SaGE : Une Proposition pour la Gestion d'Exceptions dans les Systèmes Multi-Agents

Frédéric Souchon, Christophe Dony, Christelle Urtado, Sylvain Vauttier, Jacques Ferber

SaGE : une proposition pour la gestion d'exceptions dans les systèmes multi-agents
Rapport Interne - LIRMM - 02205

Souchon Frédéric* **, Christophe Dony*, Christelle Urtado**,
Sylvain Vauttier**, Jacques Ferber*

19 décembre 2002

(*) L.I.R.M.M.
{dony, ferber,fsouchon}@lirmm.fr

(**) LGI2P - Ecole des Mines d'Alès - Site EERIE
{Frederic.Souchon, Christelle.Urtado, Sylvain.Vauttier}@ema.fr

Résumé

L'émergence de nouvelles architectures logicielles, plus complexes que celles communément développées jusqu'alors, comme les systèmes multi-agents (SMA) ou les architectures à base de composants, distribués, communiquant éventuellement de façon asynchrone, amène à se poser de façon radicalement nouvelle les problèmes de la sécurité et de la fiabilité des applications. Cet article, dans une première étape, s'intéresse à la problématique de la gestion des exceptions dans le contexte du développement de systèmes multi-agents avec diverses contraintes, dont la présence d'asynchronisme, que ne prennent pas parfaitement en compte les systèmes de gestion d'exceptions (SGE) des SMA existants. Nous proposons un modèle de SGE dédié à un SMA qui propose des solutions à un ensemble de problèmes que nous recensons : il s'appuie notamment sur des travaux proposant des solutions pour la gestion d'exceptions en milieu concurrent. Ce modèle a été expérimenté dans le SMA MADKit.

1 Introduction

Les nouvelles architectures logicielles (systèmes multi-agents [Fer95] ou architectures à base de composants logiciels [Bri00]) reposent sur l'intégration d'un grand nombre d'entités logicielles assemblées selon des modèles complexes, pouvant s'exécuter de manière concurrente, communiquer de manière asynchrone [CR86, Gär99] ou encore se déplacer (code mobile). Nous nous intéressons de façon globale aux problèmes de fiabilité dans ces nouveaux contextes et nos premiers travaux ont porté sur les problèmes que pose la gestion des exceptions dans les systèmes multi-agents. Ces problèmes sont relatifs au caractère cognitif des agents (par exemple rupture de contrat entre agents) ou à leur autonomie et à leur mode de communication asynchrone (agent ne répondant pas ou répondant hors délai...). Si différentes études [KrA00, KD00] proposent un recensement et une taxonomie des exceptions liées à ces systèmes, cette problématique pourtant fondamentale n'a pas encore été suffisamment prise en compte par la communauté multi-agent, comme le montre l'absence de proposition de normalisation FIPA dans ce domaine.

Les systèmes de gestion d'exceptions des Systèmes multi-agents (SMA) actuels posent au moins l'un des deux problèmes suivants.

- Soit les possibilités de gestion d'exceptions reposent sur celles du langage d'implantation sous-jacent [RD01, DF94] (c'est notamment le cas de *MADKit* [MAD]) et il s'avère alors difficile de gérer correctement l'asynchronisme des communications (mauvaise gestion de

l'autonomie des agents). Ces systèmes n'intègrent par ailleurs pas un ensemble de résultats de recherche permettant de mieux gérer les exceptions en milieu concurrent.

- Soit il existe des systèmes dédiés de gestion d'exceptions. Ceux-ci sont alors basés sur des modèles à base de *superviseurs* (agents spécialisés dans la gestion d'exceptions), issus de propositions pour les systèmes d'acteurs [The83, IY91] intégrant des modèles de programmation par continuation avec lesquels il est possible de spécifier, pour tout envoi de message, quel est l'acteur à qui envoyer le résultat calculé et quel est celui à qui envoyer une information d'exception. Le problème fondamental des superviseurs est qu'ils n'ont pas connaissance du contexte ayant généré la situation exceptionnelle et ne peuvent mettre en œuvre que des traitements très génériques (affichage de message...) au lieu de s'adapter au mieux aux objectifs propres à chaque agent (non respect du caractère cognitif).

Dans cet article, nous proposons un modèle de SGE dédié à la programmation SMA, dont la mise en œuvre a été réalisée dans la plate-forme MADKit [MAD] et répondant aux deux impératifs précédemment évoqués (respect du caractère cognitif et de l'autonomie-asynchronisme des agents). Le SGE proposé est le fruit d'une réflexion globale sur ces points[RK00] et d'une adaptation de plusieurs propositions pour SGE concurrents [Lac90, Iss01].

La section 2 rappelle des définitions et idées de base relatives à la gestion des exceptions. La section 3 étudie des travaux traitant de l'impact de la concurrence sur la gestion d'exceptions puis fait une synthèse sur les systèmes dédiés de gestion d'exceptions dans les SMA. Notre proposition (le modèle SaGE) est présentée dans la section 4 qui contient également une mise en lumière des ses possibilités au travers d'un exemple concret.

2 Gestion d'exceptions

Les évènements exceptionnels (ou exceptions [Goo75]) sont les situations qui empêchent la poursuite de l'exécution standard d'un programme (par exemple, divisions par zéro, dépassements de pile...). Lors de l'occurrence d'une exception, les logiciels fiables sont capables de réagir de manière appropriée afin de reprendre leur exécution ou, tout du moins, de s'interrompre en préservant au mieux l'intégrité des données. Un *système de gestion d'exceptions* (SGE) [Goo75, Lis88, Mey88, Don90, Ano79, Wei83, KS89] offre les structures de contrôle qui permettent au programmeur de définir de tels comportements de substitution au comportement standard. Ceci se matérialise par la possibilité de signaler des exceptions, de définir des continuations exceptionnelles (handlers), et, au sein d'un handler, de replacer le système dans un état cohérent : soit en reprenant l'exécution là où elle s'était interrompue, après modification des données du problème (*reprise)*, soit en abandonnant une partie du calcul et en reprenant l'exécution à un point fiable (*terminaison*), soit en déléguant le traitement de l'exception par le signalement d'une nouvelle exception.

Signaler une exception provoque l'interruption de l'exécution en cours et la recherche d'un *handler* adapté à l'exception. Définir un handler (pour un ou plusieurs types d'exceptions) revient à associer du code à des *unités de programmes* (code source d'une procédure ou définition d'une classe, par exemple). Suite au signalement d'une exception, un handler adapté est recherché parmi ceux associés à l'unité de programme en cours d'exécution. S'il est trouvé, il est invoqué, sinon la recherche se poursuit dans l'unité de programme englobante, et ainsi de suite. L'ensemble des unités de programme dont les exceptions peuvent être traitées par un handler est appelé portée du handler.

Selon les modèles, cette portée peut être déterminée de façon *statique* ou *dynamique :*

- La portée d'un handler est statique lorsque la recherche de l'unité de programme englobante est déterminée par la structure lexicale du code. C'est par exemple le cas pour un handler associé à un bloc de code qui peut traiter les exceptions signalées dans tous les blocs imbriqués ou pour un handler associé à une classe qui peut traiter les exceptions signalées dans toutes les méthodes de cette classe. Ce type de fonctionnement est, par exemple, celui du langage Smalltalk originel.

- La portée d'un handler est dynamique lorsque la recherche de l'unité de programme englobante est déterminée, en cours d'exécution, par l'*historique des contextes d'exécution*. Cet historique se construit dynamiquement lors de l'activation d'une unité de programme (l'appelée) par une autre (l'appelante). Un *contexte d'exécution* est alors créé et associé à l'unité appelée ; il contient les informations relatives à son exécution (données locales, adresse de retour, paramètres...). Ainsi, l'unité de programme englobant l'unité en cours d'exécution est son unité appelante : elle la précède dans l'historique des contextes d'exécution. Par exemple, la portée d'un handler associé à une procédure couvre l'exécution de cette procédure et celles toutes les procédures qu'elle a appelées et ainsi de suite. Ce type de fonctionnement est, par exemple, celui des langages C++ et Java.

Indépendamment de la politique de définition de handlers et de signalement du langage considéré, il y a un accord assez large sur les points suivants, en rapport avec l'idée de contrat logiciel [Mey88], qui seront importants pour la suite de notre discours :

- toute exécution est réalisée dans le but de rendre un service en réponse à une requête d'un client (une exécution de procédure peut ainsi être vue comme un service rendu à l'appelant),
- lors du signalement d'une exception durant la réalisation d'un service, le client demandeur du service est l'entité qui a connaissance de l'*intention* et qui est la mieux placée pour savoir comment traiter la situation exceptionnelle,
- le traitement efficace de l'exception par le client peut nécessiter la connaissance et donc la possibilité d'accès, au contexte d'exécution du client (données locales, adresse de retour i.e. client du client, paramètres...).

Dans la suite de cet article, nous serons amenés à considérer les cas où plusieurs processus concurrents réalisent une tâche en collaboration pour un unique client et la distinction entre les systèmes permettant un accès au contexte d'exécution du client et ceux ne le permettant pas.

3 Problématiques et travaux existants

Nous étudions dans cette section quatre problèmes que pose notre contexte d'étude et examinons les réponses qu'y apportent les modèles existants. Le premier est relatif à la nécessité de disposer de système de gestion d'exceptions dédiés, les deux suivants sont liés à l'asynchronisme (coordination, collaboration), et le quatrième au pouvoir d'expression.

3.1 Gestion d'exceptions et paradigme "agent"

Est-il nécessaire de gérer les exceptions lorsque l'on programme un système à base d'agents ? Comme nous l'avons dit, l'absence de proposition de normalisation FIPA [FIP97] dans le domaine de la gestion des exceptions montre que la problématique de la tolérance aux pannes n'est encore jugée suffisamment importante. Pourtant, comme tout système informatique, les systèmes multi-agents (SMAs) posent ainsi des problèmes spécifiques relatifs à la gestion des situations exceptionnelles : que faire lorsqu'un agent ne répond pas, ou répond hors délai, lorsqu'il n'accomplit pas ce qu'il est censé faire, lorsqu'il tombe en panne. On peut étendre le problème à la rupture de contrats entre agents, etc... D'une manière générale, les problèmes de fiabilité, de sécurité et de robustesse devraient être au cœur du développement de tels systèmes et justifier par eux-mêmes le développement de systèmes de gestion des exceptions dédiés.

Un SMA est aussi une couche logicielle qui permet de développer des applications suivant un paradigme propre aux agents et qui assure l'autonomie des agents en gérant, de manière transparente, leur concurrence et l'asynchronisme de leurs communications[1]. La question de savoir si cette gestion doit demeurer transparente en présence de situation exceptionnelle est traitée dans la section suivante.

Parmi les systèmes à agents existants, deux [TM00, KD01] proposent des systèmes de gestion d'exception spécifiques, nous en reparlerons en section 3.3. D'autres, extension de langages

[1] ainsi que leur distribution, point que nous n'abordons pas dans cet article

existants, permettent l'utilisation du système de gestion des exceptions sous-jacent. Par exemple, beaucoup de SMA actuels [RD01, Ret99, BRHL99] sont développés en Java et ne possèdent pas de SGE dédié. Les seuls traitements d'exceptions possibles sont alors ceux permis par Java. Du point de vue de Java, chaque agent est à la base un processus léger (*thread*). Une exception signalée par un agent peut uniquement être traitées localement à ce processus. Si aucun handler local adapté n'est trouvé, elle est propagée et traitée au niveau de la machine virtuelle qui interrompt alors l'exécution du processus. Du point de vue du SMA, la destruction du processus Java correspond à la mort accidentelle d'un agent. Mais, en l'absence de SGE dédié, aucune exception informant de la mort de l'agent, ne sera signalée au niveau du SMA. Plus globalement, la seule solution permettant à un agent de signaler une défaillance au demandeur d'un service est de lui envoyer une réponse avec une valeur particulière, que le demandeur devra être à même de distinguer d'une réponse possible à la demande de service. Cela revient donc à renoncer aux « bonnes pratiques » qu'un SGE permet d'adopter pour en revenir à des solutions *ad hoc* [Goo75, Mey88]. Cette solution est d'autant moins satisfaisante qu'un SMA est un système ouvert qui doit pouvoir intégrer dynamiquement de nouveaux agents développés séparément (par exemple des agents mobiles) : il est hautement improbable que les agents aient alors une interprétation concordante des valeurs spéciales.

Pour cet ensemble de raisons, le développement d'un SGE spécifique est indispensable à la gestion d'exception dans les SMA.

3.2 Gestion d'exceptions et asynchronisme

Un SMA repose sur des communications asynchrones entre un ensemble d'entités autonomes (les agents). Gérer les exceptions aux sein d'un SMA suppose donc en premier lieu de considérer les problèmes que pose la concurrence associée à ce mode de communication.

3.2.1 Coordination d'activités

Un SMA est constitué par un ensemble d'agents travaillant de manière autonome mais collaborant parfois, sans en avoir nécessairement conscience, pour la réalisation d'une tâche globale. Des agents initiateurs de cette tâche globale font appel de façon asynchrone à des agents prestataires de services indépendants et s'exécutant de manière concurrente. Lorsque tout se passe bien les prestataires font connaître leurs réponses aux demandeurs.

[RK00] a classifié les types de concurrence et a étudié leur impact sur la gestion des exceptions. Il qualifie de *c*oncurrence disjointe celle dans laquelle plusieurs entités s'exécutent concurremment sans qu'il y ait une possibilité de contrôle global de leur exécution et de *concurrence coopérative* celle où une telle possibilité de contrôle existe. Dans le cas des SMA, nous avons affaire par défaut à de la concurrence disjointe. [RK00] et [Iss01] montrent que la concurrence disjointe est, dans un grand nombre de cas, incompatible avec une gestion efficace des exceptions. En particulier, l'occurrence d'une exception dans une entité en cours d'exécution doit pouvoir affecter l'exécution d'autres entités concurrentes dans le cas où ces entités participent à la réalisation d'une même tâche globale. Par exemple, dans un SMA pour la gestion d'achats en ligne, il est nécessaire qu'une exception interrompant l'action d'un agent chargé de réaliser un paiement sécurisé affecte l'action de l'agent qui effectue conjointement la réservation du produit commandé. Pour passer d'une concurrence disjointe à une forme de concurrence coopérative, nécessaire à un contrôle fin des situations exceptionnelles, il est préconisé d'utiliser des unités de structuration de l'exécution représentant et gérant les actions globales auxquelles participe un ensemble d'entités concurrentes [RK00]. Gérer la coordination lors du traitement d'une exception revient alors à pouvoir associer des handlers aux unités représentant ces actions globales et à permettre la propagation d'exceptions depuis les unités participantes vers les unités représentant les actions auxquelles elles participent.

Les SMA existants, dotés de modèles de gestion d'exceptions à base de superviseurs, permettent une forme de coordination : Il est possible au programmeur d'associer, au moment de l'envoi d'une requête, un même handler (un superviseur d'exceptions) à tous les agents traitant globalement une même tâche et ce pour la durée de la réalisation de cette tâche. Les limitations liées à ces

modèles sont relatives à la contextualisation, nous en parlons en section 3.3. Pour les SMA non dotés de SGE dédiés et dont les possibilités de gestion d'exception reposent éventuellement sur les systèmes des langages d'implantation sous-jacents, les possibilités peuvent exister mais sont généralement insuffisantes. Nous avons vu en section 3.1 qu'en Java, pour prendre un exemple, la concurrence utilise le mécanisme des processus légers (*threads*) qui, par défaut, ne permettent que la concurrence disjointe.

C'est pour pallier aux limitations que nous avons évoquées et pour permettre une gestion coordonnée des processus qu'ont été créés les groupes de processus (les *ThreadGroups*). Tout processus peut, lors de sa création, être rattaché à un groupe de processus. Toute exception levée et non traitée par un processus associé à un groupe est propagée vers ce dernier. Il est donc possible de rattraper et de traiter cette exception au niveau du groupe, et donc, par exemple, d'intervenir sur l'exécution d'autres processus du groupe. Les groupes peuvent eux-même être regroupés et une hiérarchie de groupes peut être définie pour contrôler différents niveaux d'imbrication d'exécutions concurrentes. Si ces mécanismes représentent une base intéressante, nous allons voir qu'ils ne suffisent pas en tant que tels à fournir un véritable SGE pour un SMA.

3.2.2 Concertation

Par défaut, les langages permettant une coordination de l'exécution d'entités concurrentes fournissent des mécanismes permettant à l'entité responsable d'une action globale d'être informée de l'ensemble des exceptions qui peuvent survenir dans les entités qui participent à son exécution. Des systèmes spécifiques de gestion des exceptions ont été bâtis pour ces langages. Le modèle de [Iss01] fait une synthèse d'un ensemble de travaux relatifs à cette question et utilise notamment un mécanisme de concertation.

Ce modèle de gestion d'exceptions est proposé dans le contexte d'un langage intégrant la notion de *multi-procédure* (procédure décrite par un ensemble de sous-procédures qui sont exécutées concurremment) [BMP86]. Le concept de concertation est le suivant : les exceptions signalées à une multi-procédure par ses sous-procédures sont traitées par une *fonction de résolution* qui doit éventuellement en déduire une exception unique propagée en lieu et place de toute autre exception et reflétant l'état global de l'exécution des sous-procédures (*exception concertée*).

Un tableau est associé à chaque multi-procédure avec une entrée par sous-procédure. A chaque signalement d'exception par une sous-procédure, l'exception signalée est stockée en bonne place dans le tableau et la fonction de résolution est exécutée. Elle détermine, au vu de l'état du tableau, si une exception doit effectivement être signalée et si oui la retourne. Si non, l'exécution des sous-procédures n'ayant pas signalée d'exceptions se poursuit. Si oui, l'exception retournée est effectivement signalée. Il s'agit soit de celle initialement signalée par une des sous-procédures (elle est suffisamment importante pour influencer l'exécution globale de la multi-procédure), soit une nouvelle exception dite *concertée*. Un tel mécanisme de concertation permet de définir simplement différentes politiques de gestion d'exception en présence de concurrence, traite un problème posé par l'asynchronisme et semble indispensable à la réalisation d'un SGE suffisamment souple et générique pour couvrir un large éventail de situations.

Cette idée de signalement concerté est une extension d'un modèle antérieur [Lac90] dans lequel le signalement effectif d'une exception pour un groupe de processus (équivalent d'une multi-procédure) n'est pas décidé par le résultat d'une fonction de résolution mais par l'examen d'une expression logique spécifiée par le programmeur. Cette expression combine des variables booléennes (une par processus fils) reflétant l'état de chacun des processus fils. Cette variable vaut vrai si le processus s'est terminé de façon standard et faux sinon. Le programmeur peut ainsi définir une condition de terminaison valide de l'exécution du processus père en fonction de l'état de ses fils (à l'aide des variables présentées ci-dessus). Lorsqu'un processus fils se termine normalement, si cette condition est vérifiée, les autres processus fils sont interrompus et l'exécution du père se termine normalement (la terminaison normale de ce processus fils a suffit à satisfaire les besoins du processus père). Par contre, si la terminaison exceptionnelle d'un processus fils empêche définitivement la condition d'être vérifiée (la terminaison normale de ce processus fils est indispensable au bon déroulement de l'exécution du père), l'exécution concurrente est interrompue.

Cette idée de fournir une politique standard de signalement paramétrée par une expression logique nous semble complémentaire du modèle de Issarny. Une combinaison équilibrée des deux systèmes précédents nous parait présenter une alternative intéressante qui est intégrée à notre proposition.

3.3 Contrat logiciel et Contextualisation

Les derniers problèmes qu'il est nécessaire d'évoquer sont relatifs aux notions de contrat logiciel et de pouvoir d'expression. La notion, maintenant classique, de contrat logiciel [Mey88] suggère qu'une entité logicielle puisse répondre à une invocation d'un client quoi qu'il arrive durant son exécution soit par une réponse normale soit par une réponse exceptionnelle. Ceci suppose qu'elle soit elle-même capable de rattraper toute exception qu'elle pourrait signaler indirectement. Pour cela il faut que le signalement des exceptions respecte l'historique de l'exécution.

Secondement, par pouvoir d'expression, nous parlons de ce qu'il est possible d'exprimer dans un handler. La puissance d'expression offerte par un SGE tient en premier lieu en sa capacité à permettre le signalement d'exceptions variées, riches en information sur l'origine d'une défaillance[2] et en second lieu en la possibilité de définir des handlers capables d'apporter des réponses pertinentes. Comme nous l'avons dit en section 2, la pertinence du traitement est grandement liée à la connaissance de l'intention du demandeur du service ayant provoqué la défaillance. En d'autres termes, traiter efficacement une exception suppose avoir accès au contexte d'exécution dans lequel se situe l'appel au service fautif. Connaître l'intention permet d'y substituer éventuellement un comportement de remplacement. En effet, que faire en présence d'une "division par zéro" si l'on ignore pourquoi et dans quel contexte la division a été demandée.

Les deux SGE spécifiques aux SMA que nous avons identifiés [TM00, KD01] répondent mal à ces deux exigences.

Pour gérer les exceptions, [TM00] introduit le concept des agents *superviseurs* tenant le rôle de gestionnaires d'exceptions pour un ensemble d'agents du système. Les exceptions survenant pendant l'exécution d'un comportement d'un agent peuvent être *internes* à l'agent (si celui-ci est capable de les traiter lui-même) ou *externes*. Pour pouvoir être traitée, toute exception externe est propagée vers le superviseur associé à l'agent. Les superviseurs jouent donc un rôle de coordinateurs centraux de la gestion d'exceptions. Ils disposent à cette fin d'un accès global au système et à l'état interne (état mental) des agents pour effectuer les traitements et peuvent interrompre ou modifier l'exécution des agents qu'ils surveillent. [KD01] propose une variante de ce modèle dédiée à la tolérance à la mort des agents. Chaque agent, à son entrée dans un SMA, se voit affecter un superviseur, appelé ici *agent sentinelle*, qui achemine tout message adressé à l'agent ou sortant de l'agent. Le SGE proposé contrôle régulièrement la présence de chaque agent afin que toute mort d'agent soit notifiée, de manière globale, à l'ensemble des agents sentinelles, au travers d'une base de données qu'ils partagent[3].

Avec ces deux modèles, le traitement des exceptions revient à un agent générique qui ne peut accéder au contexte d'exécution de l'entité ayant demandé le service qui a signalé l'exception. Les agents superviseurs mettent ainsi généralement en oeuvre des traitements très génériques. La *possibilité de contextualiser le traitement des exceptions*, s'est imposée dans tous les systèmes de gestion d'exceptions des langages impératifs et à objets [Don90] ; c'est une qualité qui conditionne la puissance du SGE en matière de traitement spécifique des exceptions. Elle doit être conservée dans la réalisation d'un SGE dédié aux SMA.

Par ailleurs, le modèle à superviseur ne permet pas le respect des contrats logiciels. En effet, le demandeur d'un service ne peut récupérer le contrôle si l'exécution de ce service signale une exception.

[2] Le mécanisme de concertation présenté précédemment, peut être considéré comme un moyen de garantir la pertinence des exceptions qui sont signalées en milieu concurrent (des exceptions isolées peuvent être agrégées afin de refléter l'état global d'un système).

[3] Accessoirement, une critique que nous pouvons formuler sur ce système, relativement à la gestion d'exceptions, est que seules des exceptions de type général, concernant la mort des agents, sont prises en compte.

4 SaGE : une proposition de Système dédié aux Agents de Gestion des Exceptions

Le modèle décrit dans cette section propose des solutions aux pré-requis posés dans les sections précédentes dans un cadre de programmation par agents : gestion coopérative et concertée des exceptions signalées par des activités en concurrence, possibilité de définir des réponses contextuelles aux évènements exceptionnels et possibilité de respecter la notion de contrat logiciel.

4.1 Modèle d'exécution

Nous formulons notre proposition de système de gestion d'exception dans le contexte suivant. Un agent est une entité logicielle autonome qui s'exécute concurremment aux autres agents d'une plate-forme SMA. Au sein de cette plate-forme, les agents communiquent entre eux de manière abstraite et asynchrone. Chaque agent est capable de rendre un certain nombre de services aux autres agents du système. Tout comportement d'un agent est effectué suite à une requête. Toute requête d'un agent à un autre agent (ou à lui-même) résulte en la création d'une tâche par l'agent destinataire, tâche qui traite la requête et s'exécute dans son propre processus. Chaque tâche ainsi créée connaît son appelante. De même, si elle en a, chaque tâche connaît l'ensemble des tâches qu'elle a appelé. Au fil des requêtes et des créations de tâches, les agents du SMA construisent implicitement des arborescences de tâches qui permettent de contrôler le fonctionnement du système tout en assurant l'imbrication des contextes d'exécution.

Pour illustrer notre propos, prenons l'exemple d'une application *Agence de voyage* (cf. Figure 1) grâce à laquelle un client peut organiser un voyage. Pour ce faire, un agent *Client* exécute une tâche *Organisation d'un voyage* qui adresse à un agent *Agence* une requête *Recherche de voyage*. Cette requête crée une tâche de même nom au sein de l'agent *Agence*. Lors de l'exécution de cette tâche, l'agent *Agence* s'adresse une requête de *Recensement des offres* (pour connaître les offres existant sur le marché), une requête de *Sélection du voyage* (pour choisir le prestataire le moins cher) et une requête de *Mise en relation des contractants* (le client et le prestataire choisi). Ces requêtes créent les tâches appelées correspondantes et ainsi de suite comme le montre la figure 1 pour cet exemple.

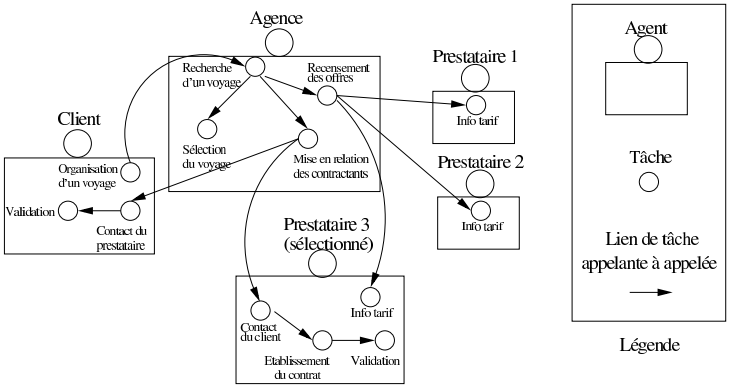

FIG. 1 – Exécution d'une requête à une agence de voyage

4.2 Interface utilisateur du système de gestion des exceptions

Définition de handlers

Avec SaGE, les handlers peuvent être associés au corps des tâches pour traiter un ensemble d'exceptions (leur portée est dynamique). Il est possible d'associer plusieurs handlers pour différents types d'exceptions à la même tâche. Le corps du handler est défini dans le même environnement

lexical que le corps de la tâche. Il est donc possible de traiter contextuellement les exceptions. Le modèle de traitement est le modèle à terminaison. L'étude du modèle avec reprise en présence d'asynchronisme restant à faire. Les handlers peuvent donc uniquement signaler une nouvelle exception, entraîner la terminaison ou la réexécution complète (*retry* à la Eiffel) de la tâche à laquelle ils sont associés.

```
Agence extends Agent {
...
Task t = new Task ("Mise en Relation des contractants", this.getAddress, msg, taskID) {

  public void live () { ...
    addSubTask (this,true,compteur,client) ;
    sendMessage(client, new SageMessage("request_contract", prestataire.toString(), compteur)) ;
  ...}

  public Exception concert(Vector subtasksInfo){ // Corps de la fonction de concertation}

  public void handle (ContratRefuse exc) {
    // On demande la terminaison des sous-tâches en cours
    for (int i :=0 ;i<subtasks.size() ;i++)
      (SubTask)(subtasks.elementat(i)).terminate() ;
   // Puis on réessaie retry() après modification partielle du contexte
  }
}
...
}
```

FIG. 2 – Exemple de handler associé à la tâche Mise en relation des contractants

La figure 2 montre par exemple un handler associé à la tâche *Mise en Relation des Contractants* de l'agent *Agence* pour l'exception *ContratRefuse* signalée par la sous-tâche *Contact du Prestataire*. Ce handler provoque la terminaison d'une part de la sous-tâche interrompue *Contact du Prestataire* et de la sous-tâche encore en exécution *Contact du Client* puis relance l'exécution du corps de la tâche après avoir désigné un nouveau prestataire. Les exemples sont écrits en Java/Madkit, système dans lequel la proposition a été implantée. A la syntaxe près, les structures de contrôles proposées peuvent être mentalement transposées à d'autres systèmes.

Signalement d'une exception

Le signalement d'une exception s'effectue syntaxiquement de façon similaire à ce qui est proposé en Java (cf Figure 3). Par contre l'algorithme de signalement est spécifique (voir plus loin) et prend en compte un modèle de concertation inspiré de [Iss01] et [Lac90].

```
signal(new TaskException ("adresse_client_erronée", getOwnerAddress())) ;
```

FIG. 3 – Signalement d'une exception

Définition d'une fonction de concertation

Une fonction de concertation[4] peut être associée à tout corps de tâche. Si elle existe, elle est prise en compte par l'algorithme de signalement dans sa recherche de handler. Comme dans [Iss01], cette fonction permet de faire la synthèse de tout ce qui s'est passé dans les sous-taches de la tâche courante et de finalement retourner une exception appropriée. Afin de permettre à une tâche

[4]Cette fonction joue le même rôle que la fonction de résolution du modèle coopératif de Valérie Issarny

de poursuivre son exécution en cas de défaillances mineures dans l'exécution des tâches qu'elle a appelées, nous avons intégré à la politique de concertation la notion de criticité, inspirée de [Lac90]. Une tâche est dite *critique* si sa terminaison exceptionnelle interdit l'exécution standard de sa tâche appelante. Une tâche est dite *non-critique* si sa terminaison exceptionnelle n'empêche pas forcément le bon déroulement de sa tâche appelante (cf Figure 4). C'est le demandeur d'un service qui décide si la tâche correspondante sera critique ou non.

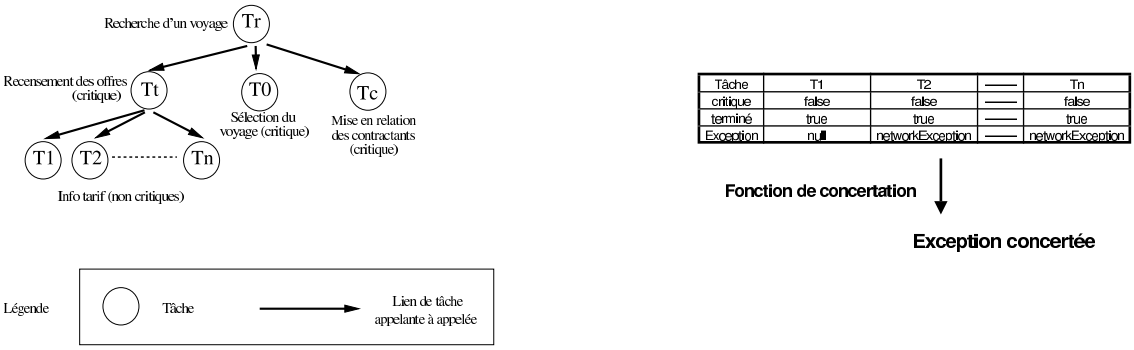

FIG. 4 – *Gestion de la concertation des exceptions*

Au niveau de chaque tâche sont mémorisées dans une collection un certain nombre d'informations sur ses sous-tâches. Pour chaque sous-tâche sont stockés (cf. figure 4) :

- son état (vrai si la tâche est terminée et faux sinon),
- une valeur indiquant si la tâche a signalé une exception et si oui laquelle,
- sa criticité (vrai si la tâche est critique)

Celle collection est passée en argument à la fonction de concertation qui peut l'utiliser à sa guise.

La figure 5 montre un exemple de définition de fonction de concertation associée à la tâche *Recensement des offres* de l'agent *Agence* qui teste si plus de 30% des n demandes de tarifs réalisées par n sous-tâches sont interrompues par le signalement d'une exception. Si oui, on peut suspecter un problème plus général lié au réseau et le code de la fonction de concertation synthétise le comportement global des sous-tâche en signalant l'exception *ExceptionReseau*. La définition d'une fonction de concertation n'a rien d'obligatoire, elle permet de gérer au mieux les cas les plus complexes.

4.3 Principe de signalement d'une exception

Le cœur du système proposé réside dans la façon dont les handlers sont recherchés puis invoqués dans leur environnement de définition. Le signalement d'une exception E par une tâche T interrompt l'exécution de T et provoque la recherche d'un handler associé à T qui soit capable de traiter E. En cas d'échec de cette première recherche, l'exception est propagée à la tâche appelante de T (T') au niveau de laquelle se poursuit le mécanisme de signalement. Effectuer la recherche de handlers en remontant de l'appelé à l'appelant assure le respect d'un modèle contractuel qui permet à un appelant de parfaitement contrôler tout ce qui peut se passer pendant l'exécution des entités qu'il utilise (besoin identifié dans la sous-section 3.3).

Par la suite, au niveau de la tâche appelante T', le mécanisme de concertation entre en jeu. Le tableau d'informations sur les tâches appelées par T' est mis à jour : dans l'entrée correspondant à T sont stockées, l'exception E et la nouvelle valeur de l'état de T (tâche terminée). Si la tâche T était une tâche critique pour T' ou si la tâche T était la dernière tâche appelée par T' à ne pas être terminée, la fonction de concertation d'exceptions associée à T' est évaluée. La fonction de concertation d'exceptions retourne une exception concertée qui sera alors signalée. L'écriture de la fonction de concertation d'exceptions est à la charge du programmeur ; dans l'exemple que nous avons donné, une exception plus globale était substituée à E. Si le programmeur n'a défini aucune fonction de concertation, une fonction par défaut s'y substitue : Si une tâche critique

```
Agence extends SaGEAgent {
...
Task t = new Task ("Recensement des offres", this.getAddress, msg, taskID) {

  public void live () { // Corps de la tâche}

  public Exception concert(Vector subtasksInfo){
    int i :=0 ;
    for (int j :=0 ;j<subtasksInfo.size() ;j++)
      if ((SubTask)(subtasksInfo.elementat(j)).exc !=null) i++ ;
    if (i>(0.3*subtasksInfo..size()))
      return(new ExceptionReseau()) ;
  }

  public void handle (PasDeProvider exc) { // Un handler }
 }
...
}
```

FIG. 5 – Exemple de fonction de concertation associée à la tâche Recensement des offres

avait signalé une exception, cette dernière est retournée ; sinon une exception composite agrégeant toutes les exceptions signalées est retournée. Dans tout les cas, suite à l'exécution de la fonction de concertation de T', la recherche de handler se poursuit par l'examen de ceux définis au niveau de T'.

Lorsqu'une tâche T se termine de façon normale, elle en informe sa tâche appelante (T') qui met à jour l'information sur son état dans le tableau d'informations. Si T était la dernière tâche appelée par T' à ne pas être terminée et qu'au moins une des tâches appelées s'était terminée de manière exceptionnelle, la fonction de concertation d'exceptions associée à T' est évaluée pour permettre éventuellement de signaler une exception concertée qui refléterait la survenue d'un ensemble d'exceptions dans des tâches appelées non-critiques.

4.4 Exemples

Si l'on reprend l'exemple de l'agence de voyage (cf. figures 1 et 4), on peut illustrer les problèmes que nous traitons dans ce modèle et ainsi synthétiser les avancées apportées par SaGE :

- Prise en compte de la criticité : Lorsque l'agent *Agence*, dans le cadre de la tâche *Recensement des offres* effectue n demandes de tarifs auprès des n *Prestataires* qu'elle connaît, aucune des tâches appelées correspondantes n'est critique car on n'a pas forcément besoin de disposer de tout les tarifs, la fonction de concertation d'exceptions associée à *Recensement des offres* ne sera donc pas évaluée avant la terminaison standard ou exceptionnelle de toutes ses tâches appelées. Par contre, *Sélection du voyage* et *Mise en relation des contractants*, sont critiques car *Recherche d'un voyage* ne peut remplir son rôle si ces deux tâches ne s'exécutent pas correctement.
- Concertation des exceptions levées concurremment : Un comportement de la fonction de concertation d'exceptions associée à *Recensement des offres* pourrait être de considérer que l'exécution s'est déroulé normalement si moins de 30% des n demandes ont échoué et, dans le cas contraire, de signaler une exception reflétant un problème de réseau.
- Structuration : On permet la propagation d'une exception signalée dans une tâche d'un agent vers sa tâche appelante même si elle est encapsulée par un autre agent. La tâche C*ontacter le prestataire* de l'agent *Client* peut, par exemple, propager une exception vers sa tâche

appelante qui est encapsulée par l'agent *Agence* pour lui signifier que le prestataire ne lui répond pas. La tâche *Mise en relation des contractants* pourra alors demander la terminaison de son autre tâche appelée *Contact du client* puis demander au client d'entrer en contact avec un autre prestataire qui se verra affecté une demande de contact du client. Une nouvelle tentative est alors initiée par *Mise en relation des contractants.*

– Contextualisation : Contrairement aux modèles de gestion d'exception à superviseur, dans notre proposition, les exceptions sont propagées en remontant l'historique des calculs. Cette approche est préférable car c'est le demandeur d'un service qui est le mieux placé pour évaluer ce qui doit être fait en cas d'erreur lors de l'exécution de ce service. Par exemple, c'est la tâche *Mise en relation des contractants* de *Agence* qui dispose du contexte d'exécution nécessaire pour notifier l'annulation du contrat à un contractant si l'autre a propagé une exception.

5 Conclusion et Perspectives

La problématique de la tolérance aux pannes dans le contexte des systèmes multi-agents a été sous-étudiée. Cet article propose une première évaluation des problèmes que pose la gestion des exceptions dans le contexte des SMA et une proposition de système de gestion des exceptions originale dans la mesure où elle se différencie fondamentalement des quelques propositions de SGE dédiés que nous avons pu recenser. Cette première proposition apporte des solutions à un premier ensemble de problèmes liés à la communication et à la concertation entre processus concurrents, à la définition de réponses aux situations exceptionnelles dépendantes du contexte et à la réalisation concrète de véritables entités logicielles autonomes capables de réaliser des contrats.

D'un point de vue prospectif, les réponses attendues pour un SMA ne se situent pas uniquement dans une gestion des problèmes de communication entre processus. Les systèmes multi-agents apportent deux éléments nouveaux que l'on ne rencontre pas dans les systèmes à base de processus : la notion d'intention d'une part (un agent cherche à accomplir un but, à satisfaire un besoin, etc...) et celle d'organisation d'autre part, notamment avec le modèle AGR/Aalaadin [Fer95] où chaque agent est situé dans un ensemble de groupes dans lesquels il joue des rôles. Ces points constitueront une des suites de notre réflexion.

Par ailleurs, les conséquences des modifications que nous avons été contraints de proposer au modèle d'exécution des agents (introduction nécessaire d'un certain contrôle) n'ont pas été extraites et devront l'être. Bien que des possibilités aient été ajoutées, des contraintes ont du être induites. L'autre pôle de prospection concerne, plus largement, les recherches actuelles sur les problèmes de fiabilité et de tolérance aux pannes que posent les nouvelles architectures logicielles comme par exemple ceux liés à l'utilisation de composants (de type agent ou autre) mobiles.

Références

[Ano79] Anonymous. Preliminary Ada reference manual. *ACM SIGPLAN Notices*, 14(6A) :1–139, June 1979.

[BMP86] J-P Banâtre, M.Banâtre, and F. Ployette. The concept of multi-functions, a general structuring tool for distributed operating system. In *Proceedings of the Sixth Distributed Computing Systems Conference*, 1986.

[BRHL99] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java, 1999.

[Bri00] Frédéric Peschanski Thomas Meurisse Jean-Pierre Briot. Les composants logiciels : évolution technique ou nouveau paradigme ? In *OCM 2000*, 2000.

[CR86] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering (SE)*, SE-12 number 8, August 1986.

[DF94] A. Drogoul and J. Ferber. Multi-agent simulation as a tool for modeling societies : Application to social differentiation in ant colonies. In C. Castelfranchi and E. Werner,

editors, *Artificial Social Systems : Selected Papers from the 4th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'92)*, pages 3–23. Springer, Berlin, Heidelberg, 1994.

[Don90] Christophe Dony. Exception handling and object-oriented programming : towards a synthesis. *ACM SIGPLAN Notices*, 25(10) :322–330, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[Fer95] Jacques Ferber. *Les systemes multi-agents, vers une intelligence artificielle distribuée.* InterEditions, 1995.

[FIP97] FIPA. *FIPA 97 Specification Part 2 : Agent Communication Language*, 1997.

[Gär99] Felix C. Gärtner. Fundamentals of fault tolerant distributed computing in asynchronous environments. *ACMCS*, 31(1) :1–26, March 1999.

[Goo75] John B. Goodenough. Exception handling : Issues and a proposed notation. *Communications of the ACM*, 18(12) :683–696, December 1975.

[Iss01] Valerie Issarny. Concurrent exception handling. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2001.

[IY91] Y. Ichisugi and A. Yonezawa. Exception handling and real-time features in an object oriented concurrent language, 1991.

[KD00] Mark Klein and Chrysanthos Dellarocas. Towards a systematic repository of knowledge about managing multi-agent system exceptions, 2000.

[KD01] Mark Klein and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems : The case of agent death. In *Journal for Autonomous Agents and Multi-Agent Systems*, 2001.

[KrA00] Mark Klein and Juan Antonio rodriguez Aguilar. Using role commitment violation analysis to identify exceptions in open multi-agent systems, 2000.

[KS89] A. R. Koenig and B. Stroustrup. Exception handling for C++. In *Proc "C++ at Work" Conference*, November 1989.

[Lac90] S. Lacourte. Exceptions in Guide, an object-oriented language for distributed applications. Technical Report 5-90, Bull-IMAG, Grenoble (France), December 1990.

[Lis88] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3) :300–312, March 1988.

[MAD] MADKit. *MADKit.* http ://www.madkit.org.

[Mey88] Bertrand Meyer. Disciplined exceptions. Tr-ei-22/ex, Interactive Software Engineering, Goleta, CA, 1988.

[RD01] Pierre-Michel Ricordel and Yves Demazeau. From analysis to deployment : A multi-agent platform survey. *Lecture Notes in Computer Science*, 1972 :93–? ?, 2001.

[Ret99] Reticular Systems, external documentation. *AgentBuilder User's Guide*, 1999. http ://www.agentbuilder.com/.

[RK00] Alexander B. Romanovsky and Jorg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Advances in Exception Handling Techniques*, pages 147–164, 2000.

[The83] Daniel G. Theriault. Issues in the design and implementation of act2. Technical Report TR 728, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1983. pages 213 :adnum AD-A132326 $7.00.

[TM00] Anand Tripathi and Robert Miller. Exception handling in agent oriented systems. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2000.

[Wei83] Daniel L. Weinreb. Signalling and handling conditions. Technical report, Symbolics, Inc., Cambridge, MA, January 1983.