Académie de Montpellier
Université Montpellier II
Sciences et Techniques du Languedoc

MÉMOIRE DE STAGE DE MASTER M2

effectué au Laboratoire d'Informatique de Robotique
et de Micro-électronique de Montpellier

Spécialité : **Professionnelle et Recherche unifiée en Informatique**

Gestion des exceptions dans un système multi-agents avec réplication

par **Selma Kchir**

Date de soutenance :

22 Juin 2010

Sous la direction de :

Christophe Dony **Chouki Tibermacine** - **Lirmm**
Christelle Urtado **Sylvain Vauttier** - **Lgi2p**

Je dédie ce mémoire

A mes parents pour leur amour inestimable, leur confiance, leur soutien, leurs sacrifices et toutes les valeurs qu'ils ont su m'inculquer.

A ma grand-mère Dalila pour sa douceur et sa gentillesse.

A mes soeurs ainsi qu'à mes beaux frères pour leur tendresse, leur complicité et leur présence malgré la distance qui nous sépare.

A ma tante Samia et mon oncle Jean-Marie pour toute l'affection qu'ils m'ont donnée et pour leurs précieux encouragements.

A ma nièce Lina, ma plus grande source de bonheur, j'espère que la vie lui réserve le meilleur.

A toute ma famille ainsi qu'à mes amis.

Remerciements

Je tiens à remercier mon directeur de stage Monsieur Christophe Dony, professeur à l'Université Montpellier II, de m'avoir accueillie dans son équipe et d'avoir accepté de diriger ce travail. Sa rigueur scientifique, sa disponibilité et ses qualités humaines m'ont profondément touchée.

Mes remerciements s'adressent également à mon encadrant Monsieur Chouki Tibermacine, maître de conférence à l'Université Montpellier II, pour avoir accepté de diriger ce travail. Son soutien, sa clairvoyance et ses compétences m'ont été d'une aide inestimable.

Je remercie également mon encadrante Madame Christelle Urtado, enseignant-chercheur à l'école des mînes d'Alès, pour le priviliège qu'elle m'a fait en acceptant de diriger ce travail. Sa gentillesse, sa modestie, sa riche expérience et l'accueil cordial qu'elle m'a toujours réservé m'ont inspiré une grande admiration à son égard.

Je tiens à remercier également mon encadrant Monsieur Sylvain Vauttier, enseignant-chercheur à l'école des mînes d'Alès, pour l'honneur qu'il m'a fait en acceptant de diriger ce travail. Sa disponibilité, sa gentillesse et ses précieuses directives tout au long de la réalisation de ce travail m'ont beaucoup impressionnée.

Qu'ils puissent trouver dans ce travail le témoignage de ma sincère gratitude et de mon profond respect.

Je souhaite remercier Sylvain Ductor pour toute l'aide qu'il m'a apportée.

Je tiens à remercier sincèrement les membres du jury qui me font le grand honneur d'évaluer ce travail.

Mes remerciements les plus chaleureux vont à tous mes camarades au LIRMM pour leurs encouragements et pour l'ambiance agréable tout au long de ce stage et en particulier à Pierre pour sa présence dans les moments difficiles et grâce à qui j'ai passé d'excellents moments.

Résumé

La gestion des exceptions et la réplication sont deux mécanismes complémentaires de tolérance aux fautes. La gestion des exceptions offre aux programmeurs la possibilité de définir des structures de contrôle pour rattraper et traiter des exceptions signalées. La réplication traite les fautes du système en se basant sur la redondance. Combiner ces deux techniques est une première étape vers la construction de systèmes fiables. Ce mémoire présente les résultats du projet FACOMA. Il propose une spécification du système de gestion d'exceptions, XSaGE, qui est une adaptation de SaGE pour gérer des agents répliqués. Il décrit ensuite l'implémentation de ce système dans un environnement multi-agents répliqués, DIMAX, et son application à un exemple de commerce électronique.

Mots clés : tolérance aux fautes, réplication, gestion d'exceptions.

Abstract

Exception handling and replication are two complementary mechanisms that increase software reliability. Exception handling helps programmers in controlling situations in which the normal execution flow of a program cannot continue. Replication handles system failures through redundancy. Combining both techniques is a first step towards building a trustworthy software engineering framework. This report presents the results of the FACOMA project. It proposes the specification of an exception handling system for replicated agents as an adaptation of the SaGE proposal : XSaGE. It then describes its implementation in the DIMAX replicated agent environment and its application to an e-commerce example.

Keywords : Fault tolerance, replication, exception handling.

Table des matières

Table des figures

Introduction générale

Avec les progrès de la technologie de l'information, on note une constante évolution dans plusieurs domaines tels que le commerce électronique, les jeux vidéos, les effets spéciaux dans les oeuvres de fiction, etc. Des entités intelligentes, réactives et autonomes ont pris une place prépondérante dans ces systèmes ; ces entités sont appelées des agents. Ces derniers interagissent dans leur environnement constituant ainsi un système multi-agents [9] (SMA). Au sein de ces systèmes, chaque agent possède un rôle bien déterminé et doit être capable de prendre des décisions pour atteindre ses objectifs. Toutefois, le dysfonctionnement d'un agent en raison de l'interruption de la connexion ou de la panne de la machine sur laquelle il était en train de s'exécuter peut engendrer la défaillance du système. Les systèmes multi-agents doivent alors être tolérants aux fautes. De ce fait, plusieurs approches ont été appliquées à ces systèmes tels que l'approche préventive (Réplication) et l'approche curative (Gestion des exceptions). L'approche préventive consiste à gérer un dysfonctionnement possible du système en se basant sur la redondance et l'approche curative permet au programmeur de gérer dynamiquement les situations qui engendrent un dysfonctionnement d'un processus donné.

L'objectif de ce travail est de combiner ces deux approches afin d'assurer la fiabilité des systèmes multi-agents. Dans ce cadre, nous avons étudié la mise en place d'un système de gestion des exceptions dans la plate-forme multi-agents DIMAX [8] gérant, de façon transparente au porgrammeur, des agents répliqués dans le but de permettre la cohabitation des mécanismes et d'étudier les éventuelles améliorations de l'un par l'autre. Ce travail constitue l'un des objectifs du projet ANR FACOMA [1].

La plate-forme multi-agents DIMAX est la combinaison de DARX (fra-

1. Le projet FACOMA : Fiabilisation Adaptative d'applications COopératives Multi-Agents (`http://facoma.lip6.fr/`) est financé par l'Agence Nationale de Recherche (ANR) et a été développé par plusieurs équipes du LIP6, de l'INRIA, du LIRMM et du LGI2P.

mework de réplication) [17] et de DIMA (système multi-agents) [11]. Pour assurer la fiabilité de cette plate-forme, la réplication et plus précisément, la réplication active est utilisée comme approche préventive. Suivant l'importance (criticité) [3] [7] [12] d'un agent dans le système, des copies de cet agent, appelées répliques, sont créées formant ainsi un groupe de réplication et l'une de ces répliques est définie comme leader du groupe. Les messages envoyés à un agent répliqué sont alors envoyés automatiquement et d'une façon transparente à toutes ses répliques qui traitent le même message en parallèle et renvoient leurs réponses. Il est ainsi possible de remplacer un agent défaillant par l'une de ses répliques. Ce mécanisme est complètement transparent pour l'utilisateur.
Contrairement à la réplication, la gestion des exceptions est une approche curative dédiée aux développeurs. Un système de gestion d'exceptions offre aux programmeurs trois ensembles de primitives pour signaler des exceptions, définir des handlers (continuations exceptionnelles) [19] et remettre le système dans un état cohérent. Une exception n'est pas considérée comme une faute tant qu'elle n'a pas entraîné l'arrêt de l'exécution du programme. Lorsqu'une exception est signalée par un agent, l'exécution de celui-ci est suspendue et la recherche d'un handler est lancée.
Gérer les exceptions dans les systèmes multi-agents, tout en utilisant la réplication, est une contribution importante dont le but est d'augmenter la fiabilité de ces systèmes. Cela pose de nombreuses questions dont les suivantes : Comment un système de gestion d'exceptions peut être combiné à la réplication tout en conservant la transparence de celle-ci ? Que se passe-t-il lorsqu'un agent répliqué signale une exception ? A quel niveau le système de réplication doit-il prendre le contrôle lorsqu'une exception est signalée par une réplique ? Quelles sont les types d'exceptions qui peuvent être signalées ? Peut-on ignorer des exceptions signalées par une réplique tant que les autres n'ont pas répondu ?

Nous répondrons à toutes ces questions dans ce mémoire qui s'organise en cinq chapitres. Le premier chapitre présente tout d'abord le contexte de notre travail. Nous dresserons ensuite un état de l'art sur les systèmes de gestion d'exceptions existants dans le deuxième chapitre de ce rapport. Nous présenterons dans le troisième chapitre notre système de gestion d'exceptions du point de vue du programmeur d'applications SMA. Le quatrième chapitre portera sur les approches et solutions possibles pour combiner la gestion des exceptions et la réplication. Le cinquième chapitre présentera enfin l'implémentation de notre système de gestion des exceptions dans DIMAX. Nous conclurons ce rapport par un rappel de la contribution de ce travail de recherche et des discussions sur les perspectives.

Chapitre 1

Contexte général

Contents

1.1 Introduction

Les systèmes distribués tels que les systèmes multi-agents ont fait l'objet de plusieurs recherches, plusieurs mécanismes ont été mis en place pour garantir leur fiabilité. C'est dans cette optique que Darx [17] a été intégré à Dima [11] formant ainsi un système multi-agents avec réplication appelé Dimax [8]. Dans ce chapitre, nous commencerons par présenter le modèle d'agent dans Dima et nous présenterons ensuite le mécanisme de réplication de Darx. La troisième section de ce chapitre présente enfin Dimax.

1.2 Modèle d'agent Dima

Les agents sont des entités actives, autonomes et capables de mettre en oeuvre certains comportements pour répondre à d'autres agents. Il est possible de distinguer deux catégories d'agents :

- Les agents cognitifs : ces agents sont essentillement utilisés dans les sciences cognitives. Ils ont des capacités d'apprentissage et des connaissances qu'ils acquièrent en communiquant avec d'autres agents.

– Les agents réactifs : contrairement aux agents cognitifs, les agents réactifs sont des agents dont les connaissances ne sont pas modifiables mais ont des objectifs précis et des comportements spécifiques dans le cas de l'occurence d'actions inattendues dans leur environnement.

Les agents utilisés dans DIMA sont des agents réactifs. Ce système définit à chaque agent un méta-comportement qui peut contrôler, explicitement et dynamiquement son comportement et son architecture [6]. Le modèle d'agent dans DIMA est représenté par le diagramme de classe de la figure1.1. Chaque agent possède un nom unique (*agentId*) et est encapsulé dans un thread. Les agents de DIMA possèdent également une boîte aux lettres (représentée par la classe `MailBox`) et sont une instance de la classe `BasicCommunicatingAgent`. Leurs comportements sont contrôlés par la classe `AgentManager`. Les agents

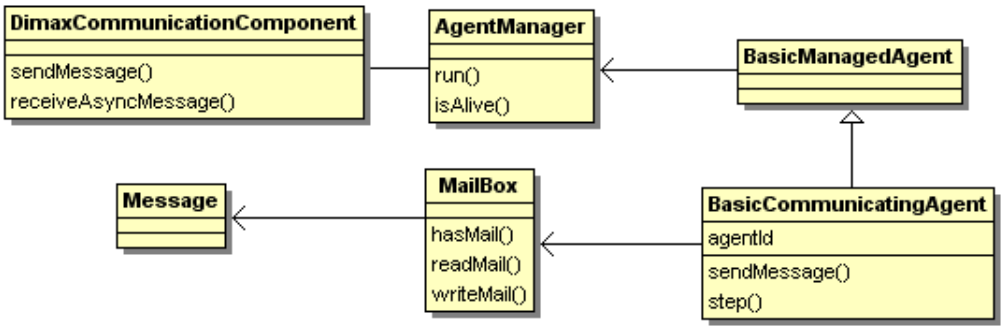

FIGURE 1.1 – Modèle d'agents dans DIMA

communiquent par envoi de messages, leurs identifiants leur permettent de s'envoyer des messages en précisant le destinataire et l'emetteur du message. Lorsqu'un agent A reçoit le message d'un agent B, la méthode *run()* de la classe `AgentManager` est exécutée. Celle-ci détecte la présence d'un nouveau message dans la boîte aux lettres de l'agent B. En effet, cette méthode implémente une boucle infinie exécutée tant que l'agent est en vie (méthode *isAlive()*) et appelle la méhode *step()* de la classe `BasicCommunicatingAgent` à chaque itération. La méthode *step()* décide alors du traitement à appliquer lorsqu'un message est reçu. Le modèle d'agent appliqué se base sur le contrat *Requête/Réponse* qui impose à un agent qui a reçu une requête de retourner une réponse. Dans le cas contraire, une exception `TimeOutException` est signalée.

1.3 Framework de réplication DARX

DARX (Dynamic Agent Replication eXtension) est un framework assurant la tolérance aux fautes dans les systèmes distribués et ce, en utilisant un mécanisme de réplication implémenté en java-RMI. Il permet de lancer des objets (composants, agents, etc.) sur un réseau de machines et de créer des répliques associées à un objet donné grâce à la méthode *replicateTo()* de la classe `TaskShell` (voir figure 1.2). Ces groupes correspondent à des tâches systèmes représentées par la classe `DarxTask`. Chaque groupe est dirigé par une réplique appelée le leader. Les répliques d'un même agent sont identiques et ont le même comportement sauf le leader. La classe `ReplicationPolicy` contient toutes les informations relatives à un groupe de réplication.

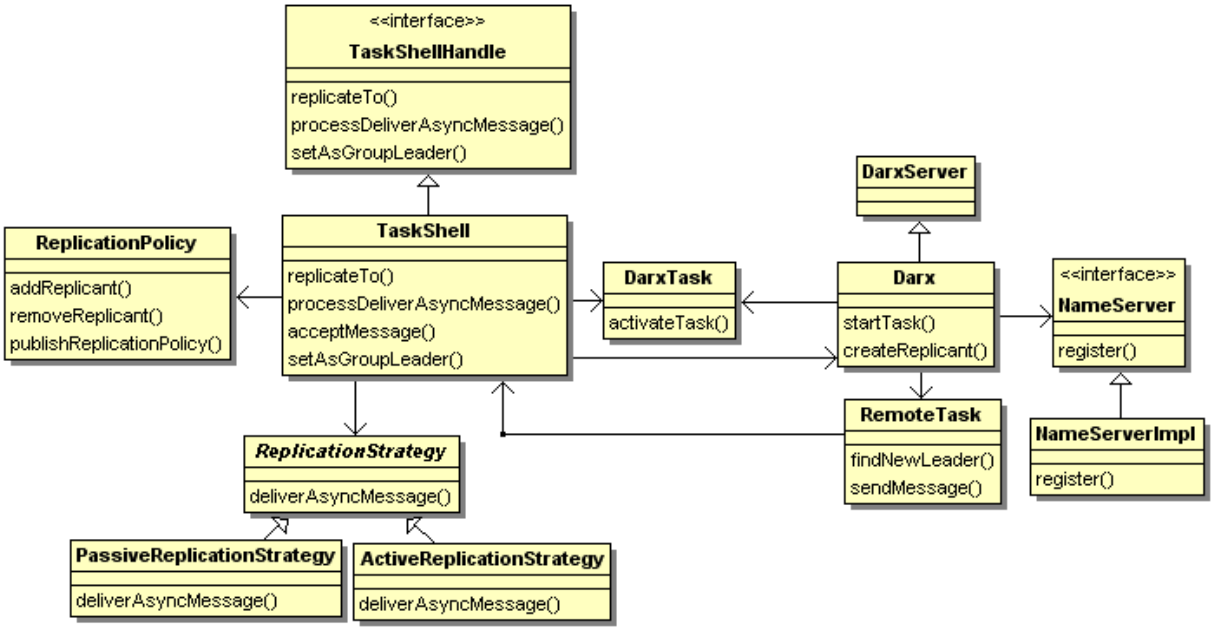

FIGURE 1.2 – Extrait du diagramme de classe de DARX

DARX assure la communication dans un même groupe de réplication. En effet, lorque le leader reçoit un message d'un agent client, il se charge de le transmettre aux différentes répliques de son groupe en utilisant la méthode *deliverAsyncMessage()* de la classe `ReplicationStrategy`. Ces dernières renvoient leurs réponses à l'objet qui a envoyé initialement le message au leader de leur groupe. Etant donné que les répliques sont identiques et

qu'elles sont supposées envoyer les mêmes réponses, l'objet receveur n'accepte que les réponses provenant du leader. Cette fonctionnalité est assurée par le mécanisme de filtrage de messages implémenté dans la méthode *acceptMessage()* de la classe `TaskShell`. DARX dispose de plusieurs services tel que le service de nommage qui permet à la classe `Naming` de Java-RMI d'accéder à une machine en utilisant un format de type URL. D'autres services sont offerts par cette plate-forme. Le service de détection de défaillance, par exemple, permet de sélectionner une réplique qui remplace le leader en cas de défaillance de celui-ci selon la stratégie appliquée (active/passive). Cette fonctionnalité est assurée par la méthode *findNewLeader()* de la classe `RemoteTask`. DARX permet également de changer de stratégie de réplication dynamiquement. La classe `ReplicationStrategy` représente la stratégie appliquée et contient la liste des répliques d'un objet donné.

1.4 Plate-forme multi-agents avec réplication DIMAX

DIMAX est la fusion de DARX et de DIMA. Le diagramme de classe de la Figure 1.4 montre les classes de DIMAX. Le lien entre DARX et DIMA est assuré par la classe `AgentManager` et le lancement du serveur passe pas la classe `DimaXServer`. L'envoi de message d'un agent à l'autre passe désormais par les couches de DARX pour qu'un agent répliqué puisse transmettre le message qu'il reçoit à ses répliques. DIMAX implémente une couche de monitoring qui permet de contrôler le comportement des agents et de les répliquer si nécessaire. La réplication des agents est très coûteuse, il était donc souhaitable de répliquer uniquement les agents qui ont le plus de probabilité de tomber en panne.

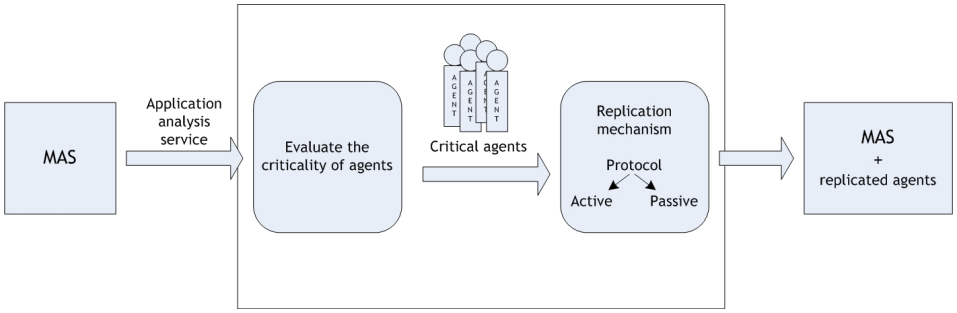

FIGURE 1.3 – Réplication des agents dans DIMAX

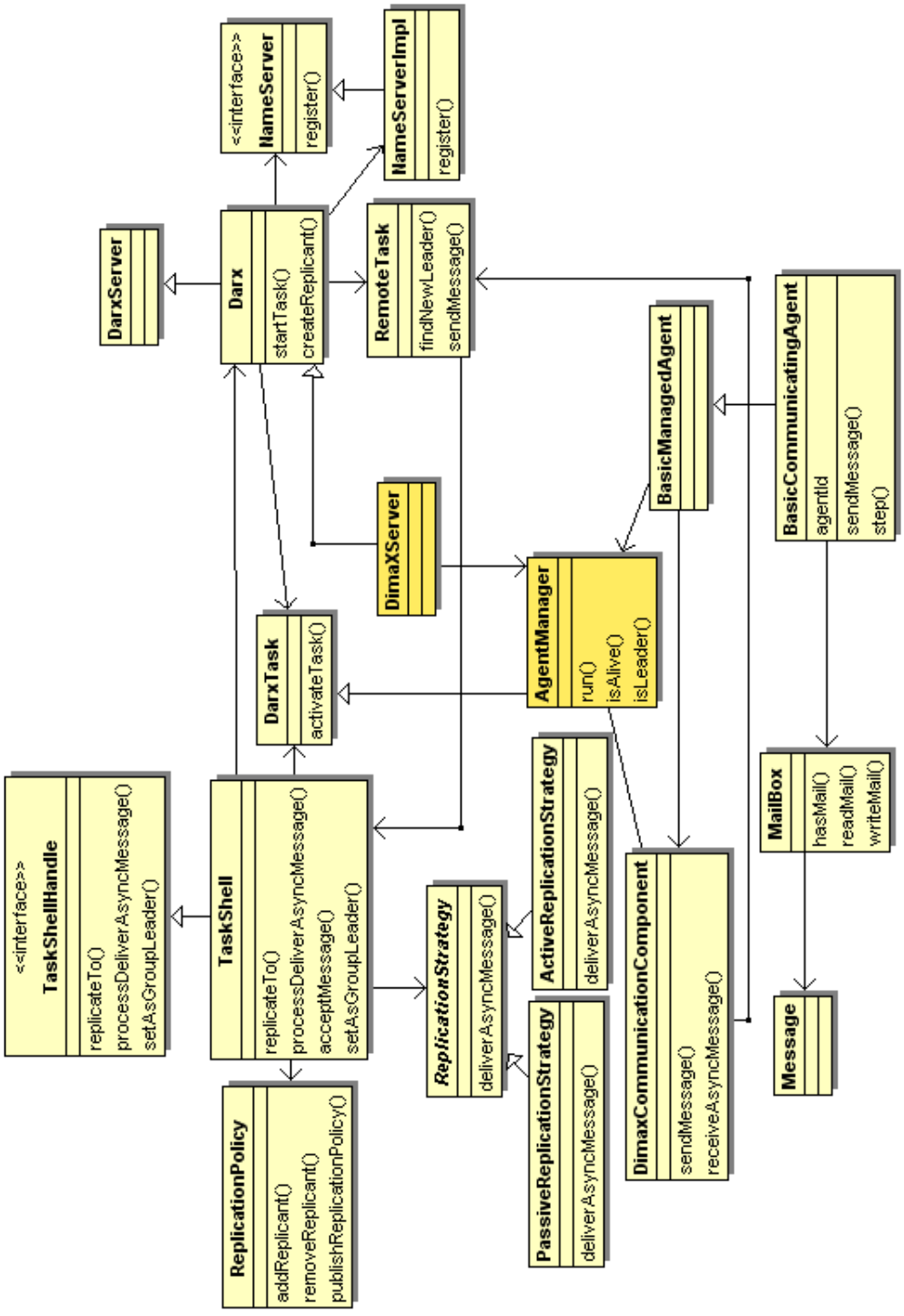

FIGURE 1.4 – Fusion de DARX et de DIMAX

La Figure 1.3 montre les différentes étapes par lesquelles passe le système pour répliquer un agent. A chacune des machines sur lesquelles on souhaite répliquer les agents est associée une probabilité de panne appelée criticité. Un agent tombe en panne si toutes les machines sur lesquelles s'exécutent ses répliques sont en panne. Le calcul de la criticité est décrit en détail dans [3] [7] [12]. Après avoir déterminé la criticité des agents, seuls les agents dont la criticité est élevée sont répliqués ce qui permet d'augmenter la disponibilité de ces derniers tout en optimisant l'utilisation des ressources du système. Toutefois, certaines réponses provenant de plusieurs répliques peuvent causer la défaillance du système. En effet, si les répliques d'un agent ne répondent pas au moment adéquat ou signalent une exception inattendue, celà peut engendrer l'arrêt de l'exécution du système. Il est donc nécessaire de gérer les réponses exceptionnelles au sein d'un groupe de réplication.

1.5 Conclusion

Nous avons présenté dans ce chapitre la pate-forme multi-agents DIMAX. En dépit de la capacité de DARX de créer des répliques d'un agent pour assurer la fiabilité de ce système, des réponses inattendues provenant de différentes répliques peuvent être sources de problèmes. C'est pour cela que nous avons choisi d'améliorer le mécanisme de réplication utilisé dans DIMAX en intégrant un système de gestion d'exceptions que nous présenterons dans le chapitre 3. Le chapitre suivant dresse un état de l'art des travaux déjà effectués dans ce domaine.

Chapitre 2

Etat de l'art

Contents

2.1 Introduction

La tolérance aux fautes dans un système peut se définir comme une propriété qui garantirait qu'une application ne se terminera pas brutalement suite à un signalement d'exception. Un SGE ne garantit pas qu'un système produit sera parfaitement tolérant aux fautes mais il permet aux développeurs d'oeuvrer en ce sens en leur donnant des moyens pour la gestion des erreurs empêchant l'exécution standard de celui-ci. Des systèmes de gestion d'exceptions(SGE) [10] [4] ont été implémentés dans différents langages depuis le début des années 70 dans le but de capturer et de traiter les exceptions signalées. Les SGEs se basent sur des blocs de traitement d'erreurs [1] appelés également des continuations exceptionnelles (handlers) pour rattraper ces exceptions [19]. L'un des domaines qui ont suscité l'intérêt des chercheurs plus récemment est la gestion des exceptions dans les systèmes concurrents. En effet, plusieurs problèmes ont été relevés tel que le signalement d'exceptions concurrentes. Plusieurs SGEs ont été proposés pour pallier à ces problèmes et nous présentons les principaux récents dans ce chapitre.

2.2 Systèmes de Gestion d'Exceptions (SGEs) dans les systèmes concurrents

Les systèmes concurrents sont des systèmes composés par des entités indépendantes éventuellement placées sur différentes machines interconnectées en cas de distribution[1]. Ces entités peuvent être des objets, des agents, etc. et communiquent par envoi de messages asynchrones. Lorsqu'un processus signale une exception, celle-ci va affecter le comportement des autres processus. Le principal problème posé dans ces systèmes est la gestion des exceptions concurrentes. Les modèles suivants proposent des soutions à ce problème.

Atomic Actions

La solution proposée dans [20] consiste à utiliser des *Coordinated Atomic Actions* (CAA). Les *Atomic Actions* ont été définies dans [15] comme étant des groupes d'objets n'intéragissant pas avec le reste du système pendant leur activité. Les CAA se basent sur le concept d'*Atomic Action* et présentent une technique générale pout traîter les exceptions dans les systèmes distribués. Lorsqu'un processus signale une exception, elle est propagée aux autres participants de la CAA. Ces derniers la traîtent dans leurs handlers respectifs et la CAA synchronise leurs états et termine l'exécution normalement de sorte que les processus externes ne soient pas affectés. Lorsque plusieurs exceptions sont signalées simultanément, une exception concertée [14] qui couvre toutes les exceptions signalées est générée comme le montre la figure 2.1. Chaque participant de la CAA rattrape alors cette exception dans son propre handler et signale sa terminaison à celle-ci. A la fin de l'exécution de tous les participants, la *CAA* signale sa terminaison en cas de terminaison avec succès ou signale une exception globale en cas d'échec.

Guardian Model

[18] a proposé un modèle basé sur les *Guardians*. Un *Guardian* est un objet global à la CAA décrite précédemment. Ce modèle sépare les exceptions globales des exceptions locales (qui peuvent être traîtées à l'intérieur d'un processus donné) et utilise la notion de contexte ; le contexte est un nom symbolique donné par le programmeur qui représente une phase d'exécution d'un processus. Ce modèle définit également un ordre de priorité pour

1. Notre système supporte la distribution via les middlewares sur lesquels il est implémenté. La distribution est ici automatique du point de vue du SGE.

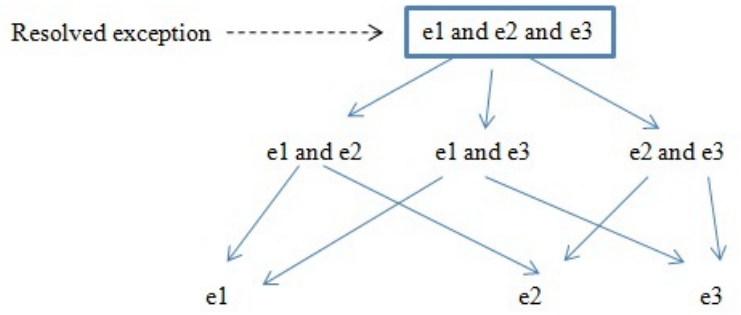

FIGURE 2.1 – Graphe d'exceptions repris de [20]

chaque type d'exception et un ensemble de règles qui doivent être appliquées lorsqu'une exception est signalée. Deux types de règles ont été définies :

- des règles à appliquer lorsqu'une seule exception est signalée (des règles pour les exceptions séquentielles).
- des règles à appliquer lors de l'occurence de plusieurs exceptions (des règles pour les exceptions concurrentes).

Lorsqu'une exception est signalée par un processus, la liste des contextes de l'exception permet de déterminer à quel niveau l'exception doit être traitée dans le processus appelant. Lors de l'occurence d'exceptions globales, le *Guardian* suspend l'exécution de tous ses participants et collecte toutes les exceptions concurrentes. L'ensemble d'exceptions signalées est ensuite divisé en sous-ensembles. Les règles définies précédemment permettent de déterminer quelle est l'exception globale qui doit être levée dans chaque sous-ensemble. A ce niveau, une liste contenant des exceptions séquentielles est obtenue et chaque exception de cette liste est traitée suivant les règles déjà définies.

SaGE

Le SGE proposé dans [5] traîte le problème de communication entre objets asynchrones et se base sur le protocole d'intéraction *Requête/Réponse*. La requête envoyée par un objet appelle le service d'un autre objet. Ce service peut être complexe ou atomique. Les services complexes sont les services qui envoient des messages à leur tour pour invoquer les services d'autres objets. SaGE utilise un arbre d'exécution de services qui enregistre l'ordre d'appel de ces derniers. En d'autres termes, si le service X d'un objet A appelle le service Y d'un objet B, ces services seront représentés dans un arbre où X sera le noeud parent de Y. Lorsqu'une exception est signalée, le handler est tout d'abord recherché localement dans l'objet comme pour les exceptions locales

dans le modèle des *Guardians*. Si aucun handler n'a été défini localement, la recherche de handler est propagée à l'appelant.

Ce dernier est retrouvé grâce à l'arbre d'exécution des services. Supposons que le service X de l'objet A a envoyé plusieurs requêtes et qu'il attend plusieurs réponses. Ce type d'objet est appelé (broadcast group) *bgroup* car il reçoit une requête dans son service et son service la transmet à d'autres objets et attend leurs réponses. Pour pallier au problème de concurrence, SaGE offre la possibilité de définir des fonctions de concertation attachées aux services complexes. Cette fonction est appelée à chaque fois qu'un handler est recherché au niveau de l'appelant donc au niveau du *bgroup*. Cette fonction compare les exceptions reçues et retourne une exception concertée si les exceptions sont jugées critiques pour les autres objets ou ignore l'exception signalée si elle est jugée comme peu critique. Le tableau de la figure 2.2 résume les modèles précédents et montre leurs comportements lors du signalement d'une seule exception ou lors de l'occurence de plusieurs exceptions concurrentes.

Comparaison entre SGEs	SGEs dans les systèmes concurrents		
	Coordinated Atomic Actions	Guardian Model	SaGE
Exception signalée par un processus	Tous les processus traitent la même exception.	Seuls les processus concernés traitent l'exception.	Les processus qui participent à l'arbre d'exécution des services traitent l'exception.
Exceptions concurrentes	Modèle basé sur une hiérarchie des exceptions qui permet de couvrir toutes les exceptions signalées.	Utilisation de règles qui permettent de partir d'un ensemble d'exceptions concurrentes pour traiter à la fin des exceptions séquentielles une par une.	Utilisation d'une fonction de concertation qui retourne une réponse normale ou une exception concertée.

TABLE 2.1 – Comportement des SGEs dans les systèmes concurrents

Pour résumer, nous allons présenter les avantages et les inconvénients de chacun des SGEs présentés dans cette section. Les avantages de ces SGEs sont :

- Dans la *CAA*, les exceptions signalées n'affectent pas le comportement des objets externes grâce à celle-ci.
- La rapidité du traitement d'une exception signalée par un seul processus dans les *Guardians*. Ce modèle reste donc efficace malgé sa complexité.
- SaGE permet de traiter les exceptions critiques pour le système et d'ignorer les exceptions peu-critiques. Il permet également de traiter les exceptions concurrentes d'une façon simple grâce à sa fonction de concertation qui augmente la possibilité de retourner une réponse normale.

Cependant, nous soulignons certaines limites dans ces modèles tels que :

- L'envoi d'un nombre de messages important dans les *CAA* pour informer les participants d'une exception signalée par un processus ce qui empêche ce modèle de couvrir des larges systèmes concurrents.
- Dans [18], les guardians gèrent les participants mais ne possèdent qu'une vue globale du système.
- Dans SaGE, la définition des handlers spécifiques à plusieurs niveaux du programme requiert une bonne connaissance de celui-ci.

Le SGE que nous proposons dans ce rapport se base sur le fonctionnement de SaGE, nous expliquerons nos choix dans le chapitre suivant.

2.3 Combinaison de Gestion d'Exceptions (GE) et de mécanismes alternatifs de tolérance aux fautes

En premier lieu, des mécanismes alternatifs à la GE ont été développés pour la tolérance aux pannes depuis les années 70 et en particulier, des mécanismes basés sur la redondance : N-versions [2] et systèmes divers de réplication [13] [7] [16]. Dans [2], les versions d'un programme sont des modules indépendants (représentés par des classes) développés par plusieurs programmeurs et sont conformes à un même type (les classes des versions peuvent implémenter la même interface ou la même classe abstraite). Lors de l'exécution du programme, toutes les versions reçoivent les mêmes requêtes et fournissent des résultats indépendants. Ces résultats sont collectés par un contrôleur qui les transmet à la fin de l'exécution de toutes les versions à l'*adjudicator* qui lui retourne la réponse envoyée par la majorité des versions (les réponses différentes sont considérées comme des réponses erronées). Le

contrôleur transmet alors cette réponse à l'appelant comme le montre la figure 2.2.
La gestion des exceptions a été intégrée aux N-versions [2] pour gérer leurs

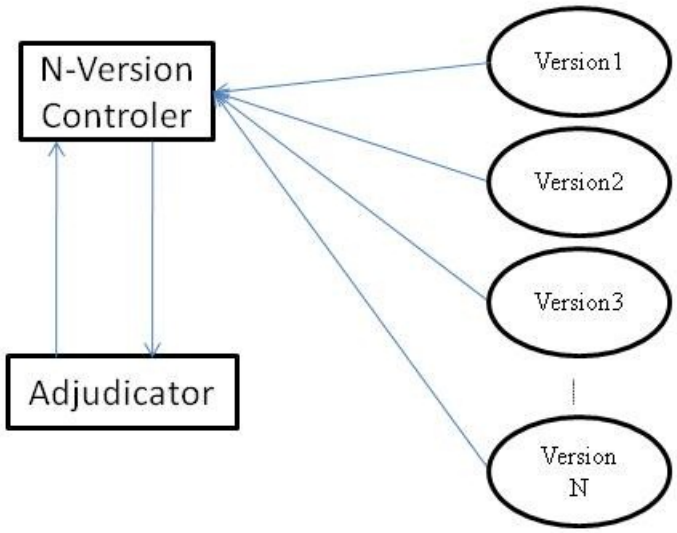

FIGURE 2.2 – N-Versions : Principe

réponses exceptionnelles et tolérer le plus de fautes possibles. Pour ce faire, les exceptions ont tout d'abord été classées en deux types :

- Exceptions internes : ces exceptions sont définies par le développeur de chaque version. Un handler pour chacune de ces exceptions est défini dans la même classe.
- Exceptions externes : ces exceptions sont identiques pour toutes les versions. Elles sont définies sur l'interface implémentée par toutes versions et sont signalées si la version ne peut pas fournir un service qui lui a été demandé ou si le handler d'une exception interne signale une exception externe.

Lorsqu'une exception interne est signalée, elle est traitée dans le handler local défini dans la version et son exécution se termine normalement d'une façon complètement transparente pour les autres versions.
Si le handler de la version signale une exception externe, cette dernière est signalée au contrôleur des versions toujours de façon transparente (l'exception n'est pas propagée directement à l'appelant de la version qui a signalé l'exception car des réponses normales peuvent être envoyées par les autres

versions). Après la fin de l'exécution de toutes les versions, l'*adjudicator* devra décider quelle réponse doit être retournée :

- Si la majorité des versions a envoyé une réponse normale alors celle-ci est retournée au contrôleur.
- Si la majorité des versions a envoyé une réponse exceptionnelle, cette exception est transmise par l'*adjudicator* au contrôleur qui la signale à l'appelant.
- S'il n'y a pas de réponse majoritaire, une exception prédéfinie dans le contrôleur est signalée.

D'autres mécanismes comme la réplication ont été utilisés pour garantir la tolérance aux fautes. [16] décrit un mécanisme de réplication avec votes. Chaque noeud du système possède n copies qui lui sont identiques. Si la méthode d'un noeud Ni est appelée par un noeud Nj, toutes les copies de Ni exécutent la même méthode et retournent leurs réponses à Nj. Le mécanisme de vote de ce dernier vérifie l'exactitude des réponses. En cas de réponses exceptionnelles, des handlers prédéfinis par le programmeur traitent ces réponses au niveau de l'appelant (c'est à dire au niveau de Nj) comme le montre la figure 2.3.

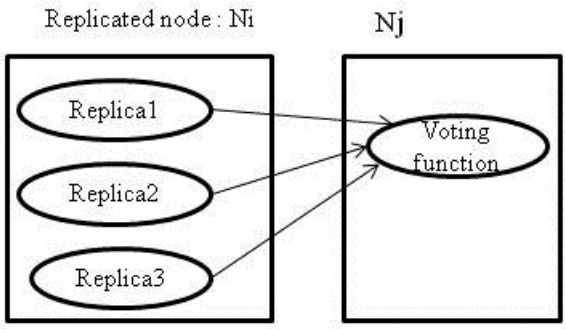

FIGURE 2.3 – Structure du système de vote pour le filtrage des réponses

2.4 Conclusion

Nous avons vu, dans ce chapitre, différents SGEs utilisés pour le traitement des exceptions concurrentes dans les systèmes distribués. Dans ces systèmes, des objets les communiquent par envoi de messages asynchrones. Les systèmes multi-agents sont des systèmes concurrents distribués qui sont confrontés aux problèmes présentés dans cette section. Le SGE que nous proposons dans ce travail : XSaGE se base sur le fonctionnement de SaGE

pour la recherche de handler et le comportement de la fonction de concertation. La classification des exceptions dans le modèle des *Guardians* et dans les N-Versions a influencé nos choix pour la gestion des exceptions dans un environnement avec réplication. Le chapitre suivant présente XSaGE et les différentes stratégies que nous avons utilisées pour gérer les exceptions dans un environnement avec réplication.

Chapitre 3

Spécification de XSaGE : SGE du point de vue du programmeur

Contents

3.1 Introduction

Les agents de DIMAX que nous utilisons dans notre système communiquent par envoi de messages. Ces derniers encapsulent des services que les agents destinataires exécutent. Cependant, une exception peut surgir lors de l'exécution du service d'un agent. Il est alors indispensable de rattraper cette exception dans un handler défini à cet effet. Dans ce chapitre, nous allons présenter XSaGE : le SGE que nous proposons d'intégrer à DIMAX. Cette présentation est faite selon le point de vue du programmeur, cela signifie que nous présentons les structures de contrôle que nous ajoutons au langage de programmation pour rattraper et traiter les exceptions signalées par des agents. La réplication étant transparente pour le développeur, nous présenterons le processus de recherche de handler, qui doit prendre la réplication en compte, dans le chapitre suivant.

3.2 Définition des handlers avec XSaGE

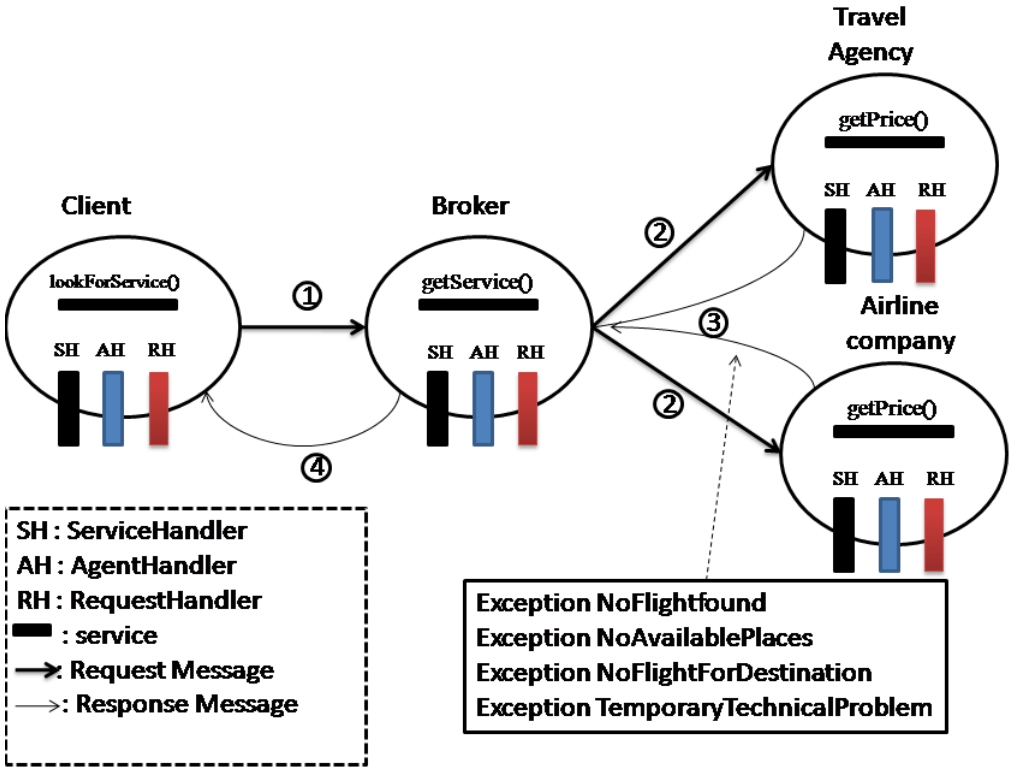

FIGURE 3.1 – Exceptions signalées et Handlers définis dans un exemple de commerce électronique.

XSaGE offre la possibilité de définir des handlers pour rattraper les exceptions signalées par des agents. Nous avons choisi d'utiliser un exemple simple de commerce électronique pour présenter notre SGE ainsi que la définition des handlers. Dans cet exemple, un agent appelé `Client` envoie initialement sa requête à un agent appelé `Broker` (étape 1 de la figure 3.1). Ce dernier transmet ensuite la requête du `Client` à une agence de voyage (agent `TravelAgency`) et à une compagnie aérienne (agent `AirlineCompany`) qui exécutent le message. La communication entre les agents se base sur le contrat *Requête/Réponse*. Ce contrat impose à un agent qui a reçu une requête de retourner une réponse. Il est alors important de souligner le rôle des services, des requêtes et des agents. Néanmoins, des exceptions peuvent surgir lorsque les agents exécutent leurs messages si, par exemple, aucun vol n'a été trouvé ou si un problème technique apparaît. Nous avons alors pro-

posé dans notre système une méthode simple pour attacher des handlers à des expressions et rattraper ces exceptions.

```
1 public class Broker extends SaGEManagerAgent{
2 ...
3 // service provided by the Broker agent
4 @service public void getService (Destination destination,
     Date date){
5   sendMessage(companies, new SageRequestMessage("getPrice",
       destination,date,"getService") {
6   @requestHandler public void handle(Exception e){
7     retry();
8  }
9 });
10 }
11 //handler associated to getService service
12 @serviceHandler(serviceName="getService")
13 public void badParametersService(noAvailablePlaces e){
14   date = date.nextDay();
15   retry();
16 }
17 //handler associated to getService service
18 @serviceHandler(serviceName="getService")
19  public void handle(TemporaryTechnicalProblem e){
20    wait(120);
21    retry();
22 }
23 //handler associated to Broker Agent
24 @agentHandler
25  public void handle(noFlightForDestination e){
26   signal(e);
27  }
```

Listing 3.1 – Handlers et annotations

Tout d'abord, les handlers peuvent être attachés à des requêtes. Cela permet de spécifier le traitement à appliquer dans le cas où le service de l'agent qui a reçu la requête retourne une exception. Cette exception peut être dûe, par exemple, à des invocations simultanées d'un même service. L'absence de handlers pour rattraper l'exception signalée par ce service à des niveaux plus bas du programme, engendre la réception d'une réponse exceptionnelle par la requête qui a appelé initialement ce service. Il est alors important d'attacher des handlers à des requêtes. Les handlers peuvent aussi être attachés à des services. Cela permet rattraper les exceptions signalées par leurs exécutions.

Les handlers peuvent également être attachés aux agents fournissant ainsi des composants avec des comportements par défaut pour la gestion des exceptions. Ces handlers peuvent rattraper les exceptions communes à tous les services d'un même agent. Pour résumer les handlers peuvent être attachés à des services, à des agents ou à des requêtes et ce grâce à des annotations Java que nous détaillerons dans ce qui suit. Ces handlers sont représentés par les cadres noirs de la figure 3.1.

Le listing 3.1 montre un extrait du code source de l'agent `Broker`. Le service *getService()* est défini grâce à l'annotation (*@service*) dans les lignes 4-10. Il est appelé lorsque le `Broker` reçoit la requête du `Client` et utilise la méthode *sendMessage()* pour transmettre cette requête aux autres agents comme nous l'avons montré dans la figure 3.1. Lorsque le programmeur définit un handler attaché à un service X, il précise le type de l'exception qu'il souhaite rattraper. Il est ainsi possible d'attacher plusieurs handlers à un même service selon le type de l'exception signalée. Par exemple, des handlers, attachés à *getService()*, ont été définis pour rattraper les exceptions `TemporaryTechnicalProblem` et `NoFlightAvailablePlaces` et ce, grâce à l'annotation *@serviceHandler* (lignes 12-22). Des handlers qui peuvent rattraper des exceptions communes à tous les services peuvent être attachés à l'agent grâce à l'annotation *@agentHandler*. Les lignes 24-27 montrent un handler attaché à l'agent `Broker`. Ce handler a été défini pour rattraper l'exception `noFlightForDestination`. On peut également attacher des handlers aux requêtes avec l'annotation *@requestHandler* (lignes 6-8).

Nous avons montré dans cette section comment on peut définir des handlers attachés à des expressions en utilisant les annotations Java. Nous allons présenter dans la section suivante comment ces handlers peuvent rattraper les exceptions signalées.

3.3 Spécification du comportement des handlers et de la fonction de concertation

Chaque handler définit un comportement spécifique à l'exception qu'il doit traiter dans le but de réduire la possibilité que celle-ci affecte le comportement des autres agents et d'augmenter les possibilités de retourner une réponse normale à l'agent client. Un handler peut :

- Retourner une nouvelle valeur qui devient la valeur de l'expression à laquelle il est associé. Par exemple, le handler peut modifier la valeur

de la variable qui était à l'origine du signalement de l'exception dans le service auquel il est associé (lignes 14-15 du listing 3.1)
- Relancer l'exécution standard de différentes façons. Tout d'abord le handler suspend l'exécution du service courant, fixe un délai au bout duquel la faute peut être réparée (comme pour les problèmes techniques par exemple) et relance enfin l'exécution avec la méthode *retry()* (lignes 20-21).
- Signaler une nouvelle exception de plus haut niveau conceptuel ou re-signaler la même exception avec la méthode signal(e) (lignes 25-27). Etant donné qu'il n'existe pas de handlers attachés aux handler, cette exception sera alors rattrapée à d'autres niveaux du programme.
- Ignorer l'exception signalée en affichant juste un message comme pour les *warnings* par exemple.

Pour résoudre le problème de concurrence dû à la réception de plusieurs requêtes simultanées provenant de plusieurs agents, des fonctions de concertation attachées aux services peuvent être définies. La fonction de concertation est exécutée à chaque fois qu'un handler est recherché au niveau du service. Il n'existe pas de fonction de concertation attachées aux requêtes car les requêtes sont atomiques. Il est également inutile de définir des fonctions de concertation attachées aux agents. En effet, ces derniers communiquent pas envoi de messages encapsulant des services. Dans ce cas, le problème de concurrence est déjà géré par les fonctions de concertation attachées à leurs services.

```
1 // resolution function associated to the getService service
2 @serviceResolutionFunction(servicename="getService") public
      TooManyProvidersException concert ()
3 {
4     int failed = 0;
5     for (int i=0; j<subServicesInfo.size(); i++)
6         if ((ServiceInfo) (subServicesInfo.elementAt(i)).
              getRaisedException() != null)  failed++;
7
8    if (failed > 0.3*subServicesInfo.size())
9               return new TooManyProvidersException(
                    numberOfProviders);
10        return null;
11 }
```

Listing 3.2 – Fonction de concertation attachée au service du `Broker`

Ces fonctions sont utilisées dans plusieurs SGEs qui traitent le problème de concurrence tel que [14].
Comme pour les handlers, nous pouvons attacher les fonctions de concertation aux services grâce à l'annotation *@serviceResolutionFunction* comme le montre le listing 3.2. La fonction de concertation définie dans cet exemple collecte toutes les réponses exceptionnelles provenant des autres agents et compare à chaque fois le nombre d'exceptions signalées au nombre de réponses reçues. Si le nombre d'exceptions signalées dépasse le tier du nombre d'exceptions reçues, une exception concertée est alors signalée.

3.4 Conclusion

Nous avons présenté dans ce chapitre la spécification de la partie de XSaGE utilisable par le programmeur, c'est à dire l'ensemble des structures de contrôle pour la gestion des exceptions. Nous avons présenté également une fonction de concertation qui permet de gérer les problèmes de concurrence. Cette fonction est l'élément clé de notre stratégie de gestion d'exceptions dans un système multi-agents avec réplication. Cette stratégie fera l'objet du chapitre suivant.

Chapitre 4

Spécification de l'intégration d'un SGE dans un langage d'agents avec Réplication

Contents

4.1 Introduction

La réplication est un mécanisme qui se base sur la redondance pour assurer la fiabilité des systèmes multi-agents. Comme précisé précédemment, chaque agent, selon sa criticité, possède un nombre déterminé de répliques. Lorsqu'une exception est signalée par un agent (et ses répliques si ce dernier est répliqué), une recherche de handler est lancée pour traiter cette exception. Cependant, si plusieurs répliques signalent une ou plusieurs exceptions, il faut les traiter avant de les envoyer à l'agent client. Dans ce chapitre, nous allons présenter tout d'abord le mécanisme d'envoi de messages dans DARX en tenant compte des exceptions qui peuvent être signalées ainsi que les solutions que nous proposons pour traiter ces messages. Nous présenterons par la suite les étapes de recherche de handler ainsi que les différents types d'exceptions pouvant être signalées par les répliques et l'algorithme de signalement

complet incluant la prise en compte de la réplication.

4.2 Gestion des exceptions signalées par les répliques des agents

Nous avons présenté dans la section précédente l'exemple de commerce électronique `Search For Flight`. Supposons maintenant que le calcul de la valeur de la criticité des agents `TravelAgency` et `AirlineCompany` indique que ces agents doivent être répliqués. Le mécanisme d'envoi de messages utilisé dans DARX permet de transmettre les réponses envoyées par les répliques d'un agent à l'agent client (celui qui a envoyé initialement la requête). Supposons que ces répliques signalent plusieurs exceptions, l'agent client va alors recevoir plusieurs réponses différentes. De quelle façon faut-il alors gérer ces réponses ? Nous allons présenter dans cette section le mécanisme d'envoi de message initialement utilisé dans DARX et la solution que nous proposons pour que le traitement des messages soit cohérent avec notre mécanisme de gestion des exceptions.

4.2.1 Analyse du mécanisme d'envoi de messages dans DARX

Lorsqu'un agent A envoie un message à un agent B répliqué, ce dernier diffuse le message qu'il a reçu à ses répliques. Le message est alors traîté et envoyé par les répliques à l'agent destinataire A d'une manière à la fois concurrente et asynchrone. L'agent destinataire va alors recevoir des messages redondants provenant de différentes répliques. Etant donné que les messages sont identifiés par des numéros de série et que les répliques d'un agent ont le même comportement et sont supposées envoyer exactement les mêmes messages, DARX filtre les messages en mémorisant les numéros de série des messages reçus. Si le message reçu est identifié comme redondant, il est ignoré. Ainsi, le traitement des messages envoyés par les répliques est effectué côté client comme le montre la figure 4.1 ce qui n'est pas cohérent avec notre philosophie de traitement des messages car dans le cas où des répliques envoient des messages différents à cause d'une exception signalée par exemple, le client va recevoir des réponses différentes provenant des répliques d'un même agent.

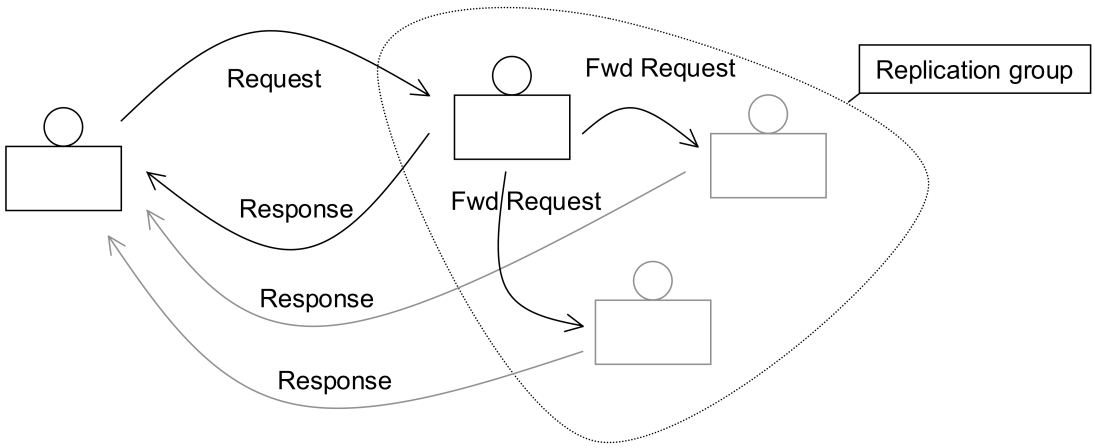

FIGURE 4.1 – Mécanisme d'envoi de messages dans DARX

4.2.2 Solution pour traiter les réponses au niveau du leader

Pour pallier au problème précédent, nous proposons de distinguer les messages de requête (représentés par la classe `SageRequestMessage`) des messages de réponse(représentés par la classe `SageResponseMessage`). Ainsi, les messages de requête seront envoyés directement au client et les messages de réponses seront traités comme suit :

- Si la réponse est envoyée par une réplique qui n'est pas le leader, il faut transmettre cette réponse au leader pour permettre sa collecte et son traitement. Pour ce faire, nous proposons un nouveau type de messages appelé `Shadow Message` qui encapsule la réponse initialement envoyée par la réplique. Les `Shadow Messages` sont représentés par la classe `ShadowMessage` qui est une spécialisation de la classe `SageResponseMessage` (Figure 4.2). L'intérêt de l'utilisation des `Shadow Messages` est de permettre à DARX de reconnaître les réponses envoyées par les répliques et de les envoyer à l'agent qu'elles représentent. Ce message sera alors reçu par le leader de leur groupe de réplication au lieu d'être reçu et filtré par l'agent client. Cela est illustré par la figure 4.3.
- Lorsque la réponse est envoyée par le leader, cette réponse sera encapsulée dans un `SageResponseMessage` et le gestionnaire de réplication qui lui est associé (présenté dans les sections suivantes) sera directement invoqué pour décider de l'envoi d'une réponse normale ou d'une exception.

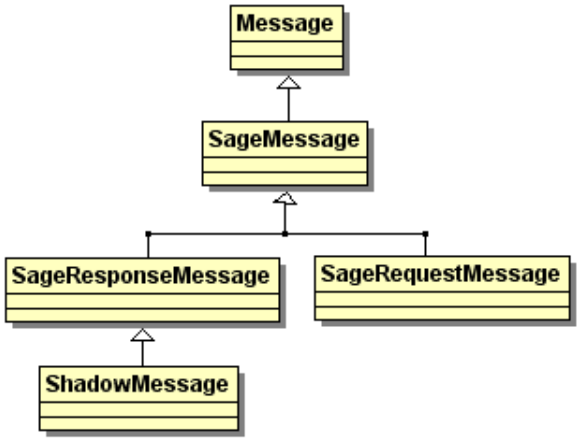

FIGURE 4.2 – Structure des messages dans DIMAX

– Si la réponse est envoyée par un agent non répliqué et si cette réponse est normale, elle sera encapsulée dans un `SageResponseMessage`. Sinon, si l'agent signale une exception, le processus de recherche de handler est lancé comme nous l'avons décrit dans le chapitre précédent.

Nous présentons dans ce qui suit les étapes de recherche de handler ainsi que le gestionnaire de réplication.

4.3 Recherche de handler dans un système multi-agents avec réplication

Dans l'exemple de commerce électronique présenté précédemment, lorsque les leaders des agents `TravelAgency` et `AirlineCompany` reçoivent le message du `Broker`, il le transmettent à leurs répliques respectives. Chaque leader récupère ensuite la réponse de sa réplique sous forme de `Shadow Message` comme le montre la figure 4.4.

Si l'une des répliques signale une exception, à quel niveau la recherche de handler doit-elle être effectuée ? Nous allons présenter dans cette section les différentes étapes de recherche de handler lors du signalement d'une exception par une réplique.

4.3.1 Etapes de recherche de handler

Les messages envoyés par les agents contiennent les noms des services qui seront exécutés par les agents destinataires. Les services exécutés sont repré-

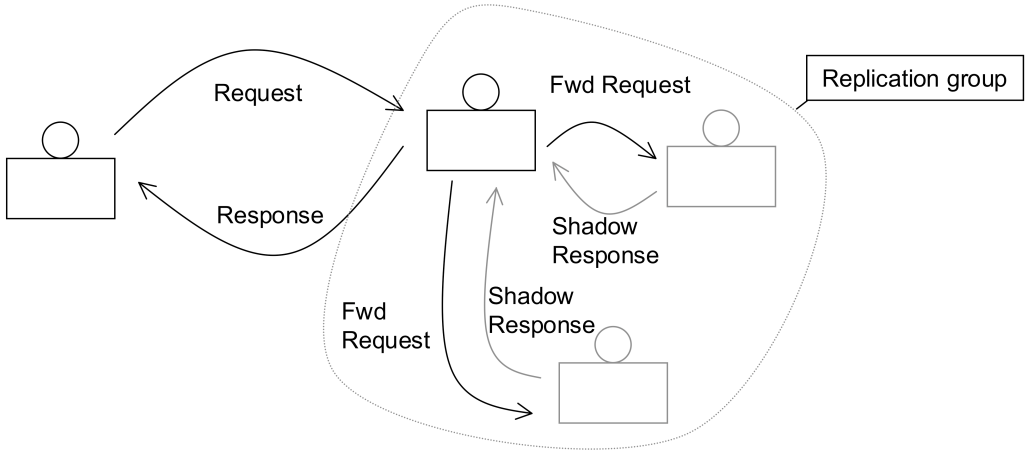

FIGURE 4.3 – Redirection des messages envoyés par les répliques à leur leader

sentés par des noeuds contenant leur nom ainsi que l'identifiant de l'agent où ils ont été définis. Si le service X d'un agent A envoie une requête au service Y d'un agent B, le noeud de X sera défini comme le noeud parent du noeud de Y et la requête de X sera enregistrée dans ce noeud parent. Pour résumer, chaque noeud peut avoir un noeud parent qui relie son service au service de l'agent appelant. Les liens entre ces services sont utilisés pour la recherche de handler.

La figure 4.5 représente les différentes étapes de recherche de handler. Lorsqu'une exception est signalée par le service d'un agent (leader ou non), l'exécution de ce service est suspendue et le processus de recherche de handler est lancé au niveau des fonctions de conceration associées à ce service et au niveau des handlers qui lui sont attachés. Si une fonction de concertation associée au service a été trouvée dans la classe de l'agent (comme dans le listing 3.2), elle est exécutée et l'une des réponses suivantes est envoyée :

- Un objet de type Exception, si l'exception est considérée comme critique pour le service.
- Un objet de valeur nulle, si l'exception est considérée comme peu-critique pour le service.
- Une objet de type Exception, si l'exception est peu-critique mais a été signalée par plusieurs répliques.

Si aucune fonction de concertation associée au service n'a été définie, la recherche est effectuée au niveau des handlers attachés au service (définis par `@serviceHandler` - voir chapitre 3 section 2). A ce niveau, si un handler a été trouvé, il est exécuté et son exécution arrête le service et si aucun hand-

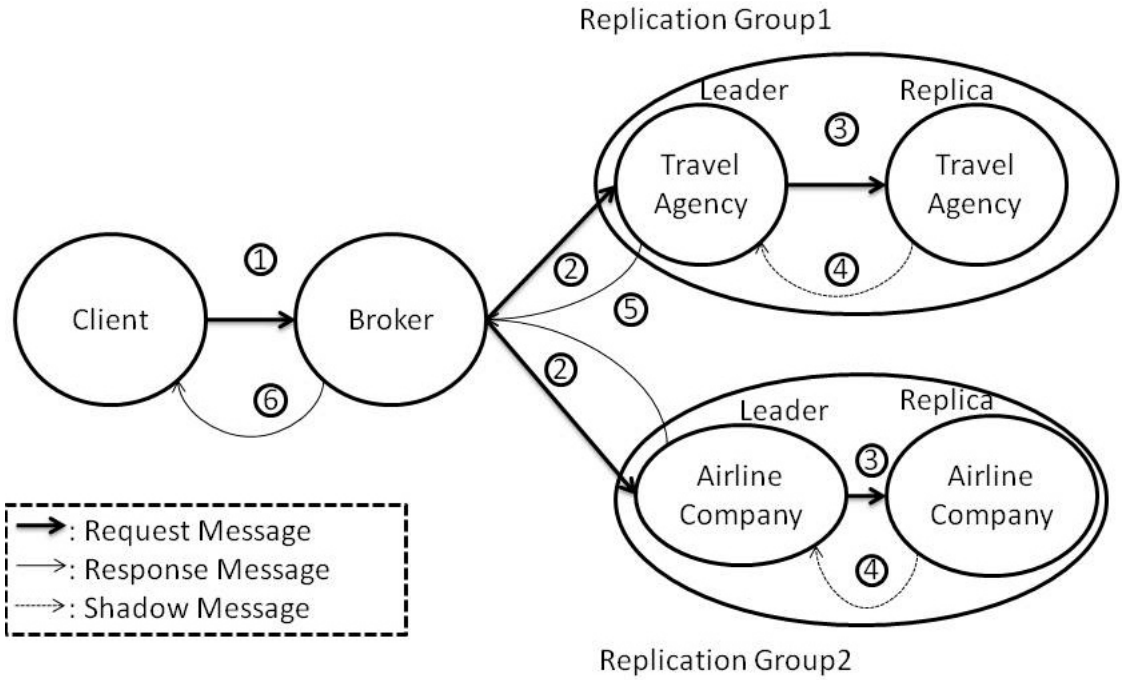

FIGURE 4.4 – Exemple avec agents répliqués

ler adéquat n'a été trouvé, la recherche de handler est effectuée au niveau de l'agent. Le traitement de l'exception est effectué de la même façon qu'à l'étape précédente, si un handler attaché à l'agent a été trouvé. Supposons maintenant qu'aucun handler n'a été trouvé et que l'agent soit répliqué, le contrôle est alors passé au gestionnaire de réplication (GR). Chaque agent possède son propre gestionnaire de réplication. Ce dernier utilise une fonction de conceration, dont le comportement est différent de celle définie précédemment, dans le but de coordiner les différentes réponses reçues par plusieurs répliques et d'envoyer enfin une seule réponse à l'agent client. Nous allons définir le comportement du gestionnaire de réplication dans la section suivante.

4.3.2 Gestionnaire de réplication

Lorsque la réplique d'un agent signale une exception, la recherche de handler passe par les étapes décrites dans la section 4.3.1 et si aucun handler n'a été trouvé, la recherche est effectuée au niveau du gestionnaire de réplication associé au leader de cet agent. La fonction de conceration définie au sein du gestionnaire de réplication décide, selon le **type de l'exception** signa-

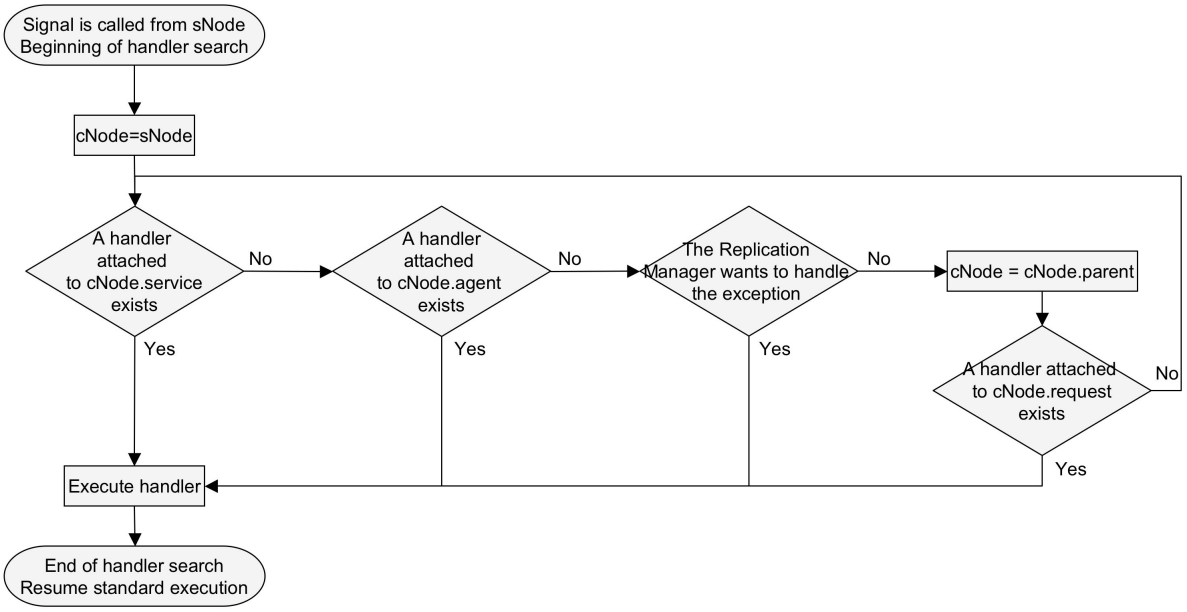

FIGURE 4.5 – Etapes de recherche de handlers

lée, de propager ou d'ignorer cette exception. Dans cette section, nous allons commencer par présenter nos choix de classification des types d'exceptions qui peuvent être signalées par les répliques d'un agent et nous présenterons ensuite le fonctionnement du gestionnaire de réplication.

Classification des exceptions selon leurs types

Si la même exception est signalée par toutes les répliques d'un agent, cela va diminuer les performances du système car le signalement et le traitement de l'exception par chaque réplique va nécessiter un temps d'exécution additionnel. C'est pour cela que nous avons choisi de classer les exceptions en exceptions internes (`ReplicaSpecific`) et en exceptions globales (**ReplicaIndependant**). Nous pourrons ainsi reconnaître directement les exceptions qui seront signalées par toutes les répliques et les exceptions qui ne seront signalées que par une ou quelques répliques, ce qui rend notre mécanisme à la fois curatif et préventif. On distingue alors deux catégories d'exceptions :

- Les exceptions de type `ReplicaIndependant` : Ces exceptions sont signalées par toutes les répliques d'un agent donné car elles résultent d'un problème commun à celles-ci comme par exemple un mauvais paramètre passé à la requête que l'agent a reçu. En général, nous considérons les

exceptions métier comme des exceptions de type `ReplicaIndependant`.

- Les exceptions de type `ReplicaSpecific` : Ces exceptions sont relatives à l'exécution de la réplique et n'affectent pas le groupe de réplication ni les autres agents. Elles peuvent être signalées suite à un problème technique rencontré par la réplique d'un agent par exemple. Les autres répliques de l'agent ne signalent pas forcément la même exception vu qu'elles sont exécutées dans des contextes différents sur des machines distantes.

Fonctionnement du gestionnaire de réplication

Le gestionnaire de réplication a pour fonction de gérer les différentes réponses envoyées par les répliques d'un agent donné. Ces réponses, qu'elles soient normales ou exceptionnelles, sont traîtées dans la fonction de concertation associée au leader du groupe de réplication qui, grâce à son handler, assure la gestion des réponses exceptionnelles.

Pour traîter ces réponses, plusieurs approches sont possibles :

- **Approche Pessimiste** : Dans cette approche, la fonction de concertation du leader ne renvoit une réponse qu'après avoir reçu toutes les réponses des répliques de l'agent. Si l'une des répliques signale une exception de type `ReplicaSpecific`, l'exception est enregistrée et la fonction de concertation attend les réponses des autres répliques. Supposons maintenant que l'une des répliques signale une exception de type `ReplicaIndependant`. L'exception est enregistrée avec les exceptions de type `ReplicaSpecific` en attendant les autres réponses des répliques. Si une réponse normale est envoyée par l'une des répliques, celle-ci est enregistrée avec les exceptions. A la fin de l'exécution de toutes les répliques, les différentes réponses envoyées par celles-ci sont comparées. Si la majorité des réponses reçues sont normales, alors une réponse normale est retournée à l'agent client (celui qui a envoyé initialement la requête). Si une exception de type `ReplicaIndependant` a été signalée, le handler attaché à la fonction de concertation, dans le gestionnaire de réplication, propage l'exception à l'agent appelant et le handler est recherché dans la liste des handlers attachés à la requête qui a invoqué initialement le service courant (le service qui a signalé l'exception). Le handler de l'appelant est retrouvé grâce à l'arbre de l'exécution des services défini dans la section 4.3.1 qui fait le lien entre le service courant et le service appelant. Supposons qu'aucune exception de type `ReplicaIndependant` n'a été signalée et que toutes les exceptions signalées sont de type `ReplicaSpecific`, le handler asso-

cié à la fonction de concertation parcourt alors la liste des exceptions enregistrées pour détecter l'exception la plus signalée et recherche un handler associé à cette exception au niveau de l'appelant du service comme pour les exceptions de type `ReplicaIndependant`.

- **Approche Optimiste** : Dans cette approche, la fonction de concertation du leader est invoquée à chaque fois que la réponse d'une réplique est reçue. Si l'une des répliques signale une exception de type `ReplicaSpecific`, l'exception est enregistrée en attendant les réponses des autres répliques. Si l'une des répliques signale une exception de type `ReplicaIndependant`, le handler associé à la fonction de concertation est directement invoqué, l'exécution de toutes les répliques est arrêtée et une recherche de handler est lancée au niveau du service appelant (le service qui a invoqué le service courant). La figure 4.6 décrit les étapes par lesquelles passe le processus de recherche de handler lorsqu'une exception de type `ReplicaIndependant` est signalée.

Le tableau page 33 résume ces deux approches.

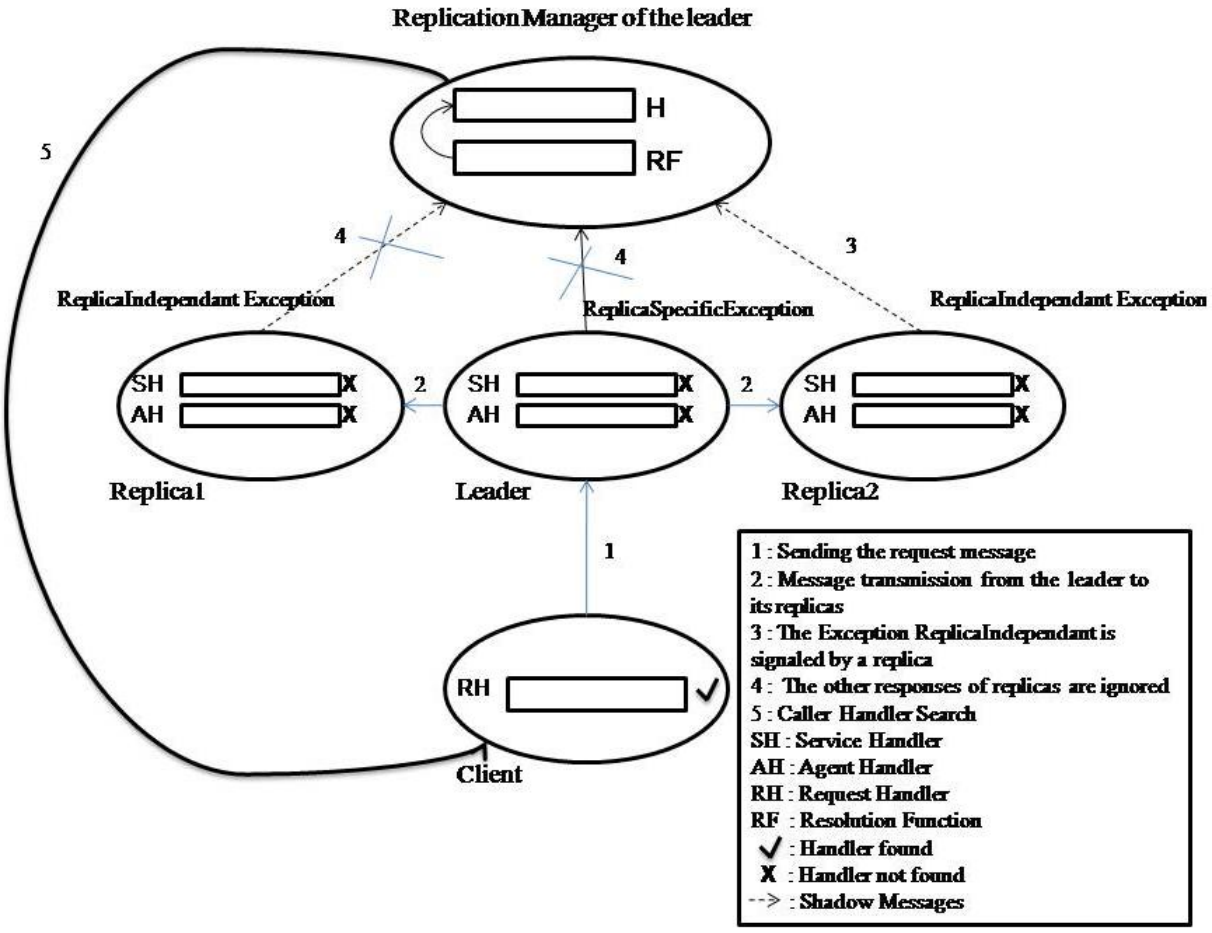

FIGURE 4.6 – Approche optimiste pour la gestion des réponses exceptionnelles de type `ReplicaIndependant`

Réponses des répliques	Approches	
	Approche Optimiste	Approche Pessimiste
Exception : "Replica Specific"	Enregistrement de l'exception et attente des réponses des autres répliques.	Enregistrement de l'exception et attente des réponses des autres répliques.
Exception : "Replica Independant"	Arrêt de l'exécution des autres répliques du groupe de réplication et recherche de handler au niveau de l'appelant.	Enregistrement de l'exception et attente des réponses des autres répliques.
Réponse normale	Envoi immédiat d'une réponse normale et arrêt de l'exécution des autres répliques.	Attente de la fin de l'exécution des autres répliques. A la fin de l'exécution de toutes les répliques, si la majorité des réponses reçues sont normales, une réponse normale est envoyée à l'agent client. Sinon les réponses normales sont ignorées et une exception est signalée.

TABLE 4.1 – Comparaison des approches possibles dans le gestionnaire de réplication

Nous soulignons la rapidité de l'envoi des réponses (normales/exceptionnelles) dans l'approche optimiste et nous remarquons que l'approche pessimiste impose un temps d'exécution plus important étant donné qu'il faut attendre les réponses de toutes les répliques. De plus l'approche pessimiste ne permet pas de retourner une réponse normale si cette dernière n'a pas été envoyée par la majorité des répliques.

4.4 Conclusion

Nous avons présenté au cours de ce chapitre les solutions que nous avons proposé pour améliorer le mécanisme de réplication utilisé dans DIMAX. Ces

solutions nous ont permis d'appliquer notre système à l'exemple de commerce électronique introduit précédemment et que nous allons présenter dans le chapitre suivant.

Chapitre 5

Implémentation et Expérimentation du SGE

Contents

5.1 Introduction

Nous avons défini dans le chapitre précédent les différentes approches qui peuvent être appliquées lorsqu'une réplique signale une exception et lorsque le contrôle est passé au gestionnaire de réplication. L'approche optimiste est l'approche la plus adaptée à notre système en raison de sa dynamicité et de la rapidité d'émission des réponses. Il nous a alors semblé plus adéquat d'utiliser cette approche pour l'implémentation de notre SGE dans DimaX ce qui constitue l'objet de la première section de ce chapitre. La deuxième section présente l'application du SGE implémenté dans DimaX à l'exemple introduit précédemment : `Search for flight`.

5.2 Implémentation du SGE

Les agents de DIMAX sont des agents réactifs qui possèdent des services et des handlers. Afin de les gérer, nous avons créé la classe `SaGEManagerAgent`. Tous les agents de notre système sont définis comme étant des sous-classes

de cette classe comme le montre la figure 5.1. Les annotations `@service`, `@serviceHandler`, `@agentHandler`, `@requestHandler` ont été définies dans des `@interface` Java. La partie claire du diagramme de classe de la figure 5.1 représente les classes de DIMAX décrites dans le chapitre 1 de ce rapport. La partie foncée représente les classes de notre SGE que nous avons intégrées à DIMAX.

Lorsqu'une exception est signalée par le service d'une réplique (méthode *signal(e)* de la classe `SaGEManagerAgent`), l'exécution de ce service est suspendue et le processus de recherche de handler est lancé grâce à la méthode *localHandlerSearch(e)*. La recherche de handler est tout d'abord effectuée au niveau du service et se poursuit par la suite au niveau de l'agent si aucun handler associé au service n'a été trouvé comme nous l'avons décrit précédemment dans le chapitre 3. A ce niveau, s'il n'existe pas de handler qui rattrape l'exception, la recherche est déléguée au gestionnaire de réplication grâce à la méthode *invokeLeaderExceptionHandler(m)* comme le montre le listing de la figure 11.

Le GR, représenté par la classe `SageReplicationManager` (voir figure 5.1), se charge alors de la concertation des réponses envoyées par les répliques. Cette classe est instanciée à chaque fois qu'un nouvel agent est créé de sorte que chaque leader possède son propre gestionnaire de réplication. Si l'exception signalée est de type `ReplicaSpecific`, le GR enregistre l'exception en attendant les réponses des autres répliques. Sinon, si l'exception est de type `ReplicaIndependant`, l'exécution de toutes les répliques est interrompue et l'exception est propagée à l'appelant du service qui a signalé l'exception grâce à la méthode *callerHandlerSearch(e)* de la classe `SaGEManagerAgent`.

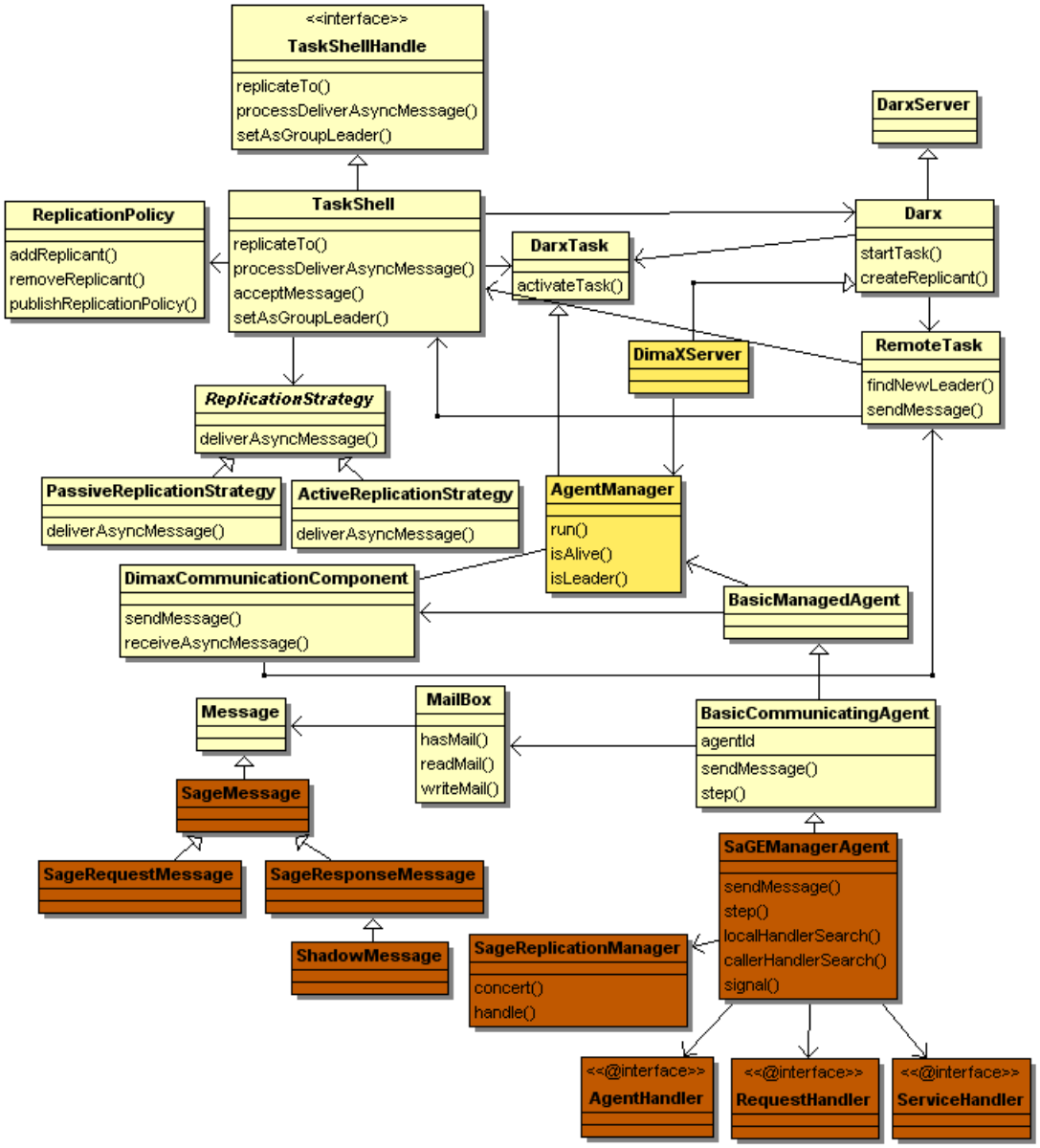

FIGURE 5.1 – SGE intégré à DIMAX

```
1 public void LocalHandlerSearch(String serviceName,
    Exception e){
2 /*                Service Handler                */
3 Method[] meths = this.getClass().getMethods();
4 for(Method m : meths){
5 if(m.getAnnotation(serviceHandler.class) != null){
6  if(m.getAnnotation(serviceHandler.class).serviceName().
     toString().equals(serviceName))
7          if(m.getParameterTypes()[0].equals(e.getClass())) {
8          print("Finding and Runnig Service handler");
9          runHandler(this, m, e);
10         return;
11         }
12 }
13 }
14 /*                 agent Handler                 */
15 for(Method m : meths){
16  if(m.getAnnotation(agentHandler.class) != null){
17    if(m.getParameterTypes()[0].equals(e.getClass())) {
18         print("Finding and Running agent handler");
19         runHandler(this, m, e);
20         return;
21    }
22 }
23 }
24 /*The method invokeLeaderExceptionHandler of the
     Replication Manager is called thanks to this method*/
25         invokeRpcExceptionHandler(e);
26 }
```

Listing 5.1 – Recherche locale de handler et délégation au gestionnaire de réplication

Comme nous l'avons précisé dans le chapitre précédent, pour que le traitement des messages soit cohérent avec la philosophie de notre traitement des exceptions et avec le traitement des messages asynchrones, nous avons spécialisé explicitement les messages de requête et les messages de réponse, en créant des sous-classes de `SageMessage` : `SageRequestMessage` et `SageResponseMessage` (voir figure 5.1). Nous avons spécialisé ensuite la primitive d'envoi de message *sendMessage(m)* de la classe `SaGEManagerAgent` pour qu'elle envoie directement les messages de requête et pour qu'elle traite spécifiquement les messages de réponse. Si le message envoyé est une réponse de la réplique, il est encapsulé dans un `Shadow Message` dans la classe

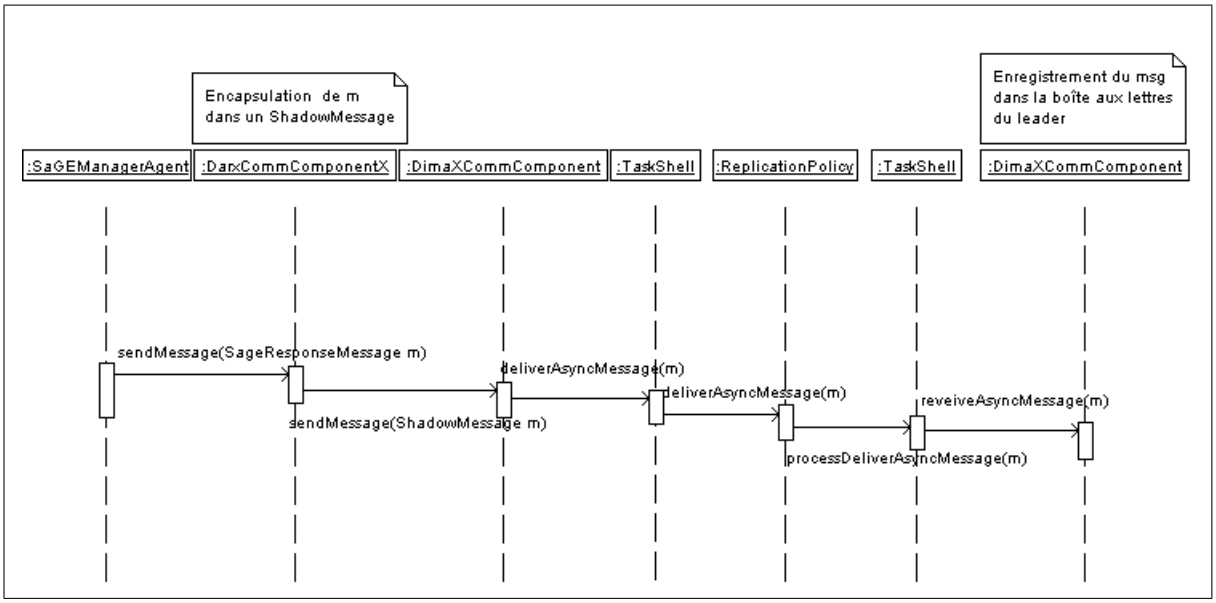

FIGURE 5.2 – Diagramme de séquences : Acheminement de la réponse d'une réplique à travers les classes de DIMAX

`DarxCommunicationComponentX`. Les `Shadow Messages` passent ensuite par les classes de Darx et sont enregistrés dans la boîte aux lettres du leader grâce à la méthode *receiveAsyncMessage(m)* de la classe `DimaXCommunicationComponent`. Pour résumer, toutes les réponses envoyées par les répliques sont encapsulées dans des `Shadow Messages` quelque soit leur type et sont redirigées vers la boîte au lettre du leader comme le montre le diagramme de séquence de la figure 5.2. Le GR se charge ensuite de la décision à prendre : signaler une exception, l'ignorer ou envoyer une réponse normale.

5.3 Etude de cas : Search For Flight

Nous avons choisi d'appliquer le SGE implémenté à l'exemple décrit dans les chapitres précédents : `Search For Flight`. L'exécution de l'application passe tout d'abord par les couches de DARX pour créer un serveur de nom et lancer le serveur sur les machines locales et distantes. Darx passe ensuite le contrôle à DIMAX pour la création des agents sur la machine locale. Chaque agent est représenté par un thread et est encapsulé dans une tâche système `DarxTask`. Cette tâche est ensuite encapsulée dans un `TaskShell` qui sert à gérer son groupe de réplication et assure la tranparence de la réplication vis-à-vis des autres agents. Ceci est illustré par la figure 5.3. Une tâche système peut être représentée par plusieurs agents (le leader et ses répliques).

Le leader d'une tâche est le premier agent créé associé à celle-ci. Lors de son enregistrement dans la machine locale, il crée un groupe de réplication et s'enregistre en tant que leader de ce groupe grâce à la méthode *setAsGroupLeader()* de la classe `TaskShell`.
Comme nous l'avons mentionné dans le chapitre précédent, les agents `Travel-`

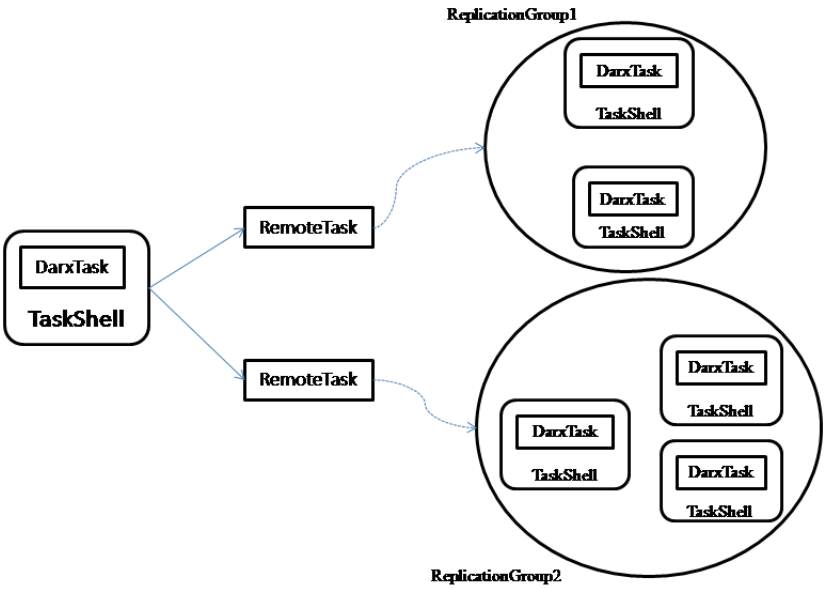

FIGURE 5.3 – Goupes de réplication repris de [8]

`Agency` et `AirlineCompany` sont répliqués. Après que tous les agents (leaders et répliques) soient créés, le service lookForService() du client est exécuté et envoie un message de requête : *getService()* au `Broker`. Ce dernier transmet la requête du client aux agents `TravelAgency` et `AirlineCompany` qui invoquent respectivement le service : *getPrice()*. Comme ces agents sont répliqués, ils transmettent le message qu'ils ont reçu du Broker à leurs répliques (méthode *deliverAsyncMessage()* de la classe `ActiveReplicationStrategy`). Chaque réplique exécute alors son service et retourne sa réponse, encapsulée dans un `Shadow Message` à son leader.
Pendant l'exécution de son service, l'une des répliques de l'agent `TravelAgency` signale l'exception `NoFlightAvailableException` qui implémente l'interface `ReplicaIndependent`. L'exécution du service *getPrice()* est alors suspendue et un handler qui correspond à l'exception signalée est recherché au niveau de la réplique. Etant donné qu'aucun handler n'a été trouvé localement, le processus de recherche se poursuit au niveau du GR comme nous l'avons

décrit précédemment. La fonction de résolution détecte que le type de l'exception est `ReplicaIndependent`, le GR arrête alors l'exécution des autres répliques et propage l'exception au Broker grâce à la méthode : *callerHandlerSearch(e)*. Le `Broker` est ainsi capable de gérer l'exception grâce au handler `@requestHandler` défini dans le broker.
En ce qui concerne l'agent `AirlineCompany`, on peut imaginer que l'une de ses répliques signale une exception de type `ReplicaSpecific` : `TemporaryTechnicalProblem`. La recherche de handler passe par les étapes décrites précédemment. La fonction de résolution du GR détecte que l'exception est `ReplicaSpecific`, l'exception est alors enregistrée et la fonction de résolution attend les réponses des autres répliques. Une autre réplique envoie une réponse normale, la première exception enregistrée est alors ignorée et la réponse normale est transmise par la suite au `Broker`.

5.4 Conclusion

Nous avons présenté dans ce chapitre l'implémentation de XSaGE et son application sur l'exemple `SearchForFlight`. Cet exemple montre que le SGE permet d'assurer la transparence de la réplication pour les programmeurs tout en leur offrant un moyen de signaler et de gérer les exceptions. Sans notre SGE, les exceptions de type `ReplicaSpecific` signalées par les répliques des agents n'auraient pas pu être rattrapées et enregistrées et auraient causé un conflit avec les réponses normales envoyées par les autres répliques.

Conclusion et perspectives

L'objectif de ce travail consistait à assurer la tolérance aux fautes dans une plate-forme multi-agents répliquée et ce, en intégrant un système de gestion d'exceptions (SGE) au mécanisme de réplication utilisé. Ce dernier consiste à créer des copies identiques de chaque agent appelées répliques et à les déployer sur différentes machines ; la stratégie de réplication utilisée est la réplication active. Le nombre de répliques associées à un agent est déterminé selon la criticité de celui-ci d'une façon complètement transparente. En cas de défaillance du leader, l'une des répliques de son groupe de réplication le remplace. Cependant, la perte d'une réplique causée par une exception (interne ou globale) non rattrapée peut engendrer différentes incohérences du système ou une réduction de ses performances.

Le SGE que nous avons proposé (XSaGE) offre la possibilité de rattraper les exceptions internes (`ReplicaSpecific`) signalées par les répliques d'un agent en conservant la transparence de la réplication. Ceci permet de maintenir la fiabilité d'un agent jusqu'au signalement d'une exception globale (`ReplicaIndependant`) ou l'envoi d'une réponse normale par l'une des répliques de celui-ci. Nous avons proposé également un mécanisme simple de définition de handlers, dédié aux développeurs, basé sur les annotations Java. Ce mécanisme permet d'attacher des handlers à des services, à des agents ou même à des requêtes pour faciliter le processus de recherche de handler. Ce dernier consiste à suivre la trace d'exécution des services pour traiter l'exception levée. En effet, lorsqu'une exception est signalée par l'une des répliques d'un agent, la recherche de handler est lancée tout d'abord dans le contexte de celui-ci (au niveau des handlers attachés au service qui a signalé l'exception puis au niveau des handlers attachés à l'agent) et est déléguée ensuite au gestionnaire de réplication du leader (du groupe de réplication de la réplique) si aucun handler adéquat n'a été trouvé. Le gestionnaire de réplication défini un traitement spécifique pour chaque type exception et maximise ainsi la possibilité d'avoir une réponse normale.

Nous avons enfin appliqué le SGE que nous avons implémenté dans DIMAX à un exemple simple de commerce électronique pour la recherche du meilleur prix d'un vol dans une agence de voyages et dans une compagnie aérienne. Ces dernières sont représentées par des agents dont la criticité indique qu'ils doivent être répliqués. Nous avons montré que le signalement d'une exception de type `ReplicaSpecific` n'affecte pas le comportement du système et que le signalement d'une exception de type `ReplicaIndependant` doit être propagé à l'appelant du service qui a signalé l'exception.

A l'issue des travaux que nous avons réalisés, nous pourrons envisager dans un premier temps d'optimiser la statégie de traitement des exceptions en ajoutant un mécanisme d'apprentissage. Ce mécanisme permettra d'éviter l'attente des réponses de toutes les répliques si plusieurs exceptions internes (`ReplicaSpecific`) similaires ont été signalées. Il pourra alors traiter ces exceptions de la même façon que les exceptions globales (`ReplicaIndependant`) et assurera ainsi la distinction dynamique des types d'exceptions et un gain important en temps d'exécution.
Nous pourrons envisager également d'appliquer notre SGE à d'autres stratégies de réplication comme la réplication passive et la réplication semi-active. Nous pourrons imaginer un système qui regroupe la réplication et le N-versionning [2]. Par exemple, si la réplication d'un agent devient coûteuse (en raison du manque de ressources, etc.), plusieurs versions d'un service jugé critique peuvent-être développées séparément. La gestion des exceptions dans un système similaire pourrait être l'une de nos perspectives.

Annexes

```
1 package dimaxx.sage;
2
3 /**
4  * SaGEManagerAgent is a subclass of BasicCommunicatingAgent
5  * that defines agents able to deal with exceptions
6  */
7
8 public class SaGEManagerAgent extends BasicCommunicatingAgent
9  implements requestHandler, serviceHandler, agentHandler {
10 private SageMessage messageXXInProcess;
11 private boolean active = true;
12 private BasicManagedAgent myAgent;
13 private SaGEReplicationManager rm = null;
14 /**
15  * Method to initialize the current processed message to null
16  */
17 private void initial(){
18     messageXXInProcess=null;
19 }
20
21 public SaGEManagerAgent(AgentIdentifier newAgent){
22   super(newAgent);
23   initial();
24 }
25
26 @Override
27 public boolean isActive() {
28   return active;
29 }
30
31 /**added by Selma Kchir*/
32 @Override
33 public void setReplicaManager(BasicManagedAgent agent,RemoteTask rem,
34 AgentManager manager) {
35    myAgent = agent;
36    myManager = manager;
37    rm = new SaGEReplicationManager(agent,manager);
38 }
39 /**
40  * This method sends the message of type MessageXX to another agent.
41  * @param agentId AgentIdentifier the destination agent's name
42  * @param am MessageXX the sending message
43  */
44
45 public void sendMessageXX(AgentIdentifier agentId, SageMessage am){
46         am.PrevMessage= getMessageXXInProcess();
47         super.sendMessage(agentId, am);
48         System.out.println("Send message to agent: "+agentId+" to Service: "
               +am.getReceiverService());
49 }
50
51 /** added by Selma Kchir*/
52 /** distinguer les messages de reponses des messages de requetes*/
53 public synchronized void sendMessageXX(AgentIdentifier agentId ,
       SageResponseMessage msg)
54 throws IllegalArgumentException, IllegalAccessException,
       InvocationTargetException{
55          searchForReceiverService(msg);
56         super.sendMessage(agentId, msg);
57 }
58
```

```
59 public void sendMessageXX(AgentIdentifier agentId , SageRequestMessage msg)
60 throws IllegalArgumentException, IllegalAccessException,
       InvocationTargetException{
61         msg.PrevMessage= getMessageXXInProcess();
62         searchForReceiverService(msg);
63         super.sendMessage(agentId, msg);
64 }
65
66 /**
67  * This method sends the message of type MessageXX to an agent.
68  * @param agents  Collection the destination agents
69  * @param am MessageXX the sending message
70  */
71 public void sendMessageXX(Collection agents, SageMessage am)
72 throws IllegalArgumentException, IllegalAccessException,
       InvocationTargetException {
73         searchForReceiverService(am);
74         am.PrevMessage = getMessageXXInProcess();
75         super.sendMessageX(agents, am);
76         for (int i = 0; i < agents.size(); i++)
77         System.out.println("Send message to agent:" + agents.toArray()[i]
78                                         + " to Service:" + am.
                                              getReceiverService());
79 }
80
81 @FipaACLEnvellopeHandler(
82         performative=Performative.Request,
83         content=SaGEProtocol.SendMessageXX,
84         protocol=SaGEProtocol.class,
85         attachementSignature= {AgentIdentifier.class, SageMessage.class})
86         public void receiveMessageXXToSend(FipaACLMessage s){
87                 sendMessageXX((AgentIdentifier) s.getAttachement()[0],
88                                 (SageMessage) s.getAttachement()[1]);
89         }
90         /**
91          * This method sends the message of type MessageXX to a set of
                agents.
92          * @param am MessageXX the sending message
93          */
94 public void sendAllXX(SageMessage am)
95 {
96         am.PrevMessage=getMessageXXInProcess();
97         super.sendAll(am);
98         System.out.println("Send message to agent:ALL to Service:"+am.
               getReceiverService());
99 }
100
101 /**
102  * This method returns the current processed message
103  * @return MessageXX
104  */
105 public SageMessage getMessageXXInProcess(){
106         return messageXXInProcess;
107 }
108
109 /**
110  * This method sets the value of the message
111  * @param  messageXXInProcess MessageXX
112  */
113 public void setMessageXXInProcess(SageMessage messageXXInProcess){
114         this.messageXXInProcess=messageXXInProcess;
115 }
```

```
116
117 /** The global signaling method
118          * @param  e Exception
119  */
120 @Override
121 public void signal(Exception e)
122 {
123         try{
124                 localHandlerSearch(getMessageXXInProcess().
                        getReceiverService(),e);
125                 print("The service which raises the exception is stopped: "+
                        getMessageXXInProcess().getReceiverService());
126                 Thread.currentThread().stop();
127                 }
128                 catch(Exception ex){
129                         print("The service which raises the exception is
                                stopped after executing it "+
130                                         new Exception().getStackTrace()[1].
                                                getMethodName());
131                         Thread.currentThread().stop();
132                 }
133 }
134
135 @FipaACLEnvellopeHandler(
136         performative=Performative.Inform,
137         content=SaGEProtocol.Signal,
138         protocol=SaGEProtocol.class,
139         attachementSignature={Exception.class}
140 )
141 public void receiveSignal(FipaACLMessage s){
142         signal((Exception) s.getAttachement()[0]);
143 }
144 /** * This method receives a handler find by the signaling method and
        invokes it.
145  * @param  o Object, the object which the method m Method execute within it
146  * @param  m Method, the handler to be executed
147  * @param  e Exception, the current exception object
148  * @return Object
149  */
150 public Object runHandler(Object o,Method m, Exception e)
151 {
152         try{
153             return m.invoke(o,new Object[]{e});
154         }
155         catch(Exception ex){
156                 return null;
157         }
158 }
159
160 /**
161 * Search for an handler in the local context of an agent.
162 * Handler search starts at the service level.
163 * If there is no suitable handler associated to the current service,
164 * a handler is searched at the agent level, and if there is no suitable
        agent level handler,
165 * the search will continue at the father agent level by sending the message
        sendExceptionMessage
166 * @param  serviceName String, the service in which a handler is searched
167 * @param  e Exception the current exception exception object
168 * @see #sendExceptionMessage(String, Exception)
169 */
170 public void localHandlerSearch(String serviceName,Exception e)
```

```
171 {print("localHandlerSearch:" + getIdentifier());
172 Method[] meths = this.getClass().getMethods();
173 for (Method m : meths)
174 {       if (m.getAnnotation(serviceHandler.class) != null)
175                 if (m.getAnnotation(serviceHandler.class).serviceName().
                        toString().equals(serviceName))
176                     if (m.getParameterTypes()[0].equals(e.getClass())) {
177                     print("Finding and Running Service handler");
178                     runHandler(this, m, e);
179                     return;
180                     }
181 }
182
183 for (Method m : meths)
184 {
185         if (m.getAnnotation(agentHandler.class) != null)
186                 if (m.getParameterTypes()[0].equals(e.getClass())) {
187                 print("Finding and Running agent handler");
188                 runHandler(this, m, e);
189                 return;
190                 }
191 }
192 invokeRpcExceptionHandler(e);
193 }
194
195 /**added by selma Kchir
196  * searchForReceiverService search for the annotation /@receiverService in
        the sender class and update the
197  * message with the relevant receiver service.
198  * @param msg
199  */
200 public void searchForReceiverService(SageMessage msg) throws
        IllegalArgumentException, IllegalAccessException,
        InvocationTargetException{
201 Method[] meths = this.getClass().getMethods();
202 for (Method m : meths){
203                 if (m.getAnnotation(receiverService.class) != null)
204         if (m.getAnnotation(receiverService.class).serviceName().toString().
                equals(msg.getSenderService())){
205          if (m.getParameterTypes()[0].getSimpleName().equals("Message")||
206          m.getParameterTypes()[0].getSimpleName().equals("SageMessage")||
207          m.getParameterTypes()[0].getSimpleName().equals("SageRequestMessage
                ")||
208          m.getParameterTypes()[0].getSimpleName().equals("
                SageResponseMessage")) {
209                 int x = m.getParameterTypes().length;
210                     m.invoke(this,msg);
211         }
212         }
213 }
214 }
215
216 /**
217  * This method delegate the handler search to the caller of the current one.
218  * callerHandlerSearch first look for the suitable handler associated to the
        active message in the current service,
219  * If it find it, it execute it by sending it the message runHandler.
220  * If it does not find the suitable handler,
221  * it calls recursively the method localHandlerSearch for the current
        service
222  * @param  serviceName String the service name which send the message
223  * @param  e Exception
```

```
224  */
225 public void callerHandlerSearch(String serviceName,Exception e)
226 {
227 try {
228         print("Agent Name :" + getId() + " / callerHandlerSearch :"     +
            serviceName);
229         //messageXX is the current message
230         SageMessage messageXX = getMessageXXInProcess();
231         //messageXX.PrevMessage is the message sent by the caller
232         messageXX = messageXX.PrevMessage;
233
234         Method[] meths = messageXX.getClass().getMethods();
235         for (Method m : meths)
236         if (m.getAnnotation(requestHandler.class) != null){
237                 if (m.getParameterTypes()[0].equals(e.getClass())) {
238                 System.out.println("method " + m);
239                 print("Finding and Running request handler");
240                 m.setAccessible(true);
241                 runHandler(messageXX, m, e);
242                 return;
243                 }
244         }
245         localHandlerSearch(serviceName, e);
246         }
247         catch (Exception ex) {
248                 print(ex.getMessage());
249         }
250 }
251
252 /**
253  * This method implements the asynchronous propagation of exception, while
        searching for a handler, within agents.
254  * @param  serviceName String the service name which send the message
255  * @param  e Exception
256  */
257 public void sendExceptionMessage(String serviceName,Exception e)
258 {
259         SageMessage m=getMessageXXInProcess();
260         if(m.getReceiverService().equals("callerHandlerSearch"))
261         m=m.PrevMessage.PrevMessage;
262         if(m!=null)
263         {
264                 SageMessage mCalle=new SageMessage("callerHandlerSearch",m.
                    getSenderService(),e,"sendExceptionMessage");
265                 mCalle.PrevMessage=m;
266                 print("Sending message Exception to parent agent");
267                 sendMessage(m.getSender(),mCalle );
268                 setMessageXXInProcess(null);
269         }
270         else     print("********* Root Agent *********");
271 }
272 /**added by Selma Kchir*/
273 public void invokeRpcExceptionHandler(Exception e)
274 {  SageMessage m = getMessageXXInProcess();
275 if(m!=null)
276 {
277         SageResponseMessage msg=new SageResponseMessage("
            invokeLeaderExceptionHandler",
278             m.getSenderService(),e,"invokeRpcExceptionHandler");
279         msg.setSender(this.getId());
280         msg.setReceiver(m.getReceiver());
281         msg.setDescription(m.getSenderService());
```

```
282          print("Sending exception Message to the replication manager !!!");
283          sendMessage(msg.getReceiver(),msg );
284          }
285 }
286
287 /** invoke the resolution function of the leader */
288 @Override
289 public void invokeLeaderExceptionHandler(SageMessage msg)
290 {
291 SageMessage m;
292 if(msg == null){
293  msg = getMessageXXInProcess();
294 }
295 if(msg.getType().equals("ShadowMessage"))
296  m = (SageMessage)msg.getContent();
297 else m =msg;
298 rm.concert(m);
299 }
300
301 /**
302  * This method implements the "retry" mechanism which re-execute the service
        to which the handler asking
303  * for the retry was attached.
304  */
305
306 public void retry(){
307          print("retry Method : "+getMessageXXInProcess().getReceiverService()
              );
308          getMessageXXInProcess().process(this);
309 }
310
311 @FipaACLEnvellopeHandler(
312          performative=Performative.Request,
313          content=SaGEProtocol.Retry,
314          protocol=SaGEProtocol.class
315 )
316 public void retry(FipaACLMessage s){
317          retry();
318 }
319 /**
320  * This method prints the output of the system on the screen
321  * @param  str String
322  */
323 @Override
324 public void print(String str){
325          Logger.exception(this, str);
326 }
327
328 /**added by Selma Kchir*/
329 @Override
330 public void step(){
331          if(this.hasMail()){
332          this.processNextMessage();
333          }
334 }
335 @Override
336 public Class<? extends Annotation> annotationType() {
337          // TODO Auto-generated method stub
338          return null;
339 }
340
341 @Override
```

```
342 public String serviceName() {
343         return((this.getMessageXXInProcess().getSenderService()));
344 }
345 }
```

Listing 2 – SaGEManagerAgent

```
 1 package dimaxx.sage.exceptionhandling;
 2 /**
 3  * @author Selma Kchir
 4  */
 5 import java.io.Serializable;
 6 import java.util.Vector;
 7
 8 import dima.basicagentcomponents.AgentIdentifier;
 9 import dima.kernel.communicatingAgent.BasicCommunicatingAgent;
10 import dimaxx.sage.SaGEManagerAgent;
11 import dimaxx.sage.SageMessage;
12 import dimaxx.server.AgentManager;
13 import dimaxx.serviceslibraries.BasicManagedAgent;
14
15 public class SaGEReplicationManager implements Serializable{
16 BasicCommunicatingAgent myAgent;
17 AgentManager myManager;
18 protected Vector<Exception> ReplicaResponse = null;
19 public SaGEReplicationManager(BasicManagedAgent agent,AgentManager manager){
20 myAgent = (BasicCommunicatingAgent)agent;
21 myManager = manager;
22 ReplicaResponse = new Vector<Exception>();
23 }
24
25 public void handle(SageMessage m){
26 SageMessage mCalle = new SageMessage("callerHandlerSearch",m.description(),r
       [j],"concert");
27 mCalle.PrevMessage=m;
28 receiver = m.getReceiver();
29 myAgent.sendMessage(receiver,mCalle );
30 }
31
32 public void concert(SageMessage m){
33 AgentIdentifier receiver = null;
34 Object[] r = m.getArgs();
35 for(int j = 0; j< r.length;j++)  {
36    if(r[j] instanceof ReplicaSpecific){
37    System.err.println("Logging exception "+ r[j]);
38    ReplicaResponse.add((ReplicaSpecific)r[j]);
39    //if all exceptions signaled are ReplicaSpecific, the most signaled
40    //exception type become replica independant and is propagated to the
          caller
41    if(myManager.handle.rs.getReplicants().size()+1 == ReplicaResponse.size()
          ) {
42         System.out.println("All exceptions signaled are ReplicaSpecific");
43    }
44 }
45 //else if the exception type is replica independant, the exception is
       signaled to the caller agent
46      else if(r[j] instanceof ReplicaIndependant){
47         System.err.println("Exception "+ r[j]);
48         handle(m);
49             return;
50     }
```

```
51 else      {
52          myAgent.sendMessage(m.getReceiver, m);
53          return;
54          }
55 }
56 }
```

Listing 3 – SaGEReplicationManager

```
1 /**
2 * requestHandler
3 * Defines the annotation "requestHandler" used to associate a handler to a
      message sending (or request).
4 */
5
6 @Retention(RetentionPolicy.RUNTIME)
7 @Target(ElementType.METHOD)
8 public @interface requestHandler {
9 }
```

Listing 4 – RequestHandler

```
1 /**
2 * Defines the annotation to associate a handler to an agent
3 */
4 @Retention(RetentionPolicy.RUNTIME)
5 @Target(ElementType.METHOD)
6 public @interface agentHandler {
7 }
```

Listing 5 – AgentHandler

```
1 /**
2 * serviceHandler
3 * Defines the "serviceHandler" annotation used to associate a handler to a
      service.
4 */
5
6 @Retention(RetentionPolicy.RUNTIME)
7 @Target(ElementType.METHOD)
8 public @interface serviceHandler {
9     String serviceName();
10 }
```

Listing 6 – ServiceHandler

```
1 /**
2  * @author Selma Kchir
3  */
4 public class DarxCommunicationComponentX extends DarxCommunicationComponent
      {
5
6 int msgNbr = 0;
7 public DarxCommunicationComponentX(DarxTaskEngine task) {
8         super(task);
9 }
10
11 public DarxCommunicationComponentX(){
12         super();
```

```
13 }
14
15 public DarxCommunicationComponentX(BasicCommunicatingAgent com) {
16         super (com);
17 }
18
19 public void sendMessage(AgentAddress addr, SageResponseMessage msg){
20          this.sendMessage(msg);
21 }
22
23 public synchronized void sendMessage(SageResponseMessage msg,Boolean leader)
       {
24 if(!leader){
25         ShadowMessage m = new ShadowMessage(msg);
26         m.setSender(msg.getSender());
27         m.setType("ShadowMessage");
28         m.setReceiver(msg.getSender());
29         sendMessage((Message) m);
30         }
31         else
32         {
33         sendMessage((Message)msg);
34         }
35 }
36 public void sendMessage(AgentIdentifier id, SageResponseMessage message) {
37         this.sendMessage(message);
38 }
39 }
```

Listing 7 – DarxCommunicationComponentX

```
1 public class AgentManager extends DarxTask implements Runnable{
2 public AgentManager(final BasicManagedAgent agent,
3                          final ServiceSupplier ...agents) {
4 super(agent.getId().toString());
5 this.agent = agent;
6 com = new DimaXCommunicationComponent();
7 agent.setCommunicationComponent(this.getCom());
8 this.addMethods(agent);
9 this.services = new ServicesHandler(agents);
10 for (ServiceSupplier s : agent.getRequiredServiceSuppliers())
11 this.services.addService(s);
12 if (agent instanceof ServiceSupplier)
13         ((ServiceSupplier) agent).setReplicaManager(agent, this.handle, this
               );
14 if (agent instanceof ServerManager)
15         ((ServerManager) agent).setReplicaManager(services);
16 if (agent instanceof BasicManagedAgent)
17         ((BasicManagedAgent) agent).setReplicaManager(agent, this.handle,
               this);
18 }
19 }
20 public class DimaXCommunicationComponentX extends
       DarxCommunicationComponentX{
21 @Override
22 public synchronized void sendMessage(final Message mess) {
23 //Defining the sender
24 AgentIdentifier sender;
25 if (AgentManager.this.getCurrentlyExecutedAgent()==null){
26 System.err.println("CurrentlyExecutedAgent()==null !!! message:\n"+mess);
27 sender = AgentManager.this.getIdentifier();
```

```
28 mess.setSender(sender);
29 }
30  else {
31 sender = AgentManager.this.getCurrentlyExecutedAgent().getIdentifier();
32 mess.setSender(sender);
33 }
34
35 //Defining reply to
36 if (mess instanceof FipaACLMessage && ((FipaACLMessage) mess).getReplyTo()==
       null)
37 ((FipaACLMessage) mess).setReplyto(sender);
38 /**added by Selma Kchir*/
39 //Defining the receiver
40 final AgentIdentifier receiver = mess.getReceiver();
41 DarxMessage m;
42 if(mess.getType().equals("ShadowMessage")){m = new DarxMessage((Message)mess
       ,sender.getFullId(), msgNbr++);}
43 else
44 //Defining the message
45 m = new DarxMessage(mess, sender.getFullId(), msgNbr++);
46 //On ne logue pas les message de log
47 //ou les notifications de message de log (stack overflow)
48 if (!isLogMessage(mess))
49 if (receiver.equals(AgentManager.this.getIdentifier()))
50 Logger.message(m, MessageStatus.LocalSending);
51 else
52 Logger.message(m, MessageStatus.Sending);
53
54 if (activationType.equals(ActivationType.FIPA)){
55 AgentManagementSystem.DIMAams.receive(mess);
56 } else {
57 RemoteTask remote = null;
58 try {
59 remote = AgentManager.this.findTask(receiver.toString());
60
61 if (remote != null){
62         remote.sendAsyncMessage(m);
63 } else {
64         Logger.exception(this, "Message perdu : \n"+m.getContents());
65         return;
66 }
67 } catch (final DarxException e) {
68         Logger.exception(this,
69         "Getting " + receiver + " from nameserver failed :",e);
70         return;
71 }
72 }
73 }
74
75 public synchronized void receiveAsyncMessage(final Object msg) {
76 final Message m = (Message) msg;
77 final String name = m.getReceiver().getFullId();
78 if (name.equals(AgentManager.this.agent.getIdentifier().getFullId()))
79 //THE RECEIVER IS THE AGENT
80 {
81         agentTemporaryMailbox.writeMail(m);
82         /** added by Selma Kchir*/
83         //if the current agent is a replica, the method processMessage of
              BasicCommunicatingAgent will be called to process the message.
84         if(!AgentManager.this.isLeader()) {
85         AgentManager.this.com.agentBehavior = (BasicCommunicatingAgent)
              AgentManager.this.agent;
```

```
86          AgentManager.this.com.agentBehavior.processMessage(m);
87          }
88  }
89
90  else if (AgentManager.this.services.containsService(name)){
91  //THE RECEIVER IS A SERVICE
92  //New mail box added if it is the first message receive by this monitor
93  if (!AgentManager.this.services.mailBoxes.containsKey(name))
94  AgentManager.this.services.mailBoxes.put(name, new SimpleMailBox());
95  //New mail box added if it is the first message receive by this monitor
96          AgentManager.this.services.mailBoxes.get(name).writeMail(m);
97  }
98  else{
99  //THE RECEIVER IS UNKNOWN
100 Logger.exception(this,"Unknown agent "+name);
101 try {
102 final FipaACLMessage reponse = new FipaACLMessage(
103 Performative.Failure,"unknown agent "+m.getReceiver(),NoneProtocol.class);
104 //INFORM THE SENDER THAT THE MESSAGE HAS NOT BEEN DELIVERED
105 setCurrentlyExecutedAgent(agent);
106 setCurrentlyReadedMail(m);
107 setCurrentlyExecutedMethod(new MethodHandler(getClass().getMethod(
108         "receiveAsyncMessage", Object.class)));
109
110 sendMessage(m.getSender(), reponse);
111 }
112 catch (Exception e) {
113         Logger.exception(this,"bad method name!!!",e);
114 }
115 finally {
116         resetCurrentlyExecutedAgent();
117         resetCurrentlyExecutedMethod();
118         resetCurrentlyReadedMail();
119 }
120 return;
121 }
122
123 //LOG IF THE MESSAGE IS NOT A LOG MESSAGE
124 if (!isLogMessage(m))
125 if (m.getSender().equals(AgentManager.this.agent.getIdentifier()))
126     Logger.message((Message) msg, MessageStatus.ReceivedLocal);
127 else
128     Logger.message((Message) msg, MessageStatus.Received);
129 }
130 }
131 }
```

Listing 8 – Extrait du code de la classe AgentManager

```
1  public abstract class BasicCommunicatingAgent extends BasicManagedAgent {
2  protected Map<String, AgentAddress> aquaintances;
3  private AbstractMailBox mailBox;
4  protected DarxCommunicationComponentX com;
5  protected Vector<Message> ReplicaResponse = null;
6
7  public void setMailBox(AbstractMailBox mailBox) {
8          this.mailBox = mailBox;
9  }
10
11 /**
12  * Returns the first message of the mailBox
```

```
13  * @return boolean
14  */
15 public AbstractMessage getFirstMessage() {
16         // Test hasMail first ...
17         return mailBox.getFirstMessage();
18 }
19
20 / ** Returns the message contained in the maiBox
21  */
22 public AbstractMessage getMessage() {
23 // Test hasMail first ...
24 return mailBox.readMail();
25 }
26
27 public boolean hasMail() {
28 return getMailBox().hasMail();
29 }
30
31 public void processNextMessage() {
32 SageMessage m = (SageMessage) getMessage();
33 /**added by Selma Kchir*/
34 if (m.getType().equals("ShadowMessage"))         {
35                 invokeLeaderExceptionHandler(m);
36 }
37
38
39 else if (m.getType().equals("java"))     {
40         m.process(this);
41 }
42
43 else     processAclMessage(m);
44 }
45
46 public Object processMessage(Message m) {
47 /**added by Selma Kchir*/
48 if (m.getType().equals("ShadowMessage"))         {
49         System.out.println("Replica Answer !! ");
50 }
51 else{
52 if (m.getType().equals("java"))
53         return m.process(this);
54 else
55         processAclMessage(m);
56 }
57             return null;
58 }
59
60 public synchronized boolean put(AbstractMessage m) {
61  return mailBox.writeMail(m);
62 }
63
64 public void readAllMessages() {
65         while (hasMail())
66         readMailBox();
67 }
68
69 public void readMailBox() {
70         if (hasMail())
71         processNextMessage();
72 }
73
74 public void receive(Message m) {
```

```
75          put(m);
76  }
77
78  public void sendMessage(AgentIdentifier agentId, Message am) {
79  am.setSender(getIdentifier());
80  am.setReceiver(agentId);
81  if (aquaintances.containsKey(agentId.toString()))
82          com.sendMessage(aquaintances.get(agentId.toString()), am);
83  else
84          com.sendMessage(am);
85  }
86
87
88  public void sendMessage(AgentAddress agentId, Message am) {
89          am.setSender(getIdentifier());
90          am.setReceiver(agentId.getId());
91          com.sendMessage(agentId, am);
92  }
93
94  public synchronized void sendMessage(AgentIdentifier agentId ,
        SageResponseMessage msg){
95          msg.setSender(getIdentifier());
96          msg.setReceiver(agentId);
97          if (aquaintances.containsKey(agentId.toString())) {
98                  com.sendMessage(aquaintances.get(agentId.toString()), msg);
99          }
100         else {
101                 com.sendMessage(msg,myManager.isLeader());
102         }
103 }
104
105 public void sendMessage(AgentIdentifier agentId , SageRequestMessage msg){
106         msg.setSender(getIdentifier());
107         msg.setReceiver(agentId);
108         if (aquaintances.containsKey(agentId.toString())) {
109                 com.sendMessage(aquaintances.get(agentId.toString()), msg);
110         }
111         else {
112                 com.sendMessage(msg);
113         }
114 }
115
116 public void sendMessage(Collection<AgentAddress> agents, Message am) {
117 Iterator<AgentAddress> iter = agents.iterator();
118 am.setSender(getIdentifier());
119 while (iter.hasNext())
120         this.sendMessage(iter.next().getId(), am);
121 }
122
123 /**added by Selma Kchir
124  * this method sends a message to a collection of agents  */
125
126 public void sendMessageX(Collection<AgentIdentifier> agents, Message am) {
127         Iterator<AgentIdentifier> iter = agents.iterator();
128         am.setSender(getIdentifier());
129         while (iter.hasNext())
130         this.sendMessage(iter.next(), am);
131 }
132
133 @Override
134 public  void step(){};
135
```

```
136 public AbstractMessage removeFirstMessage() {
137         return ((SimpleMailBox) getMailBox()).removeFirstMessage();
138 }
139
140 public void invokeLeaderExceptionHandler(SageMessage m) {
141         System.out.println("-----------Leader Replication Manager-----------
                ");
142 }
143 }
```

Listing 9 – Extrait du code de la classe BasicCommunicatingAgent

```
 1 public class AgentClient extends SaGEManagerAgent{
 2
 3 @service
 4 public void lookForService() {
 5 Object destination = "t";
 6 Object date = "01/10/09";
 7 System.out.println("Client launched ^^");
 8 SageRequestMessage msg = new SageRequestMessage(destination,date, "
       lookForService")
 9 {
10         @requestHandler public void     handler(final Exception e){
11         System.out.println("CLIENT : Request handler execution");
12         }
13 };
14 try {
15         msg.setSender(getIdentifier());
16         msg.setReceiver(AgentBroker.getBrokerId("Broker"));
17         sendMessageXX(AgentBroker.getBrokerId("Broker"), msg);
18 }
19 catch (Exception e) {
20 System.out.println("\n impossible d'envoyer le MSG au BROKER \n");
21 e.printStackTrace();
22 }
23 }
24
25 @receiverService(serviceName = "lookForService")
26 public void invokeReceiverService(SageRequestMessage m){
27 m.setContent("getService");
28 }
29
30 @service
31 public void receiveService(Object agence, Object price) {
32 System.out.println("The requested flight costs " + price + " euros in the
       agence "
33                 + agence);
34 System.out.println("End of service");
35 }
36
37 @serviceHandler(serviceName = "lookForService")
38 public void badParametersAgence(Exception e) {
39         System.out.println("CLIENT : LookForService service handler
               exception !");
40 }
41
42 @agentHandler
43 public void ressourceFailureAgence(noFlightAvailableException e) {
44         System.out.println("CLIENT : Agent handler exception!");
45 }
46
```

```
47 public static AgentIdentifier getClientId(String Id) {
48         return new AgentName(Id);
49 }
50
51 @MessageHandler
52 @TransientStepComposant(100)
53 @FipaACLEnvellopeHandler(performative = Performative.Inform, content =
       CreationProtocol.AppLaunch, protocol = CreationProtocol.class,
       attachementSignature = { Date.class })
54 @Override
55 public boolean applicationStart(FipaACLMessage init) {
56 System.out.println("Start Trip search ! Gooo");
57 lookForService();
58 return true;
59 }
60 }
```

Listing 10 – AgentClient

```
 1 public class TravelAgency extends SaGEManagerAgent{
 2 @agentHandler
 3 public void handlerAgent(noFlightAvailableException e){
 4         System.out.println("AGENCE2 : Agent Handler execution");
 5 }
 6
 7 @serviceHandler(serviceName="getPrice")
 8 public void handlerService(noFlightAvailableException e){
 9         System.out.println("AGENCE2 : Agence2 getPrice Service Handler
               execution");
10 }
11
12 Vol [] offre = new Vol[4];
13 HostIdentifier hostwhereToReplicate;
14
15 public void volInit(){
16         offre[1] = new Vol("m","01/08/09","120");
17         offre[2] = new Vol("n","01/08/09","150");
18 }
19
20 public TravelAgency (final AgentIdentifier ID,HostIdentifier
       whereToReplicate){
21         super(ID);
22         volInit();
23         hostwhereToReplicate = whereToReplicate;
24 }
25
26 public static AgentIdentifier getAgenceId(String agtId){
27         return new AgentName(agtId);
28 }
29
30 public Object searchPrice(Object destination , Object date){
31 if(myManager.isLeader())
32         offre[0] = new Vol("t","01/10/09","110");
33 else
34         offre[0] = new Vol("l","31/11/09","110");
35 Object price = new Object();
36 for(int i = 0 ; i<3 ; i++)
37 if(this.offre[i].getDestination().equals(destination))
38 if(this.offre[i].getdate().equals(date)){
39           price = (Object) this.offre[i].getcost();
40           return(price);
```

```
41 }
42 else price = 0;
43 return price;
44 }
45
46 @service
47 public void getPrice(Object destination,Object date)
48      throws IllegalArgumentException, IllegalAccessException,
           InvocationTargetException, InterruptedException{
49 Object price = searchPrice(destination , date);
50 if( ! price.toString().equals("0")){
51         SageResponseMessage MsgToBrk = new SageResponseMessage((Object)this.
               getIdentifier(),price,"getPrice"){
52         @SuppressWarnings("unused")
53         @requestHandler
54         public void handle(Exception e){
55                 System.out.println("Agence2 : requestHandler exception");
56         }};
57         sendMessageXX(AgentBroker.getBrokerId("Broker"), MsgToBrk);
58         }
59 else     signal(new TemporaryTechnicalProblem());
60 }
61
62 @receiverService(serviceName = "getPrice")
63 public void invokeReceiverService(SageResponseMessage m){
64        m.setContent("receivePrice");
65 }
66
67 @MessageHandler
68 @TransientStepComposant(100)
69 @FipaACLEnvellopeHandler(
70         performative=Performative.Inform,
71         content=CreationProtocol.AppLaunch,
72         protocol=CreationProtocol.class,
73         attachementSignature={Date.class}
74 )
75 @Override
76 public boolean applicationStart(FipaACLMessage init){
77 try {
78         myManager.handle.replicateTo(hostwhereToReplicate.getUrl(),
              hostwhereToReplicate.getPortNumber());
79 }
80  catch (RemoteException e) {
81         System.out.println("Impossible to replicate on this host");
82 }
83 return true;
84 }
```

Listing 11 – TravelAgency

Bibliographie

[1] Système de gestion d'exceptions. `http://fr.wikipedia.org/wiki/Systeme_de_gestion_d'exceptions`, 2009.

[2] Avizienis. The n-version approach to fault tolerant systems. pages 1491–1501. IEEE TSE, 1985.

[3] A. de Luna Almeida, S. Aknine, J.-P. Briot, and J. Malenfant. A predictive method for providing fault tolerance in multi-agent systems.

[4] C. Dony. A fully object-oriented exception handling system : rationale and smalltalk implementation. chapter 2, pages 18–38.

[5] C. Dony, C. Urtado, and S. Vauttier. Exception handling and asynchronous active objects : Issues and proposal. In *Advanced Topics in Exception Handling Techniques*, chapter 5, pages 81–101.

[6] S. Ductor. Dimax. `http://www-poleia.lip6.fr/~ductors/fr/dimax.html`, 2010.

[7] S. Ductor, Z. Guessoum, and M. Ziane. Gestion des ressources et réplication adaptative pour fiabiliser les sma. In *(to appear) Rencontre des Jeunes Chercheurs en Intelligence Artificielle*, 5 2009.

[8] N. Faci, Z. Guessoum, and O. Marin. Dimax : A faulttolerant multi-agent platform. In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, pages 13 – 20, 2006.

[9] J. Ferber. *LES SYSTEMES MULTI-AGENTS : Vers une intelligence collective*. InterEditions, 1995.

[10] J. B. Goodenough. Exception handling : Issues and a proposed notation. *In CACM*, 18(12) :683–696, 1975.

[11] Z. Guessoum, N. Faci, and J.-P. Briot. Adaptive replication of large-scale multi-agent systems : towards a fault-tolerant multi-agent platform. In *Proc. of SELMAS'06*. Vol. 3914 LNCS, Springer, 2006.

[12] Z. Guessoum, M.Ziane, and N. Faci. Monitoring and organizational-level adaptation of multi-agent systems. *AAMAS*, pages 514–521, 2004.

[13] Z. Guessoum, B. J. pierre Briot, A. N. Faci, and O. M. A. Towards reliable multi-agent systems - an adaptive replication mechanism. multiagent and grid systems (mags) an international journal. accepté pour publication avec révisions mineures.

[14] V. Issarny. Concurrent exception handling. In *Advances in Exception Handling Techniques*, number 2022 in LNCS, pages 111–127. Springer, 2001.

[15] P. LEE and T. ANDERSON. Fault tolerance : Principles and practice, 1980.

[16] L. Mancini and S. Shrivastava. Exception handling in replicated systems with voting. In *Digest of papers, Fault Tol. Comp. Symp-16*, pages 384–389, 1986.

[17] O. Marin, M. Bertier, and P. Sens. Darx—a framework for the fault-tolerant support of agent software. In *Proc. of ISSRE'03*, page 406. IEEE CS, 2003.

[18] R. Miller and A. Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Trans. Software Eng.*, 30(12) :1008–1022, 2004.

[19] F. Souchon, C. Dony, C. Urtado, S. Vauttier, and J. Ferber. Sage : une proposition pour la gestion d'exceptions dans les systèmes multi-agents. *Lecture Notes in Computer Science*, 2002.

[20] J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems : from model to system implementation.