Fiabilité des applications multi-agents : le système de gestion d'exceptions SaGE

Frédéric Souchon [*] [**] — Sylvain Vauttier[*] — Christelle Urtado[*] — Christophe Dony[**]

[*] *LGI2P - Ecole des Mines d'Alès - Site EERIE - Parc Scientifique G. Besse - F30035 Nîmes {Frederic.Souchon, Sylvain.Vauttier, Christelle.Urtado}@site-eerie.ema.fr*

[**] *LIRMM - 161 rue Ada - 34392 Montpellier Cedex 5 {dony}@lirmm.fr*

RÉSUMÉ. *Les paradigmes de développement à base de composants et d'agents répondent aux tendances actuelles du développement d'applications par intégration d'éléments logiciels distribués sur de nombreux systèmes. Les systèmes multi-agents (SMA) proposent des mécanismes qui permettent de gérer des systèmes ouverts et dynamiques, les agents étant des entités actives, dotées de capacités de communication évoluées, capables d'établir seules des collaborations avec d'autres agents. Malheureusement, les SMA n'offrent pas le même support que les systèmes à base de composants pour assurer la rigueur du développement et la fiabilité de l'exécution. Ainsi, peu de SMAs intègrent un système de gestion d'exceptions (SGE) dédié et aucun de ces SGE, à notre connaissance, ne respecte toutes les caractéristiques du paradigme agent (autonomie des agents, concurrence coopérative). Cet article présente Sa*GE*, une proposition de système de gestion d'exceptions dédiée aux systèmes multi-agents, que nous avons implémentée et testée dans la plateforme M*ADKit.

ABSTRACT. *Both component and agent-based developpement paradigms provide solutions to develop applications through the integration of distributed software parts. On the one hand, multi-agent systems (MASs) provide mechanisms that allow to manage open, dynamic systems. Indeed, agents are active entities that uses high-level communication capabilities to autonomously manage collaborations with other agents. On the other hand, muti-agent systems lack some mechanisms to enforce rigorous developpment and run-time reliability as compared to component-based systems. For example, very few MASs provide a dedicated exception handling system (EHS) and, to our knowledge, none of the existing ones fully conforms to the agent paradigm (agent autonomy, cooperative concurrency). This paper presents Sa*GE*, our proposition for an EHS that is dedicated to multi-agent systems, which has been implemented and tested over the M*ADKit platform.

MOTS-CLÉS : *Système de gestion d'exceptions, Systèmes multi-agents, Fiabilité, Concurrence*

KEYWORDS: *Exception handling system, Multi-agent systems, Reliability, Concurrency*

1. Introduction

Les paradigmes de développement à base d'agents [WOO 99] et de composants [SZY 98] fournissent des réponses différentes aux tendances actuelles de développement d'applications par intégration d'éléments logiciels distribués sur de nombreux systèmes. Les agents sont des entités actives, dotées de capacités de communication évoluées (envoi asynchrone de messages), capables de gérer de manière autonome les collaborations nécessaires à leur bon fonctionnement. Ainsi, les systèmes multi-agents (SMA) proposent des mécanismes adaptés à la gestion de systèmes complexes, tels que des systèmes ouverts et dynamiques. Les composants, quant à eux, sont déployés et intégrés dans des environnements d'exécution fortement typés. Leurs collaborations reposent sur l'utilisation de bus logiciels au travers desquels les composants échangent des appels de méthodes, des valeurs de retour et des exceptions. Les plateformes à base de composants imposent une rigueur qui améliore la fiabilité du système, aussi bien lors de la phase de développement (vérifications syntaxiques) qu'en cours d'exécution (gestion des exceptions). Mais cette rigueur est obtenue au prix d'une certaine rigidité (génération de glue logicielle, compilation) qui contraint le déploiement et l'assemblage des composants à être essentiellement statiques. Nous pensons qu'il manque encore une solution hybride, reprenant les qualités des deux paradigmes, tels que de véritables composants orientés messages. Cet article explore une autre voie pour une telle solution hybride : l'intégration, dans un modèle d'agents, d'un mécanisme de gestion d'exceptions. En effet, la gestion d'exceptions est un mécanisme utile pour coordonner avec rigueur les activités des agents pour produire des schémas d'exécution complexes (activités redondantes, concurrentes et asynchrones) [KNU 01]. Après étude, on constate que peu de SMA intègrent un système de gestion d'exceptions (SGE) dédié et qu'aucun de ces SGE, à notre connaissance, ne respecte toutes les caractéristiques du paradigme agent (autonomie des agents, concurrence coopérative). Cet article présente SaGE, une proposition de système de gestion d'exceptions dédiée aux systèmes multi-agents, que nous avons implémentée et expérimentée dans la plateforme MADKit [MAD]. La section 2 introduit les principaux concepts relatifs à la gestion d'exceptions et au paradigme agent. La section 3 présente la problématique de la gestion d'exceptions dans le contexte des SMA et étudie les solutions existantes. La section 4 décrit le SGE SaGE, illustré par un exemple classique. Finalement, la section 5 décrit l'implémentation de SaGE dans MADKit et présente une expérimentation.

2. Concepts de base

2.1. *La gestion d'exceptions*

Les événements exceptionnels, souvent appelés *exceptions* [GOO 75], sont utilisés au sein d'un programme pour notifier une situation indésirable qui empêche son exécution standard de se poursuivre. Un *système de gestion d'exceptions* (SGE) [WEI 83, MEY 88, KOE 89] fournit des structures de contrôle et des mécanismes permettant de lever, de signaler et de traiter les exceptions. La levée d'une exception in-

terrompt l'exécution standard du programme. Sa continuation est alors prise en charge par le SGE qui procède au signalement de l'exception, c'est-à-dire à la recherche d'un traitement pouvant être appliqué pour gérer cette erreur. De tels traitements sont définis par le programmeur dans des gestionnaires d'exceptions, appelés *handler*s. L'exécution des actions correctives définies par un handler permet soit de reprendre l'exécution là ou elle a été interrompue (reprise), soit en un autre point, en abandonnant tout ou partie des calculs en cours au moment de la levée d'exception (terminaison). Un handler est associé à une unité de programme dont la granularité et la nature sont variables d'un système à un autre (classe, procédure, bloc, expression) [DON 90] : le code réalisant le traitement des erreurs est ainsi clairement séparé du code applicatif. Lorsqu'une exception est levée dans une unité de programme, le mécanisme de signalement recherche un handler adapté (correspondant au type de l'exception) parmi les handlers associés à cette unité. Si un tel handler existe, il est exécuté. Sinon, la recherche se poursuit dans l'ensemble des unités de programme affectées par l'interruption de l'exécution. Typiquement, le mécanisme de signalement utilise la pile des appels pour rechercher les handlers associés aux unités de programme ayant directement ou indirectement fait appel à l'unité de programme qui a levé l'exception. Ce type de mécanisme est le plus fréquemment utilisé[1], par exemple dans les langages C++ [KOE 89] et Java. Le SGE permet ainsi de lever une exception dans une unité de programme sans rendre son traitement local obligatoire : si un handler capable de traiter l'exception est associé à une autre unité du contexte d'exécution, il est automatiquement (sans programmation ad hoc) invoqué par le SGE.

2.2. *Le paradigme agent*

La principale caractéristique qui distingue les agents des autres composants logiciels est leur autonomie [WOO 99]. Un agent a la capacité de décider spontanément de réaliser une activité afin de remplir des objectifs individuels. Les agents s'exécutent donc concurremment dans des threads séparés. Ils ne sont cependant pas isolés : ils interagissent en échangeant des messages grâce à des mécanismes de messagerie asynchrones. Ainsi, les agents peuvent collaborer tout en préservant leur autonomie. Envoyer un message à un autre agent pour lui demander un service est une action non bloquante : l'agent client (l'expéditeur de la requête) n'attend pas la réponse de l'agent serveur et peut poursuivre son activité courante. Le client récupère la réponse plus tard, dans un message envoyé en retour par le serveur. Réciproquement, recevoir un message n'est pas un événement préemptif : l'agent récepteur peur choisir de reporter le traitement de ce message afin de terminer des activités plus prioritaires. L'asynchronisme des communications donne aux agents les moyens de gérer des schémas d'exécution avancés comme requérir de manière redondante le même service à différents agents pour obtenir de meilleures garanties de performance et de fiabilité. Par exemple, un agent client peut consolider un résultat à partir des diverses réponses

1. Il existe, plus rarement, des systèmes pour lesquels le mécanisme de recherche s'appuie, *a contrario*, sur la structure statique (c'est-à-dire lexicale) du programme.

qu'il reçoit d'un ensemble d'agents serveurs et décider d'arrêter ce processus quand un délai fixe est écoulé (pour garantir des performances en termes de temps d'exécution) ou quand un certain nombre de réponses lui sont parvenues (pour garantir la représentativité du résultat).
Les systèmes multi-agents sont organisés en structures sociales. Par exemple, dans Aalaadin [FER 98], le modèle social de MADKit, les agents sont rassemblés en *groupes* au sein desquels ils jouent des *rôles*. Les agents ne communiquent qu'entre agents d'un même groupe. Les interactions entre groupes sont donc gérées par des agents appartenant à plusieurs groupes et servant d'intermédiaires. Les rôles définissent des capacités communes à un sous-groupe d'agents. Un même agent peut remplir plusieurs rôles. Les rôles servent, en outre, à s'adresser collectivement à un ensemble d'agent : une requête adressée à un rôle est diffusée à l'ensemble des agents concernés. L'émetteur de la requête ignore l'identité et le nombre des agents à qui il s'est adressé et il peut recevoir, en retour, une réponse collective unique.

3. Gestion d'exceptions dans les SMAs : Problématique et travaux existants

3.1. *Agents et exceptions : besoin d'un SGE spécifique*

Les systèmes à base d'agents sont développés à l'aide d'environnements écrits dans de langages de programmation «ordinaires» (typiquement *java*). Cela permet d'utiliser ce même langage pour programmer les agents en utilisant les éléments de modèle fournis par la plateforme. Cette imbrication entre le système à base d'agents et les éléments du langage de programmation nécessaires pour implémenter ce système (typiquement des classes et des objets) entretient la confusion entre les deux paradigmes. Cette confusion se traduit en termes de gestion d'exceptions : de nombreux SMA (AgentBuilder, Jack, MADKit etc.) [RIC 00] ne disposent pas d'un SGE spécifique et s'appuient donc totalement, pour la gestion d'exception, sur les capacités du langage de programmation sous-jacent. Si ces SGE permettent de gérer les exceptions internes aux agents, liés à leur implémentation, ils n'offrent pas de solution pour assurer le signalement d'exceptions entre agents. En effet, une exception ne peut être signalée à d'autres agents que sous la forme de messages adressés à ces agents. Si le SMA ne propose pas un type de message explicitement identifié comme une exception, la gestion des erreurs retombe dans les travers des solutions ah hoc basées sur l'utilisation de messages standard contenant des valeurs particulières. La réception d'un message avertissant d'une exception doit produire une réaction chez l'agent. L'intégration d'un véritable SGE au modèle d'exécution de l'agent permet alors au programmeur d'identifier explicitement les comportements destinés à traiter les exceptions puis de laisser le modèle d'exécution prendre en charge leur déclenchement (ce qui évite à nouveau le recours à de la programmation ad hoc, donc certains risques d'erreurs). Un exemple typique concerne la destruction des agents [KLE 03]. Les agents sont en général implantés dans des threads distincts afin d'assurer leur autonomie. Une exception levée au niveau du langage est signalée en remontant la pile des appels du thread. Si aucun handler est trouvé, le thread est détruit. Du point de

vue du SMA, cette terminaison s'interprète comme la mort accidentelle d'un agent. Le SMA doit prendre en charge le signalement de cet événement aux autres agents sous la forme d'envois de messages les informant de la mort de l'agent. Les agents ainsi avertis pourront alors réagir en conséquence (par exemple ne plus attendre un service demandé à l'agent défunt).

3.2. *Exceptions dans les collaborations entre agents : qualités attendues*

3.2.1. *Respect du contrat*

Les modes d'interaction entre agents d'un système peuvent être classés en deux catégories principales : information (acte qui consiste à notifier un agent de situations susceptibles de l'intéresser sans en attendre quelque réaction que ce soit en retour) ou requête / réponse (acte qui consiste à émettre une requête auprès d'un agent, lui demandant de rendre un service dont est attendu un résultat). Dans ce travail, nous nous intéressons au second mode d'interaction, les requêtes / réponses, comme support des collaborations entre agents. Lorsqu'un agent (client) demande un service à un agent (serveur) et que le serveur accepte de rendre le service, nous considérons que le serveur est engagé vis-à-vis du client. Cet engagement, à rapprocher de la notion de contrat logiciel [MEY 88], porte, selon nous, à la fois sur la fourniture d'une réponse en cas d'exécution «normale» mais aussi sur la fourniture d'une réponse exceptionnelle lorsque l'agent est empêché de rendre le service demandé. Le client est ainsi en mesure de mettre en place une stratégie correctrice adaptée. Ainsi, dans une collaboration, nous considérons que l'agent client d'un service doit être prévenu si le service ne peut pas lui être rendu comme convenu. C'est la première qualité que nous attendons d'un système de gestion des exceptions dans les collaborations entre agents.

3.2.2. *Respect de l'autonomie*

Les agents ont pour caractéristique d'être des entités logicielles autonomes qui encapsulent l'intégralité de leur comportement et interagissent sans contrôle centralisé. La deuxième qualité que nous attendons d'un système de gestion des exceptions dans les collaborations entre agents est qu'il se conforme à ce principe en respectent l'autonomie des agents.
L'étude que nous avons menée des SGE existants pour les plateformes à agents, nous a montré que les rares propositions [TRI 01, KLE 03] fondent la gestion des exceptions sur des agents spécialisés, appelés superviseurs ou gardiens, qui «externalisent» la gestion d'exception. Dans [KLE 03], les exceptions survenant durant l'exécution d'un ensemble d'agents sont signalées à un agent superviseur, auquel sont associés les handlers permettant de les gérer. Ce système est adapté à la gestion d'exceptions «globales», c'est-à-dire susceptibles de concerner l'ensemble des agents du système (mort des agents, problème technique externe au système compromettant son exécution, etc.). Il ne convient, par contre, pas à la gestion des collaborations entre agents puisqu'il fait intervenir un agent tiers qui, soit n'a qu'une connaissance extérieure des agents qu'il supervise et ne peut donc pas appliquer des actions correctives pertinentes,

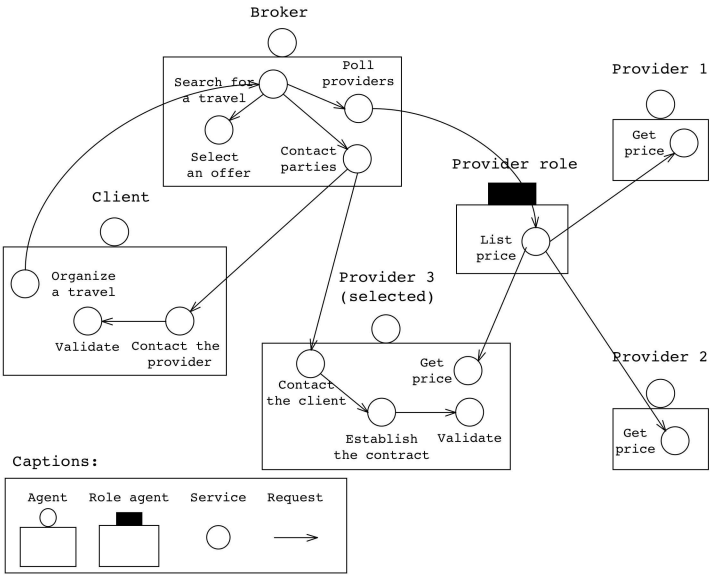

Figure 1. *Exécution résultant d'une requête à l'agence de voyage*

soit n'est capable d'avoir des réactions adaptées à la situation qu'au prix d'une intrusion dans le comportement des agents qu'il supervise. Dans [TRI 01], c'est un agent unique qui est destinataire de toutes les exceptions survenues dans les agents supervisés par un gardien. A chaque nouvelle exception, le système exécute un ensemble de règles, définies par le programmeur, qui établissent quels sont, parmi les agents supervisés par le gardien, ceux qui doivent se voir signaler une exception. Ce système, s'il est moins centralisé que le précédent car le gardien ne traite pas lui-même les exceptions mais en assure seulement le signalement présente plusieurs points faibles dans le cadre des collaborations entre agents. Tout d'abord, la gestion d'exception repose toujours partiellement sur une entité extérieure aux collaborations. Ensuite, les règles de signalement des exceptions comportent souvent des références au comportement interne des agents. Enfin, il impose un effort de paramétrisation supplémentaire au programmeur. Pour l'ensemble des raisons ci-dessus, les SGE existant dans les plateformes à agents ne nous semblent pas adaptés à la gestion des exceptions dans les collaborations entre agents.

3.2.3. *Prise en compte de spécificités liées à la concurrence*

Gestion des activités collectives. Lorsque les agents d'un système collaborent, ils participent, conjointement les uns avec les autres, à la réalisation d'*activités collectives*, tout en étant concurrents. Ce mode de concurrence est qualifié par [ROM 01a] de *concurrence coopérative*. [ROM 01a] montre que la gestion d'exceptions dans un système concurrent coopératif nécessite de disposer, à l'exécution, d'une représen-

tation des activités collectives, que le SGE peut utiliser pour retrouver les activités potentiellement affectées par une exception afin de la leur signaler. Ainsi, un système de gestion d'exceptions dans les collaborations entre agents doit permettre l'association de handlers à une activité collective afin d'exprimer et de gérer l'impact des défaillances de chaque agent participant à ce contexte d'exécution global. Il est alors possible d'assurer la gestion d'exceptions dans les différents contextes d'exécution qui se construisent dynamiquement au fil des collaborations qui s'établissent entre les agents, depuis le contexte d'exécution local de l'activité individuelle d'un agent, vers le contexte d'exécution global d'une activité collective à laquelle il participe et récursivement, vers le contexte d'exécution d'une activité collective plus englobante. La troisième qualité que nous attendons d'un système de gestion des exceptions dans les collaborations entre agents est qu'il s'appuie sur une représentation des activités concurrentes collectives des agents.

Gestion d'exceptions concertées. L'échec d'une activité peut être mineure au regard de l'objectif de l'activité collective à laquelle elle contribue : par exemple, lorsqu'un service est demandé de façon redondante, l'échec d'une demande peut ne pas être préjudiciable à la bonne réalisation du but poursuivi. Cependant, l'accumulation ou la co-occurence de telles exceptions mineures peut devenir critique. Pour gérer ces situations, [ISS 01] propose de gérer collectivement l'impact des exceptions propagées depuis les activités individuelles vers l'activité collective grâce à l'intégration d'un mécanisme de concertation des exceptions. Celui-ci filtre et historise les exceptions mineures et ne signale que les exceptions critiques ou le résultat de l'accumulation d'exceptions mineures. Les exceptions signalées par les activités individuelles ne sont pas prises en compte directement. Elles sont traitées par une fonction de concertation, associée à l'activité globale, qui a pour rôle de les agréger et de signaler, si nécessaire, une exception unique, appelée *exception concertée*, reflétant l'état global de l'activité collective ce qui permet le déclenchement d'un handler adapté à la situation globale de l'activité collective. Ainsi, la gestion d'exceptions concertées est la quatrième qualité que nous attendons d'un système de gestion des exceptions dans les collaborations entre agents.

4. SaGE : un SGE dédié aux SMAs

Cette section présente notre proposition, SaGE, un SGE dédié aux agents qui se conforme aux attentes identifiées dans la section précédente. Nous utilisons l'étude de cas classique d'une *Agence de voyage* comme illustration (*cf.* Figure 1). Dans cet exemple, un agent *Client* contacte un agent *Broker* dans le but d'établir un contrat de voyage par train ou avion. Suivant la requête de l'agent *Client*, l'agent *Broker* envoie des demandes de tarif aux agents prestataires de voyages en train ou en avion (les agents *Provider*). Enfin, l'agent *Broker* choisit la meilleure offre et met en relation l'agent *Client* et l'agent *Provider*, représentant le prestataire sélectionné, afin qu'ils établissent un contrat.

```
public class Broker extends SaGEAgent
{
   Service s = new Service ("Search for a travel", getAddress(), serviceID)
      {
      public void live ()
         {  // body of the service
         }
      public SaGEException concert (Vector subServInfo)
         {  // body of the exception resolution function
         }
      public void handle (NetworkException exc)
         {  // handler for NetworkException
         }
       public void handle (NoProviderException exc)
         {  // handler for NoProviderException
         }
      };
}
```

Figure 2. *Création d'un service et association de handlers à un service dans Sa*GE

4.1. *Gestion des activités collectives : Services et RôleAgents*

Dans SaGE, les agents sont capables d'exécuter concurremment la scrutation de leur boîte aux lettres et leur comportement interne. Cette capacité de «concurrence interne» leur permet de rester réactifs à l'arrivée de messages prioritaires (notamment, les exceptions).

4.1.1. *Services*

Lorsqu'un agent serveur accepte une requête, il lance l'exécution d'un service correspondant. Dans SaGE, chaque service est représenté par une entité qui posséde son propre thread d'exécution et définit ainsi un contexte d'exécution distinct. Un agent contrôle explicitement le cycle de vie de ses services. La figure 2 présente un exemple de création de service. Il existe deux catégories de services : les services atomiques dont l'exécution ne dépend pas d'autres services et les services complexes dont l'exécution requiert celle d'autres services. Sur la Figure 1, les services *Get price* des agents *Provider* sont atomiques et le service *Search for a travel* de l'agent *Broker* est un service complexe. Contrairement aux services atomiques, qui se suffisent d'un simple thread d'exécution, les services complexes ont besoin de disposer de la capacité d'envoyer et de recevoir des messages de manière asynchrone, à l'image des agents.
Les requêtes en cascade produisent des exécutions de services en cascade. Le graphe d'appel des services définit la structure logique qui relie les contextes d'exécution des différents services. La figure 1 montre un exemple de graphe d'appel de services dans l'exemple de l'agence de voyages. Cette structure, bien qu'arborescente, est comparable à la pile d'exécution utilisée dans la programmation séquentielle : elle fournit une représentation explicite des activités individuelles (services atomiques) et collectives (services complexes) des agents, ainsi que leur structuration.

4.1.2. *Gestion des requêtes collectives grâce aux RôleAgents*

Dans la section 2.2 nous avons introduit le concept de rôle qui permet la diffusion d'un message à un ensemble d'agents ayant des compétences communes. Afin de gérer de telles requêtes collectives, SaGE introduit les *RôleAgents* pour représenter et gérer les rôles. Les RôleAgents sont des agents spécialisés qui maintiennent la liste des agents qui jouent un rôle. Ils possèdent un comportement générique qui consiste à diffuser chaque requête reçue aux agents qui jouent le rôle, puis à collecter les réponses des agents pour les combiner en une réponse collective ou en une exception concertée. Les Services et les RôleAgents permettent de représenter les activités des agents. Comme indiqué dans la section 3.2.3, ils sont le support du système de gestion d'exceptions que nous proposons.

4.2. *Un SGE dédié aux SMA*

4.2.1. *Définition des handlers*

Dans SaGE, les handlers peuvent être associés aux services, aux agents et aux RôleAgents pour fournir des traitements d'exceptions à des unités du système de granularités différentes.

Les handlers associés à un service sont conçus pour traiter les exceptions qui pourraient être levées, directement ou indirectement, par l'exécution de ce service. Ceci permet la définition contextuelle de handlers : l'objectif du service concerné, son état courant et l'impact de l'exception sur son bon achêvement peuvent être gérés par de tels handlers.
Les handlers associés à un agent sont conçus pour permettre d'écrire un handler unique pour l'ensemble des services d'un agent. Par exemple, la mort d'agents ou le maintien de la cohérence des données spécifiques d'un agent peuvent être gérés grâce à de tels handlers.
Les handlers associé à un rôle sont conçus pour permettre de gérer les exceptions qui concernent l'intégralité des agents remplissant un rôle donné. Par exemple, si des exceptions surviennent en réponse à une requête collective, des résultats partiels ou des statistiques de QoS peuvent être envoyées par de tels handlers.
Tous ces handlers sont spécifiques au système SaGE et invoqués lors de la réception d'un message signalant une exception à un agent. Classiquement, ces handlers définissentt l'ensemble des types d'exceptions qu'il peuvent traiter et les traitements associés (comme illustré sur la figure 2). SaGE est un SGE à terminaison qui permet à un handler d'exécuter des traitements pour gérer les conséquences d'une interruption abrupte de l'exécution d'un service (comme remettre l'agent dans un état cohérent, envoyer des résultats partiels, etc.), de signaler à son tour une nouvelle exception notifiant qu'il n'a pas réussi à traiter l'exception originelle, et, s'il est associé à un service, de relancer l'exécution de celui-ci après avoir éventuellement modifié des éléments du contexte d'exécution.

```
signal (new SaGEException ("Bad client address", getOwnerAddress ());
```

Figure 3. *Levée d'exception dans Sa*GE

4.2.2. *Levée d'exception*

Les exceptions de SaGE sont levées, durant l'exécution de services ou de handlers, au moyen d'une primitive dédiée (*cf.* Figure 3) qui prend en paramètre l'exception à traiter. L'invocation de cette primitive déclenche le mécanisme de signalement de SaGE.

4.2.3. *Recherche de handler*

Le cœur d'un SGE est sa politique de recherche de handlers. Quand une exception est levée, l'exécution du service défaillant est suspendue. Un handler approprié est recherché d'abord localement, c'est-à-dire dans la liste des handlers associés au service. Si un tel handler est trouvé, il est exécuté. Sinon, la recherche se poursuit parmi les handlers associés à l'agent qui exécute le service. Dans tout les cas, le service défaillant se termine.
Si aucun handler n'est trouvé ainsi, cela signifie que le service a échoué et que les conséquences de cet échec doivent être gérées par le client du service. La recherche se poursuit donc dans l'agent qui a requis le service. Un message signalant l'exception est envoyé à l'agent en réponse à sa requête. Le traitement de ce message est pris en charge par le SGE qui interrompt l'exécution du service complexe qui a émis la requête et reprend la recherche d'un handler au niveau de ce service. Si l'agent client est un *RôleAgent*, les handlers sont directement cherchés au niveau de l'agent car le programmeur n'a pas accès au service de diffusion et ne peut donc pas y associer de handlers. Le processus de recherche est réitéré (*cf.* Figure 4) jusqu'à ce qu'un handler approprié soit trouvé ou qu'une racine du graphe d'appel des services soit atteinte. Ce dernier cas correspond à la terminaison d'une activité initiée spontanément par un agent et non pour répondre à une requête. La terminaison d'un service complexe, suite au signalement d'une exception, entraîne automatiquement la terminaison de tous les services qu'il a demandé. Le graphe des appels est alors utilisé pour retrouver l'ensemble des services demandé par un service complexe.

4.2.4. *Concertation d'exceptions*

Inspiré par [ISS 01] et [LAC 90], SaGE intègre la gestion des exceptions concertées dans son mécanisme de signalement afin de ne pas réagir face à des exceptions non critiques et de collecter les exceptions signalées pour mieux refléter les problèmes collectifs ou globaux. Le mécanisme de concertation est disponible au niveau des services et des rôles car il n'est pas pertinent au niveau des agents. En effet, l'association d'un handler à un agent revient à l'associer à l'ensemble de ses services. Les exceptions pouvant déclencher l'exécution de ce handler sont, si nécessaire, déjà concertées au niveau des services, par les fonctions de concertation qui leur sont associées.

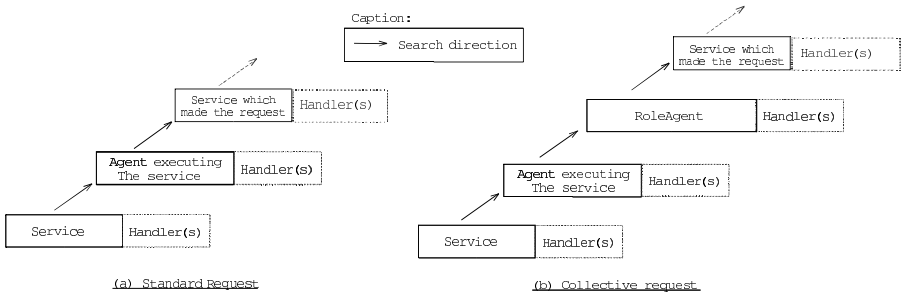

Figure 4. *Processus de recherche de handler dans Sa*GE

Concertation d'exceptions au niveau des services. Une exception signalée par un service n'est pas toujours fatale pour son exécution. C'est le cas quand un *Provider-RoleAgent* envoie n requêtes à n prestataires : la défaillance de quelques uns n'est pas critique. Seule la défaillance d'un ratio suffisant d'entre eux est préjudiciable.
Pour permettre la concertation, les exceptions signalées à un service ne sont pas directement traitées. Elles sont stockées dans un log associé à ce dernier pour maintenir un historique des exceptions signalées. Chaque fois qu'une nouvelle exception survient, la fonction de concertation associée au service est exécutée pour évaluer la situation et jouer le rôle de filtre et de fonction de composition. Au regard du nombre et de la nature des exceptions stockées, cette fonction détermine s'il est nécessaire de signaler une exception concertée ou non. Si c'est le cas, il peut s'agir de la dernière exception signalée (si elle est critique) ou une nouvelle exception (concertée) «composée» à partir des exceptions stockées dont la conjonction révèle d'une situation globale critique.

Concertation d'exceptions au niveau des rôles. L'ensemble des requêtes émises par un RôleAgent pour gérer une requête collective est transparent pour l'agent client. Le RôleAgent sert de collecteur de réponses et retourne une réponse composite unique au client. Ce comportement «normal» est transposé à la gestion d'exceptions au niveau du rôle : Lorsqu'une exception est signalée au RôleAgent, la fonction de concertation associée au RôleAgent est invoquée, l'exception est mémorisée et, lorsque cela est nécessaire au regard du nombre et de la nature des exceptions stockées, la fonction de concertation calcule et signale une exception concertée. Dans l'exemple de l'agence de voyage, il y a des situations où il est nécessaire de mettre en œuvre des exceptions concertées. Quand l'agent *Broker* diffuse des demandes de tarif aux agents *Provider* par le biais du RôleAgent *ProviderRole*, aucune de ces requêtes n'est critique individuellement. Il faut donc définir une fonction de concertation associée au RôleAgent *ProviderRole* (*cf.* Figure 5) qui collecte les exceptions signalées sans réagir jusqu'à ce qu'une certaine proportion de ces agents ait échoué. Inversement, des services comme *Select an offer* ou *Contact parties* sont critiques pour leur service client *Search for a travel* : leur échec entraîne immédiatement l'échec du service client.

```
public SaGEException concert (Vector subServInfo)
{
   int failed = 0, pending = 0;

   // count exceptions raised in subservices and subservices still running
   for (int i=0; j<subServInfo.size (); i++)
      {
      if ((ServiceInfo) (subServInfo.elementAt (i)).getRaisedException () != null) failed++;
      if ((ServiceInfo) (subServInfo.elementAt (i)).isFinished () == false) pending++;
      }

   // if more than 30% failed, there are two many bad providers
   if (failed > (0.3*subServInfo.size()))
      return new SaGEException("too_many_bad_providers",getAddress());

   // if not, at the end, only few providers failed
   if (failed != 0 && pending == 0)
      return new SaGEException("few_bad_providers",getAddress());

   // else, computing still running - no critical situation
   return null;
}
```

Figure 5. *Fonction de concertation associée au TravelProviders_RoleAgent*

5. Implémentation de SaGE et expérimentation

L'implémentation de SaGE[2] est réalisée dans le SMA MADKit [MAD], lui-même programmé en Java. Elle a consisté en la spécialisation de concepts de MADKit (agents, messages), en l'ajout de nouveaux concepts (services, RôleAgents) et en la spécialisation de concepts java (exceptions).

L'exemple de l'agence de voyage est mis en œuvre dans SaGE à l'aide de trois types d'agents (implémentés comme des sous-classes de *SaGEAgent*) : *Client*, *Broker* et *Provider*. Les RôleAgents *PlaneProvidersRoleAgent* et *TrainProvidersRoleAgent*, implémentés comme des sous-classes de la classe *RoleAgent* de SaGE, gèrent les requêtes collectives adressées aux agents *Provider* jouant les rôles correspondants. Ces RôleAgents fournissent les fonctions de concertation (*cf.* Figure 5) et les handlers (*cf.* Figure 6) nécessaires à la gestion des exceptions au sein de ces rôles. Ainsi, si un nombre limité d'agents *Provider* ne fournissent pas d'offre mais signalent un échec en levant une exception, l'exécution de l'activité collective est maintenue car elle peut continuer à produire des résultats partiels mais exploitables. Lorsque trop d'agents *Provider* échouent, l'accumulation des exceptions provoque le signalement immédiat d'une exception concertée indiquant qu'il n'est plus possible de fournir de résultat exploitable. Dans ce dernier cas, le signalement de l'exception concertée permet de procéder au plus tôt à la terminaison du service gérant la requête collective au niveau du *TravelProviderRoleAgent*. La terminaison du service entraîne la terminaison de tous les sous-services invoqués (*cf.* section 4.1) en cours d'exécution, en l'occurence répartis sur des agents *Provider* appartenant au rôle et qui continuent à exécuter une tâche devenue inutile. Les exceptions signalées par les agents *Provider* sont propa-

2. http ://www.lgi2p.ema.fr/~fsouchon/sage_applet/sage_applet.html

```
public void broadcasthandle (SaGEException exc, BroadcastService bs)
{
   if (exc.getMessage ().equals ("few_bad_providers"))
   {
      bs.setalive (true);
      bs.broadcastavailablereponses ();
   }
   else
   {
      sendMessage (bs.getparentowneraddress (), new SaGEMesage ("exception",
                   exc.getMessage (), bs.getrequestID ()));
      bs.terminate ();
   }
}
```

Figure 6. *Handler associé au TravelProviders_RoleAgent*

gées au *TrainProviderRoleAgent* auquel ils sont associés. La fonction de concertation du *TrainProviderRoleAgent* signale à son tour une exception concertée qui est propagée à l'agent *Broker* ayant envoyé la demande de service. Le signalement de cette exception entraîne la terminaison du service appelant *Poll Providers* de l'agent *Broker*, ce qui entraîne la terminaison de l'autre sous-service, appelé *Contact parties*.

6. Conclusion et travaux futurs

Dans cet article, nous avons présenté le système de gestion d'exceptions SaGE que nous proposons pour renforcer la fiabilité des systèmes multi-agents en permettant de gérer explicitement le signalement et le traitement des erreurs qui surviennent lors des collaborations entre agents. SaGE s'appuie sur l'idée générale qu'une défaillance qui survient lors de l'exécution d'un service demandé par un agent à un autre agent doit être signalée à l'agent demandeur par une exception. SaGE se distingue des travaux existants car sa gestion d'exception ne s'appuie pas sur des entités extérieures aux agents et permet en cela aux agents de conserver la maîtrise de leur comportement, y compris lors des situations exceptionnelles. Dans SaGE, les handlers peuvent être associés à des entités de granularités différentes – services, agents et rôles – afin de permettre la gestion d'exceptions dans différents contextes (celui d'une collaboration, celui d'un agent ou celui d'un ensemble d'agents). De plus, dans SaGE, les exceptions signalées individuellement par des agents contribuant à une même activité collective peuvent être concertées en une exception unique, plus pertinente au regard de la gestion de cette activité globale. Nous avons implémenté et expérimenté ce système dans MADKit. Diverses perspectives sont envisagées pour compléter ces travaux, telle que l'étude d'un modèle de gestion d'exceptions avec reprise (qui permettrait de reprendre l'exécution d'un service interrompu par le signalement d'une exception après en avoir modifié des paramètres) ou l'intégration d'un système de gestion de transactions (qui permettrait de restaurer l'intégrité de l'état d'un agent après le traitement infructueux d'une exception). De plus, nous étudions l'adaptation de ces travaux aux composants orientés messages dans les plateformes de développement à base de composants [SOU 03].

7. Bibliographie

[DON 90] DONY C., « Exception handling and object-oriented programming : towards a synthesis », *ACM SIGPLAN Notices*, vol. 25, n° 10, 1990, p. 322–330, *OOPSLA/ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[FER 98] FERBER J., GUTKNECHT O., « A meta-model for the analysis and design of organizations in multi-agent systems », *Proceedings of the 3rd international conference on multi-agent systems*, 1998, p. 128–135.

[GOO 75] GOODENOUGH J. B., « Exception handling : issues and a proposed notation », *Communications of the ACM*, vol. 18, n° 12, 1975, p. 683–696.

[ISS 01] ISSARNY V., « Concurrent exception handling », Romanovsky et al. [ROM 01b].

[KLE 03] KLEIN M., DELLAROCAS C., « Using domain-independent exception handling services to enable robust open multi-agent systems : the case of agent death », *Journal for autonomous agents and multi-agent systems*, vol. 7, n° 1/2, 2003.

[KNU 01] KNUDSEN J. L., « Fault Tolerance and Exception Handling in BETA », Romanovsky et al. [ROM 01b].

[KOE 89] KOENIG A. R., STROUSTRUP B., « Exception Handling for C++ », *Proceedings of the C++ at Work conference*, 1989, p. 322–330.

[LAC 90] LACOURTE S., « Exceptions in Guide, an object-oriented language for distributed applications », *ECOOP 91*, n° 5-90 LNCS, Grenoble, France, 1990, Springer, p. 268–287.

[MAD] MADKIT, http ://www.madkit.org.

[MEY 88] MEYER B., « Disciplined exceptions », Technical report tr-ei-22/ex, 1988, Interactive Software Engineering, Goleta, CA.

[RIC 00] RICORDEL P.-M., DEMAZEAU Y., « From analysis to deployment : a multi-agent platform survey », *Engineering societies in the agents world*, vol. 1972 de *LNAI*, Springer, 2000, p. 93–105.

[ROM 01a] ROMANOVKSY A., KIENZLE J., « Action-oriented exception handling in cooperative and competitive object-oriented systems », Romanovsky et al. [ROM 01b].

[ROM 01b] ROMANOVSKY A., DONY C., KNUDSEN J. L., TRIPATHI A., Eds., *Advances in Exception Handling Techniques*, n° 2022 LNCS, Springer, 2001.

[SOU 03] SOUCHON F., URTADO C., VAUTTIER S., DONY C., « Exception handling in component-based systems : a first study. », ROMANOVSKY A., DONY C., KNUDSEN J., TRIPATHI A., Eds., *Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003*, p. 84–91, Darmstadt, Germany, July 2003.

[SZY 98] SZYPERSKI C., *Component software : beyond object-oriented programming*, ACM Press and Addison-Wesley, New York, NY, 1998.

[TRI 01] TRIPATHI A., MILLER R., « Exception handling in agent oriented systems », Romanovsky et al. [ROM 01b].

[WEI 83] WEINREB D. L., « Signalling and Handling Conditions », Technical report, 1983, Symbolics, Inc., Cambridge, MA, USA.

[WOO 99] WOOLDRIDGE M., CIANCARINI P., *Agent-oriented software engineering*, Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing Company, 1999.