N° d'ordre : 02/2014 - D/INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene (USTHB)
Faculté d'Eléctronique et Informatique

THÈSE

Présentée pour l'obtention du grade de DOCTEUR

En : INFORMATIQUE

Spécialité : Systèmes Informatiques

Par : ZEGHACHE Linda

Sujet :

Tolérance aux pannes dans les systèmes d'agents mobiles transactionnels

Soutenue publiquement, le 27/04/2014, devant le jury composé de :

M AHMED NACER Mohamed	Professeur	à l'USTHB	Président
M BADACHE Nadjib	Professeur	au CERIST	Directeur de thèse
M HURFIN Michel	Directeur de Recherche	à l'INRIA	Examinateur 1
M ALIOUAT Makhlouf	Maître de Conférence/A	à l'Univ. Sétif1	Examinateur 2
M ZAFOUNE Youcef	Maître de Conférence/A	à l'USTHB	Examinateur 3
M TANDJAOUI Djamel	Maître de Recherche/A	au CERIST	Examinateur 4

Remerciements

Louange à Dieu

Je voudrais d'abord exprimer mes vifs remerciements à mon directeur de thèse, le Professeur **BADACHE Nadjib** :
- pour m'avoir gratifié de sa confiance en me proposant le thème de recherche et en acceptant de diriger mes travaux,
- pour m'avoir prodigué les conseils judicieux et m'avoir donné des orientations et des éclairages sur la conduite de mes travaux de thèse.

Je tiens à remercier vivement et nommément :

Monsieur **AHMED NACER Mohammed**, Professeur à l'USTHB, pour l'honneur qu'il me fait en acceptant de présider le jury,

Monsieur **HURFIN Michel**, Directeur de recherche à l'INRIA, pour avoir bien voulu examiner mes travaux de recherche,

Monsieur **ALIOUAT Makhlouf**, Maître de conférence à l'université de Sétif1, pour avoir également accepté de faire partie du jury en qualité d'examinateur,

Monsieur **ZAFOUNE Youcef**, Maître de conférence à l'USTHB pour son expertise et sa qualité d'examinateur,

Monsieur **TANDJAOUI Djamel**, Maître de recherche au CERIST, pour son acceptation de faire partie de ce jury.

Ma reconnaissance va au Centre de Recherche sur l'Information Scientifique et Technique (CERIST) pour avoir financé toutes les étapes du projet, et ma gratitude s'adressera aux différentes structures (La direction, service formation, bibliothèque, SNDL) pour leur disponibilité et aides diverses et utiles.

Il ne doit pas m'échapper de dire mes remerciements et mon affection aux collègues de la division Réseaux dont Mme ElMaouhab ainsi que Faiza, Nassima, Assia, Faiza, Amel, Louiza, Sabera, Amina, Hassina, Ouafa, Houria, Ikram, Hassina, Sahima, Kahina, Mahdia, Ryma, Widad, Karima, Sabrina, Ghania, Chafika, Khira, Rabab, Nadia, Malika, Amine, Ali, Hamid, Abdelouhab, Mourad, Mohammed, Nassim, pour m'avoir accompagné de leur soutien agréable durant la réalisation des mes travaux.

Monsieur **DAHMANE Madjid** ex-Directeur de recherche au CERIST et Monsieur **BELLAL Mohammed** Enseignant à l'ESI méritent d'être amplement remercié pour leur facilitation.

Enfin, je resterai redevable à l'égard du Professeur **HURFIN Michel** et Dr. **MOISE Izabela** de l'IRISA(FRANCE) de n'avoir ménagé aucun effort pour m'apporter aide et soutien et pour lesquels je leur adresse tous mes remerciements.

Également, je remercie le Professeur **GUERAOUI Rachid** de l'EPFL (SUISSE) pour ses conseils éclairés ainsi que son assistante **Mme. Christine**.

Mes remerciements vont également à Mademoiselle **BERAOUI Yasmina**, Responsable des formations et Monsieur **EDDOUD Abdelkader**, Responsable Technique régional du campus numérique francophone d'Alger, pour leur aide hautement appréciable.

Ces remerciements ne peuvent s'achever sans citer ma famille : Mon frère et mes sœurs pour les plaisirs et les rires, pour les fois où nous avons partagé nos rêves et pour les souvenirs à garder au fond du cœur ; Ma mère et mon père pour leur affection et leurs prières, pour leur présence et leurs encouragements ; Vous êtes mes maîtres à l'école de la vie et la raison de ce que je suis et de ce que je fais.

Linda.

Dédicaces

Louange à Dieu

Gloire à l'Algérie ma chère Patrie

... A mes aïeux nés à Tyana - Erraguène -

A vous - maman et papa - dont les mots ne sauraient traduire les bienfaits de votre bénédiction qui m'a toujours accompagnée,

A mon frère Zine El-Abidine,

A mes sœurs Safia, Schahrazed, Karima, Houria, Khalissa, Ahlem et Esmahane,

A mes nièces et mes neveux,

A mes oncles et mes tantes et leurs familles,

A mon grand père Hadj-Larbi,

A la mémoire de mes grand-mères Oumessaad, Rebeha et mamma Zineb,

A la mémoire de mon grand père maternel Messaoud,

A mes amies Mokhtaria, Nabila, Amel, Nassima, Faiza, Dalila et Houda,

Et à tous ceux qui me sont chers,

... Auxquels je dédie ce modeste travail et auxquels j'exprime toute mon affection !

Linda.

Table des matières

Table des figures

Liste des tableaux

Résumé

Le concept d'agents mobile transactionnel a été introduit comme un amalgame entre la technologie d'agents mobile et le modèle transactionnel. durant son parcours, l'agent visite n places pour satisfaire les n requêtes de la transaction. Les nœuds visités gardent assez d'information afin de pouvoir valider ou annuler la transaction globale.
Les travaux de recherches sur les agents mobiles transactionnels ont identifiés deux problèmes que l'agent seul ne pas résoudre. Le premier est lié à la fiabilité des agents. Des agents et des nœuds peuvent tomber en pannes et donc, des mécanismes de détection de pannes et de recouvrement doivent être définis. La solution classique consiste à utiliser la dernière place visitée pour contrôler l'agent quand il migre vers une nouvelle place. Dans ce cas, le dernier nœud visité qui détient le mécanisme de détection risque de tomber en panne et doit être contrôlé à son tour. En conséquence, toutes les places visitées (y compris le client ayant initié la transaction)doivent former une chaîne où chaque nœud contrôle son successeur. Lorsque l'agent est en panne, un nouvel agent est créé par la place qui a suspecté la panne.
Il à noter que dans un système réparti asynchrone, où les détecteurs de pannes ne sont pas fiables, le lancement d'un nouvel agent ne garantit pas son unicité. De plus, le client, étant le premier nœud de la chaine de contrôle, il doit rester connecté durant l'exécution de toute la transaction.
Le second problème pour lequel l'agent doit être assisté est lié la validation atomique de la transaction. Une fois l'agent aura identifié et visité les n nœuds capables de satisfaire les n requêtes de la transaction, il doit déléguer la supervision du processus de validation à une entité externe et surtout fiable. Pour plusieurs raisons, cette tâche ne devrait pas être faite uniquement par l'agent. Tout d'abord, par définition, un agent mobile est une entité qui exécute un code local sur le nœud où il se trouve. Cependant, un protocole de validation est un service distribué qui exige une communication entre l'initiateur et les n participants. D'autre part, la puissance du calcul d'un agent dépend de son nœud hôte qui peut avoir des capacité de calcul, de stockage ou des ressources limités. Enfin, le principe d'un protocole de validation atomique repose sur le fait qu'un coordinateur existe, ou généralement plusieurs, et que chaque coordinateur agit de d'une manière conforme à ce qui a été fait par les coordinateurs précédents. Pour répondre à toutes les difficultés avancées ci-dessus, une entité centralisée et surtout fiable peut être définie pour assister les agents mobiles dans leurs tâches de contrôle, de validation et de journalisation.
En effet, nous proposons une approche uniforme basée sur le concept d'accord pour fournir deux services, la disponibilité de la source (AS) et la validation atomique (AC), qui offrent un support pour l'exécution de l'agent d'une manière efficace, fiable et homogène.

Abstract

The concept of transactional mobile agent has been introduced as a mix between the agent technology and the transactional model. During its move, the agent discovers and visits n places that satisfy the n requests. During the agent's visit, a node records enough information so that it can subsequently either commit or abort the transaction.
Different works on transactional mobile agents have identified two problems that can not be solved by the agent alone. The first one is related to the reliability of the agent. Agents and nodes may fail. Failure detection and recovery procedures have to be defined. A classic solution consists in using the last visited node to monitor the state of an agent once it moves to another place. In that case, the last visited node which is hosting a failure detection mechanism becomes also a critical point of failure and thus, it has also to be monitored. One thing leading to another, all the nodes already visited by the agent (and the client who initiates the transaction) have to form a chain where each node observes its successor. When the agent is suspected to be crashed, a new agent is created by the node that suspects the occurrence of a failure.
Note that in an asynchronous distributed system where failure detectors are sometimes unreliable, the generation of a new agent in case of problems no more guarantees its uniqueness. Since the client is the first node in the monitoring chain, it has to remain connected during the whole execution of the transaction. The second problem for which the agent needs assistance is related to the atomic validation of the transaction. Once an agent has identified and visited n nodes able to satisfy the n requests of a transaction, it has to delegate the supervision of the validation process to an external reliable entity. For several reasons, this final task should not be made solely by the agent. First, by definition, an agent is a moving entity that executes a local code on the node where it is located. But a validation protocol is typically a distributed service that requires communications between its initiator and the n involved nodes. Second, the computing power of an agent depends on its hosting node which may have limited processing, storage and power resources. Third, a validation protocol relies on the fact that a single coordinator exists or, more generally, on the fact that any coordinator acts in a manner consistent with what has been done by previous coordinators. To address all the above remarks, a centralized and reliable entity can be defined to assist all the agents in their monitoring, validation and logging tasks.
Indeed, we propose a uniform approach based on the concept of agreement to provide two services, Availability of the Source (AS) and Atomic Commit (AC), that provide support for the agent execution in an efficient, reliable and homogeneous way.

Introduction générale

Contexte

L'approche agent mobile fut introduite pour la première fois en 1994 comme un paradigme de programmation adéquat et efficace pour les applications distribuées notamment pour des ordinateurs partiellement connectés.
L'agent mobile est une entité autonome qui agit par délégation pour le compte d'un client et se déplace d'une machine à une autre sur un réseau hétérogène, sans perdre son code ni son état.
Avec les agents mobiles, il est possible d'apporter le code près des ressources, ce qui n'est pas prévu par le paradigme traditionnel client/serveur.
Avec l'utilisation de plus en plus courante des technologies sans fil, les réseaux évoluent de plus en plus vers de nouvelles architectures dynamiques à large échelle. Les sites deviennent mobiles, apparaissent et disparaissent fréquemment, les connexions évoluent de façon continue. Ces nouvelles architectures ont amené et amènent aujourd'hui encore à reconsidérer les systèmes et applications réparties qui étaient conçus selon les anciennes structures fixes. La conception des applications et des systèmes répartis repose sur un ensemble de concepts et d'abstractions tels que le client-serveur, le pair à pair ou le schéma de publication/abonnement. La notion d'agent mobile constitue une abstraction déjà ancienne mais présentée parfois comme une approche pouvant apporter des performances meilleures par rapport au schéma de base client/serveur. Cette faculté est essentiellement due à leur aptitude au déplacement en fonction de leurs besoins propres pour accomplir au mieux leurs tâches.
La technologie d'agents mobiles a donc été considérée pour diverses applications telles que la gestion des systèmes et des réseaux, les workflows et le e-commerce. Dans ces domaines d'applications les opérations locales exécutées par l'agent mobile sont généralement liées et peuvent nécessiter une exécution atomique (c-à-d si l'une d'entre elles n'est pas accomplie elles doivent toutes être abandonnées). La technologie des agents mobiles doit supporter la notion de transaction pour être utile dans ce type d'applications.
Considérons un agent mobile qui réserve un billet d'avion et loue une voiture à la destination du vol. Louer une voiture à la destination n'est d'aucune utilité s'il n'y a pas de vol disponible (si aucune place n'est disponible pour toutes les compagnies aériennes). Dans cet exemple, il existe une forte dépendance entre les actions de l'agent mobile, dans la mesure où elles ont toutes besoin de réussir ensemble (sinon aucune). Cette propriété, appelée exécution atomique, est cruciale et doit être assurée pour tout agent mobile exécutant des

opérations dépendantes. Il est appelé dans ce cas un agent mobile transactionnel.
L'exécution atomique exige la fiabilité de l'agent mobile. Elle introduit également un autre niveau de difficulté quand à l'exécution de l'agent mobile. D'habitude, l'agent valide son exécution sur chaque nœud avant de migrer vers le nœud suivant, cependant, si la transaction n'est pas validée à la fin, des opérations de compensation doivent être exécutées pour chaque nœud c'est à dire chaque opération de l'agent doit avoir sa procédure de compensation pour annuler ses effets en cas d'abandon de la transaction. Ceci n'est pas toujours évident.
L'agent mobile peut choisir de ne pas valider les tâches jusqu'à son arrivée à la destination. Néanmoins, les ressources allouées par l'agent sont verrouillées et ne sont libérées que lorsque la décision de transaction est connue (validation ou abandon). Dans ce cas d'autres transactions, demandant une même ressource, sont pénalisées jusqu'à la terminaison de l'exécution de l'agent sachant que ce dernier peut abandonner la transaction à la fin.

La grande flexibilité du modèle agent mobile s'accompagne de coûts supplémentaires. Ces coûts comprennent, entre autres, la complexité de développement, la fiabilité et la sécurité.

Problématique

Les agents mobiles comme dans tout environnement distribué, se heurtent à de grandes difficultés de fiabilité. Des problèmes de défaillances peuvent survenir et avoir un impact sur l'exécution des agents mobiles. Une défaillance (par exemple, celle d'un agent ou d'une machine) peut empêcher l'agent de continuer son exécution ; plus grave encore, l'état actuel de l'agent, et même son code, peuvent être perdus.
Pour contourner ce problème le propriétaire de l'agent peut essayer de détecter la défaillance de l'agent, dans le but d'envoyer un nouvel agent de remplacement. Par contre, ceci nécessite un mécanisme de détection de défaillance très fiable, capable de faire la distinction entre un agent défaillant et un agent qui aurait été retardé par des processeurs ou des liens de communication lents. Malheureusement, ceci n'est pas possible dans un environnement ouvert tel que Internet. Cette incertitude peut mener à deux situations :

- Le propriétaire de l'agent considère par erreur que l'agent est défaillant. Lancer un autre agent va générer une double exécution.

- Le propriétaire de l'agent attend le retour d'un agent défaillant. Cela conduit à une situation de blocage.

Dans un tel contexte, et afin d'aider l'agent mobile dans sa tâche, il convient de trouver la meilleure solution possible (approches, méthodes et techniques) pour lui fournir un environnement d'exécution fiable et approprié à ses caractéristiques d'exécution transactionnelles.
Bien que plusieurs solutions soient proposées dans la littérature pour traiter les défaillances matérielles ou logicielles, l'exploitation efficace et fiable des systèmes d'agents mobiles reste limitée.

Objectifs

L'objectif du présent travail est le développement d'un mécanisme de tolérance aux pannes pour les systèmes d'agents mobiles transactionnels exécutés sur un ensemble de sites de façon atomique. Ce mécanisme doit permettre à un agent mobile de s'exécuter correctement, avec des garanties d'efficacité, sur un réseau ouvert où les liens de communications sont versatiles.
L'objectif de notre travail consiste donc à proposer un protocole de tolérance aux pannes pour les agents mobiles transactionnels permettant :

- La détection et recouvrement de la panne de l'agent mobile,
- Le non blocage en cas de pannes,
- Une seule bonne exécution de l'agent mobile,
- Une exécution atomique de l'agent mobile,
- Une validation atomique non bloquante de la transaction exécutée par l'agent mobile.

Organisation de la thèse

Les lectures synthétisées et les réflexions développées dans le cadre de ce travail de thèse ont fait l'objet de cinq chapitres organisés comme suit :

Chapitre 1 Dans ce chapitre nous nous intéressons à la tolérance aux pannes des agents mobiles. Dans une première partie nous présentons le concept d'agents mobiles, les avantages, les domaines d'applications et les inconvénients. Ensuite, nous abordons la sûreté de fonctionnement dans les systèmes répartis ainsi que les techniques utilisées pour assurer la tolérance aux pannes. Enfin, nous présentons les approches de tolérance aux pannes des agents mobiles proposées dans la littérature.

Chapitre2 Ce chapitre est consacré aux agents mobiles transactionnels. Nous commençons par présenter les modèles transactionnels existants, puis nous décrivons les agents mobiles transactionnels et leurs spécificités. Une synthèse des travaux proposés pour les agents mobiles transactionnels est présentée.

Chapitre3 Il aborde la validation atomique non bloquante. Nous étudions la validation atomique dans les systèmes synchrones puis dans les systèmes asynchrones en citant les principaux protocoles existants. Nous introduisons les détecteurs de défaillances et les problèmes d'accord qui n'interviennent pas directement dans la validation atomique mais qui peuvent être utilisés en tant que brique de base pour assurer la tolérance aux pannes de l'entité qui exécute la validation atomique.

Chapitre4 Il est dédié à la solution proposée. Nous présentons le modèle du système d'agents et le modèle transactionnel que nous adoptons dans notre solution. Le protocole est basé sur la chaîne de contrôle et le recouvrement en avant pour assurer la tolérance aux pannes. Deux services principaux sont identifiés pour assurer la disponibilité de l'agent et

sa terminaison puis la validation de la transaction. Ces deux services sont basés sur un protocole d'accord appelé MIC-Paxos utilisé comme brique de base. Les pseudo codes du protocole sont également présentés.

Chapitre5 Ce chapitre a comme intérêt de valider la solution proposée. Nous présentons les résultats de la mise en œuvre en fournissant des scénarios de pannes réelles et en analysant les performances du protocole.

Enfin, ce rapport sera clôturé par une conclusion générale résumant le travail de recherche mené, la contribution apportée pour la tolérance aux pannes des agents mobiles transactionnels, suivie des perspectives de recherche consécutives à ce travail.

Chapitre 1

Agents mobiles et tolérance aux pannes

L'origine du terme agent vient du mot Grec *agein*, pour signifier conduire ou mener. Aujourd'hui, ce terme désigne une entité qui produit ou peut produire un effet. Cela pourrait être une substance chimique, physique ou biologique. Avec la connotation de mener, ce terme est très adapté pour décrire les nouvelles tendances informatiques où l'ordinateur est considéré comme un instrument actif à qui des tâches peuvent être déléguées. Afin de donner plus d'autonomie à l'ordinateur dans l'exécution des tâches, beaucoup de logiciels et de techniques de programmation ont été proposés. Ces logiciels sont les agents intelligents.
Selon [1], les agents logiciels peuvent être classés en termes d'espace défini par trois dimensions : l'intelligence, l'autorité (agency) et la mobilité (figure 1.1).
La première dimension, l'intelligence, remonte aux années cinquante, où le terme agent a été utilisé pour désigner un 'robot logiciel'. Les agents intelligents diffèrent selon leur capacité d'exprimer les préférences et d'exécuter les tâches suivant un raisonnement, un planning et des techniques d'apprentissage.
La seconde dimension, l'autorité, représente le degré d'autonomie accordé à l'agent. Il est mesuré par la nature des interactions entre l'agent et les autres entités du système. Selon leur capacité d'interaction, les agents sont dits autonomes, collaboratifs, coopératifs ou négociateurs[2].

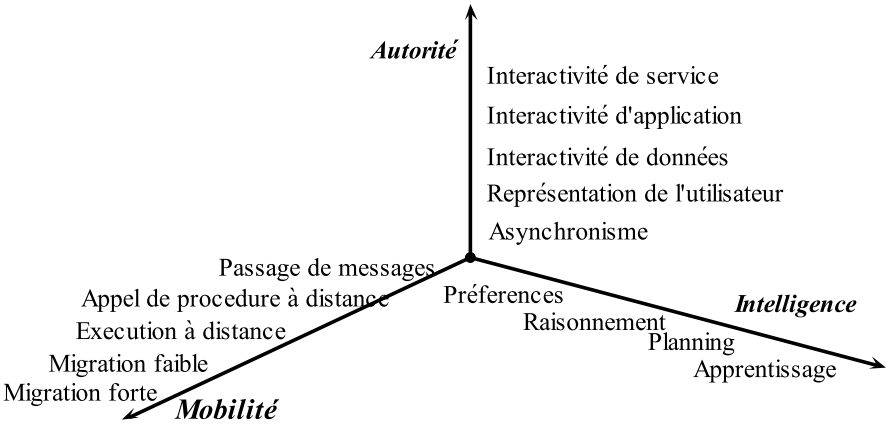

FIGURE 1.1 – Espace des agents logiciels

La troisième dimension, à savoir la mobilité, a émergé dans les années quatre vingt dix et a été motivée par l'essor et le développement rapide des quantités d'information manipulées par les différentes applications dépassant largement le rythme des améliorations apportées aux réseaux.
L'idée qu'un assistant digital parcourant le réseau afin d'exécuter une tâche pour le compte de l'utilisateur en exploitant intelligence, autorité et mobilité, a alimenté plusieurs recherches dans ce domaine.
Néanmoins, l'utilisateur, peut être amené à exécuter des tâches de façon imprévisible, basées sur ses propres croyances et émotions. Dans ce cas, il risque d'être inconfortable en présence d'un compagnon intelligent. En fait, l'utilisateur a besoin d'une entité active qui exécute sa demande en essayant d'étendre ses capacités de calcul et ce, en s'exécutant sur des machines distantes.
Les agents mobiles agissent dans ce sens ; ils permettent d'aider l'utilisateur à exploiter au maximum un système distribué tel que l'Internet.
Dans ce chapitre, nous allons décrire plus en détail le concept d'agent mobile. Nous présentons les domaines d'application ainsi que les avantages et les inconvénients qui en découlent. Ensuite, nous mettons l'accent sur la tolérance aux pannes en tant que propriété qui s'impose pour pouvoir utiliser les agents mobiles.
Nous commençons par décrire les techniques de tolérance aux pannes dans les systèmes répartis puis nous passons en revue les principaux travaux autour de la tolérance aux pannes dans les systèmes d'agents mobiles.

1.1 Les agents mobiles : Concepts de base

Les agents mobiles ont émergé il y a plus de deux décennies comme un nouveau paradigme pour concevoir les systèmes répartis. Un des problèmes clés qui a motivé l'introduction des agents mobiles était la non performance des modèles d'interaction à travers des liens de communication faibles, notamment avec l'introduction des réseaux ad-hoc où les infrastructures de communication sont limitées.
Avec l'arrivée des technologies sans fil, d'autres types d'architectures physiques sont appa-

rues dont principalement les modèles à station de base et ad-hoc. Le mode ad-hoc permet de former dynamiquement des réseaux d'ordinateurs sans aucune architecture fixe (l'architecture est complètement décentralisée).
Les différentes méthodes d'interaction ont montré leur efficacité dans les environnements stables à grande échelle, type Internet. Cependant, l'introduction de la mobilité physique, due aux technologies sans fil, apporte un degré de dynamisme qui ne semble pas permettre à ces méthodes d'atteindre le même niveau d'efficacité mais elle peuvent être améliorées en introduisant la mobilité du code.

1.1.1 Communication par messages vs. mobilité

La communication entre les entités d'un système distribué peut être prise en charge par différents mécanismes. L'échange de messages fut le premier modèle proposé, qui permet aux processus de communiquer par l'envoi et la réception de messages. Ce paradigme est réalisable à travers des bibliothèques de bas niveau généralement difficiles à mettre en œuvre. De plus, cette méthode n'est pas directement intégrée aux langages de programmation et nécessite de définir un protocole de communication stricte. Ces deux contraintes ne permettent pas une programmation simplifiée aux développeurs.
Une abstraction de communication de haut niveau est fournie par le concept d'**appel de procédure à distance** ou RPC (Remote Procedure Call). Ce mécanisme est une abstraction permettant aux développeurs de ne pas manipuler directement le service de communication réseau mais d'appeler une procédure distante comme si elle était locale. Ainsi, un processus utilisant le RPC ne fait aucune différence entre appel local et distant. Notons qu'il faut connaître à l'avance la procédure que l'on souhaite utiliser et que cette procédure doit être disponible sur le site distant. Cette contrainte limite l'utilisation du RPC dans un système réparti ouvert. Il est parfois souhaitable d'envoyer la procédure vers un nœud distant (où se trouve les données) et l'exécuter sur ce nœud. Ce niveau de flexibilité est assuré par un concept appelé évaluation à distance, qui est une extension naturelle du RPC[2].
Avec **l'évaluation à distance**, un processus non seulement transmet le paramètre de la dite procédure mais aussi le code du procédé lui-même. Dans cette première utilisation de la mobilité, le client peut envoyer directement le code au serveur, comme dans le cas du *rsh* (Remote Shell) qui permet d'exécuter un script sur une machine distante (voir figure1.2).

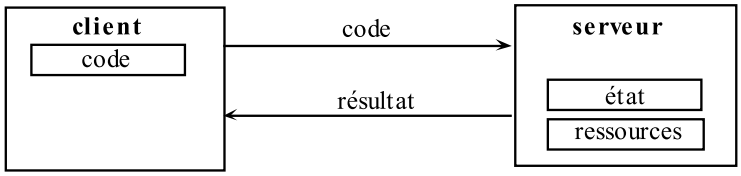

FIGURE 1.2 – Évaluation à distance

Un autre schéma de mobilité appelé **code à la demande** est introduit. Le client dispose de l'unité d'exécution et des ressources mais va récupérer le code auprès du serveur. Il s'agit donc de l'inverse du cas précédent. Ainsi, un client adresse une requête uniquement pour récupérer un code précis afin de l'exécuter localement avec les ressources présentes. Cette

méthode permet d'étendre les fonctionnalités d'une application directement chez le client (voir figure1.3). Les Applets Java reposent sur cette technologie de code mobile, il s'agit d'un programme chargé à partir d'une page Web pour être exécuté sur la machine du client.

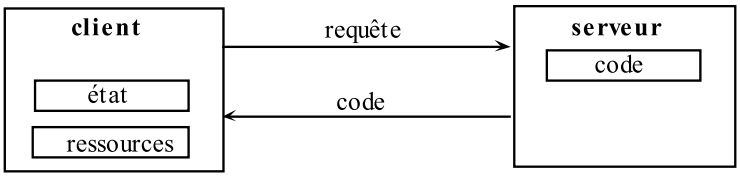

FIGURE 1.3 – Code à la demande

Par comparaison avec les deux schémas précédents qui introduisent la mobilité du code, l'**agent mobile** permet la mobilité du processus. Dans la mesure où le client a besoin d'interagir avec le serveur, ce même processus (code, état d'exécution et données) se déplace à travers le réseau pour continuer son exécution et pour interagir localement avec les ressources du serveur (voir figure 1.4). A la fin de l'exécution, l'agent mobile retourne éventuellement vers son client afin de lui fournir les résultats. Dans ce schéma, le savoir-faire appartient au client, l'exécution du code est initiée côté client et continuée sur les différentes machines visitées[3].

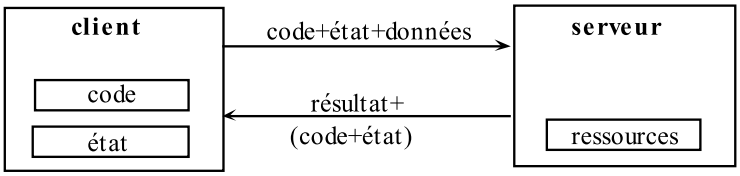

FIGURE 1.4 – Agent mobile

1.1.2 Définition d'un agent mobile

Les agents mobiles ont été introduits initialement en 1994 avec l'environnement Telescript qui permettait à des processus de choisir eux mêmes de se déplacer sur les sites d'un réseau afin de travailler localement sur les ressources[4]. De cette définition découlent les propriétés suivantes :

- Comportement : Un agent est pourvu d'un programme qui lui dicte la tâche à accomplir. On dit qu'il agit par délégation pour le client. Un agent peut ainsi venir avec une compétence particulière sur une machine hôte.
- Autonomie : Un agent agit indépendamment du client. Il décide lui-même là où il va se déplacer et ce qu'il doit y faire, en fonction du comportement qui lui a été donné. Il n'est donc pas toujours possible de prévoir que l'on ne peut pas toujours prévoir l'itinéraire des agents.
- Mémoire : Un agent mobile dispose d'une capacité de mémorisation lui permettant

de récolter des informations sur les sites visités. Les informations mémorisées seront livrées par l'agent après son retour, au client.

- Environnement : L'environnement dans lequel l'agent évolue est constitué par :
  - Les machines qui vont accueillir l'agent lors de son exécution ; ainsi l'agent utilise les ressources (unité de calcul, mémoire, etc...) disponibles sur chacune des machines afin d'accomplir la tâche demandée par le client,
  - Les liens réseau que l'agent utilise pour se déplacer entre les différents sites,
  - Les autres agents fixes et mobiles avec qui l'agent interagit afin de réaliser son but[3].

1.1.3 Structure d'un agent mobile

Un agent mobile est un programme pouvant se comporter d'une façon autonome au profit de son client. Chaque agent a son flot d'exécution pour pouvoir prendre l'initiative d'exécuter des tâches. Un agent doit avoir la capacité de migrer d'une machine à une autre sur le réseau. Pour parler de la migration d'un agent, il est important de connaître les éléments à transférer avec l'agent. Une instance d'agent mobile comporte trois parties :

- Un code exécutable qui représente la suite d'instructions définissant le comportement statique de l'agent mobile,
- Un contexte d'exécution qui reflète l'état d'exécution courant de l'agent mobile (valeurs des registres, pile d'exécution),
- Les données ou les ressources utilisées par l'agent mobile ; ces ressources sont divisées en deux parties :
  1. Les ressources transférables qui sont les valeurs des attributs de l'agent et qui constituent son état global,
  2. Les ressources non transférables qui constituent l'environnement d'exécution fourni par le système (par exemple les fichiers ouverts, les connexions sockets, les registres, etc.) et les matériels physiques utilisés par l'agent (imprimante, écran, etc.).

1.1.4 Migration d'un agent mobile

La migration permet le transfert d'un agent en cours d'exécution d'un site à un autre à travers le réseau. Nous allons prendre le cas d'un agent qui désire migrer entre deux machines avec la possibilité de recevoir des messages pendant son déplacement. Cette migration comporte les étapes suivantes[3] :

1. La sérialisation du contexte de l'agent et de son code, le résultat est inclus dans un message ;
2. Le processus de l'agent étant suspendu sur sa machine d'origine, toutes les communications avec d'autres agents sont donc suspendues et l'agent est détruit ;
3. L'envoi du contexte et du code de l'agent vers la machine de destination ;
4. L'état de l'agent migrant est restauré sur la machine de destination, un nouveau processus léger est créé pour la poursuite de son exécution ;

5. Les communications en cours sont redirigées vers la machine de destination, l'agent n'étant pas encore activé, ces messages seront traités après sa réactivation ;
6. La réactivation de l'agent sur la machine de destination, dès que la partie de son contexte nécessaire à son exécution est reçue ; la machine de destination s'occupe des liens dynamiques entre l'agent et son code. À partir de ce moment, la migration prend fin. Le processus sur la machine d'origine peut être définitivement effacé.

Les différentes stratégies de migration d'agents dépendent de la manière dont sont traitées ces diverses étapes et les données transférées avec l'agent lors de sa migration. Deux types de migrations ont été proposées dans les plateformes existantes :

1. La migration forte permet à un agent de se déplacer quelque soit l'état d'exécution dans lequel il se trouve. Dans ce type de migration l'agent se déplace avec son code, son contexte d'exécution et ses données. Dans ce cas, l'agent reprend son exécution après la migration exactement là où elle était avant son déplacement. La migration forte nécessite un mécanisme de capture instantanée de l'état d'exécution de l'agent. Elle peut être proactive ou réactive. Dans la migration proactive, la destination est déterminée par l'agent. Dans la migration réactive, la migration de l'agent est dictée par une partie ayant une relation avec l'agent mobile.
2. La migration faible ne fait transférer avec l'agent que son code et ses données. Sur le site de destination, l'agent redémarre son exécution depuis le début en appelant la méthode qui représente le point d'entrée de l'exécution de l'agent, et le contexte d'exécution de l'agent est réinitialisé. Pour que l'agent se branche sur une instruction particulière de son code après sa migration, le programmeur doit inclure dans l'état de l'agent des moments privilégiés (explicites) dans le code de l'agent (point d'arrêt) pour pouvoir le relancer.

La migration forte, bien que beaucoup plus exigeante à implanter que la migration faible, n'en est pas moins indispensable pour toutes les applications pour lesquelles les notions de fiabilité et de tolérance aux pannes sont primordiales.

1.1.5 Modèle d'exécution d'agents mobiles

Même s'il existe de nombreuses approches pour la mise en œuvre de la migration dans les systèmes d'agents mobiles, la migration elle-même n'est qu'une partie du modèle d'exécution.
L'exécution de l'agent mobile est constituée d'une séquence d'étapes appelés 'stades d'exécution'. Chaque étape est exécutée par l'agent mobile sur une place p_i, où une place p_i $(0 \leq i \leq n)$ offre l'environnement d'exécution logique pour l'agent. La première et dernière place où l'agent mobile doit s'exécuter, sont appelées respectivement source et destination. La séquence de places $p_0, p_1, ..., p_n$ est appelée l'itinéraire de l'agent mobile. Un itinéraire statique est entièrement défini à la source et ne change pas au cours de l'exécution de l'agent, alors qu'un itinéraire dynamique subit des modifications par l'agent.
Logiquement, un agent mobile s'exécute en une séquence de stades d'actions (voir figure 1.5). Chaque stade S_i consiste en de multiples opérations $op_0, op_1, ...$ L'agent a_i au stade S_i représente l'agent a qui a exécuté les actions sur les places $p_0, ..., p_{i-1}$ et s'apprête à s'exécuter sur la place p_i[5].

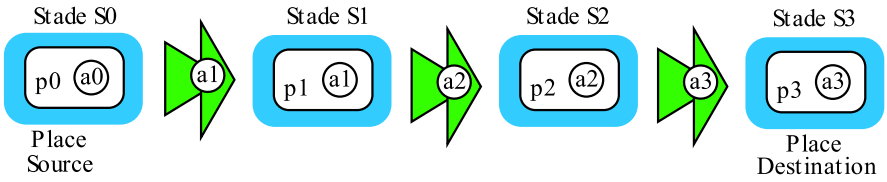

FIGURE 1.5 – Modèle d'exécution d'agents mobiles

1.1.6 Les plateformes d'agents mobiles

Lorsque l'on parle de plateformes, nous regroupons deux niveaux élémentaires. Le premier est le langage de programmation qui offre les primitives nécessaires à l'exécution, à la communication et à la mobilité ; il constitue le niveau de base. Le deuxième est l'intergiciel proprement dit qui offre des services de plus haut niveau, comme les annuaires, permettant aux agents la réalisation de leurs tâches.

1.1.6.1 Les langages sous-jacents

Le premier langage à proposer ces propriétés fut Telescript de General Magic qui fut suivi de nombreux autres tels Obliq[6], Safe-Tcl[7], etc. Ils possèdent tous des qualités propres, comme la prise en compte de la mobilité forte par exemple, mais se voient supplantés par la popularité grandissante d'un langage à objets, à savoir Java. Une des premières plateformes à utiliser cette technologie fut Mole[8].
En effet, grâce à sa machine virtuelle (JVM) largement répandue, Java a su s'imposer pour la programmation d'applications réparties dans des environnements hétérogènes en permettant une indépendance envers les réseaux et les systèmes d'exploitation. La migration est supportée grâce au mécanisme de sérialisation pour le transport d'objets de sites en sites, la sérialisation étant la transcription de l'état et de la structure complexe d'un objet dans un format particulier. Pourtant Java présente un inconvénient : il ne gère pas la mobilité forte. Malgré des travaux sur l'extension de la machine virtuelle, c'est toujours au développeur de gérer les différents états rencontrés après la migration de ses agents[4].

1.1.6.2 Les standards

L'émergence de nombreuses plateformes expérimentales a rendu nécessaire la proposition d'une harmonisation grâce à la standardisation des différents concepts communs pouvant être identifiés. On peut trouver deux normes principales : il s'agit de la norme FIPA (Foundation for Intelligent Physical Agents) [9] et de la norme MASIF (Mobile Agent System Interoperability Facility) [10].
Lorsque deux normes existent au sein d'une même technologie, elles ont une tendance naturelle à s'opposer alors que, dans notre cas, elles penchent plus vers la complémentarité due à la différence de leurs domaines d'origine.

MASIF La norme MASIF a été spécifiée par l'Object Management Group (OMG) qui se préoccupe généralement de l'hétérogénéité entre les systèmes, comme dans le cas de CORBA. Dans cette optique, le but, dans la norme MASIF, est de décrire les notions

élémentaires permettant l'échange des agents entre différentes plateformes. Pour ce faire, elle standardise la manière de gérer le code des agents, leur identification, la migration et l'adressage local.

FIPA En revanche, la communauté d'origine de FIPA [1] étant celle des systèmes multi-agents, plus proche de l'intelligence artificielle, elle va se situer à un niveau plus élevé i.e le niveau applicatif en décrivant les éléments nécessaires à la réalisation d'une application et principalement en détaillant la communication entre les agents. Le but est de décrire un ACL (Agents Communication Language), les ontologies et des protocoles de négociation permettant ainsi de définir parfaitement les interactions entre les agents.

1.1.6.3 Les plateformes existantes

Un agent mobile s'exécute sur une plateforme qui lui offre des services pour son déplacement, sa localisation et la communication avec les autres agents. Ainsi, la performance et la fiabilité de l'environnement d'exécution vont être déterminants pour la qualité d'une application basée sur la technologie d'agents mobiles. Une centaine de plateformes existent à nos jours ; elles se différencient par le type de migration de l'agent (faible ou forte), par la sécurité et la protection de l'agent et du site d'accueil, par la fiabilité offerte lors de l'exécution (tolérance aux pannes), par la facilité de retrouver un agent mobile, par les facilités d'implémentations offertes aux programmeurs et par les services offerts pour la communication entre les agents[3].
Dans le tableau 1.1, une liste des plateformes d'agents mobiles les plus connues, issues des milieux industriels et commerciaux, [11, 12, 13, 14, 15, 16, 17], et des travaux de recherche[18, 5, 19, 20], est présentée.

1. FIPA, l'organisme de normalisation pour les agents et systèmes multi-agents a été officiellement accepté par l'IEEE en tant que onzième comité de normalisation le 08 Juin 2005

Les Plateformes		Institution	Standard	Langage	Mobilité	Commu-. nication
issues des milieux indus. et commerc.	Telescript	General Magic	-	Python	forte	évènement
	MAF	Baan Technology	-	java	faible	-
	Voyager	ObjectSpace	-	java	faible	via proxy
	Aglets	IBM	-	java	faible	message
	Concordia	Mitsubishi ElectricITA	-	java	faible	évènement
	Jumping Beans	AdAstra	-	java	faible	message
	Grasshopper	GMD FOKUS and IKV++	MASIF FIPA	java	faible	évènement
	Jade	Telecom Italia	FIPA	java	faible	ACL
	JATLite	Stanford Center for Design	-	java	faible	KQLM
issues des milieux de recherche	Agent Tcl/ D'agent	Dartmouth College	-	Tcl	forte	message
	James	Univ. Combra, Siemens	-	java	faible	message
	Mole	Univ. of Stuttgart	-	java	faible	RMI, message
	Tacoma	Univ.of Tromsø, Cornell univ.	-	Tcl,Perl, C,Scheme, Python	forte	fichier
	Ara	Univ. of Kaiserslautern	-	Tcl, C, java	forte	messages C/S local
	Nomads	DARPA, NASA, Univ. of Florida	-	java aroma vm	forte	RMI
	MoPros	IBM Zurich Research Lab	-	java	faible	RMI
	Gypsy	Tech. Univ. of Vienna	-	java	faible	RMI, Email
	Javact	IRIT Lab	-	java	faible	message
	Zeus	BT Labs	FIPA	java	faible	ACL
	Anchor	Berkeley Lab.	-	java	faible	SSL
	Springs	Univ. of Zaragoza	-	java	faible	via proxy
	Lime	Univ.of Rocheste, Politecnico di Milano	-	java	faible	mémoire partagée

TABLE 1.1 – Comparaison entre les plateformes d'agents mobiles issues des milieux industriels et commerciaux et des travaux de recherche.

1.1.7 Avantages des agents mobiles

Cette section, portera sur l'exposé des différents apports des agents mobiles. Les premières améliorations apportées par les agents mobiles concernent le gain de performance dû à une meilleure utilisation des ressources physiques mises à disposition. L'amélioration interviendra à différents niveaux et permettra d'optimiser la tâche globale des agents.

1.1.7.1 Diminution de l'utilisation du réseaux

Le déplacement des agents mobiles permet de réduire significativement, voir supprimer, les communications distantes entre les clients et les serveurs. En privilégiant les interactions locales, l'utilisation du réseau va se limiter principalement au transfert des agents. Cette situation présente trois principaux avantages[4].
Le premier avantage est la diminution de la consommation de bande passante. En effet, plusieurs études montrent qu'en comparaison de l'envoi à distance de requête (procédures et méthodes), la mise en place des agents mobiles permet d'obtenir une réduction significative de la charge réseau en terme du nombre total de données transférées. Cette diminution est constatée dans différents types d'applications nécessitant d'intenses échanges d'informations entre le client et le serveur.
Le second avantage notable est la diminution des temps de latence. Dans le contexte des réseaux à large échelle, la mise en place d'applications reparties, nécessitant de fréquentes interactions entre client et serveur, se heurte aux temps de latence propres aux communications réseaux. Il arrive fréquemment que le temps d'attente de la réponse d'une requête soit plus long que le temps de traitement nécessaire à la réalisation du service. En rapprochant client et serveur dans un même sous-réseau, voire sur un même site, on les place dans un environnement où les temps de réponse des interactions sont limités, ce qui permet de réduire d'autant les temps de latence.
Le dernier avantage à souligner vient des brèves périodes de communication. En réduisant le plus possible les communications distantes aux seuls transferts d'agents mobiles, on diminue considérablement les périodes de connexion entre deux sites. Cette diminution de la fenêtre d'utilisation des communications réseaux permet de moins se soucier des ruptures de liens physiques qui peuvent intervenir fréquemment dans les environnements sans fil[21].

1.1.7.2 Des calculs indépendants

Dans le modèle classique d'évaluation à distance, le client et le serveur doivent rester connectés tant que le service est en cours d'exécution. En plus des problèmes liés aux fluctuations des performances réseaux, certains services, nécessitant de longues phases de traitement, ne supportent pas facilement une rupture de connexion avec le client. Dans ce cas, ils doivent souvent redémarrer entièrement leurs calculs. Mais, le maintien du lien de communication peut s'avérer difficile dans des réseaux à large échelle ou sans fil. Avec les agents mobiles, un client peut déléguer les interactions avec le service sans maintenir une connexion de bout en bout. Avec cette possibilité d'un calcul indépendant, le client peut demander un service, se déplacer (ou simplement terminer une session) puis venir récupérer les résultats plus tard. Ce mode de fonctionnement est particulièrement intéressant lors du lancement à distance de simulations numériques[4].

1.1.7.3 Optimisation du traitement

L'optimisation des phases de traitement se produit à deux niveaux. Premièrement, comme nous l'avons déjà dit, en localisant les ressources et le savoir faire sur un même site, on supprime les phases de dialogue entre le client et le serveur qui sont perturbées par des temps de latence dus aux communications réseaux. Ensuite, le déplacement du savoir faire va permettre de déléguer les calculs sur des machines serveurs, un super-calculateur par exemple, qui sont généralement plus puissantes qu'une machine cliente. Cela est particulièrement vrai dans l'informatique nomade où la miniaturisation s'accompagne d'une perte de puissance significative[4].

1.1.7.4 Tolérance aux fautes physiques

En se déplaçant avec leur code et données propres, les agents mobiles peuvent s'adapter facilement aux erreurs systèmes. Ces erreurs peuvent être d'ordre purement physique, disparition d'un nœud par exemple, ou d'ordre plus fonctionnel, arrêt d'un service par exemple. Si on prend le cas d'un site perdant une partie de ses fonctionnalités, un service tombant en panne, l'agent pourra alors choisir de se déplacer vers un autre site contenant la fonctionnalité désirée. Ceci permet une bien meilleure tolérance aux fautes que le modèle statique classique[4].

1.1.8 Agents mobiles et applications réparties

Ces avantages font des agents mobiles une technologie adéquate et bénéfique pour différents domaines d'application :

1.1.8.1 Le commerce électronique

Le commerce électronique peut se définir comme toute transaction dont les parties utilisent comme support de transmission un support électronique. Avec l'évolution du réseau des terminaux mobiles et la possibilité de les utiliser pour se connecter sur le réseau Internet, nous avons vu l'apparition d'une nouvelle forme de commerce électronique, le M-commerce.
Le M-commerce ou mobile commerce est une forme du e-commerce où une transaction peut s'effectuer à partir d'un PC mais également à partir d'un terminal mobile comme un téléphone mobile ou un assistant personnel (smartphone). Le développement de l'ubiquitous computing (n ordinateurs pour 1 personne) amène à penser que les agents mobiles vont devenir un outil essentiel dans le cadre du commerce électronique sur l'Internet. En effet, les utilisateurs de ces services Web désirent continuer leurs transactions quel que soit le lieu où ils se trouvent, et quelle que soit la machine qu'ils utilisent. Cela implique une forte hétérogénéité des machines disponibles pour exécuter une seule et même transaction commerciale (ordinateur de bureau, téléphone portable, SetTopBox de télévision interactive). Dans ce cadre, l'agent mobile va jouer le rôle de l'utilisateur, pour rassembler les informations, les adapter au terminal auquel elles sont destinées, et les transmettre pour permettre de poursuivre l'exécution de l'application. Ci-après nous citons quelques exemple d'agents mobiles dans le commerce en ligne :

MAgNET : acronyme de (Mobile Agents for Networked Electronic Trading) est un système de négociation électronique en réseau, basé sur le framwork Aglets. Le site de l'acheteur maintient une liste de fournisseurs potentiels avec leurs listes de produits. L'acheteur qui est intéressé par l'acquisition d'un produit crée un agent mobile, spécifie les critères pour l'acquisition du produit, et distribue l'agent mobile aux fournisseurs potentiels. L'agent mobile se rend à chaque site de fournisseur, cherche les catalogues de produits selon des critères de l'acheteur, et revient à l'acheteur avec la meilleure affaire qu'il trouve. Soit l'acheteur confirme la transaction et procède à la transaction monétaire, soit il annule la requête.

Supplier driven marketplace : cet autre modèle de e-commerce utilisant les agents est appelé 'Supplier driven market-place'. Cette approche est particulièrement intéressante pour les produits ayant une courte durée de vie. Un fournisseur crée et distribue un agent mobile pour les acheteurs potentiels, en lui donnant une liste de sites à visiter. L'agent mobile porte avec lui les informations sur les stocks disponibles et le prix du produit.

1.1.8.2 La recherche d'information sur le Web

Avec le développement du réseau Internet, la plupart des fournisseurs de services ont développé leur interface Web. Cette interface leur permet de présenter leurs produits et de donner des informations (numéro de téléphone, plan d'accès etc.) sur la société. L'adoption de cette solution permet aux utilisateurs du réseau Internet d'avoir accès au monde à travers leurs ordinateurs à moindre coût (Internet est presque gratuit). Actuellement, le modèle « client/serveur », où les échanges se font par envoi de messages à travers le réseau, est le modèle le plus utilisé. Ce modèle possède l'inconvénient d'augmenter le trafic sur le réseau et exige une connexion permanente. Dans le but de diminuer les échanges intermédiaires, nous pouvons envisager d'utiliser la technologie d'agents mobiles. Ainsi l'utilisateur crée un agent dont le but est de visiter les sites et de lui retourner les résultats.

1.1.8.3 L'administration du réseau

L'utilisation de la technologie d'agents mobiles pour l'administration du réseau permet de réduire les coûts de communication sur un réseau à faible débit. L'exécution d'agents mobiles sur les éléments du réseau permet de répartir des analyses traditionnellement effectuées sur la plate-forme d'administration, ce qui offre l'avantage de diminuer la charge de cette dernière. Les agents mobiles peuvent aussi être utilisés pour réaliser de façon décentralisée des tâches d'administration impliquant plusieurs éléments de réseau. L'exécution de tâches répétitives comme par exemple l'installation ou la mise à jour de logiciels, peut être déléguée à des agents mobiles.

1.1.8.4 Le calcul distribué et les agents mobiles

Un programme parallèle peut être vu comme un ensemble de tâches réparties sur le réseau ; ces tâches utilisent le réseau pour communiquer et pour échanger des données. L'un des problèmes principaux de l'implantation d'un algorithme sur une architecture distribuée est celui de la répartition de charges sur les machines disponibles. Le placement dynamique des tâches nécessite la migration de ces dernières entre les différentes machines du réseau.

Cette migration devient opérationnelle grâce à l'utilisation de la technologie des agents mobiles. Dans une application distribuée, une tâche est représentée par un agent et la terminaison de celle ci correspond à la fin de la vie de l'agent. Les échanges d'informations entre les tâches correspondent à la communication entre les agents. Quant au déplacement d'une tâche, il correspond à la migration d'un agent à travers le réseau, il s'agit de ce fait, de représenter une tâche par un agent mobile.

1.1.8.5 Les agents mobiles et l'informatique nomade

Une application distribuée est constituée de sites variés inter-connectés par des réseaux de divers types. En particulier, certains sites peuvent être mobiles (ordinateur portable, smartphone, etc.) et peuvent être connectés au réseau Internet par l'intermédiaire d'une connexion sans fil à faible débit. Une application distribuée intégrant des terminaux mobiles et des réseaux sans fil ajoute des contraintes inhabituelles dans les systèmes distribués traditionnels. Ainsi, les ordinateurs portables sont sujets à de fréquentes déconnexions volontaires (à l'initiative de l'utilisateur) ou involontaires (suite à une interruption temporaire de communication dans un réseau sans fil par exemple). De plus, les réseaux sans fil sont caractérisés par un faible débit, un coût de communication élevé et une faible qualité de service.
L'utilisation de la technologie d'agents mobiles permet de surmonter les problèmes liés à la déconnexion des sites. En effet, un terminal mobile crée un agent mobile et lui demande d'agir pour son compte. L'agent créé, après sa migration, peut s'exécuter dans le système même si le site mobile créateur fonctionne en mode déconnecté. Une fois le site du client connecté, il va contacter l'agent mobile afin de lui demander de revenir sur son site d'origine. Cependant, l'exécution d'un environnement d'agents mobiles dans le contexte d'un système d'informatique nomade pose plusieurs problèmes. Le premier est relatif à l'apparition et la disparition dynamique des places pouvant accueillir les agents. Cette situation se produit lorsqu'une place s'exécute sur un site mobile et que son moyen de communication est déconnecté. Un second problème se pose au niveau du nommage des agents mobiles du fait de la mobilité des sites. Celle-ci entraîne un changement d'adresse IP ou même un changement de nom de site. Dans ce contexte, se pose le problème de la localisation des agents mobiles du fait de la mobilisation des sites d'accueil.

1.1.8.6 Le code mobile et les cartes à puce

La carte à puce a été conçue comme un médiateur électronique de confiance entre l'utilisateur et les infrastructures matérielles qu'il utilise. Une carte à microprocesseur n'est pas autonome en terme d'énergie et ne possède pas d'horloge interne, l'ensemble lui étant fourni par le lecteur auquel elle est connectée. En tant que telle, la carte migre de système d'information en système d'information au gré du déplacement de son porteur. Lors de chaque nouvelle connexion, elle assure la continuité du service et la persistance des informations critiques exploitées, mais surtout elle sécurise le code qu'elle véhicule. De ce point de vue, une carte à puce est vue sous la forme d'un serveur.
Les nouvelles cartes à puce sont devenues des supports mobiles d'application. Ainsi, un programme placé dans une carte est amené à migrer et à s'exécuter sur différents terminaux d'accueil pour représenter le porteur de la carte. En ce sens, tout code encarté est un code mobile. Par ailleurs, les dernières générations de cartes permettant de charger de

nouveaux programmes tout au long de leur cycle de vie, peuvent être elles-mêmes vues comme des structures d'accueil pour code mobile. Des travaux montrent la réalisation d'une plate-forme d'accueil pour code mobile sur une carte à puce, il s'agit de la plate-forme CAMILLE [22].

1.1.9 Limites des agents mobiles

Les agents mobiles ne pourront pas remplacer intégralement le modèle classique du client/serveur lequel s'applique parfaitement dans un grand nombre de cas. Les agents mobiles ne peuvent donc pas être utilisés à grande échelle sans avoir résolu leurs problèmes inhérents au développement, à la sécurité et à la fiabilité.

1.1.9.1 Le développement

Le domaine des agents mobiles se heurte à de fortes contraintes lors des phases de développement. La première d'entre elles est qu'il existe à l'heure actuelle un grand nombre d'intergiciels pour agents mobiles qui possèdent leurs propres défauts et qualités. Avec cette offre pléthorique, il est difficile de parler de standardisation et aucune ne semble encore s'imposer. En sachant que les agents doivent s'adapter aux conditions de l'environnement, les développeurs sont confrontés à un éventail de choix trop large.

1.1.9.2 La sécurité

Le problème de sécurité des agents mobiles n'est pas encore intégralement résolu. C'est d'ailleurs le principal argument avancé pour expliquer la faible utilisation de ce paradigme. En effet, les agents mobiles représentent un nouveau champ d'investigation pour le domaine de recherche en sécurité, d'une part dans la protection des sites vis-à-vis des agents malveillants et d'autre part dans la protection des agents vis-à-vis des sites malveillants.

Protection des sites La protection des sites contre des attaques menées pas des agents malveillants est un problème qui est aujourd'hui bien maîtrisé. En effet, plusieurs solutions permettent maintenant de se prémunir d'éventuelles attaques et voici les méthodes les plus connues :

- Bac à Sable : La technique du bac à sable consiste à exécuter un agent à l'intérieur d'un environnement restreint, en interdisant l'accès au système de fichiers par exemple. Ainsi, un site peut exécuter un agent douteux dans le bac à sable sans trop se soucier des problèmes de sécurité. Cette approche peut facilement se mettre en place en utilisant des interpréteurs de code dont les possibilités sont limitées. Pour illustrer cette technique nous pouvons citer les applet Java exécutées à l'intérieur d'un navigateur Web.
- Signature de code : La signature de code intervient lors de la création d'un agent, son créateur le signant numériquement afin qu'il puisse s'identifier durant ses déplacements. Cette technique permet d'obtenir une authentification de haut niveau pour les sites. Les applet Java adoptent désormais ce mode de fonctionnement. Ainsi si une applet est signée, elle est considérée comme un code de confiance et peut accéder à toutes les fonctionnalité de Java. Elle sera placée dans un bac à sable dans le cas contraire.

– Contrôle d'accès : Pour améliorer les deux précédentes techniques, on met en place une politique de contrôle d'accès plus complexe. On peut la voir comme un raffinement d'une politique de bac à sable générale vers une politique spécifique à chaque application ou classe d'agents. En fonction des agents, le site pourra autoriser l'accès à un ensemble précis de fonctionnalités. Le contrôle d'accès permet de mixer les deux techniques en offrant aux agents signés plus de fonctionnalités qu'un simple bac à sable sans pour autant accéder à toutes les fonctionnalités.

Protection des agents À l'opposé de la protection des sites, la protection des agents contre des sites malveillants ne dispose pas de solution éprouvée et reste encore aujourd'hui un champ de recherche ouvert. Pour comprendre ce que risque un agent lors de son exécution sur un site malveillant, nous pouvons référencer les éléments transportés pouvant être cible d'attaque :

– Le code : ensemble des instructions composant la tâche de l'agent.

– Les données statiques : données ne changeant pas durant les déplacements (la signature par exemple)

– Les données collectées : ensemble des résultats obtenus au cours des déplacements réalisés par l'agent depuis son lancement.

– L'état courant : ensemble de données servant à l'exécution courante de l'agent.

La sécurité des agents mobiles consiste alors à garantir les critères de confidentialité et d'intégrité de l'ensemble de ces éléments. Du point de vue des données, il est évident qu'un agent ne souhaite pas divulguer des informations critiques à n'importe quel site. Par exemple, un site malveillant pourrait récupérer la signature d'un code et l'utiliser pour créer un nouvel agent afin de s'introduire dans des environnements auxquels il n'a normalement pas accès. Pour le code, un agent transporte un savoir-faire propre à son concepteur qui pourrait tomber aux mains de ses concurrents.
Le problème le plus important empêchant l'adoption actuelle des agents mobiles est sans nul doute la sécurité. En effet, même si la protection des sites est quasiment assurée, celle des agents reste un réel problème qui n'a pas de solution définitive. Différentes études ont été menées pour permettre d'obtenir un niveau de sécurité satisfaisant[4].

1.1.9.3 La fiabilité

Le modèle d'exécution de l'agent mobile implique son interaction avec plusieurs sites ; ce qui expose l'agent à une éventualité de disparition à cause de la défaillance ou de la déconnexion soudaine et imprévue d'un site sur lequel il s'exécute. La disparition d'un agent entraîne un dysfonctionnement de l'application basée sur ce dernier. Dans les systèmes d'agents mobiles, une erreur qui se produit ne peut être prise en compte par son site d'origine notamment dans le cas d'une déconnexion.
Une application distribuée sûre doit pouvoir continuer de fonctionner en cas de défaillance d'une partie du système. Pour certains types d'applications, il est essentiel que les environnements d'exécution d'agents mobiles offrent des mécanismes de tolérance aux pannes[3].

1.2 La tolérance aux pannes

La recherche en tolérance aux pannes des agents mobiles a été très active. Les approches proposées sont différentes et chacune possède des points forts et des points faibles. Cependant, l'utilisation de différents modèles et approches rend la comparaison difficile. Avant de présenter les concepts de la tolérance aux pannes et les approches proposées nous allons passer en revue les concepts de la sûreté de fonctionnement d'un système informatique d'une manière générale.

1.2.1 Sûreté de fonctionnement

La tolérance aux fautes est l'aptitude d'un système informatique à accomplir sa fonction malgré la présence ou l'occurrence de fautes, qu'il s'agisse de dégradations physiques du matériel, de défauts logiciels, d'attaques malveillantes, d'erreurs d'interaction homme-machine. Elle apparait comme un moyen de garantir une sûreté de fonctionnement (dependability)[23].
La sûreté de fonctionnement consiste à connaître, évaluer, prévoir, mesurer et maîtriser les défaillances des systèmes technologiques[24].

1.2.1.1 Entraves à la sûreté de fonctionnement

Les entraves à la sûreté de fonctionnement sont de trois types : les fautes, les erreurs et les défaillances. Fautes, erreurs et défaillances sont liées par des relations de causalité illustrées sur la figure 1.6 :

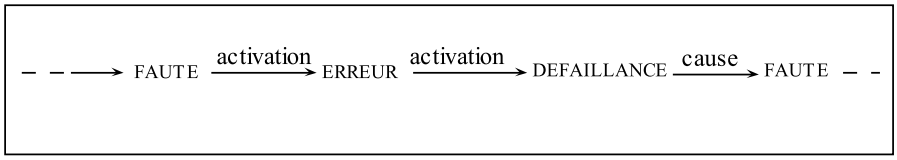

FIGURE 1.6 – La chaîne des entraves à la sûreté de fonctionnement.

Une défaillance (ou panne) est l'évènement qui survient lorsque le comportement du système dévie de sa spécification. **L'erreur** est la partie de l'état du système qui est susceptible d'entrainer une défaillance. La **défaillance** survient lorsque l'erreur affecte le service délivré à l'utilisateur. La faute est définie comme la cause certaine ou supposée de l'erreur.
Les pannes peuvent être classées selon plusieurs critères : leur degré de gravité, leur degré de persistance et leur nature. Un premier classement des pannes selon le degré de gravité est proposé dans [25] :

- pannes franches (crash fault) : soit le système fonctionne normalement (les résultats sont corrects), soit il ne fait rien. Il s'agit du modèle de panne le plus simple auquel on essaie de se ramener chaque fois que cela est possible ;
- pannes byzantines : le système peut faire n'importe quoi, y compris avoir un comportement malveillant.

Un second classement des pannes selon la nature de la panne :

- pannes accidentelles : elles se produisent de manière accidentelles ;
- pannes intentionnelles : elles sont créées avec une intention qui peut être malicieuse.

1.2.1.2 Moyens d'assurer la sûreté de fonctionnement

Les moyens utilisés pour assurer la sûreté de fonctionnement sont définis par les méthodes et les approches utilisées pour assurer cette propriété. Les approches les plus connues sont [23] :

- La prévention des fautes qui s'attache aux moyens permettant d'éviter l'occurrence de fautes dans le système. Ce sont généralement les approches de vérification des modèles conceptuels ;
- L'élimination des fautes qui se focalise sur les techniques permettant de réduire la présence de fautes ou leurs impacts. Cela est réalisé par des méthodes statiques de preuve de la validité du système (simulation, preuves analytiques, tests,...) ;
- La prévision des fautes qui prédit l'occurrence des fautes (temps, nombre, impact) et leurs conséquences. Ceci est réalisé généralement par des méthodes d'injection de fautes afin de valider le système relativement à ces dernières ;
- La tolérance aux pannes (fautes) qui essaye de fonctionner en dépit des fautes. Le degré de tolérance aux pannes se mesure par la capacité du système à continuer à délivrer son service en présence de fautes.

1.2.2 La tolérance aux pannes dans les systèmes répartis

L'objectif de la tolérance aux fautes consiste à éviter les défaillances du système malgré la présence de fautes. Cela revient à casser la chaine décrite à la figure 1.6, qui conduit de la faute à la défaillance. La tolérance aux fautes est mise en œuvre par la détection d'erreur et le rétablissement du système.

1.2.2.1 Détection d'erreur

La détection d'erreur peut être réalisée lors d'une suspension de service. On dit alors qu'elle est préemptive. À l'opposé, on dit qu'elle est concomitante lorsqu'elle est réalisée lors de l'exécution normale du service. Les techniques de détection concomitante utilisent la redondance au niveau information ou composant, ou la redondance temporelle ou algorithmique[23]. Les formes les plus utilisées sont les suivantes :

- Les codes détecteur d'erreur : ils introduisent une redondance dans la représentation de l'information.
- Le doublement et la comparaison : les unités de traitement sont dupliquées et leurs résultats sont comparés.
- Les contrôles temporels et d'exécution : un « chien de garde » (watchdog) contrôle les temps de réponse de l'exécution.

1.2.2.2 Rétablissement du système

Le rétablissement du système vise à transformer l'état erroné en un état exempt d'erreur et de faute. Le traitement de la faute se fait en identifiant le composant fautif et en l'excluant.

Le traitement de l'erreur peut se faire par trois techniques : la reprise, la poursuite et la compensation.

- La reprise est la technique la plus couramment utilisée. L'état du système est sauvegardé régulièrement. Lorsqu'une erreur est détectée, le système est ramené à un état antérieur à l'occurrence de l'erreur. Cet état sauvegardé est appelé point de reprise.
- La poursuite consiste à rechercher un nouvel état exempt d'erreur. Ceci peut par exemple être réalisé en associant un traitement exceptionnel lorsqu'une erreur est détectée.
- La compensation nécessite que l'état du système comporte suffisamment de redondance pour permettre sa transformation en un état exempt d'erreur. Elle est transparente vis-à-vis de l'application car elle ne nécessite pas de réexécuter une partie de l'application (reprise), ni d'exécuter une procédure dédiée (poursuite). Elle peut par exemple être réalisée en répliquant des composants et en effectuant un vote majoritaire sur les résultats.

1.2.3 Gestion de la redondance

La tolérance aux fautes dans un système distribué est assurée par l'utilisation de la redondance. Cette redondance peut être spatiale (réplication de composants), temporelle (traitements multiples) ou informationnelle (redondance de données, codes, signatures). Les mécanismes de redondance mis en œuvre appartiennent à deux catégories : les mécanismes utilisant la réplication et les mécanismes s'appuyant sur une mémoire stable.

1.2.3.1 La réplication

La tolérance aux fautes par réplication consiste à utiliser des copies multiples d'un même composant sur des processeurs différents. Cette approche par réplication rend possible le traitement des pannes en les masquant. La principale difficulté de cette approche est de conserver une cohérence forte entre les copies. Il existe quatre stratégies principales permettant d'assurer cette cohérence :

- Pour la réplication passive, on distingue la copie primaire et les copies secondaires. La copie primaire est la seule qui reçoit les requêtes et qui effectue toutes les opérations. Pour assurer la cohérence, la copie primaire diffuse son nouvel état aux copies secondaires après chaque modification. Cet état sert de point de reprise en cas de défaillance.
- La réplication active désigne les stratégies dans lesquelles toutes les copies jouent un rôle identique. Toutes les copies reçoivent la même séquence ordonnée de requêtes, qui sont toutes traitées dans le même ordre. Cette stratégie évite d'utiliser des points de reprise coûteux. En revanche, elle nécessite un mécanisme de diffusion atomique et requiert que l'exécution des requêtes soit déterministe pour garantir la cohérence.
- La réplication semi-active est une amélioration de la réplication active. À la différence de la réplication active, les copies secondaires attendent une notification de la copie primaire avant de traiter la requête. Cette notification comporte les informations nécessaires qui permettent de résoudre le problème de l'indéterminisme du traitement des requêtes.

– La réplication coordinateur/cohortes est également une solution hybride entre la réplication active et la réplication passive. La copie primaire est appelée coordinateur et les copies secondaires sont appelées cohortes. Cette méthode est une réplication passive pour laquelle les requêtes sont transférées à toutes les copies pour éviter de les perdre en cas de défaillance[23].

1.2.3.2 Redondance informationnelle : mémoire stable

La mémoire stable représente un support de stockage. Son rôle est de conserver les sauvegardes des informations du système qui permettront de reprendre l'exécution de l'application dans un état cohérent. Une mémoire stable doit préserver l'intégrité des données et les garder accessibles, même en cas de défaillance.
Un support de stockage est vu comme une mémoire stable si et seulement si les trois conditions suivantes sont vraies :

1. Accessibilité : il existe à tout moment de l'exécution un chemin permettant d'accéder aux données sauvegardées même en présence des pannes.
2. Protection : les pannes affectant le système ou l'application n'altèrent pas les données.
3. Atomicité : les mises à jour des données sur le support de stockage se font de manière atomique.

Le point critique pour réaliser la tolérance aux pannes par mémoire stable dans les systèmes répartis est la constitution d'un état de reprise cohérent qui répond aux objectifs attendus de la tolérance aux pannes. Deux approches ont été proposées dans la littérature pour construire un état global cohérent d'un système réparti :

– A priori : les sauvegardes des différents états des processus coopérants sont coordonnées pour constituer un état global cohérent ;
– A posteriori : les sauvegardes des états des processus sont indépendantes ; la reconstitution d'un état global cohérent se fait lors de la reprise.

1.3 La tolérance aux pannes dans les systèmes d'agents mobiles

La capacité d'identification du modèle de pannes joue un rôle très important si l'on veut réaliser la tolérance aux pannes. En effet, les conséquences et le traitement d'une panne diffèrent selon son modèle[25].

1.3.1 Modèles de fautes dans les systèmes d'agents mobiles

Dans les systèmes d'agents mobiles, les machines, les places, ou les agents peuvent tomber en panne et peuvent ensuite reprendre l'exécution (recouvrement). Un composant (machine, place ou agent) en panne, mais pas encore recouvert est dit down, sinon il est up. Le composant et dit correct, s'il ne tombe jamais en panne (ne devient pas défaillant) lors de l'exécution de l'agent mobile.
Les défaillances de machines, places, ou agents sont appelées défaillances d'infrastructure. Une situation de blocage se produit si un composant défectueux empêche la progression de l'exécution de l'agent mobile.

FIGURE 1.7 – Le modèle de fautes dans les systèmes d'agents mobiles.

Panne de l'agent : Ce document porte principalement sur les pannes franches (pannes permanentes, crash). Lorsqu'un agent tombe en panne franche, c'est de manière définitive. Il n'émet et ne reçoit aucun message avant recouvrement. Un processus est en panne par omission (sur émission ou sur réception) s'il n'émet ou s'il ne délivre pas la réponse attendue suite à certaines requêtes pendant une certaine période. On parle également de pannes transitoires, pannes passagères ou de pannes intermittentes lorsque ces pannes sont répétitives. Finalement, on parle de panne temporelle lorsque la spécification du protocole est respectée, mais pas dans les temps imposés.
Ces pannes sont toutes produites de manière non intentionnelles. Elles sont qualifiées de accidentelles. Ainsi, soit l'agent est correct et il exécute exactement ce qu'il doit faire, soit il est incorrect, dans ce cas il n'agit pas.
Inversement, une partie importante des travaux en algorithmique répartie concernent des pannes byzantines. Ces pannes sont produites par les agents dans le but de corrompre le système. C'est un cas extrême englobant les types précédents de pannes puisqu'un processus byzantin peut toujours simuler une panne par omission ou par une panne franche[26].

Panne du site hôte : Quand on parle de site hôte on sous-entend la machine physique, qui fait partie du réseau dans lequel l'agent migre ainsi que la place qui s'exécute sur cette machine offrant ainsi l'environnement d'exécution logique pour l'agent mobile.
La défaillance du site hôte, que ce soit la place où s'exécute l'agent ou la machine sur laquelle se trouve cette place peut causer la perte de l'agent.
La défaillance d'une machine provoque la défaillance de toute place et tout agent s'exécutant sur cette machine. La défaillance d'une place cause la défaillance de tout agent sur cette place, cette panne peut être provoquée par une surcharge, indisponibilité de ressources ou une erreur de programmation. Cela n'affecte généralement pas la machine mais la rend incapable de recevoir ou d'exécuter des agents.

Panne des liens de communication : Les pannes peuvent concerner les liens de communication. Un message peut ainsi être perdu mais jamais altéré ou reproduit. Les liens de communication qui perdent les messages sont équivalents à des liens de communication en pannes qui peuvent être récupérés plus tard. La panne des liens de communication peut engendrer le morcellement du réseau, où un ensemble de nœuds peut se retrouver complètement isolé. Notons qu'un nœud ne restera jamais isolé[5].

La panne des liens des communication peut se produire durant la migration de l'agent mobile, dans ce cas l'agent est perdu.

1.3.2 Propriétés des agents mobiles tolérants aux pannes

L'apparition d'une faute dans l'environnement d'exécution d'un agent mobile engendre la perte partielle ou totale de cet agent. Même dans le cas où le propriétaire de l'agent garde une copie de son code et de son état interne, des problèmes de défaillances peuvent survenir et avoir un impact sur l'exécution des agents mobiles. La défaillance d'un agent (ou d'une machine) peut l'empêcher de continuer son exécution ; plus grave encore, l'état actuel de l'agent, et même son code, peuvent être perdus.
Pour contourner ce problème, le propriétaire de l'agent peut essayer de détecter la défaillance de cet agent en envoyant un nouvel agent. Par contre, ceci nécessite un mécanisme de détection de défaillance très fiable, capable de faire la distinction entre un agent défaillant et un agent qui aurait été retardé par des processeurs ou des liens de communication lents. Malheureusement, ceci n'est pas possible dans un environnement ouvert tel que l'Internet. Cette incertitude peut mener à deux situations :

- Le propriétaire de l'agent considère que l'agent est perdu alors qu'il ne l'est pas. Lancer un autre agent va causer des exécutions multiples du même agent.
- Le propriétaire de l'agent attend le retour d'un agent défaillant. Ceci conduit à une situation de blocage.

Pour cela, tout agent mobile tolérant aux pannes doit assurer les deux propriétés suivantes :

1. Le non blocage (non blocking) : L'agent mobile ne doit pas resté bloqué. Malheureusement, le blocage ne peut être évité si toutes les places défaillent en même temps. Pour cela, chaque approche de tolérance aux pannes a des limites dans le nombre de fautes à tolérer.
2. Une seule fois (*Exactly_once*) : L'exécution d'un agent vérifie la propriété d'exécution 'une seule fois' si et seulement si l'agent réalise son itinéraire dans l'ordre fixé et chacune des actions est exécutée exactement une fois par étape.

1.3.3 Les techniques de tolérance aux pannes pour les agents mobiles

Les développements récents dans le domaine de la tolérance aux pannes des agents mobiles ont donné lieu à plusieurs méthodes basées sur des techniques et des modèles différents.
Les techniques de tolérance aux pannes des systèmes d'agents mobiles sont dans la plupart des cas des hybrides de plusieurs techniques de redondance, utilisées dans les systèmes répartis, qui sont adaptées pour convenir au modèle de fautes des agents mobiles. Selon [27], nous distinguons trois principales techniques de tolérance aux pannes dans les systèmes d'agents mobiles :

- Les points de reprise(Checkpointing)
- La réplication temporelle (Temporal Replication)).
- La réplication spatiale (Spatial Replication).

1.3.3.1 Les points de reprise (checkpoint)

Cette technique consiste à injecter une copie de l'agent et de son état sur une unité de stockage juste après son arrivée à la place courante. En cas de panne, l'agent peut reprendre l'exécution à partir de l'état sauvegardé. En particulier, si l'application prend des sauvegardes périodiques de l'agent il serait possible de relancer l'agent à partir de la dernière sauvegarde. Pour restreindre l'impact de la panne sur d'autres agents dans le même groupe de communication de l'agent défaillant, la journalisation des messages est utilisée.
L'utilisation des points de reprise de l'état et du code de l'agent sur la place courante empêche la perte de l'agent. Cependant, dans le cas où la machine tombe en panne, l'agent est bloqué et la sauvegarde est indisponible jusqu'au recouvrement de la machine figure (1.8)[27].

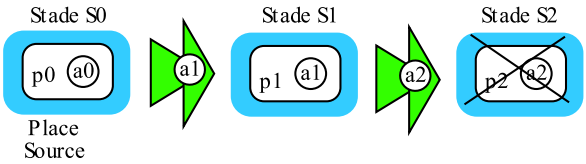

FIGURE 1.8 – Blocage de l'agent dans le cas d'une panne.

1.3.3.2 La réplication temporelle

Elle est basée sur l'approche de points de reprise présentée précédemment. Mais, au lieu de stocker le code et l'état de d'agent sur la même place, une copie de celui-ci est maintenue sur la place précédente (voir figure 1.9)[27]. Dans ce cas, p_1 permet d'envoyer a_2 à la place p'_2, si elle détecte la défaillance de p_2. Ainsi, p_1 surveille son successeur. Cette approche évite la situation de blocage dans le système de points de reprise.

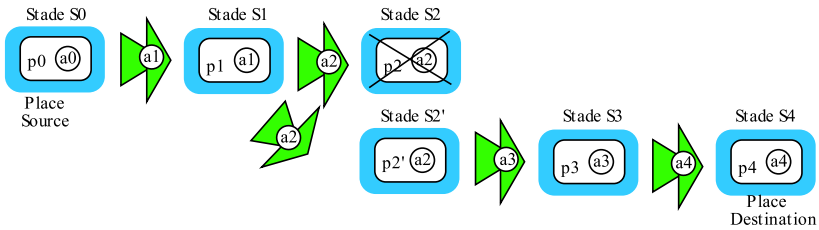

FIGURE 1.9 – Principe de la réplication temporelle.

1.3.3.3 La réplication spatiale

Dans cette approche, au niveau de chaque stade S_i, l'agent est exécuté sur un ensemble de places $M_i = p_i^0, p_i^1, p_i^2, ...$ Une place p_i envoie une copie de l'agent à toutes les places dans l'ensemble M_{i+1} du stade suivant S_{i+1} (figure 1.10)[27]. L'agent a_{i+1} est exécuté sur l'une de ces places. Si cette place est défaillante, l'agent a_{i+1} n'est pas perdu, car les autres places dans

M_{i+1} ont également reçu a_{i+1}, le blocage est évité par une autre place qui prend en charge l'exécution de a_{i+1}.

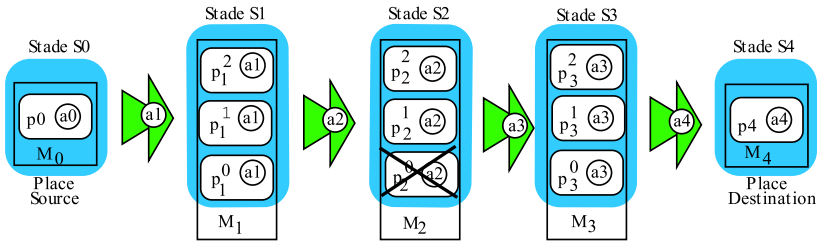

FIGURE 1.10 – Principe de la réplication spatiale.

1.3.4 Les approches de tolérance aux pannes des agents mobiles

La grande majorité des chercheurs classent les approches de tolérance aux pannes dans les systèmes d'agents mobiles selon la technique de redondance utilisée (temporelle, spatiale ou points de reprise). Par contre, dans [28],[5],[29], les auteurs proposent une nouvelle classification pour les approches de tolérance aux pannes des agents mobiles, cette classification est basée sur la validation de l'exécution de l'agent dans le temps et dans l'espace :

- Après chaque stade (Approche Commit-After-Stage)
- A la destination (Approche Commit-At-Destination).

Commit-After-Stage Les approches Commit-After-Stage consistent à valider les actions de l'agent mobile juste après la terminaison de l'exécution de chaque stade S_i avant le déplacement de l'agent au stade suivant S_{i+1}. Dans ce contexte, nous distinguons deux cas :

- (1) L'exécution de l'agent sur une place unique
- (2) l'exécution de l'agent sur un ensemble de places (utilisation de la réplication).

La validation peut être décidée par une seule place comme elle peut être distribuée. La décision de validation et l'exécution de l'agent peuvent partager une ou plusieurs places (co-localisées).
Les combinaisons de ces trois critères conduisent à huit classes, qui peuvent être représentées dans un espace à trois dimensions : (1) emplacement de l'exécution de l'agent, (2) l'emplacement de la validation, (3) la co-localisation.

Commit-At-Destination Contrairement aux approches Commit-After-Stage, les actions de l'agent mobile sont validées à la fin de l'exécution de tous les stades. Lorsque l'agent arrive à la destination, il envoie un message aux différentes places de l'itinéraire pour valider les actions de l'agent. Il peut retourner lui même vers ces places pour valider ses propres actions. Cette approche est intéressante dans certains types d'application notamment les transactions réparties ou la validation doit s'exécuter quand toutes les actions sont terminées avec succès. Cette approche permet d'assurer la propriété d'exécution unique dans le cas où des doublons empruntent deux chemins différents. Seule la destination, entité de validation, est une place commune. Par conséquent, une seule exécution est validée,

habituellement, le premier agent arrivé, tandis que les autres agents sont implicitement annulés.

Chaque approche de tolérance aux pannes des agents mobiles se distingue par la manière de gérer le compromis entre le non-blocage et la propriété d'unicité (*exactly_once*). Dans ce qui suit, nous allons présenter les approches de tolérance aux pannes en spécifiant les approches bloquantes et les approches non bloquantes.

1.3.4.1 Les approches bloquantes

Points de reprise Le principe de cette technique est de remplacer l'état d'erreur par un état correct en se basant sur l'abstraction d'une mémoire stable. Ceci nécessite tout d'abord la détection de l'erreur, c'est à dire, identifier la partie incorrecte de l'état, et pouvoir ensuite effectuer le recouvrement d'erreur.
Dans [30], les auteurs adoptent cette approche en intégrant deux stratégies ; le point de reprise et le recouvrement arrière. Cette technique consiste à sauvegarder l'état intermédiaire de l'agent mobile sur un support de stockage à intervalles de temps réguliers durant la période d'exécution sans panne. Quand une panne survient, le dernier état de reprise sauvegardé permettra de restaurer l'état de l'agent mobile. Les informations de reprise sauvegardées permettent à l'agent mobile de reprendre les calculs à partir d'un état intermédiaire au lieu de reprendre les calculs depuis le début.

Transactions et votes L'approche proposée dans [31] utilise une réplication spatiale pour assurer le non blocage. Chaque stade est exécuté par une seule place appelée *worker*. La progression de l'exécution du *worker* est contrôlée par un ensemble de places appelées *observers*. Ce protocole assure la propriété *exactly_once* à l'aide d'une série de transactions locales suivie d'un vote. Chaque place implémente une file de messages transactionnelle utilisée pour transférer l'agent entre deux stades consécutifs comme suit :

- Le worker et les observers prennent l'agent dans la file d'entrée,
- Le worker exécute intégralement le code une et une seule fois,
- Le worker place l'agent dans la file de sortie,
- Le worker valide la transaction (c'est à dire, commit).

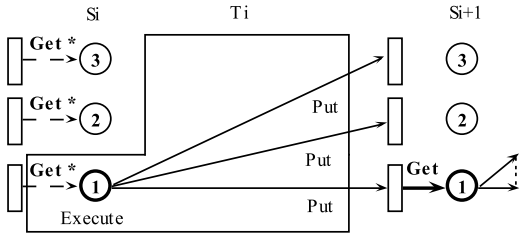

FIGURE 1.11 – Exécution transactionnelle sur chaque stade

Le protocole choisit le *worker* qui va exécuter le stade courant sous forme de transaction locale. Si le *worker* ne réussit pas à valider l'exécution (à cause d'une panne), les *observers* relancent le protocole pour choisir un nouveau *worker*. La validation des transactions locales

se fait via le protocole de validation à deux phases (2PC). Le fait que le protocole 2PC soit bloquant rend cette approche bloquante.

L'arrière garde L'arrière garde ou 'rear guard' a été introduite initialement dans le protocole NAP [2] proposé par [32]. L'arrière garde est une place que l'agent a déjà visitée et qui contrôle l'exécution de l'agent mobile sur la place suivante. Si l'agent tombe en panne son arrière garde exécute un code de recouvrement spécifique. Les auteurs supposent que la détection de pannes est fiable, ce qui n'est pas toujours possible.
Dans [33, 34], les auteurs ont proposé des techniques de tolérance aux pannes en utilisant le principe de l'arrière garde. Dans cette approche, l'agent mobile est appelé *master*, l'arrière garde est un clone de l'agent mobile appelé *shadow*. L'agent *master* est créé par la place source pour exécuter une tâche sur une séquence de sites qui forment son itinéraire. Initialement le *master* crée un agent *shadow* sur la place source. Ensuite, avant chaque migration, le *master* crée un *shadow* sur la place courante et envoie un message de terminaison à l'agent *shadow* sur la place précédente. Le *shadow* suit l'exécution de l'agent mobile et se charge de relancer les calculs quand il détecte une panne de l'agent. De même, si l'agent *master* détecte la panne de l'agent *shadow*, il crée un nouveau *shadow* sur l'un des sites précédents.
Cette solution est valable dans le cas des pannes simples. Par contre, quand l'agent et son 'rear guard' (*shadow*) tombent en panne en même temps, l'agent est perdu.
Afin de contourner ce problème, les auteurs dans [35], ont proposé des techniques de tolérance aux pannes en utilisant plusieurs arrières-gardes. Cependant, l'arrière-garde n'est pas un clone de l'agent mobile mais un agent de type différent appelé 'witness'. L'agent mobile effectue des sauvegardes de son code et son état avant chaque migration. L'agent de contrôle (witness) utilise ce chekpoint pour créer un nouvel agent mobile en cas de panne. L'ensemble des agents 'witness' créés tout au long de l'itinéraire forme une chaîne de contrôle dans laquelle chaque agent surveille son successeur et le dernier ajouté à la chaîne contrôle l'agent mobile :

$$\omega_0 \rightarrow \omega_1 \rightarrow \ldots \omega_{i-1} \rightarrow \omega_i \rightarrow \alpha$$

Ils supposent que l'agent de contrôle sur la place source est correct (ne tombe jamais en panne).
Ce mécanisme de contrôle a été amélioré par [36]. Afin de minimiser la dépendance de la chaîne, les auteurs supposent que pas plus de k nœuds ne tombent en panne à la fois. Dans ce cas, k agents de contrôle sont suffisants pour assurer la disponibilité de l'agent mobile. La longueur de la chaîne doit être toujours inférieure ou égale à k, la dépendance devient comme suit :

$$\begin{array}{c} \text{si } (i < k)\ \omega_0 \rightarrow \omega_1 \rightarrow \ldots \rightarrow \omega_i \rightarrow \alpha \\ \text{sinon } \omega_{i-k+1} \rightarrow \omega_{i-k+2} \rightarrow \ldots \rightarrow \omega_i \rightarrow \alpha \end{array}$$

Ces approches considèrent uniquement la défaillance des agents. Dans ce cas, si le site où s'exécute l'agent de contrôle tombe en panne, la chaîne de dépendance est rompue jusqu'au recouvrement du site.
Pour cela, les auteurs dans [37] apportent une autre amélioration. Ils essaient de maintenir la dépendance de la chaîne en cas de panne du serveur d'agent de contrôle (la machine où

2. NAP pour 'Norwegian Army Protocol', ce protocole était motivé par une approche de détection de faute et de recouvrement utilisée par une troupe militaire lors d'un déplacement dans un territoire hostile

s'exécute l'agent de contrôle). L'idée consiste à envoyer l'adresse de chaque site visité à tous les agents de la chaîne de contrôle. Par conséquent, chaque agent de contrôle maintient une table qui contient les adresses de tous les sites visités par l'agent mobile et donc tous les sites où se trouvent les agents de contrôle. Si un site d'un agent de contrôle tombe en panne, son prédécesseur dans le chaîne de contrôle va détecter la panne, et va recouvrer la dépendance de la chaîne comme suit :

- Retirer l'agent suspecté de la chaîne de contrôle,
- Créer une nouvelle liaison de contrôle avec le successeur dans la chaîne de contrôle (voir figure1.12).

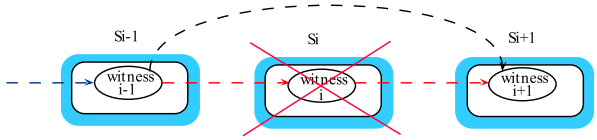

FIGURE 1.12 – Changement de la dépendance en cas de panne du serveur d'agent de contrôle

Les approches basées sur l'arrière garde se focalisent sur la résolution de la panne de l'agent. Elles supposent que les sites se rétablissent après une panne et que l'agent mobile reste en attente jusqu'au recouvrement du site défaillant. Ceci peut provoquer des situations de blocage si le site défaillant n'est pas rétabli.
Pour éviter ce cas de blocage, les auteurs dans [38] proposent de faire une seconde migration de l'agent en cas de détection d'une défaillance. Si la place précédente suspecte la place courante, elle envoie l'agent mobile vers une autre place disponible. Cependant cette seconde migration peut violer la propriété *exactly_once* étant donné que la détection de panne n'est pas fiable. Pour contourner ce problème, les auteurs définissent le stade dit *orphelin*.
Un stade orphelin désigné par ϕS_i apparaît lorsqu'une place soupçonne par erreur la place courante et envoie l'agent vers une nouvelle place (voir figure 1.13).

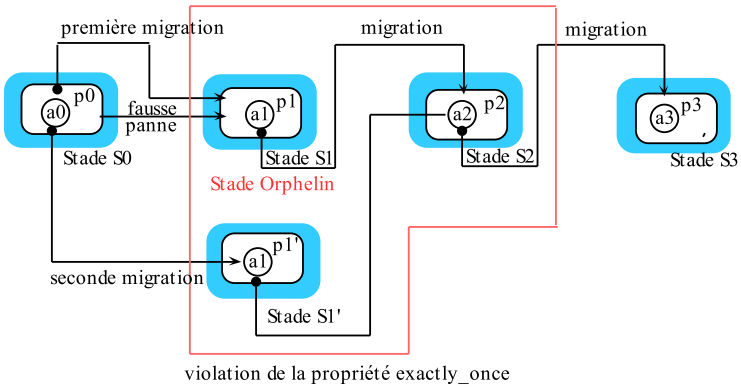

FIGURE 1.13 – Apparition du stade orphelin

Afin d'éviter la violation de la propriété d'unicité (*exactly_once*), la place du stade orphelin (c'est à dire la place suspectée) demande une autorisation auprès de la place précédente,

avant de laisser l'agent migrer vers le stade suivant. Si la place précédente a déjà lancé un agent sur une autre place (seconde migration), elle confirme qu'il s'agit bien d'un stade orphelin et envoie un message pour annuler l'exécution de l'agent pour arrêter sa progression. Dans le cas contraire, la place précédente autorise la place courante à valider l'exécution de l'agent et à continuer sa migration.
Pour leur part, les auteurs dans [39] ont utilisé dans le même contexte, le principe de l'arrière garde. L'agent mobile laisse toujours derrière lui un agent *shadow* pour le contrôler. La nouveauté dans cette approche est que l'agent mobile retourne régulièrement vers la place source après quatre stades d'exécution pour sauvegarder son état (il effectue un chekpoint). Lorsqu'une panne survient au niveau de l'agent mobile et de son arrière garde, l'agent n'est pas complètement perdu. La place source est en mesure de relancer l'agent à partir de la dernière sauvegarde effectuée.
Les auteurs fondent des hypothèses fortes sur les liens de communication lesquels sont considérés comme toujours fiables. La place source demeure également disponible.
D'autre part, le fait que les toutes les sauvegardes (checkpoints) sont effectuées au niveau de la place source risque de créer une surcharge. De plus le temps d'exécution de l'agent est influencé par les aller-retours vers la place source.

1.3.4.2 Les approches non bloquantes

James Dans [19] les auteurs intègrent plusieurs techniques de tolérance aux pannes dans une plateforme appelée 'JAMES'. Cette plateforme supporte la panne de l'agent, de la place, de la machine et des liens de communication (morcellement du réseau).
La plateforme définit des gestionnaires d'agents qui ont en charge le contrôle des agents. L'agent mobile s'exécute sur toutes les places de l'itinéraire (exécution atomique) ou sur le maximum possible (best-effort). Lorsque la panne d'un agent est détectée, la place qui détient la copie la plus récente de l'agent relance l'exécution. Cette place est choisie à l'aide d'un protocole d'élection (généralement la place précédente). Cette approche permet d'éviter le blocage, mais les doubles exécutions peuvent se produire.
JAMES utilise un répertoire de recherche pour prévenir les doubles exécutions. Ce répertoire est répliqué sur un ensemble de nœuds appelés nœuds de surveillance. Chaque agent qui termine son exécution insère une entrée dans ce répertoire. Si cette entrée existe déjà (ce stade a déjà été exécuté par un autre agent) alors cet agent annule son exécution et s'autodétruit.
Le répertoire de recherche n'est pas suffisant pour assurer la propriété *exactly_once*. Examinons l'exemple suivant : l'agent s'exécute sur la place p_{i+1} donc l'entrée correspondant au stade i indique que l'exécution de l'agent sur la place p_i s'est terminée avec succès. Supposons que les places p_i et p_{i+1} sont suspectées par les places précédentes, l'exécution d'un protocole d'élection choisit la place p_{i-1} comme étant celle qui contient le dernier état sauvegardé de l'agent. Lorsqu'une place recommence l'exécution du stade i, elle consulte le répertoire de recherche et va trouver que ce stade est déjà exécuté. Donc, soit l'agent est arrêté (risque de blocage), ou alors il décide d'ignorer l'entrée dans le répertoire de recherche et continue son exécution (violation de la propriété d'unicité si la détection de panne est fausse).
Dans le cas du morcellement du réseau, les doubles exécutions ne sont détectées qu'à la destination (place commune pour tous les agents). Dans ce cas, JAMES propose de valider l'exécution du premier agent arrivé, les autres sont annulés.

Les modifications fréquentes du répertoire de recherche sont coûteuses puisque toutes les copies de ce répertoire doivent être mises à jours pour assurer la cohérence.

Fatomas Pleisch et Schiper [27],[28],[41] proposent une approche appelée Fault-Tolerant Mobile Agent System (Fatomas). Chaque stade d'exécution est composé de plusieurs places vers lesquelles l'agent est répliqué (réplication spatiale). Une seule place exécute l'agent, les autres l'observent, si elle tombe en panne une autre place reprend l'exécution de l'agent. Ainsi le non blocage est assuré.
La détection de pannes n'étant pas fiable, des exécutions multiples peuvent se produire suite à de fausses suspicions. Afin d'assurer la propriété *exactly_once*, les places du même stade ont besoin de se mettre d'accord pour le choix de la place qui exécute l'agent. Pour cela, les auteurs modélisent l'exécution de l'agent par une séquence de problèmes d'accord. A la fin de chaque stade, un DIV consensus (Deferred Initial Value) est exécuté pour choisir une seule place, appelée primaire. Seule la place primaire peut valider l'exécution du stade courant, les autres places annulent les modifications apportées par l'agent mobile. Le consensus choisit également l'ensemble des places qui constitueront le prochain stade.

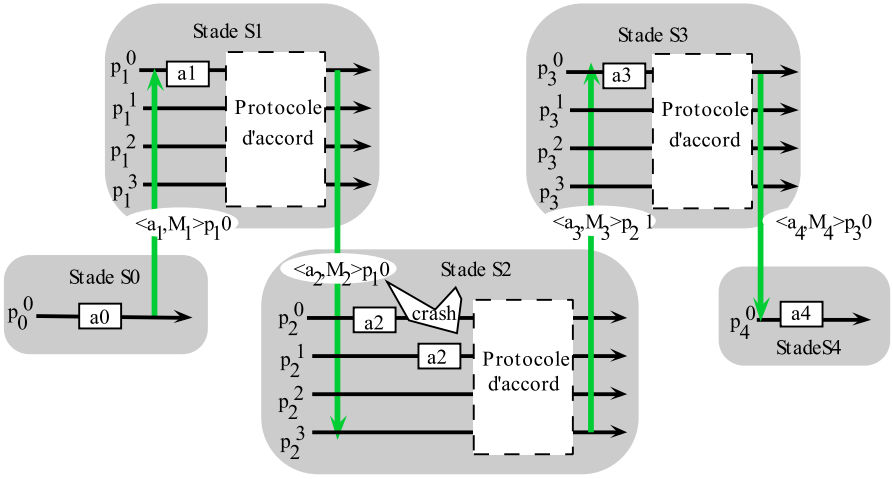

FIGURE 1.14 – Approche Fantomas.

La technique proposée introduit un coût en termes de performance dû à l'envoi de plusieurs agents vers un ensemble de places pour chaque stade d'exécution, augmenté par l'exécution d'une instance de consensus après chaque stade.

NetPebbles [40] définit sa proposition dans l'environnement NetPebbles où un agent est défini comme un script qui se déplace entre les places. La tolérance aux pannes est basée sur le non déterminisme où l'agent effectue un choix aléatoire entre un ensemble de nœuds alternatifs à chaque migration. L'agent est contrôlé par son prédécesseur via l'envoi des battements de cœur. Un nœud est en panne s'il est réellement en panne ou si tous les nœuds alternatifs vers lesquels il mène sont en panne. Si une panne est détectée sur un nœud, le nœud précédent envoie l'agent vers un nœud alternatif. Si tous les nœuds alternatifs ne sont pas disponibles, le nœud courant est considéré en panne. L'agent est restauré en uti-

lisant l'état sauvegardé à l'étape précédente pour essayer de faire un autre choix (retour arrière) ; Les modifications effectuées par l'agent sur le nœud courant sont annulées. Lors d'une fausse détection de panne une double exécution de l'agent peut avoir lieu. A chaque étape d'exécution, l'itinéraire de l'agent est comparé à son dernier itinéraire connu par le nœud. S'il contient un nœud déclaré défaillant ce dernier est ignoré. Si un nœud de l'itinéraire tombe en panne, c'est à dire son prédécesseur ne reçoit plus de battements de cœurs, ce dernier restaure l'agent et l'envoie vers un autre nœud alternatif et adresse un message d'annulation à tous les nœuds suivants dans l'itinéraire.

1.4 Conclusion

Le modèle agents mobiles est très adapté pour effectuer des tâches de longue durée sur des réseaux distants limitant ainsi l'utilisation de la bande passante inter-réseaux. Cependant, ce modèle présente des limites qui freinent son utilisation à large échelle.
Dans ce chapitre, nous avons étudié un des problèmes importants des agents mobiles à savoir la tolérance aux pannes.
Nous avons introduit les concepts de base de la sûreté de fonctionnement dans les systèmes répartis. Ensuite, nous avons abordé la tolérance aux pannes dans les systèmes d'agents mobiles.
Les techniques de tolérance aux pannes utilisées ont été présentées ainsi que les approches les plus représentatives avec les différentes classifications existantes dans la littérature.
Comme nous l'avons déjà mentionné, notre objectif est de proposer une approche de tolérance aux pannes pour les systèmes d'agents mobiles transactionnels. Pour cela, le chapitre suivant est consacrer à l'introduction des agents mobiles transactionnels.

Chapitre 2

Agents mobiles et support transactionnel

Les propriétés des agents mobiles, à savoir, l'autonomie et la mobilité les rendent très adaptés aux transactions réparties. Les agents mobiles transactionnels sont une extension du modèle agent mobile pour supporter l'exécution atomique des transactions réparties. L'intérêt d'une exécution transactionnelle est qu'elle assure (i) une exécution correcte (en terme de fiabilité) d'une transaction prise individuellement et (ii) des exécutions correctes (en terme de cohérence de données) de plusieurs transactions concurrentes.
Dans ce chapitre nous détaillons l'approche transactionnelle dans les systèmes répartis puis le modèle d'exécution des agents mobiles transactionnels et leur spécificités. Ensuite, nous présenterons un état de l'art sur les approches des agents mobiles transactionnels proposées dans la littérature. Nous terminerons par une comparaison des différentes approches.

2.1 L'approche transactionnelle

L'approche transactionnelle a émergé initialement dans le contexte des bases de données. Le modèle des transactions permet d'encapsuler une séquence d'opérations de lecture ou d'écriture sur une base de données comme une unité atomique de traitement, appelée transaction.

2.1.1 Définition d'une transaction

Une transaction est une séquence d'actions s'exécutant comme un ensemble, et qui agit sur des objets d'une base de données (attributs, enregistrements...). Les transactions ont comme objectif de permettre une exécution concurrente des requêtes et des mises à jour et de garantir qu'elles ne sont pas perdues. Deux aspects sont pris en compte dans les implémentations des transactions : le contrôle de la concurrence et la gestion des pannes[42].

2.1.2 Exécution d'une transaction

L'exécution d'une transaction est caractérisée par le fait qu'une et une seule des assertions suivantes est vraie :

- Toutes les opérations de l'ensemble sont effectuées (ce cas correspond à la validation de la transaction) ;
- Aucune des opérations n'est effectuée (ce qui correspond au rejet, ou à l'abandon de la transaction).

Quelle que soit la cause du rejet, les effets partiels de la transaction sont annulés. Ce comportement est garanti par la propriété d'atomicité des transactions (appelée aussi tout ou rien)[42].
Les états d'exécution d'une transaction sont présentés dans la figure 2.1.

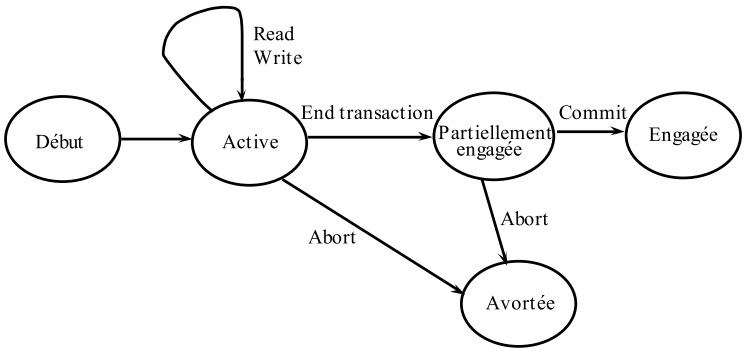

FIGURE 2.1 – États d'exécution d'une transaction locale

A l'état *active* la transaction effectue des opérations de lecture et écriture. Quand le calcul est terminé, la transaction peut passer à deux états possibles : *Avortée* ou *Partiellement engagée*. la transaction passe à l'état *partiellement engagée* pour se préparer à valider la transaction, ensuite elle passe automatiquement à l'état engagée lorsque le calcul est validé. Si par conte le transaction ne peut pas valider les résultats du calcul, elle passe de l'état *Active* à l'état *Avortée* pour annuler les opérations effectuées.
Afin d'assurer le déroulement et la gestion des transactions, les systèmes de gestion de base de données utilisent des techniques de verrouillage et d'estampillage.

2.1.3 Propriétés d'une transaction

Les opérations sont appliquées à des objets qui peuvent être partagés (plusieurs applications utilisant ces objets peuvent s'exécuter de manière concurrente). De plus, les objets peuvent être modifiés. Par conséquent, les objets partagés doivent être protégés contre les incohérences engendrées par des accès concurrents incontrôlés. Les transactions ont cette capacité de protection grâce à leur propriété d'isolation. Cette propriété garantit que, si plusieurs transactions s'exécutent de manière concurrente, alors tout se passe comme si chacune s'exécutait seule dans le système.
Lorsque les transactions sont utilisées dans les bases de données, elles sont caractérisées par une troisième propriété : la cohérence. Cette propriété garantit que, si une transaction qui s'exécute seule dans le système modifie des données dans une base et si la base se trouve

dans un état cohérent avant l'exécution de la transaction, alors celle-ci passe dans un état cohérent après l'exécution de la transaction[42].
La dernière propriété des transactions est la permanence ou la durabilité. Celle-ci garantit que les effets d'une transaction validée ne peuvent pas disparaître suite à des pannes logicielles ou matérielles.
Les quatre propriétés décrites ci-dessus sont désignées dans la littérature par le sigle ACID (Atomicity, Consistency, Isolation, Durability)[42]. Une transaction dite ACID possède les propriétés suivantes :

- Atomicité : La séquence d'actions d'une transaction est indivisible : soit toutes les actions de la transaction sont exécutées et la transaction est validée (le système atteint un nouvel état cohérent), soit aucune ne l'est et la transaction est rejetée (le système reste dans l'état cohérent initial). C'est le principe du tout ou rien.
- Cohérence ou Consistance : L'exécution d'une transaction dans un environnement sans concurrence et sans pannes ne doit pas introduire d'incohérence.
- Isolation : Les actions d'une transaction sont isolées. Le résultat des actions intermédiaires (état temporairement incohérent) est masqué aux autres transactions. Une autre manière de présenter cette propriété est que le résultat d'une exécution parallèle de plusieurs transactions est équivalent à celui d'une exécution séquentielle. On parle alors de sérialisation (en théorie la sérialisation est plus restrictive que l'isolation).
- Permanence ou Durabilité : Les résultats d'une transaction qui s'est bien terminée ne peuvent pas être détruits ultérieurement par une quelconque panne.

Si toutes ces propriétés sont vérifiées, on parle de cohérence forte (ou cohérence stricte). Si une des propriétés n'est pas vérifiée la cohérence est dite dans un état dégradé ou faible.

2.2 Les transactions distribuées

2.2.1 Définition

Une transaction distribuée est un ensemble d'opérations, dans lesquelles deux ou plusieurs hôtes du réseau sont impliqués (figure 2.2). La transaction distribuée, comme toute autre transaction, doit avoir les quatre propriétés dites ACID citées ci-avant.
Les transactions distribuées utilisent le service d'un coordinateur ou moniteur de transaction.
Les systèmes de transaction traditionnels fonctionnent en deux étapes pour garantir le bon fonctionnement d'un processus distribué entre les participants. Au cours de la première étape dite "phase de préparation", un participant doit maintenir les changements d'états, qui ont eu lieu au cours de l'exécution de la transaction, de sorte que ces modifications puissent être annulées ou « validées » ultérieurement, une fois que l'état final de la transaction a été déterminé. Dans le cas où aucune défaillance n'a eu lieu pendant la première phase, la deuxième phase dite "phase de commit" vise à remplacer l'état d'origine avec l'état issu de l'exécution de la transaction. Dans ce cas, les changements au cours de la première phase deviendront durables[43].

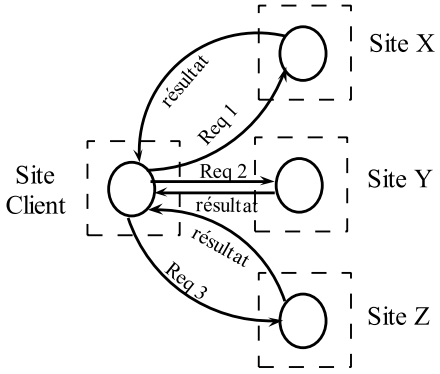

FIGURE 2.2 – Transaction distribuée.

Il n'est pas certain qu'une transaction répartie puisse être validée, malgré le fait qu'elle se soit déroulée normalement jusqu'au point de terminaison. Par exemple, elle a pu modifier des objets se trouvant sur un site qui est ensuite tombé en panne. Si les effets de la transaction sont perdus sur le site défaillant, la transaction ne peut pas être validée. Autrement dit, la validation d'une transaction répartie nécessite un protocole permettant d'établir un consensus entre les participants de la transaction.
La mise en œuvre d'un tel protocole est compliquée par le fait qu'il doit être résistant aux pannes. Par ailleurs, il est souhaitable qu'il puisse permettre aux participants d'abandonner la transaction à tout moment (par exemple, pour débloquer une ressource afin qu'une autre transaction puisse y accéder). En fait, ceci n'est pas toujours possible, chaque protocole ayant *une fenêtre de vulnérabilité* dans laquelle les participants n'ont pas le droit d'abandonner la transaction unilatéralement. Un participant entre dans la fenêtre de vulnérabilité lorsqu'il est prêt à valider ; après avoir communiqué cette décision aux autres participants, il reste bloqué jusqu'à l'obtention d'un consensus sur le mode de terminaison (par validation ou par abandon)[42].

2.2.2 Moniteur de transaction

Les transactions distribuées utilisent le service d'un coordinateur ou moniteur de transactions qu'on note TM. Celui-ci peut être le premier serveur contacté par la transaction distribuée, ou bien un moniteur transactionnel dédié.
Le moniteur de transactions (TM) gère les transactions et leur affecte des identificateurs uniques. Tout sous-système qui participe à la transaction distribuée est géré par un moniteur de ressources (RM). Lors des annulations d'opérations (appelés les ROLLBACKs), le TM appelle les ROLLBACK de tous les RMs.

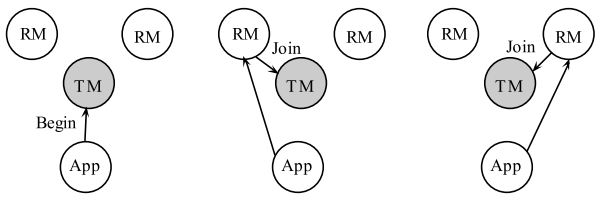

FIGURE 2.3 – Moniteur de transactions (TM) dans une transaction distribuée.

2.3 Les modèles de transactions avancés

L'approche transactionnelle a été étendue et adoptée par des applications plus avancées voulant bénéficier de cette fiabilité et de cette correction. Ces applications varient de simples interactions avec plusieurs bases de données à l'automatisation de procédés métiers. Cependant, le modèle ACID s'est avéré limité pour supporter de telles applications qui (i) nécessitent des structures de contrôle plus complexes qu'une simple séquence d'opérations, (ii) ont des exécutions de longue durée et (iii) où la sémantique de l'annulation n'est pas aussi évidente que pour les opérations de bases de données classiques.
Les modèles transactionnels avancés (MTA) ont été alors proposés pour répondre à ces nouveaux besoins. Dés lors, une transaction désigne plus généralement toute unité de travail encapsulant un ensemble d'opérations. Les MTA permettent de grouper des opérations dans une structure hiérarchique et de relâcher, dans la plupart des cas, certaines des propriétés ACID (généralement l'isolation et l'atomicité). Ces modèles peuvent être classés selon plusieurs caractéristiques, à savoir : la structure de la transaction, la concurrence intra-transaction, les dépendances d'exécution, les besoins d'isolation et de semi-atomicité.

2.3.1 Les transactions avec points de reprise et les transactions enchaînées

L'objectif est de permettre la définition de points de contrôle intermédiaires, afin de permettre un retour en arrière partiel (et non pas total, comme dans le cas de l'abandon). Les points de reprise sont des points où, à la demande du programme, l'état de la transaction est sauvegardé. Il s'agit d'une sauvegarde non persistante des données utilisées par la transaction. Le programme peut ensuite demander le retour à un point de reprise donné. Cependant, en cas de panne, la transaction est annulée.
Dans le cas des transactions enchaînées, les points de contrôle sont des points de validation du travail déjà effectué par la transaction (on ne peut plus annuler les effets ainsi validés, mais on peut revenir en arrière au dernier point de validation). L'avantage de ce modèle par rapport au précédent est que la transaction peut libérer, aux points de contrôle, des ressources qu'elle n'utilise plus[42].

2.3.2 Modèle des transactions emboitées

Une transaction emboîtée est un ensemble de sous-transactions qui peuvent récursivement contenir d'autres sous-transactions, formant ainsi un arbre de transactions emboîtées(figure 2.4). Une transaction fille ne peut démarrer qu'après que sa transaction mère ait démarré. Une transaction mère ne peut se terminer que lorsque toutes ses transactions filles sont terminées. La validation d'une transaction est conditionnée par la validation de sa transaction mère. L'abandon d'une transaction entraîne l'abandon de toutes ses transactions filles. Cependant, une transaction mère peut survivre à l'échec d'une de ses transactions filles en exécutant, par exemple une sous-transaction alternative.[42].

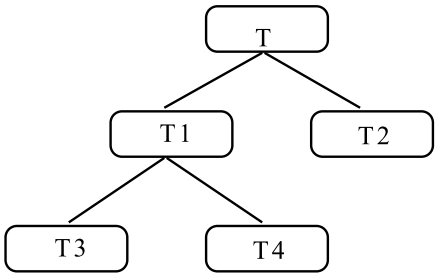

FIGURE 2.4 – L'arbre représentant une transaction emboîtée

Les transactions emboîtées ouvertes relâchent l'isolation en rendant visibles les résultats des sous-transactions validées aux autres transactions emboîtées s'exécutant en parallèle. Ainsi une sous-transaction peut valider et libérer ses ressources avant que sa transaction mère ne se termine correctement et valide. En cas d'abandon d'une transaction, le travail exécuté par ses sous-transactions validées peut être sémantiquement compensé en exécutant des sous-transactions de compensation. Une transaction de compensation t' annule sémantiquement le travail d'une transaction t. Ainsi l'état d'une base de données avant et après l'exécution de la séquence tt' est un état éventuellement différent de l'état initial avant l'exécution, mais considéré comme cohérent[44].

2.3.3 Modèles des sagas

Le modèle des Sagas a été proposé comme une solution pour les transactions de longue durée. L'idée de base est de permettre à une transaction de libérer certaines ressources qu'elle a acquises avant de valider. Une transaction de longue durée ou Saga est une séquence de sous-transactions, $T_1, ..., T_n$, qui peuvent s'intercaler avec d'autres transactions. Chaque sous-transaction T_i est une transaction $ACID$ qui préserve la cohérence de la base de données.
Les exécutions partielles d'une saga sont inacceptables. Si une saga abandonne (suite à l'échec de l'une de ses sous-transactions) alors le travail réalisé jusqu'à lors (par les précédentes sous-transactions) doit être compensé. Ainsi, chaque sous-transaction T_i possède une transaction de compensation C_i qui annule sémantiquement son travail.

2.3.4 Modèle des transactions flexibles

Initialement, les transactions flexibles ont été proposées comme un modèle transactionnel convenable dans un environnement multi-bases de données. Une transaction flexible est un ensemble de tâches, chacune ayant un ensemble de sous-transactions fonctionnellement équivalentes. Un ensemble de dépendances d'exécution sur les sous-transactions peut être spécifié.
[45] adopte les sémantiques transactionnelles de tâche rejouable, compensable et pivot. Une tâche t est dite rejouable si elle se termine toujours avec succès après un nombre n_i d'activations. Une tâche est dite compensable si son travail peut être sémantiquement annulé. Une tâche est dite pivot si une fois qu'elle se termine avec succès, ses effets ne peuvent pas être compensés. [45] exploite ces propriétés transactionnelles pour définir des transactions flexibles et correctes. Une transaction flexible doit respecter certaines règles : toute tâche entre deux pivots ou avant le premier pivot doit être compensable. Après un

pivot, plusieurs branches alternatives peuvent exister avec un ordre de préférence entre elles. La dernière branche alternative après un pivot doit être sûre de se terminer. Ceci implique que toutes les tâches de la dernière branche alternative doivent être rejouables.
L'originalité de ce modèle est l'utilisation de propriétés transactionnelles en plus de la compensation permettant d'atteindre plus de flexibilité tout en préservant un certain degré de correction. Ainsi, l'utilisation des alternatives comme mécanismes de recouvrement en avant permet d'éviter d'annuler la totalité du travail effectué en cas d'échec et permet de continuer l'exécution et d'atteindre l'objectif désiré en suivant un autre chemin alternatif[44].

Le modèle transactionnel traditionnel a évolué au cours du temps pour incorporer des structures de transaction plus complexes et relâcher les propriétés d'atomicité et d'isolation. Les modèles transactionnels avancés, résultats de cette évolution, ont permis de dépasser les limites du modèle ACID pour satisfaire les besoins de nouvelles applications avancées. Bien que ces modèles aient été proposés dans des contextes plus ou moins similaires, ils partagent certaines caractéristiques :

- Une structure plus complexe qu'une simple séquence d'opérations,
- Un degré de concurrence et de parallélisme plus élevé,
- Le relâchement des propriétés d'atomicité et d'isolation,
- L'adoption de nouvelles sémantiques transactionnelles comme la compensation (qui correspond au recouvrement en arrière).

2.4 Les agents mobiles transactionnels

Un agent mobile transactionnel est un agent mobile destiné à exécuter une transaction distribuée sur des nœuds du réseau. L'ordre d'exécution des tâches est important et donc l'ordre de visite des différents nœuds est important. Cette définition pose quelques contraintes quant à l'exécution de l'agent mobile. Un agent mobile transactionnel doit satisfaire les propriétés suivantes :

- Décider de manière autonome du prochain site à visiter en cas de pannes.
- Migrer entre un ensemble d'ordinateurs et manipuler des objets sur chaque ordinateur.
- Valider la transaction seulement si la condition de validation telle que l'atomicité est remplie sinon abandonner la transaction.

2.4.1 Le modèle d'exécution atomique

La mobilité et l'autonomie des agents mobiles les rendent très adaptés aux transactions distribuées. Un agent mobile peut visiter différents nœuds du réseau pour exécuter les différentes tâches de la transaction répartie. Par ailleurs, les opérations de l'agent mobile doivent s'exécuter de façon atomique pour assurer la validité de la transaction. Ce type d'exécution qualifie les agents mobiles transactionnels.
Imaginons le scénario suivant : un agent mobile qui doit réserver un billet d'avion, réserver une chambre d'hôtel et enfin louer une voiture. Le propriétaire de l'agent exige en général que les trois opérations réussissent ensembles. En effet, louer une voiture n'a pas de sens si aucun vol n'est disponible. De même, le billet d'avion est inutile si aucune chambre ne peut

être réservée. Les agents mobiles transactionnels doivent assurer ces contraintes d'exécution atomique (tout ou rien).
En parallèle, l'atomicité de l'exécution doit être garantie même en cas de défaillance de composants logiciels ou matériels. Donc les agents mobiles transactionnels doivent être tolérants aux pannes.
Dans une transaction globale distribuée chaque partie du calcul se trouve sur un nœud du réseau. L'agent mobile va parcourir tout le réseau pour visiter les nœuds où se trouvent les parties du calcul de la transaction globale. Sur chaque nœud, l'agent mobile exécute la partie du calcul correspondante mais ne valide pas les résultats partiels jusqu'à la fin de l'exécution sur tous les nœuds de l'itinéraire.
Une fois tous les nœuds visités et les calculs sont effectués avec succès, l'agent mobile peut valider les résultats partiels sur ces nœuds ainsi que le résultat final. Pour que l'exécution de la transaction soit correcte, l'exécution de toutes les parties du calcul doit se dérouler sans fautes.
Les agents mobiles transactionnels tolérants aux pannes doivent satisfaire les spécifications des transactions à savoir les propriétés ACID (i.e., Atomicity, Consistency, Isolation, Durability) ainsi que la propriété *exactly_once* des agents mobiles tolérants aux pannes.

2.4.2 Agents mobiles transactionnels vs non transactionnels

Généralement l'exécution des agents mobiles n'exige pas l'atomicité. La progression de l'agent est arrêtée seulement dans le cas d'une panne d'infrastructure. Par nature, l'exécution de l'agent est validée sur chaque stade avant de passer au stade suivant. Dans certaines solutions, la validation de l'exécution de l'agent se fait à la destination, à ce niveau l'agent est libre de valider l'exécution sur une partie de l'itinéraire et de l'annuler sur d'autres.
Par ailleurs, les agents mobiles transactionnels exécutent des transactions et doivent assurer la propriété d'atomicité de tous les stades exécutés. L'exécution de deux stades consécutifs S_i et S_{i+1} est atomique, si et seulement si les deux stades réussissent ensembles c'est à dire l'annulation de l'un entraine l'annulation de l'autre. Si on prend l'exemple d'un agent qui achète le billet d'avion au stade S_i et réserve une chambre d'hôtel au stade S_{i+1}. Manifestement, la réservation de la chambre d'hôtel est annulée s'il n'existe pas de places dans le vol. Dans ce cas, S_i et S_{i+1} doivent s'exécuter d'une manière atomique, si l'un des deux n'est pas satisfait alors l'autre est annulé[5].
La validation de l'agent transactionnel se fait à la destination à l'aide d'un protocole qui fait participer tous les nœuds de l'itinéraire pour décider de valider ou d'abandonner la transaction exécutée par l'agent.

2.4.3 Le modèle de fautes

Nous avons vu qu'une panne d'un composant qui empêche la progression de l'exécution de l'agent mobile est une panne de type infrastructure. Cette panne peut être causée par un agent défaillant, une place en panne, une machine en crash ou alors des liens de communications non fiables. Généralement l'exécution des agents mobiles ne nécessite pas l'atomicité. Donc, les exécutions des agents sur différentes places (étapes) sont indépendantes. La faute d'exécution sur une étape n'entraîne pas l'annulation de l'autre. Dans le cas des agents mobiles transactionnels ces pannes peuvent engendrer la violation de la propriété d'atomicité de la transaction exécutée par l'agent.

2.4.3.1 Défaillance sémantique

Un échec ou défaillance sémantique est différent d'une défaillance de type infrastructure dans le sens où ni la machine, ni la place, ni l'agent mobile ne sont défaillants. Une panne sémantique n'empêche pas la progression de l'agent mais empêche la validation de la transaction. En vérité, ça se produit quand un service demandé n'est pas accompli en raison de la logique d'application ou à cause de l'échec du service. Par exemple, une demande d'un billet d'avion est refusée si aucun siège n'est disponible pour le vol demandé. Néanmoins, l'opération de l'agent, c'est à dire, la demande d'un billet, est exécutée dans son intégralité sans faute. En fait, dans cet exemple il n'y a pas de véritable "échec" qui est survenu, car le résultat est valide du point de vu du service. Toutefois, du point de vue de l'agent (i.e., le client du service), le résultat est indésirable. Nous appelons ce résultat un échec sémantique (ou défaillance sémantique).
Parmi les propriétés ACID de la transaction distribuée, l'atomicité est la plus importante. L'atomicité englobe les défaillances sémantiques en assurant que toutes les actions sont exécutées avec succès ou aucune d'elles n'est exécutée. Pour assurer l'atomicité, l'exécution de l'agent mobile décide soit de valider toutes les transactions locales ou de les annuler toutes. Comme conséquence de l'exécution séquentielle des actions de tous les stades (S_i), le résultat est seulement connu (et donc la décision est seulement possible) à la destination. A la destination, l'agent envoie un message (ou un autre agent) avec la décision (commit/abort) à toutes les places précédentes. À la réception de ce message la place valide ou annule la transaction locale.
Si une action au stade S_2 subit une défaillance sémantique, la transaction globale (distribuée) est immédiatement annulée (voir figure 2.5).

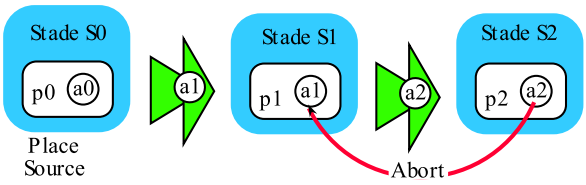

FIGURE 2.5 – Un message « Abort » est immédiatement envoyé à toutes les places

2.4.4 Propriétés des agents mobiles transactionnels

Les agents mobiles transactionnels doivent être tolérants aux pannes et donc assurer les propriétés des agents mobiles tolérants aux pannes à savoir :

- Le non blocage : Une défaillance ne doit pas empêcher la progression de l'agent mobile (vivacité).
- Exactement une fois (exactly_once) : Les actions des agents mobiles sont exécutées exactement une seule fois (sûreté).

De plus l'agent mobile transactionnel doit assurer les propriétés ACID des transactions réparties :

- Atomicité : L'exécution de tous les stades se fait de manière atomique (tout ou rien).
- Cohérence :une exécution correcte de la transaction sur un état cohérent de la place, des services offerts et les agents, doit résulter en un autre état cohérent.

- Isolation : Les changements sur chaque stade d'exécution ne sont pas visibles aux autres agents mobiles transactionnels jusqu'à la validation complète de la transaction globale.
- Durabilité : Les changements validés par la transaction sont reflétés sur le système et deviennent permanents.

2.5 Approches pour les agents mobiles transactionnels

Plusieurs travaux de recherche se sont intéressés aux agents mobiles transactionnels. Certains se focalisent sur la tolérance aux pannes et le support transactionnel, d'autres considèrent uniquement le support transactionnel.
Dans [2], les auteurs avancent une approche basée sur les transactions et l'élection de leader. L'agent mobile est répliqué en n exemplaires sur n sites. Toutefois, une seule copie effectue le calcul. Les $n-1$ autres répliques sont passives et ne prennent la relève qu'en cas de défaillance de la copie active.
Les sites secondaires observent par intervalles réguliers l'exécution de l'agent sur le site primaire. S'il tombe en panne, les sites secondaires élisent un nouveau site primaire via un protocole d'élection. Le site élu doit fournir le même service sinon, il exécute un mécanisme de recouvrement.
Les auteurs proposent un concept d'itinéraire qui permet de définir le chemin de l'agent d'une manière flexible pour supporter les pannes sémantiques (indisponibilité de service). Si un nœud que l'agent doit visiter est temporairement non disponible cette visite peut être décalée automatiquement et d'autres nœuds seront visités le temps que ce dernier soit recouvré.
Ils définissent l'itinéraire comme un ensemble d'entrées $(e_1, e_2, e_i, ...)$. Chaque entrée est un triplet $(precondition, node, method)$ qui signifie que l'agent peut migrer vers le nœud 'node' pour exécuter les actions définie dans 'method' lorsque 'precondition' est vérifiée. Un ordre de priorité existe entre les différentes entrées de l'itinéraire.
Pour tolérer les pannes sémantiques, le protocole définit des nœuds alternatifs pour chaque nœud de l'itinéraire. Pour cela, tous les chemins possibles sont calculés à l'avance en utilisant les pré-conditions ainsi que les priorités. L'agent parcourt les chemins un par un jusqu'à trouver celui qui exécute la transaction avec succès.

Dans [46], pour chaque transaction, un agent mobile appelé 'originator' est créé. Sur chaque nœud visité, l'agent mobile vérifie la disponibilité de la ressource ou du service demandé. S'il juge que l'offre est intéressante, il crée un agent statique appelé 'worker' sur ce nœud. Lorsque l'agent mobile termine son parcours, il aura placé les agents 'workers' nécessaires sur chaque nœud de l'itinéraire. Il déclenche ensuite la transaction multi-agents. Durant son premier passage l'agent mobile n'effectue aucune réservation de ressources. Toutefois, l'état des ressources peut être changé par d'autres transactions. Dans ce cas la transaction risque de ne pas être validée. Pour éviter cela, les agents 'workers' surveillent les ressources. Si l'état de la ressource ne permet plus la validation de la transaction (ressource non disponible), l'agent worker peut déclencher un abandon prématuré de la transaction en envoyant un message à l'agent 'originator', ce dernier informe les autres agents workers d'abandonner la transaction.
L'agent worker peut aussi chercher un autre nœud qui peut fournir le même service. Dans

ce cas, il migre vers ce nœud pour surveiller l'état de la ressource et informe l'agent mobile de son nouvel emplacement. L'agent worker peut réagir lorsque la ressource atteint un certain seuil. Avec la coordination de l'agent originator, le worker peut valider cette partie de la transaction et la séparer de la transaction globale. Cette séparation n'est possible que si la procédure de compensation de cette partie de la transaction existe.
Le problème qui se pose dans cette proposition est que l'emplacement de l'agent mobile doit être connu par tous les agents workers. De plus les auteurs ne traitent pas les pannes des agents et des nœuds qui risquent de bloquer l'exécution de la transaction.

Dans [47], la transaction est définie par un ensemble de tâches exécutées dans un ordre spécifique. Les tâches peuvent s'exécuter en séquentiel, en concurrence ou en alternance. L'approche proposée diminue la probabilité de blocage en autorisant des transactions parallèles à s'exécuter sur différentes parties de l'itinéraire et sont combinées de nouveau en utilisant ce qu'on appelle des médiateurs. Par exemple, avec le médiateur *XORjoin*, seule une transaction parmi les transactions parallèles doit arriver. La figure 2.6 présente un exemple d'un médiateur *XORjoin*. L'exécution de l'agent mobile transactionnel se divise en deux transactions parallèles représentées par les agents b et c. Par exemple, b_{i-1} réserve un vol avec la compagnie aérienne Air-Algérie, tandis que c_{i-1} réserve un vol avec la compagnie Aigle-Azur. Au stade S_i, le médiateur *XORjoin* garde seulement une sous-transaction (représenté par c_i et b_i), tandis que l'autre est abandonnée. L'agent a continue l'exécution du stade S_{i+1}.
Pour éliminer le problème de blocage dans le cas des transactions non parallèles, le médiator *ANDsplit* réside dans la source et le médiateur *XORjoin* à la destination.
Dans ce cas, l'exécution complète de l'agent mobile s'effectue sous forme de transactions parallèles. Cependant, exécuter des transactions parallèles pour ne valider qu'une seule à la fin, peut causer une surcharge considérable du réseau.
Pour tolérer les pannes, chaque place sauvegarde l'agent et son état dans une unité de stockage stable. Après le recouvrement d'une place en panne, l'agent est restauré puis activé.
Les places qui exécutent les médiateurs doivent être visitées par les agents qui exécutent des tâches parallèles. Ceci, rend des parties de l'itinéraire statiques. De plus, les pannes sur les parties de l'itinéraire exécutant des tâches non parallèles ou celles des médiateurs sont bloquantes.

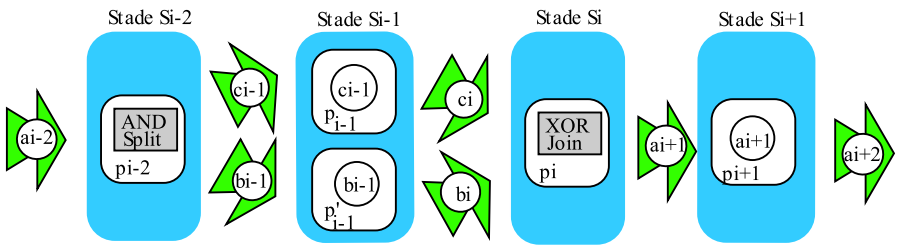

FIGURE 2.6 – Les médiateurs (rectangles) permettent l'exécution de transactions parallèles

Dans [40], l'environnement *NetPebbles* définit les agents comme des scripts qui se déplacent sur le réseau. La tolérance aux pannes est assurée en utilisant le non détermi-

nisme dans le routage de l'agent en cas de pannes. L'algorithme fonctionne comme suit : lorsqu'un agent migre d'une place p vers une place p', la place p garde une copie de l'agent et surveille l'agent sur la place p'. Si p détecte la panne de l'agent sur p' l'agent est envoyé vers une place p''. Une panne simultanée des deux places p et p' entraîne la perte de l'agent et donc le blocage de l'exécution. Ce problème est résolu en mettant en place un système de surveillance où les nœuds des étapes précédentes surveillent leurs successeurs. Chaque place envoie des battements de cœur aux places précédentes. La fréquence d'envoi des battements de cœur décroît avec la distance entre les nœuds.

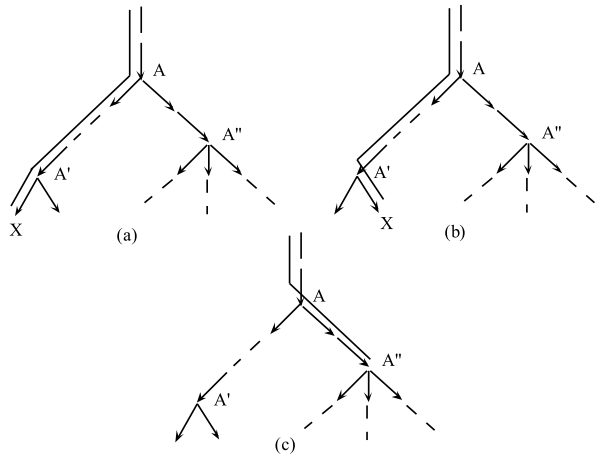

FIGURE 2.7 – Mécanisme de recouvrement en cas de pannes

Le mécanisme de détection de pannes peut entrainer l'exécution de deux copies de l'agent suite à une fausse détection de pannes. NetPebbles suppose que la destination est la même que la source. Par conséquent, tous les agents (original et copies) arrivent à la même destination. À ce stade, l'agent qui arrive en premier est validé, les autres sont annulées.
Selon [29], cette approche peut être étendue pour supporter les agents mobiles transactionnels, et ce , en exécutant un médiateur *XORJoin* à la destination.
Le problème de comment valider ou abandonner les actions des agents est laissé ouvert ?

Les auteurs dans [41] proposent une approche, appelée *Transuma*, pour les agents mobiles transactionnels, basée sur leur approche de tolérance aux pannes pour les agents mobiles. Pour masquer le blocage, l'agent est répliqué sur plusieurs places. Si une place tombe en panne, une autre place reprend l'exécution de l'agent. Comme la détection de panne n'est pas fiable, les auteurs utilisent un DIV consensus (Deferred Initial Value) pour choisir une seule place, primaire, pour valider l'exécution du stade. Dans cette approche, les actions du stade courant sur la place primaire ne sont pas validées immédiatement. Sur chaque place primaire, l'agent mobile crée un agent stationnaire qui reste en attente d'un message de validation ou d'annulation. Lorsque l'agent mobile arrive à la destination, il envoie un message 'Commit' à tous les agents stationnaires sur les places primaires visitées.

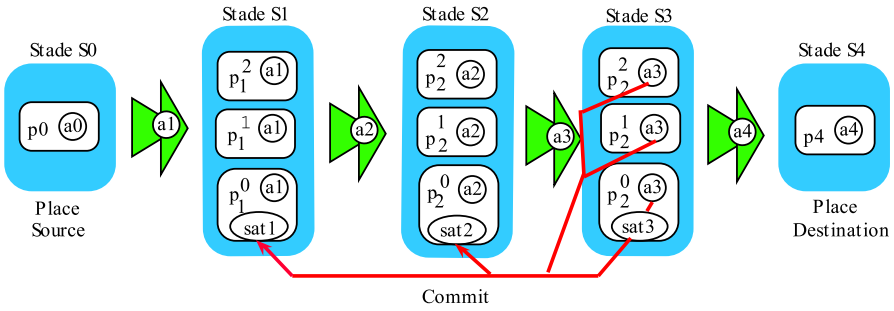

FIGURE 2.8 – Validation de l'exécution transactionnelle de l'agent mobile

Après l'exécution de chaque stade l'agent génère le message 'YES-VOTE' ou le message 'NO-VOTE' selon que le service est satisfait ou non. Si l'un des stades suivants n'est pas satisfait tous les stades précédents sont annulés immédiatement et donc l'abandon de l'exécution de l'agent mobile transactionnel.
Cette approche n'est pas intéressante dans le cas des transactions longues. Si un stade n'est pas satisfait la transaction est abandonnée automatiquement même s'il s'agit du dernier stade.
Dans [48], l'agent transactionnel est décomposé en plusieurs parties : un agent de routage et des agents de manipulation. L'agent de routage décide du prochain nœud à visiter et migre vers ce nœud. Un agent de manipulation est une partie du code qui manipule les objets locaux sur un nœud.
L'agent transactionnel laisse un agent de manipulation sur chaque nœud visité. Chaque agent de manipulation contrôle son successeur.
Le réseau est supposé fiable. Cette solution considère trois types de pannes :

- La panne du nœud où se trouve l'agent de routage : l'agent de manipulation précédent détecte cette panne, il crée une nouvelle instance de l'agent de routage qui prend une nouvelle destination.

- La panne du nœud destination (vers lequel l'agent de routage va migrer) : l'agent de routage détecte cette panne après un délai d'attente (timeout), et cherche une autre destination. Si par contre, l'agent de routage ne trouve pas un nœud alternatif il retourne vers le nœud précédent pour chercher une autre destination et l'agent de manipulation est détruit.

- La panne du nœud que l'agent de routage a déjà visité (où se trouve un agent de manipulation) : l'agent de manipulation sur le nœud précédent détecte cette panne puis envoie le message *Detect* à tous les agents successeurs y compris l'agent de routage. Quand il reçoit ce message, l'agent de routage envoie une message d'abandon *Abort* à tous les prédécesseurs puis envoie un message de recréation *recreate* à l'agent de manipulation dont le successeur direct est en panne. L'agent de manipulation crée une nouvelle instance de l'agent de routage pour trouver un nouveau chemin.

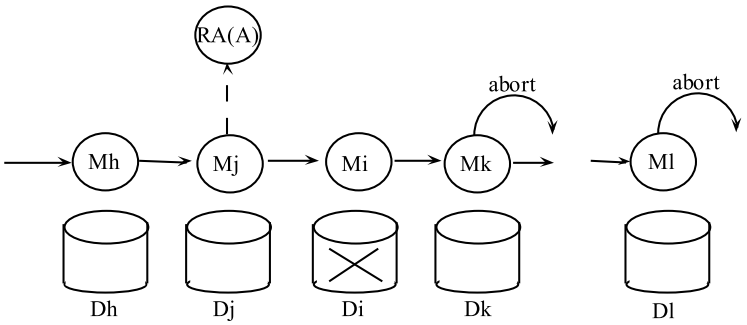

FIGURE 2.9 – Exemple de pannes sémantiques

Cette approche est intéressante sauf que les auteurs supposent le réseau fiable et donc la détection de panne est parfaite. Ceci n'est pas réaliste dans un environnement ouvert tel que Internet. Cette solution est bloquante dans un environnement où les liens de communication ne sont pas fiables : si le message *recreate* est perdu l'agent de manipulation en attente de ce message reste bloqué. De plus, dans le cas d'une fausse détection de panne de l'agent de routage, un autre agent est créé par l'agent de manipulation ce qui viole la propriété d'unicité *exactly_once*.

2.6 Synthèse

Les différentes approches de tolérance aux pannes pour les agents mobiles transactionnels proposées diffèrent légèrement dans leur façon de gérer le compromis entre la préservation des propriétés de tolérance aux pannes (exactement une seule fois et le non blocage) et la propriété d'atomicité (support transactionnel).
Dans la plupart des travaux, le blocage est évité par l'utilisation de la réplication. Quand un agent est suspecté, la place précédente crée une copie de l'agent pour trouver un autre chemin. Cependant, chaque approche assure la propriété 'une seule exécution' différemment. Dans [48], on considère une détection de panne fiable ; dans [40] cette propriété n'est pas forcée durant la phase de calcul mais comme les agents reviennent vers la place source, les multiples exécutions sont détectées et sont annulées. Néanmoins, les doubles exécutions peuvent se rencontrer sur la même place avant d'arriver à la destination. Aucune approche n'a pris en considération ce cas de figure.
La tolérance aux pannes sémantiques est très importante dans un contexte transactionnel. En fait, il existe trois types de pannes sémantiques :

1. Durant la phase d'exécution quand l'agent n'obtient pas la ressource (service non disponible) ;
2. Lorsque la ressource est attribuée à une autre transaction dans le cas d'une préréservation (préemption de ressources) ;
3. Quand une place déjà visitée tombe en panne avant la validation de la transaction.

Le premier type de panne sémantique est considéré dans presque toutes les approches. Cependant, le deuxième type est abordé seulement dans [46]. La panne des places déjà visitées est discutée dans [40] et dans [48]. Par contre il n'existe aucune approche qui traite

tous ces types de pannes simultanément.

En résumé, le Tableau 2.1 reprend la comparaison entre les approches présentées précitées.

Propriété	Panne tolérée						Caractéristiques
	agent	place	lien	Sémantique			
				1	2	3	
[2]	✓	✓	✓	✓	*X*	*X*	bloquante
[46]	*X*	*X*	*X*	✓	✓	*X*	bloquante (résout seulement les pannes sémantiques)
[40]	✓	✓	✓	*X*	*X*	✓	bloquante
[47]	✓	✓	*X*	✓	*X*	*X*	la panne de médiateurs est bloquante
[41]	✓	✓	✓	✓	*X*	*X*	non bloquante mais pas adaptée pour les transactions longues
[48]	✓	✓	*X*	*X*	*X*	✓	panne des liens est bloquante

TABLE 2.1 – Comparaison entre les approches existantes. ✓ : la panne est tolérée, *X* : la panne n'est pas tolérée.

2.7 Conclusion

Dans ce chapitre nous avons introduit le modèle d'exécution des agents mobiles transactionnels. Ces derniers doivent assurer les propriétés ACID (les propriétés d'une transaction distribuée et en particulier l'atomicité) ainsi que les propriétés de tolérance aux pannes à savoir le non blocage et l'unicité (exactly_once).
Outre les défaillances d'infrastructure, les agents mobiles transactionnels doivent également adresser les défaillances sémantiques.
L'utilisation des agents mobiles dans l'exécution des transactions réparties renforce le modèle transactionnel avec plus d'autonomie et de fiabilité par l'adoption de nouvelles sémantiques transactionnelles comme la compensation et l'alternative (qui correspond au recouvrement en avant) où l'agent explore plusieurs chemins pour exécuter la transaction.
La validation d'une transaction répartie nécessite un protocole permettant d'établir un consensus entre les participants de la transaction. Le chapitre suivant est consacré à l'étude des différents protocoles de validation atomique.

Chapitre 3

La validation atomique non bloquante

Le problème de la validation atomique (atomic commitment)[49] est au cœur des systèmes transactionnels répartis. Il a été introduit comme une abstraction du problème des transactions sur un réseau. La validation atomique a pour but d'assurer l'atomicité d'une transaction distribuée même en présence de pannes. La validation atomique garantit donc une terminaison uniforme de l'ensemble des branches qui composent une transaction distribuée. Chaque site accédé par une branche de transaction distribuée est appelé participant. Les différents participants se mettent d'accord sur l'issue d'une transaction en exécutant un protocole de validation atomique. Dans ce protocole, chacun révèle sa capacité à valider en émettant un vote. Un vote oui (ou prêt) implique que le participant s'engage à rendre permanentes les mises à jour effectuées par la transaction même s'il tombe en panne[50].

3.1 Définition

Le problème de la validation atomique est un problème d'accord entre les participants sur l'issue de la transaction. Formellement, le problème de la validation atomique est défini par les quatre propriétés suivantes, qui doivent être satisfaites par tout protocole de validation atomique :

- **Accord Uniforme :** Tous les participants qui décident (i.e., de valider ou d'abandonner), prennent la même décision.
- **Validité :** Si un participant décide de valider, alors tous les participants doivent avoir voté oui.
- **Intégrité :** Chaque participant décide au plus une fois.
- **Non-Trivialité :** Si tous les participants votent oui et aucune panne ne se produit, alors tous les participants décident de valider.

La validation atomique non-bloquante a pour but de garantir, en plus de la propriété d'atomicité, la propriété de terminaison. Cette propriété permet à tous les sites qui sont corrects d'aboutir à une décision uniforme concernant une transaction, malgré la panne d'un ou de plusieurs autres sites.

Le problème de la validation atomique non-bloquante est un problème d'accord tolérant aux pannes (c-à-d, non-bloquant) entre les participants à une transaction sur le résultat final de cette dernière. Formellement, le problème de la validation atomique non-bloquante est défini par les quatre propriétés du problème de la validation atomique (i.e., accord uniforme, validité, intégrité et non-trivialité), auxquelles s'ajoute la propriété de terminaison suivante :

– **Terminaison :** Tous les participants corrects qui exécutent le protocole de validation atomique finissent par prendre une décision.

Un protocole de validation atomique est dit non-bloquant s'il satisfait les cinq propriétés citées ci-dessus.

3.2 Les protocoles de validation atomique dans les systèmes synchrones

On suppose pour l'instant que le système est synchrone et qu'il existe donc des bornes objectives sur le délai d'acheminement des messages sur un canal de communication et de traitement des tâches. Ces bornes doivent être connues et peuvent être exploitées par tous les sites du système. Un site peut donc suspecter un autre s'il ne reçoit pas de réponse. Finalement, on suppose qu'un processus peut être relancé après une panne (récupération). Cependant, les pannes ne doivent pas être considérées comme transitoires : un processus est toujours redémarré après la fin de la décision collective.

3.2.1 Le protocole de validation à deux-phases

La validation en deux phases d'une opération distribuée est la solution qui permet de maintenir les propriétés ACID de cette opération en présence de défaillances. Lorsque la transaction a terminé son exécution, le protocole de validation à deux phases doit s'assurer que tous les processeurs impliqués dans l'exécution de la transaction (appelés participants) sont d'accord pour rendre permanents les effets de cette transaction, c'est à dire **valider** la transaction. Si l'unanimité des processeurs impliqués n'est pas obtenue, la transaction est annulée. La non-unanimité peut provenir du désaccord de l'un des processeurs impliqués ou de l'impossibilité de communiquer avec l'un des participants (ex. : processeur isolé ou défaillant).
Comme son nom l'indique, le protocole 2PC proposé par [49] se déroule en deux phases : une phase de vote et une phase de décision. On distingue deux rôles : le coordinateur est celui qui initie le protocole, tandis que les participants lui répondent. Le coordinateur a la responsabilité de diriger les participants dans une transaction distribuée afin d'aboutir à une décision uniforme.
Pendant la phase de vote, le coordinateur, averti de la fin de l'exécution, diffuse à tous les participants un message de vote leur demandant s'ils sont prêts à valider, appelé aussi message *prepare*. A la réception de ce message, chaque participant prend les mesures nécessaires pour être capables de rendre permanents les effets de la transaction en cours de validation. Puis il répond au coordinateur en donnant son accord. Le participant est alors dans un état appelé "prêt à valider" dans lequel il ne pourra plus ni valider ni abandonner sauf s'il reçoit la décision du coordinateur.

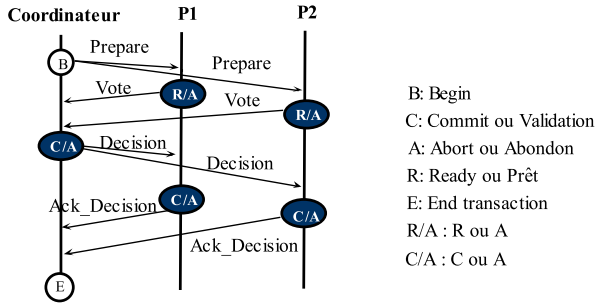

FIGURE 3.1 – Le protocole de validation à deux phases

Dans la deuxième phase, le coordinateur qui a reçu l'accord de tous les participants, diffuse l'ordre de valider. Sur réception de ce message, un participant rend permanents les effets de la transaction qui est dorénavant validée, reporte les valeurs mises-à-jour et libère les verrous. Puis, il acquitte la validation du coordinateur. Si par contre le coordinateur est dans l'impossibilité de recevoir tous les accords, celui-ci diffuse l'ordre d'annuler la transaction. Cet ordre est acquitté par les participants[50][51].
Le protocole de validation à deux phases est supporté par tous les systèmes de bases de données commerciales et a été normalisé par l'ISO (International Standardization Organization) avec son protocole OSI-TP (Open Systems Interconnection - Transaction Processing)[52], l'X/Open et son modèle DTP (Distributed Transaction Processing)[53] et l'OMG (Object Management Group) avec le service OTS (Object Transaction Service)[54].
Bien que le 2PC soit un protocole simple, il présente un inconvénient majeur. Il est bloquant : si le coordinateur tombe en panne entre la phase de vote et la phase de décision (cette période est couramment appelée fenêtre de vulnérabilité), tous les participants qui sont en attente de la décision finale seront bloqués jusqu'à la reprise du coordinateur. Ainsi, une transaction peut détenir des verrous sur des ressources partagées pour une durée indéterminée, pénalisant d'autres transactions.
Un protocole de validation est dit bloquant si, en cas de panne d'un ou de plusieurs sites, les sites corrects peuvent se retrouver bloqués jusqu'à la reprise des sites incorrects avant de pouvoir terminer la transaction.
Le protocole 2PC est bloquant : une panne du coordinateur peut bloquer les participants qui sont en attente de sa décision. Plusieurs protocoles de terminaison ont été proposés pour résoudre cette situation de blocage. Si le coordinateur tombe en panne, un participant qui est en attente de sa décision lance un protocole de terminaison. Dans ce protocole, le participant essaie de contacter un autre participant, qui a pu décider, pour tenter de récupérer cette décision.

3.2.2 Le protocole de validation à trois phases

Le 2PC a donc l'inconvénient de pouvoir bloquer. Skeen [55] a proposé le 3PC (Three Phase Commit Protocol) afin de remédier à ce problème. Le protocole 3PC est obtenu par une extension directe du protocole 2PC en ajoutant à ce dernier une phase supplémentaire, appelée phase de pré-validation. Cette phase supplémentaire va permettre à tous les sites

corrects, en cas de pannes d'autres sites, de prendre une décision uniforme pour terminer la transaction sans attendre les sites incorrects.
La phase de vote se déroule exactement de la même façon que dans le 2PC. Pendant la phase de pré-validation, le coordinateur reçoit les votes des participants. Si un ou plusieurs participants votent non, le coordinateur prend la décision finale d'abandonner la transaction. Par contre, si tous les participants votent oui, le coordinateur envoie un message de demande de pré-validation à tout le monde et se met en attente des acquittements. Lorsqu'un participant reçoit la demande de pré-validation, il envoie un message d'acquittement au coordinateur. Une fois que le coordinateur a reçu tous les messages d'acquittement, il prend sa décision finale de valider la transaction. Finalement, le coordinateur informe les participants du résultat de sa décision. Les participants qui sont en attente du résultat se conforment à cette décision et envoient un message d'acquittement de la décision[26].
La figure 3.2 montre les trois phases du protocole. Si une panne du coordinateur du 3PC est détectée (i.e., par le mécanisme de temporisation), tous les participants qui sont en attente du message de pré-validation ou de la décision lancent un protocole de terminaison. Le but de ce protocole est de terminer la transaction d'une façon uniforme sur tous les sites corrects. L'idée de base consiste à choisir un nouveau coordinateur parmi les sites corrects. Ce dernier va essayer de terminer la transaction en dirigeant les autres sites vers une validation ou un abandon. La décision du nouveau coordinateur sera basée seulement sur son état local.

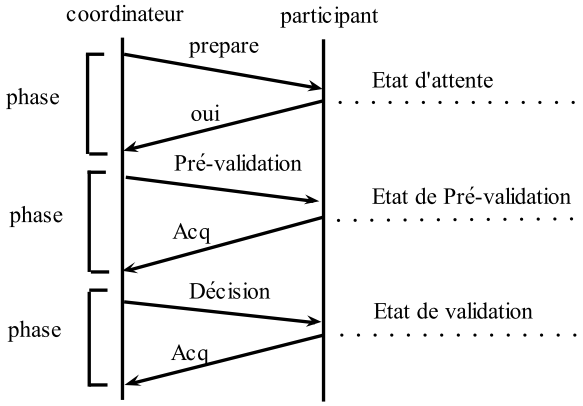

FIGURE 3.2 – Le protocole de validation à trois phases

Le nouveau coordinateur décide de valider s'il est en état de prévalidation ou de validation. Cela veut dire que l'ancien coordinateur avait reçu un vote oui de la part de tous les participants et avait commencé la phase de pré-validation avant de tomber en panne. Ainsi, le nouveau coordinateur est sûr qu'aucun autre participant n'a pu abandonner la transaction. Si le nouveau coordinateur est en état d'attente, il décide d'abandonner la transaction. En effet, son état d'attente implique qu'il n'a pas encore reçu la demande de pré-validation de l'ancien coordinateur. Par conséquent, l'ancien coordinateur ne pouvait pas avoir commencé la phase de validation avant sa panne. Donc, aucun participant n'a pu valider la transaction. Il en résulte que le nouveau coordinateur est toujours capable de prendre une décision sans compromettre l'atomicité de la transaction. Finalement, il est important de noter que le nou-

veau coordinateur fait avancer les autres participants vers son état local avant d'envoyer sa décision. Cela garantit qu'en cas d'une éventuelle panne de ce dernier, un deuxième nouveau coordinateur prendra la même décision pour terminer la transaction.
Le 3PC est non-bloquant au prix de deux séquences supplémentaires de messages (demande de pré-validation et acquittement) nécessaires pour terminer une transaction même en absence de pannes, ce qui introduit un délai très élevé dans le système. On déduit que ce protocole est peu adapté aux systèmes à grande fiabilité où le délai entre deux pannes est très élevé.
Dans les sections précédentes, le système était suffisamment synchrone pour permettre de déterminer une borne supérieure sur attente d'un processus. La question qui se pose est de savoir si la validation atomique non-bloquante est réalisable dans un système asynchrone ?

3.3 La validation atomique dans les systèmes asynchrones

Un système réparti asynchrone est caractérisé par l'absence de borne supérieure sur le temps d'acheminement des messages. Dans ce contexte, il est impossible de différencier un site qui est en panne d'un site avec lequel les communications sont extrêmement lentes.De ce fait, la détection de pannes dans un système asynchrone ne peut être qu'approximative (i.e., non-fiable).
La détection de pannes est un point crucial pour la résolution de plusieurs problèmes d'accord tolérants aux pannes comme le problème de la validation atomique non-bloquante. En effet, les protocoles présentés jusqu'ici ne sont corrects que dans l'hypothèse où la détection de pannes est fiable.

Dans un système asynchrone pure, Fisher, Lynch et Paterson ont démontré dans [56] qu'il est impossible de résoudre d'une façon déterministe un problème d'accord tolérant aux pannes, du fait de la non-fiabilité des détections de pannes. Ce résultat est communément appelé résultat d'**impossibilité FLP**. Intuitivement, ceci est dû à l'impossibilité de distinguer de manière sûre un processus défaillant d'un processus lent ou d'un processus avec lequel les communications sont lentes. Le résultat d'impossibilité a motivé de nombreux travaux de recherche.
Chandra et Toueg ont proposé dans [57] une approche pour contourner ce problème. Cette approche consiste à augmenter le système asynchrone d'un mécanisme de détection de pannes qui peut "faire des erreurs". En particulier, ils ont introduit le concept de détecteurs de défaillances non-fiables[50].

3.3.1 Les détecteurs de pannes

Le rôle d'un détecteur de défaillance est de maintenir une liste des processus corrects (c'est-à-dire, non défaillants) et des processus suspectés d'être défaillants.
Dans [57], Chandra et Toueg ont proposé des détecteurs de pannes distribués : chaque site a accès à un détecteur de pannes local qui lui fournit la liste des sites qu'il soupçonne d'être en panne. Chaque détecteur peut faire des erreurs en ajoutant, à tort, des sites à sa liste de suspects (i.e., un détecteur peut soupçonner un site S même si ce dernier n'est pas en panne). Si le détecteur s'aperçoit ensuite qu'il a fait une erreur en soupçonnant S, il enlève S de sa

liste de suspects. Les détecteurs de pannes sur deux sites différents peuvent avoir des listes de suspects différentes.
Il est important de noter que ces détecteurs de pannes sont définis en termes de propriétés abstraites. Cela veut dire qu'aucune hypothèse n'est faite sur la façon dont ces détecteurs sont implémentés. Bien sûr, à cause de l'asynchronisme du système, il est impossible d'écrire des détecteurs parfaits. Mais un ensemble minimal de propriétés doit être assuré, à savoir que (i) tout processus défaillant finit par être suspecté *(propriété de complétude)* (ii) il existe un instant à partir duquel un processus correct ne sera suspecté par aucun processus (*propriété d'exactitude*) Deux propriétés de complétude et quatre propriétés d'exactitude ont été définies :

- Complétude forte (faible) : chaque site incorrect finit par être soupçonné de façon permanente par tout (au moins un) site correct.
- Exactitude forte (faible) : tout (au moins un) site correct n'est jamais soupçonné.
- Exactitude finalement forte (faible) : il existe un temps après lequel la propriété d'exactitude forte (faible) sera satisfaite.

Huit classes de détecteurs de pannes peuvent être définies par combinaison des propriétés de complétude et d'exactitude. On considère généralement une seule notion de complétude (forte) et quatre notions d'exactitudes ce qui permet de définir quatre types de détecteurs de défaillances :

1. Un détecteur parfait, de type P, vérifie la complétude forte et l'exactitude forte. Avec un détecteur parfait, chaque processus correct finit par suspecter tous les processus fautifs (complétude forte), au bout d'un temps fini, sans jamais suspecter un processus correct (exactitude forte). Ainsi, l'ensemble des processus suspectés ne peut qu'augmenter.
2. Un détecteur fort, de type S, vérifie la complétude forte et l'exactitude faible. Seul un processus correct est sûr de ne jamais être soupçonné à tort. Cet agent pourra servir de leader.
3. Un détecteur finalement parfait, de type $\Diamond P$, vérifie la complétude forte et l'exactitude finalement forte.
4. Un détecteur finalement fort, de type $\Diamond S$ vérifie la complétude forte et l'exactitude finalement faible.

3.3.2 L'élection de leader (détecteur de meneur)

Le détecteur de meneur est utilisé pour attribuer des privilèges à un processus pour qu'il agisse en tant que leader. On définit un détecteur de meneur comme un objet partagé DM tel que chaque processus possède une primitive atomique $DM.MENEUR_i$ lui indiquant une estimation du meneur actuel. Il se peut donc que deux processus aient deux estimations différentes du meneur actuel. On peut ainsi classifier les détecteurs de meneur en fonction de la rapidité avec laquelle il converge vers un meneur unique ou la rapidité avec laquelle l'oracle répond (un détecteur de meneur peut masquer un protocole classique d'élection d'un leader et donc prendre un certain temps avant de répondre). On dira qu'un détecteur de meneur est de type ω s'il vérifie la propriété suivante :
"finalement, un unique processus correct p est considéré comme le leader par tous les processus corrects."
On peut se référer à [58] pour une implémentation efficace de ce type de détecteurs.

Supposer l'existence d'un détecteur de meneur ω revient à faire une hypothèse sur la synchronisation du système. Dans [57], les auteurs démontrent qu'à partir de tout détecteur de défaillances permettant de résoudre le problème du consensus (en particulier à partir d'un détecteur de classe P, S, $\Diamond$ P ou $\Diamond$ S), il est possible de construire un détecteur ω. Ils proposent pour cela un algorithme permettant d'extraire un meneur par l'analyse des traces du détecteur de défaillances. En particulier, un détecteur de type $\Diamond$ S permet de définir un détecteur de meneur de type ω. Les détecteurs de types ω sont équivalents aux détecteurs de types $\Diamond$S. La différence entre ces deux détecteurs est qu'un détecteur ω renvoie un seul processus correct en qui tous les processus peuvent faire confiance, alors que le détecteur $\Diamond$S renvoie une liste de processus suspects [26].

3.3.3 Les protocoles de validation atomique non bloquante dans les systèmes asynchrones

La plupart des travaux présentés dans la littérature, proposant des protocoles de validation atomique non bloquante, considèrent un système asynchrone avec une architecture trois tiers.
L'architecture trois tiers est un système réparti avec un ensemble fini de processus qui communiquent par échange de messages. Trois types de processus sont définis : le client (noté c_i), le serveur d'application (Application server), noté a_i et les serveurs de base de données (Data Base Server), noté S_i). Chaque processus peut tomber en panne aléatoirement. Les pannes malicieuses ne sont pas considérées.
Chaque requête lancée par le client est une transaction répartie qui implique tous les serveurs de bases de données (Chaque serveur de base de données correspond à une base de données). Cependant, elle invoque un seul serveur d'application à la fois. Les requêtes n'ont pas d'effets sur les serveurs d'application mais uniquement sur les bases de données. De plus les serveurs d'application sont sans état (c'est à dire, aucune information concernant la requête n'est gardée au niveau du serveur d'application).
Les protocoles de validation atomique présentés jusqu'ici assurent au plus une validation de la transaction. Si une panne se produit, au niveau des serveurs d'application ou des bases de données, ou lorsque le délai d'attente s'est écoulé, le client reçoit un message d'exception sans préciser si la requête est validée ou pas. Le client, en fait, ignore l'origine de l'exception. Dans la pratique, il relance la requête en prenant le risque de valider la transaction plus d'une fois.
Frolund et Guerraoui ont défini dans [59] une abstraction appelée *e-transction* (exactly once transaction) pour assurer exactement une seule exécution de la transaction même en présence de pannes. Cette abstraction ajoute au modèle transactionnel la propriété de vivacité (liveness) qui implique aussi le coté client.
L'abstraction *e-transaction* est définie par sept propriétés appartenant à trois catégories *Terminaison, Accord et Validité*. La Terminaison prévient les situations de blocage, l'accord assure la cohérence des bases de données et du client, la validité limite les éventuelles valeurs du résultat en excluant celles qui n'ont pas de sens.

- La terminaison.
  - (T.1) Si le client lance une requête, alors il délivre le résultat.
  - (T.2) Si une base de données vote pour un résultat, alors la base de données valide

ou annule ce résultat.

- L'accord.
  - (A.1) Un résultat n'est délivré par le client que si ce résultat est validé par toutes les bases de données
  - (A.2) Aucune base de données ne décide plus d'un résultat pour la même requête
  - (A.3) Deux bases de données ne décident pas différemment pour le même résultat
- La validité.
  - (V.1) Si un client lance une requête et délivre un résultat, alors le résultat est calculé par un serveur d'application
  - (V.2) Une base de données ne valide un résultat que si tous les serveurs de base de données ont voté oui.

L'implémentation de cette abstraction assure la cohérence des bases de données qui se mettent d'accord quant à la validation de chaque résultat calculé, et ce, même en présence de pannes (validation atomique non bloquante). De plus, elle assure la cohérence du coté client par le fait que le résultat validé est toujours retourné au client.

Nous présentons ci-après les plus importantes contributions assurant la validation atomique dans les systèmes asynchrones (généralement les architectures trois tiers) qui implémentent l'abstraction *e-transaction*.

3.3.3.1 La réplication asynchrone

L'idée maîtresse du protocole proposé dans [60] consiste en l'utilisation de la réplication asynchrone du serveur d'application. Un serveur primaire exécute la requête du client dans le cadre d'une transaction et lui retourne le résultat. Le client essaie de récupérer le résultat à travers le serveur primaire. Si le primaire ne répond pas, le client envoie la requête à l'ensemble des serveurs secondaires (copies). Si un serveur secondaire suspecte le primaire il devient lui même primaire et tente de terminer ce que le serveur primaire précédent a commencé en essayant d'annuler la transaction. Sachant que les détecteurs de fautes ne sont pas fiables, plusieurs serveurs d'application peuvent être primaires en même temps et agissent en concurrence pour valider ou annuler la transaction. Afin de contourner ce problème, les auteurs définissent deux registres, le premier noté $RegA$ destiné à sauvegarder l'identité du serveur d'application qui exécute la transaction en cours, le deuxième noté $RegD$ permet de garder la décision de la transaction (Commit/Abort). Les deux registres sont partagés par tous les serveurs d'application. Néanmoins, une seule écriture est tolérée (write once register noté wo-register). Si plusieurs processus tentent d'écrire dans le registre, seulement une seule valeur est écrite et aucune autre valeur ne peut être insérée.

Lorsque le serveur primaire commence une transaction, il écrit son identifiant dans le registre $RegA$. Quand il décide de la transaction, il écrit la valeur de décision (Commit/Abort) dans le registre $RegD$.
Dans le cas d'une panne le nouveau serveur primaire procède à l'annulation de la transaction en essayant d'écrire la valeur *Abort* dans le registre $RegD$. S'il arrive à le faire (c'est à dire, le serveur primaire suspecté n'a pas écrit la décision avant de tomber en panne),

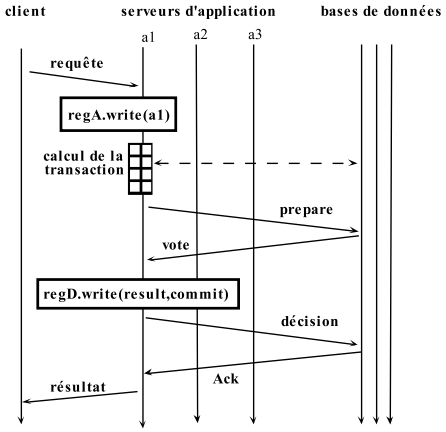

FIGURE 3.3 – Réplication asynchrone

il envoie la décision aux serveurs de bases de données et retourne la décision au client. Cependant, s'il n'arrive pas écrire dans le registre *RegD* (c'est à dire, le serveur en panne a déjà écrit la décision) alors il récupère la valeur écrite et en informe le client.
Il est facile de remarquer l'inconvénient que présente ce protocole. En cas de suspicion de panne, le nouveau serveur d'application force l'annulation de la transaction si elle n'est pas encore décidée, au lieu de la reprendre, même si cette dernière a été exécutée correctement.

3.3.3.2 La réplication primaire secondaire

La solution présentée dans [61] utilise une variante de l'algorithme de la copie primaire (primary backup). Cet algorithme assure qu'au moins un serveur d'application est disponible pour exécuter la transaction. Si la copie primaire tombe en panne, une copie secondaire devient primaire et continue l'exécution de la requête. Le client retransmet la requête à l'ensemble des serveurs d'application. Le serveur d'application primaire calcule le résultat et se coordonne avec les serveurs de bases de données pour valider ou annuler le résultat obtenu. Le protocole utilisé est vu comme un protocole 2PC avec réplication du coordinateur (figure 3.4).
Le serveur d'application primaire sauvegarde les informations nécessaires (le résultat, les votes collectés) au niveau des serveurs d'application secondaires (backups) sans l'utilisation d'une unité de stockage stable. Si le primaire tombe en panne, une des copies secondaires reprend l'exécution à partir d'un état cohérent. Les auteurs supposent qu'il existe un groupe statique de copies secondaires et que chaque serveur d'application possède un accès local à un détecteur de défaillance parfait.
Le problème majeur du présent protocole est le fait de considérer la détection de panne fiable. Ceci n'est pas toujours assuré, notamment dans un environnement ouvert.

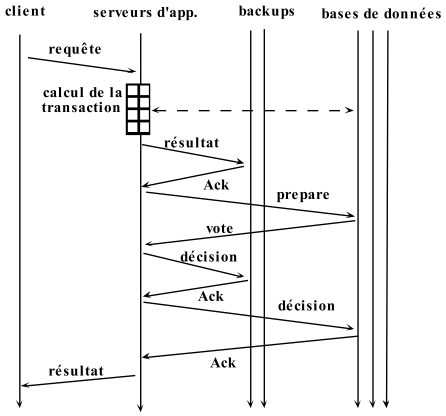

FIGURE 3.4 – Réplication primaire secondaire

3.3.3.3 La réplication asynchrone non coordonnée

La contribution définie dans [62] est basée sur l'idée de stocker les informations concernant la transaction distribuée au niveau des serveurs de bases de données où chacun d'eux maintient une table appelée ITP (Information Transaction Processing). Quand il reçoit la requête pour la première fois, le serveur d'application exécute la transaction et obtient un résultat. Il contacte ensuite les serveurs de bases de données pour valider ou annuler ce résultat, et ce, en suivant le protocole 2PC. Une fois la décision prise, il envoie le résultat au client (voir figure 3.5).
Dans le cas où le serveur d'application reçoit un message de terminaison de la part du client ; Il essaie de terminer la transaction initiée par un autre serveur d'application suspecté. Le serveur d'application récupère l'état de la transaction enregistrée dans la table ITP au niveau des serveurs de bases de données. Il leur envoie un message 'resolve' qui essaye d'insérer dans la table ITP l'identificateur de la transaction avec un état 'Abort'. Sachant que l'identifiant de la transaction est unique, les serveurs de bases de données insèrent cette nouvelle entrée dans la table ITP seulement si la transaction n'est pas encore enregistrée. Sinon chaque serveur de base de données retourne l'état correspondant à cet identifiant dans la table ITP au serveur d'application. Ce dernier réagit en fonction des réponses des serveurs de bases de données (valider si toutes les réponses sont 'commit' ou 'prepared', annuler si les réponses sont abort).
Le problème dans cette solution réside dans le fait que le serveur d'application tente d'annuler la transaction si elle n'est pas préparée ou décidée. Le second problème concerne la détection de fautes du serveur d'application qui se fait uniquement par le client. Si ce dernier tombe en panne en même temps que le serveur d'application actif l'exécution de la transaction est bloquée.

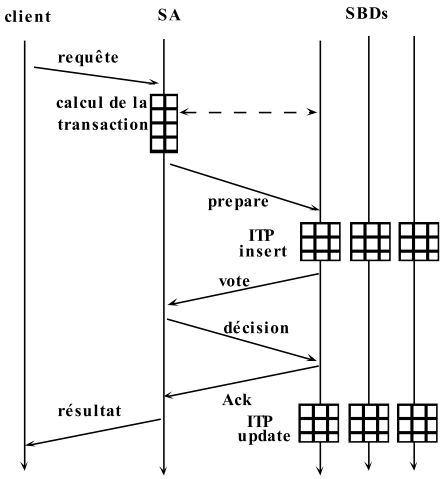

FIGURE 3.5 – Le protocole basé sur les tables ITP

3.3.3.4 La pré-validation multi-instances

Il existe de fortes similitudes entre la solution précédente et celle présentée dans [63] où les informations concernant la transaction sont gardées au niveau des serveurs de bases de données. Cependant, les auteurs essaient de ne pas forcer l'annulation de la transaction qui est déjà préparée (phase de préparation du protocole 2PC accomplie) en introduisant la notion de *pré-validation multi-instances*.
Le client choisit un serveur d'application et lui envoie la requête avec un identificateur unique. Il associe à cette requête la valeur d'un compteur d'instance (incrémenté à chaque renvoi), appelé identifiant de l'instance de la transaction. Si après un délai d'attente limité le client ne reçoit pas de réponse, il suspecte le serveur d'application. Dans ce cas il renvoie la requête à un autre serveur d'application avec un identificateur d'instance différent (il incrémente le numéro d'instance).
Lorsque le serveur d'application reçoit la requête, il calcule la transaction en interagissant avec les bases de données puis entame la phase de préparation du protocole de validation atomique en envoyant un message de vote aux serveurs de bases de données. Si le serveur d'application ne reçoit pas tous les votes au bout d'un temps limité, il retransmet le message de vote et ainsi de suite jusqu'à l'obtention de toutes les réponses.
Chaque serveur de base de données maintient une table de pré-validation (*Muti-Instance Pre-Commit Table notée MIPT*) pour garder les instances de transaction de la même requête, on les appelle transactions sœurs (sibling transaction).
Chacun des serveurs de base de données répond au message de vote en créant une table $MIPT_{Req_{id}}$ pour la requête en cours (si elle n'existe pas), puis insère une nouvelle entrée contenant l'identifiant de la requête et celui de l'instance de la transaction. Une fois la transaction préparée, le serveur de base de données envoie sa réponse au serveur d'application tout en joignant la table $MIPT_{Req_{id}}$ à sa réponse. (voir figure 3.6).
En recevant les votes munis des tables $MIPT$, le serveur d'application prend connaissance de toutes les transactions sœurs (instances d'une même transaction) déjà préparées au niveau des serveurs d'application. De ce fait, il valide (*commit*) l'instance préparée sur

toutes les bases de données ayant le plus petit identifiant, les autres instances sont annulées. Si par contre, le serveur d'application ne trouve aucune instance préparée sur l'ensemble des serveurs de bases de données il annule uniquement l'instance courante.
Il est supposé que les serveurs de bases de données se rétablissent après chaque panne. Dans ce cas, après recouvrement, le serveur de base de donnés vérifie l'existence de transactions préparées et non encore validées. Il crée une nouvelle requête qu'il envoie au serveur d'application en gardant le même identifiant mais en incrémentant le numéro d'instance. En fait, le serveur de base de données relance la requête afin d'éviter de maintenir le blocage des instances préparées.
Le fait que le serveur d'application doit attendre la réponse de tous les tiers finaux même ceux qui sont défaillants, risque de bloquer des ressources pendant longtemps pour une transaction qui peut être annulée à la fin et donc priver d'autres transactions de s'exécuter.

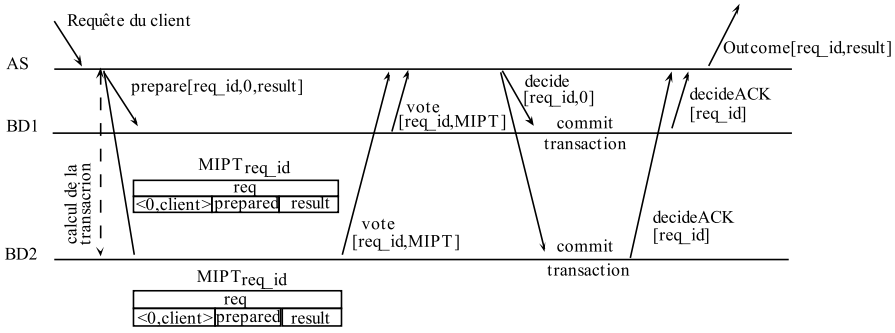

FIGURE 3.6 – Le protocole basé sur la pré-validation multi-instances

Les travaux cités précédemment partagent tous un point commun. Ils se basent tous sur le protocole 2PC pour la validation de la transaction sauf que chacun d'entre eux essaie de contourner le blocage en cas de panne du coordinateur. La majorité des travaux font appel à la réplication du serveur d'application et proposent des solutions différentes pour éviter les validations multiples d'une même transaction tout en assurant la propriété *e-transaction*.

3.4 Le problème de consensus

Le problème du consensus est introduit car il intervient dans la résolution du problème de la validation atomique non-bloquante dans un système asynchrone.
Le problème du consensus est peut-être le problème majeur en algorithmique répartie et possède un intérêt certain. Il peut être rapidement défini, de manière informelle, de la façon suivante : "Soit un ensemble de processus, chacun proposant une valeur, trouver un protocole réparti afin de mettre tous les processus d'accord sur une des valeurs proposées initialement".
En début de consensus, chaque processus possède une connaissance locale ou un avis initial. Le but est de trouver un avis unique qui représentera l'avis du groupe. Au cours d'une instance de consensus, des valeurs initiales (potentiellement distinctes) peuvent être

proposées par un ou plusieurs processus. Les participants doivent collaborer pour assurer que celui-ci converge inéluctablement (terminaison) vers une valeur de décision unique (accord uniforme) qui doit nécessairement être l'une des valeurs proposées (validité). La valeur de décision est alors retournée vers tous les auteurs qui ont fourni (ou fourniront) une valeur initiale concernant cette instance de consensus particulière.
Le problème du consensus suscite beaucoup d'intérêts. Sur un plan pratique, une solution à ce problème peut être une brique de base pour résoudre d'autres problèmes d'accord (diffusion atomique, tolérance aux défaillances fondée sur la réplication, ...).
Formellement, le problème du consensus est défini par deux primitives, appelées *propose* et *decide*. Le protocole de consensus doit satisfaire les trois propriétés suivantes[64] :

- Terminaison : Tout processus correct finit par décider une valeur.
- Accord uniforme : Deux processus (corrects ou pas) ne doivent pas décider deux valeurs différentes.
- Validité : La valeur décidée doit être une des valeurs proposées.

La propriété d'accord uniforme est importante si l'on considère que les participants incorrects peuvent reprendre leur exécution en lançant un protocole de reprise après panne. Cette propriété leur garantit alors d'être toujours en accord avec les autres participants sur la valeur décidée.
L'intérêt d'un tel protocole vient du fait que, si nous étions capables de résoudre le problème du consensus, nous serions alors capables d'implémenter des services tolérants aux fautes. En pratique pour mettre en place un système tolérant aux fautes, il suffit de répartir le calcul sur plusieurs ordinateurs. Ainsi, si l'un d'entre eux vient à tomber en panne, le système continuera à fonctionner de manière transparente, les autres ordinateurs continuant à fournir le service[65].
La principale difficulté d'un tel schéma, est qu'il est nécessaire de s'assurer de la cohérence du service, et pour cela tous les ordinateurs doivent effectuer les différents calculs dans le même ordre. Il a été démontré qu'un tel système distribué était implémentable si et seulement si le problème du consensus possède une solution. Malheureusement, dans le cas général des systèmes asynchrones, cela a été prouvé impossible [56] si, ne serait-ce qu'un seul processus tombe en panne.
[57] montre que le problème de consensus, une forme abstraite de problèmes d'accord, peut être résolu dans un système asynchrone augmenté d'un détecteur de pannes non-fiable. En fait, plusieurs protocoles ont été proposés selon la classe de détecteurs utilisée.

3.4.1 L'algorithme de Chandra et Toueg

L'algorithme de Chandra et Toueg se base sur le paradigme du coordinateur tournant, et la majorité du quorum.
Un tour est une tentative pour converger vers une valeur de décision. Chaque tour est identifié par un numéro et est géré par un coordinateur unique. Le numéro de tour doit vérifier les propriétés suivantes :

- un numéro de tour est unique
- le nombre de tour requis pour résoudre une instance de consensus n'est pas limité
- une relation d'ordre total est définie entre les numéros de tours

Au tour (r), le coordinateur (c) sera le processus avec le numéro $((rmodn) + 1)$.
Un tour est découpé en quatre phases (voir figure 3.7). Dans la première phase, chaque processus envoie son estimation courante au coordinateur, en y joignant le numéro de tour durant laquelle cette estimation a été mise à jour pour la dernière fois. Dans la deuxième phase, le coordinateur attend une majorité de ces estimations, sélectionne la plus récente et l'envoie à tous les processus. Dans la troisième phase, quand p_i reçoit la valeur du coordinateur, il lui envoie un acquittement (ACK) pour indiquer au coordinateur qu'il adopte la nouvelle estimation. Si le détecteur de fautes de p_i suspecte le coordinateur avant de recevoir sa valeur, p_i enverra un acquittement négatif (NACK) au coordinateur et passera au tour suivant. Finalement, lors de la quatrième phase, le coordinateur attend une majorité d'ACKs ou un NACK. S'il reçoit une majorité d'ACKs, il décide l'estimation courante et effectue une diffusion atomique (Reliable Broadcast) de cette décision. Quand les autres processus reçoivent cette décision, ils décident immédiatement. Par contre, si le coordinateur reçoit un NACK, il passe au tour suivant sans décider et sans envoyer de messages[**?**].

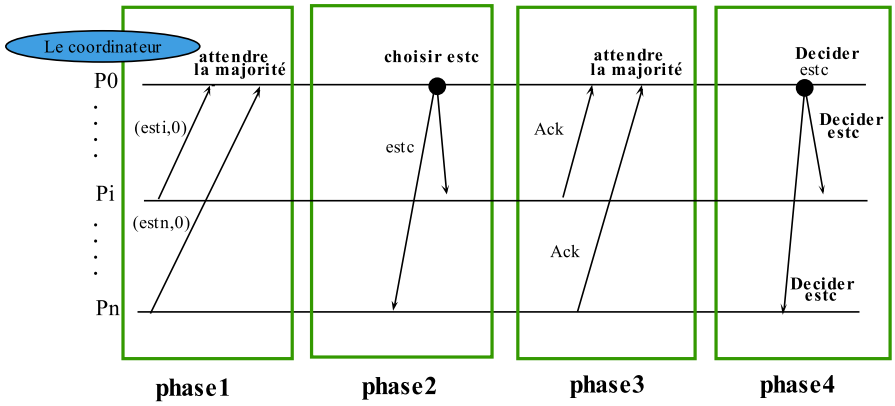

FIGURE 3.7 – Le protocole de Chandra et Toueg

3.4.2 Le protocole Paxos

Le protocole Paxos a été initialement présenté par Lamport dans [66] puis reformulé de manière plus conventionnelle dans [67]. L'intérêt de cet algorithme est qu'il fait très peu d'hypothèses (communications éventuellement non fiables). Le Paxos adopte une approche optimiste : c'est un algorithme 'best-effort'. Cela signifie qu'il garantit toujours la cohérence du système, c'est à dire, la condition d'accord, mais ne garantit la terminaison du protocole que dans le cas où le système se comporte suffisamment bien et sur une période suffisante.
Leslie Lamport a proposé dans [67] une définition originale du problème du consensus. Au lieu de considérer tous les processus comme jouant des rôles symétriques, il définit quatre types de participants :

1. Les auteurs (*Proposer* noté P_i)
2. Les apprenants (*Learner* noté L_i)
3. Les coordinateurs (*Coordintaor* noté C_i)
4. Les accepteurs (*Acceptor* noté A_i)

Les auteurs, sont des entités externes qui proposent les valeurs initiales. Au moins un auteur correct est supposé proposer une valeur durant chaque instance de consensus.
Les apprenants sont chargés de connaître la valeur décidée par le consensus. Les auteurs et les apprenants ne sont pas impliqués dans la procédure de prise de décision exécutée par le consensus.
Le coordinateur est chargé de choisir une valeur et de l'imposer comme valeur de décision. Le coordinateur est activé quand il est choisi comme étant le meneur de cette instance de consensus. Paxos est basé sur un service d'élection de meneur ou de leader fourni par une oracle de leader. L'oracle de leader délivre un processus unique correct, ce processus est appelé le leader courant.
Les accepteurs sont des entités passives qui peuvent soit accepter ou refuser la valeur proposée par le leader. Les accepteurs sont utilisés pour implémenter le quorum. Le quorum dans paxos est défini par la majorité des accepteurs. La majorité des accepteurs doivent être corrects durant le calcul du consensus courant.
Chaque processus peut prendre un ou plusieurs rôles. Les nœuds corrects sont supposés être majoritaires au sein du groupe de n acteurs : si f est le nombre maximum de pannes pouvant affecter le groupe de n nœuds alors $f < n/2$. Une majorité d'accepteurs doit être correcte : le nombre d'accepteurs doit donc être supérieur ou égal à $2f + 1$. Au moins un coordinateur doit être correct : le rôle de coordinateur est joué par au minimum $f + 1$ acteurs.
Lorsque la qualité des informations fournies par l'oracle est parfaite, un seul coordinateur agit en tant que leader et coordonne la prise de décision. Ainsi, la progression du calcul durant une instance de consensus est rythmée par la notion de tour (round) noté r.
Lorsque le coordinateur agit en tant que leader, il doit définir son numéro de tour. Pour garantir que deux leaders ne choisissent pas un même numéro de tour, chaque coordinateur C_i choisit un numéro de tour de type $i, i + n, i + 2n, ...$ dans cet ordre. Dans ce cas chaque numéro de tour r est lié à un coordinateur dont l'identité est r mod n.
Un tour se compose de deux phases (voir figure 3.8) : dans la première (phase de préparation) le leader vérifie si les autres processus ont déjà décidé une valeur avec un numéro de tour supérieur au sien ; dans la seconde (phase de proposition) il essaie de décider une valeur.

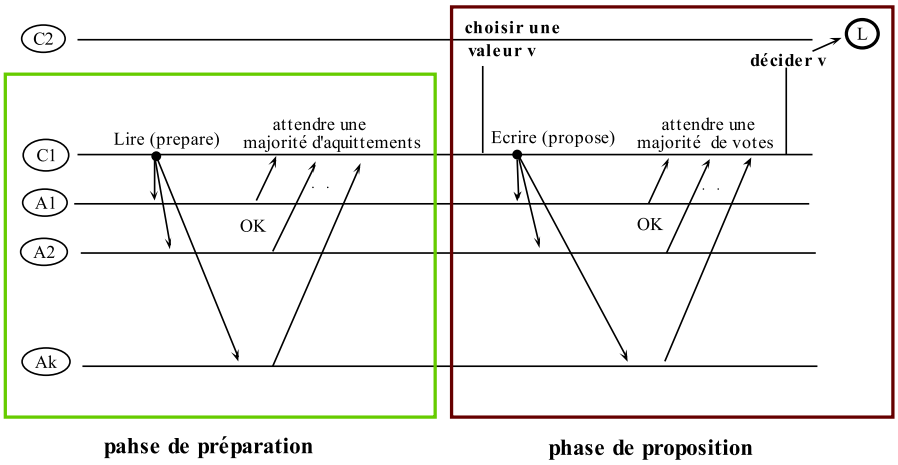

FIGURE 3.8 – Le protocole Paxos

La phase de préparation. Au cours de la première phase, le leader s'assure que la valeur qu'il soumettra lors de la seconde phase n'est pas incompatible avec celles éventuellement soumises par d'autres coordinateurs ayant agi comme leader au cours du même consensus mais durant des tours précédents. Le leader envoie une requête de lecture (prepare) avec son numéro de tour à l'ensemble des accepteurs. Lorsqu'un accepteur reçoit la requête, il accepte la proposition si le numéro de tour de la proposition acceptée jusqu'à présent est inférieur au numéro de tour courant. S'il s'est déjà engagé dans un tour ayant un numéro plus grand, il répond négativement (nack). Sinon, il répond positivement (ack) et envoie la dernière valeur qu'il a déjà acceptée (ou $\perp$ s'il n'a déjà rien accepté), ainsi que le numéro du tour dans lequel il a accepté la valeur. Il s'oblige alors à ne pas accepter de s'engager dans un tour ayant un numéro inférieur. Le leader attend la réponse d'une majorité d'accepteurs. Si l'un des accepteurs a déjà agréé une valeur, le leader sélectionne la valeur ayant le numéro de tour le plus grand, sinon, il choisit une valeur parmi celles proposées par les auteurs.
Si tout se passe bien, le leader peut lancer une seconde phase dans laquelle il demande aux accepteurs d'agréer la valeur qu'il propose. Cependant, il ne peut pas proposer n'importe quoi. Si certains accepteurs lui ont envoyé une valeur lors de la phase de préparation, cela signifie qu'un autre leader a déjà tenté de leur faire accepter une valeur (mais est peut-être tombé en panne). Pour garantir la cohérence, il doit alors proposer la valeur ayant le numéro de tour le plus grand. Si aucun accepteur ne lui a envoyé de valeur, il peut alors choisir la valeur qu'il souhaite. Il lance alors la phase 2.

La phase de proposition. Elle débute une fois que le leader a identifié une valeur qu'il peut soumettre sans risquer de violer les propriétés de sûreté (Accord et Validité). Le leader diffuse une requête d'écriture (propose) à tous les accepteurs pour leur proposer cette valeur. Cette requête est munie du numéro de tour courant. Le leader attend de collecter une majorité de réponses favorables avant de pouvoir conclure que cette valeur soumise est la valeur de décision. Lorsqu'un accepteur reçoit cette valeur, il l'accepte sous réserve de ne s'être pas engagé dans un tour ayant un numéro plus grand. Sinon, il l'ignore.
Une fois qu'une majorité a répondu favorablement, le leader peut décider de la valeur adoptée par l'ensemble des accepteurs et diffuse cette valeur aux apprenants.

Une des métriques utilisées pour évaluer les protocoles de consensus est la latence. Elle est définie par le nombre d'étapes de communication requises depuis la proposition de la valeur jusqu'à la décision de cette valeur.
Le protocole Paxos requiert donc 4 étapes de communications entre les acteurs, auxquelles viennent s'ajouter 2 étapes "externes" correspondant à la diffusion des valeurs initiales des auteurs vers les coordinateurs et à la diffusion de la valeur de décision du leader vers les apprenants. Le chemin de communication correspond à :
$auteur \rightarrow leader \rightarrow accepteurs \rightarrow leader \rightarrow accepteurs \rightarrow leader \rightarrow apprenants$

La latence de ce protocole peut être diminuée si nous considérons que le coordinateur peut jouer le rôle de l'apprenant. D'autre part, lorsque le message de l'auteur est en transit vers le leader ce dernier peut commencer la première phase (lancer l'opération de lecture) en parallèle. Dans ce cas, la latence du protocole Paxos consiste en 4 étapes de communication. La description de paxos se focalise sur une seule instance de consensus. Ce protocole peut être utilisé comme une brique de base pour résoudre une séquence de plusieurs instances de consensus successives. Le Multipaxos consiste simplement à établir des consensus sur des séquences de valeurs plutôt que sur une valeur unique. Si le leader est fiable sur une période assez longue, il peut servir pour plusieurs consensus d'affiler. Donc, pour chaque session, il effectue uniquement la phase 2 du protocole. Dans ce qui suit nous présentons les optimisations pour améliorer la performance du protocole paxos lorsque les conditions sont favorables.

3.4.3 Les optimisations du protocole Paxos : Le Paxos rapide

Pour accroître les performances, deux optimisations ont été proposées, visant à améliorer la latence du protocole Paxos original en réduisant le nombre d'étapes de communication. Ces optimisations sont appelées respectivement SO et RO. Ces notations font référence au fait que la première optimisation, SO, est Sûre alors que la seconde, RO, est Risquée.

3.4.3.1 L'optimisation sûre

L'optimisation proposée dans [68] appelée FastPaxos, notée *SO* est valable lorsque les conditions sont favorables pour exécuter une séquence de consensus. Cette optimisation peut être utilisée dans un contexte où le leader reste stable sur une longue durée sans l'apparition de fautes. Quand le coordinateur est le leader ultime pour plusieurs instances de consensus successives, il n'a pas besoin d'exécuter la phase de lecture (prepare phase) pour chaque instance de consensus. La première phase est exécutée uniquement lorsque le coordinateur devient leader afin d'assurer la cohérence avec les actions faites par les leaders précédents. A cet effet, il n'utilisera que la phase 2 tant qu'aucun autre leader ne viendra convoiter sa place. Par conséquent, les instances de consensus s'exécutent avec une même phase de préparation (figure3.9).

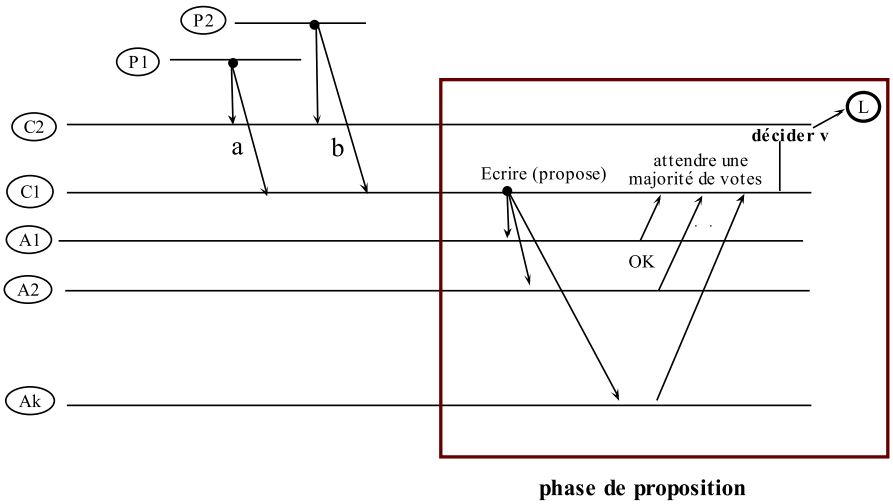

FIGURE 3.9 – L'optimisation du protocole Paxos : optimisation sûre

Cette optimisation consiste donc à supprimer une partie du calcul (la phase de préparation) lorsque celle-ci est inutile. Si le leader courant N_i a réussi à imposer une valeur de décision concernant le consensus c durant la tentative r et si, du point de vue de N_i, aucun autre coordinateur ne semble avoir agi avec un numéro de tour supérieur à r ni durant l'instance de consensus $c-1$ ni durant l'instance c alors N_i peut agir en temps que leader durant le consensus k en utilisant le même numéro de tour r. De ce fait, un tour r comporte alors une seule phase de préparation qui est suivie par autant de phase de proposition que N_i peut en lancer avant d'être destitué.
Si un nouvel auteur croit être le leader, il commence par une phase 1 et peut ainsi mettre fin au règne de l'ancien leader. Puis il n'utilisera que la phase 2 tant qu'aucun autre leader ne viendra convoiter sa place.
En conséquence, plusieurs instances de consensus peuvent être exécutées durant le même tour. Quand le leader élu reste stable (pas de défaillance, interactions avec les autres acteurs suffisamment synchrones), le chemin de communication est de longueur 4 et correspond à : *auteur* $\rightarrow$ *leader* $\rightarrow$ *accepteurs* $\rightarrow$ *leader* $\rightarrow$ *apprenant*.

3.4.3.2 L'optimisation risquée

Lamport a proposé un algorithme de Paxos rapide (Fast Paxos) [69], notée *RO*, dans lequel les auteurs s'adressent directement aux accepteurs sans passer par un leader. L'objectif est de tirer profit du fait que, durant la plupart des instances de consensus, un seul auteur participe au consensus et donc une seule valeur initiale est disponible. Lorsque la durée qui sépare deux instances de consensus successives est très grande, au lieu de rester inactif le leader peut anticiper une partie du calcul pour la prochaine instance de consensus. Le leader ne possédant aucune valeur initiale, il soumet une valeur par défaut appelée *ANY* lors de la phase de proposition aux accepteurs.
Tout accepteur recevant cette valeur fictive est alors autorisé à la remplacer par une valeur réelle directement reçue d'un auteur qui envoie la valeur initiale (proposition) aux coordi-

nateurs et aux accepteurs. Le chemin de communication est de longueur 3 :
auteur $\rightarrow$ *accepteurs* $\rightarrow$ *leader* $\rightarrow$ *apprenant*.

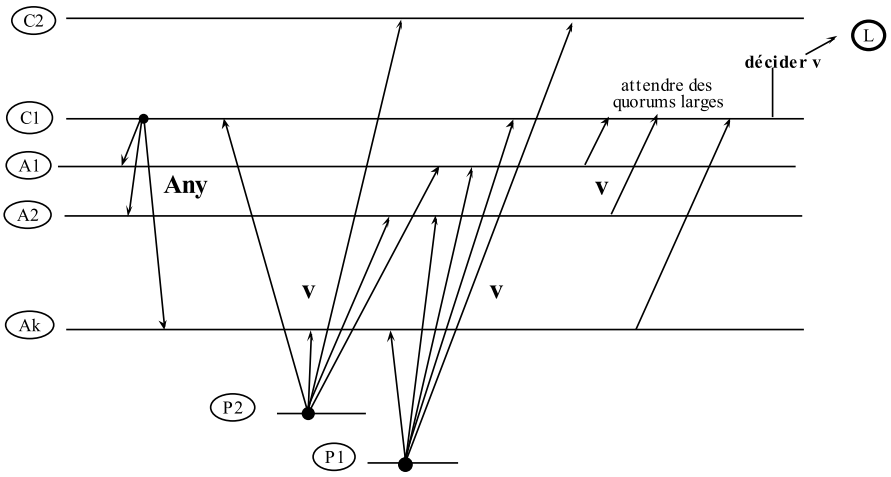

FIGURE 3.10 – L'optimisation du protocole Paxos : optimisation risquée

Dans la version paxos classique, une seule valeur initiale est soumise durant une phase de proposition, et donc tous les accepteurs qui acceptent une valeur durant cette phase adoptent nécessairement la même valeur. Cette propriété n'est plus toujours vérifiée dès lors que deux auteurs fournissent deux valeurs initiales distinctes durant l'instance *c*. Ceci a une conséquence majeure. L'optimisation est un échec lorsque les retours collectés par le leader font référence à plus d'une valeur (occurrence d'une *collision*). Dans ce cadre, le leader sert à détecter les collisions éventuelles entre les propositions simultanées de plusieurs auteurs. Lorsque cela se produit, un recouvrement est alors nécessaire : un nouveau tour classique de paxos est démarré par le leader (via l'exécution d'une phase de préparation) pour conclure le consensus. L'optimisation *RO* peut parfois avoir un impact négatif : elle présente un intérêt uniquement lorsque la probabilité de collision est faible.

3.4.4 Fast Paxos adapté pour les consensus multiples intégrés (MIC Paxos)

Dans le cas d'un mécanisme de réplication destiné à assurer la tolérance aux défaillances, une cohérence forte doit être maintenue entre les différentes copies d'un serveur critique. Beaucoup de services essentiels, tels que la diffusion atomique ou la validation atomique peuvent se réduire en partie à un problème d'accord entre composants. L'approche "State Machine" en est une illustration. Dans cette approche, les réplicats d'un serveur critique doivent s'entendre pour définir une séquence unique de requêtes entrantes. La construction de cette séquence se fait généralement en appelant de façon répétée un service de consensus. Dans [70, 71], les auteurs proposent un protocole adaptatif qui gère une séquence d'instances de consensus et garantit la persistance de toutes les décisions prises. Le protocole proposé s'appelle Paxos-MIC. Comme son nom l'indique, il s'appuie sur les principes généraux du protocole Paxos. Pour accroître les performances, les deux optimisations déjà présentées qui visent à améliorer la latence du protocole Paxos original sont intégrées.

L'optimisation RO est compatible avec l'optimisation SO. Alors que l'optimisation SO concerne la première phase du protocole Paxos (la phase de préparation), l'optimisation RO se focalise sur la deuxième et dernière phase du protocole (la phase de proposition). L'optimisation RO consiste, d'une part, à anticiper une partie du calcul (c'est à dire, à exécuter entièrement la phase de préparation et à entamer la phase de proposition avant même que des valeurs initiales soient proposées par des auteurs) et, d'autre part, à désigner les accepteurs comme étant les destinataires directs des valeurs initiales proposées par les auteurs (au lieu de faire transiter ces valeurs par le leader comme cela est le cas dans le protocole Paxos original). Dans les cas favorables, la combinaison des deux optimisations requièrent seulement 3 étapes de communication. Comme l'optimisation SO est sûre, Paxos-MIC intègre l'optimisation SO et permet au leader de décider (après évaluation d'un test local) s'il déclenche ou pas l'optimisation RO (pour éviter les collisions).

3.4.4.1 Description du protocole MIC-Paxos

MIC-Paxos (Multiple Integrated Consensus Paxos) est destiné à résoudre une séquence de consensus infinie, dans un système asynchrone augmenté avec une oracle d'élection de leader. Chaque instance de consensus est identifiée par un numéro c où $c \geq 1$. Le numéro de consensus est incrémenté de 1 après chaque décision. Il définit n processus, impliqués directement dans l'exécution du consensus, pouvant jouer deux rôles distincts : Coordinateur et Accepteur. Un participant peut jouer les deux rôles. Il suppose que la majorité des participants doivent être corrects, si f est le nombre maximum de pannes alors $(f < n/2)$.
MIC-Paxos définit également un ensemble de nœuds externes appelés Auteurs/Apprenants (Proposers/Learners) notés $PLext_k$. Ces nœuds vont jouer le rôle des auteurs et des apprenants (défini dans paxos). Pour chaque instance de consensus c, il existe au moins un nœud $PLext$ correct capable de fournir une valeur initiale (en appelant la fonction $propose(c, v)$) et de récupérer la valeur de décision finale notée $< v, c >$.
Un coordinateur n'est actif que lorsque le service d'élection de leader le désigne comme leader. En pensant alors agir comme un leader unique et incontesté, le coordinateur tente d'imposer une valeur de décision aux accepteurs. Les coordinateurs et les accepteurs interagissent en utilisant deux types de messages : $Operation$ (noté Op, envoyé par les coordinateurs) et $State$ (noté St, envoyé par les accepteurs). Contrairement à Pxos, MIC-Paxos ne fait pas la distinction entre les messages *prepare* et *propose*. Le message Op est interprété comme une lecture et une écriture en même temps[72].
A chaque coordintaeur est associé un module interne appelé $Proposer/Learner$ qui s'exécute sur la même machine (noté $PLint$). Le $PLint$ est chargé des interactions entre le coordinateur associé et les auteurs/apprenants externes ($PLext$). Le coordinateur communique uniquement avec le $PLint$ local, et ce, via les deux fonctions suivantes : $ProposePull$ et $DecidePush$ (voir figure 3.11).
Le service d'élection de leader est invoqué uniquement par les accepteurs et jamais par un coordinateur. Un accepteur exécute périodiquement la fonction $GetLeader$ pour connaître le leader courant et lui envoie un message d'état $State$. Ce message contient, entre autres, l'identité du leader choisi.
D'autre part, chaque coordinateur maintient une liste d'accepteurs qui le considère leader. Il agit en tant que tel, quand il a le support d'un quorum d'accepteurs (une majorité des

accepteurs l'ont choisi leader).

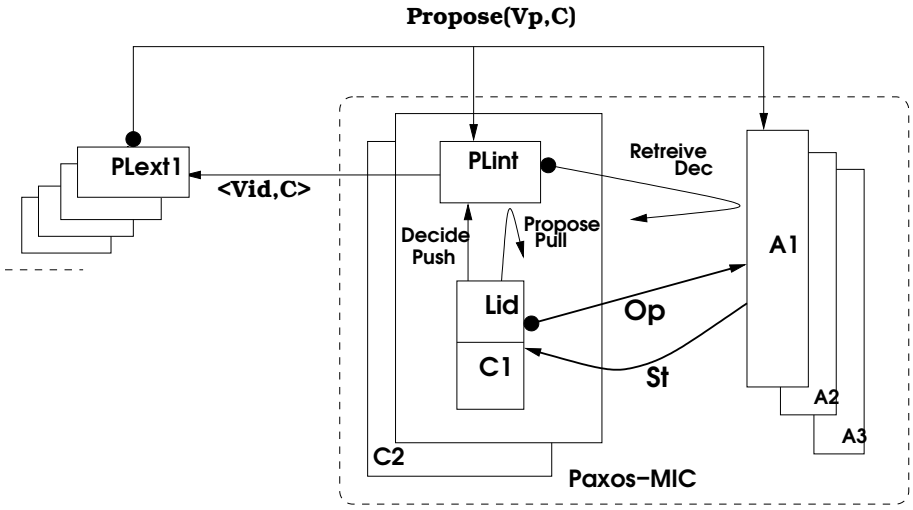

FIGURE 3.11 – Le protocole MIC Paxos

Le coordinateur peut exécuter une séquence de consensus durant le même tour (s'il est toujours leader). De même, une instance de consensus peut être exécutée durant plusieurs tours (si le leader est suspecté). Le leader commence l'exécution par une phase de préparation quand il devient leader. Cette phase lui permet de continuer l'exécution du consensus à partir d'un état cohérent. Il envoie un message *Operation* (équivalent à Prepare) aux accepteurs en leur communiquant son numéro de tour. Lorsqu'il reçoit une majorité de réponses, il passe à la phase de proposition. Le leader utilise les informations collectées durant la première phase. Dans le cas où au moins un accepteur retourne une valeur proposée par un leader précédent pour cette même instance de consensus, le leader courant doit choisir la valeur la plus récente entre les valeurs retournées (la valeur ayant le plus grand numéro de tour). Sinon le leader demande à son *PLint* local de lui fournir une valeur initiale en appelant la fonction *Proposepull*.
Le *PLint* du leader collecte les propositions faites par les auteurs externes concernant l'instance du consensus en cours. Quand il reçoit la demande de son leader (*ProposePull*), il retourne une valeur proposée. Cependant, si aucune valeur n'est disponible il retourne une valeur spéciale $\perp$.
Durant la seconde phase (phase de proposition), le leader essaye d'imposer une valeur aux accepteurs. Si la majorité est d'accord, cette valeur devient la décision. Les accepteurs adoptent cette valeur et envoient un message *State* au leader ainsi qu'à tous les coordinateurs. Le leader envoie à son tour la valeur de décision aux apprenants via le message *Decidepush*[71].
Quand le leader reste stable, il enchaîne sur l'instance de consensus suivante en incrémentant le numéro de l'instance c tout en gardant le même numéro de tour. Lorsque le leader dispose déjà d'une proposition concernant le prochain consensus c (en appelant la fonction *proposepull*), il effectue un consensus utilisant l'optimisation SO : il exécute la phase de proposition. Si au contraire, aucune proposition n'est disponible , le leader est

temporairement inactif et il évalue alors un test de déclenchement pour déterminer si RO doit être activée ou pas. Si le test est faux (ou plus généralement s'il ne devient pas vrai avant qu'une proposition ne parvienne au leader), le consensus c s'exécute en n'utilisant que SO : nous dirons dans ce cas que RO est non activée (par choix). Si le test est vrai (l'optimisation RO est activée), le leader exécute la phase de préparation en envoyant la valeur Any, appelée valeur directe, notée $\perp$, comme valeur de proposition aux accepteurs. Quand le leader reçoit les réponses du quorum, cela veut dire que la majorité des accepteurs est d'accord pour adopter la première valeur reçue délivrée directement par les auteurs externes $PLext$[71].
Différents auteurs externes peuvent fournir des valeurs distinctes. Ces valeurs peuvent arriver dans un ordre différent d'un accepteur à l'autre. Dans ce cas, les accepteurs risquent d'adopter des valeurs de décision différentes (apparition d'une collision).
Le leader prend connaissance qu'une collision a eu lieu quand il reçoit les messages $State$ des accepteurs. Ce message contient toujours la dernière instance de consensus décidée ainsi que la valeur de décision adoptée. Les valeurs adoptées sont différentes et le leader doit exécuter une procédure de recouvrement de collision. La manière la plus simple, mais coûteuse, serait d'abandonner le tour courant et passer au tour suivant ; donc forcer une nouvelle phase de préparation. Le leader étant toujours le même il peut choisir une valeur parmi celles proposées durant le tour précédent.

Les décisions sont toujours sauvegardées au niveau des accepteurs. Chacun d'entre eux maintient un tableau de log appelé $LogDVal$. Une entrée k dans ce registre contient la valeur de décision de l'instance de consensus numéro k. L'objectif d'un tel mécanisme est d'assurer que pour chaque instance de consensus, exécutée avec succès, il existe au moins un accepteur correct qui connaît la décision correspondante.
Le $PLint$ peut récupérer une ancienne décision, si cette dernière est demandée par une entité externe ($PLext$), en invoquant la fonction $RetrieveDec$. L'appel à cette fonction déclenche l'envoi périodique d'un message pour l'obtention de la décision. Ce message est envoyé à au moins une majorité d'accepteurs. L'accepteur qui en trouve la décision dans son registre log ($LogDVal$) la retourne au $PLint$[72].

Les optimisations SO et RO ne sont pas des contributions nouvelles. Par contre, le mécanisme d'activation de RO à la demande est nouveau. Dans Paxos-MIC, la décision de déclencher ou de ne pas déclencher l'optimisation RO pour l'instance de consensus c est prise de manière centralisée par le leader lorsqu'il n'y a pas de valeur de proposition disponible quand le consensus $c-1$ se termine. Cette décision conditionne le comportement du consensus c qui sera exécuté. En cas de déclenchement de RO , les accepteurs se verront octroyer l'autorisation par le leader pour lui renvoyer une valeur sur la base des propositions reçues directement des auteurs. Si un quorum d'accepteurs renvoie une valeur unique au leader, cette valeur deviendra la valeur de décision. Dans le cas contraire (absence de quorum d'accepteurs renvoyant une même valeur), il s'agit alors d'une collision et une procédure de recouvrement (coûteuse) sera nécessaire.
Pour la mise en œuvre du test de déclenchement, différentes stratégies sont possibles[72].
Paxos-MIC est un protocole adaptatif qui permet de générer et de stocker efficacement des décisions prises durant une séquence d'instances de consensus.

3.5 Le consensus et la validation atomique

La résolution du problème de consensus fait apparaître la question suivante : le problème de la validation atomique non-bloquante peut-il être résolu dans un système asynchrone augmenté de détecteurs de pannes non-fiables ?
[73] répond négativement à cette question et introduit naturellement une seconde question : pourquoi le problème de la validation atomique non-bloquante est-il plus difficile à résoudre que le problème de consensus dans un système asynchrone avec des détecteurs de pannes non-fiables ?
[73] montre que le problème de la validation non-bloquante ne peut pas être résolu dans un système asynchrone avec des détecteurs de pannes non-fiables à cause de la propriété de non-trivialité. En effet, cette propriété nécessite une connaissance précise, donc fiable, des pannes des participants. [73] montre ensuite qu'en affaiblissant la propriété de non-trivialité, on obtient un nouveau problème, appelée validation atomique non-bloquante faible, qui peut être résolu dans un système asynchrone augmenté de détecteurs de pannes non-fiables[50].

3.5.1 Un protocole de validation basé sur Paxos

Les auteurs dans[74] proposent une solution de validation atomique pour les environnements mobiles où trois rôles sont définis : l'initiateur, les bases de données (les participants) et les coordinateurs. Lorsqu'une application commence une transaction, le nœud où s'exécute l'application devient l'initiateur. Ce dernier choisit un ensemble de coordinateurs qui vont décider de la transaction. Ils définissent un groupe de f nœuds dans le réseau mobile qui sont relativement stables et proches l'un de l'autre (exemple, un saut). Ce groupe est choisi pour former l'ensemble des coordinateurs appelé *cluster de coordinateurs*.
En l'absence de pannes, le protocole fonctionne comme montré dans la figure 3.12. L'initiateur divise la transaction en sous-transactions et envoie les sous-transaction aux bases de données participant à la transaction. Chaque base de données exécute une sous-transaction puis déclenche le protocole de validation en envoyant un message de vote au coordinateur correspondant. Chaque coordinateur est responsable de collecter les votes d'un ensemble de bases de données. Il reste en attente de ces votes pendant un temps limité puis envoie à son tour les votes collectés au coordinateur principal. Ce dernier décide de la transaction en fonction des votes collectés puis envoie le message 'prepare to commit' aux coordinateurs et attend un acquittement de la majorité des coordinateurs. Dans ce cas, le coordinateur principal envoie la décision à l'ensemble des coordinateurs qui se chargent de transmettre la décision aux bases de données associées. La base de données peut demander la décision globale dans le cas où le message est perdu ou si le coordinateur associé tombe en panne avant la réception de la décision.
Le protocole utilise le principe du quorum défini dans Paxos pour terminer la transaction en cas de partitionnement du réseau. Le quorum permet la prise de décision lorsque la majorité des coordinateurs sont d'accord.
Chaque coordinateur peut suspecter le coordinateur principal et devenir lui-même le coordinateur par intérim. Le concept d'un numéro de rang est introduit afin d'avoir un seul coordinateur intérimaire à la fois. Le nouveau intérimaire s'attribue un numéro de rang supérieur à celui du principal et à n'importe quel autre intérimaire déjà observé. Ensuite, il

demande aux coordinateurs de lui transmettre l'état de la transaction dont ils disposent. Si les coordinateurs sont en état d'attente, la transaction est annulée. Cependant, si au moins un coordinateur est en état 'prepare to commit' l'intérimaire adopte cette décision et valide la transaction.

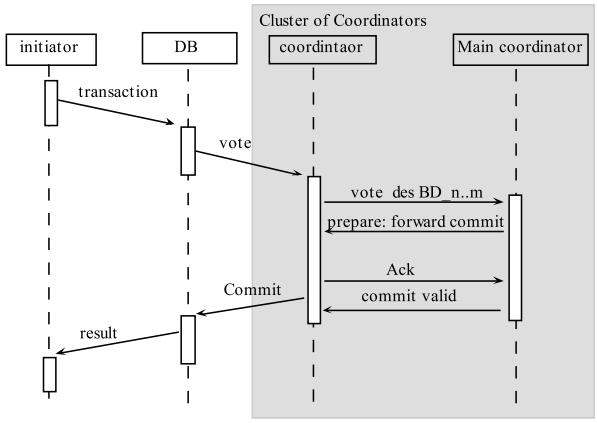

FIGURE 3.12 – Le protocole basé sur les coordinateurs multiples

La panne d'un coordinateur peut causer l'annulation de la transaction si la panne s'est produite avant la transmission des votes au principal.

3.6 Conclusion

Nous avons présenté les différents protocoles de validation atomique proposés pour assurer la propriété d'atomicité des transactions réparties. Bien que le problème de la validation atomique soit étudié depuis longtemps, la recherche reste très active sur ce point, montrant bien le caractère fondamental de ce problème et son intérêt aussi bien théorique que pratique. Nous avons fait le point sur les protocoles de validation non-bloquants dans le cadre des systèmes synchrones et asynchrones.
Ensuite, nous avons introduit le problème de consensus, intensivement utilisé comme brique de base pour construire des applications réparties.
Le consensus n'intervient pas directement dans la résolution de la validation atomique non bloquante. Cependant, un protocole de validation atomique peut être augmenté avec une brique de consensus pour tolérer la panne du coordinateur. Et donc, assurer l'accord (sûreté) et la terminaison (vivacité) du protocole dans un environnement asynchrone où la détection de pannes n'est pas fiable.

Chapitre 4

Solution Proposée

UN agent mobile transactionnel tolérant aux pannes est destiné à exécuter une transaction distribuée sur un réseau ouvert (un environnement asynchrone) sujet à des pannes. Une transaction est un ensemble de requêtes exécutées dans un ordre séquentiel. Lorsque l'agent exécute une requête sur un nœud le résultat ne dépend pas uniquement de l'état actuel du nœud mais aussi des informations fournies par l'environnement extérieur. Par exemple, les réseaux de capteurs, qui sont maintenant intégrés partout, peuvent fournir des données (température, bruit, mouvement, débit d'air, lumière, etc.) pouvant influencer l'exécution de la requête. Une transaction peut être utilisée pour organiser les activités d'un client dans un futur proche. Considérons le scénario suivant :
Arrivant au centre commercial, le client lance une transaction pour exprimer ses besoins d'achats : avant le déjeuner, le client désire trouver une place à l'ombre dans le parking du centre commercial (requête P pour Parking). Ensuite, il veut prendre un petit déjeuner dans une terrasse avec beaucoup de monde mais qui soit calme et à condition que la table soit protégée du soleil et du vent (requête R pour Restaurant). Aujourd'hui c'est l'anniversaire de son amie, après le déjeuner, il veut lui acheter un album d'un groupe de musique parmi les dix meilleurs albums de la semaine mais seulement s'il apparait sur la liste de lecture d'au moins un client qui a le même âge que son amie (requête M pour Musique). De plus, il veut lui offrir un T-shirt noir sur lequel le nom du groupe est imprimé (requête T pour T-shirt).

La transaction $P; R; M; T$ est composée de quatre requêtes où chaque requête est exécutée sur un nœud qui offre le service demandé. Des capteurs sont utilisés pour assurer que les besoins spécifiés sont satisfaits (exemple, pas de bruit dans le restaurant). Des gens à proximité sont aussi consultés (accéder à leur liste de lecture).
Les agents mobiles transactionnels sont bien adaptés à exécuter ce genre de scénarios quand il sont tolérants à des pannes qui peuvent survenir durant leur exécution.

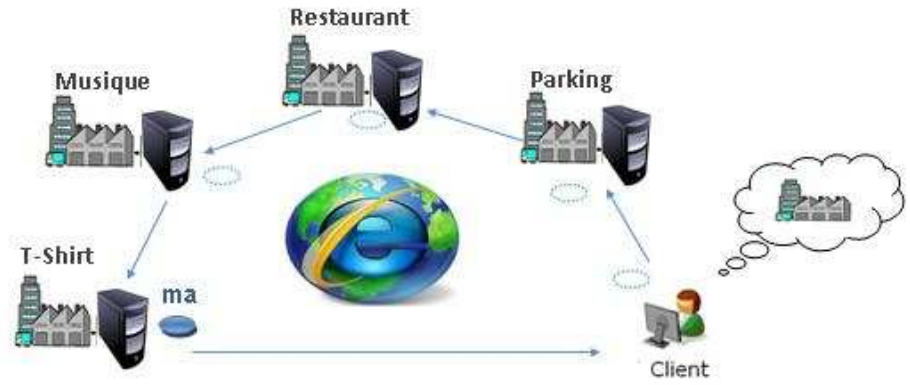

FIGURE 4.1 – Exemple de transaction distribuée exécutée par l'agent mobile

Dans notre solution nous proposons un protocole qui offre un support transactionnel et une tolérance aux pannes pour les agents mobiles afin que l'exécution du scénario présenté ci-dessus devienne une réalité dans un futur proche.

4.1 Le modèle du système d'agents

Nous considérons un système distribué asynchrone où les délais de transmission et la vitesse des processeurs ne sont pas limités. Un exemple d'un tel environnement c'est l'Internet.

4.1.1 Agent mobile

L'agent mobile a_i s'exécute sur une séquence de machines où chaque place p_i $(0 < i < n)$ fournit l'environnement d'exécution logique pour l'agent mobile. L'exécution de l'agent sur la place p_i représente le stade d'exécution de l'agent noté S_i. L'exécution de a_i sur la place p_i génère un changement de l'état interne de l'agent et de la place. L'agent résultat est noté par a_{i+1} et va migrer vers p_{i+1}. Les places où les premier et dernier stades s'exécutent sont appelés respectivement la source et la destination (c'est à dire, p_0 et p_n). La séquence des places entre la source et la destination définit l'itinéraire de l'agent mobile.
Chaque place implémente un service d'annuaire que nous appelons Lookup Directory (LD) qui permet de référencer à partir d'un identifiant toutes les propriétés d'un agent (il est présenté plus loin).

4.1.2 Agent de veille (Watch agent)

C'est un agent stationnaire destiné à contrôler l'exécution de l'agent mobile pour éventuellement détecter sa panne. L'agent de veille est créé par l'agent mobile avant chaque migration. On note wa_i l'agent de veille qui s'exécute sur la place p_i et contrôle un agent sur la place p_{i+1}.

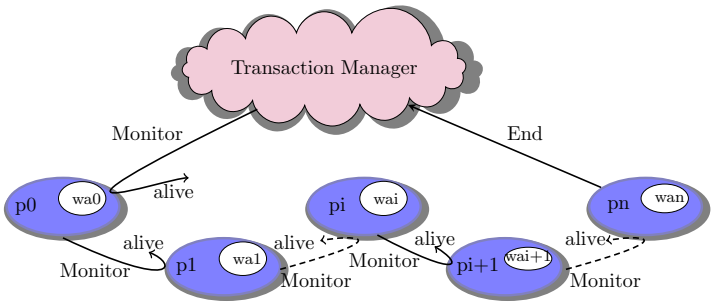

FIGURE 4.2 – Modèle du système d'agents

4.1.3 L'itinéraire

Nous considérons un itinéraire dynamique qui peut être reconfiguré par l'agent mobile lui-même : la place suivante est identifiée à la phase de migration. A chaque stade d'exécution, l'agent est capable de découvrir plusieurs nœuds offrant le même service. Ces nœuds sont appelés *places alternatives*. L'atomicité de l'itinéraire de l'agent est assurée si l'exécution de l'agent réussit au niveau de tous les stades d'exécution sinon elle est annulée (dans le cas où l'exécution d'au moins un stade n'a pas abouti).

4.1.4 Places alternatives

Elles correspondent à un ensemble de places qui offrent un service similaire mais qui ne sont pas des replicas [1] : exemple vente de billet d'avion Alger - Paris. Les places sont fournies par différentes compagnies aériennes (exemple : Air Algérie, Aigle Azur ou Air France). Les places alternatives du stade S_i sont notées en ajoutant un deuxième index à la place : p_{ix} où x identifie le fournisseur de service.

4.1.5 Le gestionnaire de transactions (Transaction manager)

C'est un service qui prend en charge le contrôle de l'exécution de la transaction répartie (Enregistrement et identification, initialisation, validation ou annulation) et dont le fonctionnement sera détaillé lors de la présentation de la solution.

4.1.6 Les fautes d'infrastructure

Dans notre approche nous considérons uniquement les pannes par arrêt (crash failures) ; les fautes malicieuses ne sont pas considérées. Les pannes peuvent survenir à différents niveaux : une place peut tomber en panne provoquant la perte de tous les agents qui s'exécutent sur cette place ; la panne de la machine cause le même effet : toutes les places et les agents sur cette machine sont aussi perdus. La panne des liens de communication peut causer l'isolation des nœuds ou le morcellement du réseau.
Une seule panne peut bloquer l'exécution de l'agent jusqu'au recouvrement de l'entité en panne. La tolérance aux pannes peut être assurée en utilisant la réplication. Si une place tombe en panne l'agent peut être exécuté sur une place alternative. Ce mécanisme de réplication permet de masquer les pannes et rend l'exécution de l'agent mobile non bloquante.

1. Copies exactes

Comme nous considérons que la détection de pannes n'est pas parfaite, ce mécanisme de redondance peut mener vers des exécutions multiples de l'agent si jamais il y a une fausse suspicion de l'agent.

4.1.7 L'indisponibilité de service

L'indisponibilité de service provoque une faute dans la sémantique d'exécution de la transaction. L'agent qui exécute cette transaction ne peut pas continuer son déplacement vers les places suivantes à cause de la propriété d'atomicité. Elle survient lorsque le service demandé n'est pas délivré. Prenons l'exemple de la réservation d'un billet d'avion et d'une chambre d'hôtel où les deux services doivent être satisfaits ou aucun. Lorsqu'il n'y a plus de billets d'avion, l'agent ne peut pas enchainer sur l'exécution de la requête suivante à savoir la réservation d'une chambre d'hôtel. Nous appelons cette indisponibilité de service **faute sémantique**.
Nous considérons les trois types de pannes sémantiques (présentées dans le chapitre 2) :

1. Durant la phase d'exécution quand l'agent n'obtient pas la ressource (service non disponible) ;
2. Lorsque la ressource est attribuée à une autre transaction dans le cas d'une pré-réservation (préemption de ressources) ;
3. Quand une place déjà visitée tombe en panne avant la validation de la transaction.

4.2 Le modèle transactionnel

Dans les systèmes répartis, une transaction est définie comme une composition de requêtes séparées destinée chacune à être exécutée sur un nœud distant. Dans ce qui suit nous reprenons quelque définitions et introduisons des notations pour définir le modèle transactionnel adopté dans notre solution.

4.2.1 La transaction distribuée

Une transaction T est une séquence de n requêtes. Elle est caractérisée par un identificateur unique α et elle est notée T_α. Nous supposons que le client génère un nouvel identificateur pour chaque nouvelle transaction. Une transaction consiste en n requêtes successives :$T_\alpha \equiv R_\alpha^1; R_\alpha^2; ...; R_\alpha^i; ...; R_\alpha^n$. L'indice i représente le stade d'exécution S_i de la requête R_i, n représente la longueur de la transaction (nombre de stades d'exécution) en terme de requêtes. Le modèle transactionnel est caractérisé par les propriétés ACID (*Atomicity, Consistency, Isolation, Durability*). L'exécution de la transaction passe par deux phases : une phase d'exécution et une phase de validation. Dans la première phase, pour chaque requête le nœud qui peut exécuter le service est trouvé. La seconde phase commence lorsque n nœuds auront exécuté les n requêtes avec succès : un protocole de validation atomique est utilisé pour assurer l'atomicité de la transaction.

4.2.2 La requête

Une requête R_α^x exprime le besoin du client pour demander un service qui peut être fourni par un nœud. Une requête est une transaction locale : son exécution correspond à une

séquence d'opérations de lecture et d'écriture sur les données locales du nœud fournisseur de service. Pour une requête donnée, plusieurs nœuds alternatifs, en mesure de satisfaire cette requête, sont identifiés. Le nombre de nœuds alternatifs dépend de la demande du client. Prenons la requête de réservation du restaurant dans l'exemple précédent, un client qui cherche un endroit pour manger est moins difficile à satisfaire qu'un client qui cherche un restaurant très fréquenté mais qui soit calme et avec une table protégée du soleil et du vent. Lorsque la demande du client est très précise et stricte elle risque de ne pas être satisfaite et par conséquent la transaction distribuée est annulée. Cependant, si la requête est plus large en terme de conditions de satisfaction, elle sera fort probablement satisfaite, ce qui augmente les chances de validation de la transaction globale. Pour cela, nous définissons pour chaque requête des *niveaux de satisfaction*.
Dans ce cas, une requête R_α^x est représentée par k requêtes alternatives notées $R_\alpha^{x_1}$, $R_\alpha^{x_2}$,...$R_\alpha^{x_k}$ où la requête $R_\alpha^{x_k}$ représente le niveau de satisfaction le plus élevé alors que $R_\alpha^{x_1}$ exprime les besoins minimaux de la requête. Au niveau de chaque stade d'exécution S_i, les requêtes sont examinées dans un ordre décroissant de k à 1 jusqu'à ce qu'un nœud satisfaisant une requête $R_\alpha^{x_{k'}}$ où $1 \leq k' \leq k$ soit trouvé.
En définissant plusieurs niveaux de satisfaction pour chaque stade d'exécution le nombre de nœuds alternatifs augmente et la requête aura plus de chance d'être satisfaite. Dans notre exemple, si après avoir trouvé une place dans le parking dans le premier stade d'exécution S_0, le client peut exiger un restaurant avec un menu qui ne dépasse pas 15 euros. S'il est impossible de satisfaire cette demande le client peut atténuer sa contrainte et accepter un menu inférieur à 25 euros (une requête alternative avec un niveau de satisfaction moins élevé). Il est important de définir les niveaux de satisfaction notamment dans un contexte transactionnel où les transactions distribuées peuvent être très longues (le nombre de stades d'exécution est très grand) et qui risquent d'être annulées à cause d'une requête non satisfaite et pour laquelle le client est prêt à faire des concessions.
Dans notre exemple, la requête M lorsqu'elle est satisfaite retourne une variable pour contenir le nom du groupe de musique. Cette variable est instanciée lorsque la requête est satisfaite. Nous appelons ***variable libre*** une valeur qui doit être retournée par l'exécution de la requête quand elle est satisfaite. Une requête est définie par des niveaux de satisfaction et par un ensemble de variables libres :

$$R_\alpha^{x_i} = \{\{N_i/i = 1, n\}, \{v_j/j = 1, m\}\}$$

Nous appelons requête fermée, une requête qui a un seul niveau de satisfaction et qui n'a pas de variables libres : $R_\alpha^{x_i} = \{\{N_i/i = 1, n\}\}$.
Nous appelons requête ouverte, une requête avec plusieurs niveaux de satisfaction et/ou plusieurs variables libres : $R_\alpha^{x_i} = \{\{N_i/i = 1, n\}, \{v_j/j = 1, m\}\}$.
Deux exécutions d'une même requête ouverte sur des nœuds différents donnent des résultats différents si elles sont satisfaites avec des niveaux différents ou des valeurs distinctes des variables libres. Deux exécutions d'une même requête fermée donnent des résultats *incompatibles* s'ils utilisent des résultats d'une requête ouverte précédente (voir figure 4.3).

Lorsque la requête R_α^x est satisfaite le résultat peut être utilisé par une autre requête R_α^y tel que $(x \leq y)$. Dans ce cas R_α^y est dite dépendante de R_α^x. Par exemple, dans la transaction $P; R; M; T$, la requête de réservation d'une table au restaurant(R) doit savoir la tranche de temps pour la quelle le Parking est réservé (P). De même pour acheter un T-shirt avec le

nom du groupe de musique imprimé dessus (T), le disque de musique choisi doit être connu (M). Pour cela, nous posons la règle suivante : les requêtes d'une transaction doivent être exécutées séquentiellement selon l'ordre défini par le client et non pas en parallèle.
Cette règle a un impact négatif sur la durée d'exécution de la transaction qui dépend du nombre de requêtes composant la transaction. Cependant, elle assure la cohérence d'exécution des requêtes.

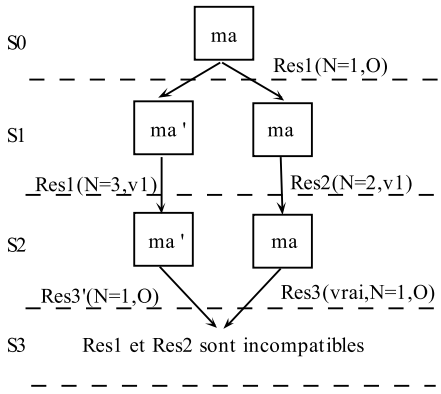

FIGURE 4.3 – Exécutions incompatibles de la même requête

4.3 Le protocole de tolérance aux pannes

La solution proposée est basée sur la réplication temporelle et les points de reprise pour assurer la tolérance aux pannes de l'agent mobile. Dans notre approche, l'agent mobile exécute une transaction distribuée composée d'un ensemble de requêtes où chaque requête est exécutée sur un nœud (place) du réseau. L'agent mobile assure l'atomicité de la transaction en suivant l'approche de validation à la destination (*Commit-At-Destination*). Un protocole de validation atomique est appliqué à la fin de l'exécution de toutes les requêtes de la transaction distribuée.

4.3.1 L'exploration de chemins et la réservation de services

Lorsque le client lance une transaction T_α, un agent mobile a_α est créé. Cet agent migre vers plusieurs places pour exécuter les requêtes de la transaction. L'agent commence son exécution sur une place p_0 et migre n fois pour découvrir son itinéraire complet. Sur chaque place p_i l'agent identifie les places alternatives du stade suivant S_{i+1} capables de satisfaire la requête suivante R_α^{i+1}. Les identités des nœuds qui peuvent satisfaire les requêtes ne sont pas spécifiées dans la transaction, pour cela un mécanisme de découverte des nœuds alternatifs pour chaque requête est requis. Nous supposons que les nœuds sont enregistrés dans des dépôts publiques en tant qu'alternatives éventuelles pour ce type de requêtes. Dans un contexte commercial cet enregistrement est naturel et fait partie de la promotion de service. Ce mécanisme permet d'informer les clients potentiels sur les services offerts par ces nœuds. Le processus de découverte de service ne fait pas l'objet de cette étude sachant qu'il existe plusieurs mécanismes de découvertes décrits dans la littérature.

Une fois les places alternatives identifiées, elles sont visitées par l'agent mobile l'une après l'autre jusqu'à trouver la place qui satisfait la requête. L'ordre de visite des nœuds alternatifs est défini selon les niveaux de satisfaction disponibles sur chaque nœud en commençant par le niveau le plus élevé. A titre d'exemple si le client veut que le restaurant soit proche du magasin de musique les nœuds alternatifs pour la requête de musique sont ordonnées selon leurs distances physiques par rapport au restaurant réservé ; l'agent mobile commence par visiter le nœud alternatif le plus proche.
Lorsque la place visitée est capable de satisfaire la requête de l'agent mobile, ce dernier, avant de migrer vers le stade suivant, effectue une pré-réservation du service où des ressources sont allouées à cette requête. Cette pré-réservation n'est validée qu'après la satisfaction de toutes les requêtes.
Cependant, si l'une des requêtes n'est pas satisfaite toute la transaction est annulée, cela veut dire que dans une transaction longue une pré-réservation effectuée par l'agent mobile sur une place peut être annulée après une longue durée d'allocation de ressources. Cette pénalisation des ressources qui ne sont pas validées à la fin peut être nuisible au fournisseur de service qui refuse de satisfaire d'autres requêtes demandant la même ressource. Pour contourner ce cas indésirable, nous supposons que le fournisseur de service garde le contrôle de ses ressources le plus longtemps possible, et ce, en allouant les ressources pour une période de temps limitée. Donc le fournisseur de service peut annuler la réservation (expiration du temps limite sans que la transaction soit décidée) et ré-allouer la ressource pour une autre requête. Le mécanisme d'allocation est flexible de telle sorte que le fournisseur de service peut ré-allouer les ressources à la requête annulée une nouvelle fois pour augmenter les chances de validation de la transaction si elle n'est pas encore décidée.

Dans notre exemple, le fournisseur de service peut mettre à part un article (une place dans le parking, une table dans un restaurant, un disque de musique ou un T-shirt). Durant la période de réservation ce produit est considéré déjà vendu. Cependant, le fournisseur de service peut annuler cette allocation temporaire. Pour garder la trace de ces réservations qui peuvent changer (expiration du timeout) d'état, nous définissons sur chaque place une table que nous appelons table de pré-préservation de requêtes **(*Precommit Request Table : PRT*)**.

4.3.1.1 Table de pré-réservation de requêtes (PRT)

Au niveau de chaque stade d'exécution S_i l'agent mobile exécute la requête R_α^i et obtient le résultat. Si la requête est satisfaite (résultat retourné non nul), une table de pré-réservation PRT_α^i associée au stade d'exécution S_i est créée pour garder le résultat de cette réservation en insérant une nouvelle entrée dans cette table. Une entrée dans la table PRT_α^i contient les champs suivants :

1. Tag : c'est une étiquette associée à l'agent mobile pour chaque stade d'exécution. Nous verrons plus loin comment ce tag est construit.
2. Result : le résultat obtenu suite à l'exécution de la requête avec succès.
3. State : l'état actuel de la requête exécutée, il est initialisé à *Booked* qui signifie que la requête est satisfaite et des ressources lui sont allouées.

Le champs *State* de la table **PRT** peut changer de valeur selon l'état de la requête. La requête peut passer par six états entre la phase d'exécution et la phase de validation à savoir :

Booked, *Cancelled*, *PrepareYes*, *PrepareNo*, *Commit* et *Abort*. Les transitions entre ces différents états sont présentées dans la figure 4.4.

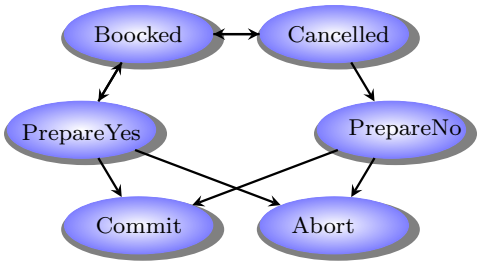

FIGURE 4.4 – Les différents états de la requête et les transitions entre eux

Quand l'état de la requête R_α^i est à *Booked*, les ressources sont allouées à la requête, cette réservation peut être annulée, on parle donc de préemption de ressources. Cette préemption de ressources peut avoir lieu pour plusieurs raisons (expiration du temps de réservation, allocation de ressources à une autre requête, etc.). Les conditions et les règles de priorités qui régissent la préemption de ressources ne font pas partie de cette étude. Dans le cas où les ressources sont préemptées, l'état de la requête dans la table **PRT** passe de *Booked* à *Canceled*. Une nouvelle réservation est toujours possible lorsque le service est disponible ; il est réalloué à nouveau pour une autre période de temps. Quand la place est invoquée pour préparer la validation de la transaction via un message de vote, l'état de la requête passe à *PrepareYes* ou à *PrepareNo* selon la réponse de vote *Yes* ou *No*. Ensuite, l'état de la requête passe à *Commit* ou *Abort* selon le message de décision de la transaction.
Le message de décision peut contenir la valeur *Release* ce qui entraine le changement de l'état de *PrepareYes* à *Booked* ou reste à l'état *PrepareNo*.
La place doit recevoir un message de la part de l'entité qui exécute le protocole de validation atomique pour faire transiter la requête à un des quatre derniers états. Lorsque l'état de la requête est mis à *PrepareYes* la place s'engage à assurer la validation de l'exécution de la requête (pas d'annulation entre le PrepareYes et le Commit).

4.3.1.2 L'interface de réservation de service

Pour offrir aux agents un accès homogène et direct au service présent sur chaque site, nous définissons une interface de réservation de services (voir figure 4.5) qui permet aux agents d'exécuter leurs requêtes et de valider les résultats.

Interface Reservation Service
(1) Execute (T_α, R_i))
(2) vote(T_α, R_i, Tag)
(3) decide$(T_\alpha, R_i, Tag, outcome)$
(4) getOutcome(T_α, R_i)

FIGURE 4.5 – Interface de réservation de Service

La méthode Execute (T_α, R_i) exécute la requête R_i de la transaction T_α et retourne le

résultat. Ce résultat est nul quand le service n'est pas disponible et la requête ne peut pas être satisfaite. Nous supposons que le service est réservé pour une période limitée, si pendant cette période la transaction n'est pas décidée (validée ou annulée) la réservation est annulée.
Chaque place définit deux primitives non bloquantes, $vote()$ et $decide()$, pour représenter les fonctionnalités de la validation. La primitive $vote()$ prend en entrée trois paramètres : l'identifiant de la transaction T_α, la requête R_i et le *Tag* de l'agent mobile. Elle retourne le vote qui prend ses valeurs dans le domaine $Vote = yes, no$. Le vote par un 'yes' indique que la place est capable de valider le résultat. La primitive $decide()$ prend en entrée quatre paramètres : l'identifiant de la transaction T_α, la requête R_i, le *Tag* de l'agent mobile et la décision *outcome* qui prend ses valeurs dans le domaine $Outcome = \{Commit, Abort, Release\}$. Elle retourne un accusé de réception. Lorsque la primitive decide() est invoquée avec la valeur de décision *Commit* ou *Abort* ; elle modifie l'état de la requête dans la table PRT(T_α, R_i) à la valeur qui correspond à la décision (Commit ou Abort) et force toutes les autres entrées de la même table à *Abort* pour s'assurer qu'une seule exécution de la requête est validée sur chaque place.
Cependant, si la valeur de la décision est *Release*, ceci veut dire que la transaction globale n'est pas encore décidée et que la place doit faire un retour arrière à l'état précédent. Si l'état est *PrepareYes*, ceci signifie que la place a voté *Yes* et qu'elle est capable de valider le résultat et peut être invoquée une autre fois pour voter. Dans ce cas, l'état est remis à *Booked*. Inversement, *PrepareNo* veut dire que la place a voté *No* pour cette transaction et ne va plus être sollicitée pour un vote. Donc l'état *PrepareNo* est maintenu.
La place offre la méthode $getOutcome()$ pour permettre aux agents d'accéder aux tables *PRT* et prendre connaissance de l'état de la transaction. Cette méthode est invoquée avec les deux paramètres : T_α et la requête R_i. Elle parcourt la Table *PRT* et vérifie la présence d'une entrée avec un état validé (Commit ou Abort) et retourne le résultat.

En identifiant les nœuds fournisseurs de services, l'agent mobile découvre son chemin et crée son itinéraire. Deux évènements peuvent arrêter la progression de l'agent mobile : l'apparition d'une panne (faute d'infrastructure : place, agent ou lien) ou alors l'indisponibilité du service demandé (faute sémantique). Pour éviter le blocage de l'agent et assurer sa terminaison nous proposons un protocole de détection de pannes et de recouvrement que nous décrivons ci-après.

4.3.2 Détection de fautes et recouvrement

Un système réparti asynchrone est caractérisé par l'absence de borne supérieure sur le temps d'acheminement des messages. Dans ce contexte, il est impossible de différencier un site qui est en panne d'un site avec lequel les communications sont extrêmement lentes.
De ce fait, la détection de pannes dans un système asynchrone ne peut être qu'approximative (i.e., non-fiable). La détection de pannes est un point crucial pour la résolution des problèmes de tolérance aux pannes.
Nous définissons un mécanisme de contrôle créé et activé par l'agent mobile tout au long de son itinéraire. Plus précisément, sur chaque place p_k et avant sa migration, l'agent mobile effectue un checkpoint pour sauvegarder son code et son état puis crée un agent de veille que nous appelons *watch agent* pour le contrôler. L'agent envoie périodiquement des messages *Alivema*, appelés battements de cœur, à l'agent de veille.

Avant de commencer son exécution, l'agent s'enregistre dans l'annuaire local (Lookup Directory) hébergé par chaque place pour enregistrer les agents ayant déjà exécuté leurs requêtes sur cette place. Chaque entrée dans l'annuaire contient les informations suivantes :

- T_α : Identificateur de la transaction.
- *Tag* : Le tag associé à l'agent mobile.
- *Stage* : Le Stade d'exécution de l'agent mobile.
- *State* : L'état d'exécution de l'agent, il peut prendre quatre valeurs possibles : *Computing* (l'agent est en court d'exécution de la requête), $Satisfied$ (l'exécution de l'agent s'est terminée avec succès et la requête est satisfaite), $Unsatisfied$ (l'agent a terminé son exécution sans faute mais la requête n'est pas satisfaite), *Crashed* (Terminaison anormale de l'exécution de l'agent). La figure 4.6 représente ces différents états.

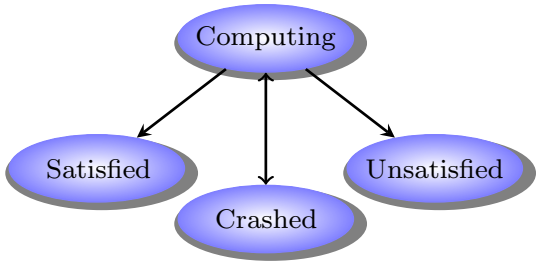

FIGURE 4.6 – Les différents états d'exécution de l'agent mobile

Le principal avantage de l'annuaire décentralisé proposé est que les mises à jour et les consultations sont purement locales ce qui permet d'éviter les problèmes de rupture de communication et évite d'engendrer une surcharge supplémentaire dans un réseau.
L'annuaire LD permet de renseigner l'état actuel de l'agent. Si jamais l'agent de veille ne reçoit pas les battements de cœurs au bout d'un temps limité (timeout) il suspecte une panne. Il envoie une copie de l'agent (en utilisant le checkpoint) vers la place suspectée. Le nouvel agent consulte l'annuaire local pour connaitre l'état de l'agent suspecté et réagit en conséquence.
Lorsque l'état est *crashed* (l'agent est en panne), il recommence l'exécution de la requête.
Cependant, si l'état est *computing* (fausse suspicion : le message *alivema* est retardé ou perdu) il envoie le message *alivema* et s'autodétruit.
Quand il est *unsatisfied* (l'agent a terminé son exécution mais le message correspondant est perdu), il envoie le massage *failed* à l'agent de veille et s'autodétruit.
Enfin, s'il est à *satisfied* il vérifie la présence de l'agent de veille s'il n'existe pas. Il le crée puis il envoie le message *alivewa* et s'autodétruit.
Par ailleurs, si le nouvel agent ne réussit pas sa migration vers la place suspectée, il migre vers une place alternative pour recommencer l'exécution du stade courant. Ce mécanisme de réplication permet de masquer les pannes des nœuds et assure la propriété du non blocage de l'agent mobile.
L'agent mobile est contrôlé par un seul agent de veille qui lui-même peut tomber en panne et doit donc être contrôlé. Nous utilisons une chaine de dépendance composée de l'ensemble des agents de veille créés par l'agent mobile tout au long de l'itinéraire. Dans cette chaine, chaque agent de veille wa_k contrôle son successeur wa_{k+1} et le dernier ajouté à

la chaine contrôle l'agent mobile. Chaque agent de veille envoie le message *Alivewa* à son prédécesseur dans la chaine. Ce mécanisme de contrôle permet la détection des pannes de l'agent ou de la place où il s'exécute à condition que la tête de la chaine soit également contrôlée. Le client peut assurer cette tâche. Dans ce cas il doit rester connecté durant toute la phase d'exploration de chemins. Dans notre solution, nous proposons un service que nous appelons *AS* (Avalability of the Sources) qui assure la disponibilité de la source à la place du client.

Notre détecteur de panne assure la propriété de complétude définie par Chandra et Toueg dans [57], toute panne finit par être détectée par ce mécanisme. Cependant, il ne vérifie pas la propriété d'exactitude c'est à dire, des places correctes peuvent être suspectées. Cette fausse détection de pannes génère la création de plusieurs copies de l'agent mobile sur des places alternatives et donc des exécutions multiples apparaissent. Pour ne pas violer la propriété *exactly_once* de l'agent mobile tolérant aux pannes, la validation de l'exécution de l'agent se fait à la destination (Commit at Destination). Lorsque les copies de l'agent arrivent à la destination une seule copie est validée (le première arrivée) les autres sont annulées. Cependant, les copies de l'agent en double exécutions peuvent se rencontrer sur une place de l'itinéraire avant d'arriver à la destination. A ce niveau là un seul agent continue son exécution les autres sont arrêtés. Nous décrivons ci-après une procédure pour résoudre les doubles exécutions sur les places de l'itinéraire.

4.3.2.1 Résolution des exécutions multiples

Une double exécutions apparaît lorsque le détecteur de pannes fait une fausse suspicion et lance un nouvel agent sur une place alternative alors que l'agent suspecté n'est pas tombé en panne. Les deux agents peuvent se rencontrer sur une place commune de l'itinéraire. Dans notre protocole une place p_i autorise un agent à exécuter une requête R^i_α s'il s'agit de la première exécution du stade S_i sur la place p_i. Une seconde exécution de la même requête R^i_α sur la même place p_i est autorisée si l'agent qui exécute la requête est plus récent.
Nous parlons *d'agent plus récent* s'il est créé sur une place alternative suite à la suspicion d'une panne de l'agent contrôlé.
Avant de résoudre les exécutions multiples, il faut les détecter. La détection anticipée d'une double exécution avant d'arriver à la destination permet de diminuer les réservations multiples des mêmes services pour une même transaction ; cela est possible grâce à l'annuaire *LD*.

Comme l'agent doit s'enregistrer auprès du *LD*, il commence par vérifier l'existence d'un autre agent qui exécute la même requête (même stade) de la même transaction. S'il s'avère que cette entrée existe dans le *LD* et que l'agent correspondant est plus récent que lui, il suspend son exécution et migre vers la destination pour signaler qu'une double exécution a eu lieu sur un préfixe d'itinéraire, et ce, via le message *Double* contenant les places qui composent le chemin parcouru. Par contre si l'agent enregistré dans le *LD* est moins récent, l'agent mobile est autorisé à continuer son exécution sur la place courante.
Afin de distinguer l'agent le plus récent du moins récent nous utilisons un système de marquage qui permet d'associer un *tag* à l'agent mobile à chaque stade d'exécution. Pour un agent donné qui exécute le stade S_i , le *tag* est une séquence de i entiers $t_1 t_2 ... t_i$ où chaque t_j $1 \leq j \leq i$ représente le nombre d'agents qui ont exécuté le stade S_j sur les t_j places alter-

natives. L'entier t_j est le numéro de l'agent courant ayant réussi l'exécution de la requête du stade S_j sur la place alternative $p_i^{t_j}$. Dans le cas d'une double exécution, deux agents mobiles a et a' exécutant une requête d'une même transaction n'ont pas le même *tag*. On dit que a est plus récent que a' si le *Tag* de a est plus grand que celui de a'.

La figure 4.7 illustre les scénarios possibles dans le cas de la transaction $P; R; M; T$. Les fausses suspicions sont désignées par des croix en lignes pointillées alors que les vraies pannes sont désignées par des croix en lignes continues.

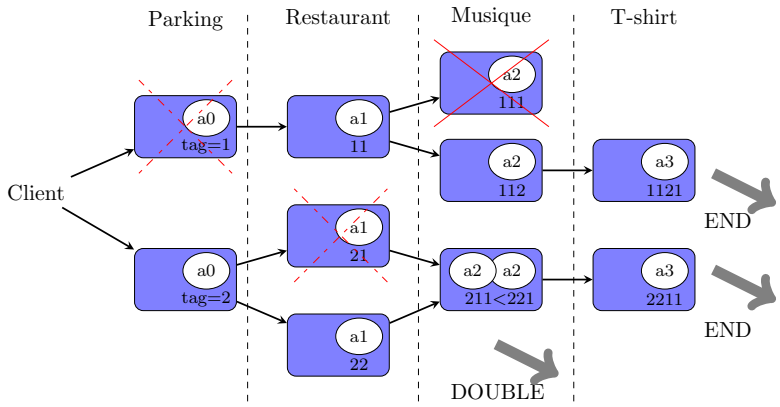

FIGURE 4.7 – Double exécution de l'agent mobile

4.3.2.2 La Résolution des fautes sémantiques

Une faute sémantique survient lorsque le service demandé est indisponible. L'agent mobile envoie le message *failed* à son agent de veille pour l'informer qu'il ne peut pas continuer son chemin car la requête courante n'est pas satisfaite. Lorsque l'agent de veille reçoit le message *failed*, il envoie une nouvelle copie de l'agent mobile vers un nœud alternatif. Cependant, s'il n'existe pas de nœud alternatif qui peut satisfaire la requête, l'agent de veille fait un retour arrière dans la chaîne de contrôle en renvoyant le message *failed* à son prédécesseur. Ce dernier essaie à son tour de trouver un nœud alternatif pour relancer la transaction à partir de ce stade d'exécution et ainsi de suite (voir figure 4.8).

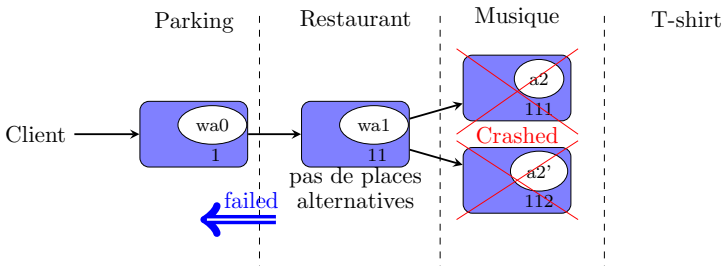

FIGURE 4.8 – Panne sémantique suite à l'indisponibilité de service

La chaîne de dépendance qui assure la disponibilité de l'agent peut être brisée à cause d'une panne d'un nœud de la chaîne (voir figure 4.9). Cette panne est détectée par le prédécesseur du nœud en panne dans la chaîne. Ceci implique que la pré-réservation sur cette

place risque de ne pas être validée si la place n'est pas rétablie avant la validation globale de la transaction. De ce fait, l'agent de veille essaie de trouver un nœud alternatif pour relancer la transaction à partir de ce stade.

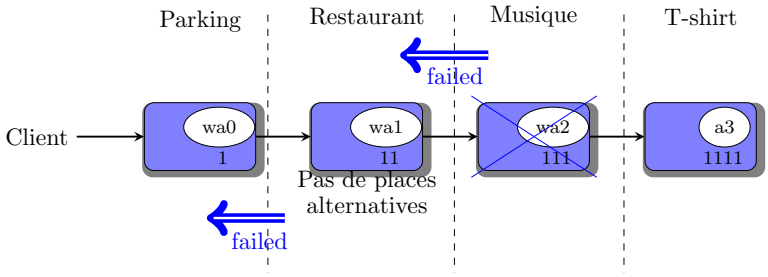

FIGURE 4.9 – Panne de l'agent de veille

Dans notre solution la réservation des ressources pour un service donné peut être annulée par le fournisseur de service pour plusieurs raisons (*timeout*, allocation de ressources pour une autre requête). Dans ce cas, le fournisseur de service se charge d'informer l'agent de veille de cette annulation. Ce dernier informe à son tour son prédécesseur dans le chaîne de dépendance par l'envoi du message *failed*. Suite à la réception de ce message, l'agent de contrôle relance la transaction sur un nœud alternatif 4.9.

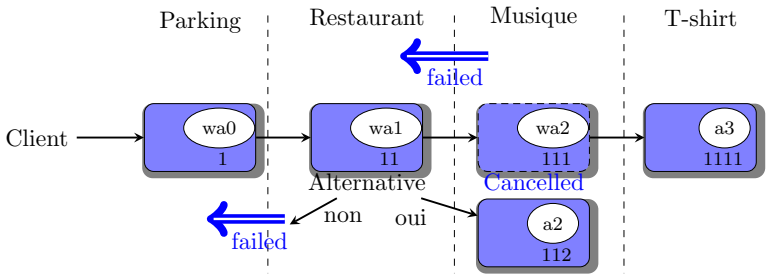

FIGURE 4.10 – Annulation de la pré-réservation

Lorsque l'agent mobile termine, avec succès, l'exécution de toutes les requêtes de la transaction, il migre vers la destination avec le message *END* incluant le chemin complet (itinéraire) où ont été satisfaites les requêtes de la transaction distribuée. Ce message déclenchera le protocole de validation qui permet de rendre permanentes toutes les réservations faites par l'agent mobile durant son premier passage par les places de l'itinéraire.

4.3.3 Implémentation des agents

4.3.3.1 Pseudocode de l'agent mobile

L'algorithme de l'agent mobile est présenté dans la figure 4.11. L'agent commence par s'enregistrer dans le *LD* en ajoutant une entrée contenant les propriétés d'exécution, et ce, après avoir vérifié que le *LD* ne contient aucune entrée relative à la même transaction et le même stade d'exécution.
Dans le cas contraire (un autre agent exécutant la même requête de la même transaction s'est déjà exécuté sur ce site), l'agent compare son *tag* à celui dans le *LD* noté tag' :

```
Class MobileAgent
Transaction[n]={R0, R1, ..., Rn} ; Transaction Identifier Tid ;
Stage ← 0 ;
Tag ← 1 ;
Upon arrival do {
(1) Ri ← [Stage];
(2) if LookDir(Tid, Stage, Tag) ≠ null{
(3)         if state='computing' send alivema to previous ; Die() ;
(4)         if state='unsatisfied' send failed to previous ; Die() ;
(5)         if state='satisfied' {
                Create wa if doesn't exist ;
                Send alivewa to previous ; Die() ;}
(6)        if state='crashed' LookDir (Tid,Tag).state='computing' ;
(7) } else{
(8)         if (LookDir (Tid, stage)≠ null and (Tag is smaller) {
(9)                 Send 'double' to previous ; Moveto (TM,double) ;
(10)          } else {
(11)                  if (LookDir (Tid, stage) ≠ null and (Tag is bigger)
                    and getOutcome (Tid,Ri)≠ null)
(12)                  { Send 'double' to previous ; Moveto (TM,double) ; }
(13)                  Else {
(14)                  insert (Tid, stage, Tag, 'computing') in LookDir ; }
                }
(15) Create wa ; % create the watch agent
(16) Res= Execute (Tid, Ri) ;
(17) if (Res = null) { % request not satisfied
(18) LookDir (Tid,Tag).state='unsatisfied' ;
(19) Send 'failed' to wa ; Die() ;
(22) if (stage ≤ n) {
(23)     Checkpoint () ;
(24)     next ← getAlternative() ;
(25)     if (next ≠ null){
(26)             Tag ← getNextTag() ;
(27)             Moveto (next) ;
(28)     } else {
(29)             Send 'failed' to wa ; Die() ; }
(30) } else { send end to wa ;
(31)     Moveto(TM, End) ; }
(32) } % end
```

FIGURE 4.11 – Le Pseudo code de l'agent mobile

- si $tag = tag'$: Il s'agit du même agent mobile qui a migré vers ce site suite à une suspicion de panne. Il vérifie l'état d'exécution de l'agent mobile suspecté et agit en conséquence.
- si $tag < tag'$: Un agent plus récent a déjà exécuté le service sur ce site. Dans ce cas,

l'agent arrête son exécution et migre vers la destination avec le message *Double* pour signaler une double exécution.

– si $tag > tag'$: Un agent s'est déjà exécuté sur cette place mais l'agent courant est plus récent et peut donc continuer son exécution.

Une fois enregistré, l'agent mobile crée un agent de contrôle et lui délègue l'envoi de ses battements de cœur (les messages *alivema*). Il calcule ensuite le résultat de la requête en appelant la fonction $Execute()$ définie dans l'interface de réservation de service. Cette fonction retourne *nul* si le service n'est pas délivré (requête insatisfaite). Sinon, elle crée une entrée dans la table de pré-réservation PRT relative à cette transaction et y sauvegarde le résultat de la requête en mettant son état à *Booked*. La fonction $Execute()$ ne valide pas l'exécution de la requête, elle fait uniquement une pré-réservation du service.
Si la requête n'est pas satisfaite, l'agent est considéré en panne sémantique et ne peut continuer son exécution. Dans ce cas, il met son état d'exécution dans le LD à $unsatisfied$, envoie le message $failed$ à l'agent de veille sur le site précédent et s'autodétruit.
A l'étape de migration, l'agent doit choisir le nœud vers lequel il va migrer parmi les nœuds alternatifs découverts. Avant sa migration, l'agent mobile calcule son *tag* pour le stade suivant.
Si l'agent mobile ne trouve pas de nœuds alternatifs, il arrête son exécution et informe l'agent de veille sur la place précédente de l'impossibilité de continuer son parcours en lui envoyant le message $failed$. Enfin, l'agent mobile sauvegarde son code et son état sur la place courante (il fait un checkpoint qui sera utilisé pour créer une nouvelle instance de l'agent en cas de panne) et migre vers la place suivante. L'agent mobile indique le succès de l'exécution sur tous les nœuds de l'itinéraire avec le message END.

4.3.3.2 PseudoCode de l'agent de veille

L'agent de veille est créé à chaque stade d'exécution pour contrôler l'exécution de l'agent mobile. L'agent de veille lui-même peut être sujet aux pannes. Pour assurer sa tolérance aux pannes, nous avons utilisé la chaine de dépendance composée de l'ensemble des agents de veille créés tout au long de l'itinéraire. Chaque agent de veille wa_i contrôle son successeur wa_{i+1} et le dernier créé contrôle l'agent mobile *ma* :

$$wa_0 \rightarrow wa_1 \rightarrow \ldots wa_{i-1} \rightarrow wa_i \rightarrow ma$$

Chaque agent de veille envoie le message *alivewa* à son prédécesseur dans la chaine. Quand l'agent de veille reçoit ce message pour la première fois, il commence le contrôle de l'agent de veille sur la place suivante.
La tête de chaine doit être contrôlée afin de préserver la dépendance de la chaine de contrôle. Cette tâche est assurée par le service AS (Availability of the Source) présenté dans les sections qui vont suivre ; la dépendance devient comme suit :

$$AS \rightarrow wa_0 \rightarrow wa_1 \rightarrow \ldots wa_{i-1} \rightarrow wa_i \rightarrow ma$$

Le pseudo code de l'agent de veille est présenté par la figure 4.12. Il est chargé de trois tâches principales :

- Tâche 1 (la délégation) : quand l'agent de veille est créé, il prend en charge l'envoi des battements de cœur (les messages alivema) de l'agent mobile. Il vérifie périodiquement l'état de l'agent mobile pour savoir s'il est toujours en vie et envoie les messages *alivema* à la place précédente. Lorsque l'agent mobile migre vers le nœud suivant, l'agent de veille arrête l'exécution de cette tâche.

- Tâche 2 (le contrôle) : il commence l'exécution de cette tâche après la migration de l'agent mobile. S'il ne reçoit pas des messages après un délai d'attente limité (timeout), il suspecte la panne de l'agent ou celle de la place où il s'exécute. Dans ce cas, il envoie une nouvelle copie de l'agent mobile vers la place suspectée.
  L'agent de veille détecte une panne sémantique dans les trois cas suivants :

  1. La requête n'est pas satisfaite et l'agent mobile envoie le message $failed$

  2. Pas de place alternative pour le prochain stade d'exécution,

  3. Le fournisseur de service annule la pré-réservation (le service n'est plus disponible, délai d'attente expiré, etc.) et envoie le message *cancelled*.

- Tâche 3 (la validation) : En parallèle avec la tâche de contrôle, l'agent de veille assure une troisième tâche qui consiste en la participation dans la validation de la transaction globale. Dans cette tâche l'agent peut recevoir deux types de messages :

  1. $Prepare(Tid, Tag)$: En recevant ce message, l'agent de veille invoque la méthode de vote et retourne sa réponse.

  2. $Decide(Tid, Tag, outcome)$: à la réception de ce message, l'agent de veille invoque la primitive $decide()$ qui agit selon la valeur du *outcome* : si le *outcome* égale *Commit* : la requête est validée sur la place courante, si par contre la valeur du outcome égale *Abort* l'agent de veille abandonne la requête et s'autodétruit. Si enfin, le *outcome* est *Release* : la transaction préparée (phase de vote effectuée) est libérée ceci signifie que les ressources sont déverrouillées sans pour autant valider ou annuler la transaction.

Class Watch Agent
{ $Control = false$;
$Dependency = \{previous, next\}$;
Task 1 :
(1) Periodically do
(2) $getAgentState()$;
(3) $if state = computing$ send alivema to previous
(4) $if state = satisfied$ $Control \leftarrow true$;
(5) $if state = crashed$ $LookDir.state \leftarrow crashed$; $Die()$;
(6) While true {
Task 2 :
$If(Control)\{$
(7) || wait receive alivema
(8) $Reset(timeout)$;
(9) || wait receive alivewa
(10) $Reset(timeout)$;
(11) || wait receive failed
(12) Send ma to an alternative place
(13) || wait receive double
(14) Send double to previous ; $control \leftarrow false$;
(15) || wait receive end
(16) Send end to previous ; $control \leftarrow false$;
(17) || wait Timeout
(18) Restore ma from the checkpoint
(19) || wait receive Canceled
(20) Send failed to previous
(21) || wait receive End
(22) send end to previous, $control \leftarrow false$
(23) Periodically $send(alivewa)$ to previous ;
}
Task 3 :
(24) || wait receive prepare
(25) $Sendvote(Tid, R_i, Tag)$;
(26) || wait receive $decide(outcome)$
(27) $Senddecide(Tid, R_i, Tag, outcome)$; $Die()$;
} % end while
} % end class

FIGURE 4.12 – Le Pseudo code de l'agent de veille

4.4 La validation atomique

Nous adoptons une approche de validation à la destination (Commit at destination approach). Un protocole de validation atomique est lancé lorsque n nœuds, qui ont satisfait les n requêtes de la transaction, sont identifiés dès l'arrivée de l'agent mobile à la destination

avec le message *END*.
Le protocole de validation assure que les *n* nœuds de l'itinéraire se mettent d'accord sur une décision unique quant à la validation ou l'annulation de la transaction globale. Le protocole de validation est généralement initié par un leader (statique ou dynamique) qui se charge de coordonner les nœuds qui ont participé à la transaction afin de prendre une décision.
En fait, sur chaque nœud, il existe un agent de veille créé par l'agent mobile dans le but d'assurer la tolérance aux pannes. En parallèle à la tâche de contrôle, cet agent reste à l'écoute des messages en provenance du coordinateur pour participer à la prise de décision.
Notre protocole est basé sur le principe de validation à deux phases. Dans la première phase, appelée phase de préparation, le coordinateur envoie une demande de vote à tous les agents de veille sur les nœuds participants afin de préparer ces nœuds pour valider la transaction. En clair, sur chaque nœud, l'agent de veille vote par un *'Yes'* s'il est en mesure de valider la transaction. Cela veut dire que le nœud s'engage à garantir la validation de l'exécution par le verrouillage effectif des ressources à son niveau. Cependant, l'agent de veille répond à la demande de vote par un *'No'* quand il est dans l'impossibilité de valider la transaction et dans l'incapacité de rendre permanent l'exécution de la requête sur le nœud correspondant (réservation annulée par le fournisseur). Durant la seconde phase, appelée phase de *Dcision*, le coordinateur décide de valider (*Commit*) ou d'annuler (*Abort*) la transaction. Lorsque le coordinateur reçoit *n* réponses positives (de type *'Yes'*), il décide de valider la transaction et envoie le message *Commit* vers les *n* nœuds de l'itinéraire (voir figure 4.13).

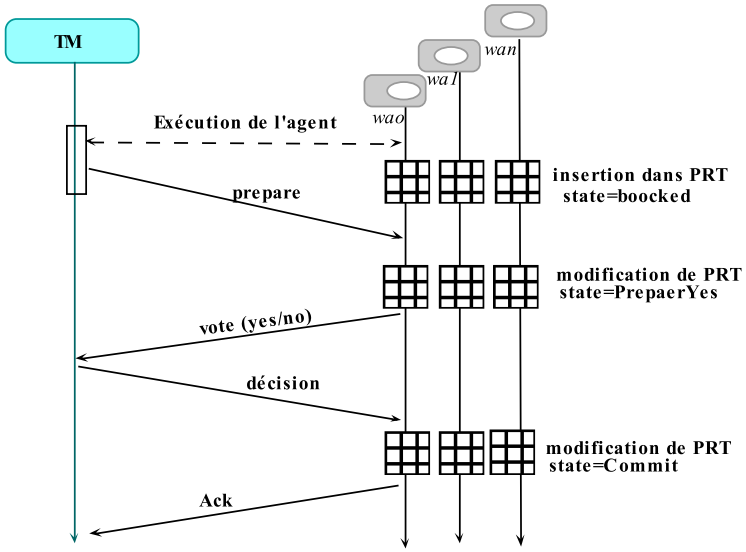

FIGURE 4.13 – Validation de l'exécution de l'agent mobile

Un nœud vote par un *'Yes'* quand il accepte de valider la transaction. La réponse *'No'* est toujours retournée par un nœud qui a détecté un problème local qui risque d'empêcher la validation de l'exécution de la requête. Durant la phase de décision (*Commit* phase), le coordinateur décide d'annuler ou de valider la transaction et en informe les *n* nœuds. Pour décider de valider (commit), le coordinateur doit recevoir *n* votes de types *'Yes'*. Sinon, lorsque le coordinateur reçoit au moins un vote de type *'No'* ou alors le nombre de réponses reçues est inférieur à *n* (des votes sont manquants à cause de la panne d'un nœud, la lenteur

d'exécution du nœud, perte des messages ou délais de transmission très grands) la décision d'annulation est nécessaire.
Quelle que soit la décision prise par le coordinateur, les nœuds doivent exécuter les actions nécessaires correspondantes. Si le nœud répond par un *'Yes'*, il accepte d'exécuter complètement sa requête locale quelque soit les circonstances. Ceci dit, les ressources locales sont verrouillées. Partant de là, la décision finale (Commit ou Abort) doit être connue par ce nœud quelles dans toutes les circonstances. Le protocole de validation, en général, doit assurer une propriété forte : tout nœud contacté par le coordinateur durant la phase de préparation doit être informé de la décision finale. Donc il doit notifier la décision à tous ces nœuds même à ceux qui n'ont pas répondu. Ainsi les ressources ne risquent pas de rester verrouillées au niveau des nœuds.
Dans notre solution, les replicas de l'agent peuvent réussir l'exécution d'une même requête sur des nœuds alternatifs. Ces doubles exécutions génèrent des chemins complets ou des préfixes de chemins où des requêtes de la transaction ont été exécutées avec succès. De ce fait et afin d'augmenter les chances de validation d'une transaction, le coordinateur ne décide pas de l'abandonner juste après la collecte des votes. Il regarde dans les préfixes des itinéraires parcourus par les replicas de l'agent, s'il existe des nœuds qui peuvent remplacer ceux qui n'ont pas répondu ou qui ont répondu négativement durant la phase de préparation.
Une place peut en remplacer une autre dans un itinéraire si et seulement si les deux places ont exécuté avec succès le même stade (même requête) et que leurs résultats sont *compatibles*.
Pour une transaction donnée, le coordinateur peut ré-exécuter le protocole de validation plusieurs fois. Durant chaque exécution, n nœuds sont identifiés pour construire le nouvel itinéraire à tester jusqu'à arriver à décider de la transaction.
Pour mettre en place cette solution, nous définissons trois valeurs possibles que peut prendre la décision d'une transaction : *Abort, Commit ou Release*.

- *Commit* : Elle est prise par le coordinateur lorsqu'il reçoit n réponses de vote de type *Yes*. Après cette décision, le coordinateur ne ré-exécute plus le protocole de validation pour cette transaction.
- *Abort* : Elle est prise par le coordinateur lorsqu'il ne reçoit pas les n réponses de vote de type *Yes* et qu'il n'existe plus de nouveaux nœuds alternatifs à tester.
- *Release* : Lorsque les réponses de vote ne sont pas toutes positives (*Yes*), le coordinateur décide de reconstruire un nouveau chemin en utilisant les nœuds alternatifs visités par les replicas de l'agent. Avant de lancer à nouveau le protocole, il envoie un *Release* à tous les nœuds participants afin de les informer que la transaction n'est pas encore décidée. A la réception du message *Release*, les nœuds reviennent à l'état précédant la phase de préparation (déverrouillage des ressources).

Le protocole de validation atomique est exécuté par une entité que nous appelons gestionnaire de transaction (Transaction Manager) noté TM. Lorsque les n nœuds sont identifiés, l'interaction entre le TM et ces nœuds se fait via deux primitives *Vote()* et *Outcome()*. Les deux primitives sont appelées par le TM. Dans les deux cas un message est diffusé aux nœuds. Ce message contient l'identificateur de la transaction, l'identificateur de la requête et le *tag*. La primitive $Vote()$ déclenche le vote. En réponse, chaque nœud retourne un *Yes* ou un *No*. La primitive $Outcome()$ informe les nœuds de la décision prise ($Abort, Commit, Release$). En réponse, chaque nœud retourne un acquittement.

4.5 Les services du gestionnaire de transactions

Le gestionnaire de transaction est destiné à gérer l'exécution de la transaction depuis son lancement jusqu'à sa terminaison par une validation ou un abandon. Deux services sont fournis par le gestionnaire de transaction : La disponibilité de la source (*Availability of the source*,AS) et la validation atomique (*Atomic Commitment*, AC).
Lorsque le client lance une nouvelle transaction T_α il active le service AS par l'envoi du message *Req_AS*. A travers ce service, le client délègue au TM toutes les activités liées à l'activation de l'agent et le contrôle de la place source (tête de la chaîne de contrôle). Ceci dit, le client peut se déconnecter et se reconnecter plus tard pour récupérer le résultat de la transaction.
Le message *Req_AS* contient l'identifiant de la transaction *Tid* et un descripteur *Desc* où sont sauvegardées les informations statiques concernant la transaction à savoir : le nombre de stades (variable *Stages*), le code utilisé pour créer une instance de l'agent sur la place source (variable *code*), les places sources potentielles (variable *Places0*) et un *timeout* dont l'utilisation est détaillée plus loin (représenté par la variable Δ).
Périodiquement, le service AS vérifie l'état de la place source où un agent a_α est déjà activé. Si cette place n'est plus disponible (elle ne répond pas), le service AS cherche une autre place source : il appelle la fonction *NewPlace()* avec la variable *Places0* comme paramètre. Cet appel retourne la place source suivante sinon la valeur $\perp$ est retournée pour indiquer qu'il n'y a plus de place sources. Le mécanisme qui peut être utilisé à ce niveau n'est pas spécifié. Dans une solution simple la variable *Places0* est une liste de places prédéfinies et la fonction *NewPlace* retourne une place qui n'est pas encore explorée. Des solutions plus sophistiquées basées sur des services de découverte dynamique peuvent être utilisés. Indépendamment de la solution adoptée, si la recherche parvient à trouver une place source disponible, l'agent est activé sur cette place. Cependant, s'il n'existe plus de place source, un message *Req_AC* avec un itinéraire vide et un état d'exécution égal à *Failed* est adressé au service AC. C'est ainsi que le service AS informe le service AC qu'après l'échec d'exécution sur toutes les places sources, l'agent est incapable de terminer l'exécution de la transaction.
Le service AS doit garder les identités de toutes les places sources où l'agent a_α a été lancé. Le détecteur de pannes vérifie périodiquement l'état de place source et maintient une liste des places suspectées. Le détecteur de fautes n'étant pas fiable, toutes les places sources déjà visitées sont contrôlées et pas uniquement la dernière visitée. Si le détecteur de pannes était fiable seulement une seule place, la dernière place source où l'agent mobile est activé, est à contrôler. Nous définissons la fonction $Alive(Kp)$ dont l'invocation retourne *false* si et seulement si toutes les places sources contenues dans la liste Kp sont suspectées. Le service AS a en charge le contrôle des places sources (où la copie initiale de l'agent a été lancée) où se trouve la tête des chaines de contrôle. Ceci permet d'assurer la détection et le recouvrement des pannes tout au long de l'itinéraire de l'agent.

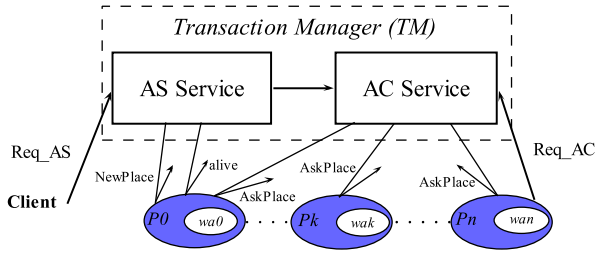

FIGURE 4.14 – Interaction entre l'agent et le Transaction Manager

Le service AC est activé lorsque le *TM* reçoit un message *Req_AC* envoyé par un agent ou le service *AS*. Comme illustré dans la figure 4.14, l'agent migre dans le réseaux ad-hoc pour découvrir son itinéraire puis retourne vers le *TM* pour signaler la terminaison de l'exécution de la transaction. L'agent transmet au *TM* un message décrivant l'état d'exécution de la transaction. Il peut être *End*, *Double* ou *Failed*. Ce message contient également les places visitées par l'agent durant son périple. Les services *AS* et *AC* sont indépendants. Lorsque plusieurs transactions sont exécutées simultanément, ces deux services s'exécutent en concurrence. Mais en considérant une transaction isolée des autres, ces deux services sont utilisés séquentiellement : dès qu'un itinéraire est connu par le service *AC* la disponibilité de la source n'est plus à assurer. De même si aucun itinéraire n'est connu, le service *AC* n'est pas activé. Comme les deux services ne s'interfèrent pas, ils peuvent partager les mêmes structures de données d'une même transaction.

Une seule machine peut agir comme un seul gestionnaire de transaction stable pour fournir les deux services. Cette machine peut connaître des défaillances et doit être tolérante aux pannes.

4.5.1 Une approche basée sur l'accord

Nous prévoyons un mécanisme de réplication où des copies sont définies pour assurer les tâches du *TM* en cas en panne. Une cohérence totale doit être maintenue entre les différentes copies du *TM*. Pour cela, chaque réplique doit être informée de toutes les actions effectuées par le *TM* afin de pouvoir reprendre l'exécution à partir d'un état cohérent, en cas de panne.
Chaque acte du *TM* est une décision qui implique toutes les copies. Plus précisément, la liste des places sources à contrôler, le choix de l'itinéraire et son évaluation nécessitent un accord entre tous les *TMs* potentiels. Cela se fait en appelant de façon répétée un service de *consensus* (voir figure 4.15).

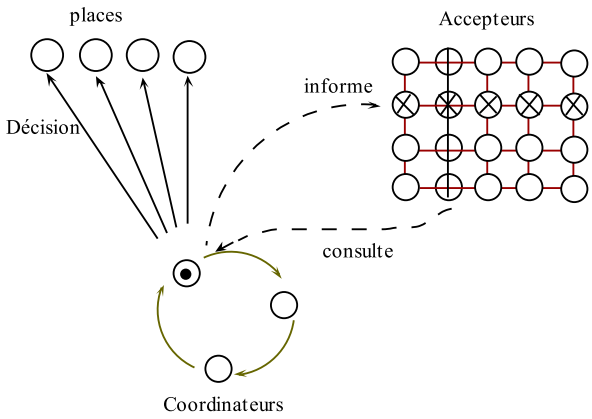

FIGURE 4.15 – Coordination des gestionnaires de transactions

Résoudre efficacement des problèmes d'accord tel que le *Consensus* est un défi important. Ce problème dont la spécification est relativement simple a fait l'objet de nombreux travaux de recherche. Durant une instance de consensus (identifiée par un entier c), des valeurs initiales (potentiellement distinctes) peuvent être proposées par un ou plusieurs nœuds appelés des auteurs de propositions. Chaque valeur est communiquée par son auteur à un sous ensemble unique de n nœuds parfaitement identifiés au sein du système global.
Ces nœuds particuliers, appelés les *acteurs*, sont en charge d'exécuter le protocole de consensus. Les n acteurs ont pour mission de générer une séquence de décisions et d'en assurer la persistance afin que tout nœud intéressé puisse prendre connaissance des valeurs déjà décidées. Les nœuds intéressés sont appelés les *apprenants*. Un nœud qui joue le rôle d'auteur durant une instance de consensus joue très souvent le rôle d'apprenant durant cette même instance. En général, un nœud est pressenti pour jouer qu'un seul de ces deux rôles.
Les nombres d'auteurs et d'apprenants ne sont pas nécessairement bornés. Auteurs et apprenants peuvent être externes aux sous-ensembles des acteurs ou membres de ceux-ci. Durant une instance de consensus c, les n acteurs coopèrent entre eux afin de déterminer la valeur qui sera adoptée comme nouvelle valeur de décision. Par définition, le protocole de consensus exécuté par les acteurs doit permettre la convergence inéluctable (propriété de terminaison) vers une valeur de décision unique (propriété d'accord uniforme) qui doit impérativement être l'une des valeurs proposées (propriété de validité). La valeur de décision est alors retournée vers tous les apprenants qui se sont fait connaître.
Dans un environnement réparti asynchrone où au moins un processus peut être défaillant, il a été prouvé qu'aucun protocole déterministe ne peut résoudre ce problème. Néanmoins, ce résultat d'impossibilité peut être contourné en s'appuyant sur un service (détecteurs de défaillances, élection de leader,...) qui sera utilisé comme oracle par le protocole. De nombreux protocoles indulgents ont été proposés : ils ne remettent jamais en cause les deux propriétés de sûreté qui caractérisent le problème du consensus (accord et validité). Par contre, dans ces protocoles, la propriété de vivacité (i.e., la propriété de terminaison) n'est satisfaite que si l'oracle utilisé offre inéluctablement un niveau minimum de qualité de service.
Dans notre solution nous nous intéressons aux protocoles ayant recours à un service

d'élection de leader. En théorie, l'oracle doit être un service d'élection de leader ultime : il existe un instant à partir duquel un acteur correct est considéré par tous comme étant le seul leader au sein du groupe d'acteurs. En pratique, le service d'élection de leader doit identifier un acteur correct et le désigner comme étant le seul leader possible durant un laps de temps suffisant pour que le protocole de consensus puisse converger vers une valeur de décision. Durant cette période, à chaque consultation de l'oracle, c'est l'identité de ce leader unique et correct qui est retournée.
Le protocole Paxos est le plus connu des protocoles de consensus s'appuyant sur un service d'élection de leader. Pour notre solution, nous utilisons un protocole de la famille Paxos appelé Paxos-MIC (voir chapitre 3).

Nous définissons un service d'accord (*AgreementService*) noté AG. Ce service implémente le protocole Paxos-MIC pour construire une séquence de décisions sur laquelle les deux services AS et AC s'appuient. Ces deux services communiquent avec le service AG à travers un niveau intermédiaire appelé *Proposer Learner Dispatcher* noté **PLD**. Un PLD, un AC et un AS sont implémentés au niveau de chaque coordinateur. Ces modules interagissent localement mais ne communiquent pas avec les nœuds distants. Par définition seulement les modules du leader sont activés. L'architecture utilisée est présentée dans la figure 4.16.
Les messages échangés entre le module AG et les services AS/AC passent à travers le PLD. Chaque appel de la fonction *ProposePull* initié par le module AG est d'abord reçu par le PLD du leader. La dernière valeur décidée est jointe à cet appel et est fournie uniquement au PLD du Leader qui agit en tant que *Learner* unique. Cette fonction est utilisée pour fournir la dernière valeur décidée et pour demander une nouvelle proposition qui sera utilisée dans la prochaine instance de consensus.

A travers ses deux services (AS et AC), le TM contrôle un ensemble de places sources dont la cardinalité augmente de 1 à chaque fois qu'une nouvelle place source est contrôlée (service AS), il choisit un itinéraire à tester, collecte le vote puis décide de la transaction (service AC). Le TM peut tomber en panne à n'importe quelle étape durant le contrôle de la source ou la validation atomique et donc chaque action effectuée par le TM est une décision d'accord entre toutes ses répliques. Plusieurs instances de consensus sont exécutées et plusieurs décisions sont prises avant de décider de la transaction globale (Commit/Abort). Ces décisions appelées *small decisions* sont : la liste des places sources à contrôler, la liste des places à tester (l'itinéraire), et enfin l'ensemble des votes collectés.
Afin de limiter l'utilisation répétée du service AG, chaque (proposition) décision est une concaténation de plusieurs petites décisions (*small decisions*), où chaque *small decisions* fait référence à une seule transaction. Dans ce cas plusieurs transactions peuvent être traitées durant une instance de consensus simultanément.
Une *small decision* (*small proposition*) liée à une transaction T_α, contient l'identificateur de la transaction et le type de service actuellement fourni pour T_α. Cette structure est nécessaire uniquement au niveau du PLD et les services AS/AC. Le service AG n'a pas besoin de connaitre la structure de la valeur qu'il décide.
Lorsque le PLD reçoit *ProposePull*, il vérifie d'abord s'il connait la séquence complète des décisions précédentes. Ceci n'est pas nécessaire si le coordinateur est devenu leader récemment. Si ces décisions ne sont pas connues par le leader, il appelle la fonction *RetreiveDec* par le PLD au service AG. Chaque décision reçue par le PLD est divisée en

plusieurs *smalldecisions* pour être dirigées vers le service correspondant, et ce, en invoquant la fonction *DecidePush*. Dans une seconde étape, l'appel de *ProposePull* est transféré vers les deux services AS et AC. La valeur retournée passe aussi par le PLD qui représente l'auteur(*proposer*) pour le module AG.

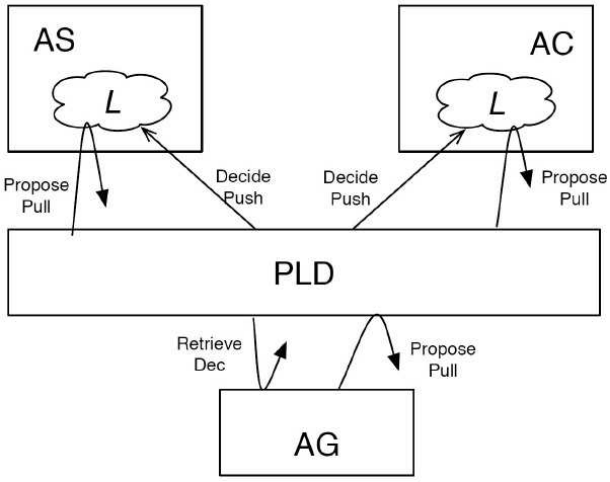

FIGURE 4.16 – Architecture des services basés sur l'accord

Si nous considérons toutes les *small decisions* liées à la transaction T_α, cette séquence est composée de deux parties. Le préfixe contient au moins une *small decision* relative au service AS. Chacune des ces *small decisions* correspond a un ensemble de places sources dont la cardinalité augmente de 1 à chaque décision prise.
La dernière partie est une séquence de décisions concernant le service AC. Une fois un itinéraire est identifié, le service AC procède à la construction et à l'évaluation des itinéraires. Deux types de décisions sont utilisées. D'abord AC décide sur l'itinéraire à évaluer : une *small decision* est la liste des places à tester. Ensuite le service AC décide de valider ou d'annuler la transaction selon les votes collectés : une *small decision* consiste en la liste des votes fournis par les places de l'itinéraire évalué. Le service AC peut terminer (*Commit/Abort*) la transaction ou peut attendre de nouvelles informations pour construire et évaluer de nouveaux itinéraires. Cette information est contenue dans un champ appelé *Data* et consiste en un ensemble de places ou un ensemble de votes. Dans chaque *small decision* un champ *StartTime* est inclus afin de contourner le problème du Δ. Dans le cas où l'itinéraire testé n'est pas validé, *Delta* représente le temps d'attente du TM pour recevoir de nouveaux itinéraires avant de décider d'annuler la transaction.

4.5.2 Implémentation des services d'accord

Une requête envoyée par un client ou un agent pour activer un des deux services (*Req_AS* ou *Req_AC*) est destinée à un *acceptor* disponible qui se charge de transférer le message (*Monitor* ou *Path*) à son leader. Les requêtes passent par les *accpetors* car ils sont les seuls à connaître l'identité du leader (pas forcement le bon). Comme nous supposons $2f + 1$ acceptors et $f + 1$ *coordinateurs*, la probabilité de contacter un *acceptor* correct est supérieure à celle de trouver un coordinateur disponible. Nous supposons qu'un agent peut avoir l'information dont il a besoin pour contacter plusieurs *acceptors* jusqu'à en trouver un correct.

Selon les messages reçus, chaque *acceptor* et/ou *coordinateur* construit une structure de données appelée *TransactionDirectory*, notée TD, qui permet de garder pour chaque transaction les informations utiles. Toutes les structures de données manipulées sont définies dans la figure 4.17. Lorsqu'un *Coordinateur / Acceptor* prend connaissance d'une nouvelle transaction, il crée une nouvelle entrée dans la TD qui sera modifiée durant l'exécution de la transaction. Cette nouvelle entrée est créée en invoquant la fonction *Put* durant l'exécution des tâches $Task_{AS}A1$, $Task_{AS}C1$, $Task_{AC}A1$ and $Task_{AC}C1$.

```
% Transaction Descriptor :Desc ← (Stages, Code, Places0, Δ)
% Tag : has replied Yes, has replied No, not tested yet
% Last : true if the place belongs to the last tested itinerary
Tag ∈ Yes, No, NT; Place = (Id, Tag, Last) ;
% Knowledge is either a list of places or a list of lists of places
% Type of Service or Type of the next decision
Type ∈ AS, AC1, AC2; Data ∈ "listofplaces", "listofvotes"
dec ← (Type, Tid, StartTime, Data)
% Transactions Directory : keep useful information for each transaction
TD ← [(Tid, Desc, Type, Knowledge, Result, StartTime)] ;
TD ← ∅; Val ← [dec]; LastCon ← 0 ; % Last consensus number
```

FIGURE 4.17 – Structure de données

```
% A client sends a request to initiate the execution of a transaction to an acceptor
Task_AS A1 : When A_i receives msg Req_AS(Desc, Tid) from Client
(1)  if(Get(Req_AS.Tid, _, _, _, _, _) = false)then
(2)       Put(Req_AS.Tid, Req_AS.Desc, AS, , , );
(3)       send Monitor(Req_AS.Desc, Req_AS.Tid) to C_Lid;

Task_AS C1 : When C_i receives msg Monitor(Desc, Tid) from A_j
(4)  if(Get(Monitor.Tid, _, _, _, _, _) = false) then
(5)       Put(Monitor.Tid, Monitor.Desc, AS, _, _, _) ;
```

FIGURE 4.18 – AS service : Contrôle de la place source.

Toutes les tâches[2] citées sont exécutées par un *acceptor* quand il reçoit les requêtes à partir des clients ou des agents et par le coordinateur quand il reçoit un message à partir d'un *acceptor*.

Pour chaque transaction T_α, l'information sauvegardée dans la TD consiste en plusieurs champs. L'identificateur de la transaction noté *Tid* et le descripteur de la transaction *Desc* pour sauvegarder les informations statiques concernant la transaction T_α. Le service actuellement fourni (AS ou AC) est indiqué dans le champ *Type*. Lorsque le service fourni est AC, ce champ indique aussi le type de la valeur de la prochaine proposition que doit

2. La notation AS/AC identifie les services, par contre Ax/Cx est utilisée pour distinguer les tâches du coordinteur et celles de l'acceptor

fournir le service AC durant l'appel à la fonction *ProposePull*. Cette proposition peut contenir la liste des places à évaluer ($Type = AC1$) ou la liste des votes ($Type = AC2$). Toute information utile concernant la transaction est sauvegardée dans un champ spécial appelé *Knowledge*. Dans le cas du service AS, ce champ contient la liste des places sources actuellement contrôlées. Lorsqu'il s'agit du service AC, *Knowledge* est implémenté sous forme d'une liste de listes de places : pour chaque étape d'exécution i, $Knowledge[i]$ représente la liste des nœuds alternatifs où l'étape i a été exécutée par des agents mobiles différents. Chaque *acceptor* envoie périodiquement son *Knowledge* courant à son leader (figure 4.19$TaskA2$) via le message *Monitor* ou *Path*. Les coordinateurs et les acceptors adoptent le même comportement : ils augmentent leurs *Knowledge* avec les nouvelles places alternatives à chaque fois qu'un nouvel itinéraire est reçu par l'acceptor dans le message Req_AC (figure 4.19,$Task_{AC}A1$) et par le coordinateur dans les messages *Path* envoyés par les acceptors(figure 4.20,$Task_{AC}C1$).

$Task_{AC}A1$: When A_i receives msg $Req\_AC(Tid, State, Itinerary)$ from $Agent_k$
(1) if $(Get(Req\_AC.Tid, \_, Type, K, \_, \_) = false)$ then
(2) IncreaseKnowledge $(K, Req\_AC.Itinerary)$;
(3) $Put(Req\_AC.Tid, \_, AC1, K, \_, \_)$;
(4) else if (Type = AS) then $K \leftarrow \emptyset$; Type $\leftarrow$ AC1 ;
(5) IncreaseKnowledge$(K, Req_AC.Itinerary)$;
(6) Update$(Req_AC.Tid, _, Type, K, _, _)$;
(7) send Path$(Req_AC.Tid, K)$ to C_{Lid} ;

TaskA2 : Periodically
(8) for each $(i < TD.size())$ do
(9) if $(TD[i].Type = AS)$ then
(10) send Monitor$(TD[i].Tid, TD[i].Knowledge)$ to C_{Lid} ;
(11) else
(12) send $Path(TD[i].Tid, TD[i].Knowledge)$ to C_{Lid} ;

FIGURE 4.19 – A_i : Valider le meilleur chemin.

$Task_{AC}C1$: When C_i receives msg $Path(Tid, K)$ from A_j
(1) if $(Get(Path.Tid, \_, Type, K, \_, \_) = false)$ then
(2) $Put(Path.Tid, \_, AC1, Path.K, \_, \_)$;
(3) else if $(Result \neq \perp)$ then
(4) if (Type = AS) then K $\leftarrow \emptyset$; Type $\leftarrow$ AC1 ;
(5) IncreaseKnowledge(K, Path.K)
(6) Update$(Path.Tid, _, Type, K, _, _)$;

FIGURE 4.20 – C_i : Valider le meilleur chemin.

La fonction $IncreaseKnowledge()$ étend le *Knowledge* courant avec la nouvelle informa-

tion reçue. La décision de validation concernant la transaction est stockée dans le champ *Result*.
Différentes places dans le *Knowledge* sont testées pour construire le meilleur chemin. Tester une place p_i pour une transaction T_α consiste à obtenir un vote (Yes/No) de la part de la place p_i concernant T_α. Un coordinateur doit pouvoir distinguer entre une place qui n'a jamais été testée et une place qui a déjà voté et quel a été son vote ? (*Yes* ou *No*). Le coordinateur marque les places testées via un *Tag*. Un booléen *Last* est utilisé pour indiquer si cette place a fait partie du dernier chemin testé.
Chaque module *PLD* maintient l'information concernant les valeurs de décision envoyées par le niveau *AG* telle que la dernière instance de consensus terminée, *LastCon*.
L'exécution de tâche $Task_{PLD}C1$ dans la figure 4.21 est faite uniquement par le *PLD* du leader courant. Elle est déclenchée suite à l'invocation de la fonction *ProposePull*. Le leader est invité à proposer une valeur pour l'instance de consensus courante désignée par la variable *Con*. Cette fonction fournit au leader la dernière valeur de décision connue (variable *DVal*). Afin de construire une nouvelle proposition, le leader doit acquérir l'information complète du processus de validation du stade courant. Donc, il doit d'abord récupérer toutes les décisions manquantes relatives aux instances de consensus i, avec $LastCon < i < Con - 1$.
Les valeurs de décisions précédentes sont acheminées à travers le *PLD* vers le service *AS* ou *AC* selon le type du service de la décision. Pour chaque transaction T_α, le leader va exécuter l'une des deux tâches $Task_{AC}C2$ ou $Task_{AS}C2$ décrites dans la figure4.22. La décision récupérée est utilisée par le leader pour mettre à jour son *Knowledge*.

$Task_{PLD}C1$: Upon invocation of ProposePull(Con, DVal) returns PVal
(1) for each $(i = LastCon + 1; i < (Con - 1); i++)$ do
(2) $Val \leftarrow RetrieveDec(i)$;
(3) for each $(dec \in Val)$ do
(4) if $(dec.Type = AS)$ then $DecidePush_{AS}(dec)$;
(5) else $DecidePush_{AC}(dec)$;
(6) if $(DVal.Type = AS)$ then $DecidePush_{AS}(DVal)$;
(7) else DecidePushAC(DVal) ;
(8) LastCon $\leftarrow Con - 1$;
(9) $PVal_1 \leftarrow ProposePull_{AS}()$; $PVal_2 \leftarrow ProposePull_{AC}()$;
(10) PVal $\leftarrow PVal_1 \cup PVal_2$;
(11) return PVal ;

FIGURE 4.21 – Relayer ProposePull.

```
Task_AC C2 : Upon invocation of DecidePush AC(dec)
(1)  Get(dec.Tid, Desc, Type, K, Res, ) ;
(2)  if (Res ≠ ⊥) then STOP ;
(3)  if (dec.Type = AC2) then
(4)      if (dec.StartTime ≠ 0) then
(5)          Update(dec.Tid, _, _, _, _, dec.StartTime) ;
(6)      Res ← CheckList(dec.Data) ;
(7)      if ((Res = Commit) ∧ (dec.Data.size() = Desc.Stages)) then
(8)          Update(dec.Tid, _, _, _, Res, _) ;
(9)      else FilterKnowledge(dec.Data, K) ;
(10)         Update(dec.Tid, _, AC1, K, _, _) ;
(11) else IncreaseKnowledge(K, dec.Data); SetLast(dec.Data, K) ;
(12)     Update(Dec.Tid, _, AC2, K, _, _) ;

Task_AS C2 : Upon invocation of DecidePush_AS(dec)
(13) Get(dec.Tid, Desc, _, K, _, _) ;
(14) IncreaseKnowledge(K, dec.Data) ;
(15) Update(dec.Tid, _, _, K, _, _) ;
```

FIGURE 4.22 – Usage de la décision

Pour le service AS, le leader ajoute les nouvelles places sources contrôlées à son *Knowledge*. Concernant le service AC, le leader vérifie le type d'information contenue dans cette décision. Si la décision contient une liste de votes, le leader calcule le résultat de la transaction de l'itinéraire courant. Si le résultat retourné par la fonction *CheckList* est Commit, le Processus de validation est terminé et le client est informé du résultat de la transaction. Dans le cas contraire, le leader utilise les votes pour filtrer son *Knowledge*.
Le mécanisme de filtrage met à jour le *tag* des places récemment testées dans le *Knowledge*, selon le vote fourni par chacune d'entre elles.
Si le leader trouve une liste de places, cette décision indique le dernier chemin testé pour la transaction T_α. Le leader ajoute les nouvelles places à son *knowledge* puis et après l'exploitation de toutes les informations collectées, il sera capable de bâtir une nouvelle proposition. Pour chaque transaction T_α, le leader invoque la fonction $NewProposal_{AS}$ ou la fonction $NewProposal_{AC}$ selon le service fourni à la transaction (voir figure 4.23).

```
Task_AC C3 :Upon invocation of ProposePull_AC() returns PVal
(1)  PVal ← ∅;
(2)  for each (Tid ∈ TD) do
(3)      Get(Tid, Desc, Type, K, Res, ST);
(4)      if ((Res ≠⊥) ∧ (Type ≠ AS)) then
(5)          PVal ← PVal ∪ NewProposal_AC(Type, Tid, Desc, K, ST);
(6)  return PVal;

Task_AS C3 : Upon invocation of ProposePull_AS() returns PVal (7)  PVal ← ∅;
(8)  for each (Tid ∈ TD) do
(9)      Get(Tid, Desc, Type, _, Res, _);
(10)     if((Res ≠⊥) ∧ (Type = AS))then
(11)         PVal ← PVal ∪ NewProposalAS(Tid, K, Desc);
(12) return PVal;
```

FIGURE 4.23 – Proposition d'une valeur au niveau des modules des services AC et AS

La fonction $NewProposal_{AS}$ propose une nouvelle valeur seulement si toutes les places sources sont suspectées (une nouvelle place source va être activée). Dans le cas du service AC le champ *type* indique le type de la prochaine valeur à proposer ; Deux cas sont possibles : 1) Si la dernière décision observée était une liste de votes obtenue par un itinéraire qui n'a pas pu valider (commit) la transaction, le leader essaie de construire un nouvel itinéraire (la fonction *GetNewPath*). Durant sa recherche, le leader va explorer le *knowledge* correspondant à la transaction T_α, dans le but de trouver des places alternatives utiles et compatibles (la fonction *FindAlternative*). Une place est utile si elle n'est pas encore testée ou a répondu par un '*Yes*' à la requête de vote. Si le leader ne trouve pas de places utiles, il attend (si le délai spécifié par l'utilisateur n'est pas écoulé) l'arrivée des agents avec de nouveaux itinéraires. Si le délai (timeout) expire, le leader abandonne la transaction $(Abort)$.
2) Si la dernière décision observée retalive à T_α était un itinéraire, la nouvelle proposition doit contenir la liste des votes correspondants aux places de cet itinéraire. En invoquant la fonction *GatherVotes*, le leader obtient un vote de chaque place de cet itinéraire. Après la construction de la liste des votes, le leader déclenche le délai d'attente spécifié par le client en fixant le *StartTime* au temps courant. Le client spécifie à travers ce délai, la durée d'attente pour la validation de sa transaction si le premier chemin n'est pas validé. Donc, si le premier itinéraire testé n'est pas validé, le *StartTime* de la transaction est inclus dans la nouvelle valeur de proposition qui deviendra la prochaine décision. De cette façon, n'importe quel nouveau leader aura la valeur du *StartTime* en récupérant les anciennes valeurs décidées.

Function $IncreaseKnowledge(Knowledge, List)$
% List can be a list of places or a list of lists
(1) for each $(mj \in List)$ do
(2) $Knowledge[j] \leftarrow Knowledge[j] \cup ([mj])$;

Function $NewProposal_{AC}$(Type, Tid, Desc, K, ST) **returns** PVal
(3) $PVal \leftarrow \perp$;
(4) if $(Type = AC1)$ then
(5) $List \leftarrow GetNewPath(K)$;
(6) if $(List \neq \perp)$ then PVal $\leftarrow (AC1, Tid, List)$;
(7) else if $((ST + Desc.\Delta) < CurrentTime)$ then
(8) Update$(Tid, _, _, K, Abort, _)$;
(9) else $List \leftarrow FindLastTested(K)$;
(10) $Votes \leftarrow GatherVotes(List, Tid)$;
(11) if$(ST = 0)$ then $ST \leftarrow CurrentTime$;
(12) $PVal \leftarrow (AC2, Tid, ST, Votes)$;
(13) Update$(Tid, _, _, _, _, ST)$;
(14) return PVal ;

Function $NewProposal_{AS}(Tid, K, Desc)$ returns PVal
(15) $PVal \leftarrow \perp$;
(16) if $(Alive(K) = False)$ then
(17) $p \leftarrow NewPlace(Desc.Places_0)$;
(18) if $(p = \perp)$ then send $Req\_AC(Tid, Stop, \perp)$ to Aj ;
(19) else $K \leftarrow K \cup [p]$; $PVal \leftarrow (AS, Tid, K)$;
(20) return PVal ;

Function $FilterKnowledge(votes, K)$
(21) for each $(i < K.size())$ do
(22) for each $(p \in K[i])$ do
(23) if $(p.Last = true)$ then $p.Tag \leftarrow votes[i])$; break ;

FIGURE 4.24 – Les fonctions utilisées (partie 1)

```
Function GetNewPath(K) returns Places or ⊥
(1)  Places ←⊥ ;
(2)  for each (i < K.size()) do
(3)      Places ← Places ∪ FindAlternative(K[i]) ;
(4)  return Places ;

Function SetLast(places, K)
(5)  ResetLast(K, false) ;
(6)  for each (i < K.size()) do
(7)      for each (p ∈ K[i]) do
(8)          if (p.Id = places[i])) then p.Last ← true ; break ;

Function CheckList(List) returns Abort ∨ Commit
(9)  for each (v ∈ List) do if (v = NO) then return Abort ;
(10) return Commit ;

Function GatherVotes(List, Tid) returns Votes
(11) for each (pi ∈ List) do
(12)     vote ← AskPlace(pi, Tid); Votes ← Votes∪ {vote} ;
(13) return Votes ;

Function FindAlternative(List) returns altor ⊥
(14) alt ←⊥ ;
(15) for each (p ∈ List) do
(16)     if (p.Tag = Yes) then return p ;
(17)     else if (p.Tag = NT) then alt ← p ;
(18) return alt ;

Function FindLastTested(K) returns List
(19) List ←⊥ ;
(20) for each (i < K.size()) do
(21)     for each (p ∈ K[i]) do
(22)         if (p.Last = true) then List ← List ∪ [p] ; break ;
(23) return List ;
```

FIGURE 4.25 – Les fonctions utilisées (partie 2)

4.6 Synthèse

La mise en évidence de l'apport de la solution proposée émerge de sa comparaison avec le tableau 2.1 lui même issu d'une comparaison des solutions proposées antérieurement (cf. tableau4.1).

Propriété	Panne tolérée						Caractéristiques
	agent	place	lien	Sémantique			
				1	2	3	
[2]	✓	✓	✓	✓	X	X	bloquante
[46]	X	X	X	✓	✓	X	bloquante (résout seulement les pannes sémantiques)
[40]	✓	✓	✓	X	X	✓	bloquante
[47]	✓	✓	X	✓	X	X	la panne de médiateurs est bloquante
[41]	✓	✓	✓	✓	X	X	non bloquante mais pas adaptée pour les transactions longues
[48]	✓	✓	X	X	X	✓	panne des liens est bloquante
solution proposée	✓	✓	✓	✓	✓	✓	non bloquante et bien adaptée pour les transactions longues

TABLE 4.1 – Comparaison de la solution proposée avec les approches existantes. ✓ : la panne est tolérée, X : la panne n'est pas tolérée.

4.7 Conclusion

Ce chapitre avait pour objectif la présentation de la solution que nous avons préconisée pour obtenir un fonctionnement quasi-permanent de l'agent mobile.
Nous avons commencé par la présentation du modèle du système d'agents ainsi que de modèle transactionnel ayant servi de base pour l'élaboration du protocole proposé. Ensuite, la solution de tolérance aux pannes a été détaillée, nous conduisant à adopter un protocole de consensus. Dans cette section, nous avons décrit et expliqué l'intégration du protocole paxos multiple intégré (Paxos MIC) dans la solution à l'effet de maintenir les propriétés à vérifier par un agent mobile transactionnel tolérant aux pannes.

Chapitre 5

Implémentation et Performance

Afin de mieux exploiter les agents mobiles, nous avons proposé un protocole de tolérance aux pannes permettant la détection et le recouvrement des agents en pannes. Nous avons augmenté le modèle agents mobiles avec le support transactionnel. Un agent mobile est créé pour chaque transaction afin d'effectuer un ensemble de requêtes demandées par le client. L'agent mobile doit s'exécuter de façon non bloquante, unique et atomique.
Dans ce chapitre, nous allons mesurer la performance de la solution proposée, implémentée et testée dans un environnement réel.

5.1 Environnement de développement

Le protocole de tolérance aux pannes est implémenté sur la plateforme JADE[14]. C'est une plateforme gratuite,open source, dédiée au développement des systèmes à base d'agents. Développée au sein des laboratoires "Telecom Italia", cette plateforme facilite le développement des systèmes à base d'agents répondant aux spécifications de la norme FIPA[9]. Ce choix est principalement justifié par le fait que c'est une plateforme multi-agents développée en Java, dotée d'une interface graphique et peut être répartie sur plusieurs serveurs. Ses principales caractéristiques sont :

- Les agents développés sous la plateforme Jade, sont entièrement écrits en Java. Le choix de la plateforme s'est donc imposé comme étant une conséquence de notre choix en termes de langage de programmation Java.
- Jade simplifie l'implémentation des systèmes d'agents à travers un middleware répondant aux spécifications de la FIPA, une librairie de classes que les utilisateurs peuvent utiliser et étendre, ainsi qu'un ensemble d'outils graphiques qui permettent le débogage et l'administration du système d'agents.
- Jade assure une communication transparente par échange de messages dans le langage normalisé FIPA-ACL, et offre un support de mobilité pour les agents.
- Jade gère le cycle de vie des agents depuis leur création jusqu'à leur destruction.

5.2 Environnement de test

Nous avons utilisé un réseau local composé de six machines identiques (Pentium IV 3.00GHz, 1Go of RAM) reliées par un réseau de débit $100Mbps$.
Notre application tourne dans un environnement dynamique et incertain. L'incertitude dépend de deux aspects : l'incertitude sur les nœuds participant à l'application et l'incertitude sur les interactions entre les différents nœuds ainsi que le débit des liens de communication. Le protocole de validation atomique basé sur Paxos-MIC, a été mis en œuvre en langage Java. Les expérimentations ont été conduites sur la plateforme expérimentale Grid 5000 qui comporte plus de vingt clusters répartis sur dix sites en France. Dans un cluster, les nœuds de calcul sont homogènes (même processeur) mais les processeurs peuvent différer d'un cluster à un autre. Tous les nœuds de calcul exécutent le système Debian. Nous utilisons la machine virtuelle OpenJDK6 pour réaliser nos mesures sur les nœuds de la grille.

5.3 Intégration du protocole dans le code de l'agent mobile

Jade permet de créer des agents mobiles capables de migrer vers plusieurs sites répartis dans le réseau. Ainsi, pour qu'un agent migre vers une machine distante il lui suffit d'appeler la méthode doMove(), le code et l'état de l'agent sont envoyés vers la machine destinataire. Mais comme JAVA ne permet pas la mobilité forte ; l'agent mobile, après sa migration, réexécute le code depuis le début.
Jade définit deux autres méthodes *beforeMove* et *afterMove*. Ces deux méthodes sont appelées automatiquement par l'agent mobile quand la méthode *doMove* est invoquée (l'appel à ces deux méthodes est transparent au programmeur).
Nous avons implémenté le protocole de tolérance aux pannes dans ces deux méthodes :

- La méthode $beforeMove()$: Cette méthode est exécutée par l'agent mobile juste avant sa migration, nous avons donc redéfini cette méthode. Elle crée l'agent de contrôle et effectue la sauvegarde de l'agent sur la place courante (checkpoint).
- La méthode $afterMove()$: C'est la première méthode exécutée par l'agent quand il arrive à la place destinataire c'est à dire quand la migration se termine avec succès. Nous avons redéfini cette méthode pour accéder au registre LD afin d'enregistrer l'agent mobile et vérifier si la requête courante n'a pas été exécutée sur cette place auparavant. Ceci permet de détecter les doubles exécutions

5.4 Résultats de l'expérimentation

Dans la première partie, le but étant de mesurer le temps requis par un agent mobile pour s'exécuter sur un itinéraire en augmentant à chaque fois le nombre de places. Cette expérimentation est effectuée en utilisant d'abord un agent simple puis un agent tolérant aux pannes (implémentant le protocole proposé).
Nous avons utilisé une application simple qui consiste à incrémenter un compteur sur un ensemble de machines. Nous avons mesuré le temps écoulé entre l'envoi et la réception de l'agent mobile.
Les résultats obtenus sont résumés dans le tableau 5.1 et sont présentés par la figure5.1 où l'axe des x représente l'évolution du nombre de places et l'axe des y représente le délai

moyen d'exécution de l'agent (en ms).

Type d'agent	Taille (ko)	Temps d'exécution(ms)				
		2 stades	3 stades	4 stades	5 stades	6 stades
Agent Simple	10 ko	654	938	1194	1441	1710
Agent tolérant aux pannes(FTagent)	30 ko	1047	1498	2041	2491	3085

TABLE 5.1 – Temps d'exécution en ms

Nous pouvons constater que l'agent tolérant aux pannes introduit un temps de latence de 60% par rapport à un agent mobile simple sans l'intégration du mécanisme de tolérance aux pannes. Ce temps de latence est donc introduit par le protocole de tolérance aux pannes.

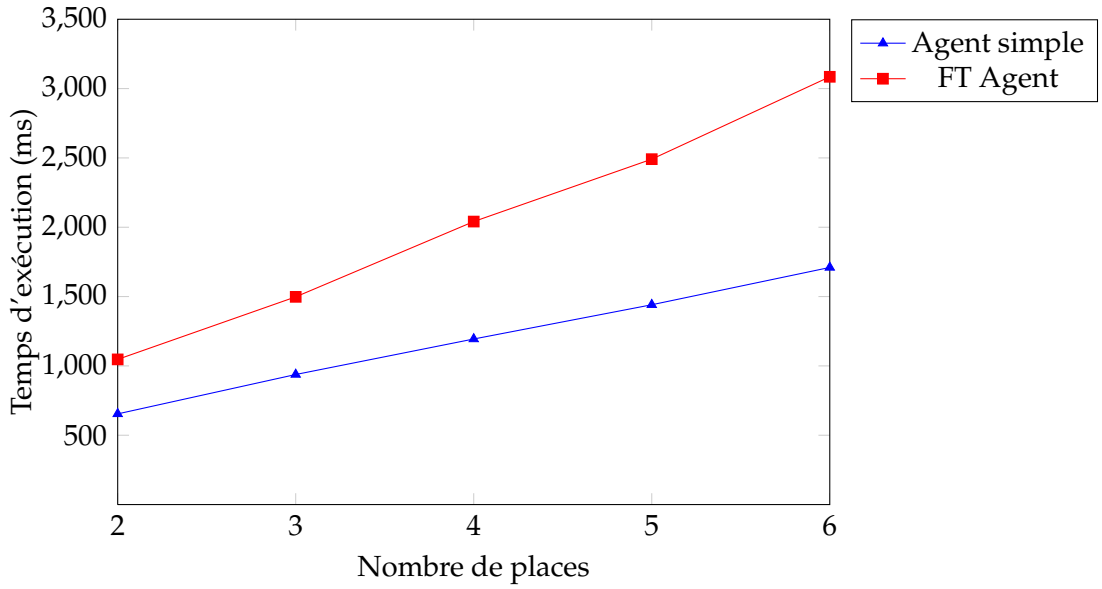

FIGURE 5.1 – Temps de latence d'un agent tolérant aux fautes (noté FT agent) par rapport à celui d'un agent simple

D'autre part, nous remarquons que la taille de l'agent tolérant aux pannes (30ko) est multiplié par trois par rapport à un agent simple (10ko). La taille de la classe de l'agent peut aussi influencer le temps d'exécution et plus précisément le temps de migration.
Pour confirmer cela, c'est à dire voir si la taille de l'agent influence le temps de migration, nous avons effectué le test suivant :
Nous avons créé un agent mobile simple (sans le protocole de tolérance aux pannes). Cet agent définit un tableau d'octets de taille variable. Nous augmentons à chaque fois la taille du tableau pour augmenter la taille de l'agent.
Nous avons mesuré le temps d'exécution de l'agent mobile en faisant varier sa taille. Les résultats sont présentés dans la figure 5.2.

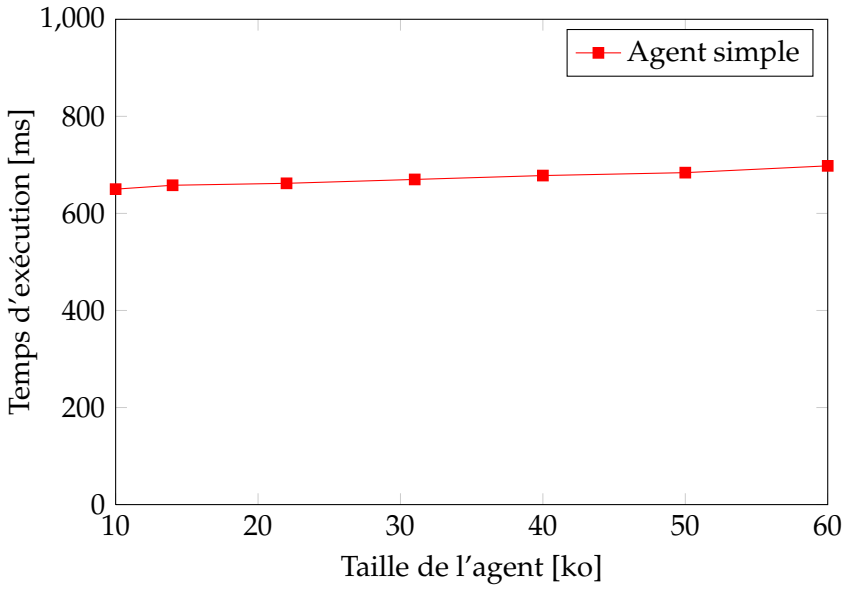

FIGURE 5.2 – Influence de la taille de l'agent sur le temps d'exécution

Nous remarquons que le temps d'exécution augmente avec la taille de l'agent mais d'une manière très négligeable par rapport au temps de latence introduit par l'agent tolérant aux pannes.
Nous pouvons conclure que dans notre cas, le temps de latence n'est pas causé par la taille de l'agent. En fait, l'écart du temps d'exécution provient des actions de l'agent tolérant aux pannes, effectuées avant et après chaque migration à savoir :

- L'enregistrement dans le répertoire de recherche *lookup − directory*. Implémenté sous forme d'un fichier XML sur chaque place. Ce fichier est accédé par l'agent mobile pour vérification et enregistrement.
- La création de l'agent de contrôle.
- Un second accès au répertoire de recherche pour mettre à jour l'état d'exécution de l'agent.
- La sauvegarde de l'état de l'agent et de son code sur la machine courante (checkpoint) juste avant sa migration

Afin de tester l'efficacité du protocole proposé, nous avons provoqué un ensemble de pannes et avons observé le comportement de l'agent. Nous avons mesuré le temps d'exécution de l'agent après recouvrement.

5.4.1 Panne de l'agent mobile

La plateforme Jade offre une interface graphique qui permet de visualiser les agents qui s'exécutent sur cette plateforme tout en pouvant créer d'autres agents ou en détruire.
Afin de simuler une panne par arrêt de l'agent mobile, nous avons lancé l'agent sur un réseau local puis nous avons détruit l'agent à l'aide de l'interface graphique de Jade. La panne est provoquée sur la place 5 (stade 5). Les résultats sont illustrés par le graphe de la figure 5.3.

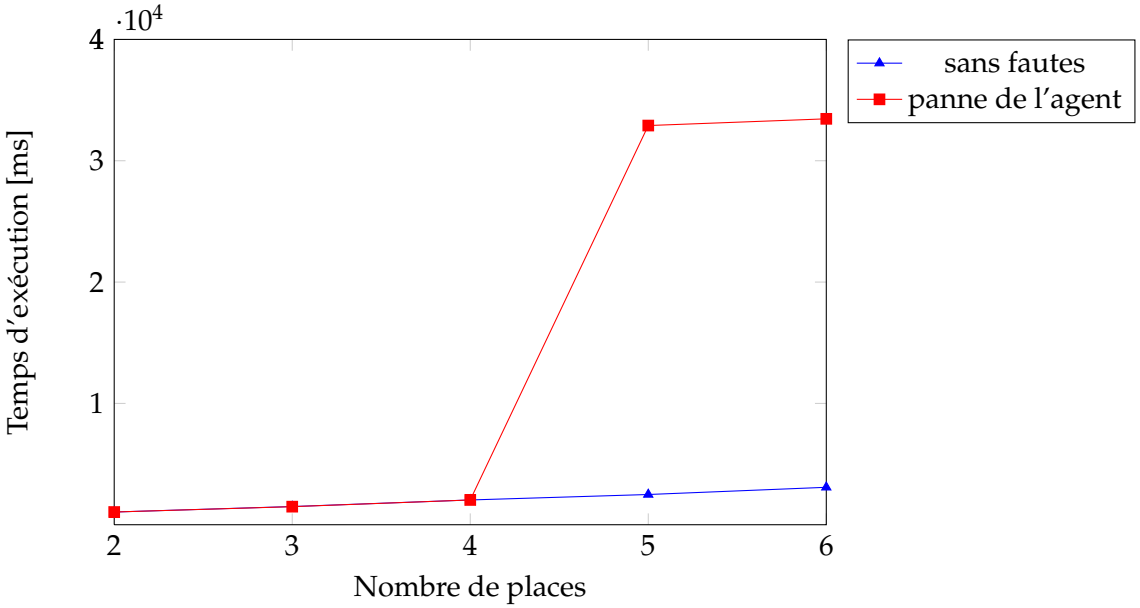

FIGURE 5.3 – Temps d'exécution de l'agent mobile dans le cas d'une panne.

L'agent mobile avait un comportement normal jusqu'au stade 5 au delà duquel le temps d'exécution augmente sensiblement. En fait, la panne de l'agent mobile est détectée par l'agent de contrôle sur la place 4. Et ce, après un délai d'attente limité (défini par 30000ms dans notre application). Juste après, l'agent de contrôle crée une nouvelle copie de l'agent mobile et le ré-envoie vers la même place. L'agent mobile ré-exécute le stade 5 puis continue l'exécution.

5.4.2 Panne de la place

La panne de place est tolérée par l'envoi de l'agent mobile vers une place alternative pour découvrir un nouveau chemin. Nous provoquons une panne de la place en arrêtant la plateforme Jade sur une machine du réseau. Dans ce cas, même si la machine est correcte l'agent mobile ainsi que la place sont simultanément perdus.
Dans cet exemple, la panne est survenue sur la quatrième place (stade 4). Le graphe des résultats est présenté dans la figure 5.4

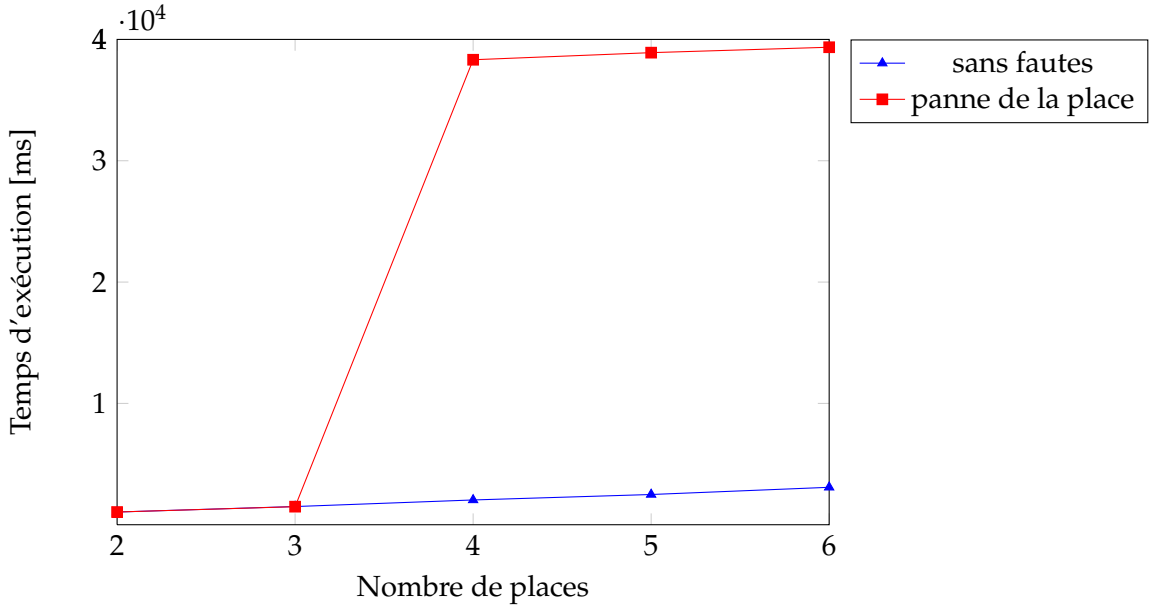

FIGURE 5.4 – Temps d'exécution de l'agent mobile dans le cas de la panne de la place.

Nous remarquons l'augmentation du temps d'exécution dès l'entame du stade défaillant. Ce temps de latence est causé par le délai d'attente écoulé avant la détection de la panne. De plus, le nouvel agent essaie de re-contacter la place suspectée et attend encore un temps suffisant (défini par 1000*ms*) avant d'aller chercher une place alternative.
Le stade 4 est ré-exécuté sur la place alternative (si elle existe) puis l'agent continue ses déplacements pour exécuter les stades restants.

5.4.3 Panne Sémantique

Nous avons créé des agents dédiés sur chaque place appelés *agentsservices*. Ces agents sont destinés à représenter le fournisseur de service. Il reçoivent les requêtes des agents mobiles, accèdent aux ressources locales pour exécuter la requête puis rendent la réponse aux agents.
Si la requête est satisfaite, l'agent mobile crée une table de pré-réservation et y sauvegarde le résultat. Par contre, si le service n'est pas disponible, l'agent *service* répond négativement à la demande de l'agent mobile. Il s'agit donc d'une faute sémantique.
Dans ce scénario de panne, nous avons simulé le cas d'une panne sémantique(le service demandé n'est pas délivré à l'agent mobile) pour voir l'impact sur l'exécution de l'agent. Nous avons fait en sorte que l'agent service de la place 4 rend une réponse négative à la demande de l'agent mobile. Le résultat de l'exécution est représenté dans la figure 5.5

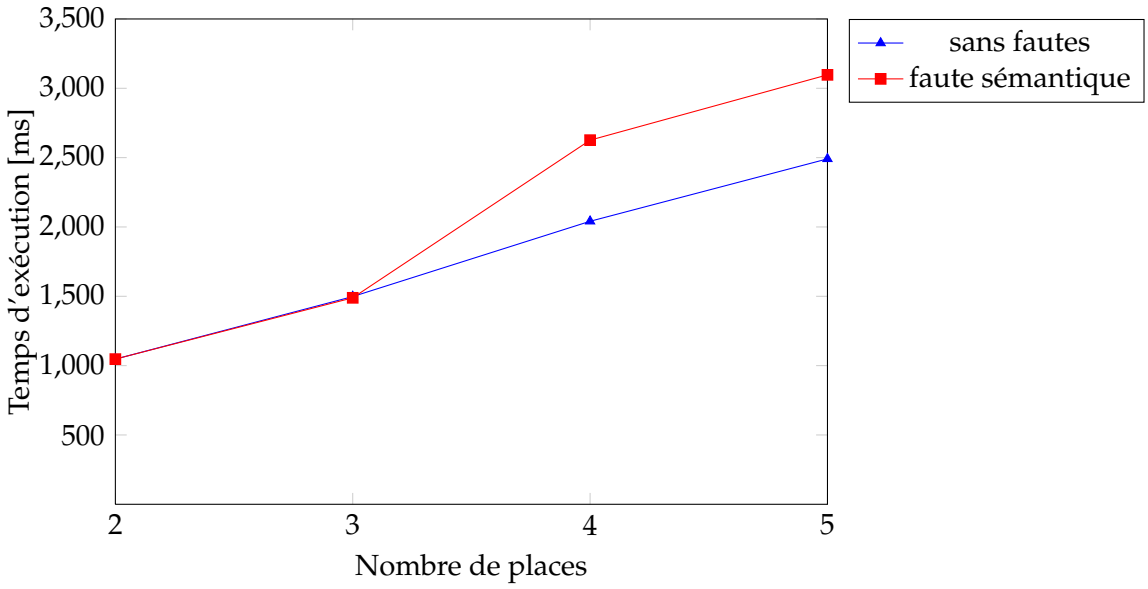

FIGURE 5.5 – Temps d'exécution de l'agent mobile dans le cas d'une panne sémantique.

Lorsque l'agent mobile reçoit une réponse négative il envoie un message *failed* à l'agent de veille. Ce dernier trouve une place alternative pour envoyer l'agent mobile. Le temps de latence introduit n'est pas très important (presque 500*ms*). Ce temps correspond aux temps de migration et d'exécution du stade 4, une seconde fois sur une nouvelle place.

5.4.4 Panne de l'agent de veille

Notre protocole de tolérance aux pannes est basé sur la chaine de dépendance, où chaque agent de veille surveille son successeur. Le dernier agent de la chaîne contrôle l'agent mobile. La surveillance se fait par l'envoi de messages périodiques appelés "battements de cœur". Si l'agent de contrôle ne reçoit aucun message après un délai d'attente limité il suspecte une panne.
Dans le cas présent, nous avons simulé une panne de l'agent de veille. A l'aide de l'interface Jade, nous avons détruit l'agent de veille créé par l'agent mobile sur la place 3 alors que l'agent mobile était déjà arrivé à la place 5. Les résultats sont présentés dans la figure 5.6.
Dans note application, nous avons défini le délai d'attente de l'agent de veille par 40000*ms*. Après ce temps d'attente, l'agent de veille considère son successeur sur la place suivante comme défaillant. Dans ce cas l'agent de veille crée une nouvelle copie de l'agent mobile. Avant d'aller vers un nœud alternatif, le nouvel agent essaie de migrer vers la place de l'agent de veille suspecté :

- S'il réussit à atteindre la place de l'agent suspecté, il essaie de recréer un nouvel agent de veille et s'autodétruit. Dans ce cas le temps d'exécution de l'agent n'est pas influencé par cette panne.
- S'il la place n'est plus accessible, il migre vers un nœud alternatif pour recommencer l'exécution sur un nouveau chemin. Le temps d'exécution est influencé par le délai d'attente puis le temps de ré-exécution des stades suivants.

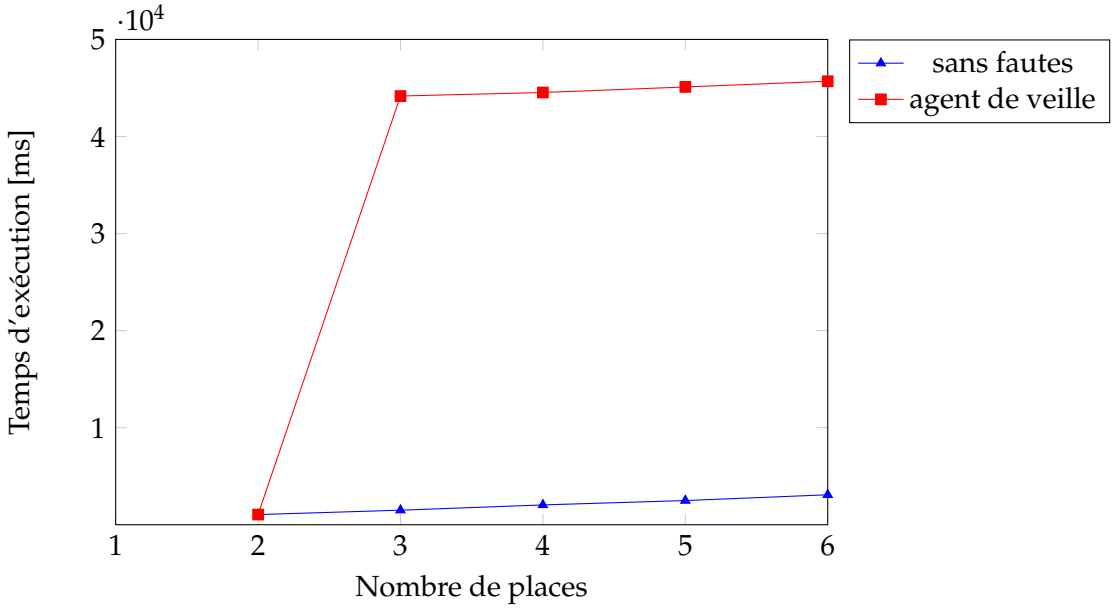

FIGURE 5.6 – Temps d'exécution dans le cas de la panne d'un agent de veille.

D'habitude, on parle de double exécution quand il y a une fausse détection de panne. Par contre ici, la double exécution apparaît même si la détection de panne est vraie.
En fait, la panne d'une place où réside un agent de veille risque de causer l'annulation de la transaction globale (si cette place n'est pas recouvrée avant la validation de la transaction). Dans le cas des transactions longues (le nombre de requêtes est grand), il est préférable de prévoir un chemin alternatif c'est à dire tolérer une double exécution pour augmenter les chances de validation. Sachant que la validation des requêtes au niveau des places se fait lorsque l'agent arrive à la destination. Donc une seule exécution est validée, les autres seront abandonnées.

5.4.5 Panne catastrophique

La figure 5.7 représente le cas d'une série de pannes en cascade des places. Chaque panne survient lorsque la place est en train de recouvrer un agent défaillant sur la place suivante. La possibilité qu'une telle panne survienne est minime. Le cas échéant, l'agent est toujours recouvré grâce à la chaîne de contrôle et le retour arrière.
Il est à noter que le temps d'exécution devient important par rapport au temps d'exécution de l'agent à l'état normal (sans pannes). Cela est justifié par les délais d'attente requis par les agents de veille pour la détection de la panne et le déclenchement de la procédure de recouvrement (y compris le temps de recherche des places alternatives).

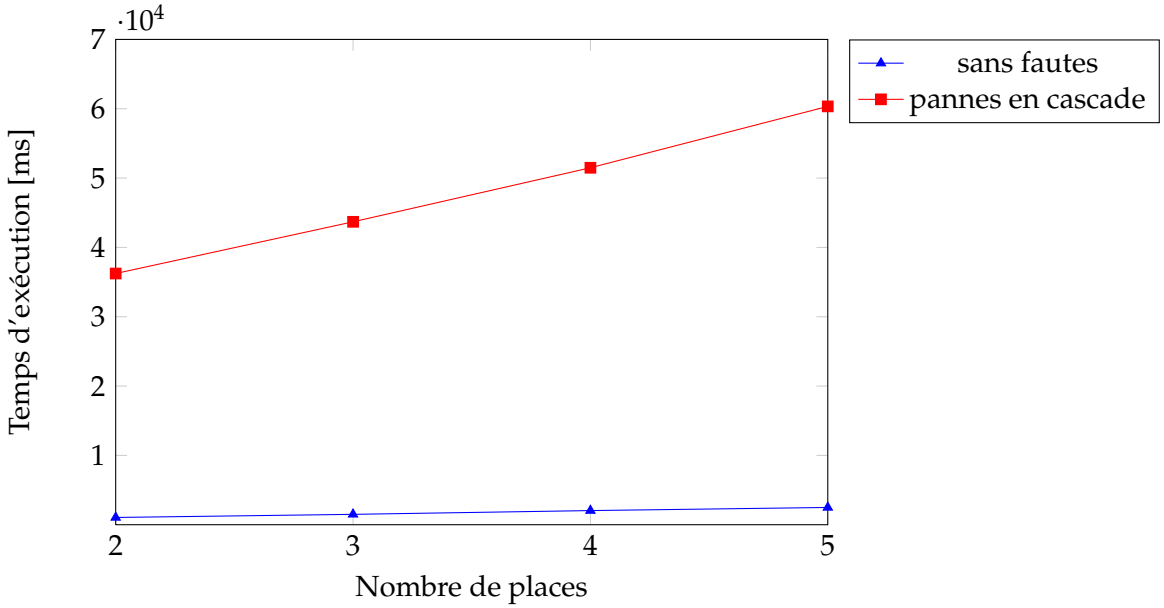

FIGURE 5.7 – Temps d'exécution de l'agent mobile dans le cas d'une série de pannes en cascade.

Le service *AS* décrit dans le chapitre précédent, permet de relancer l'agent mobile même si toutes les places sont en pannes. Au pire des cas, il envoie une réponse au client pour dire que l'agent n'a pas trouvé un chemin pour exécuter la transaction.
Ce service doit être fiable et toujours disponible. Il est implémenté au niveau du gestionnaire de transactions à l'aide d'un protocole d'accord exécuté entre le gestionnaire de transaction et ses copies.

5.4.6 La validation atomique

Dans cette partie de l'expérimentation, notre but consiste à mesurer le temps d'exécution du protocole de validation atomique basé sur le protocole MIC-PAXOS comme brique de base. La figure 5.8 montre la latence du protocole de validation atomique.
Dans la figure 5.9 nous reprenons le graphe précédent pour comparer le temps d'exécution du protocole de validation atomique avec le temps d'exécution de la transaction globale de l'agent mobile tolérant aux pannes.
La latence introduite par le protocole de validation est très négligeable par rapport au temps d'exécution de l'agent.
Ce qui va en faveur de notre solution, car le protocole de validation peut être exécuté plusieurs fois pour la même transaction. Au moins trois fois : la première pour décider de la place source à contrôler, la deuxième pour choisir l'itinéraire à tester et la troisième pour décider de la transaction selon les votes collectés.
Ce cas peut se présenter lorsqu'une place visitée ne répond pas à la demande de validation (en panne, ou préemption de ressources). Le protocole est ré-exécuté pour valider la transaction sur des nœuds alternatifs à ceux qui n'ont pas répondu ou qui ont répondu négativement (dans le cas où plusieurs itinéraires existent) afin de favoriser la validation de la transaction.

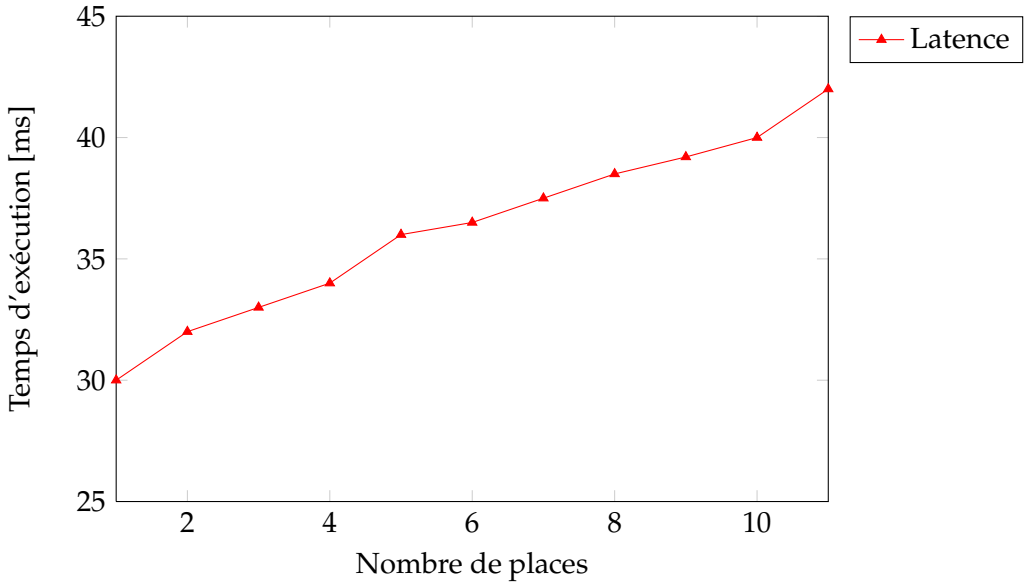

FIGURE 5.8 – Temps de latence du protocole de validation atomique (MIC-PAXOS).

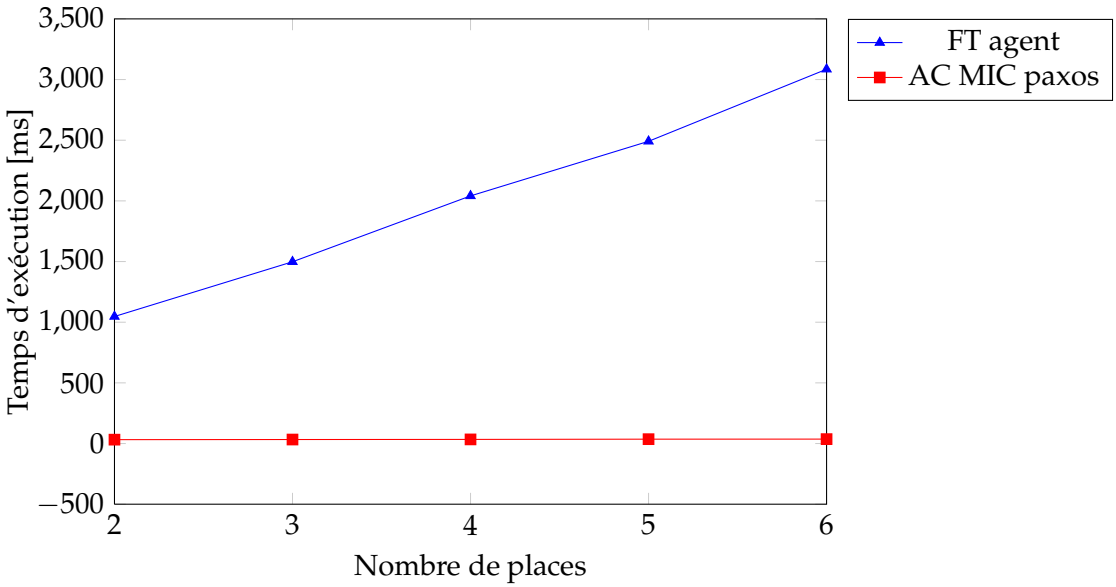

FIGURE 5.9 – Temps d'exécution du protocole de validation atomique comparé au temps d'exécution de l'agent mobile transactionnel.

Le protocole Mic-paxos est utilisé comme brique de base pour implémenter les deux services du gestionnaire de transaction : La disponibilité de la source et la validation atomique. Cela a permis d'assurer la tolérance aux pannes de cette entité qui représente le noyau de la solution. Les résultats sont très satisfaisants sachant que les tests ont été effectué sur une

plateforme grille (grid5000) qui est un environnement relativement stable par rapport à un réseau ouvert comme internet.
Les tests du protocole de tolérance aux pannes sont menés dans un environnement simple qui se rapproche le plus à l'environnement internet où seront exécutés les agents mobiles transactionnels.

5.5 Fonctionnement du protocole en cas de pannes

Un protocole de tolérance aux pannes est fiable si le système continue de fonctionner normalement en présence de pannes.
Les tests menés jusque-là ont montré que l'agent mobile continue son exécution en cas de pannes grâce au protocole de tolérance aux pannes proposé. Ce qui a permis d'ailleurs de mesurer le temps de latence introduit par le protocole.
Dans le but de mieux montrer l'efficacité du protocole, nous avons exploité l'interface graphique qu'offre la plateforme Jade pour suivre le comportement de l'agent mobile tolérant aux pannes en cas de défaillance.
Pour cela, nous avons effectué des tests mono-site. C'est à dire, nous avons lancé plusieurs places Jade sur une même machine physique, dans ce cas toutes les places ainsi que les agents qui s'y trouvent sont visibles sur l'interface de la plateforme Jade. La migration de l'agent est locale et les pannes sont provoquées volontairement. Nous avons pris des captures d'écran pour voir les déplacements de l'agent.
L'agent mobile est lancé sur la première plateforme pour exécuter une transaction du type PRMT (voir chapitre 4). Sur chaque plateforme nous avons lancé un agent de service qui porte le nom du service offert voir figure (5.10).
Lorsque l'agent mobile s'exécute avec succès il retourne au site d'origine après avoir créé un agent de contrôle sur chaque site visité (lignes 2- 9 - 15 - 21 de la figure 5.10).

Dans la figure 5.11, nous avons lancé un site alternatif au site restaurant (ligne 22). Nous nous avons provoqué une panne sur la place du restaurant en arrêtant l'exécution du site (ligne 8). Nous remarquons la création d'un agent de contrôle sur le site alternatif (ligne 23), ce qui laisse à déduire que l'agent mobile a visité ce site alternatif après la détection de la panne puis a continué son chemin normalement.

Dans la figure 5.12, nous avons lancé des places alternatives pour chaque service (lignes 12- 15 - 24). Nous avons provoqué des pannes en cascades sur les places restaurant, musique et T-shirt (lignes 8 - 9 - 10). Nous remarquons la création des agents de contrôle sur les places alternatives (lignes 13-19-25). Notons que l'agent qui retourne à la place source possède un identifiant différent à celui que l'agent original (ligne 3). On peut conclure qu'un nouvel agent mobile est créé (avec un nouvel identifiant). Ce dernier continue l'exécution sur des nœuds alternatifs et retourne à la place source.
Dans le cas où les places alternatives n'existent pas l'agent mobile retourne vers la place source pour en informer le client.

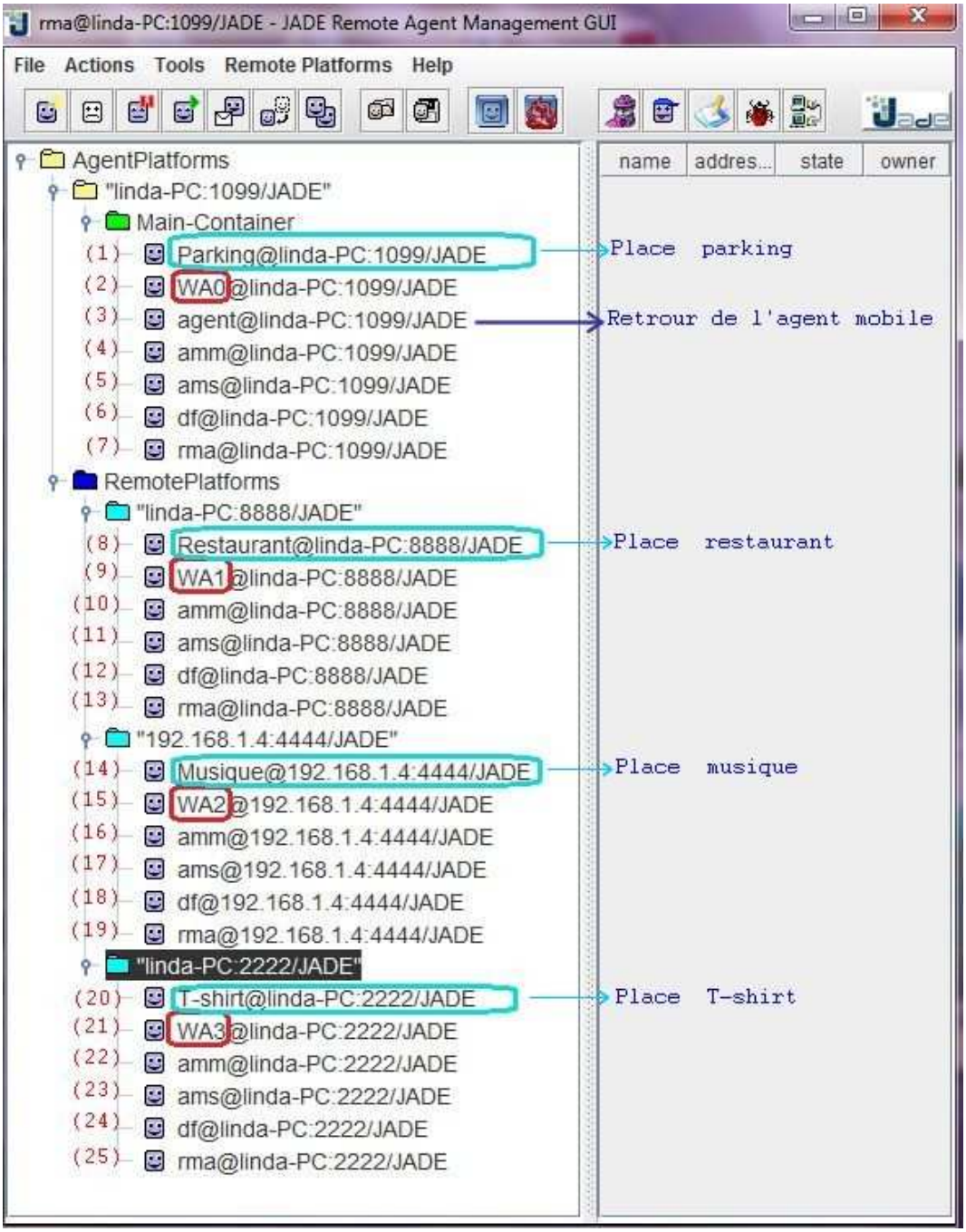

FIGURE 5.10 – Exécution de l'agent avec succès

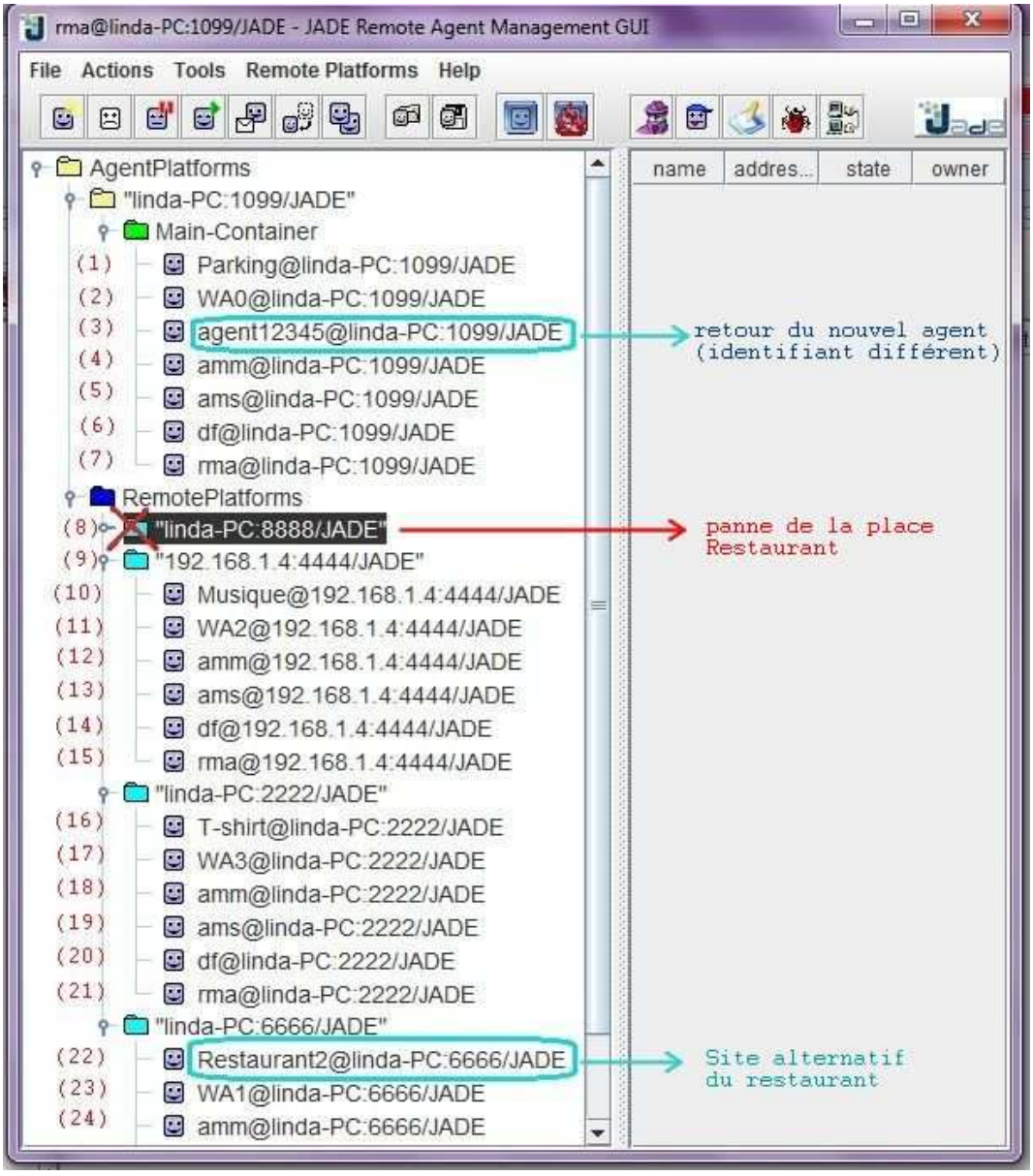

FIGURE 5.11 – Panne de l'agent mobile

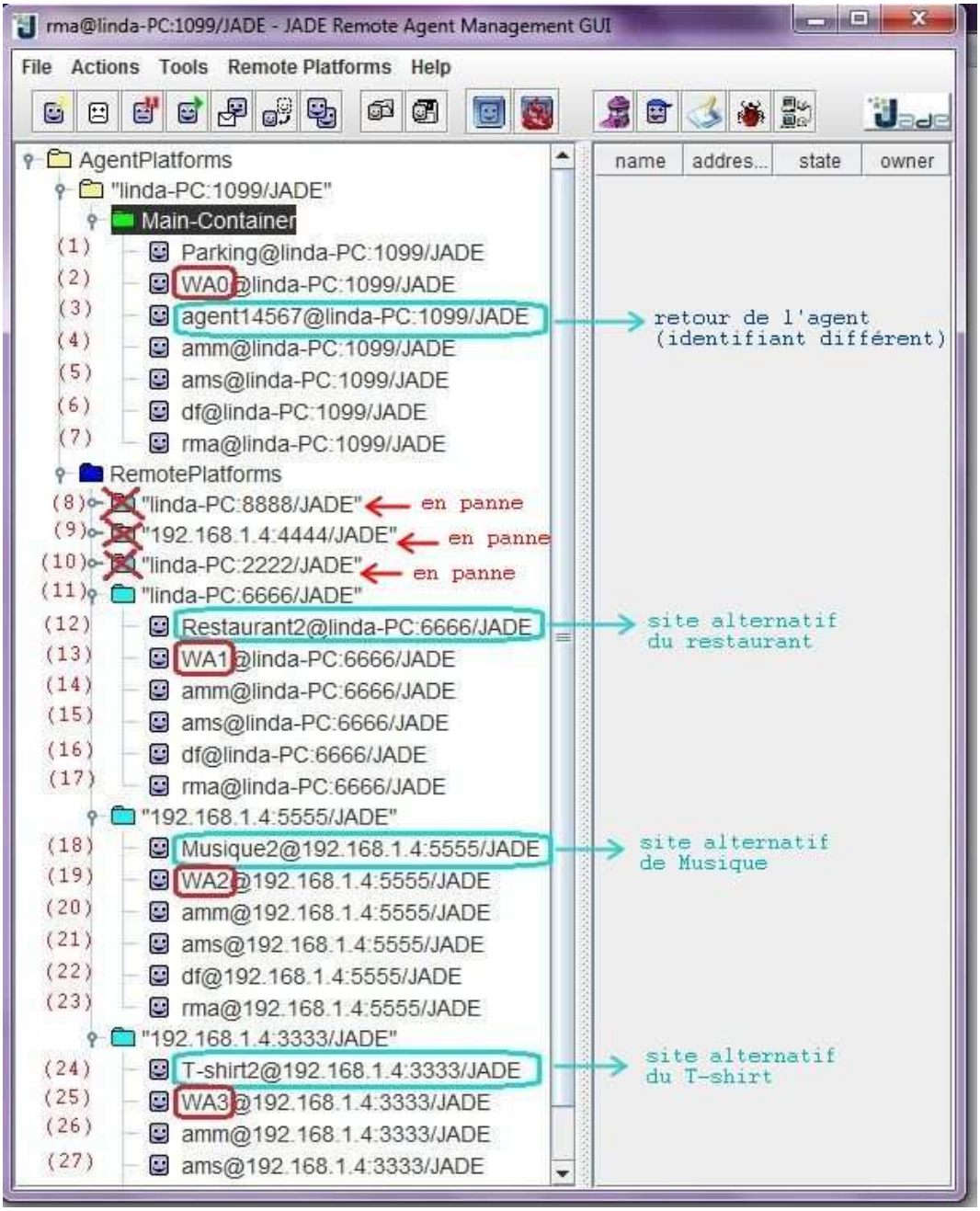

FIGURE 5.12 – Plusieurs pannes en cascade

5.6 Conclusion

Dans le présent chapitre nous nous sommes basés sur l'implémentation de la solution proposée. En présentant l'environnement de développement et de tests. Nous avons simulé quelques scénarios de pannes pour voir le comportement du protocole et son l'impact sur le temps d'exécution. En fait, le temps de latence introduit par la solution est important

notamment dans certains cas de pannes. Mais, il faut préciser qu'en terme de fiabilité la tolérance aux pannes est assurée à 100%.
D'autre part, nous avons proposé une solution basée sur l'accord pour fournir deux services qui permettent de renforcer la fiabilité de l'agent mobile transactionnel et assurer l'atomicité de son exécution. La solution d'accord est très performante en terme de temps d'exécution.

Conclusion Générale

Rendre un agent mobile tolérant aux pannes est une tâche critique découlant de la nécessité de construire des applications fiables. La tolérance aux pannes offre à l'agent mobile une exécution sûre même en cas de défaillance du système ; les erreurs peuvent ainsi être détectées et surmontées.
Au cours de cette thèse, nous nous sommes intéressés à la tolérance aux pannes des agents mobiles transactionnels. Ces derniers étendent le modèle agent mobile pour supporter l'exécution atomique des transactions réparties.
La résolution d'un problème de tolérance aux pannes lié aux agents mobiles transactionnels pose deux difficultés. La première est liée à la mobilité des agents. La seconde à l'exécution transactionnelle. Pour affronter ces deux difficultés, plusieurs approches basées sur différentes techniques ont été proposées.
La technique de réplication garantit le non-blocage des agents mobiles. Néanmoins, elle nuit à la propriété d'exécution unique (exactly-once) fondamentale pour certains types d'application notamment les transactions réparties. Afin de maintenir cette propriété, les réplicats doivent s'entendre à l'unanimité sur une seule copie pour valider l'exécution.
La technique par points de reprise (check-pointing) utilise moins de ressources que la technique de réplication. Cependant, l'augmentation de la fréquence de sauvegarde des points de reprise pourrait dégrader les performances. Par ailleurs, cette technique est bloquante quand la place qui détient les sauvegardes tombe en panne.
Les techniques de réplication et de check-pointing sont donc inadaptées pour les nouvelles générations de réseaux connectant entre autres des machines (laptop, PDAs,...) ayant des capacités de stockage et de calcul limités et caractérisés par des liens de communication versatiles.
Une technique qui commence à prendre de l'ampleur est celle du garde arrière. Plusieurs variantes existent en se basant sur un ou plusieurs gardes arrières qui sont en fait des clones (appelés aussi ombres) de l'agent ou des agents différents créés spécialement pour surveiller l'agent mobile durant son exécution.
Notre solution s'inscrit dans ce type d'approche, étendue pour prendre en charge les pannes sémantiques, supporter la validation atomique et assurer l'exécution transactionnelle.

Réalisations et contributions

Dans la proposition de la solution, nous avons suivi une démarche progressive. Dans un premier temps, nous avons considéré la tolérance aux pannes des agents mobiles transac-

tionnels. Dans ce cadre, notre contribution consiste en un protocole de détection de pannes et de recouvrement prenant en charge les pannes d'infrastructure et les pannes sémantiques également.
Nous avons utilisé la réplication temporelle et une chaîne de dépendance composée de l'agent mobile et de l'ensemble des agents de contrôle créés par cet agent mobile tout au long de son itinéraire.
La détection des pannes se fait par l'envoi de messages de type battements de cœur (heartbeat). Le recouvrement se fait en utilisant les copies de l'agent sur les nœuds précédents et le renvoi de l'agent vers des nœuds alternatifs afin d'éviter le blocage.
Dans un deuxième temps, nous avons considéré l'exécution transactionnelle ou atomique de l'agent. Nous avons défini un modèle transactionnel où chaque transaction est composée d'un ensemble de requêtes exécutées dans un ordre prédéfini. Chaque requête possède plusieurs niveaux de satisfaction. En définissant plusieurs niveaux de satisfaction pour chaque stade d'exécution le nombre de nœuds alternatifs augmente et la requête aura plus de chance d'être satisfaite.
L'utilisation des alternatives comme mécanismes de recouvrement en avant permet d'éviter d'annuler la totalité du travail effectué en cas d'échec et permet de continuer l'exécution et atteindre l'objectif désiré en suivant un autre chemin alternatif.
Ensuite nous avons proposé un protocole de validation atomique pour valider l'exécution globale de l'agent mobile transactionnel.
Pour chaque transaction, un agent mobile est créé. L'agent mobile migre de place en place dans le réseau ad-hoc afin de découvrir un itinéraire pouvant satisfaire les requêtes de la transaction. Nous avons identifié deux services : La disponibilité de la place source (AS) et la validation atomique (AC). Ces deux services doivent être fournis par une entité fiable appelée gestionnaire de transaction (TM). Le TM est répliqué afin d'assurer sa disponibilité. Une forte cohérence doit être maintenue entre le TM et ses copies. Pour cela, nous avons utilisé un protocole de consensus comme brique de base dans l'implémentation du TM.
Nous avons défini un service d'accord appelé *AgreementService*. Ce service implémente un protocole de la famille paxos, le protocole Paxos-MIC, pour construire une séquence de décisions sur laquelle les deux services AS et AC sont implémentés. Ce protocole garantit la persistance de toutes les décisions prises. Cela permet d'obtenir un gestionnaire de transaction fiable qui supervise le comportement d'un agent mobile transactionnel dès sa création, grâce au service AS et jusqu'à sa validation grâce au service AC.
Nous avons pu montrer par l'implémentation de la solution et les tests réalisés que la solution proposée est prometteuse du fait qu'elle tolère les pannes d'infrastructure et les pannes sémantiques. Elle assure les propriétés du non-blocage, l'exécution unique et l'atomicité. De plus, tandis que l'agent évolue dans un réseau qui peut être un réseau ad-hoc, le TM et ses copies évoluent dans un réseau indépendant (grille de calcul), ce qui marque la particularité intéressante de la solution proposée.
En fait, augmenter les agents mobiles avec le support transactionnel permet aux transactions réparties de s'exécuter dans un environnement massivement distribué et dynamique.

Perspectives

Les agents mobiles constituent une brique élémentaire sur laquelle peut reposer un calcul réparti, particulièrement les transactions réparties. Pour en exploiter la richesse, nous avons

essayer de résoudre un des problèmes fondamentaux qui freinent l'utilisation de cette technologie à large échelle. Cependant, beaucoup d'améliorations seront bénéfiques, à savoir :

- Nous avons considéré uniquement les pannes franches dans le protocole de tolérance aux pannes. Dans un environnement réel d'autres types de pannes peuvent survenir. Le protocole peut être étendu pour supporter les pannes malicieuses.
- La solution proposée est une approche orientée agent c-à-d le code de tolérance aux pannes est transporté avec l'agent mobile. Une autre façon de faire serait de séparer le code de tolérance aux pannes pour l'intégrer dans la plateforme qui exécute l'agent. Cela permettrait d'alléger le code de l'agent et accélérer sa migration et donc réduire son temps d'exécution.
- Les tests effectués dans le cadre de cette thèse concernent uniquement une seule transaction. Il serait intéressant de tester la solution avec plusieurs transactions concurrentes afin de vérifier l'efficacité du modèle transactionnel proposé et déterminer la mise à l'échelle de la solution en terme du nombre de transactions validées ou annulées.
- Au niveau du modèle transactionnel, nous avons adopté un modèle de transactions linéaire où chaque transaction est composée d'un ensemble de requêtes réparties sur le réseau. Les modèles de transactions avancés tels que les transactions enchaînées avec points de reprise ou les transactions emboîtées peuvent être intégrés dans le protocole proposé. Cela permet à l'agent de supporter un plus grand nombre d'applications.

Bibliographie

[1] D. Gilbert, M. Aparicio, B. Atkinson, S. Brady, J. Ciccarino, B. Grosof, P. O'Connor, D. Osisek, S. Pritko, Spagna R., and L. Wilson. IBM Intelligent Agent Strategy. Technical report, IBM Corporation, 1995.

[2] Kurt Rothermel and Markus Schwehm. Mobile agents. In *the Second International Workshop on Mobile Agents, MA'98*, 1998.

[3] Salah El Falou. *Programmation répartie, optimisation par agent mobile*. PhD thesis, Université de CAEN, 2006.

[4] Christophe Cubat and D I T Cros. *Agents Mobiles Coopérants pour les Environnements Dynamiques*. PhD thesis, Institut National Polytechnique de Toulouse, 2005.

[5] Stefan Pleisch. *Fault-tolerant and transactional mobile agent execution*. PhD thesis, 2002.

[6] Luca Cardelli. Obliq : A language with distributed scope. Research Report 122, Digital Equipment Corporation Systems Research Center. Technical report, 1994.

[7] Safetcl. Tcl, www.tcl.tk/software/plugin/safetcl.html, 2005.

[8] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. Mole – Concepts of a mobile agent system. *World Wide Web*, 1(3) :123–137, 1998.

[9] Foundation for Intelligent Physical Agents. FIPA Abstract Architecture Specification. http ://www.fipa.org, 2000.

[10] Dejan Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. MASIF The OMG mobile agent system interoperability facility, http ://www.omg.org. In Springer Berlin Heidelberg, editor, *Mobile Agents*, pages 50–67. Berlin, 1998.

[11] Tecnologia Universidade Nova. A Mobile Agent Systems ' Overview. pages 1–29. 2002.

[12] Niranjan Suri, JM Bradshaw, and MR Breedy. NOMADS : toward a strong and safe mobile agent system. In *AGENTS '00 the fourth international conference on Autonomous agents*, pages 163–164, 2000.

[13] D.B. Lange and M.Oshima. Mobile Agents with Java : The Aglet API. *World Wide Web*, 1(3) :111–121, 1998.

[14] JADE. The Java Agent DEelopement framework. http ://jade.tilab.com/, 2001.

[15] Mitsubishi Electric Information Technology Center AMERICA. http ://www.cis.upenn.edu/~bcpierce/courses/629/papers/Concordia-MobileAgentConf.html, 1997.

[16] MAF. Mobile Agent Framework (MAF). http ://maf.sourceforge.net/, 2001.

[17] Stanford Center for Design. JATlite, http ://www-cdr.stanford.edu/ProcessLink/papers/JATL.html, 1997.

[18] IRANJAN SURI and VITEK JAN. Mobile Agents. In Meyers and A Robert, editors, *Computational Complexity*, pages 1880–1893. Springer New York, 2012.

[19] L.M. Silva, V. Batista, and J.G. Silva. Fault-tolerant execution of mobile agents. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 135–143. IEEE Comput. Soc, 2000.

[20] IRIT Lab. Javact. http ://www.javact.org/JavAct.html, 2004.

[21] Siegfried Rouvrais. *Utilisation d'agents mobiles pour la construction de services distribués*. PhD thesis, Université de Rennes1, 2002.

[22] Gilles Grimaud. *Camille : un système d'exploitation ouvert pour carte à microprocesseur*. PhD thesis, Université Lille 1, 2000.

[23] Xavier Besseron. *Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle*. PhD thesis, Université de Grenoble, 2010.

[24] Gilles Zwingelstein. Sûreté de fonctionnement des systèmes industriels complexes. Technical report, 2009.

[25] Samir Jafar. *Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité*. PhD thesis, 2006.

[26] Pira Clement. *Une méthodologie pour appréhender la décision collective dans des systèmes multiagents sujets aux défaillances*. PhD thesis, Pierre et Marie Curie, 2009.

[27] S. Pleisch and A. Schiper. Modeling fault-tolerant mobile agent execution as a sequence of agreement problems. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 11–20. IEEE Comput. Soc, 2000.

[28] Stefan Pleisch and André Schiper. FATOMAS-a fault-tolerant mobile agent system based on the agent-dependent approach. In *Proceedings International Conference on Dependable Systems and Networks*, number Dsn, pages 215–224. IEEE Comput. Soc, 2001.

[29] Stefan Pleisch and André Schiper. Approaches to fault-tolerant and transactional mobile agent execution—an algorithmic view. *ACM Computing Surveys*, 36(3) :219–262, September 2004.

[30] Taha Osman, Waleed Wagealla, and Andrzej Bargiela. An Approach to Rollback Recovery of. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 34(1) :48–57, 2004.

[31] Markus Strasser and Kurt Rothermel. System Mechanism for Partial Rollback of Mobile Agent Execution. In *Proc. 20th Int'l Conf. on Distributed Computing*, pages 20–28, 2000.

[32] D. Johansen, K. Marzullo, F.B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP : practical fault-tolerance for itinerant computations. *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 180–189, 1999.

[33] Simon Pears, Jie Xu, Cornelia Boldyreff, and Computer Science. Mobile Agent Fault Tolerance for Information Retrieval Applications : An Exception Handling Approach. In *the Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03)*, 2003.

[34] Jie Xu and Simon Pears. A Dynamic Shadow Approach to Fault-Tolerant Mobile Agents in an Autonomic Environment. *Real-Time Systems*, 32(3) :235–252, December 2005.

[35] Michael R. Lyu and Tsz Yeung Wong. A Progressive Fault Tolerant Mechanism in Mobile Agent Systems. In *In Proc. of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI'03). Vol. IX. Orlando, Florida.*, pages 299–306, 2003.

[36] Sareh Beheshti and Mansoureh Ghiasabadi. Fault Tolerant Mobile Agent Systems by using Witness Agents and Probes. pages 1–5, 2007.

[37] Wang-xiao Yun, Ye-jian Nan, Huang-guo Jin, and Yu-mei He. Improvement of Temporal-replication Mechanism in Mobile Agent System Fault-tolerant Model. In *International Conference on Computational Intelligence and Security Workshops Improvement*, pages 534–537, 2007.

[38] MaengSoon Baik, SungJin Choi, Chayong Kim, and ChongSun Hwang. Optimistic Temporal Replication Based Approach to Fault-Tolerant Mobile. In *5th, International conference on advanced communication technology ; Broadband network infrastructure*, pages 514–519, 2003.

[39] Rahul Hans and Ramadandeep Kaur. Novel Dynamic Shadow Approach for Fault Tolerance in Mobile Agent Systems. In *6th International Conference on Signal Processing and Communication Systems (ICSPCS),*, pages 1–6, 2012.

[40] a. Mohindra, A. Purakayastha, and P. Thati. Exploiting non-determinism for reliability of mobile agent systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 144–153. IEEE Comput. Soc, 2000.

[41] S Pleisch and A Schiper. Execution atomicity for non-blocking transactional mobile agents. In *Proc. of Int. Conference on Parallel and Distributed Computing and Systems*, 2003.

[42] Danes Adriana. *Service transactionnel souple pour systèmes répartis à objets persistants*. PhD thesis, Université Joseph Fourier - Grenoble1, 1996.

[43] Mayyad Jaber. *Architecture de Système d'information distribué pour la gestion de la chaîne logistique : Une approche orientée servic*. PhD thesis, Institut National des Sciences Appliquées de Lyon, 2009.

[44] Sami Bhiri. *Approche Transactionnelle pour Assurer des Compositions Fiables de Services Web*. PhD thesis, Université Henri Poincaré - Nancy 1, 2005.

[45] Heiko Schuldt, Gustavo Alonso, Catriel Beeri, and Hans-Jörg Schek. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems*, 27(1) :63–116, March 2002.

[46] Hartmut Vogler and Alejandro Buchmann. Using multiple mobile agents for distributed transactions. In *3rd IFCIS International Conference on Cooperative Information Systems*, pages 114–121, 1998.

[47] Ron Sher Ý. Mobile Transactional Agents. In *21st International Conference on Distributed Computing Systems*, pages 73–80, 2001.

[48] Youhei Tanaka, Naohiro Hayashibara, Tomoya Enokido, and Makoto Takizawa. Fault-Tolerant Destributed Systems in a Mobile Agent Model. *Journal of Cluster Computing*, 10(1) :81–93, 2007.

[49] Jim Gray. "Notes on Database Operating Systems,. *Operating Systems : An Advanced Course, LNCS,*, 60, 1978.

[50] Maha Abdallah and Philippe Pucheral. Validation Atomique : état de l ' art et perspectives. Technical Report 6, Université de Versailles,Laboratoire PRiSM, 1998.

[51] Gérard Lelann, Pascale Minet, and David Powell. Tolérance aux fautes et systèmes répartis : Concepts et mécanismes. Technical report, 1993.

[52] ISO. Open System Interconnection - Distributed Transaction Processing (OSI-TP) Model, 1992.

[53] X/Open CAE Specification. Distributed Transaction Processing : the XA Specification XO/CAE/91/300, 1991., 1991.

[54] Object Management Group. Object Transaction Service. 1994.

[55] Skeen, Dale and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3) :219–228, 1983.

[56] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :374–382, 1985.

[57] Sam Toueg and Tushar Deepak Chandra. Unreliable Distributed Failure Detectors Systems for Reliable Distributed Systems. *Journal of the ACM*, 43(2) :225–267, 1996.

[58] Marcos Kawazoe Aguilera, Hugues Delporte-Gallet, Carole , Fauconnier, and Sam Toueg. *Stable Leader Election*. Springer-Verlag, London, UK, 2001.

[59] S. Frolund and R. Guerraoui. e-Transactions : end-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering*, 28(4) :378–395, April 2002.

[60] Svend Frolund and Rachid Guerraoui. Implementing e-transactions with asynchronous replication. *IEE Transaction On Parallel and Distributed Systems*, 12(2) :133–146, 2001.

[61] Svend Frolund and Rachid Guerraoui. e-Transactions : End-to-End Reliability for Three-Tier Architectures. *IEEE Transactions on Software Engineering*, 28(4) :378–395, 2002.

[62] Francesco Quaglia and Paolo Romano. Ensuring e-Transaction with Asynchronous and Uncoordinated Application Server Replicas. *IEEE Trans. Parallel Distrib. Syst.*, 18(3) :364–378, 2007.

[63] Paolo Romano and Francesco Quaglia. Providing e-Transaction Guarantees in Asynchronous Systems with No Assumptions on the Accuracy of Failure Detection. *IEEE Transactions on Dependable and Secure Computing*, 8(1) :104–121, January 2011.

[64] F Greve, Michel Hurfin, and JPL Narzul. Adam : une bibliotheque de composants d'accord pour la programmation d'applications fiables. In *Actes des Journées Composants*, number 47, pages 1–9, Lille, France, 2004.

[65] Olivier Baldellon and Michel Raynal. Le problème du consensus byzantin en environnement asynchrone. Technical report, École Normale Supérieure de Cachan, antenne de Bretagne Master informatique spécialité recherche en informatique (IRISA), 2010.

[66] L Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998.

[67] L Lamport. {Paxos} made simple. *ACM SIGACT News*, 2001.

[68] R Boichat, P Dutta, S Frø lund, and R Guerraoui. Deconstructing paxos. *ACM Sigact News*, 2003.

[69] L Lamport. Fast paxos. *Distributed Computing*, 2006.

[70] Michel . Hurfin, Izabela. Moise, and J.Pierre. Le Narzul. An adaptive Fast Paxos for making quick everlasting decisions. In *The 25th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA-2011)*, pages 208–215, 2011.

[71] Michel Hurfin and Izabela Moise. A Multiple Integrated Consensus Protocol based on Paxos , FastPaxos and Fast Paxos. Technical Report December, 2009.

[72] Izabela Moise. *Efficient Agreement Protocols in Asynchronous Distributed Systems*. PhD thesis, Université de Rennes 1 France, 2011.

[73] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus ? In Springer, editor, *Distributed Algorithms*, pages 87–100. 1995.

[74] Stefan Bottcher, Le Gruenwald, and Sebastien Obermeier. A failure tolerating atomic commit protocol for mobile environments. In *The 8th International Conference on Mobile Data Management*, pages 158–165. IEEE Computer Society, 2007.

Annexe A

Les Publications

Papier Journal

Zeghache, L., Badache, N., Hurfin, M. and Moise, I. (2014) 'Reliable mobile agents with transactional behaviour', Int. J. Communication Networks and Distributed Systems,

Chapitre de livre

Linda Zeghache, Michel Hurfin, Izabela Moise, Nadjib Badache : Providing Reliability for Transactional Mobile Agents, (revised and selected version), Advanced Infocomm Technology, Springer LNCS, Volume 7593, 2013, pp. 177-190, ISBN : 978-3-642-38226-0 (Print) 978-3-642-38227-7 (Online)

Papiers Conférences et Workshops

Izabela Moise, Michel Hurfin, Linda Zeghache, Nadjib Badache : Cloud-Based Support for Transactional Mobile Agents. 25th IEEE International Conference on Advanced Information Networking and Applications Workshops AINA 2011 : 190-197

Izabela Moise, Michel Hurfin, Linda Zeghache, Nadjib Badache : Remote Reliable Services to Support Transactional Mobile Agents. Proceedings of The Ninth IEEE International Symposium on Networking Computing and Applications, NCA 2010 : 275-279

Linda Zeghache, Nadjib Badache : Optimistic Replication Approach for Transactional Mobile Agent Fault Tolerance. 11th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, SNPD 2010, 193-198