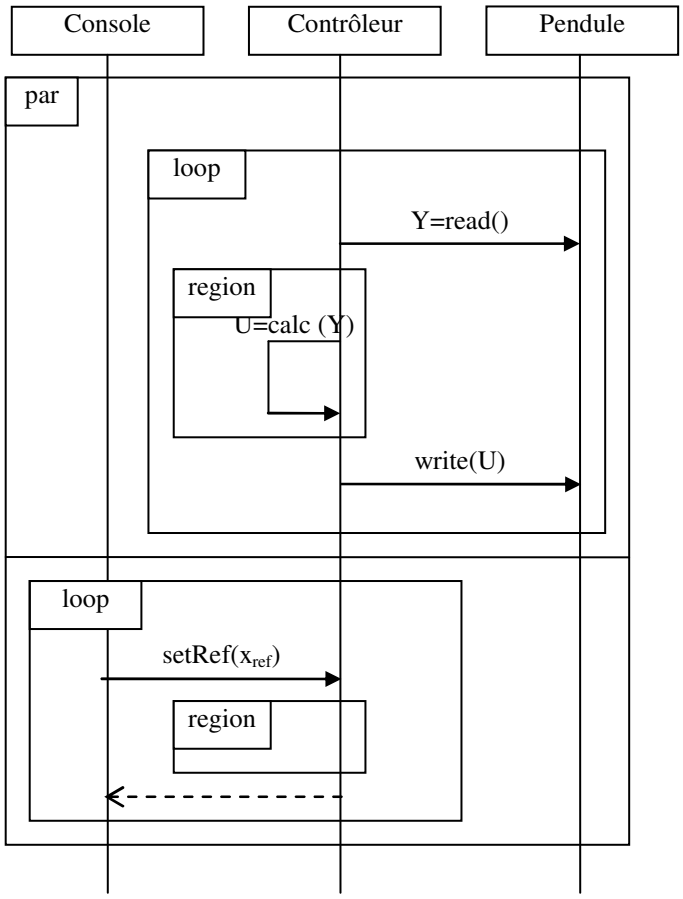


Figure 53 Tâches du système

IV.4.4 Tolérance aux fautes du contrôleur

Le crash du contrôleur provoque la chute du pendule qui est une défaillance critique du système, car elle correspond à l'échec du service.

Nous proposons de mettre en place une stratégie de tolérance aux fautes permettant de tolérer le crash du contrôleur, à l'aide de la réplication. Ainsi, deux stratégies peuvent être utilisées : le Leader Follower Replication (LFR) et le Primary Backup Replication (PBR). Leur fonctionnement sera détaillé dans la suite de cette section. Ces deux stratégies fournissent des caractéristiques différentes tant en termes d'utilisation des ressources du système que concernant les temps de recouvrement, et de réparation. Nous proposons de comparer ces stratégies dans une sous-section pour montrer ce que peut apporter l'adaptation de ces stratégies de tolérance aux fautes.

Pour insérer ces stratégies au méta-niveau, chaque composant est wrappé par un *ApplicationController*. Ainsi, les composants permettant la mise en place de ces stratégies sont introduits au méta niveau par connexion au modèle. De plus, ces deux stratégies reposent sur la réplication et la distribution des composants fonctionnels. Les techniques utilisées pour réaliser cette distribution vont maintenant être détaillées.

IV.4.4.1 Distribution

Afin de tolérer le crash matériel, il est nécessaire d'utiliser plusieurs nœuds matériels indépendants, et de distribuer l'application fonctionnelle sur chacun de ces nœuds. Ainsi, nous utilisons des stubs et des *skeletons* pour réaliser des appels de procédures distantes. Ces

stubs et *skeletons* utilisent des services de communication pour s'échanger requêtes et réponses. Les requêtes et les réponses contiennent le nom de l'interface, le nom de la méthode et les arguments. Ces informations sont sérialisées en utilisant les mécanismes de sérialisation intrinsèques à Java.

Les composants fournissant les services de communication reposent sur l'utilisation d'une librairie de communication de groupe appelée *Spread*. Cette communication de groupe permet aux clients d'envoyer les requêtes à toutes les répliques, mais aussi aux répliques de communiquer entre elles.

Ainsi, l'application fonctionnelle distribuée est illustrée à la figure 54.

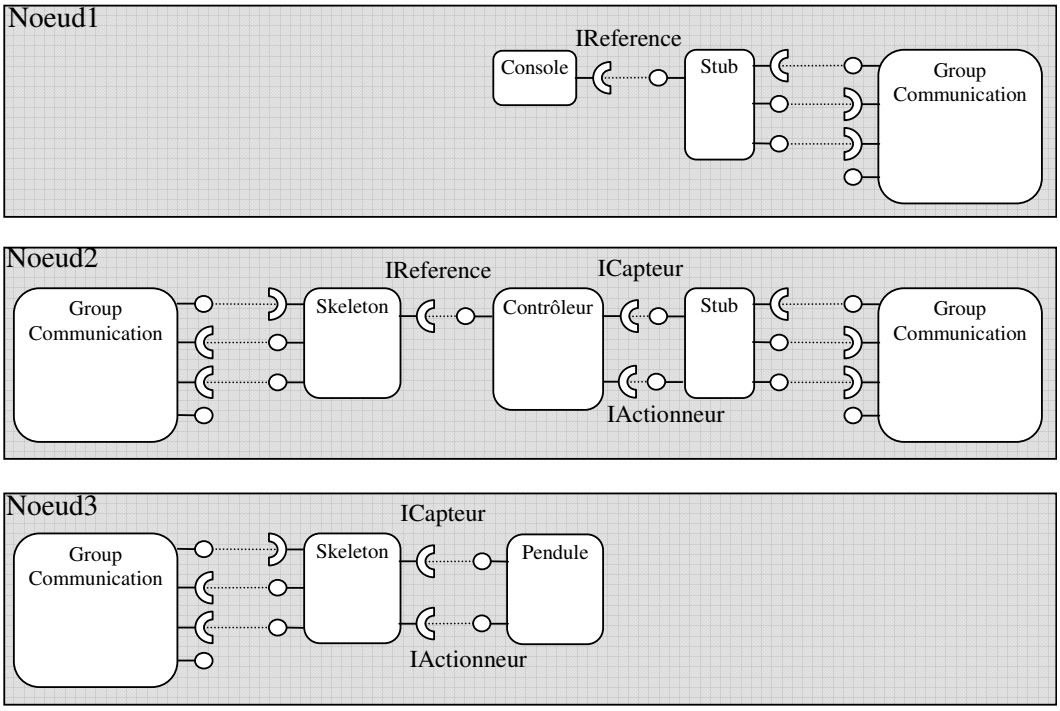


Figure 54 Architecture fonctionnelle distribuée

La réplication nécessite d'intercepter les messages reçus et envoyés par les *stubs* et *skeletons*. Nous proposons d'utiliser le wrapper utilisé pour le composant fonctionnel afin d'obtenir ces informations.

IV.4.4.2 Primary Backup Replication

La stratégie *Primary Backup Replication* repose sur une synchronisation des répliques par transfert d'état. Il existe deux types de réplique : la *primary* qui est l'unique réplique active et traite les requêtes et les *backups* qui sont des répliques passives et qui ne traitent aucune requête. Lorsqu'un crash de la réplique *primary* est détecté, une réplique *backup* est élue pour devenir la nouvelle réplique *primary*.

Il existe principalement deux façons de réaliser la synchronisation de l'état :

- L'état est stocké dans un support de stockage fiable. Sur détection du crash, l'état stocké dans le support de stockage est rechargé dans la nouvelle réplique *primary*, puis cette réplique est démarrée. La stratégie est qualifiée de *Cold Primary Backup Replication*.

– L'état est pré-chargé dans les répliques secondaires qui restent inactives. Sur détection du crash, la réplique est démarrée. La stratégie est qualifiée de *Warm Primary Backup Replication*.

De plus, la sauvegarde d'état peut être effectuée de deux manières :

– Périodiquement : l'état de la réplique est sauvegardé à des intervalles de temps réguliers

– Systématiquement : l'état de la réplique est sauvegardé dès qu'une requête est reçue, ou envoyée.

Nous proposons de mettre en place la stratégie *Cold Primary Backup Replication* avec une sauvegarde périodique de l'état de reprise de la réplique.

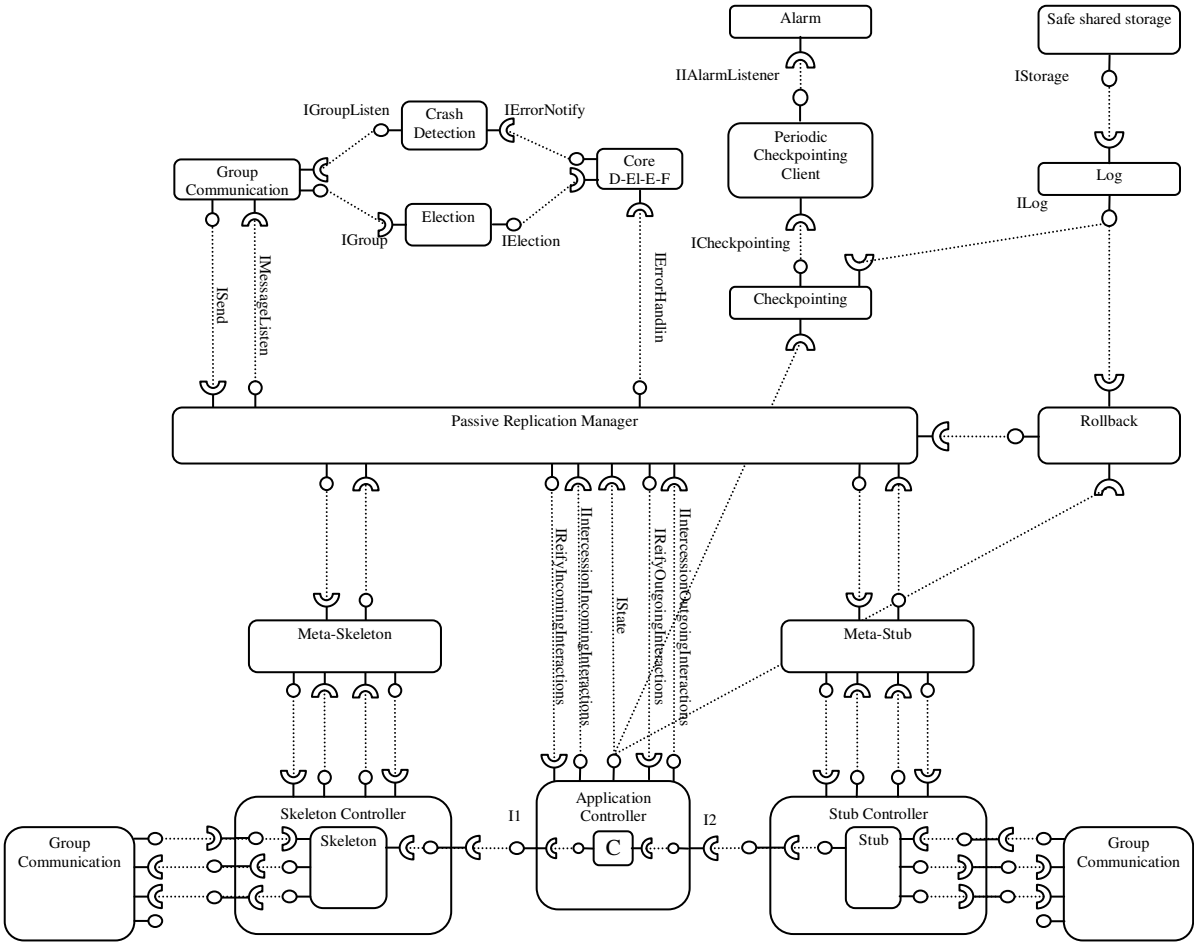


Figure 55 Architecture d'une réplique dans une *Cold Primary Backup Replication*

L'architecture à composant d'une réplique dans cette configuration est représentée à la figure 55. Nous disposons d'un support de stockage fiable accessible par toutes les répliques. L'accès à ce support est réalisé via le composant *safe shared storage*. Le protocole de réplication est réalisé par le composant *passive replication manager* qui assure que les requêtes des clients sont traitées exactement une fois par les répliques. Le mécanisme de détection repose sur l'hypothèse qu'une réplique sort du groupe des répliques si et seulement si elle défaille par crash. Le mécanisme d'élection utilise une vue globale totalement ordonnée des membres du groupe. Ainsi, le premier membre du groupe sera élu comme nouvelle réplique *primary*. La stratégie est ici de type détection, élection, traitement de l'erreur. Le composant *Core D-El-E-F* orchestre les services de détection et de recouvrement. Nous ne

faisons pas apparaître le traitement de la faute dans notre conception pour des raisons de simplification.

IV.4.4.3 Leader Follower Replication

La stratégie *Leader Follower Replication* est une stratégie qui repose sur le principe de la réplication semi-active. La synchronisation des répliques est assurée, sous hypothèse de déterminisme, par le traitement des requêtes dans le même ordre par chaque réplique. Dans cette stratégie, il existe deux types de répliques. Contrairement aux *followers*, la réplique *leader* envoie les réponses aux clients du composant fonctionnel, et les requêtes aux composants fonctionnels auxquels il est connecté. Sur détection d'un crash, une réplique *follower* est élue pour être la nouvelle réplique *leader*. Elle devient alors autorisée à envoyer les réponses aux clients et les requêtes aux autres composants fonctionnels.

La *Leader Follower Replication* possède des points communs avec la *Primary Backup Replication* présentée ci-dessus. En effet, seul le mécanisme de synchronisation des répliques est différent. La figure 56 illustre l'architecture d'une réplique dans cette configuration. Globalement, les deux stratégies fonctionnent de manière identique : les mécanismes de détection et d'élection sont les mêmes, seuls les mécanismes de synchronisation des répliques diffèrent.

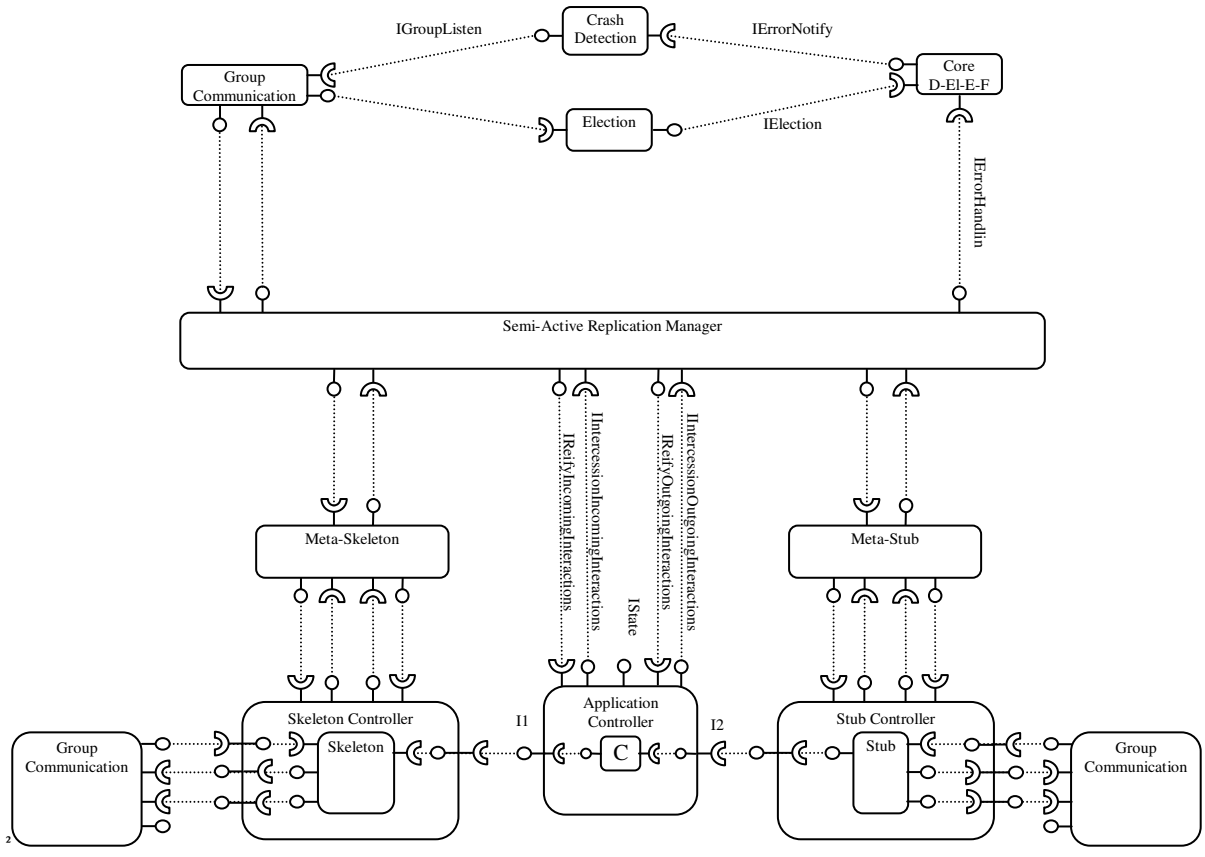


Figure 56 Architecture d'une réplique dans la configuration *Leader Follower Replication*

IV.4.4.4 Comparaison LFR vs PBR

Pour motiver l'adaptation nous proposons de comparer les des stratégies de tolérance aux fautes introduites précédemment. Notre comparaison est une comparaison qualitative des

deux stratégies. Nous nous intéressons à leurs performances lors du recouvrement qui montre la réactivité des stratégies lorsqu'une erreur est détectée, à la consommation de la bande passante réseau et enfin à l'utilisation du CPU.

Performance du recouvrement

Lorsqu'un crash est détecté, la stratégie LFR ne réalise à proprement parler aucun traitement particulier. Les messages en attente sont simplement transmis. La réplique est presque immédiatement opérationnelle.

Il n'en va pas de même pour la stratégie PBR avec reprise à froid. En plus de devoir transmettre les messages en attente, il est nécessaire de récupérer le dernier état de reprise sauvegardé sur le support de stockage puis de le charger dans le composant fonctionnel et enfin de démarrer celui-ci. Récupérer l'état de reprise peut éventuellement prendre du temps lorsque le support de stockage est distribué. De plus, charger et démarrer le composant nécessite un temps non-négligeable.

Dans cette stratégie, l'état de la réplique après recouvrement se trouve dans un état passé qui dépend de la date de sauvegarde du dernier point de reprise. La différence entre cette date et l'instant présent est une durée dépensée pendant laquelle le composant n'a pas réellement fourni son service. On peut donc ajouter cette durée dans notre calcul du temps de recouvrement.

Ainsi, dans notre implémentation, le temps nécessaire au recouvrement d'un crash est environ quatre fois plus important pour la stratégie PBR avec reprise à froid qu'avec la stratégie LFR. Ainsi, s'il existe des contextes opérationnels où la performance souhaitée lors du recouvrement varie, il peut être intéressant de modifier la stratégie courante pour l'adapter aux besoins.

Utilisation du réseau

Dans la stratégie PBR, la réplique principale doit transmettre l'état de reprise aux répliques secondaires. Lorsqu'aucun matériel spécifique n'est disponible, cette transmission a lieu via le réseau. La consommation de la bande passante lors du transfert de l'état de reprise est proportionnelle à la taille de l'état et à la fréquence de sauvegarde.

Dans la stratégie LFR comme dans la stratégie PBR, la réplique principale envoie des messages aux répliques secondaires afin de synchroniser les traitements réalisés (accusé de traitement pour un message en attentes par exemple). Toutefois, contrairement à la stratégie LFR, dans PBR, les répliques ne sont pas actives. Il n'est pas nécessaire de synchroniser les traitements réalisés sur les stubs. Ainsi, il y a moins de messages de synchronisation échangés sur PBR que sur LFR. La bande passante utilisée dépend alors de la fréquence de l'échange de messages mais aussi de la taille des messages échangés.

En conclusion, selon la taille et la fréquence des messages et des points de reprise, la meilleure stratégie varie. Pour un état du composant de petite taille et une fréquence de sauvegarde de l'état relativement faible, la stratégie PBR devient meilleure que la stratégie LFR en termes de consommation de la bande passante.

Utilisation du CPU

Dans la stratégie PBR, les répliques sont inactives. Ainsi, la consommation globale du temps CPU à nombre de répliques égal est nettement plus faible en PBR qu'en LFR. Toutefois, le traitement de l'état peut faire augmenter considérablement le temps CPU sur la réplique principale en PBR par rapport aux répliques en LFR.

IV.4.5 Adaptation coordonnée Application/Tolérance aux fautes

Nous proposons d'étudier un scénario d'évolution qui a été mis en place lors du mini-projet ASAP (*ASsessment based adAPtation*) dans le cadre du réseau d'excellence européen ReSIST (*Resilience for Survavility in IST*). Ce cas d'étude repose sur la mise à jour d'un composant logiciel critique dans une architecture à composant. Ce composant logiciel est muni de mécanismes de tolérance aux fautes dans le but de couvrir son modèle de faute. L'objectif est d'obtenir l'assurance à la fin de la mise à jour que le composant fonctionnel déployé est celui possédant les meilleures caractéristiques en termes de :

- Sûreté de fonctionnement : il est nécessaire d'introduire une phase d'évaluation de la nouvelle version pour connaître son comportement dans le système ;
- Utilisation des ressources : selon les caractéristiques intrinsèques du composants (états, tailles des messages), il est préférable d'utiliser PBR à LFR et inversement. Cela peut être déterminé hors ligne.

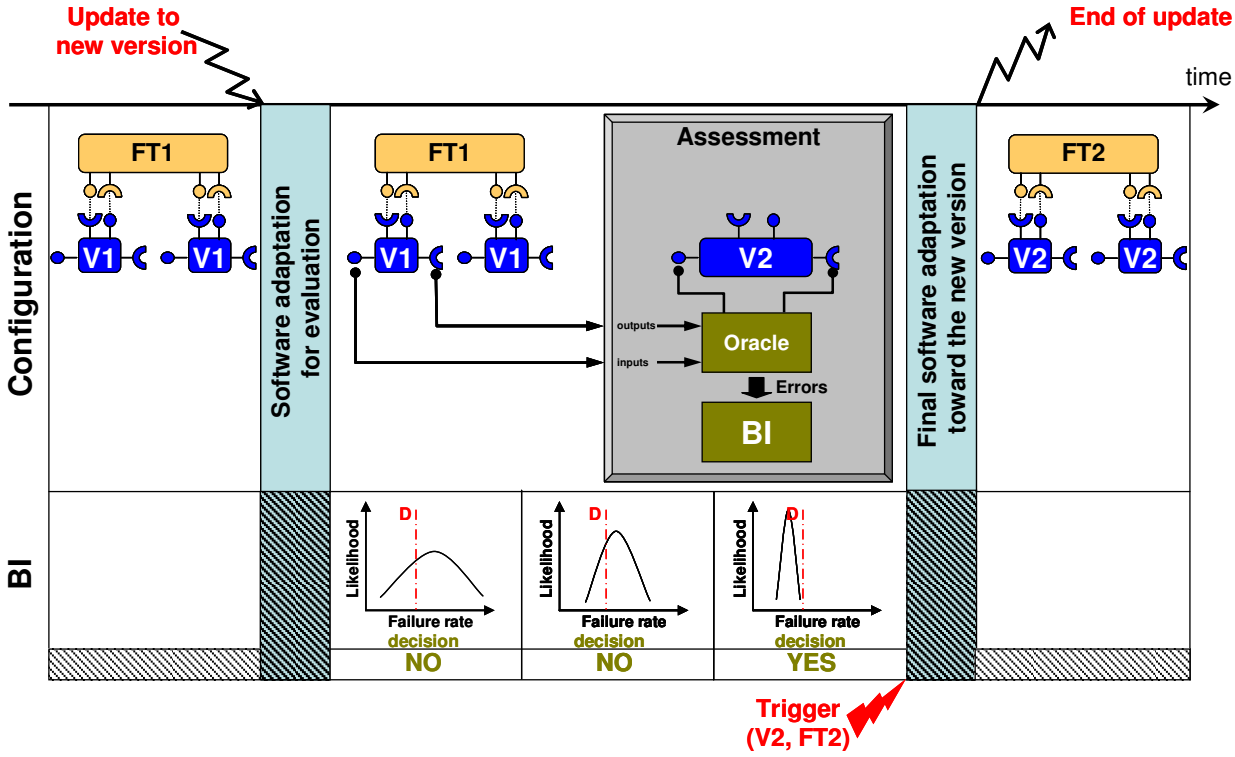


Figure 57 Scénario d'évolution

Ainsi (Figure 57), le scénario se décompose en deux étapes d'adaptation. Lorsqu'une nouvelle version est disponible (à gauche), la base de données des configurations et des règles d'adaptation est mise à jour et la présence d'une nouvelle version est signalée. Le système est alors modifié pour permettre l'évaluation de la nouvelle version. Une instance de la nouvelle version du composant est placée dans une zone de quarantaine dans laquelle sa probabilité de défaillance sur demande est évaluée. Cette probabilité de défaillance sur demande est considérée comme une variable aléatoire qu'il faut déterminer. Pour chaque interaction de la nouvelle version avec le reste du système, un oracle détermine si une erreur est présente et fournit ce booléen à un moteur d'inférence Bayésienne. L'inférence Bayésienne permet alors de déterminer la distribution de la variable aléatoire.

La décision de remplacer l'ancienne version par la nouvelle repose sur le résultat de l'inférence Bayésienne. Si l'inférence Bayésienne assure que la probabilité de défaillance sur demande est supérieure à un seuil avec une confiance suffisante ($P_f>10^{-3}$ avec une confiance de 99% par exemple) la nouvelle version est jugée trop défaillante et n'est pas déployée. Le système revient dans la configuration précédant l'apparition de cette nouvelle version. La mise à jour a alors échoué.

Dans le cas où la probabilité de défaillance sur demande est inférieure au seuil avec une certaine confidence ($P_f<10^{-3}$ avec une confiance de 99% par exemple), la nouvelle version est jugée suffisamment sûre pour remplacer la version précédente. Le système est alors modifié afin de prendre en compte d'une part cette nouvelle version, mais aussi les mécanismes de stratégie de tolérance aux fautes les plus adaptés à cette nouvelle version. La mise à jour est alors terminée avec succès.

Ce scénario a été appliqué à notre système de contrôle commande.

IV.4.5.1 Transfert de l'état pendant l'adaptation

Nous souhaitons préserver la continuité de service, ce qui signifie que le passé du système doit être transféré de l'ancienne configuration à la nouvelle. Nous proposons de transférer les rôles de la manière suivante :

- La réplique *leader* est transformée en réplique *primary*
- Les répliques *follower* sont transformées en répliques *backups*

Ces rôles sont déterminés par un paramètre dans les composants en charge de la réplication. Ce paramètre est alors transféré lors de l'adaptation entre LFR et PBR. De plus, ces composants possèdent des buffers contenant les messages en attente d'accusé de réception. Dans un état adaptable, la causalité est respectée. Ainsi, nous sommes assurés qu'il n'existe pas de messages en transit. Ainsi, ces buffers sont vides dans l'état d'adaptation. Il n'est pas nécessaire de les transférer. Les autres composants du logiciel de tolérance aux fautes nouvellement insérés peuvent être initialisés avec un état par défaut.

Il n'en va pas de même pour les composants en charge du contrôle du pendule. L'état d'un composant de contrôle est composé des variables d'états du système. Lors de l'adaptation, pour chaque instance du composant de l'ancienne version, cet état est transféré à l'instance de la nouvelle version qui le remplace.

IV.4.5.2 État adaptable

Nous supposons connus les modifications effectuées sur l'ensemble du système en termes de composants supprimés/créés/modifiés (au sens de l'état) et des connexions supprimées/crées. Nous pouvons maintenant nous poser la question de l'état adaptable du système. Pour cela, il est nécessaire d'étudier le modèle de l'exécution du système et de déterminer pour chaque tâche les ensembles $E_1/E_2/E_3$. Nous rappelons que :

- E_1 permet d'adapter une tâche en respectant la configuration après adaptation sur le cycle en cours ;
- E_2 permet d'adapter une tâche en respectant la configuration avant adaptation sur le cycle en cours ;
- E_3 est l'ensemble des états non-adaptables.

Le modèle de l'exécution du système muni d'une stratégie Leader Follower Replication générée à partir d'une connaissance locale de chaque composant est fourni à la figure 58. Il

est composé de onze tâches cycliques ce qui constitue un réseau de Petri possédant environ 600 places et transitions. Chaque réplique possède quatre tâches. La console possède deux tâches et le système physique simulé une seule. Bien que la figure 58 soit illisible, elle montre la complexité de l'exécution du logiciel distribué qui rend le service tolérant aux fautes dans notre cas d'étude. Nous allons maintenant nous focaliser sur une unique tâche du système afin de construire un graphe représentant ses états adaptables.

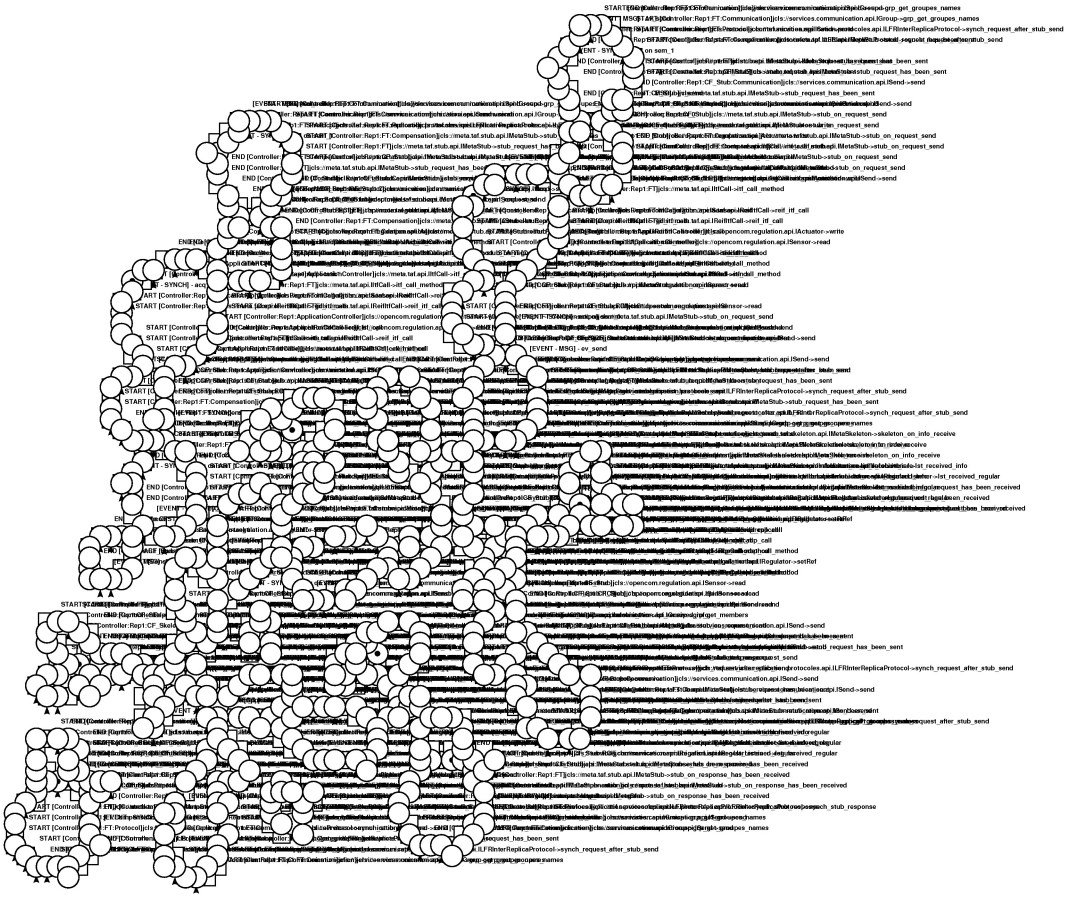


Figure 58 Modèle de l'exécution du système tolérant aux fautes

Nous proposons d'étudier la tâche du stub d'une réplique de la stratégie du LFR. Son chemin d'exécution est représenté à la figure 59. Cette tâche appartient au composant de communication connecté au stub. Après avoir démarré, deux comportements sont possibles. Le premier consiste à envoyer des informations au travers de l'interface *IInformationMessageListener*. Ces informations signalent par exemple l'entrée d'un membre dans un groupe de communication. Cet appel est réifié vers le composant de compensation, puis parcourt le stub. Le deuxième cas de figure concerne l'arrivée d'un message contenant des données. Ce message provoque un appel à partir du réceptacle *IRegularMessageListener* du composant de communication. Cet appel est réifié au composant en charge de la compensation. Ce composant réalise alors une synchronisation par message avec les autres répliques en utilisant un composant de communication de groupe. Cette synchronisation permet de faire suivre la réponse reçue par le stub aux autres répliques. Une fois cette synchronisation par message effectuée, le message est transmis au stub. Le stub stocke alors ce message dans une pile et signale l'arrivée d'un message aux tâches en attente. L'exécution se termine alors.

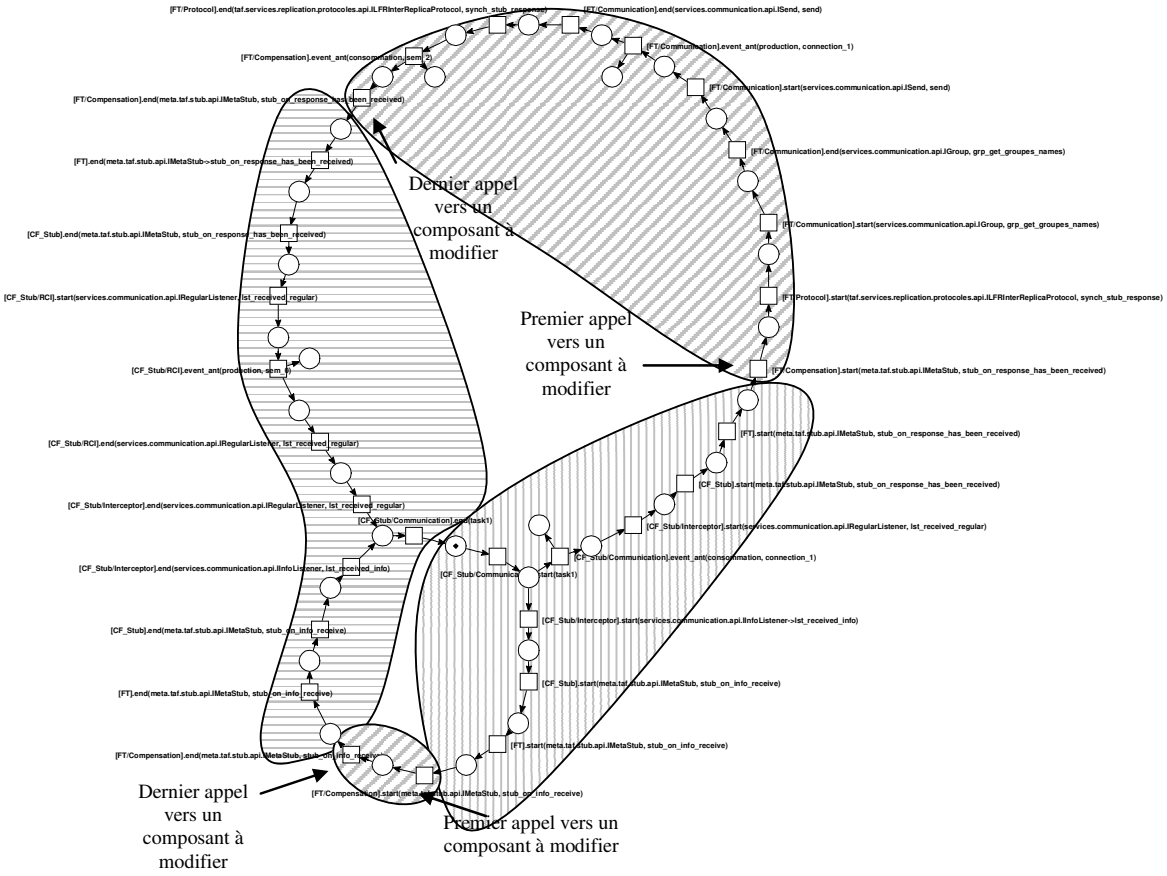


Figure 59 Modèle de la tâche du stub sur une réplique du contrôleur

Cette tâche est assez simple et possède une caractéristique intéressante : sa représentation est équivalente à ses chemins d'exécution. En d'autres termes, suivre une branche est équivalent à suivre un chemin d'exécution. Il est donc possible de déterminer le graphe des états adaptables de cette tâche directement à partir de son modèle, sans générer le graphe des chemins. Pour déterminer le début des états non-adaptables, nous recherchons pour chaque chemin d'exécution le premier début d'appel vers un composant à modifier et le dernier retour d'appel. Dans notre cas, nous cherchons à basculer vers la stratégie PBR. Le composant réalisant la compensation, appelé « Semi-Active Replication Manager » (cf. figure 56), est alors le seul composant modifié pour cette tâche.

Nous rappelons que l'état de l'exécution est caractérisé par la séquence de transitions franchises depuis la place représentant l'inactivité de la tâche (début du cycle). Ces séquences peuvent alors être regroupées de la manière suivante :

- Les séquences de transition qui ne contiennent pas la transition correspondante au premier appel vers le composant en charge de la compensation appartiennent à E_1 ;
- Les séquences de transition qui contiennent le dernier appel vers le composant en charge de la compensation appartiennent à E_2 ;
- Les autres séquences appartiennent à E_3.

Une fois ces séquences déterminées, il est possible de construire une simplification du modèle de l'exécution faisant apparaître uniquement les transitions faisant changer d'état, et les transitions sources de causalité et de synchronisation (cf. figure 60).

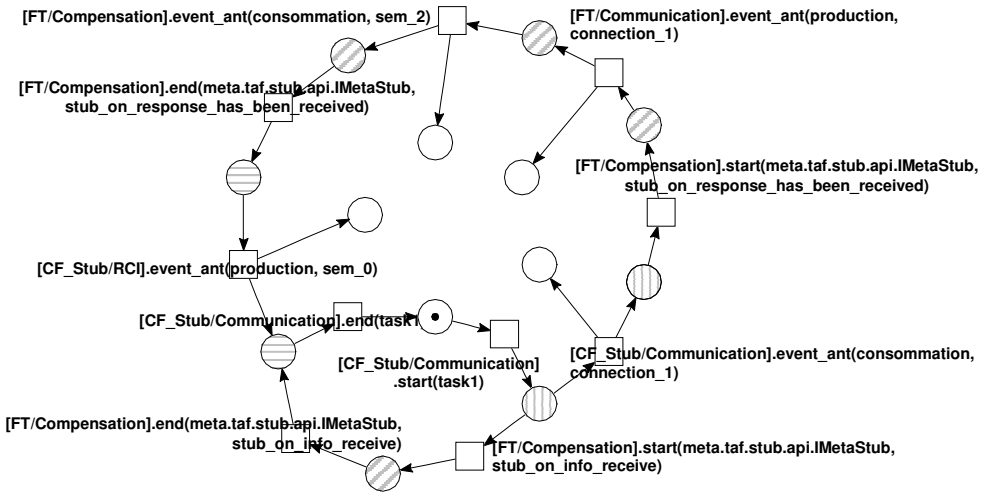


Figure 60 Graphe des états adaptables de la tâche du stub

Les tâches des répliques du contrôleur sont nettement plus impactées par les modifications du système, puisque les composants à modifier appartiennent à ces nœuds. Chacune de ces tâches possède des états de l'exécution ne permettant pas l'adaptation (E_3). Toutefois, le grand atout de cette approche est le fait que ces tâches possèdent toutes de nombreux états permettant l'adaptation. Ainsi, par l'introduction de la notion d'états adaptables et le contrôle de l'exécution assurant le maintient des tâches dans un état adaptable, **l'adaptation peut être réalisée en concurrence avec l'exécution de ces tâches, sans nécessiter leur arrêt total**.

IV.5 RESULTATS EXPERIMENTAUX

IV.5.1 Coût du modèle de l'exécution

Dans cette section, nous nous intéressons au coût de l'insertion du modèle. Le modèle impacte les performances lors de deux phases différentes du fonctionnement du système :

- Lors du déploiement et de l'application des modifications: le modèle doit alors être construit ou mis à jour, à partir de la description des chemins d'exécution internes aux composants de l'architecture. Le temps nécessaire à cette mise à jour dépend principalement de la complexité des chemins d'exécution au sein de l'architecture à composant. Il existe une autre manière de réaliser le modèle : le compiler hors ligne et le charger statiquement au démarrage. La mise à jour consiste alors à charger le modèle de la configuration suivante, et de transférer les marquages de l'ancien vers le nouveau modèle. Cette deuxième solution est plus performante, mais est nettement moins flexible, car il est nécessaire de déterminer les modèles hors-ligne, de les stocker et de définir les fonctions de transfert des états pour chaque adaptation. Nous avons choisi la première solution.
- Pendant le fonctionnement normal du système : en fonctionnement, le modèle doit être synchronisé avec l'exécution du système. Ainsi, pour tout évènement et chaque appel entre deux composants, le modèle doit être mis à jour en franchissant la transition qui correspond à l'évènement ou à l'appel. Cette mise à jour peut être très lente (si elle nécessite le parcourt de tout le réseau de Petri pour trouver la transition) ou très rapide (si lors de chaque mise-à-jour, les transitions suivantes sont pré-calculées à partir de l'état courant). Dans notre cas, nous avons choisi la deuxième solution.

Pour mettre en lumière le coût du modèle, nous avons comparé les temps nécessaires au chargement complet d'une réplique LFR avec et sans modèle, ainsi que le temps d'un cycle pour la tâche de régulation du contrôleur. Ces résultats sont fournis dans le tableau 14.

Tableau 14 **Performances et modèle**

Cible	*Temps de chargement d'une réplique*	*Temps d'exécution d'un cycle*
Sans modèle	1,3 s	5 ms
Avec modèle	8,4 s	10 ms

Nous pouvons remarquer que la construction du modèle rallonge considérablement les temps de démarrage d'une réplique. Notre implémentation n'est pas optimisée, dans le sens où la description des chemins d'exécution des composants est transformée en réseau de Petri, en ligne, pendant le déploiement de l'architecture à composant. Cette compilation contribue à hauteur de 3 secondes sur le temps total de chargement. De plus, les accès aux fichiers de descriptions ne sont pas optimisés. Pour cela, il serait possible d'insérer un tampon où tous les fichiers nécessaires seraient prés chargés et analysés. Ce tampon consommerait plus de mémoire, mais permettrait d'obtenir des performances supérieures.

IV.5.2 Adaptation VS redémarrage complet

Dans le cas d'une conception classique, le logiciel de tolérance aux fautes est fortement couplé au logiciel fonctionnel. Les modifications du logiciel de tolérance aux fautes ne peuvent se faire que par un redémarrage complet de l'application.

Notre manière de concevoir le logiciel critique permet une modification beaucoup plus fine du logiciel pendant son fonctionnement. Ainsi, il n'est pas nécessaire de redémarrer et redéployer le logiciel lors d'une adaptation.

Dans le cas du contrôleur automatique, nous allons montrer que la modification du logiciel de tolérance aux fautes est rendue possible par l'utilisation de notre conception pour l'adaptation.

Dans notre exemple du contrôleur automatique, le système possède cinq grandeurs temporelles caractéristiques :

- La période de la tâche du contrôleur T_{period}
- La durée de la tâche de contrôle T_{task} munis de la stratégie de tolérance aux fautes (avant ou après adaptation)
- La durée nécessaire à la synchronisation du système T_{synch} pour atteindre un état adaptable et l'y maintenir
- La durée de la reconfiguration architecturale et l'initialisation de l'état T_{reconf}
- La durée totale de l'adaptation T_{adapt}

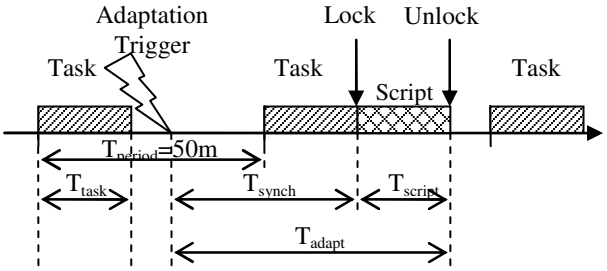


Figure 61 Temps caractéristiques

L'algorithme utilisé par le contrôleur pour réguler la position du pendule est naturellement robuste et permet l'introduction d'une latence de trois périodes. Une adaptation correcte du point de vue temporelle dans notre cas nécessite d'être capable d'appliquer les modifications sur le système sans que le pendule ne tombe. Ainsi, il existe une contrainte sur le temps nécessaire à la reconfiguration qui est :

$$T_{script} < 4 \times T_{period} - T_{task}$$

Les deux configurations ont été testées en utilisant quatre Pentium 1GhZ avec un noyau Linux 2.6.18 connectés à l'aide d'un réseau Ethernet. L'implémentation repose sur un noyau modifié du modèle à composant OpenCOM introduisant les différents modèles introduits dans ces travaux. Nous avons mesuré les temps caractéristiques de plusieurs configurations :

- Le système non-distribué (capteurs, commande, contrôleur et console sur le même nœud)
- Le système distribué (trois nœuds nécessaires)
- Le système munis de la stratégie LFR appliquée au contrôleur (tolérance d'un crash et nécessite quatre nœuds sur le réseau)
- Le système munis de la stratégie PBR appliquée au contrôleur (idem que pour le LFR)

Les mesures réalisées sont reportées dans le tableau 15. On remarque alors que la distribution est la première cause d'augmentation du coût temporel de l'exécution. Le surcoût d'une stratégie de tolérance aux fautes reste cependant raisonnable puisque l'exécution reste très largement en dessous d'un période T_{period}*=50ms*. De plus, il existe une grande variabilité de

temps d'exécution principalement issue de la technologie Java utilisée et de l'utilisation d'un *garbage collector* qui réalise des tâches de nettoyage de la mémoire non-maîtrisées ici.

Tableau 15 **Mesures du temps d'un cycle pour la tâche de régulation**

	Non-distribué	*Distribution*	*PBR*	*LFR*
Moyenne Ttâche (ms)	0,2	6,1	9,2	10,0
Pire cas Ttâche (ms)	1	13,7	21,4	26,0

Le temps disponible pour adapter la stratégie de tolérance aux fautes est donc de 174ms. Les pires temps d'adaptation enregistrés dans nos expériences sont de 120ms. Ces temps pourraient grandement améliorés par une implémentation plus optimisée du logiciel d'adaptation, par l'utilisation de technologies plus à même de respecter des contraintes temporelles ou par une maîtrise du *garbage collector*. Cette dernière problématique est récurrente à l'utilisation de Java pour les applications où le temps joue un rôle important comme l'illustre les *Real-Time Specifications for Java*[11].

Comparés aux temps de redémarrage complet d'une réplique du contrôleur (de l'ordre de 3s), l'adaptation est nettement avantageuse. Ainsi, une conception classique d'une application tolérante aux fautes ne permet pas de réaliser les modifications du logiciel de tolérance aux fautes dans ce cas d'étude, ce que permet notre conception qui repose sur une architecture réflexive et une conception orientée composant du logiciel de tolérance aux fautes.

[11] RTSJ : https://rtsj.dev.java.net/

IV.6 CONCLUSION

Dans une conception classique d'une application tolérante aux fautes, il existe un couplage fort entre le logiciel de tolérance aux fautes et le logiciel fonctionnel. Adapter les mécanismes de tolérance aux fautes nécessite alors d'arrêter le logiciel pour redémarrer une autre version contenant le logiciel de tolérance aux fautes souhaité. Ce redémarrage complet possède un fort coût en ressources et par exemple, en CPU ou en temps. De plus, ce redémarrage complet provoque une indisponibilité du service rendu par l'application. Enfin, la continuité de service n'est pas assurée du fait que le logiciel redémarre avec un état neuf, ne tenant pas compte de son exécution passée.

Pour pallier à cette problématique, ces travaux se sont intéressés à séparer le logiciel de tolérance aux fautes du logiciel applicatif et de permettre sa modification fine pendant l'exécution. Ainsi, il n'est pas nécessaire de redémarrer complètement le service pour modifier un morceau du logiciel de tolérance aux fautes.

Dans ce chapitre, nous avons conçu le logiciel de tolérance aux fautes en utilisant une conception orientée composant. Cette conception repose sur la taxonomie de la tolérance aux fautes et n'est pas figée. Ainsi, les mécanismes de tolérance aux fautes proposés ici ne sont pas les seuls sur lesquels l'adaptation fonctionne, mais uniquement un exemple illustratif. Cette conception peut être simplifiée par l'utilisation d'outils. Ces outils permettent de générer les descriptions des composants. Ainsi, les configurations peuvent être construites graphiquement à partir de ces documents à l'aide d'un outil appelé OpenCOM ADL Factory. Cet outil permet de générer les descriptions des configurations du système, de générer le modèle de l'exécution, et d'en faire l'analyse pour l'adaptation. Nous avons ensuite illustré ces travaux par un cas d'étude. Il a montré que notre conception pour l'adaptation rend possible la modification du logiciel alors qu'une conception plus classique ne le permettait pas. Ainsi, le coût de l'adaptation peut être comparé entre ces deux stratégies de conception (cf. Figure 62).

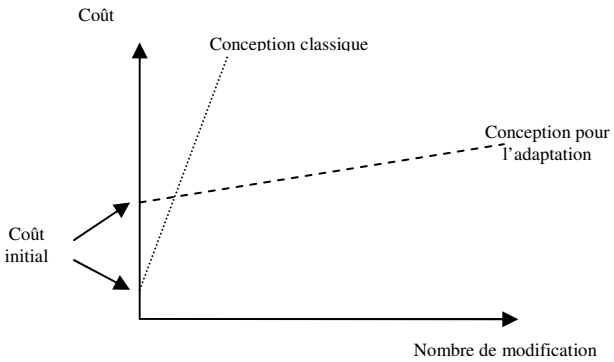


Figure 62 Allure de la comparaison du coût de modification d'un système

Dans une conception classique, le coût initial est celui du déploiement et du démarrage du système. Ce coût est relativement faible comparé au coût de mise en œuvre d'un système adaptable. Toutefois, dans le cas d'une conception classique, un changement de contexte opérationnel, et donc une modification du logiciel, possède un coût unitaire important. Cela n'est pas le cas avec une conception pour l'adaptation. Ainsi, nous pouvons conclure que l'investissement initial d'une conception pour l'adaptation est rapidement amorti pour un système subissant de nombreux changements opérationnels.

CONCLUSION ET PERSPECTIVES

L'évolution des systèmes en opération devient un impératif pour de nombreuses raisons : changement d'environnement d'utilisation, mais aussi évolution des spécifications, maintenance en ligne, optimisation des performances, etc. Cette tendance concerne aussi les systèmes critiques pour lesquels la sûreté de fonctionnement est une préoccupation majeure. Durant la vie opérationnelle du système, leur sûreté de fonctionnement est renforcée par l'utilisation de la tolérance aux fautes. Pour pallier aux variations environnementales, il est nécessaire d'adapter les mécanismes de tolérance aux fautes aux conditions opérationnelles, à savoir les ressources, le modèle de fautes, la configuration matérielle et logicielle. Dans ce contexte, nous proposons une approche permettant de modifier dynamiquement le logiciel de tolérance aux fautes afin qu'il corresponde au mieux aux conditions opérationnelles du système.

Adapter le logiciel de tolérance aux fautes nécessite de concevoir le système pour permettre sa modification à l'exécution. Les approches existantes souffrent d'un manque de flexibilité en termes de personnalisation des mécanismes de tolérance aux fautes. Notre approche vise à séparer la conception du logiciel de tolérance aux fautes et le traitement de son adaptation dans le but de pouvoir adapter un logiciel de tolérance aux fautes réalisé par un concepteur de système tiers. Nous proposons de concevoir le système tolérant aux fautes en séparant fonctionnalités, tolérance aux fautes et adaptation dans des couches du système. Cette séparation repose sur l'utilisation d'une architecture réflexive. De plus, pour permettre la modification du logiciel, nous proposons d'utiliser technologies récentes de développement logiciel à base de composants. Ces méthodes de développement logiciel permettent d'abstraire le logiciel à un ensemble de boîtes élémentaires interconnectées et de manipuler ces boîtes pendant le fonctionnement du système. Les approches classiques de l'adaptation basée composant s'intéressent à la modification de composants indépendants. Ils utilisent la notion de quiescence des composants qui permettent de maintenir un invariant local au composant pendant le processus d'adaptation. Dans le cas d'un logiciel critique distribué qu'est le logiciel de tolérance aux fautes, la correction du processus d'adaptation ne peut pas être définie à partir d'invariants locaux à chaque composant. Nous avons introduit la notion d'état adaptable reposant sur la définition d'un invariant global impliquant l'ensemble des composants à modifier. Cet invariant impose une contrainte supplémentaire sur la conception du logiciel : utiliser des tâches périodiques bornée dans le temps.

Nous avons introduit un modèle compositionnel de l'exécution construit à partir de l'architecture à composant déployée. Une analyse hors-ligne de cette représentation permet d'étudier le comportement du système pendant son adaptation, et de définir les états adaptables du système qui assurent la correction du processus d'adaptation. Cette étude amène à définir les mécanismes de contrôle de l'exécution du système afin de le placer dans un état adaptable, et de maintenir son exécution dans l'ensemble de ses états adaptables. Le service subit alors une interruption minimum, car le système continu à s'exécuter en concurrence au processus d'adaptation tant que son exécution ne l'amène pas dans un état non-adaptable.

La réflexivité a encore une fois démontrée ses qualités en termes de séparation et de généralisation du traitement de problématiques transverses d'un système. En effet, les résultats de ces travaux sont appliqués au logiciel de tolérance aux fautes, mais peuvent finalement être utilisés à d'autres fins.

Les expérimentations mises en place dans ces travaux ont permis d'illustrer la faisabilité de l'adaptation de la tolérance. Ces expérimentations ont montré comment décomposer le logiciel de tolérance aux fautes en composants. L'implémentation d'une plateforme expérimentale sous OpenCOM a permis de réaliser l'adaptation d'une stratégie Leader

Follower Replication à une stratégie Primary Backup Replication pour un système de contrôle commande. Une deuxième expérimentation a été menée pour la maintenance à chaud du même système. Cette deuxième expérience a montré d'une part comment les aspects décisionnels peuvent être couplés avec la plateforme d'expérimentation et d'autre part que notre approche s'applique aussi à l'adaptation d'un logiciel fonctionnel. Ce deuxième cas d'étude a été mené au sein du mini-projet ASAP[12] (*Assessement-based Adaptable Software Architecture for Dependability*) dans le cadre du réseau d'excellence européen ReSIST (*Resilience for Survavility in IST*).

Les travaux de cette thèse ne se sont pas intéressés aux aspects temporels. De notre point de vue, les problèmes temporels pendant l'adaptation peuvent être résolus par la planification des modifications du logiciel. Pour cela, il est nécessaire d'introduire un modèle temporel de l'exécution, mais aussi de connaître les temps pire d'exécution des différentes actions de modification du logiciel.

L'utilisation des réseaux de Petri pour représenter l'exécution du système permet d'introduire les notions temporelles, par exemple, en associant aux arcs du réseau les pires temps d'exécution. Il pourrait alors être possible de prévoir les temps disponibles pour l'adaptation et de planifier les différentes actions de modification du logiciel.

Ces travaux ont montré qu'il est très difficile d'évaluer une adaptation du logiciel principalement parce que les objectifs de l'adaptation et les propriétés que l'on souhaite conserver sont différents d'un système à l'autre. Nous pensons qu'il est aujourd'hui nécessaire de s'intéresser à cette problématique et de trouver des méthodes pour évaluer les approches de l'adaptation, afin de permettre l'injection de ces nouvelles technologies dans des processus de conception de systèmes critiques.

[12] City University (Londres/UK), Université de Kaunas (Lituanie), Université de Lancaster (UK), LAAS-CNRS (Toulouse/France)

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secure Computing*, vol. 1, pp. 11--33, 2004.

[2] A. Avizienis, "The Methodology of N-Version Programming," in *Chapter 2 of "Software Fault Tolerance"*, M. R. Lyu, Ed.: Wiley, 1995, pp. 23-46.

[3] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," presented at the 1988 ACM SIGMOD international conference on Management of data, Chicago, Illinois, United States, 1988.

[4] D. Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, vol. 1: SpringerVerlag, 1991.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, pp. 374-382, 1985.

[6] S. Maffeis and D. C. Schmidt, "Constructing Reliable Distributed Communication Systems with Corba," *IEEE Communications Magazine*, vol. 35, pp. 56-61, 1997.

[7] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "The Interception Approach to Reliable Distributed CORBA Objects," in *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Portland, Oregon, USA, 1997, pp. 245-248.

[8] P. Narasimhan, T. A. Dumitraş, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: support for Real-Time Fault-Tolerant CORBA," in *Concurrency and Computation: Practice and Experience*, vol. 17, Wiley and Sons ed, 2005, pp. 1527-1545.

[9] T. A. Dumitraş, D. Srivastava, and P. Narasimhan, "Architecting and Implementing Versatile Dependability," in *Architecting Dependable Systems III*, R. d. Lemos, C. Gacek, and A. Romanovsky, Eds., 2005, pp. 212.

[10] P. Maes, "Concepts and experiments in computational reflection," presented at Conference on Object-Oriented Programming Systems, Languages, and Applications, Orlando, Florida, 1987.

[11] M.-O. Killijian, J.-C. Fabre, J. C. Ruiz-Garcia, and S. Chiba, "A Metaobject Protocol For Fault-Tolerant CORBA Applications," presented at 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, USA, 1998.

[12] M.-O. Killijian, J. C. Ruiz-Garcia, and J.-C. Fabre, "Using Compile Time Reflection for Object State Capture," in *2nd International Conference Reflection'99*. Saint-Malo, France, 1999, pp. 150-152.

[13] F. Taïani and J.-C. Fabre, "A multi-level meta-object protocol for fault-tolerance in complex architectures," presented at International Conference on Dependable Systems and Networks, Yokohama, Japan, 2005.

[14] G. Agha, S. Frølund, R. Panwar, and D. Sturman, "A Linguistic Framework for Dynamic Composition of Dependability Protocols," presented at Proceedings of the IFIP Conference on Dependable Computing for Critical Applications, 1993.

[15] B. Garbinato, R. Guerraoui, and K. R. Mazouni, "Implementation of the GARF Replicated Objects Platform," *Distributed Systems Engineering Journal*, vol. 2, pp. 14--27, 1995.

[16] J.-C. Fabre and T. Perennou, "A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach," *IEEE Transactions on Computers*, vol. 47, pp. 78-95, 1998.

[17] M.-O. Killijian and J.-C. Fabre, "Implementing a Reflective Fault-Tolerant CORBA System," presented at 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), Nürnberg, Germany, 2000.

[18] S. Chiba, "A Metaobject Protocol for C++," presented at Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1995.

[19] P. Felber, B. Gabrinato, and R. Guerraoui, "The Design of a CORBA Group Communication Service," in *Proc. 15th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 1996, pp. 150-159.

[20] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under UNIX," presented at USENIX Technical Conference, New Orleans, Louisiana, USA, 1995.

[21] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, vol. 8, pp. 66-73, 1991.

[22] J. Penix and P. Alexander, "Efficient specification-based component retrieval," *Automated Software Engineering*, vol. 6, pp. 139-170, 1999.

[23] B. Ravindran, L. Peng, and T. Hegazy, "Proactive resource allocation for asynchronous real-time distributed systems in the presence of processor failures," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 1219-1242, 2003.

[24] EBM-WebSourcing, "PEtALS, http://ebmwebsourcing.com/produits/petals.html."

[25] S. Vinoski, "Java Business Integration," *IEEE Internet Computing*, vol. 9, pp. 89-91, 2005.

[26] M. E. Segal and O. Frieder, "On-the-fly program modification: systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53 - 65 1993.

[27] M. Franz, "Dynamic linking of software components," *Computer*, vol. 30, pp. 74-&, 1997.

[28] G. Hjálmtýsson and R. Gray, "Dynamic C++ Classes: A lightweight mechanism to update code in a running program," presented at USENIX, New Orleans, {LA}, 1998.

[29] H. Cervantes and R. S. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," presented at 26th International Conference on Software Engineering (ICSE'04), 2004.

[30] P. Costa, G. Coulson, C. Mascolo, L. Motolla, G. P. Picco, and S. Zachariadis, "A Reconfigurable Component-Based Middleware for Networked Embedded Systems," *International Journal of Wireless Information Networks*, vol. 14, pp. 149-162, 2007.

[31] H. A. Duran-Limon, G. S. Blair, A. Friday, T. Sivaharan, and G. Samartzidis, "A Resource and QoS Management Framework for a Real-Time Event System in Mobile Ad Hoc Environments," presented at Ninth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003 Fall), Capri Island, Italy, 2003.

[32] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri, "AQuA: An Adaptive Architecture that

Provides Dependable Distributed Objects," *IEEE Transactions on Computer*, vol. 52, pp. 31-50, 2003.

[33] C. Sabnis, M. Cukier, Y. J. Ren, P. Rubel, W. H. Sanders, D. E. Bakken, and D. A. Karr, "Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA," in *7th IFIP Internation Working Conference on Dependable Computing for Critical Applications*, 1998, pp. 149-168.

[34] Y. J. Ren, P. Rubel, M. Seri, M. Cukier, W. H. Sanders, and T. Courtney, "Passive Replication Schemes in AQuA," in *the 2002 Pacific Rim International Symposium on Dependable Computing*. Tsukuba, Japan, 2002, pp. 125-130.

[35] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "A Dynamic Replica Selection Algorithm for Tolerating Timing Faults," presented at Proceedings of the International Conference on Dependable Systems and Networks DSN-2001, Göteborg, Sweden, 2001.

[36] R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects," presented at Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England, 1998.

[37] S. M. Pertet and P. Narasimhan, "Proactive Recovery in Distributed CORBA Applications," in *International Conference on Dependable Systems and Networks (DSN'04)*. Florence, Italy, 2004, pp. 357.

[38] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. 1, pp. 220-232, 1975.

[39] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Transactions on Computers*, vol. 38, pp. 626 - 636, 1989

[40] K. H. Kim, J. Goldberg, T. F. Lawrence, and C. Subbaraman, "The Adaptable Distributed Recovery Block Scheme And A Modular Implementation Model," presented at Pacific Rim International Symposium on Fault-Tolerant Systems, 1997.

[41] K. H. Kim, "ROAFTS: A Middleware Architecture for Real-Time Object-Oriented Adaptive Fault Tolerance Support," *IEEE High Assurance Systems Engineering*, pp. 50-57, 1998.

[42] M. Hecht, H. Hecht, and E. Shokri, "Adaptive fault tolerance for spacecraft," in *IEEE Aerospace 2000 Conference*, vol. 5. Big Sky, MT, 2000, pp. 521-533.

[43] J. Fraga, F. Siqueira, and F. Favarim, "An Adaptive Fault-Tolerant Component Model," presented at the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, Capri Island, Italy, 2003.

[44] K. H. Kim and T. F. Lawrence, "Adaptive Fault Tolerance: Issues and Approaches," in *Proceedings of Second IEEE Workshop on Future Trends of Distributed Computing Systems*. Cairo, Egypt, 1990, pp. 38-46.

[45] OMG, "http://www.omg.org/technology/documents/formal/components.htm."

[46] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant, "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 560-579, 1999.

[47] J. C. Ruiz-Garcia, M.-O. Killijian, J.-C. Fabre, and P. Thévenod-Fosse, "Reflective Fault-Tolerant Systems: From Experience to Challenges," *IEEE Trans. Comput.*, vol. 52, pp. 237-254, 2003.

[48] B. C. Smith, "Reflection and Semantics in a Procedural Language," in *MIT-LCS-TR-272*. Massachusetts: Massachusetts Institute of Technology, 1982.

[49] F. Taïani, "La réflexivité dans les architectures multi-niveaux: application aux systèmes tolérant les fautes," Doctorat en *Systèmes Informatiques* délivré par l'Université Paul Sabatier de Toulouse, 2004, pp. 155.

[50] B. C. Smith, "Reflection and semantics in LISP," *ACM Symposium on Principles of Programming Languages*, pp. 23-35, 1984.

[51] F. Rivard, "Smalltalk: a Reflective Language," presented at Reflection '96, San Francisco, California, 1996.

[52] P. Chan, R. Lee, and D. Kramer, *The Java Class Libraries*, vol. 1, Second ed: Addison-Wesley, 1998.

[53] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," presented at European Conference on ObjectOriented Programming, London, UK, 2001.

[54] G. Coulson, G. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transactions on Computer Systems*, vol. 26, 2008.

[55] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal Component Model and Its Support in Java," *Software Practice and Experience*, vol. 36 (11-12), pp. 29, 2006.

[56] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering Architectures from Running Systems," *IEEE Transactions on Software Engineering*, vol. 32, pp. 454-466, 2006.

[57] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," presented at Middleware, 2001.

[58] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, "An Open Component Model and Its Support in Java," presented at 7th International Symposium on Component-Based Software Engineering, 2004.

[59] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63-75, 1985.

[60] D. B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms*, vol. 11, pp. 462-491, 1990.

[61] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transaction on Software Engineering*, vol. 13, pp. 23-31, 1987.

[62] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A Distributed Domino-Effect Free Recovery Algorithm," in *IEEE Int. Symposium on Reliability Distributed Software and Database*, 1984, pp. 207-215.

[63] M.-O. Killijian, J. C. Ruiz-Garcia, and J.-C. Fabre, "Portable Serialization of CORBA Objects: a Reflective Approach," presented at Conference on Object-Oriented Programming Systems, Languages and Applications, Seattle, Washington, USA. SIGPLAN Notices 37(11), 2002.

[64] S. Lin, F. Taïani, and S. G. Blair, "Facilitating Gossip Programming with the GossipKit Framework," in *Distributed Applications and Interoperable Systems*, vol. 5053: Springer Berlin / Heidelberg, 2008, pp. 238-252.

[65] J. Kramer and R. J. Cunningham, "Towards a Notation for the Functional Design of Distributed Processing Systems," presented at International Conference on Parallel Processing (ICPP'78), Bellaire, Michigan, 1978.

[66] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1293-1306, 1990.

[67] J. Hopcroft, "An n log n algorithm for minimizing the states in a finite-automaton," in *Theory of Machines and Computations*, Z. Kohavi, Ed. New York , NY , USA: Academic Press, 1971, pp. 189-196.

[68] A. T. A. Gomes, T. Vasconcelos Batista, A. Joolia, and G. Coulson, "Architecting Dynamic Reconfiguration in Dependable Systems," in *WADS*, 2006, pp. 237-261.

[69] D. Garlan, R. T. Monroe, and D. Wile, "Acme: An Architecture Description Interchange Language," presented at CASCON'97 Toronto, Ontario, 1997.

[70] M. Leclercq, A.-E. Ozcan, V. Quéma, and J.-B. Stefani, "Supporting Heterogeneus Architecture Descriptions in an Extensible Toolset," presented at 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, 2007.

[71] R. Rouvoy, N. Pessemier, R. Pawlak, and P. Merle, "Using Attribute-Oriented Programming to Leverage Fractal-Based Developments," presented at European Conference on Object-Oriented Programming, Nantes, France, 2006.

TABLE DES MATIERES

INDEX DES FIGURES

INDEX DES TABLES

Title:
On-line fault tolerance mechanisms adaptation based on open component models

ABSTRACT

On-line fault tolerance adaptation aims at enforcing system dependability by taking into account operational conditions and environment. Adapting the system requires new design techniques. This work aims at understanding and mastering the impact of such software modification in operation, especially regarding side effects on functionalities and dependability properties.

Our approach relies on a reflective architecture based on components and models of the software that reflects on the one hand the content of the software in terms of state and algorithms (architectural model) and on the other hand the expected correct behaviour (behavioural model). The first one is used to determine the modifications and apply them at runtime, and the second one is used to drive the system in a state in which modifications can be done consistently, and maintain the system in such a state.

We show that; thanks to manipulation capabilities and execution control, we can master the modification of fault tolerance software and ensure correctness properties.

KEYWORDS:
adaptation, reflection, component model, fault tolerance, behavioural modelling

AUTEUR :
Thomas Pareaud

TITRE :
Adaptation en-ligne de mécanismes de tolérance aux fautes par une approche à composants ouverts

DIRECTEUR DE THESE :
Jean-Charles FABRE et Marc-Olivier Killijian

LIEU ET DATE DE SOUTENANCE :
Toulouse, le 27 janvier 2009

RESUME

L'adaptation en-ligne du logiciel de tolérance aux fautes permet de renforce la sûreté de fonctionnement du système et prenant en compte son environnement. L'adaptation nécessite de nouvelles techniques de conception. Ces travaux visent à comprendre et maîtriser l'impact des modifications du logiciel de tolérance aux fautes en opération sur les fonctionnalités du système, pour en maîtriser les effets de bords.

L'approche proposée introduit une architecture réflexive à composants et une modélisation du logiciel. Un modèle structurel du logiciel permet de calculer et appliquer les modifications du contenu du logiciel. Un modèle comportemental décrit les observations attendues en fonctionnement. Il permet de déterminer les états permettant d'appliquer les modifications, d'amener et de maintenir le système dans ces états.

Ces travaux montrent que, grâce aux capacités de manipulation et de contrôle en ligne du logiciel, la modification des mécanismes de tolérance aux fautes peut être réalisée en ligne de manière maîtrisée.

MOTS-CLES :
adaptation, réflexivité, modèle à composant, tolérance aux fautes, modèle comportemental

DISCIPLINE ADMINISTRATIVE :
Systèmes Informatiques

INTITULE ET ADRESSE DE L'U.F.R. OU DU LABORATOIRE :

Laboratoire d'Analyse et d'Architecture des Systèmes
7 av. du Colonel Roche
31077 Toulouse Cedex 4
France