Université de Nice-Sophia Antipolis
UFR SCIENCES

École Doctorale : STIC

THÈSE

Présentée pour obtenir le titre de :
DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ DE NICE SOPHIA ANTIPOLIS

Spécialité : INFORMATIQUE

par

Fabrice HUET

Équipe d'accueil : OASIS - INRIA Sophia Antipolis

Titre de la thèse :

Objets mobiles :
conception d'un middleware et évaluation de la communication

Thèse dirigée par Françoise BAUDE et Denis CAROMEL
soutenue le 11 décembre 2002

M. :	Philippe	NAIN	Président
MM. :	Brigitte	PLATEAU	Rapporteurs
	Jean-Bernard	STEFANI	
	Bernard	TOURSEL	
MM. :	Denis	CAROMEL	Directeurs
	Françoise	BAUDE	

Remerciements

Je remercie avant tout Françoise Baude et Denis Caromel qui m'ont encadré et encouragé ces trois dernières années et sans qui cette thèse n'aurait pas pu se faire. Ils ont su me guider par leurs conseils avisés et je leur en suis reconnaissant.

L'accueil qui m'a été réservé dans l'équipe OASIS a été à la fois chaleureux et enrichissant scientifiquement et m'a permis de m'ouvrir à de nouveaux sujets. J'ai beaucoup apprécié l'énergie déployée par Isabelle Attali pour que nous tous puissions travailler dans d'excellentes conditions.

Je tiens à remercier bien entendu les membres du jury : Philippe Nain qui a accepté de présider ce jury, et Brigitte Plateau, Jean Bernard Stefani et Bernard Toursel pour avoir accepté d'être rapporteurs de cette thèse.

Plusieurs personnes ont effectué la tâche ingrate mais indispensable de relecture des nombreuses versions de ce manuscrit et je ne saurais leur exprimer ma gratitude. Je pense particulièrement à Sara Alouf, Laurent Baduel, Arnaud Contes, Christian Delbé.

Grâce à Julien Vayssière, j'ai pu découvrir que la musique des années quatre-vingt ne méritait pas de rester dans des tiroirs, ce qui nous a offert une source de divertissement non négligeable dans le bureau que nous avons partagé.

Et bien sur, mes pensées vont à Sara Alouf qui a été à mes côtés durant ces trois années de thèse et sans qui tout aurait été plus difficile.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 À la recherche d'un billet

Monsieur Dupont est un homme extrêmement occupé qui doit fréquemment se déplacer pour des raisons professionnelles. Cela ne l'empêche pas d'être économe, c'est pourquoi il est toujours à la recherche du meilleur prix pour un billet d'avion. Hélas, il habite loin de toute agence de voyage et ne bénéficie pas d'une connexion à haut débit lui permettant de chercher à loisir sur les sites des vendeurs de billets. Il n'a de toute façon pas beaucoup de temps à consacrer à cette tâche.

Il a entendu parler d'un paradigme de programmation appelé *mobilité* qui consiste à donner la possibilité aux éléments d'un programme de changer de machine en cours d'exécution. Cela permet la création d'objets mobiles dotés d'un comportement autonome et appelés agents. Ceux-ci sont capables de se déplacer de machines en machines pour effectuer des tâches pour le compte de leur créateur. Après s'être renseigné, il est convaincu que c'est la solution à ses problèmes et décide donc d'utiliser un agent mobile pour chercher les bonnes affaires. Son idée est de programmer un agent pour qu'il rende visite aux sites des principales compagnies aériennes et lui ramène le meilleur tarif. Les avantages de cette approche sont multiples. Il n'a plus à se soucier de cette tâche fastidieuse qui est maintenant effectuée automatiquement. L'agent étant autonome, Monsieur Dupont n'a pas besoin de rester connecté longtemps ce qui l'arrange bien car souvenons-nous, il est économe. Enfin, il est assuré d'avoir le meilleur tarif à chaque fois, pour peu que ceux-ci ne changent pas rapidement. Réjoui par cette idée, il se met à la tâche et après quelques heures de travail est fin prêt à tester son idée.

Pour ce premier essai, il a décidé de mettre toutes les chances de son côté en restant connecté et en ne comparant que deux compagnies. Son agent envoyé, Monsieur Dupont attend quelques minutes avant de le voir revenir, avec un prix très intéressant. Après quelques vérifications, il est convaincu d'avoir eu la meilleure offre et décide de recommencer l'expérience mais cette fois en ne restant pas connecté. Encore une fois, tout se passe bien, l'agent revenant tout seul chez Monsieur Dupont quand il se reconnecte. Extrêmement content des résultats, il décide d'augmenter la difficulté en étudiant non pas deux mais vingt compagnies. Après quelques minutes, l'agent est de retour. Monsieur Dupont est impressionné par la rapidité avec laquelle l'agent a visité tous les sites, lui qui est habitué aux délais liés à sa connexion limitée. Il se dit que vraiment il a bien fait d'écrire cet agent. Peut-être rendu euphorique par ses succès récents, il décide d'envoyer son agent pour une ultime mission : chercher le billet d'avion le moins cher parmi toutes les compagnies aériennes du monde. Il réalise vite l'énormité de la tâche demandée et décide de rappeler son agent pour lui faire arrêter son voyage. C'est à ce moment qu'il réalise qu'il y a une faille dans son application : il n'a pas prévu de moyen pour assurer les communications avec son agent!

La morale de cette histoire est qu'une application mobile doit non seulement être capable de faire se déplacer des objets, mais aussi fournir un mécanisme permettant de maintenir les communications. Ce problème est différent de ceux rencontrés dans les systèmes distribués plus classiques où les problèmes de communication n'arrivent pas systématiquement et ne sont pas liés au fonctionnement normal de l'application. Dans le cas d'applications mobiles, la possibilité qu'une communication échoue parce qu'un élément a changé de lieu d'exécution doit être intégrée dès la conception. Il s'agit là, précisément, du thème de cette thèse.

1.2 Contributions et Plan

Dans un premier temps nous avons étudié plusieurs mécanismes permettant d'assurer les communications en présence de mobilité. Cette étude nous a permis de dégager une classification de ceux-ci en trois familles, chacun d'entre elles ayant certaines caractéristiques que nous détaillons (*Chapitre 2*).

Nous nous sommes ensuite attaché à montrer comment il était possible d'ajouter à une bibliothèque Java (*ProActive*), basée sur le concept d'objet actif, la notion de mobilité faible. L'API développée pour permettre de construire des applications mobiles est à la fois simple à utiliser et puissante dans ses possibilités. Nous montrons aussi comment il est possible d'avoir des comportements de plus haut niveau grâce aux abstractions que sont les événements de migration, les itinéraires et un squelette d'agent générique (*Chapitre 3*).

Afin d'assurer les communications entre objets, nous avons étudié deux mécanismes : des répéteurs et un serveur de localisation. Un protocole précis est décrit permettant de maintenir la sémantique de communication de *ProActive* dans les deux cas. Pour chacun de ces moyens de communication, nous montrons les modifications qu'il est nécessaire d'apporter aux objets du niveau méta pour les implémenter efficacement et élégamment (*Chapitre 3*).

L'expérience acquise lors de ce premier travail nous a amené à nous poser la question des performances des deux mécanismes de communication choisis. Nous voulions en particulier étudier le temps que mettrait un objet fixe, nommé *source*, pour communiquer avec un objet mobile, nommé *agent*. Pour ne pas nous limiter aux seules conditions expérimentales accessibles, conditions forcément limitées par notre environnement de travail, nous avons choisi une approche formelle basée sur l'utilisation des chaînes de Markov. Nous avons montré que les deux mécanismes peuvent être entièrement décrits en considérant comme paramètres le taux de communication de la source, le taux et la durée de migration de l'agent, le débit du réseau et la vitesse du serveur. Grâce à l'étude des deux modèles, les métriques suivantes ont pu être exprimées : temps de réponse pour le serveur et les répéteurs, nombre moyen de répéteurs. Pour ces deux dernières une forme explicite a pu être trouvée (*Chapitre 4*).

Nos modèles ont été bâtis en utilisant des hypothèses fortes en particulier concernant les lois de distribution de nos paramètres. Pour les valider, nous avons construit un simulateur à événements discrets simulant le comportement de *ProActive*. Cela nous a permis de vérifier la correction de nos modèles, mais aussi leur robustesse quand les lois des distributions n'étaient plus exponentielles. Confiant dans nos résultats, nous avons mené une campagne d'expérimentations ce qui nous a permis de vérifier que dans le cas réel les résultats étaient tout à fait acceptables (*Chapitre 5*).

Nous avons consacré ensuite une partie de notre étude à la prédiction de performances dans d'autres conditions que celles rencontrées, montrant l'intérêt des modèles développés. Enfin, nous avons montré comment il était possible d'étendre nos résultats dans le cas où plusieurs sources et plusieurs agents essaient de communiquer en utilisant le même serveur de localisation (*Chapitre 5*).

La dernière partie de cette thèse est consacrée à l'étude d'un nouveau mécanisme pour la communication. Ce mécanisme essaie de combiner les points forts des deux précédents tout en n'en ayant pas les points faibles. Deux variantes sont décrites et leurs performances sont étudiées en utilisant des simulations, nous permettant de confirmer leur intérêt (*Chapitre 6*).

Les travaux présentés dans cette thèse ont fait l'objet de publications dans des revues et conférences nationales et internationales [8, 4, 5, 9, 10]

Chapitre 2

État de l'art et outils utilisés

La mobilité se définit simplement par le déplacement d'un programme en cours d'exécution d'une machine à une autre. L'entité mobile est appelée agent ou objet mobile. La mise en œuvre de la mobilité peut se faire de deux manières, qualifiées de fortes et faibles que nous allons détailler.

2.1 Mobilité forte, mobilité faible

Définition 2.1.1 (Mobilité forte) *La mobilité forte est le déplacement d'un programme et de son état complet à un instant arbitraire de son exécution, et la reprise de celle-ci sur le site distant au point où elle a été interrompue.*

Le point important ici est la notion d'état complet du programme. Il s'agit de tous les objets composants l'application, leurs états (i.e. attributs et variables) mais surtout l'état des threads existants dans l'application. Les premières implémentations de migration forte peuvent être trouvées dans les systèmes d'exploitation tels que Charlotte[6] Emerald[35], Amoeba [58] et V-System [40]. Pour pouvoir faire de la mobilité forte, un système doit permettre l'accès en lecture et en écriture au compteur de programme et à la pile d'appels de façon à interrompre l'exécution à n'importe quel instant et la reprendre au même point. Cette nécessité a empêché l'implémentation simple de tels systèmes dans l'environnement Java. Pour des raisons de sécurité, l'accès à l'état d'exécution des threads Java n'est pas possible pour un programme. Pour palier à ce manque plusieurs approches sont possibles. La première consiste à pratiquer une transformation du code source comme dans [56]. À chaque méthode est ajouté un objet représentant son état et lorsqu'une migration est initiée, une exception est levée et se propage à travers chacune des méthodes de la pile, provoquant l'instanciation des ces objets. La reconstruction se fait sur le site distant en ré-appelant les méthodes et en leur passant en paramètre l'objet contenant la sauvegarde de leur état. Cette technique introduit une baisse des performances comprise entre 8 et

180% suivant les applications. L'augmentation de la taille du code se situe aux alentours de 200%.

Une deuxième consiste à modifier le bytecode de l'application pour y ajouter la sauvegarde de l'état des threads comme indiqué par [55] et [68]. Dans les deux cas, il est montré que cette opération est relativement coûteuse en terme de performances (entre 1.5% et 27%) et en terme d'espace (entre 106% et 300%).

Finalement, il est possible de modifier la machine virtuelle pour Java, afin de rendre accessible à une application l'état des threads [14, 53]. Cela se fait malheureusement au détriment de la portabilité et du déploiement car les machines virtuelles standard ne peuvent pas exécuter ces programmes. Dans la plupart des cas, la migration est initiée par un intervenant extérieur qui déplace littéralement le processus ou l'objet.

Parmi les bibliothèques fournissant des mécanismes de migration forte citons D'agents (auparavant Agent-TCL) [32], WASP [27] et Sumatra [2].

Il existe une deuxième catégorie de mobilité, dite faible qui relâche les contraintes sur l'interruption et le redémarrage d'un programme.

Définition 2.1.2 (Mobilité faible) *La mobilité faible est le déplacement d'un objet et de son état à un instant précis de son exécution et la reprise de celle-ci sur un site distant à un point éventuellement différent de celui où elle a été interrompue.*

Nous voyons bien que dans le cadre de la migration faible, il y a deux différences importantes mais qui peuvent se résumer simplement : les événements relatifs à la mobilité (départ et arrivée) sont à temps discret pour la migration faible. Concrètement, la migration ne peut commencer qu'à des points précis de l'exécution. Le rôle de ces points est de permettre la sauvegarde de l'état de l'objet. La reprise de l'exécution ne peut se faire elle aussi qu'en des points précis qui permettent principalement la restauration de l'état du programme.

La liste des bibliothèques implémentant la migration faible est extrêmement longue, parmi les plus connues citons Aglet [3], Voyager [70], MOA [44], mole [12], Concordia [34] et Odyssey [48].

2.2 Mobilité et applications

Le paradigme du code mobile existe déjà depuis de nombreuses années à travers le déplacement des processus des systèmes distribués [58, 6], cependant, peu d'applications l'utilisant sont déployées. Une des raisons pour cela est que les applications mobiles sont particulièrement adaptées aux environnements ouverts et hétérogènes qui commencent seulement maintenant à atteindre un déploiement et une robustesse suffisants. Mais ce qui limite vraiment leur utilisation est qu'il n'existe pas actuellement d'application spécifique nécessitant la mobilité. Il est possible d'écrire tout aussi efficacement une application donnée en utilisant d'autres techniques comme les appels distants par exemple. Cette situation contraste énormément avec celle du peer-to-peer [59, 39, 7] qui a connu une explosion ces dernières années. Le partage de fichiers s'est immédiatement imposé comme une application incontournable de ces systèmes, difficile sinon impossible à obtenir par d'autres moyens.

Il ne faut cependant pas croire que la mobilité n'a pas de raison d'être. Elle permet d'obtenir un environnement général permettant de développer des applications extrêmement différentes. Il est ainsi possible de rassembler sous une même bannière des mécanismes qui nécessitaient autrefois des techniques différentes et difficiles à faire collaborer. Ce point sera illustré dans les prochaines sections à travers la description des résultats qu'il est possible d'obtenir avec de la mobilité. Apparaîtra alors très clairement la difficulté d'obtenir des applications exhibant ces propriétés en utilisant des techniques plus classiques. Nous n'allons ici décrire que les résultats les plus importants, une liste plus exhaustive peut être trouve dans [21] et [26].

2.2.1 Clients mobiles

De plus en plus d'utilisateurs sont mobiles, c'est à dire changent de machine ou de point de connections à Internet. Avec cette situation sont souvent associées des ressources limitées. Un agent mobile peut être crée et envoyé sur un serveur distant où il pourra faire de l'adaptation de données pour l'utilisateur. Il est ainsi possible d'adapter finement et rapidement l'information aux ressources disponibles. Il est aussi possible d'imaginer un utilisateur se déplaçant avec un minimum de logiciel et qui en se connectant les ferait se déplacer de son ordinateur fixe vers son ordinateur temporaire. Remarquons que cela se rapproche fortement des bureaux virtuels dans les terminaux légers [1].

2.2.2 Clients intermittents

Certains traitements ou opérations peuvent prendre une longue durée pour arriver à complétion. Dans le cas où l'opération se fait sur un site distant, envoyer un agent permet de ne pas rester connecter tout le temps de traitement. Lors de la prochaine reconnections, l'agent pourra revenir sur la machine de son propriétaire avec le résultat du traitement. Cela peut s'avérer très intéressant lorsqu'il n'est pas possible pour des raisons pratiques de maintenir une connectivités permanente.

2.2.3 Analyse de données, recherche d'informations

Avec le développement des réseaux et des ordinateurs aux ressources de plus en plus importantes est arrivé la création de base de données contenant des quantités énormes d'information. Pour certaines, il n'est pas physiquement possible de télécharger les données pour les analyser localement. La plupart fournissent ainsi des méthodes d'accès pour envoyer des ordres de traitement et n'obtenir que le résultat de l'opération. Le code mobile permet de généraliser cette méthode en la rendant plus générique puisque n'importe quel code peut être envoyé sur un serveur où se trouvent des informations. Il est ainsi possible d'effectuer des traitements beaucoup plus fins qu'à travers des interfaces rigides. Dans le cas où l'information est située sur plusieurs sites, le problème se traite tout aussi simplement du fait que l'agent voyage avec son état. Il peut ainsi affiner son traitement suivant les résultats déjà obtenus.

2.2.4 Amélioration des performances

Effectuer un calcul ou analyser des données nécessite une certaine durée qui dépend bien sûr de la puissance de la machine où s'effectuent les opérations, mais aussi des performances générales des mécanismes de communication. Dans le cas où tout s'effectue sur une unique machine, le coût est relativement faible et est souvent négligé. Mais si nous nous plaçons dans le cadre d'applications distribuées, alors la situation est tout autre. Considérons une série d'opération à effectuer sur une machine distante. Chacune de ces opérations nécessite l'envoi de données et amène un résultat en retour. Le problème est très simple : vaut-il mieux effectuer ces opérations en utilisant un mécanisme d'appel de procédure à distance ou envoyer un agent sur le serveur pour effectuer toutes les opérations et ne ramener que le résultat ? La réponse n'est pas triviale car pour la trouver il faut tenir compte de la latence et du débit du réseau, de la puissance de calcul de la machine et par là même de la durée de chacune des opérations. Dans [33], les auteurs montrent à travers des expérimentations sur Internet que pour une certaine quantité d'opérations à effectuer, l'appel de méthode donne de meilleures performances, mais qu'à partir d'un certain point, il vaut mieux envoyer un agent faire le traitement sur le site distant.

2.2.5 Tous ensemble : exemple d'application mobile

La force de la mobilité vient qu'elle est adaptée à la résolution de plusieurs problèmes, ce qui est illustré dans la petite histoire de l'introduction de cette thèse. Monsieur Dupont a programmé un agent pour qu'il cherche le meilleur billet sur une série de sites (*Recherche d'informations*). Son agent est autonome et il n'a donc pas à rester connecté le temps de la recherche (*Clients intermittents*) et effectue cette tâche plus rapidement (*Amélioration des performances*).

Nous allons maintenant étudier différents mécanismes permettant d'assurer les communications avec des objets mobiles et tenter une classification de ceux-ci nous permettant d'isoler certaines de leurs propriétés.

2.3 Communication et localisation

La mobilité introduit des contraintes sur les communications qui n'étaient pas présentes avec les systèmes distribués classiques où les objets ne pouvaient pas changer de lieu d'exécution. En particulier, il faut que des entités mobiles puissent communiquer indépendamment de leur localisation et de leur mobilité. Le point le plus important concerne la fiabilité, il faut pouvoir assurer les communications où que se trouvent les objets mobiles. Nous allons dans cette section tenter une description et une classification des mécanismes de localisation dans les systèmes à objets mobiles. Nous allons dans un premier temps définir précisément les entités entrant en jeu dans une communication, puis nous décrirons et classerons les mécanismes permettant la communication, et finalement nous donnerons des critères permettant de choisir certains plutôt que d'autres.

2.3.1 Communications directes, communications indirectes

Assurer des communications entre deux entités peut se faire de deux façons ; la plus intuitive consiste à avoir un mécanisme permettant une communication directe entre les objets, à la manière des appels de méthode classiques ou RPC, ce qui se prête parfaitement aux communications synchrones ou asynchrones avec rendez-vous. Une deuxième manière de procéder est d'avoir des mécanismes de communication indirecte. Un objet voulant communiquer envoie son message à un objet tiers qui a la charge de le faire suivre au destinataire. Il est aussi possible d'utiliser l'infrastructure réseau pour acheminer le message en utilisant des mécanismes de broadcast ou de multicast par exemple.

2.3.2 Taxonomie des mécanismes de communication

Des études, en particulier [44] ont suggéré de classer les mécanismes de localisation en quatre familles : mise à jour sur le site d'origine (*updating at home*), enregistrement (*registering*), recherche (*searching*) et poursuite (*forwarding*). La première consiste à avoir un objet sur le site d'origine de l'agent qui est chargé de donner la localisation la plus récente de ce dernier. Dans un cadre plus général, il est possible d'utiliser un serveur pour effectuer cette tâche, ce que les auteurs ont choisi de nommer enregistrement. Une approche plus directe consiste à rechercher l'objet dans tous les endroits où il pourrait se trouver. Et finalement, il est possible de suivre l'agent si celui-ci laisse des pointeurs indiquant où il est parti. Notons que cette classification ne tient compte que des mécanismes de localisation proprement dit et qu'elle exclut donc toutes les méthodes permettant d'assurer la communication de manière indirecte par l'utilisation de tiers ou de l'infrastructure. Il nous a semblé donc nécessaire d'introduire une classification plus générale des mécanismes de communication dans les systèmes à agents mobiles, classification qui indiquerait comment un message atteint un agent mobile plutôt que la façon dont l'agent est localisé. Dans [72] une classification se rapprochant est décrite mais présente un découpage qui nous a semblé trop fin. Par exemple une distinction est faite entre un serveur unique qui maintiendrait une liste de positions connues pour les agents et un ensemble hiérarchique de serveur avec la même fonction. Nous ne pensons pas qu'il soit nécessaire d'entrer aussi avant dans les détails c'est pourquoi nous proposons une classification en trois familles.

De l'étude de divers systèmes à agents mobiles nous avons dégagé les méthodes suivantes : la poste restante, la recherche et le routage. Avant de donner une description détaillée de chacun de ces mécanismes, nous allons en donner une brève définition.

Poste restante Les messages envoyés à un agent sont conservés dans un endroit fixe en attente d'un retrait par l'agent auxquels ils sont destinés.

Recherche Un objet voulant communiquer avec un agent cherche une référence explicite pour lui envoyer le message.

Routage L'acheminement du message se fait de manière transparente pour l'appelant qui n'a pas forcément de référence vers l'agent.

2.3.2.1 Poste restante (ressource partagée, espace commun)

Pour assurer une communication avec un agent mobile, un objet peut confier son message à un objet *tiers*, à la manière de la poste restante. Celui-ci va le stocker en attendant que l'agent en prenne connaissance. Le premier et le plus connu des instances de cette famille est la structure de donnée partagée. Les objets y posent des variables, souvent référencées par des clefs pour que d'autres puissent y accéder. Messenger [42] utilise un système de "dictionnaires" pour permettre aux messengers de partager des variables. Cela ne marche bien sûr que s'ils sont dans le même espace d'adressage. Dans le cas de communications distantes, un messenger contenant la variable à mettre à jour est créé et envoyé sur la machine distante. Notons qu'aucun système de localisation n'est fourni permettant aux messengers de se retrouver. Il s'agit donc ici de relations fortement découplées que l'on retrouve aussi dans les espaces de tuples à la Linda [23, 22, 16]. Les mobiles ambients [17] utilisent leurs conteneurs pour lire et écrire des informations ce qui leur permet de communiquer localement; dans le cas de communications distantes la même solution que pour les messengers est utilisée.

Une approche plus générale consiste à utiliser une boîte à lettres où sont stockés les messages à destination d'un agent qui en prend connaissance périodiquement.

2.3.2.2 Recherche

Communiquer avec un agent peut nécessiter de connaître explicitement sa localisation ce qui implique souvent de le rechercher. Il s'agit donc de communications en deux étapes, une phase de recherche et une phase de communication proprement dite. La solution la plus simple à mettre en œuvre est d'avoir un serveur de localisation chargé de fournir sur demande la dernière position connue de l'agent. La mise à jour du serveur est à la charge de ce dernier. Lorsqu'un objet veut communiquer avec l'agent et que celui-ci a changé de localisation, il demande une nouvelle référence au serveur. Il est nécessaire pour mettre en œuvre ce mécanisme d'avoir un identifiant unique pour les objets. Nous reviendrons plus en détail sur ce mécanisme dans la section 3.5.2. Notons que plusieurs variantes de ce mécanisme existent suivant le lieu où est situé le serveur de localisation [44], son architecture (distribué sur des domaines, hiérarchique...) [24, 44, 66].

Une autre méthode consiste à chercher explicitement la localisation de l'agent auprès des sites où il pourrait se trouver. Si l'ensemble des sites est connu alors il est possible d'utiliser des communications point-à-point avec chacun d'entre eux. Partant de cette idée, un algorithme de localisation a été proposé dans [20] qui tire parti de la distribution des temps d'exécution de l'agent sur chacun des sites. Le principe est le suivant : si en moyenne un agent passe δ_t unités de temps sur un site et qu'il n'a pas été contacté depuis T unités de temps, alors il est facile d'avoir une estimation du nombre de sites qu'il a pu visiter. Si N est le nombre de sites dans l'itinéraire, alors, cette estimation est donnée par $min(N, \lfloor \frac{T}{\delta_t} \rfloor)$. Pour localiser l'agent avec la plus grande probabilité, il faut donc demander au site retourné par notre estimateur. Si l'agent n'y est pas, il faut soit chercher plus en amont, soit plus en aval suivant les indications données par la machine contactée.

Pour la bibliothèque Sumatra [2] une approche plus originale est choisie. Quand une communication échoue parce qu'un objet a migré, l'appelant reçoit une erreur contenant une nouvelle référence vers l'objet. Cette référence pointe vers le site distant d'où l'agent peut déjà être parti. L'appelant peut utiliser cette nouvelle référence pour soit appeler l'objet, soit migrer sur son site.

Dans le cas où la liste des sites n'est pas connue il est toujours possible d'avoir recours à la diffusion (broadcast). Cependant, le coût de ce mécanisme le réserve souvent aux systèmes où la topologie est connue ou bien maîtrisée ce qui est le cas dans les systèmes distribués tel que Emerald [35].

Le point commun à tous les mécanismes de communication à base de recherche est qu'entre le moment où l'objet est localisé et celui où la communication a lieu, il est possible que l'agent se déplace à nouveau. Il peut donc être nécessaire de faire plusieurs recherches successives avant de réussir une communication, et dans le cas où l'agent se déplace très rapidement, il n'y a aucune garantie quant à la réussite de la communication.

2.3.2.3 Routage

Nous allons voir dans cette dernière catégorie des mécanismes qui permettent d'envoyer un message à un agent sans avoir de référence directe ni sur lui ni sur une boîte à lettres. Ils fonctionnent à la manière des approches utilisées dans les réseaux de communication tels que ceux basés sur IP ou ATM. Le principe est d'utiliser l'infrastructure logicielle ou matérielle pour permettre l'acheminement d'un message jusqu'à l'agent. Nous allons distinguer deux sous catégories, le routage unicast et le routage multicast.

Le plus connu des mécanismes de communication par routage unicast est le mécanisme des répéteurs, appelés aussi "forwarders" [29, 70], pointeurs de poursuite, dépendances résiduelles [58] ou paires souche-scion [57]. Tout agent quittant un site laisse derrière lui un objet qui est chargé de faire suivre les messages vers le site où il est parti. Un agent migrant plusieurs fois créera une chaîne de répéteurs entre la source et lui. Il y a donc un chemin virtuel qui se construit entre une source et un agent, et c'est ce chemin que vont emprunter les messages envoyés [70, 12, 35]. Il existe plusieurs variantes de ce mécanisme dont une que nous étudierons en détails dans la section 3.5.1. Il n'existe à notre connaissance qu'une étude formelle de ce mécanisme, étude qui peut être trouvée dans [28]. L'auteur s'est intéressé à trois variantes des répéteurs et en a étudié la complexité en terme de nombre d'opérations à effectuer pour réussir une communication.

Comme dans le cas de la recherche, le broadcast est utilisé dans les systèmes d'exploitation distribués pour envoyer un message à un objet qui a migré comme, par exemple, dans le V-System [40]. Il est aussi possible de faire du multicast à un niveau applicatif comme suggéré par [47]. Les auteurs décrivent un protocole permettant de communiquer avec un agent très mobile; dans ce cas, il est important d'avoir un mécanisme garantissant que la communication aura bien lieu. En se basant sur les résultats issus des techniques de snapshot distribué, les auteurs inondent le graphe obtenu en reliant toutes les destinations possibles de l'agent. Ces destinations peuvent être connues ou être découvertes au fur et à mesure au prix d'une complexification importante du protocole.

Notons que dans les deux situations, unicast et multicast, des difficultés supplémentaires viennent du fait que la topologie n'est pas connue et peut changer au cours du temps.

La table 2.1 résume les propriétés des systèmes ou algorithmes étudiés dans cette section. Nous indiquons si le mécanisme choisit permet d'assurer les communications dans toutes les conditions (fiabilité), passe bien à l'échelle et si il intègres des notions de sécurité.

2.3.3 Critères de choix

Choisir un mécanisme de communication dépend des contraintes rencontrées et des propriétés que l'on veut avoir. Nous donnons ici une liste non exhaustive de critères permettant de privilégier une des trois approches décrites précédemment. Pour chacun de ces critères nous indiquons, lorsque c'est possible, la meilleure approche.

Type de réseau Le débit du réseau, sa latence et sa structure vont déterminer en particulier si il est possible d'utiliser du multicast pour joindre un agent. Un réseau local se prête bien au multicast ou au broadcast ce qui est plus difficile à faire sur un réseau très ouvert comme Internet.

Sémantique des communications Dans le cas de la poste restante, la sémantique privilégiée est l'asynchrone. Le routage peut être utilisé pour faire du synchrone mais est difficile à mettre en œuvre si un rendez-vous est nécessaire.

Fiabilité La poste restante et la recherche en unicast introduisent tous les deux un point unique de défaillance ce qui les rend peu résistants aux pannes. La même remarque peut s'appliquer au routage unicast comme les répéteurs.

Ressources Les ressources mises en œuvre ne dépendent pas simplement de la famille de communication choisie. Ainsi, l'utilisation de tuples (poste restante) consommera probablement plus de ressources qu'un serveur (recherche), alors que le contraire sera vrai si nous comparons une boîte à lettres avec ce même serveur. Les répéteurs (routage) nécessitent d'utiliser des ressources sur tous les sites visités ce qui peut s'avérer problématique.

Vitesse de l'agent Un facteur limitant pour le routage et la recherche est la vitesse de déplacement de l'agent. Procédant en deux étapes, la communication par recherche présente une grande sensibilité à la vitesse de l'agent car si celui-ci migre entre le moment où un objet l'a retrouvé et le moment où il communique, alors il faut de nouveau le chercher. Dans le cas du routage multicast, nous avons vu un algorithme pour garantir la communication sous certaines conditions [47].

Passage à l'échelle La recherche peut devenir un point de ralentissement important si par exemple un unique serveur traite des centaines de requêtes. Les mécanismes de routage devraient quant à eux relativement bien passer à l'échelle.

Sécurité Il s'agit là d'un problème complexe qui dépasse largement le cadre de cette thèse. Nous n'allons donc pas entrer dans les détails, mais juste remarquer que les techniques de routage, du fait de leur nature, font intervenir des communications entre plusieurs machines, qui peuvent se situer dans des domaines de sécurité différents. Il peut donc être difficile d'utiliser ce genre de mécanisme lorsque la sécurité est une propriété importante. Dans le cas de la recherche, des solutions sécurisées ont été proposées, en particulier pour les serveurs [54].

Performance Si les performances, i.e le temps minimum pour joindre un agent, sont à considérer, alors la poste restante ne devrait pas être utilisée. En effet, la vitesse de transmission d'un message dépend directement de la fréquence à laquelle l'agent vérifie sa boîte. Il est difficile de départager intuitivement le routage de la recherche c'est pourquoi une partie de cette thèse est consacrée à cette question. Nous étudierons formellement une instance de routage et une instance de recherche dans la section 4 pour déterminer leurs performances respectives.

Nous voyons donc sans surprise qu'il n'y a pas de solution miracle qui donnerait les meilleurs résultats pour l'ensemble de nos critères. De cette simple remarque est née l'idée de créer des mécanismes hybrides, mélangeant le routage et la recherche pour obtenir un système à la fois performant et robuste. Nous présenterons dans le dernier chapitre de cette thèse un tel algorithme.

2.4 Java, outils pour la distribution et la mobilité

Java est un langage orienté objet développé par Sun et présenté officiellement en 1995. Les applications écrites en Java sont compilées en *bytecode* et exécutées sur une

	Famille	Communication	Fiabilité [1]	Passage à l'échelle	Sécurité [1]
Messenger [42]	Poste	Asynchrone	non	**oui**	non
Jada [22]	restante	Asynchrone	non	non [2]	non
Ara [50]	"	Asynchrone	non	non [2]	non
MOA [44]	Recherche	Synchrone	**oui** [3]	**oui** [3]	non
D'Agents [31]	"	Synchrone	non	non	non
Chen et al. [20]	"	Synchrone	**oui**	incertain [4]	non
Roth [54]	"	Synchrone	**oui** [5]	**oui** [5]	**oui** [6]
Amoeba [58]	Routage	Asynch. avec rdv	non	**oui**	non
Charlotte [6]	"	Asynch. avec rdv	non	**oui**	non
Voyager [70]	"	Asynch. avec rdv	non	**oui**	non
Mole [12]	"	Synchrone	**oui** [7]	**oui**	non
Murphy et al. [47]	"	Synchrone	**oui** [8]	non [9]	non
Moreau [46]	"	Synchrone	**oui** [10]	**oui**	non
Emerald [35]	Recherche	Synchrone	**oui**	**oui**	non
Nomadic Pict[73]	- routage	Synchrone	**oui**	**oui**	non
ProActive [52]	"	Asynch. avec rdv	**oui**	**oui**	non

[1] ne concerne que le protocole de communication
[2] le tuple est un unique serveur
[3] hiérarchie de serveurs
[4] la nature probabiliste de l'algorithme fait qu'il est difficile de prévoir son comportement
[5] multiples serveurs
[6] encryption, authentification
[7] raccourcissement de chaîne périodique
[8] inondation
[9] protocole coûteux
[10] redondance dans les répéteurs

TAB. 2.1 – *Propriétés des mécanismes de communication* machine virtuelle, la Java Virtual Machine (JMV). Le succès de ce langage vient pour beaucoup du fait qu'il est disponible sur de multiples plateformes qui vont des ordinateurs personnels sous Linux ou Windows et des stations de travail sous Solaris aux assistants personnels [61]. Cette propriété en fait une plateforme de développement de choix pour les applications mobiles ou distribuées car le programmeur n'a pas à gérer différents langages ou différentes versions d'un programme suivant les systèmes sur lequel il va s'exécuter.

Avec Java sont fournis toute une série de services qui permettent de construire des applications distribuées ou mobiles. Nous allons détailler maintenant quelques unes de ces fonctionnalités.

2.4.1 Sérialisation

La sérialisation [63] est un mécanisme introduit dans Java 1.1 qui permet d'écrire (resp. lire) un objet dans (resp. depuis) un flux d'octets. Il permet aussi bien de stocker un objet java dans un fichier que de l'envoyer sur le réseau. Pour pouvoir être sérialisé, un objet doit implémenter l'interface *java.io.Serializable* qui est en fait une interface marqueur ne contenant aucune méthode. La figure 2.1 donne un exemple de code Java qui permet d'écrire et de lire un objet java depuis un fichier.

```
MyObject toto = new MyObject();
ObjectOutputStream objStream = null;
// Ouverture d'un fichier pour écrire notre objet
try {
   objStream = new ObjectOutputStream(new FileOutputStream(new File(``toto''))); 
 } catch(IOException e) {
     e.printStackTrace();
 }
//écriture de l'objet dans le fichier
objStream.writeObject(toto);
//lecture d'un objet depuis le fichier
MyObject titi = (MyObject) objStream.readObject();
```

FIG. 2.1 – *Sérialisation et dé-sérialisation d'un objet Java*

La sérialisation (resp désérialisation) est effectuée grâce à la méthode *writeObject()* (resp. *readObject*). Celle-ci sérialise l'objet puis parcourt récursivement tous les objets référencés pour créer une représentation complète du graphe d'objets. La première fois qu'un objet est traversé par la sérialisation, il est sérialisé et un *handle* lui est associé. Si au cours de cette même sérialisation cet objet est de nouveau traversé (soit parce qu'il apparaît plusieurs fois dans le graphe, soit parce qu'il y a un cycle), alors une référence vers le *handle* et non l'objet lui-même est écrite. Cette technique permet de préserver toutes les références tout en évitant les cycles et en ayant une représentation aussi compacte que possible. Bien que nous n'ayons parlé que de sérialisation d'objets, ce mécanisme marche aussi pour les types primitifs et les tableaux.

Il est possible pour un objet de contrôler la sérialisation de trois manières, de la plus générale à la plus détaillée. Tout d'abord il peut indiquer les champs qui ne doivent pas être sérialisés en utilisant le mot clef *transient*. Ensuite, il peut contrôler précisément la sérialisation de ses champs en redéfinissant la méthode *writeObject()*. Enfin, un objet peut demander à fournir un remplacement grâce à la méthode *writeReplace*. Celle-ci est appelée avant que la sérialisation de l'objet ne commence ce qui permet par exemple de remplacer un objet par un autre.

2.4.2 Appel de méthode distant (Remote Method Invocation)

La communication entre deux objets situés sur des machines distantes se fait en utilisant le réseau comme médium. Il est possible d'utiliser explicitement des sockets et d'écrire un protocole de communication directement au niveau applicatif, mais cela est complexe et source d'erreurs. Une alternative existante est connue sous le terme de Remote Procedure Call (RPC) [65] qui abstrait la couche réseau et permet au programmeur de ne travailler qu'à travers des appels de méthode. Un appel de méthode sur un objet distant est automatiquement envoyé à travers le réseau, RPC se chargeant d'encoder les arguments et la valeur de retour. Sun a créé pour le langage Java un équivalent de RPC orienté objets, appelé Remote Method Invocation (RMI) [64].

2.4.2.1 Principe, architecture

RMI fonctionne en utilisant des objets d'interposition qui se placent entre les objets et le réseau. Ils sont responsables de l'échange de données avec le réseau, aussi bien du côté client que du côté serveur grâce à des objets appelés *stub* et *skeleton* générés par le compilateur *rmic* comme indiqué dans la figure 2.2. Pour être accessible par RMI, un

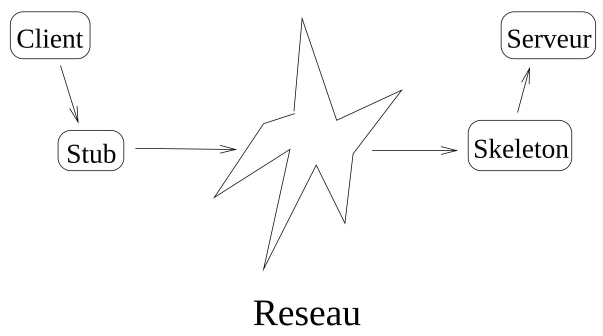


FIG. 2.2 – *Architecture générale d'une application RMI*

objet doit hériter de la classe *java.rmi.server.UnicastRemoteObject* et implémenter une interface qui décrit les services accessibles à distance. Cette interface hérite elle-même de *java.rmi.Remote*. La programmation distribuée introduit une nouvelle composante dont le programmeur doit tenir compte : les communications entre objets peuvent maintenant échouer à cause de problèmes réseau. Pour gérer cela, toutes les méthodes accessibles à distance doivent pouvoir lever une *java.rmi.RemoteException* qui est la super-classe de toutes les exceptions qui peuvent être levées par RMI. Elle permet de tenir compte au niveau applicatif des problèmes pouvant survenir au niveau réseau. De ce point de vue, RMI n'est pas totalement transparent pour le programmeur.

RMI utilise la sérialisation décrite précédemment pour faire circuler les objets entre les machines virtuelles, ce qui implique que toute classe paramètre d'un appel ou type de retour doit être sérialisable.

2.4.2.2 Chargement dynamique de code

De nombreux services sont disponibles dans RMI afin de faciliter la construction et le déploiement d'applications distribuées. Un de ces services est le chargement de code dynamique qui permet de distribuer des classes à l'exécution. Si dans le cas d'un appel RMI une des classes utilisées n'est pas disponible sur la machine distante, alors elle sera automatiquement chargée. Cela permet à un serveur de traiter des appels référençant des classes dont il ne dispose pas localement, facilitant ainsi le déploiement d'applications.

2.4.2.3 Le rmiregistry

Pour qu'une application distribuée fonctionne il faut que les objets situés sur des sites distants puissent communiquer, ce qui implique de pouvoir les mettre en relation. Le *rmiregistry* a cette fonction; il agit comme un serveur qui centralise des références vers des objets distants. Pour s'enregistrer, un objet RMI utilise au choix la méthode *bind()* ou *rebind()* qui permettent d'enregistrer une référence distante sous un nom symbolique dans le registre. Un client à la recherche d'un objet pourra demander une référence au registre en utilisant *loookup*. La principale limitation de ce mécanisme est que pour des raisons de sécurité, il faut que le serveur et le rmiregistry soient sur la même machine.

2.4.3 Ramasse-miettes

Java utilise un ramasse-miettes pour gérer la mémoire mais les spécifications [62] n'imposent pas de mécanisme particulier. Dans le cas non distribué, il est souvent fait usage d'un ramasse-miettes qui marque les objets qui ne doivent pas être supprimés. Le ramasse miette commence par enlever tout marquage aux objets, puis en partant des racines (les *threads* par exemple), il parcourt le graphe d'objets et marque tout objet qu'il rencontre. Une fois cette étape terminée, tout objet non marqué peut être supprimé. Nous voyons donc qu'il faut qu'un objet soit non accessible depuis les racines pouvoir être supprimé de la mémoire. RMI a légèrement compliqué le problème puisqu'il faut maintenant considérer les références distantes. Un deuxième ramasse-miettes (Distributed Garbage Collector - DGC) a été ajouté pour traiter ces cas et fonctionne sur le même principe que celui de Modula-3 [13]. Ce DGC fonctionne en totale coopération avec le GC local. Il maintient un compte des références sur un objet, quand ce compte atteint zéro, alors l'objet peut être ramassé. Quand un client obtient une référence sur un objet distant, ce dernier voit son nombre de références entrantes augmenté de un. Une fois que le client a fini d'utiliser cette référence, il notifie le serveur qui décrémente le compteur. Quand celui-ci atteint zéro, le *runtime* RMI supprime la référence qu'il avait sur l'objet qui peut donc être ramassé par le ramasse-miettes local. Plus précisément, un client obtient une référence pour un bail (*lease*) de durée définie dans la propriété *java.rmi.dgc.leaseValue* qui est par défaut de dix minutes. Il est de la responsabilité du client de renouveler ce bail avant son expiration s'il désire continuer à utiliser l'objet. Dans le cas où le client ne renouvelle pas celui-ci, alors une expiration de bail est notifiée du côté du serveur. Notons que cette propriété, bien qu'utile au client est en fait fixée par le serveur. Quand un client n'a plus besoin d'une référence, alors il doit notifier le serveur. Le fait qu'un client n'ait plus besoin d'une référence distante est en fait décidé par le GC local. Pour limiter les communications, RMI limite la fréquence de fonctionnement du GC grâce à la propriété *sun.rmi.dgc.client.gcInterval* qui est fixée à une minute. Le serveur doit quant à lui vérifier que les baux qu'il a donné ne sont pas expirés. L'intervalle entre deux vérifications est défini par la propriété *sun.rmi.dgc.checkInterval*. Nous avons donc trois propriétés pour influencer le ramasse-miettes, que nous résumons dans la table 2.2.

Propriété	Description	Valeur par défaut
sun.rmi.dgc.client.gcInterval	Fixe le temps moyen entre deux fonctionnements du GC	3 minutes
java.rmi.dgc.leaseValue	Durée maximale d'un bail pour un client	10 minutes
sun.rmi.dgc.checkInterval	Intervalle de vérification de l'expiration des baux	5 minutes

TAB. 2.2 – *Propriétés de base du DGC de RMI*

2.5 La bibliothèque ProActive

Les modèles à *objets actifs* proviennent de l'unification des notions d'objet et d'activité, de la même manière que le concept d'objet lui-même provient de l'unification des notions de structure de données et de procédure. Un objet actif se conforme donc à la règle des trois unités : unité de données (objets), unité de code (classes) et unité d'activité (threads).

Dans un souci de portabilité, la bibliothèque ProActive s'interdit toute modification du langage Java et de sa machine virtuelle (JVM, pour *Java Virtual Machine*). Nous sommes donc en présence d'une simple bibliothèque, qui utilise les outils standards (`javac` et JVM). Pour des raisons de debuggage des programmes (problème exacerbé en l'occurrence car ils sont répartis et parallèles), ProActive s'interdit également la modification du code source écrit par l'utilisateur (pas de pré-traitement). La seule technique utilisée consiste, outre l'utilisation de techniques de réflexion permettant de manipuler les événements de l'environnement d'exécution (appels de méthodes distantes par exemple), à générer (éventuellement dynamiquement) du code complémentaire à celui de l'utilisateur; technique qui s'apparente à la notion de bibliothèque *générative* ou *active* [69, 25].

De plus, la bibliothèque est elle-même écrite entièrement en Java, ce qui autorise l'utilisation de n'importe quelle plate-forme Java et des outils associés, en particulier le fonctionnement dans un environnement ouvert et dynamique avec télé-chargement dynamique de code ; nous utilisons le mini-serveur `http` de la plate-forme Java pour transférer dynamiquement le bytecode nécessaire.

2.5.1 Modèle de base

ProActive offre un modèle de programmation parallèle et répartie que l'on peut qualifier de "*à objets actifs*" [15] (par opposition à des approches où les activités sont orthogonales aux objets), et de "*hétérogène*" dans le sens où tous les objets ne sont pas actifs. Un des objectifs est de faciliter la programmation répartie, en particulier par le biais de la réutilisation [19].

Le modèle de *ProActive* présente les caractéristiques suivantes:

- des objets actifs et accessibles à distance,
- la séquentialité des activités (processus purement séquentiels),
- une communication par appel de méthode standard,
- des appels asynchrones vers les objets actifs,
- un mécanisme d'attente par nécessité (futur transparent),
- les continuations automatiques (un mécanisme transparent de délégation),
- l'absence d'objets partagés,
- une programmation des activités qui est centralisée et explicite par défaut,
- du polymorphisme entre objets standards et objets actifs distants.

Un *appel asynchrone* permet de ne pas se bloquer lors d'une communication. Un *futur* est alors un objet résultat d'un appel asynchrone : initialement il n'a pas encore de valeur. D'une façon duale, l'*attente par nécessité* permet de se bloquer automatiquement si l'on tente d'utiliser le résultat non encore revenu d'un appel asynchrone. Un futur est *transparent* dans le sens où le programmeur n'a pas besoin d'ajouter du code pour obtenir ce type de synchronisation.

2.5.2 Structure d'un objet actif

D'un point de vue macroscopique, la structure d'un objet actif peut être vue comme étant à deux niveaux. Au niveau de base se trouve l'objet qui a été rendu actif. Au dessus se trouve le niveau méta où se trouvent toutes les propriétés non fonctionnelles qui ont été rajoutées au moment de la création de l'objet actif. Dans le prochain chapitre nous reviendrons sur la structure des objets actifs en adoptant cette fois un point de vue microscopique.

Nous allons maintenant présenter succinctement les principales caractéristiques de *ProActive*. Plus de précisions sont disponibles dans [18].

2.5.3 Création des objets actifs

La bibliothèque a pour objectif de permettre la création d'un objet actif distant le plus simplement possible, à partir de code (donc d'une classe) initialement local et séquentiel. Un objet Java standard, créé par une instruction :

```
A a = new A ("foo", 7);
```

peut être transformé en un objet actif distant de trois manières (Exemple 2.3).

Du plus statique au plus dynamique, **a) Class-based** passe par la création d'une nouvelle classe qui implémente l'interface-marqueur `Active` et hérite (éventuellement) d'une classe existante. Cette interface marqueur pure, ne comporte aucune routine et n'impose donc aucune contrainte d'implémentation au programmeur. Ce cas permettra d'ajouter des méthodes spécifiques aux aspects répartis, et également de doter l'objet actif d'une activité propre, en place de l'activité FIFO réalisée par défaut (voir plus loin section 2.5.5). Le second paramètre de cet appel statique (`params`) correspond aux paramètres effectifs à passer au constructeur lors de la création distante. Dans notre exemple, il peut être défini par :

```
Object[] params = {"foo", new Integer (7)};
```

Afin d'éviter cette syntaxe, que nous admettons volontiers peu élégante, un motif de conception de type *factory* peut être implémenté dans la classe `pA` (voir [18]). Le dernier paramètre (`Node`) spécifie la machine virtuelle où doit être placé l'objet actif (c'est habituellement une URL de la forme `//lo.inria.fr/VM1`). Dans le cas d'un node `null`, la création se fait dans la JVM en cours. Un autre objet actif passé en paramètre à la place du paramètre node (`newActive` est surchargée) permet la co-allocation (même machine virtuelle) avec un objet actif déjà existant:

```
A a0= (A) ProActive.newActive("pA",params,null); // JVM en cours
A a1= (A) ProActive.newActive("pA",params,"//lo.inria.fr/VM1");
A a2= (A) ProActive.newActive("pA",params,a1);   // co-allocation
```

La seconde primitive de création, **b) Instanciation-based**, permet de prendre une classe existante standard (qui n'implémente pas l'interface `Active`) et de l'*instancier* directement (sans la modifier) en un objet actif distant. Les deux paramètres (`params` et `Node`) ont bien sûr le même rôle que précédemment.

Enfin, le dernier type de création, **c) Object-based**, est encore plus dynamique puisqu'il prend un objet déjà créé, et en fait un objet actif accessible à distance. La sémantique est la suivante. Tout d'abord, si le code nécessaire à un accès distant sur ce type d'objet n'existe pas encore, il est généré dynamiquement. Si `node` est la JVM en cours, les éléments nécessaires à la création d'un objet actif sont ajoutés à l'objet en question (une activité propre, une file d'attente, etc.). Si `node` n'est pas la JVM en cours, une copie de l'objet en question est transmise à la JVM `node`, et les éléments mentionnés ci-dessus sont ajoutés dans la JVM de cette copie. Dans tous les cas, l'objet original passif reste en place. Cette technique permet entre autre la répartition de code pour lequel on ne dispose pas du source.

a) Class-based: `class pA extends A implements Active {}`

```
...
A a = (A) ProActive.newActive ("pA", params, Node);
```

b) Instanciation-based: `A a = (A) ProActive.newActive ("A", params, Node);`

c) Object-based: `a = (A) ProActive.turnActive (a, node);`

FIG. 2.3 – *Création des objects actifs: Class-, Instanciation-, Object-based*

2.5.4 Asynchronisme, polymorphisme et futurs

Les trois primitives de création des objets actifs retournent un objet qui est *polymorphiquement compatible* avec le type d'origine. Cela permet d'affecter l'objet distant retourné dans une entité du type de création. Ainsi, il sera possible de réutiliser dans un cadre réparti du code prévu initialement pour fonctionner avec des objets locaux et non-actifs. Le programme de l'exemple 2.4 met en œuvre ce type de réutilisation.

Tout d'abord, selon le principe bien connu de la programmation répartie à objets, les invocations de méthodes jouent également le rôle d'IPC [1], et la cible d'un appel distant est elle-même un objet. La classe standard `A` présente trois méthodes publiques `foo`, `bar` et `bar2`. Celles-ci vont pouvoir être utilisées à distance, mais la sémantique de communication associée sera différente pour chacune d'entre elles. Intéressons-nous tout d'abord à la première méthode, `void foo(...) (1)`. Celle-ci ne retourne aucun résultat, elle ne donnera donc lieu qu'à une communication de l'appelant vers l'appelé. Nous dirons qu'elle est `one-way`. Le deuxième appel qui concerne la méthode `C bar2(...) (2)` sera synchrone, c'est à dire qu'une fois la requête envoyée, l'appelant restera bloqué jusqu'à obtenir une réponse. En effet, cette méthode peut lever une exception, or en continuant l'exécution

1. Inter-Process Communication

```
class A {
  public void foo (...) ...;
  public B bar (...) throws UneException ...;
  public C bar2 (...) ...;
  ...
}
class Client {
  static void useA (A a) {
    a.foo (...);         //  (1)
    try {
      B b = c.bar2(...); //  (2)
    catch (UneException e) {
       e.printStackTrace();
    }
     C c = a.bar (...);  //  (3)
    ...
}
main {
  A a1 = new A();
  Client.useA (a1);  // dans useA: appels standards
  A a2 = (A) ProActive.newActive("A",null,"//lo.inria.fr/VM1");
  Client.useA (a2);  //  Polymorphisme:
}                    // dans useA: appels distants et asynchrones
```

FIG. 2.4 – *Asynchronisme, polymorphisme et futurs*

il y aurait risque de sortir du bloc try-catch. Finalement, la méthode `B bar (...) (3)` retourne un résultat et ne lève pas d'exception, il y aura donc une communication bi-directionnelle. Dans *ProActive* cette communication se fera de manière asynchrone. Dans un premier temps `Client` enverra une requête à B, qui, une fois la méthode `bar` exécutée lui enverra une réponse.

Dans le cas d'un appel asynchrone, l'appelant reçoit un objet appelé futur qui représente le résultat attendu. Quand celui-ci est envoyé par l'appelé, l'objet futur est mis à jour automatiquement. Si l'appelant essaie de l'utiliser avant que le résultat ne soit disponible, alors il sera bloqué par un mécanisme appelé *attente par nécessité*.

Nous résumons dans la table 2.3 le type de communication en fonction des types des paramètres et de la signature de la méthode considérée. Le terme réifiable doit ici être compris dans le sens "peut être sous-classé". En effet, pour pouvoir créer un objet futur nous avons besoin de créer une sous classe de la classe de l'objet. Or il existe deux cas où cela n'est pas possible en Java : la classe est *finale* ou elle appartient au package *java.lang*.

Communication	Conditions
One-way	renvoie `void` et aucune exception déclarée
Synchrone	exception déclarée ou type de retour non réifiable
Asynchrone	aucune exception déclarée et type de retour reifiable.

TAB. 2.3 – *Sémantique de communication suivant la signature de la méthode*

Si le type de retour d'une de nos méthodes remplit au moins une de ces conditions, alors nous ne pouvons pas créer d'objet futur et nous devons faire un appel synchrone.

Dans le cadre de programmes gérant explicitement la répartition et les synchronisations inter-objets, il est possible d'agir directement sur un futur par deux primitives :

```
ProActive. waitFor (Object obj);   // Attente explicite d'un futur
if ProActive. isAwaited (Object obj) ... // Teste l'état d'un futur
```

La seconde primitive permet, suite à un ensemble d'appels asynchrones, de traiter le premier résultat revenu, ou encore de gérer une activité locale en attente d'un résultat.

Comme pour Java RMI et la plupart des systèmes à objets répartis, les paramètres des appels distants sont transmis par *copie profonde*, sauf bien sûr les objets distants eux même qui sont transmis par référence (ce qui confère au système une topologie dynamique). D'autre part, et cette fois-ci contrairement à RMI, on obtient une structuration forte en *sous-systèmes* (un objet actif et tous ses objets passifs) sans aucun objet partagé (objet standard accessible par plus d'un objet actif).

2.5.5 Activités, contrôle explicite et abstractions

Les appels distants étant asynchrones, ils sont mémorisés sous la forme de *requêtes* dans une file d'attente du côté de l'appelé. Ultérieurement, nous dirons que la requête est *servie* par l'objet actif. Jusqu'à présent, nous avons créé un objet actif sans préciser son activité, et par défaut un service FIFO des requêtes était alors réalisé: exécution des requêtes dans l'ordre d'arrivée. Dans ce sens, les objets actifs sont bien des processus purement séquentiels.

La création d'objets actifs de type **a) Class-based** permet de donner à un objet un comportement spécifique : activité propre (qui ne consiste pas à servir des méthodes publiques), activité mixte (services et activité propre), services purs mais de type non-FIFO.

L'exemple 2.5 présente un simple tampon borné. La routine `live` permet à l'utilisateur de spécifier, sous la forme d'une thread explicite, l'activité de l'objet. Lorsque cette routine existe, elle remplace totalement le comportement FIFO fourni par défaut. Le paramètre `myBody` donne accès à tout un ensemble de routines de services permettant de sélectionner la requête à servir. Dans ce cas, nous utilisons `serveOldest`, mais toute une bibliothèque est accessible au programmeur (Exemple 2.6). On y trouve des

```
class BoundedBuffer extends FixedBuffer implements Active {
  live (Body myBody) {
    while (true) {
      if      (this.isFull())   myBody.serveOldest ("get");
      else if (this.isEmpty())  myBody.serveOldest ("put");
      else myBody.serveOldest();
      myBody.waitForNewRequest ();
} } }
```

FIG. 2.5 – *Programmation explicite du contrôle*

```
void serveOldest (); // Sert la plus ancienne requête, bloque
void serveOldest (String s); // la plus ancienne sur  s, bloque
void serveOldest (String s1, String s2);//plus ancienne sur  s1, s2
...
void serveOldestWithoutBlocking (); // service non bloquant
...
void serveMostRecentFlush (String s); // sert plus ancienne,
...                                   // et supprime les autres
void serveOldestTimed (int t); // bloquant pour au plus  t ms
...
void waitForNewRequest (); // Attente non active d'une requête
...
```

FIG. 2.6 – *Routines de service*

services *bloquants*, *non-bloquants*, *temporisés*, etc. mais également des itérateurs sur la file d'attente permettant d'étendre la bibliothèque si nécessaire. Notons `waitForNewRequest` qui permet de programmer une attente non-active. Nous sommes ici en présence d'une programmation de l'activité qui est *explicite*, avec service lui aussi explicite, sans non-déterminisme sur l'ordre des services; dans l'exemple 2.5 si le tampon n'est ni plein ni vide, on impose de servir la requête la plus ancienne. Ce type de programmation est fort utile si l'on souhaite avoir un contrôle fin sur l'activité de l'objet, mais dans certains cas on souhaite programmer à un niveau d'abstraction plus élevé, d'une façon plus déclarative, voire non déterministe. Ce type de programmation *implicite* peut être réalisé en construisant ce que nous appelons des *abstractions de contrôle*. L'exemple 2.7 montre l'utilisation d'une telle abstraction, programmable par l'utilisateur sous la forme d'une autre interface marqueur (*ImplicitActive*) qui permet simplement d'associer une routine de blocage à une routine publique. Nous sommes ici en présence d'une programmation du contrôle que l'on peut qualifier d'implicite, déclarative, à service lui aussi implicite, et non-déterministe (si le tampon n'est ni vide, ni plein, on ne précise pas quelle méthode servir).

```
class BoundedBuffer extends FixedBuffer implements ImplicitActive {
  live (ImplicitBody myBody) {
    myBody.forbid ("put", "isFull");
    myBody.forbid ("get", "isEmpty");
} } }
```

FIG. 2.7 – *Programmation implicite et déclarative du contrôle*

Ce type de construction incrémentale d'abstraction pour le contrôle dénote l'aspect *ouvert* ("Open Implementation" [36]) de la bibliothèque *ProActive*. Cet aspect va se révéler particulièrement important pour la construction d'un service de migration (cf. section 3.2.2).

2.5.6 Propriétés

Pour conclure cette présentation rapide, attardons-nous sur quelques propriétés qui vont jouer un rôle important pour la migration.

Tout d'abord, bien que la sémantique de la communication soit asynchrone, nous suivons un protocole qui impose un rendez-vous entre l'appelant et l'appelé pour transmettre la requête. Cette stratégie apporte plusieurs propriétés importantes:

– après une instruction de communication, l'appelant est certain que la requête est bien dans le contexte de l'appelé (assure une sémantique "*at most one*"),
– les requêtes ne peuvent pas se doubler; l'ordre d'émission est identique à l'ordre de réception
– propriétés identiques pour les retours de résultat qui suivent le même protocole.

D'autre part, l'attente par nécessité d'une façon intrinsèque apporte une synchronisation fortement dirigée par les données. Par analogie à un modèle "*data-flow*" dans un cadre fonctionnel, nous pourrions parler d'"*object-flow*". Un objet actif se bloque principalement en attente du retour d'un appel asynchrone qui se trouve être représenté par un objet. De plus, dynamiquement nous connaissons dans ce cas l'objet actif qui peut le débloquer. Ceci permet par exemple de détecter dynamiquement un grand nombre de blocages.

Si notre infrastructure permet de répartir du code pour lequel on ne dispose pas du source, certains éléments de celui-ci peuvent engendrer quelques difficultés. Tout d'abord les threads Java classiques font perdre aux objets actifs leur caractère purement *séquentiel*. Sans être rédhibitoire, transformer un objet multi-threadé en un objet actif nécessitera une attention particulière. L'utilisation d'objets distants *RMI* nécessite également un traitement particulier. Cependant, des objets actifs et des objets *RMI* peuvent très bien collaborer dans la même application ; nous avons largement expérimenté cette situation. Le danger majeur réside dans le caractère synchrone des objets *RMI* qui peuvent, si l'on n'y prend garde, créer facilement des inter-blocages.

Chapitre 3

Un cadre extensible pour la mobilité

Dans ce chapitre nous allons nous intéresser à la conception et à l'implémentation d'un mécanisme de migration faible. Notre plate-forme de développement est la librairie *ProActive* présentée dans le chapitre précédent. Nous allons tout d'abord détailler l'API développée et les choix d'implémentation effectués. Ensuite, nous décrirons en détail deux mécanismes permettant d'assurer les communications en présence de migration.

3.1 Migration faible

La conception de la migration dans *ProActive* a été guidée par les fonctionnalités souhaitées, mais également par l'environnement d'exécution choisi (Java et la JVM). Nous avions trois contraintes importantes concernant l'implémentation de la migration : (1) par souci de portabilité nous nous sommes interdit toute modification de la machine virtuelle, (2) comme nous voulons pouvoir travailler à l'exécution nous ne voulions pas utiliser des outils tels qu'un préprocesseur de code source, (3) finalement, en raison des problèmes de sécurité nous nous sommes également interdit de modifier le bytecode des classes au chargement par *classloader* spécialisé. Nous avons donc implémenté la migration en nous basant uniquement sur les techniques fournies par notre protocole à méta-objets, à savoir ajout dynamique de code à l'exécution et interception des appels par polymorphisme. Nous implémentons donc une *migration faible*: un objet actif devra décider de lui-même de migrer — de façon *pro-active*. Il ne sera pas possible de l'extérieur d'un objet de déclencher une migration préemptive sans un acte volontaire de l'objet actif. Nous allons maintenant détailler le fonctionnement des primitives de migration.

3.1.1 Architecture

Du fait qu'un objet actif possède un graphe d'objets passifs non partagés, il est possible de dégager rapidement une unité de migration

Définition 3.1.1 *L'unité de migration d'un objet actif consiste en l'objet rendu actif, ses méta-objets et l'ensemble de ses objets passifs.*

Le mécanisme de migration que nous avons implémenté devra donc sérialiser l'objet actif ainsi que ses objets passifs et les envoyer sur un site distant. Une machine voulant recevoir un objet actif migrant devra fournir un certain nombre de services, en particulier les méthodes pour redémarrer un objet après une migration. L'ensemble de ces services est regroupé dans un objet appelé *Nœud* dont la définition est la suivante :

Définition 3.1.2 *Un* nœud *est un objet accessible à distance permettant la réception et l'exécution d'objets mobiles.*

Nous avons donc une architecture relativement simple et composée de deux types d'objets, les agents mobiles et les nœuds. Nous avons choisi de ne pas utiliser les nœuds pour les communications mais plutôt de privilégier les communications directes entre objets. Il n'y a donc aucun lien entre deux nœuds donnés.

3.1.2 État stable d'un objet actif

Une technique souvent mise en oeuvre pour la mobilité faible consiste à avoir des points bien précis de l'exécution où l'état d'un objet est sauvegardé. De tels points sont appelés *checkpoint* et permettent de redémarrer l'exécution de l'objet à ce point. La définition même d'un objet actif fournit naturellement et implicitement un système de *checkpoint*. En effet, l'état d'un objet actif est caractérisé par sa liste de requêtes, sa liste de futures et l'état de son activité. Entre deux services de requêtes, nous connaissons parfaitement l'état de l'objet et nous savons aussi que ses listes de requêtes et de futures ne peuvent changer qu'à cause de messages ou de réponses envoyés par d'autres objets. Si nous bloquons ces communications, alors nous avons un état stable de notre objet. C'est précisément cet état que nous allons sérialiser. Le principal problème de cette approche est que tout code suivant la demande de migration ne sera pas exécuté. De plus l'objet reprendra son activité comme si il venait d'être créé, avec néanmoins son état sauvegardé. Nous reviendrons sur ce point dans l'étude de l'API.

3.1.3 Conservation de la sémantique

Sans migration, nous garantissons un certain nombre de propriétés, en particulier par l'utilisation d'une phase de rendez-vous pour la transmission des messages (cf. section 2.5.6). Maintenir un tel rendez-vous en présence de migration est un choix relativement coûteux car cela implique de propager ce rendez-vous sur plusieurs machines virtuelles, quelle que soit la technique de communication utilisée. Néanmoins, nous avons opté pour cette solution afin de privilégier la correction par rapport aux performances.

Une situation délicate arrive lorsqu'un objet essaie de communiquer avec un autre objet qui est en train d'effectuer une migration. Il faut être ici très clair concernant la localisation d'un objet en train de migrer : tant que la migration n'est pas finie, i.e tant que l'objet mobile n'a pas recommencé son activité distante, il est toujours localisé sur son site de départ. Pour maintenir une sémantique claire et pour éviter la perte de messages, tout objet appelant un autre objet en train de migrer sera bloqué le temps de la migration. Une fois la migration terminée, il reprendra son exécution et le mécanisme sous-jacent de communication se chargera de faire parvenir son message à l'agent. Ainsi, même en présence d'une migration de la source ou de la cible d'une communication (voire des deux simultanément), la bibliothèque garantit une sémantique "*at most one*", et des messages qui ne peuvent se doubler point à point.

3.1.4 Communications locales et migration

Il existe une optimisation des systèmes répartis qui permet d'améliorer les performances des communications : deux entités se trouvant dans le même espace d'adressage doivent pouvoir communiquer directement sans passer par la pile des protocoles réseaux, tout en conservant une sémantique identique (en particulier en ce qui concerne les copies profondes d'objets). Dans un système avec migration, la réalisation de cette caractéristique est bien plus délicate car la propriété "*dans le même espace d'adressage*" n'est plus statiquement décidable (à l'allocation des objets actifs), mais varie dynamiquement en fonction des migrations.

Nous avons implémenté cette optimisation, tout en prenant soin de maintenir la sémantique de passage des paramètres. Ainsi, si un objet mobile arrive sur un site, il communique directement avec les autres objets présents sur ce site sans passer par le réseau. De manière symétrique, un objet présent sur un site communique directement avec les objets récemment arrivés. Dans tous les cas, l'absence d'objets passifs partagés est maintenue.

3.1.5 Protocole de migration

Nous allons décrire le protocole mis en œuvre pour effectuer la migration d'un objet sur un site distant. Ces opérations nous permettent de garantir le bon déroulement d'une migration, en particulier en présence de communications. Voici la liste des opérations effectuées par un objet décidant de migrer.

1. Blocage de l'ensemble des communications entrantes (requêtes et réponses),
2. Attente de la fin des communications en cours,
3. Appel au nœud pour recevoir l'objet par copie,

4. Redémarrage de la copie distante et reprise des communications,
5. Suppression de la partie locale,
6. Déblocage des communications en attente.

Bien que les objets actifs soient par définition mono-threadés, il arrive qu'ils soient à un instant donné parcouru par plusieurs threads qui viennent “apporter” les requêtes ou les réponses. Il faut donc s'assurer en cas de migration qu'aucune de ces threads n'est actuellement dans l'objet ce qui se fait dans les étapes 1 et 2. Une fois cette condition remplie, l'objet peut être migré. La copie locale n'est détruite qu'une fois le redémarrage distant bien effectué. Finalement, les éventuels appels qui se seraient trouvés bloqués au niveau de l'objet du fait de la fermeture des communications sont réactivés. Leur fonctionnement futur dépend ensuite de la politique choisie par l'utilisateur. Il est par exemple possible de leur notifier le départ de l'objet.

De la description de ce protocole nous avons en particulier les deux propriétés suivantes :

Propriété 3.1.1 *Tout objet essayant de communiquer avec un agent en train de migrer sera bloqué le temps que la migration se termine.*

Propriété 3.1.2 *Tout objet étant en train de communiquer ne pourra migrer avant la fin de la communication.*

Ces deux propriétés nous permettent de garantir la conservation de la sémantique lors des communications (cf. section 2.5.6).

3.2 API

L'API de migration fournie par *ProActive* a été écrite en essayant de rendre son utilisation aussi simple que possible. Ce choix se traduit par un nombre limité de primitives au dessus desquelles il est possible de construire des mécanismes plus complexes.

3.2.1 Primitives de migration

Il existe en fait une seule primitive implémentée pour fournir la migration des objets actifs communicants: `migrateTo()`. Elle existe en deux versions différentes au niveau du type de paramètre passé qui sont décrites dans la Table 3.1.

Un objet voulant migrer appelle lui-même la méthode statique `migrateTo()` qui se charge de toutes les opérations nécessaires à la migration, notamment la sérialisation de tout le sous-système et sa reconstruction à l'arrivée sur le site distant.

Il est possible soit de migrer vers un *Nœud* distant que l'on désigne explicitement ou en utilisant un nom symbolique (URL), soit de migrer pour rejoindre un autre agent, sans que l'on sache explicitement sur quel nœud il se trouve. Dans ce deuxième cas, la

	Permet la migration vers
`static void migrateTo(URL)`	un site référencé par une URL.
`static void migrateTo(Objet)`	le site supposé d'un autre objet actif.

TAB. 3.1 – *Primitives de migration (méthodes statiques)*

```
public class SimpleAgent implements Active, Serializable {
  public void moveToHost(String t) {
      ProActive.migrateTo(t);
  }
 public void joinFriend(Object friend) {
    ProActive.migrateTo(friend);
     // en cas de succès de la migration
     // tout code situé après ne sera pas exécuté
   }
 public ReturnType foo(CallType p) {
   ...
  }
}
```

FIG. 3.1 – *SimpleAgent: exemple d'un objet actif mobile*

migration se fait en deux étapes, la première consistant à trouver une référence vers le *Nœud* où se trouve l'objet et la deuxième constituant la migration proprement dite. La seule garantie obtenue à travers ce mécanisme c'est qu'un objet A souhaitant rejoindre un autre objet B se trouvera au final sur le site où se trouvait B quand il a initié sa migration.

Pour utiliser cette primitive, un objet actif peut implémenter une méthode publique qui appellera la méthode statique *ProActive.migrateTo()*. Ainsi, la demande de migration pourra venir d'un objet tiers. Cependant, la décision de migrer viendra *in fine* de l'objet mobile lui-même: lors du service d'une requête portant sur une telle méthode publique. A noter que du fait de la migration faible, tout code situé après *ProActive.migrateTo()* ne sera pas exécuté. L'exemple 3.1 illustre l'utilisation des méthodes de migration dans un objet actif.

3.2.2 Abstractions pour la mobilité

Le modèle et les primitives introduits précédemment permettent de construire des objets actifs mobiles en gérant explicitement les migrations. Mais il est également possible de construire, au dessus des primitives de base, des méthodes et des concepts de plus haut niveau afin de gérer la migration à différents niveaux d'abstraction. Cela permet entre autre de faciliter la spécification de comportements autonomes.

Dans la suite de cette section, nous allons présenter trois types d'abstractions réutilisables : l'exécution automatique de méthodes, le suivi d'un *itinéraire* préétabli, et un agent mobile générique.

3.2.2.1 Exécution automatique de méthodes sur départ et arrivée

La migration d'un objet actif mobile comporte deux phases principales correspondant au déroulement nominal de la migration, ce sont le *départ* d'un site et *l'arrivée* sur le site de destination. Une troisième phase existe en présence d'une levée d'*exception*. Nous avons construit une abstraction qui va associer à chacune de ces phases une méthode à exécuter. Cette association se fait en appelant une des méthodes de la table 3.2 avec comme paramètre le nom de la méthode à exécuter. Celle-ci est ensuite appelée par réflexion au moment où l'événement considéré arrive.

	Méthode à exécuter
`static void onDeparture(String s)`	au départ
`static void onArrival(String s)`	à l'arrivée
`static void onException(String s)`	lors d'une exception

TAB. 3.2 – *API d'exécution automatique sur départ et arrivée*

Nous avons volontairement imposé des contraintes concernant les méthodes susceptibles d'être exécutées automatiquement. Tout d'abord elles ne doivent retourner aucune valeur car leur exécution étant déclenchée par le runtime *ProActive* il n'y a pas d'appelant à qui retourner un éventuel résultat. Ensuite elles ne doivent pas prendre de paramètres. Une des contraintes fortes de *ProActive* est le maintien d'une sémantique claire. La valeur exacte des paramètres aurait pu être celle qu'ils avaient au moment où la méthode a été associé à un événement ou celle au moment de l'exécution. Nous avons choisi de laisser la responsabilité de ce choix à l'utilisateur final qui peut dans ces méthodes faire appel aux attributs d'autres objets.

Une utilisation typique de ces méthodes est la suppression et la reconstruction automatique d'éléments non sérialisables qui ne peuvent donc pas être déplacés avec l'objet, tels que les interfaces graphiques (figure 3.2).

3.2.2.2 Itinéraire

L'autonomie d'un agent mobile provient notamment de sa capacité à aller de site en site sans intervention extérieure. Dans le cadre de notre bibliothèque, un itinéraire est formé d'un ensemble de paires *destination-méthode* représentant les sites à visiter et les actions à effectuer à l'arrivée sur chacun d'entre eux. Les destinations sont définies soit par référence directe, soit par leur nom symbolique, auquel cas les mécanismes de nommage de RMI sont utilisés[1]. Une méthode est désignée par son nom, et est exécutée par réflexion.

Cette abstraction associe à chaque objet actif un itinéraire courant qui est suivi de manière séquentielle, et contrôlé par les méthodes de la table 3.3. Un objet mobile démarre un itinéraire en appelant la méthode `travel()`. Il est possible d'appeler la méthode `requestFirst()` pour spécifier le moment où doit intervenir le traitement des requêtes. L'agent peut ainsi suivre un itinéraire qui sera prioritaire sur le traitement des requêtes (elles seront ignorées jusqu'à la dernière destination) ou bien traiter toutes celles en attente avant de poursuivre son parcours. Il est possible de contrôler directement et dynamiquement un itinéraire afin de construire un comportement adapté à une application particulière (voir en particulier les méthodes `Stop` et `Resume` de la table 3.3).

`static void add(URL, String method)`	ajoute un site à visiter
`static void travel()`	démarre un itinéraire
`static void requestFirst(Boolean)`	service des requêtes avant migration
`static void itineraryStop()`	arrête un itinéraire
`static void itineraryResume()`	reprend l'exécution d'un itinéraire
`static void itineraryRestart()`	recommence l'itinéraire

TAB. 3.3 – *API d'utilisation d'un itinéraire*

```
 public void deleteSwingInterface() { ... }
 public void rebuildSwingInterface() { ... }
 public void migrateTo(String dest) {
  ...
    ProActive.migrateTo(dest);
  }

public void live(Body  b) {
  ...
  b.itinerary.add("//tuba.inria.fr/Node1","rebuildSwingInterface");
  b.itinerary.add("//oasis.inria.fr/Node2","rebuildSwingInterface");
  ProActive.onDeparture("deleteSwingInterface");
  ProActive.requestFirst(true);
  ...
  ProActive.travel();
}
```

FIG. 3.2 – *Exécution automatique de méthodes et itinéraires*

1. Ce qui revient à associer à chaque site une URL du type rmi://nomdusite/nomdelobjet

```
public class GenericAgent implements Active, Serializable {
...
public void shareInformation() {
 int max = agentList.size();
 for (int j=0;j<max; j++) {
  ((GenericAgent) agentList.elementAt(j)).informationFound(...);
  }
 }
 public void live(Body myBody) {
  while (...) {
    this.serveAllPendingRequests();
    this.performOperation(); this.shareInformation();
    ProActive.migrateTo(getNextDestination());
  }
 }
}
```

FIG. 3.3 – *Modèle générique d'activité d'un agent mobile*

3.3 Exemple : Agent mobile générique

Beaucoup d'applications à base d'agents mobiles présentent les même caractéristiques : un ou plusieurs agents se déplacent de site en site, effectuent sur chaque site des opérations et partagent leurs résultats. Nous allons montrer comment il est possible d'implémenter la migration et le partage d'information en utilisant *ProActive*.

L'activité de l'objet, présenté dans la figure 3.3, se décompose en plusieurs opérations. L'agent sert d'abord toutes les requêtes en attente, puis il effectue une opération (par exemple une recherche d'information). L'information est ensuite partagée avec les autres objets mobiles. Le code nécessaire pour cette tâche est extrêmement simple puisqu'il consiste à appeler une méthode sur chacun des autres objets, le moteur d'exécution se chargeant de transformer l'appel local en appel distant quelque soit le site sur lequel se trouve le destinataire. Il suffit de redéfinir la méthode *performOperation()* afin de programmer les fonctions précises de l'agent. Il est aussi possible d'avoir une politique de sélection de site (méthode *getNextDestination()*) totalement aléatoire ou bien dépendante des informations déjà recueillies.

3.4 Implémentation au niveau méta

Comme nous l'avons vu dans la présentation de *ProActive* (section 2.5) un objet actif fournit un ensemble de services en particulier concernant la répartition et les communications asynchrones. Il nous a semblé important d'essayer de découper un objet actif de manière à séparer autant que possible ces mécanismes pour faciliter d'une part la modification de ceux existant et d'autre part l'intégration future de nouveaux services. Cette approche s'inspire des travaux sur les MOPs tels que Coda [43]. Pour chacun des services fournis par un objet actif nous avons introduit des méta-objets chargés de l'implémenter. La décomposition finale peut être trouvée dans la figure 3.4.

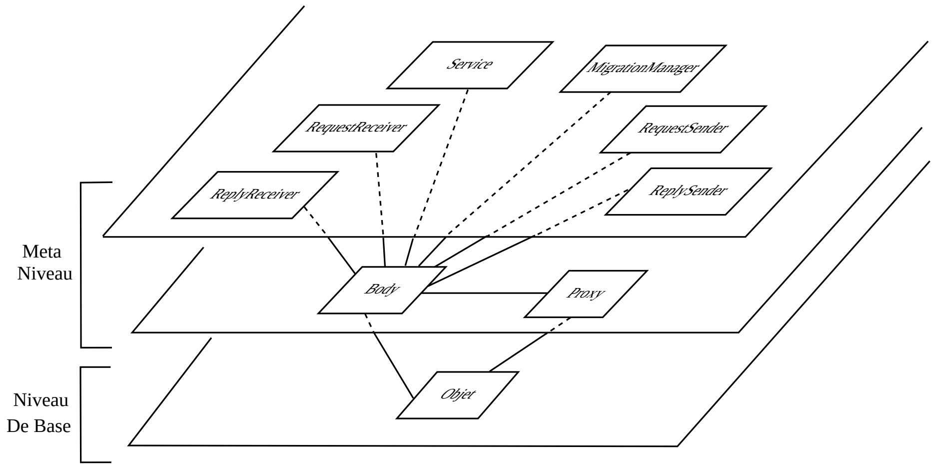


FIG. 3.4 – *Découpage canonique d'un Objet Actif asynchrone*

3.4.1 Body et Proxy

À tout objet actif sont obligatoirement associés deux méta-objets : le *Proxy* et le *Body*.

Body Point d'entrée d'un objet actif pour l'ensemble des communications, c'est la seule partie de l'objet actif accessible à distance. Il a la charge d'assurer la coordination du travail des autres méta-objets.

Proxy Le proxy sert à maintenir une référence sur un objet actif. Il masque la notion de référence locale ou distante.

Le premier mécanisme auquel nous nous sommes intéressé est les communications asynchrones. Leur utilisation nécessite trois objets distincts : requête, réponse et objet future.

3.4.2 Requêtes

Une requête est créée à chaque fois qu'un objet communique avec un objet actif. Sa création est laissée à la charge du *Proxy*. Deux types d'opérations peuvent être effectuées sur une requête : l'envoi et la réception. Nous avons donc naturellement crée deux objets, chacun chargé du bon déroulement des opérations.

RequestSender Une fois la requête créée par le *Proxy*, elle est passée, accompagnée de la référence vers l'objet distant, au *RequestSender* qui se charge de l'envoyer. Ce niveau d'indirection supplémentaire nous permet de rapprocher l'envoi d'une requête des méta-objets de l'objet actif. En effet un proxy n'est normalement pas directement relié au *Body*. En utilisant cette technique, nous pouvons implémenter un comportement spécialisé tout en gardant un *Proxy* générique, toute l'intelligence et le traitement étant localisé dans le *RequestSender*. Par exemple si les communications nécessitent un protocole sécurisé le *RequestSender* pourra se charger de l'encryption ou de l'authentification.

RequestReceiver La réception d'une requête par un objet actif est effectuée par le *RequestReceiver*. Dans l'implémentation de base de *ProActive*, il place la requête dans la queue des requêtes de l'objet. Il est possible comme nous le verrons dans l'étude des mécanismes de localisation de l'utiliser pour des tâches plus complexes.

3.4.3 Service

Le méta-objet *Service* implémente l'activité d'un OA. Il contient un thread qui exécute les requêtes présentes dans la queue. Le choix des requêtes se fait en suivant une politique définie soit à l'intérieur de cet objet, soit dans l'objet rendu actif (cf 2.5.5). À la fin de l'exécution, si une réponse est nécessaire alors il la crée et la passe au méta-objet qui sera chargé de l'envoyer.

3.4.4 Réponses

Une réponse contient le résultat de l'appel d'une méthode sur un objet actif. À une réponse correspond un objet futur dans l'espace d'adressage de l'appelant. De la même façon qu'une requête est envoyée et reçue, une réponse est traitée par deux méta-objets.

ReplySender Il effectue la communication avec l'appelant pour envoyer la réponse à une requête.

ReplyReceiver À la réception d'une réponse il faut mettre à jour le futur correspondant ce dont se charge le *ReplyReceiver*.

3.4.5 Migration

La migration est constituée de plusieurs étapes que nous avons détaillé précédemment. Un objet, le *MigrationManager* est chargé de cette tâche qu'il effectue en collaboration avec les autres méta-objets, en particulier le *RequestReceiver* et le *ReplyReceiver*.

La décomposition d'un objet actif en plusieurs objets spécialisés apparaît comme très naturelle mais ne se révèle pas moins très puissante en ce qui concerne la spécialisation des comportements dont la migration fait partie. Cependant il ne suffit souvent pas de simplement changer un objet, les comportements complexes transcendent les barrières naturelles. Pour s'en rendre compte nous donnons dans la figure 3.5 le flot d'exécution dans le cas d'une communication asynchrone entre deux objets actifs. L'ordre des étapes

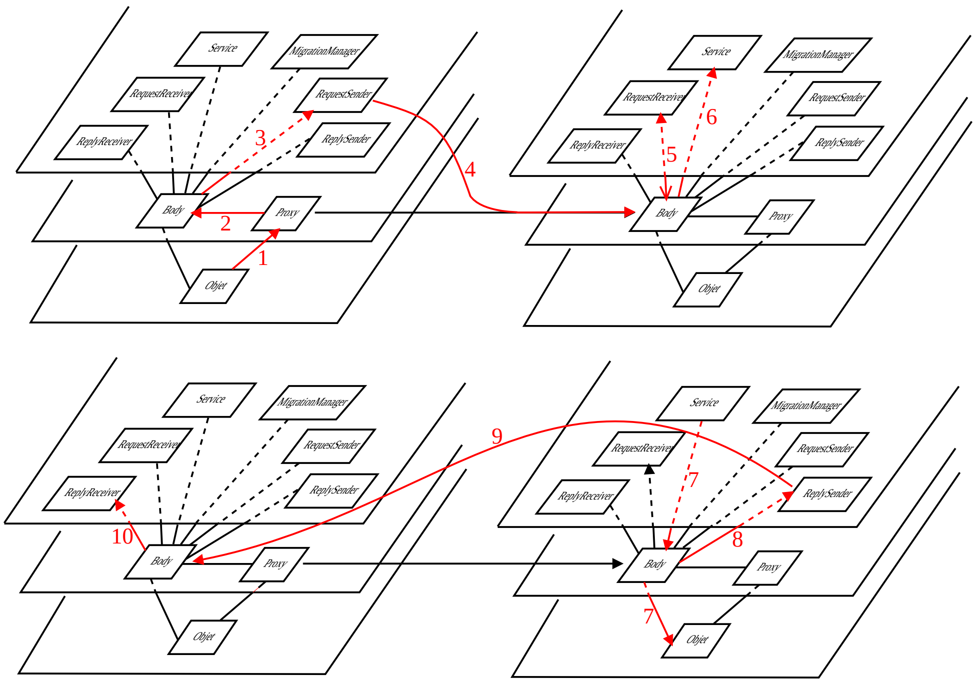


FIG. 3.5 – *Communication entre deux objets actifs*

est indiqué par les numéros croissants des opérations. Nous avons représenté un appel asynchrone qui se fait donc en deux étapes : l'envoi de la requête, et après traitement par l'appelé envoi d'une réponse.

1 L'appelant qui veut communiquer envoie un message qui est intercepté par le *proxy*
2 ce dernier le transmet au *body* pour qu'il le traite
3 l'appel est passé au *RequestSender* qui est responsable de l'envoi des messages
4 la communication réseau s'effectue directement avec le *Body* distant
5 une fois reçue, la requête est directement transmise au *RequestReceiver*
6 qui la rend au *body* après traitement pour être mise en attente de service
7 (x2) suivant la politique de service, la requête est exécutée sur l'objet réifié
8 dans le cas où une réponse est attendue, le résultat de l'exécution est passé au *Reply-Sender*
9 qui la transmet à l'appelant
10 le traitement de la réponse est finalement effectué par le *ReplyReceiver*.

3.5 Communiquer en présence de migration

Nous allons dans cette partie détailler le fonctionnement de deux mécanismes permettant d'assurer des communications avec des objets mobiles. Nous verrons comment il est possible d'avoir une implémentation rapide en utilisant le mop bien qu'une telle propriété transcende les barrières naturelles des objets [37]. Une modélisation formelle de ces mécanismes sera entreprise dans le chapitre 4.

3.5.1 Répéteurs

Les techniques à base de répéteurs ont été introduites dans des systèmes d'exploitation distribués tels que DEMOS/MP [51] pour permettre la localisation des processus mobiles. Ce mécanisme est très simple : quand il quitte un hôte (machine), un processus laisse derrière lui une référence particulière, appelé *pointeur de poursuite* qui pointe vers sa nouvelle localisation. Durant le cycle de vie du système, des *chaînes de répéteurs* se construisent. Une des conséquences de ce système est qu'un appelant ne connaît habituellement pas la localisation précise de celui qu'il appelle. Un mécanisme spécial appelé "short-cutting" permet de mettre à jour la référence une fois qu'une communication a eu lieu. Quand un message qui est passé par un répéteur arrive à un objet mobile, celui-ci envoie à l'appelant sa nouvelle localisation. Cela permet d'éviter aux messages suivants de traverser la chaîne de répéteurs.

Une illustration du raccourcissement de la chaîne est donné dans la figure 3.6 : un message est envoyé par la source vers la dernière localisation de l'agent (Hôte B). Comme l'agent n'est plus sur cette machine, le message est envoyé à l'hôte (C) que l'agent a ensuite visité. Encore une fois, celui-ci ne se trouve pas sur cette machine et le message est envoyé vers la machine D où se trouve finalement l'agent. Un message de localisation est ensuite envoyé par l'agent à la source pour lui indiquer sa nouvelle position. Les messages suivants de la source n'auront plus besoin de traverser les répéteurs et arriveront directement sur la machine D.

Pour conserver la même sémantique que dans un programme où tous les objets sont statiques (non mobiles) nous devons introduire la contrainte suivante : toutes les communications à travers une chaîne de répéteurs sont synchrones, i.e l'appelant reste bloqué durant toute la durée de la communication ce qui permet de maintenir le rendez-vous entre l'appelant et l'appelé.

Nous allons maintenant décrire le protocole :

- Après une migration, un agent laisse sur le site un répéteur;
- Ce répéteur pointe sur l'agent sur son nouveau hôte;
- Quand il reçoit un message, un répéteur l'envoie au prochain hôte (éventuellement l'objet mobile);

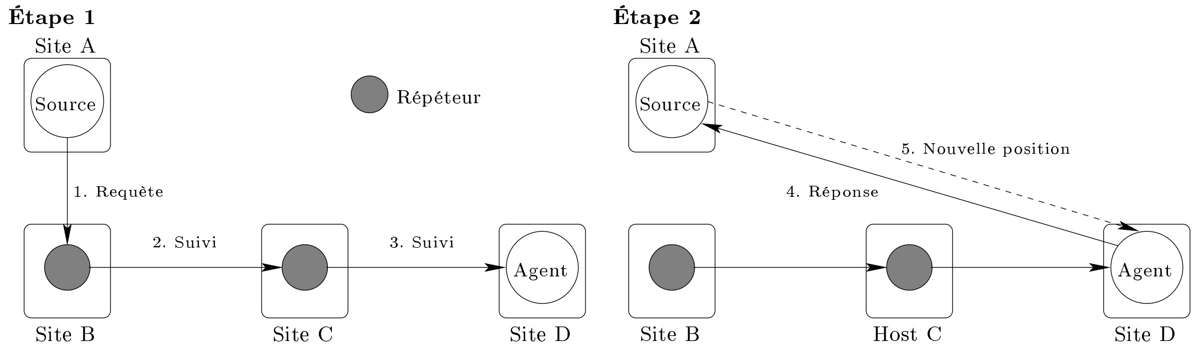


FIG. 3.6 – *Raccourcissement de la chaîne des répéteurs*

– Une communication réussie place l'objet mobile à exactement un saut [2] de l'appelant (e.g. après l'étape 2 dans la figure 3.6 , l'agent est situé à un saut de la source).

Ce protocole est implémenté dans diverses bibliothèques Java (*MOA* [44] et *Voyager* [70] par exemple) avec souvent comme seule variante le raccourcissement de la chaîne qui est parfois effectué par l'agent à la fin de l'exécution d'une requête. Nous avons choisi de raccourcir la chaîne dès réception d'une requête pour deux raisons. La première est que certaines requêtes ne demandent pas de réponse et donc un objet ne communicant avec un agent qu'avec des messages *one way* devra payer à chaque fois un coût important du fait que la chaîne ne sera jamais réduite. La deuxième est que pendant le traitement d'une requête par un agent, un appelant peut continuer à communiquer avec lui, et donc là encore traverser la chaîne des répéteurs.

3.5.1.1 Utilisation d'un MOP

Les répéteurs peuvent être implémentés de manière élégante en utilisant un MOP à méta-objets dynamiques : le *MigrationManager* laisse derrière lui un objet actif spécialement construit pour faire suivre les requêtes et les réponses qu'il reçoit. Il s'agit en fait de l'objet actif migrant dont les *RequestReceiver* et *ReplyReceiver* ont été remplacés par respectivement un *RequestReceiverForwarder* et un *ReplyReceiverForwarder* (voir figure 3.7). Tous les autres méta-objets sont laissés à la charge du ramasse miettes car devenus inutiles. Nous pouvons qualifier cette implémentation d'élégante car seul l'objet mobile est concerné par ces changements, l'appelant n'a besoin d'aucune modification.

Cependant cette version simple d'un répéteur ne permet pas de faire du raccourcissement de chaîne, ce qui risque de poser un problème de performance. Nous allons tenter d'améliorer notre répéteur. Le raccourcissement doit s'effectuer au moment du rendez-vous et est initié par l'appelé. Il faut donc lui ajouter du code pour qu'il puisse fournir sa nouvelle localisation. Les méta-objets concernés par ce changement sont le *RequestRecei-*

2. nous appelons saut une communication directe entre deux objets

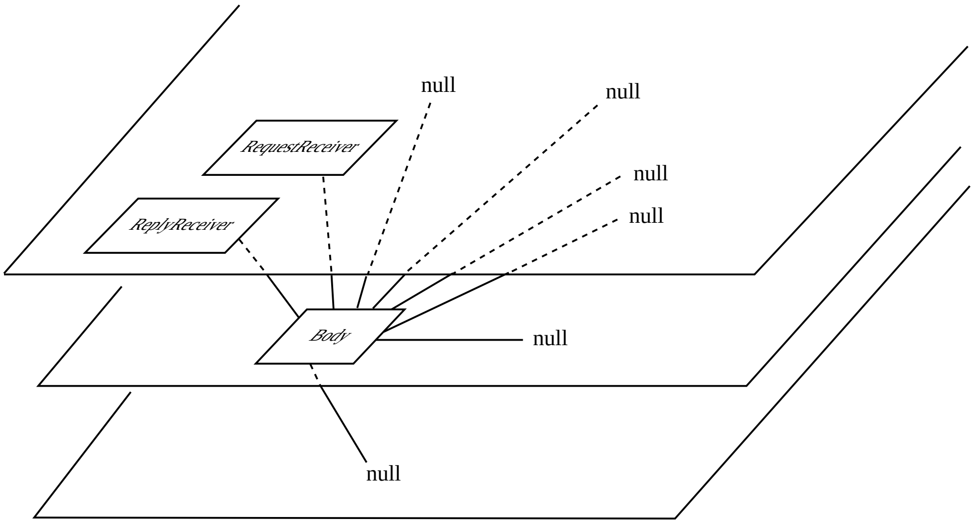


FIG. 3.7 – *Une version simpliste d'un répéteur*

ver et le *ReplyReceiver*. De même l'appelant doit avoir un mécanisme lui permettant d'une part de recevoir la nouvelle référence de l'appelé, et d'autre part de l'utiliser. Nous voyons donc qu'en pratique l'appelant et l'appelé doivent tous les deux subir des modifications pour pouvoir utiliser la localisation à base de répéteurs. Nous résumons ces modifications dans la table 3.4.

Du côté de l'appelant :
  Body : reçoit des informations de position de l'agent lors du raccourcissement de la chaîne.
Du côté de l'appelé :
  MigrationManager : crée un répéteur sur le site de départ,
  RequestReceiver : envoie la nouvelle position de l'agent si la requête a traversé des répéteurs,
  ReplyReceiver : envoie la nouvelle position de l'agent si la réponse a traversé des répéteurs.

TAB. 3.4 – *Méta-objets modifiés pour le raccourcissement de la chaine de répéteurs*

3.5.2 Serveur centralisé

Une alternative à l'utilisation des répéteurs pour localiser un objet mobile est l'utilisation d'un serveur de localisation. Celui-ci garde des références vers les objets mobiles dont il a la charge et quand une source lui fait une demande, il lui envoie la localisation d'un objet. L'idée derrière ce mécanisme est la même que pour les serveurs de nom de domaine par exemple [45] qui permettent d'associer un nom symbolique à une adresse IP. On peut légitimement se demander l'avantage que présente un serveur de localisation par rapport à une source que l'agent contacterait pour lui indiquer sa position. Nous y voyons deux avantages. Le premier est que l'utilisation d'un serveur permet de garder la trace de plusieurs agents, venant d'applications différentes créées par des utilisateurs différents. Le deuxième est qu'avoir le serveur sur une machine distante permet d'opérer en mode déconnecté voire même d'avoir une source migrable. Le serveur étant fixe il sera toujours un point de référence.

Nous supposerons par la suite que le serveur de localisation dont dépendent les objets est bien connu et ne nécessite pas de mécanisme annexe pour obtenir sa référence. Le fonctionnement du système avec serveur est relativement simple : à chaque fois qu'un objet est en fin de migration, il envoie à l'agent sa nouvelle localisation; quand une source essaie de joindre un agent, elle envoie son message à la dernière localisation connue de celui-ci; si la communication échoue, alors une demande est envoyée au serveur. Cette méthode est souvent qualifiée de paresseuse car les références vers les objets mobiles ne sont mises à jour que lorsqu'elles sont effectivement utilisées. Nous allons maintenant donner une description précise du protocole utilisé par la source et l'agent pour communiquer avec le serveur :

- L'objet mobile

  Étape 1: Effectue la migration;

  Étape 2: Envoie sa nouvelle position au serveur.

- La Source

  Étape 1: Envoie un message à l'agent en utilisant sa dernière position connue. En cas d'échec, aller à l'étape 2;

  Étape 2: Demande au serveur la localisation de l'objet;

  Étape 3: Envoie un message à l'objet en utilisant la référence fournie par le serveur. En cas d'échec, aller à l'étape 2.

Le protocole décrit ci-dessus est par exemple utilisé dans *MOA* [44]. Une illustration de son fonctionnement est donné dans la figure 3.8. Dans les figures 3.8(b)-(c) l'ordre chronologique des événements est indiqué par les numéros 1,2,.... La figure 3.8(a) montre une communication normale quand la localisation de l'objet est connue de la source. Si l'objet migre, alors nous nous trouvons dans la situation illustrée par la figure 3.8(b) où la source essaie de joindre un agent qui s'est déplacé (événements 3). Elle est donc obligée de demander au serveur une nouvelle référence vers l'agent (événements 5&6). Avec cette nouvelle référence, la source va pouvoir de nouveau essayer d'envoyer son message à l'agent. Si l'objet mobile s'est de nouveau déplacé pendant que la source attendait une réponse du serveur (figure 3.8(c)) alors le processus doit recommencer. Une fois la bonne référence obtenue, la communication peut enfin aboutir (3.8(d)).

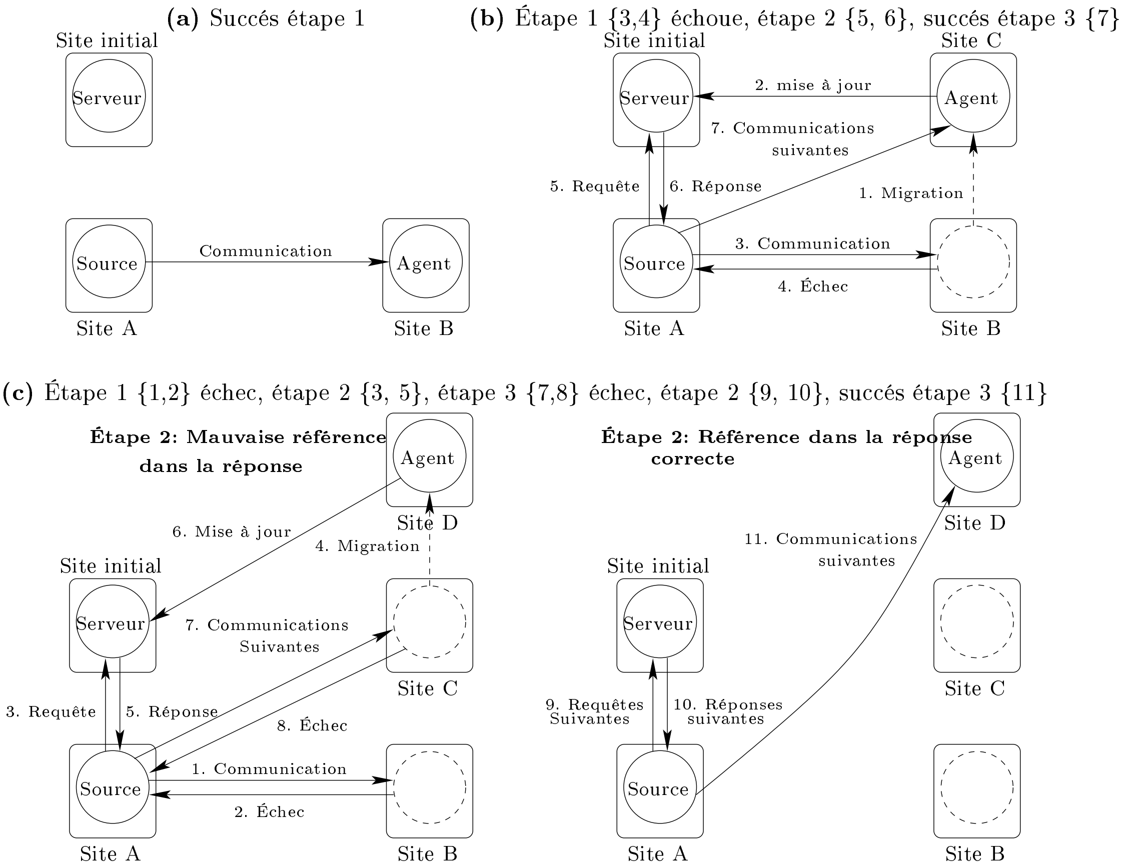


FIG. 3.8 – *Quelques scénarios de l'utilisation du serveur par une source.*

Dans notre protocole la source est totalement dépendante de la vitesse de traitement du serveur car elle ne peut pas communiquer avec l'agent si elle ne connaît pas sa localisation. Il faut donc que le serveur implémente une politique de service minimisant les délais perçus par une source. Dans la queue du serveur peuvent se trouver deux types de requête :

update location request : nouvelle localisation envoyée par l'agent

location request : demande de la position d'un agent par une source

Le service fourni par le serveur est différent suivant le type de requête. Une venant de l'agent ne nécessite que de mettre à jour une table contenant les localisations. Si il s'agit d'un message de la source, alors en plus de la recherche dans la table, il faut envoyer le résultat.

Le service des requêtes dans une queue a été implémenté en différenciant entre les deux types de requêtes. Le serveur choisit en priorité les messages venant de l'agent (figure 3.9 (a)) car ils indiquent un changement de localisation. Dans le cas où la queue ne contient qu'une demande de la source (figure 3.9 (b)), alors celle-ci peut être servie car la localisation de l'agent est connue (modulo une éventuelle migration qu'il est en train d'effectuer). Les requêtes de la source nécessitent l'envoi d'une réponse (la nouvelle référence), ce dont se charge le serveur. Dans le cas où pendant le service d'une source un message d'un agent est arrivé, nous savons que la référence qui va être envoyée à la source n'est plus à jour. Dans ce cas, le serveur, plutôt que d'envoyer une réponse fausse remet la requête de la source à la fin de la queue (figure 3.9 (c)). Pour obtenir ce comportement

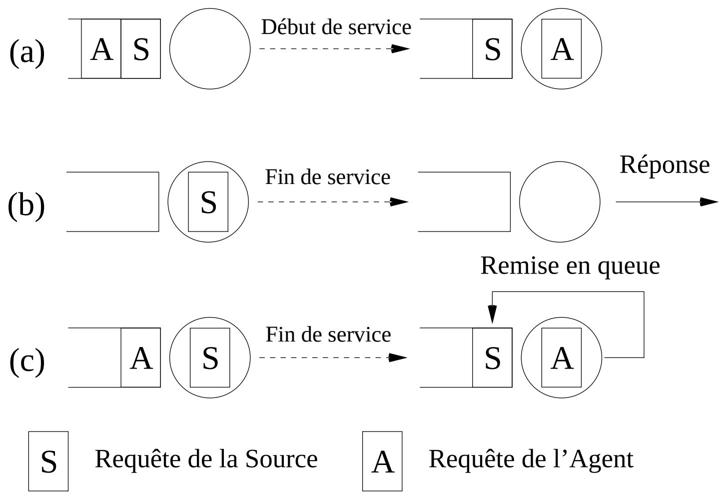


FIG. 3.9 – *Exemples de changement d'états dans le serveur*

nous avons implémenté le serveur en utilisant un objet actif dont le service des requêtes a été spécialisé pour répondre à nos besoins.

3.5.2.1 Utilisaton d'un MOP

Nous pouvons voir de la simple description du protocole qu'il ne sera pas possible d'avoir une implémentation transparente pour l'appelant car sa participation active est nécessaire. Concentrons-nous tout d'abord sur l'objet mobile. Après chaque migration, il doit contacter le serveur de localisation ce qui nécessite un *MigrationManager* modifié mais n'influe pas sur ses autres méta-objets. L'appelant par contre a besoin d'un mécanisme pour contacter le serveur, ce qui peut être nécessaire lors de l'envoi d'une requête ou d'une réponse. Nous devons donc lui adjoindre un *RequestSender* et un *ReplySender* modifiés. Étant donné que le *Proxy* peut avoir une mauvaise référence si l'agent a migré mais que le *RequestSender* en obtient une nouvelle, nous avons ajouté un système de mise à jour du *Proxy*. Celui-ci obtient une référence mise à jour après une communication réussie.

Du côté de l'appelant :
- *RequestSender :* cherche la nouvelle position de l'agent si l'envoi de requête échoue
- *ReplySender :* cherche la nouvelle position de l'agent si l'envoi de requête échoue

Du côté de l'appelé :
- *MigrationManager :* contacte le serveur pour mettre à jour la localisation

TAB. 3.5 – *Méta-objets modifiés pour le serveur de localisation*

3.6 Conclusion

A travers ce chapitre, nous avons vu comment il était possible d'implémenter un mécanisme de mobilité dans la bibliothèque *ProActive*. L'utilisation de la réflexion et le découpage en propriétés non fonctionnelles des objets actifs permet une conception simple et élégante. Cependant, pour les deux mécanismes de communication étudiés, il s'est avéré nécessaire de modifier l'appelant et l'appelé pour assurer les communications.

La figure 3.10 montre les objets modifiés pour permettre la migration et les communications en utilisant les deux mécanismes précédents. Nous voyons bien que cela touche tous les objets chargés des communications, que cela soit du côté de l'appelant ou de l'appelé. Bien sûr, chacun des mécanismes ne nécessite pas la modification de tous les objets.

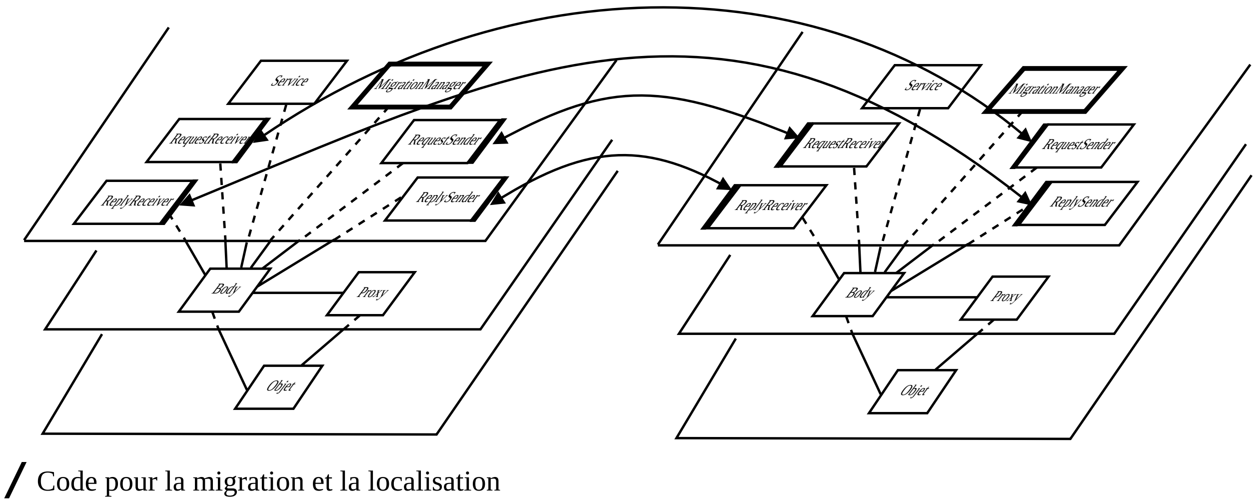


FIG. 3.10 – *Méta-objets concernés par la migration*

Chapitre 4

Étude théorique de deux mécanismes de communication

Nous allons étudier les deux mécanismes de communication introduits dans les sections 3.5.1 et 3.5.2. Le but de cette étude est double : tout d'abord avoir des modèles formels permettant de mieux caractériser le comportement de ces deux techniques; ensuite utiliser ces modèles pour prévoir le comportement, c'est-à-dire les performances, étant donné les conditions d'exécution d'une application. Au final, nous voulons être capable de choisir entre ces deux mécanismes en fonction des conditions rencontrées et des performances souhaitées.

4.1 Définitions et notations

Dans cette section nous introduisons des variables aléatoires que nous utiliserons pour construire nos modèles.

Le i-ème message est envoyé par la source à l'agent à l'instant a_i et la communication se termine au temps $d_i := a_i + \tau_i$. La variable aléatoire τ_i – que nous appellerons (i-ème) *temps de communication* – dépend du mécanisme considéré et sera explicitée plus tard. Durant l'intervalle de temps (d_i,a_{i+1}) aucun message n'est généré par la source. Soit $w_{i+1} := a_{i+1} - d_i$ la longueur de cet intervalle; nous supposerons que $w_1 := a_1$ et qu'aucun message n'est généré dans $[0,a_1]$.

La j-ème migration de l'objet mobile intervient au temps $m_j > 0$ et demande à l'objet p_j unités de temps pour atteindre son nouveau lieu d'exécution. Pendant une migration, l'agent n'est pas joignable. Il passe ensuite u_{j+1} unités de temps sur le j-ème hôte, temps durant lequel il peut être joint par un message, ensuite débute une nouvelle migration. Par convention, nous avons $u_1 := m_1$ et nous supposerons que l'agent ne migre pas dans $[0,m_1)$ (voir figure 4.1).

L'hypothèse suivante sera utilisée tout au long de ce chapitre :

H1 Les séquences $\{w_i,i \geq 1\}$, $\{p_j,j \geq 1\}$ et $\{u_k,k \geq 1\}$ sont des séquences de va-

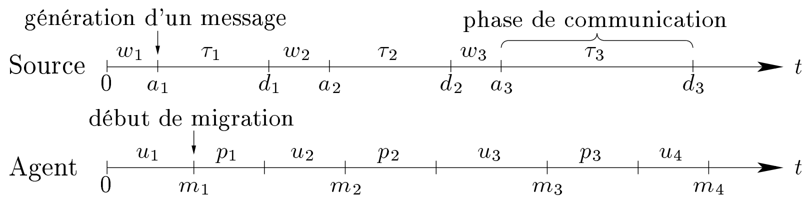


FIG. 4.1 – *Diagramme du temps incluant les variables aléatoires relatives à la source et à l'agent.*

riables aléatoires mutuellement indépendantes telles que w_i, p_j et u_k suivent des lois exponentielles de paramètres respectifs $\lambda > 0$, $\delta > 0$ et $\nu > 0$.

Nous verrons la signification de ces paramètres dans la section suivante.

4.2 Analyse Markovienne des répéteurs

4.2.1 Modèle

Nous allons étudier les performances (temps de réponse et nombre moyen de répéteurs) du mécanisme présenté dans la section 3.5.1 grâce à une analyse Markovienne. Nous allons considérer une unique source essayant de joindre un unique agent se déplaçant sur un nombre potentiellement infini de sites. De plus, nous supposerons que l'agent ne revient jamais à un site qu'il a déjà visité sur lequel se trouve encore un répéteur en activité (i.e. membre de la chaîne reliant la source à l'agent). Cette hypothèse a comme conséquence que la chaîne de répéteurs peut potentiellement avoir une longueur infinie si la source ne joint jamais l'agent. Quand la source essaie de joindre un agent elle envoie un message qui parcourt la chaîne jusqu'à l'atteindre. Nous voyons que dans la description de ce mécanisme apparaissent trois entités : la source, l'agent et le message qui parcourt la chaîne.

Proposition 4.2.1 *Le mécanisme de communication à base de répéteurs est entièrement représenté par l'état de la source, de l'agent et la distance séparant un message (où la source en l'absence de message) d'un agent.*

La distance est définie comme étant le nombre de communications réseau restant à effectuer pour atteindre l'agent. Quand il n'y a pas de répéteurs la distance vaut 1 et quand il y a n répéteurs entre un message et l'agent cette distance vaut $n + 1$.

Pour la modélisation de notre système, nous introduisons les paramètres λ, ν, δ et γ qui sont décrits dans la table 4.1. Nous allons considérer un triplet (i,j,k) décrivant l'état de notre système, avec i pour la distance message-agent ou source-agent si il n'y a pas de messages en transit, j l'état de l'agent et k celui de la source. Pour éviter d'augmenter de manière trop importante le nombre d'états nous avons introduit un exposant à k pour représenter le fait que la source attend ou non la phase de raccourcissement de la chaîne.

Paramètre	Description
$1/\lambda$	Temps moyen d'inactivité de la source
$1/\nu$	Temps moyen d'inactivité de l'agent
$1/\delta$	Durée moyenne de migration
$1/\gamma$	Délai moyen inter-sites

TAB. 4.1 – *Paramètres de la modélisation du mécanisme des répéteurs*

En effet, dans le cas où le message a traversé des répéteurs, l'agent doit contacter la source pour lui indiquer sa nouvelle localisation. Nous noterons $*$ le cas où la communication commence alors que la source se trouve à un saut de l'agent. Si ce dernier ne migre pas, alors le message n'aura pas à traverser de répéteurs et il n'y aura pas de raccourcissement. Nous avons donc, d'après la description donnée dans la section 3.5.1, les états suivants quand la source n'essaie pas de communiquer :

- L'état $(i,0,0)$, $i \geq 1$, indique que l'agent est disponible (i.e. ne migre pas) et se trouve à i sauts de la source ;
- Les états $(i,1,0)$, $i \geq 1$, indiquent que l'agent migre et qu'avant la migration il était situé à i sauts de la source.

Si la source essaie de joindre l'agent, nous avons les états suivants :

- L'état $(1,0,1^*)$ indique que l'agent est disponible et localisé à un saut d'un message, et ce dernier n'a jamais traversé de répéteur;
- L'état $(1,0,1)$ indique que l'agent est disponible et situé à un saut d'un message, ce dernier ayant déjà traversé un répéteur;
- Les états $(i,0,1)$, $i \geq 2$, indiquent qu'un message voyage et que l'agent est disponible et localisé à i sauts du message;
- Les états $(i,1,1)$, $i \geq 1$, indiquent que l'agent migre et qu'avant la migration il était localisé à i sauts du message en transit;
- L'état $(0,1,1)$ indique qu'un message qui a traversé au moins un répéteur se trouve sur le même site que l'agent mais que celui-ci est en train de migrer. (i.e. l'agent a démarré une migration juste avant l'arrivée du message);
- L'état $(0,0,1)$ indique qu'un message a atteint l'agent après avoir traversé au moins un répéteur et que l'agent est actuellement en train de communiquer sa nouvelle position à la source.

Notons qu'il ne peut y avoir dans notre système qu'un unique message en transit à cause de la propriété de rendez-vous dans nos communications.

En utilisant l'hypothèse suivante

H2 La durée de transmission d'un message d'un site (éventuellement la source) au suivant (éventuellement l'agent) est une variable aléatoire exponentielle de paramètre

$\gamma > 0$. Les temps de transmission successifs sont indépendants entre eux et indépendants des séquences d'entrée $\{w_i\}_i$, $\{p_i\}_i$ et $\{u_i\}_i$ introduites dans la section 4.1.

et l'hypothèse **H1** [1], il est facile de voir que le temps de séjour dans chacun des états est exponentiellement distribué et que n'importe quel état peut être atteint depuis un état en un nombre fini d'étapes. En d'autres termes, le processus défini précédemment est une chaîne de Markov irréductible sur l'espace des états
$\mathcal{E} := \{(0,0,1),(0,1,1),(1,0,1^*),(i,j,k), i \geq 1, j,k = 0,1\}$.

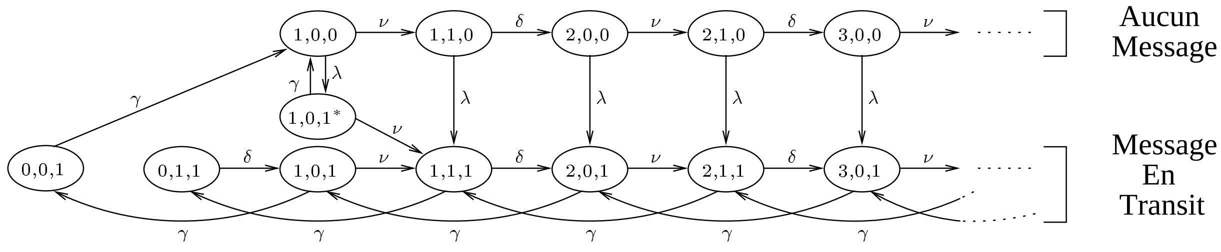


FIG. 4.2 – *États et transitions du mécanisme des répéteurs*

Les taux de transition sont indiqués dans la figure 4.2 et détaillés dans la table 4.2; par exemple depuis l'état (1,0,0) le processus peut aller à l'état ($1,0,1^*$) avec un taux λ (un nouveau message est généré) ou à l'état (1,1,0) avec un taux ν (migration de l'agent); depuis l'état ($1,0,1^*$) le processus peut aller dans l'état (1,1,1) avec un taux ν (migration). Le reste des transitions est détaillé dans la table 4.2.

Nous allons étudier en détail certaines transitions de notre diagramme pour montrer qu'elles permettent de rendre compte des subtilités du protocole de communication à base de répéteurs. La figure 4.3 montre quatre zones du diagramme de transition extraites de la vue générale.

figure 4.3 a) Cette transition illustre une communication entre la source et l'agent quand celle-ci a la bonne localisation (indice $*$). La communication commence après un temps moyen de $1/\lambda$ et finit après une durée moyenne de $1/\gamma$.

figure 4.3 b) Après une durée moyenne $1/\nu$ l'agent commence sa migration et la finit après un temps moyen $1/\delta$. Il s'éloigne donc de la source dont le message aura un répéteur en plus à traverser ce qui le place à 3 sauts.

figure 4.3 c) Quand un message ayant traversé des répéteurs atteint l'agent, celui-ci contacte la source pour lui envoyer sa nouvelle position. Durant cette phase de raccourcissement, il ne peut pas migrer. La seul façon de sortir de cet état est donc par une transition γ ce qui correspond à l'envoi d'un message sur le réseau.

1. les hypothèses **H1** et **H2** sont introduites pour des raisons de simplification mathématique. Nous avons cependant observé que nos modèles sont assez robustes aux non respect de celles-ci – cf. Section 5.

Départ	Transition	Arrivée	Description
(1,0,0)	$\xrightarrow{\nu}$	(1,1,0)	Début de migration
	$\xrightarrow{\lambda}$	(1,0,1*)	Nouveau message généré
(i,0,0) avec $i \geq 2$	$\xrightarrow{\nu}$	(i,1,0)	Début de migration
	$\xrightarrow{\lambda}$	(i,0,1)	Nouveau message généré
(i,1,0) avec $i \geq 1$	$\xrightarrow{\delta}$	($i+1$,0,0)	Fin de migration
	$\xrightarrow{\lambda}$	(i,1,1)	Nouveau message généré
(1,0,1*)	$\xrightarrow{\nu}$	(1,1,1)	Début de migration
	$\xrightarrow{\gamma}$	(1,0,0)	Message a atteint l'agent
(i,0,1) avec $i \geq 1$	$\xrightarrow{\nu}$	(i,1,1)	Début de migration
	$\xrightarrow{\gamma}$	($i-1$,0,1)	Message a effectué un saut
(i,1,1) avec $i \geq 1$	$\xrightarrow{\delta}$	($i+1$,0,1)	Fin de migration
	$\xrightarrow{\gamma}$	($i-1$,1,1)	Message a effectué un saut
(0,1,1)	$\xrightarrow{\delta}$	(1,0,1)	Fin de migration
(0,0,1)	$\xrightarrow{\gamma}$	(1,0,0)	Message a atteint la source

TAB. 4.2 – *Détails des transitions dans le modèle des répéteurs*

figure 4.3 d) Dans cette situation, la source atteint un agent en train de migrer. Pour maintenir la sémantique de rendez-vous (propriété 3.1.1), la communication reste bloquée le temps que l'agent finisse sa migration, soit $1/\delta$. Après cette durée, le message se retrouve à un saut de l'agent et celui-ci attend le début de sa prochaine migration.

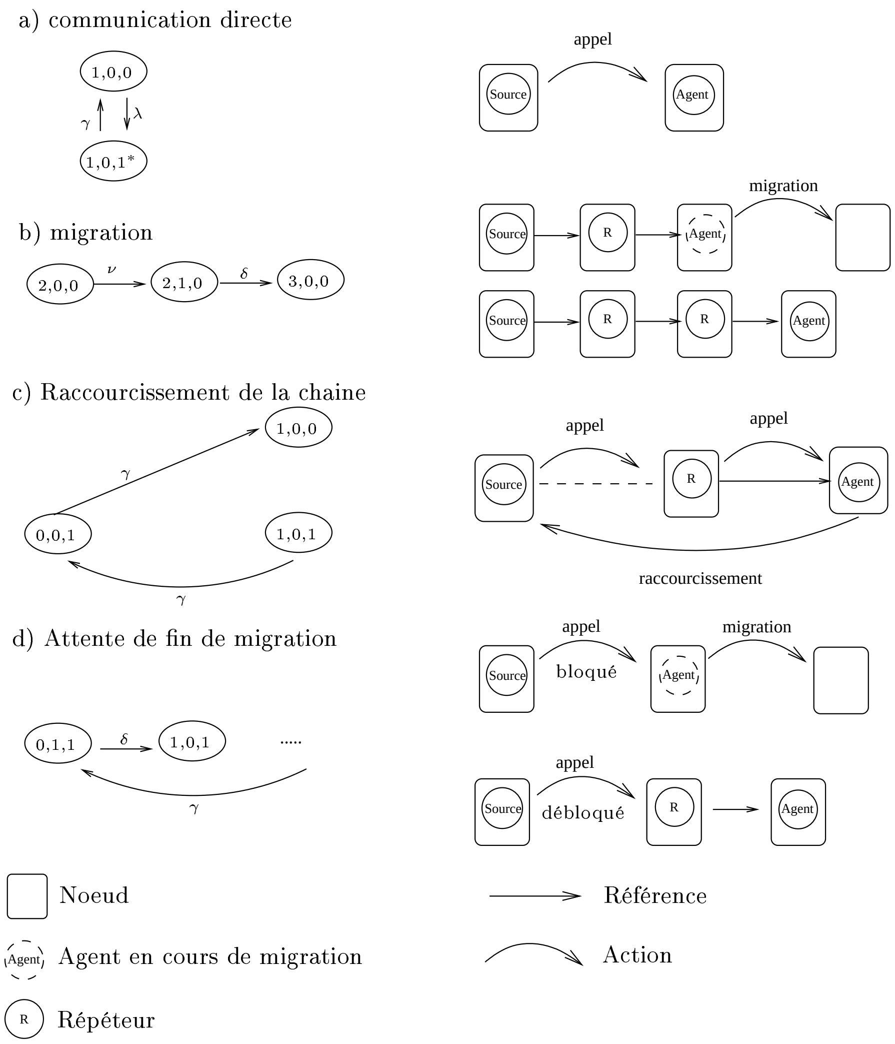


FIG. 4.3 – *Détails des transitions du diagramme des répéteurs*

Soit $p_{i,j,k}$ la probabilité stationnaire que le processus soit dans l'état $(i,j,k) \in \mathcal{E}$. Si le processus Markovien est ergodique, alors les probabilités stationnaires $\{p_{i,j,k},(i,j,k) \in \mathcal{E}\}$ sont l'unique solution strictement positive de l'équation de Chapman-Kolmogorov (C-K) [38]. Ces équations sont obtenues en utilisant la propriété de conservation des flux d'une chaîne de Markov dans l'état stationnaire :

Propriété 4.2.1 (Conservation des flux) *Quand une chaîne de Markov est dans l'état stationnaire, le flux entrant dans un état donné est égal au flux sortant.*

Regardons sur la figure 4.2 l'état (1,0,0) pour bien comprendre l'utilisation de cette propriété. Le flux entrant dans un état est symbolisé par les flèches qui pointent vers cet état, alors que le flux sortant d'un état est symbolisé par les flèches qui en partent. Ainsi nous pouvons écrire

$$(\lambda+\nu)\, p_{1,0,0} = \gamma\,(p_{0,0,1} + p_{1,0,1^*}).$$

En appliquant le même principe à l'ensemble des états de notre chaîne nous obtenons les égalités suivantes :

$$
\begin{aligned}
(\lambda+\nu)\, p_{1,0,0} &= \gamma\,(p_{0,0,1} + p_{1,0,1^*}) && && (4.1)\\
(\lambda+\nu)\, p_{i,0,0} &= \delta\, p_{i-1,1,0} && i = 2,3,\ldots && (4.2)\\
(\lambda+\delta)\, p_{i,1,0} &= \nu\, p_{i,0,0} && i = 1,2,\ldots && (4.3)\\
\delta\, p_{0,1,1} &= \gamma\, p_{1,1,1} && && (4.4)\\
(\delta+\gamma)\, p_{1,1,1} &= \nu\,(p_{1,0,1} + p_{1,0,1^*}) + \lambda\, p_{1,1,0} + \gamma\, p_{2,1,1} && && (4.5)\\
(\delta+\gamma)\, p_{i,1,1} &= \nu\, p_{i,0,1} + \lambda\, p_{i,1,0} + \gamma\, p_{i+1,1,1} && i = 2,3,\ldots && (4.6)\\
(\nu+\gamma)\, p_{1,0,1^*} &= \lambda\, p_{1,0,0} && && (4.7)\\
p_{0,0,1} &= p_{1,0,1} && && (4.8)\\
(\nu+\gamma)\, p_{1,0,1} &= \delta\, p_{0,1,1} + \gamma\, p_{2,0,1} && && (4.9)\\
(\nu+\gamma)\, p_{i,0,1} &= \delta\, p_{i-1,1,1} + \lambda\, p_{i,0,0} + \gamma\, p_{i+1,0,1} && i = 2,3,\ldots. && (4.10)
\end{aligned}
$$

Étant donné que nous considérons des probabilités, nous avons aussi $\sum_{(i,j,k)\in\mathcal{E}} p_{i,j,k} = 1$ (condition de normalisation).

La prochaine étape de notre calcul va consister à trouver la valeur des probabilités (distribution stationnaire) en fonction des seuls paramètres de notre modèle. Nous avons choisi de ne pas résoudre explicitement les équations (4.1)-(4.10) car cela se révèle très compliqué. Nous allons plutôt utiliser une transformée en z, technique souvent utilisée dans le cas où l'espace des états est infini. Pour cela nous introduisons les fonctions suivantes pour $|z| \leq 1$

$$
\begin{aligned}
f(z) &:= \sum_{i=1}^{\infty} z^i p_{i,0,0} & \qquad g(z) &:= \sum_{i=1}^{\infty} z^i p_{i,1,0}\\
h(z) &:= \sum_{i=0}^{\infty} z^i p_{i,0,1} & \qquad k(z) &:= \sum_{i=0}^{\infty} z^i p_{i,1,1}
\end{aligned}
$$

qui sont les transformées en z des probabilités stationnaires $\{p_{i,0,0}\}_{i\geq 1}$, $\{p_{i,1,0}\}_{i\geq 1}$, $\{p_{i,0,1}\}_{i\geq 0}$ et $\{p_{i,1,1}\}_{i\geq 0}$ respectivement. Enfin, nous introduisons pour $|x| \leq 1$,$|y| \leq 1$ et $|z| \leq 1$

$$\begin{aligned} F(x,y,z) &:= \sum_{(i,j,k)\in\mathcal{E}} z^i x^j y^k p_{i,j,k} && (4.11)\\ &= \sum_{i=1}^{\infty} z^i p_{i,0,0} + x\sum_{i=1}^{\infty} z^i p_{i,1,0} + y\sum_{i=0}^{\infty} z^i p_{i,0,1} + xy\sum_{i=0}^{\infty} z^i p_{i,1,1} + zy^2\, p_{1,0,1^*} && (4.12)\\ &= f(z) + x\,g(z) + y\,h(z) + xy\,k(z) + zy^2\, p_{1,0,1^*} && (4.13) \end{aligned}$$

Nous allons maintenant essayer d'exprimer $F(x,y,z)$ en fonction des paramètres (λ, ν, δ, γ). Ce calcul se fera en plusieurs étapes et nécessitera d'exprimer les fonctions $f(z)$, $g(z)$, $h(z)$ et $k(z)$ en fonction de $p_{1,0,1}$ ou $p_{0,1,1}$.

Proposition 4.2.2

$$f(z) = \frac{z\,\gamma(\lambda+\delta)(\lambda+\nu)(\gamma+\nu)}{\nu(\nu+\lambda+\gamma)\,a(z)}\, p_{1,0.1} \tag{4.14}$$

$$g(z) = \frac{z\,\gamma(\lambda+\nu)(\gamma+\nu)}{(\nu+\lambda+\gamma)\,a(z)}\, p_{1,0,1} \tag{4.15}$$

avec

$$a(z) := -\delta\nu z + (\lambda+\delta)(\lambda+\nu)$$

Preuve : En utilisant les équations (4.2) et (4.3) et en faisant une récursion triviale pour $i \geq 1$, nous avons

$$p_{i,0,0} = \left[\frac{\delta\nu}{(\lambda+\delta)(\lambda+\nu)}\right]^{i-1} p_{1,0,0} \qquad\qquad p_{i,1,0} = \left[\frac{\delta\nu}{(\lambda+\delta)(\lambda+\nu)}\right]^{i-1} \frac{\nu}{\lambda+\delta}\, p_{1,0,0}$$

pour $i \geq 1$.

En utilisant les résultats sur les sommes des termes d'une suite géométrique, nous avons

$$f(z) = \left[\frac{z\,(\lambda+\delta)(\lambda+\nu)}{(\lambda+\delta)(\lambda+\nu) - z\,\delta\nu}\right] p_{1,0,0} \qquad\qquad g(z) = \left[\frac{z\,\nu(\lambda+\nu)}{(\lambda+\delta)(\lambda+\nu) - z\,\delta\nu}\right] p_{1,0,0}. \tag{4.16}$$

Pour finir la preuve, il nous reste à exprimer $p_{1,0,0}$ en fonction de $p_{1,0,1}$. En utilisant (4.1), (4.7) et (4.8) nous obtenons

$$p_{1,0,0} = \frac{\gamma(\gamma+\nu)}{\nu(\nu+\lambda+\gamma)}\, p_{1,0,1} \tag{4.17}$$

et en remplaçant cette valeur dans (4.16) nous trouvons (4.14) et (4.15) ce qui conclut la preuve. ■

Avant de procéder à un calcul similaire pour $k(z)$ et $h(z)$, nous allons montrer une relation entre $p_{1,0,1^*}$ et $p_{1,0,1}$.

Proposition 4.2.3

$$p_{1,0,1^*} = \frac{\lambda\gamma}{\nu\,(\nu+\lambda+\gamma)}\,p_{1,0,1} \tag{4.18}$$

Preuve : En utilisant (4.1), (4.7) et (4.8), nous avons

$$p_{1,0,1} + p_{1,0,1^*} = \frac{(\lambda+\nu)\,(\gamma+\nu)}{\lambda\gamma}\,p_{1,0,1^*} = \frac{(\lambda+\nu)\,(\gamma+\nu)}{\nu\,(\nu+\lambda+\gamma)}\,p_{1,0,1} \tag{4.19}$$

ce qui après simplification nous donne la relation cherchée. ■

Intéressons nous maintenant à $k(z)$ et $h(z)$.

Proposition 4.2.4

$$\begin{aligned} h(z) &= -\frac{z^2\,\delta\gamma}{b(z)}\,p_{0,1,1} + \left[\frac{\delta(\nu-\gamma)z^2 - \gamma(\nu+\delta)z + \gamma^2}{b(z)}\right. \\ &\quad \left. - \frac{z^3\delta\gamma[\lambda\,a(z) + (\gamma+\nu)(\lambda\gamma + (\lambda+\delta)(\lambda+\nu))]}{(\nu+\lambda+\gamma)\,a(z)\,b(z)}\right]p_{1,0,1} &(4.20)\\ k(z) &= \frac{\gamma\,(\gamma - z\,(\gamma+\nu))}{b(z)}\,p_{0,1,1} \\ &\quad - \frac{z^2\gamma(\lambda+\nu)(\gamma+\nu)(\lambda\gamma + (\lambda+\delta)(\lambda+\nu))}{(\nu+\lambda+\gamma)\,a(z)\,b(z)}\,p_{1,0,1} &(4.21)\end{aligned}$$

avec

$$\begin{aligned} a(z) &:= -\delta\nu z + (\lambda+\delta)(\lambda+\nu) \\ b(z) &:= \delta\nu z^2 - \gamma(\gamma+\nu+\delta)z + \gamma^2 \end{aligned}$$

Preuve : A partir de (4.4)-(4.6) et (4.8)-(4.10) nous obtenons

$$\begin{aligned} [z(\gamma+\nu)-\gamma]\,h(z) &= z\nu[p_{1,0,1} - zp_{1,0,1^*}] - \gamma[p_{1,0,1} + z^2 p_{1,0,1^*}] \\ &\quad + z\lambda f(z) + z^2\delta k(z) &(4.22)\\ [z(\gamma+\delta)-\gamma]\,k(z) &= \gamma(z-1)p_{0,1,1} + z\lambda g(z) + z\nu h(z) - z\nu[p_{1,0,1} - zp_{1,0,1^*}] &(4.23)\end{aligned}$$

En remplaçant $f(z)$ et $g(z)$ dans (4.22)-(4.23) par leurs valeurs déterminées dans (4.14) et (4.15), nous obtenons un système linéaire de deux équations dont les inconnues sont

$h(z)$ et $k(z)$. La résolution formelle de ce système et l'utilisation de (4.18) donne (4.20) et (4.21). ■

Proposition 4.2.5 (Condition de stabilité) *Pour que notre processus Markovien soit stable il faut et il suffit que*

$$\frac{1}{\gamma} < \frac{1}{\nu} + \frac{1}{\delta}. \tag{4.24}$$

Preuve : Considérons la condition de normalisation $\sum_{(i,j,k)\in\mathcal{E}} p_{i,j,k} = 1$ et remarquons qu'elle est équivalente à $F(1{,}1{,}1) = 1 = p_{1,0,1^*} + f(1) + g(1) + h(1) + k(1)$. Avec les équations (4.18), (4.14), (4.15), (4.20) et (4.21) nous voyons que la condition de normalisation sera satisfaite si et seulement si

$$p_{0,1,1} + \frac{(\lambda+\nu)(\gamma+\nu)(\gamma+\lambda)}{\lambda\nu(\nu+\lambda+\gamma)}\, p_{1,0,1} = \frac{\delta\nu}{\delta+\nu}\left(\frac{1}{\nu} + \frac{1}{\delta} - \frac{1}{\gamma}\right)(1 - p_{1,0,1}). \tag{4.25}$$

Nous pouvons conclure de (4.25) que la condition $\frac{1}{\gamma} < \frac{1}{\nu} + \frac{1}{\delta}$ est nécessaire pour que le processus Markovien soit stable. En effet, si $\frac{1}{\gamma} = \frac{1}{\nu} + \frac{1}{\delta}$ alors (4.25) n'est valable que si $p_{0,1,1} = p_{1,0,1} = 0$, alors le processus Markovien n'est pas ergodique sur l'espace des états $\mathcal{E}$; si $\frac{1}{\gamma} > \frac{1}{\nu} + \frac{1}{\delta}$ alors (4.25) n'est pas validé si $p_{0,1,1}, p_{1,0,1} > 0$ et encore une fois le processus Markovien n'est pas ergodique sur $\mathcal{E}$. Cette condition est suffisante car elle permet de calculer les probabilités stationnaires comme nous le verrons dans la proposition 4.2.6 ■

Nous avons réussi à exprimer la transformée en z en fonction des paramètres, de $p_{0,1,1}$ et $p_{1,0,1}$; pour conclure cette partie il nous reste donc à trouver une expression pour ces probabilités.

Proposition 4.2.6

$$p_{0,1,1} = \frac{1 - \delta\nu/(\gamma(\delta+\nu))}{1 + [1 - \delta\nu/(\gamma(\delta+\nu)) + (\lambda+\nu)(\gamma+\nu)(\gamma+\lambda)/(\lambda\nu(\nu+\lambda+\gamma))]\, c(z_0)} \tag{4.26}$$

$$p_{1,0,1} = \frac{[1 - \delta\nu/(\gamma(\delta+\nu))]c(z_0)}{1 + [1 - \delta\nu/(\gamma(\delta+\nu)) + (\lambda+\nu)(\gamma+\nu)(\gamma+\lambda)/(\lambda\nu(\nu+\lambda+\gamma))]\, c(z_0)} \tag{4.27}$$

avec

$$c(z) := \frac{(\gamma - z(\gamma+\nu))(\nu+\lambda+\gamma)\, a(z)}{((\lambda+\nu)(\gamma+\nu)(\lambda\gamma + (\lambda+\delta)(\lambda+\nu))\, z^2)},$$

$$z_0 = \gamma\left(\gamma+\nu+\delta - \sqrt{(\gamma+\nu+\delta)^2 - 4\nu\delta}\right)/(2\nu\delta).$$

Preuve : Le polynôme $b(z)$ introduit dans la proposition 4.2.4 a deux solutions en z mais en utilisant la condition de stabilité nous pouvons montrer qu'une seule, z_0, est positive et strictement inférieure à 1.

$$z_0 := \gamma\,\frac{\gamma+\nu+\delta - \sqrt{(\gamma+\nu+\delta)^2 - 4\nu\delta}}{2\nu\delta}.$$

Nous avons donc par définition $b(z_0) = 0$. Or pour que $h(z)$ (resp. $k(z)$) soit bien définie, il faut que le coefficient de $1/b(z)$ dans (4.20) (resp. dans (4.21)) s'annule en ce point. En utilisant (4.21) nous obtenons la relation suivante entre $p_{0,1,1}$ et $p_{1,0,1}$

$$p_{1,0,1} = \frac{(\gamma - z_0\,(\gamma + \nu))\,(\nu + \lambda + \gamma)\,a(z_0)}{(\lambda + \nu)(\gamma + \nu)(\lambda\gamma + (\lambda + \delta)(\lambda + \nu))\,z_0^2}\,p_{0,1,1}. \tag{4.28}$$

Avec (4.25) et (4.28) nous obtenons $p_{0,1,1}$ et $p_{1,0,1}$ ce qui nous permet de conclure le calcul de $F(x,y,z)$. ■

En résumé, nous avons montré que $F(x,y,z)$ est définie dans $|z| < 1$ pour toute valeur de x et y, continue dans $|z| \leq 1$ pour toute valeur de x et y et satisfait la condition $F(1,1,1) = 1$ si la condition (4.24) est respectée. Nous pouvons trouver$\epsilon > 0$ tel que $F(x,y,z)$ est définie dans $|z| < 1 + \epsilon$ pour toute valeur de x et y (si z_1 et z_2 sont les zéros de $b(z)$ et $a(z)$ respectivement, dans $|z| > 1$, alors $\epsilon = \min(|z_1|,|z_2|) - 1$). En utilisant un résultat classique des transformées en z [41], nous pouvons conclure que (4.24) est la condition de stabilité et que $F(x,y,z)$ est la transformée en z des probabilités stationnaires.

4.2.2 Temps moyen de communication

Nous allons maintenant nous intéresser au temps moyen de communication, c'est-à-dire le temps moyen que met une source pour envoyer un message à un agent mobile. Plus précisément, nous allons utiliser la définition suivante :

Définition 4.2.1 *Le temps moyen de communication pour le mécanisme des répéteurs est le temps pris par un message pour rejoindre un agent plus le temps mis par cet agent pour éventuellement mettre à jour sa localisation auprès de la source.*

La mise à jour de la localisation de l'agent n'intervient que si le message a traversé au moins un répéteur. En effet, si le message arrive à l'agent après exactement un saut, alors la source connaît sa localisation et envoyer une mise à jour est inutile (état $(1,0,1^*)$).

Immédiatement après la fin d'une communication l'agent ne migre pas, la source est inactive et à un saut de celui-ci du fait du raccourcissement de la chaîne. Cela correspond à l'état (1,0,0). À cet instant, la source reste inactive durant une durée exponentiellement distribuée de moyenne $1/\lambda$. Une fois cette période finie, une nouvelle communication est initiée et le système peut être dans n'importe lequel des états suivants :

- $(1,0,1^*)$
- $(i,0,1)$ pour $i \geq 2$
- $(i,1,1)$ pour $i \geq 1$

Nous allons appeler $T_{i,j,k}$ avec $i \geq 1$, $j = 0,1$, $k = 1$ ou avec $(i,j,k) = (1,0,1^*)$ le temps moyen de communication pour un message quand le système se trouve dans l'état (i,j,k) juste après son émission.

Définition 4.2.2 *Le temps moyen de communication T_F (l'indice F indique "forwarder") est donné par*

$$T_F = q_F(1{,}0{,}1^*)\,T_{(1,0,1^*)} + \sum_{i=2}^{\infty} q_F(i{,}0{,}1)\,T_{(i,0,1)} + \sum_{i=1}^{\infty} q_F(i{,}1{,}1)\,T_{(i,1,1)}$$

où $q_F(i{,}j{,}k)$ est la probabilité d'atteindre l'état $(i{,}j{,}k)$ sachant que le système était initialement dans l'état $(1{,}0{,}0)$.

$q_F(i{,}j{,}k)$ représente la probabilité qu'une communication commence quand le système passe de l'état $(i{,}j{,}0)$ à l'état $(i{,}j{,}k)$. Arrêtons-nous un instant sur cette formule pour la détailler. Une communication peut commence lorsque la source a fini d'attendre, ce qui est matérialisé par une transition λ. De la figure 4.2 nous voyons qu'il y a trois familles d'états possibles : $(1{,}0{,}1^*)$,$(i{,}0{,}1)$ et $(i{,}1{,}1)$. Le temps d'une communication correspondra donc au temps qu'il faudra au système pour rejoindre l'état $(1{,}0{,}0)$ qui marque la fin d'une communication à partir de n'importe quel état parmi ces trois familles possibles. Pour obtenir le temps moyen de communication il faut tenir compte de toutes les communications possibles et de leurs probabilités respectives, d'où la formule de la définition 4.2.2.

Proposition 4.2.7

$$T_F = \begin{cases} \dfrac{1}{\alpha(\lambda)}\left[((\lambda+\delta)(\lambda+\nu)-\gamma\nu)\,T_{0,1,1} - \dfrac{(\lambda+\delta)(\lambda+\nu+\delta)}{\delta} \right. \\ \qquad \left. - \dfrac{(\lambda+\delta)(\lambda+\nu)(\lambda(\lambda+\nu)+\nu(\gamma+\nu))+\delta\gamma\nu\lambda}{\lambda(\lambda+\nu)(\gamma+\nu)}\right] & \text{pour } \lambda \neq \lambda_0 \\[2ex] \dfrac{(2\lambda+\nu+\delta)(2\gamma(\gamma+\nu)(\lambda+\nu+\delta)+\delta\lambda(\lambda+\nu)((\gamma+\nu)\,T_{0,1,1}-1))}{\lambda\delta(2\lambda+\nu+\delta-\gamma)(\lambda+\nu)(\gamma+\nu)} \\ \qquad - \dfrac{(2\lambda+\nu)(\lambda+\delta)(\lambda+\nu-\gamma(\gamma+\nu)\,T_{0,1,1})+\gamma\nu\delta}{\lambda(2\lambda+\nu+\delta-\gamma)(\lambda+\nu)(\gamma+\nu)} & \text{pour } \lambda = \lambda_0 \end{cases} \tag{4.29}$$

avec

$$\begin{aligned} \lambda_0 &= \frac{\gamma-\nu-\delta+\sqrt{(\gamma+\nu+\delta)^2-4\delta\nu}}{2} \\ \alpha(\lambda) &= (\lambda+\delta)(\lambda+\nu)-\gamma(\lambda+\nu+\delta) \\ T_{0,1,1} &= \frac{\gamma(\lambda_0+\nu+\delta)+\delta\,\lambda_0}{\gamma\delta\lambda_0}. \end{aligned}$$

Preuve : En utilisant la figure 4.2 nous trouvons que

$$\begin{aligned} q_F(1{,}0{,}1^*) &= \frac{\lambda}{\lambda+\nu} && (4.30)\\ q_F(i{,}0{,}1) &= \frac{\lambda(\lambda+\delta)}{\delta\nu}\, r^i \quad i = 2{,}3,\dots && (4.31)\\ q_F(i{,}1{,}1) &= \frac{\lambda}{\delta}\, r^i \quad i = 1{,}2,\dots && (4.32) \end{aligned}$$

où $r := \dfrac{\delta\nu}{(\lambda+\delta)(\lambda+\nu)} < 1$.

La première relation s'obtient en remarquant que si le processus est dans l'état (1,0,0), il peut aller vers l'état (1,1,0) avec un taux ν et vers l'état $(1{,}0{,}1^*)$ avec un taux λ. La probabilité de passer de (1,0,0) à $(1{,}0{,}1^*)$ est alors de $\frac{\lambda}{\lambda+\nu}$. Le même raisonnement nous permet de trouver l'expression de $q_F(i{,}0{,}1)$ et $q_F(i{,}1{,}1)$.

En utilisant (4.30)-(4.32) nous pouvons récrire T_F comme suit :

$$T_F = \frac{\lambda}{\lambda+\nu}\, T_{1,0,1^*} + \frac{\lambda(\lambda+\delta)}{\delta\nu} \sum_{i=2}^{\infty} r^i\, T_{i,0,1} + \frac{\lambda}{\delta} \sum_{i=1}^{\infty} r^i\, T_{i,1,1}. \qquad (4.33)$$

Avec $G(z) := \sum_{i=0}^{\infty} z^i\, T_{i,0,1}$ et $H(z) := \sum_{i=0}^{\infty} z^i\, T_{i,1,1}$, (4.33) devient

$$T_F = \frac{\lambda}{\lambda+\nu} T_{1,0,1^*} + \frac{\lambda(\lambda+\delta)}{\delta\nu}\, \left(G(r) - r\, T_{1,0,1} - T_{0,0,1}\right) + \frac{\lambda}{\delta}\, \left(H(r) - T_{0,1,1}\right).$$

Il nous reste à déterminer les fonctions génératrices $G(z)$ et $H(z)$ pour $z = r$. Pour cela nous allons utiliser les équations récursives qui découlent directement de la description Markovienne du protocole (figure 4.2) :

$$\begin{aligned} T_{1,0,1^*} &= \frac{1}{\nu+\gamma} + \frac{\nu}{\nu+\gamma} T_{1,1,1}\\ T_{0,0,1} &= \frac{1}{\gamma}\\ T_{i,0,1} &= \frac{1}{\nu+\gamma} + \frac{\nu}{\nu+\gamma} T_{i,1,1} + \frac{\gamma}{\nu+\gamma} T_{i-1.0,1} \qquad i = 1{,}2,\dots\\ T_{0,1,1} &= \frac{1}{\delta} + T_{1,0,1}\\ T_{i,1,1} &= \frac{1}{\delta+\gamma} + \frac{\delta}{\delta+\gamma} T_{i+1,0,1} + \frac{\gamma}{\delta+\gamma} T_{i-1,1,1} \quad i = 1{,}2,\dots \end{aligned}$$

ce qui donne

$$\begin{aligned}(\nu+\gamma\,(1-z))\,G(z)-\nu\,H(z) &= \frac{1}{1-z}+\frac{\nu}{\gamma}-\nu T_{0,1,1}\\ -\delta\,G(z)+(\delta+\gamma\,(1-z))\,zH(z) &= \frac{z}{1-z}-\frac{\delta}{\gamma}+\gamma z\,T_{0,1,1}.\end{aligned}$$

Et en isolant $G(z)$ et $H(z)$ nous trouvons :

$$G(z) = \frac{1}{D(z)}\left(\frac{(\delta+\nu)\,z}{1-z}+\frac{\nu}{\gamma}(1-z)(\gamma z-\delta)+\gamma z+\nu(\gamma z-\delta)\,zT_{0,1,1}\right) \quad (4.34)$$

$$H(z) = \frac{1}{D(z)}\left(\frac{(\delta+\nu)\,z}{1-z}+(\gamma+\delta)\,z-(\gamma^2\,z^2-\gamma(\gamma+\nu)\,z+\delta\nu)T_{0,1,1}\right) \quad (4.35)$$

avec

$$D(z) := (z-1)\left(\gamma^2\,z^2-\gamma(\gamma+\nu+\delta)\,z+\delta\nu\right). \quad (4.36)$$

En utilisant (4.34)-(4.36) et $T_{1,0,1^*}=T_{0,1,1}-1/\delta-1/(\gamma+\nu)$ nous obtenons

$$\begin{aligned}T_F &= \frac{1}{\alpha(\lambda)}\Bigg[((\lambda+\delta)(\lambda+\nu)-\gamma\nu)\,T_{0,1,1}-\frac{(\lambda+\delta)(\lambda+\nu+\delta)}{\delta}\\ &\quad -\frac{(\lambda+\delta)(\lambda+\nu)(\lambda(\lambda+\nu)+\nu(\gamma+\nu))+\delta\gamma\nu\lambda}{\lambda\,(\lambda+\nu)(\gamma+\nu)}\Bigg]\end{aligned} \quad (4.37)$$

avec

$$\alpha(\lambda) := (\lambda+\delta)(\lambda+\nu)-\gamma(\lambda+\nu+\delta).$$

Il reste à identifier la constante $T_{0,1,1}$ dans (4.37). Il suffit de remarquer que $\alpha(\lambda)$ a un unique zéro $\lambda=\lambda_0$ dans $[0,\infty)$ qui est

$$\lambda_0=\frac{\gamma-\nu-\delta+\sqrt{(\gamma+\nu+\delta)^2-4\delta\nu}}{2}.$$

Pour que T_F soit défini pour toute valeur non négative de λ, le coefficient de $1/\alpha(\lambda)$ dans (4.37) doit s'annuler quand $\lambda=\lambda_0$ ce qui nous donne une nouvelle relation nous permettant de déterminer $T_{0,1,1}$. En utilisant l'égalité $(\lambda_0+\delta)(\lambda_0+\nu)=\gamma(\lambda_0+\nu+\delta)$ et en fixant à 0 le coefficient de $1/\alpha(\lambda)$ dans (4.37) quand $\lambda=\lambda_0$, nous obtenons

$$T_{0,1,1}=\frac{\gamma(\lambda_0+\nu+\delta)+\delta\,\lambda_0}{\gamma\delta\lambda_0}.$$

Il suffit ensuite d'injecter cette valeur dans l'expression de T_F. Dans le cas où $\lambda=\lambda_0$ l'expression de T_F s'obtient en appliquant la règle de l'Hôpital qui consiste, pour trouver la limite d'une fraction, à rechercher la limite de la fraction construite en remplaçant les numérateur et dénominateur par leur dérivée. ∎

4.2.3 Nombre moyen de répéteurs

Dans cette section nous allons calculer le nombre moyen de répéteurs entre l'agent et la source quand le système est à l'équilibre. Soit $q(i)$ la probabilité que l'agent soit situé à $i \geq 1$ sauts de la source, ce qui correspond à la situation où il y a exactement $i-1$ répéteurs dans le système. Nous avons pour $i \geq 1$

$$q(i) = p_{i,0,0} + p_{i,1,0} + p_{i,0,1} + p_{i,1,1} + \mathbf{1}\{i = 1\} p_{1,0,1^*}$$

Proposition 4.2.8 *Le nombre moyen de répéteurs N_s (l'indice "s" se réfère à la source qui est le point de référence pour le comptage des répéteurs) est donné par la formule*

$$\begin{aligned} N_s &= \frac{\delta^2(\gamma^2 - \gamma\nu + \nu^2)(1 - p_{1.0.1})}{\gamma(\delta+\nu)(\gamma(\delta+\nu) - \delta\nu)} \\ &\quad + \left(\frac{\nu\gamma(\gamma+\lambda)(\gamma+\nu)}{\lambda^2(\nu+\lambda+\gamma)} - \frac{\gamma^2 - \nu^2}{\nu} \right) \frac{\delta\, p_{1,0,1}}{\gamma(\delta+\nu) - \delta\nu} \end{aligned} \tag{4.38}$$

Preuve : Par définition, nous avons

$$\begin{aligned} N_s &= \sum_{i=1}^{\infty} (i-1)\, q(i) \\ &= \sum_{i=1}^{\infty} i\, q(i) - \sum_{i=1}^{\infty} q(i) \\ &= f'(1) + g'(1) + h'(1) + k'(1) + p_{1,0,1^*} - (1 - p_{0,0,1} - p_{0,1,1}) \end{aligned} \tag{4.39}$$

avec $f'(1)$ la dérivée de $f(z)$ en $z = 1$...

De (4.14) et (4.15) nous avons

$$f'(1) + g'(1) = \frac{\gamma(\lambda+\delta)(\lambda+\nu)^2(\gamma+\nu)}{\lambda^2\nu(\nu+\lambda+\gamma)(\lambda+\delta+\nu)}\, p_{1,0,1} \tag{4.40}$$

tandis qu'en utilisant (4.20) et (4.21) nous obtenons

$$\begin{aligned} h'(1) + k'(1) &= \frac{\delta\nu(\delta\gamma+\nu^2) - (\nu^2\delta(\nu-\gamma) + \gamma(\nu+\delta)(\delta\gamma - \nu^2))p_{1,0,1}}{\nu(\delta+\nu)(\gamma(\delta+\nu) - \delta\nu)} \\ &\quad + \frac{\delta\gamma(\gamma+\nu)(\delta\lambda(\nu-\gamma) + \nu(\lambda+\nu)(\lambda+\delta))}{\lambda^2(\nu+\lambda+\gamma)(\lambda+\nu+\delta)(\gamma(\delta+\nu) - \delta\nu)} p_{1,0,1} \end{aligned} \tag{4.41}$$

En combinant (4.39), (4.40) avec l'expression trouvée pour $p_{0,0,1}$, $p_{1,0,1^*}$ et $p_{0,1,1}$ (dans (4.8), (4.18) et (4.25), respectivement), nous obtenons l'expression de N_s. ■

Le nombre moyen de répéteurs N_s représente le nombre moyen de répéteurs que devra traverser un message pour atteindre un agent *si* celui-ci ne migre pas pendant la communication. Cependant, il serait intéressant de connaître aussi le nombre de répéteurs effectivement traversés par un message si l'agent décide de migrer. En d'autres termes, nous devons tenir compte du fait que le nombre de répéteurs peut augmenter entre le début d'une communication et sa fin. Pour calculer ce nombre, noté N, une analyse similaire à celle utilisée dans la section 4.2.2 doit être effectuée. Notons au passage que nous devrions avoir $N > N_s$.

Quand une communication commence, un message est généré et le système peut être dans un des états suivants : $(1{,}0{,}1^*)$, $(i{,}0{,}1)$ pour $i \geq 2$, ou $(i{,}1{,}1)$ pour $i \geq 1$. Si le nombre de répéteurs est n au début de la communication, ce nombre diminue au fur et à mesure que le message voyage vers l'agent, mais il peut aussi augmenter si l'agent migre et donc s'éloigne de celui-ci. Appelons $N_{i,j,k}$ avec $i \geq 1$, $j = 0{,}1$, $k = 1$ ou avec $(i{,}j{,}k) = (1{,}0{,}1^*)$ le nombre moyen de répéteurs que le message devra traverser si le système est dans l'état $(i{,}j{,}k)$ juste après la génération du message.

Définition 4.2.3 *Le nombre moyen de répéteurs N traversés par un message est donné par*

$$N = q_F(1{,}0{,}1^*)\, N_{(1,0,1^*)} + \sum_{i=2}^{\infty} q_F(i{,}0{,}1)\, N_{(i,0,1)} + \sum_{i=1}^{\infty} q_F(i{,}1{,}1)\, N_{(i,1,1)}$$

où $q_F(i{,}j{,}k)$ *indique la probabilité d'atteindre l'état* $(i{,}j{,}k)$ *sachant que le système était dans l'état* $(1{,}0{,}0)$ *(comme présenté dans la section 4.2.2).*

$q_F(i{,}j{,}k)$ représente en fait la probabilité qu'une communication commence quand le système passe de l'état $(i{,}j{,}0)$ à l'état $(i{,}j{,}k)$. Les expressions de $q_F(i{,}j{,}k)$ sont données dans (4.30)-(4.32).

Proposition 4.2.9

$$N = \begin{cases} \dfrac{((\lambda+\delta)(\lambda+\nu) - \gamma\nu)\nu/\lambda_0 - \nu\gamma(\lambda+\delta)/\lambda}{(\lambda+\delta)(\lambda+\nu) - \gamma(\lambda+\nu+\delta)} & \textit{pour } \lambda \neq \lambda_0, \\[2ex] \dfrac{\nu(2\lambda_0^2 + (\nu+\delta)\lambda_0 + \gamma\delta)}{\lambda_0^2(2\lambda_0 + \nu + \delta - \gamma)} & \textit{pour } \lambda = \lambda_0, \end{cases} \tag{4.42}$$

où λ_0 *est donné dans la proposition 4.2.7.*

Preuve : Nous allons procéder de la même manière que pour le temps de réponse. En utilisant les expressions de $q_F(i{,}j{,}k)$, nous pouvons écrire

$$N = \frac{\lambda}{\lambda+\nu}\, N_{1,0,1^*} + \frac{\lambda(\lambda+\delta)}{\delta\nu} \sum_{i=2}^{\infty} r^i\, N_{i,0,1} + \frac{\lambda}{\delta} \sum_{i=1}^{\infty} r^i\, N_{i,1,1}. \tag{4.43}$$

Avec les définitions $N_0(z) := \sum_{i=0}^{\infty} z^i\, N_{i,0,1}$ et $N_1(z) := \sum_{i=0}^{\infty} z^i\, N_{i,1,1}$, (4.43) devient

$$N = \frac{\lambda}{\lambda+\nu} N_{1,0,1^*} + \frac{\lambda(\lambda+\delta)}{\delta\nu}\left(N_0(r) - r\, N_{1,0,1} - N_{0,0,1}\right) + \frac{\lambda}{\delta}\left(N_1(r) - N_{0,1,1}\right). \tag{4.44}$$

Il reste à déterminer les fonctions génératrices $N_0(z)$ et $N_1(z)$ quand $z = r$. Pour cela, nous allons utiliser les équations récursives qui découlent de la description Markovienne du protocole dans la figure 4.2 :

$$N_{1,0,1^*} = \frac{\nu}{\nu+\gamma} N_{1,1,1} \tag{4.45}$$
$$N_{0,0,1} = 0 \tag{4.46}$$
$$N_{1,0,1} = \frac{\nu}{\nu+\gamma} N_{1,1,1} + \frac{\gamma}{\nu+\gamma} N_{0,0,1} \tag{4.47}$$
$$N_{i,0,1} = \frac{\nu}{\nu+\gamma} N_{i,1,1} + \frac{\gamma}{\nu+\gamma}\left(1 + N_{i-1,0,1}\right) \qquad i = 2,3,\ldots \tag{4.48}$$
$$N_{0,1,1} = N_{1,0,1} \tag{4.49}$$
$$N_{i,1,1} = \frac{\delta}{\delta+\gamma} N_{i+1,0,1} + \frac{\gamma}{\delta+\gamma}\left(1 + N_{i-1,1,1}\right) \quad i = 1,2,\ldots. \tag{4.50}$$

Remarquons en utilisant les équations (4.45)-(4.47) et (4.49) que

$$N_{1,0,1^*} = N_{1,0,1} = N_{0,1,1}. \tag{4.51}$$

Après calcul, nous obtenons en utilisant (4.45)-(4.50) le système linéaire suivant

$$\begin{aligned}(\nu + \gamma\,(1-z))\, N_0(z) - \nu\, N_1(z) &= \frac{\gamma z^2}{1-z} - \nu N_{1,0,1}\\ -\delta\, N_0(z) + (\delta + \gamma\,(1-z))\, z N_1(z) &= \frac{\gamma z^2}{1-z} + \gamma z\, N_{0,1,1}.\end{aligned}$$

En le résolvant pour $N_0(z)$ et $N_1(z)$ nous obtenons

$$N_0(z) = \frac{1}{D(z)}\left(\frac{\gamma\, z^2}{z-1}\Big(\gamma z^2 - (\gamma+\delta) z - \nu\Big) + \nu(\gamma z - \delta)\, z N_{1,0,1}\right) \tag{4.52}$$
$$N_1(z) = \frac{1}{D(z)}\left(\frac{\gamma\, z^2}{z-1}\Big(\gamma z - (\gamma+\delta+\nu)\Big) - (\gamma^2 z^2 - \gamma(\gamma+\nu)\, z + \delta\nu) N_{1,0,1}\right) \tag{4.53}$$

avec $D(z)$ donné dans (4.36). En utilisant (4.52)-(4.53) avec (4.51) et (4.36), (4.44) peut se récrire en

$$N = \frac{((\lambda+\delta)(\lambda+\nu) - \gamma\nu) N_{1,0,1} - \nu\gamma(\lambda+\delta)/\lambda}{(\lambda+\delta)(\lambda+\nu) - \gamma(\lambda+\nu+\delta)} \tag{4.54}$$

où le dénominateur est le polynôme $\alpha(\lambda)$ introduit dans la section précédente. Pour identifier la constante $N_{1,0,1}$ de (4.54), nous procédons de la même façon que dans la section 4.2.2. Pour que N soit défini pour toutes les valeurs non négatives de λ, le numérateur de (4.54) doit s'annuler quand $\lambda = \lambda_0 = 1/2 \times \left(\gamma - \nu - \delta + \sqrt{(\gamma + \nu + \delta)^2 - 4\delta\nu}\right)$ qui est la seule racine positive du dénominateur $\alpha(\lambda)$ (voir section 4.2.2). En utilisant la relation $(\lambda_0 + \delta)(\lambda_0 + \nu) = \gamma(\lambda_0 + \nu + \delta)$ nous trouvons $N_{1,0,1} = \nu/\lambda_0$. En utilisant la règle de l'Hôpital, nous obtenons l'expression de N dans le cas $\lambda = \lambda_0$. ■

Les deux nombres moyens (N_s et N), calculés dans cette section, devraient être des fonctions croissantes en δ et ν et décroissantes en λ et γ, ce qui est illustré par la figure 4.4. Dans la partie supérieure de la figure 4.4 où sont tracées les évolutions de N et N_s en fonction de λ (graphique supérieur gauche) et ν (graphique supérieur droit), nous pouvons vérifier que $N > N_s$. Les croix dans chacun des graphiques indiquent la valeur de N (le nombre moyen de répéteurs qu'un message traverse) correspondant à certaines valeurs de nos expérimentations qui seront décrites dans la section 5 ($\lambda = 1$,$\nu = 10$,$\delta = 20$,$\gamma = 50$). Dans la figure 4.4, les courbes pour $\delta = 30s^{-1}$,$\delta = 40s^{-1}$ (graphique inférieur gauche) et $\gamma = 9s^{-1}$ (graphique inférieur droit) montrent le comportement de notre modèle à la limite de l'instabilité. Dans ce cas, le nombre moyen de répéteurs tend vers l'infini.

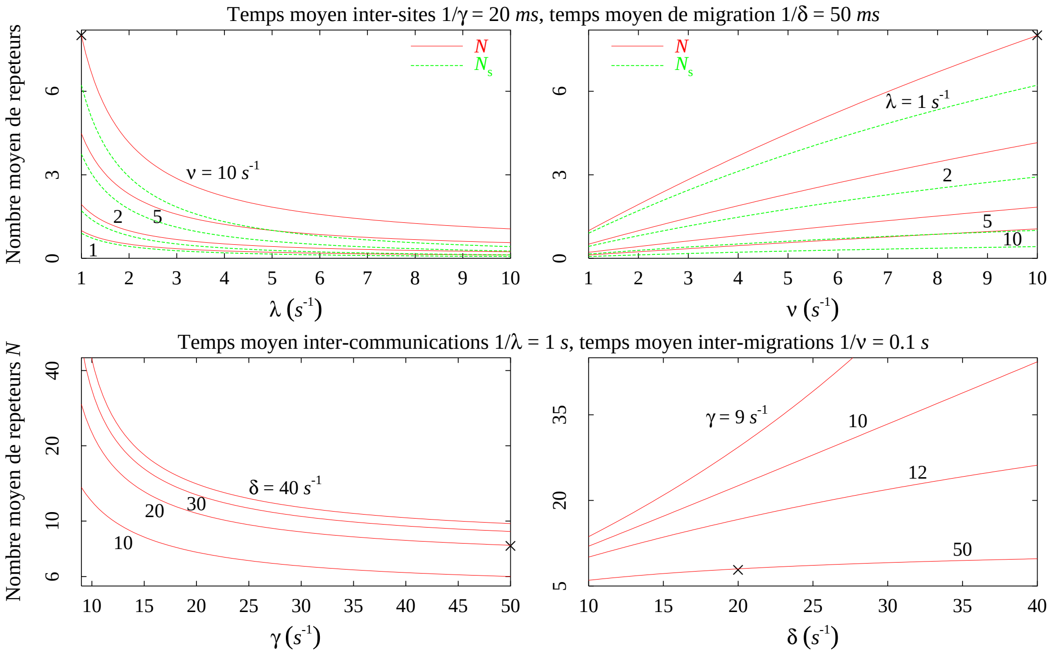


FIG. 4.4 – *Nombre moyen de répéteurs*

4.2.4 Influence des paramètres

Dans le cadre d'une utilisation pratique du modèle développé, il sera vraisemblablement difficile d'obtenir une valeur exacte pour l'ensemble des paramètres. Nous allons souvent avoir une incertitude sur ceux-ci ce qui aura une influence sur l'exactitude du résultat théorique obtenu. Nous allons dans cette section essayer de déterminer la réaction de notre modèle aux incertitudes des paramètres d'entrée. Pour quatre couples (λ, ν) nous allons introduire une erreur, exprimée en %, sur chacun de nos paramètres (λ, ν, δ, γ) et calculer l'erreur relative entre le résultat obtenu, noté T_{erreur}, et le résultat correct, noté $T_{théorique}$, donné par la formule suivante :

$$\frac{T_{erreur} - T_{théorique}}{T_{théorique}}$$

Une erreur relative positive (resp. négative) indique une surévaluation (resp. sous-évaluation) du paramètre ou du résultat. La figure 4.5 montre les résultats obtenus. Sur l'abscisse est indiquée l'erreur relative sur le paramètre considéré. L'axe des ordonnées indique l'erreur relative sur le résultat. La courbe en 0 correspond au cas où tous les paramètres sont justes. Nous avons indiqué le +10% et le −10% que nous jugeons être des erreurs acceptables. Remarquons tout d'abord qu'une erreur sur ν amène une erreur de même signe sur le résultat et que celle-ci est pratiquement linéaire. Le paramètre λ entraîne l'erreur la plus faible en général, tandis que l'erreur la plus importante est obtenue avec le paramètre γ dans trois cas (figure 4.5 (a),(b) et (c)) et δ pour le dernier (figure 4.5 (d)).

La table 4.3 donne la marge d'incertitude tolérée en entrée pour avoir en sortie entre −10% et 10% d'erreur. Il peut y avoir une très grande plage de valeurs, comme par exemple pour λ dans le cas $\lambda = 10$, $\nu = 1$ qui tolère jusqu'à 65% d'incertitude. La plus faible (10%) est obtenue pour δ quand $\lambda = 10$ et $\nu = 10$.

	$\lambda = 1$, $\nu = 1$	$\lambda = 1$, $\nu = 10$	$\lambda = 10$, $\nu = 1$	$\lambda = 10$, $\nu = 10$
λ	[−24%,31%]	[−18%,28%]	[−25%,65%]	[−26%,29%]
ν	[−16%,20%]	[−17%,22%]	[−24%,25%]	[−12%,22%]
δ	[−19%,29%]	[−19%,35%]	[−19%,29%]	[−13%,10%]
γ	[−13%,14%]	[−14%,17%]	[−13%,16%]	[−21%,19%]

TAB. 4.3 – *Marge d'incertitude des paramètres en entrée pour une erreur relative de* ±10%

Finalement la table 4.4 indique le sens d'évolution du temps de réponse suivant l'évolution d'un des paramètres.

Quand λ augmente le temps de réponse diminue car la source maintient une chaîne de répéteurs courte. Inversement, si le taux de migration (ν) augmente, le nombre moyen de répéteurs à traverser sera plus élevé, de même que la probabilité de communiquer avec

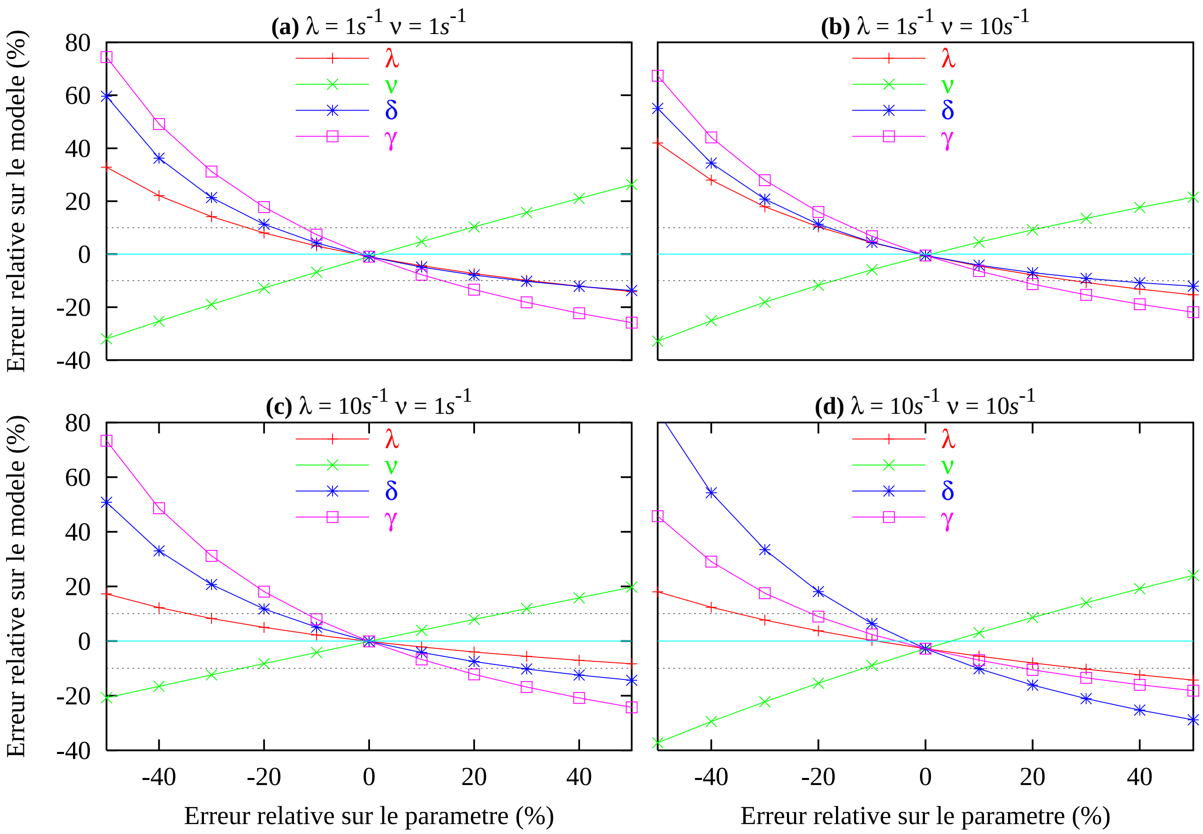


FIG. 4.5 – *Propagation des erreurs dans le modèle des répéteurs* $\delta = 10.9$, $\gamma = 115.6$

l'agent alors qu'il est en train de migrer ; cela entraînera une augmentation du temps de réponse. Le cas de γ s'explique aussi simplement en remarquant que si la latence réseau diminue (γ augmente), alors il faut moins de temps pour joindre l'agent et donc le temps de réponse diminue. Dans le cas de la durée de migration (δ), le résultat semble paradoxal car plus l'agent migre vite (δ augmente), moins il faut de temps pour le joindre. Il faut y voir l'importance de l'état (0,1,1) qui représente le cas où un message atteint un agent en train de migrer. Si cela se produit, alors non seulement le message devra attendre la fin de la migration, mais il devra en plus effectuer un nouveau saut pour rejoindre l'agent sur sa nouvelle localisation. En diminuant la durée de migration, nous diminuons cette attente ce qui a pour effet global de diminuer le temps de réponse. Intuitivement, nous voyons qu'il y aura une limite à ce phénomène en particulier suivant les valeurs des autres paramètres que sont la latence inter-sites et le taux de migration. Nous allons vérifier cela formellement.

Paramètre	Sens de variation	Sens de variation de T_F
Inverse attente de la source (λ)	$\nearrow$	$\searrow$
Inverse attente de l'agent (ν)	$\nearrow$	$\nearrow$
Inverse du temps de migration(δ)	$\nearrow$	$\searrow$
Inverse de la latence réseau (γ)	$\nearrow$	$\searrow$

TAB. 4.4 – *Évolution du temps de réponse en fonction de l'évolution des paramètres pour le serveur*

4.2.4.1 Influence de la durée de migration

Le temps de migration ($1/\delta$) doit être plus grand que 0 par définition. La condition de stabilité de notre modèle (4.24) est :

$$\frac{1}{\gamma} < \frac{1}{\nu} + \frac{1}{\delta} \tag{4.55}$$

qui peut être récrite en

$$\begin{cases} \delta & < & \dfrac{\nu\gamma}{\nu-\gamma} & \text{si } \nu-\gamma > 0 \\ \delta & > & \dfrac{\nu\gamma}{\nu-\gamma} & \text{si } \nu-\gamma < 0 \end{cases} \tag{4.56}$$

La deuxième condition est toujours vérifiée car nous avons $\dfrac{\nu\gamma}{\nu-\gamma} < 0$ et par définition $\delta > 0$, nous avons donc deux intervalles de définition pour δ

$$\delta \in \quad [0, \tfrac{\nu\gamma}{\nu-\gamma}[\quad \text{si } \nu-\gamma > 0 \tag{4.57}$$

$$\delta \in \quad [0, +\infty[\quad \text{si } \nu-\gamma < 0 \tag{4.58}$$

Proposition 4.2.10 (Migration infiniment lente)

$$\lim_{\delta\to 0} T_F = +\infty \tag{4.59}$$

Preuve : En utilisant la formule (4.29) et pour le cas $\lambda \neq \lambda_0$ nous obtenons par la règle de l'Hopital $\lim_{\delta\to 0} T_F = +\infty$. Dans le cas $\lambda = \lambda_0$ nous obtenons de la même façon $\lim_{\delta\to 0} T_F = +\infty$. ■

Proposition 4.2.11 (Migration infiniment rapide) *Si* $\nu - \gamma < 0$

$$\lim_{\delta\to\infty} T_F = \begin{cases} \dfrac{\gamma^2\lambda^2 + 3\gamma^2\lambda\nu + \gamma^2 + \nu^2 + 2\lambda\gamma\nu^2 + 2\lambda^2\nu\gamma + \gamma\nu^3 - \lambda\nu^3 - \lambda^2\nu^2}{\gamma\lambda(\lambda\gamma^2 + \gamma^2\nu - \lambda\nu^2 - \nu^3)}, & \lambda \neq \lambda_0 \\ \dfrac{-\lambda\nu^3 + 4\lambda\gamma^3 + \gamma^3\nu - \lambda^2\nu^2 + 4\lambda^2\nu\gamma + 3\lambda\gamma\nu^2 - \gamma^2\nu^2 + 2\gamma^4 - \gamma^2\lambda^2}{\gamma^3\lambda^2 + \gamma^3\lambda\nu - \lambda^2\nu^2\gamma - \gamma\lambda\nu^3}, & \lambda = \lambda_0 \end{cases} \tag{4.60}$$

Preuve : Dans le cas $\lambda \neq \lambda_0$ la formule 4.29 se décompose en trois termes

$$\frac{1}{\alpha(\lambda)}\Big(A - B - C\Big) \tag{4.61}$$

qui sont obtenus par une identification triviale. Nous avons

$$\lim_{\delta\to+\infty} \alpha(\lambda) = \lim_{\delta\to+\infty} \delta(\lambda + \nu - \gamma)$$

la limite pour λ_0 s'obtient en remarquant que

$$\begin{aligned}
\lim_{\delta\to+\infty} \gamma - \nu - \delta + \sqrt{(\gamma+\nu+\delta)^2 - 4\delta\nu} &= \lim_{\delta\to+\infty} \gamma - \nu - \delta \\
&\quad +\delta\left(1 + 2\frac{\gamma-\nu}{\delta} + \frac{(\nu+\gamma)^2}{\delta^2}\right)^{\frac{1}{2}} \\
&= \lim_{\delta\to+\infty} 2\gamma - 2\nu
\end{aligned}$$

ce qui nous donne $\lim_{\delta\to+\infty} \lambda_0 = \gamma - \nu$ et par suite directe $\lim_{\delta\to+\infty} T_{0,1,1} = \dfrac{2\gamma-\nu}{\gamma(\gamma-\nu)}$.

Nous avons donc

$$\lim_{\delta\to+\infty} \frac{A}{\alpha(\lambda)} = \frac{(\lambda+\nu)(2\gamma-\nu)}{\gamma(\lambda+\nu-\gamma)(\gamma-\nu)} \tag{4.62}$$

De la même façon nous obtenons

$$\lim_{\delta\to+\infty} \frac{B}{\alpha(\lambda)} = \frac{1}{\gamma(\lambda+\nu-\gamma)} \tag{4.63}$$

En mettant δ en facteur dans l'expression de C nous obtenons

$$\lim_{\delta\to+\infty} \frac{C}{\alpha(\lambda)} = \frac{\gamma\nu\lambda + (\lambda+\nu)(\lambda(\lambda+\nu) + \nu(\gamma+\nu))}{\lambda(\lambda+\nu)(\gamma+\nu)(\lambda+\nu-\gamma)} \tag{4.64}$$

Pour le cas $\lambda = \lambda_0$ nous décomposons la formule 4.29 en deux termes qui se traitent en mettant δ^2 et δ en facteur. ■

Proposition 4.2.12 *Si* $\nu - \gamma > 0$

$$\lim_{\delta\to\frac{\nu\gamma}{\nu-\gamma}} T_F = +\infty \tag{4.65}$$

Preuve : Remarquons que

$$\lim_{\delta\to\frac{\nu\gamma}{\nu-\gamma}} \alpha(\lambda) = 0^+$$

ce qui nous donne directement

$$\lim_{\delta \to \frac{\nu\gamma}{\nu-\gamma}} T_{0,1,1} = +\infty$$

et en remplaçant dans l'expression de T_F permet de conclure la preuve. ∎

Nous avons montré plusieurs résultats qu'il était difficile de trouver intuitivement. Le premier est que plus un agent met de temps à migrer ($\delta \to 0$) plus la source mettra de temps à le joindre (proposition 4.2.10) car elle devra attendre la fin de sa migration pour qu'il puisse recevoir un message.

Le deuxième est qu'il y a une limite au temps que met une source pour joindre un agent qui se déplace infiniment vite (proposition 4.2.11). Ce résultat semble paradoxal mais il ne faut pas oublier que nous travaillons seulement quand la condition de stabilité $\frac{1}{\gamma} < \frac{1}{\nu} + \frac{1}{\delta}$ est vérifiée. Si l'agent migre infiniment vite, alors la condition se récrit $\frac{1}{\gamma} < \frac{1}{\nu}$. Il est facile de voir que la condition de stabilité implique qu'un message prend moins de temps pour faire un saut que l'agent ne reste sur un site. Autrement dit, même si la migration est instantanée, le message rattrapera l'agent car celui-ci perdra du temps sur chaque site.

Enfin, nous avons montré que dans le cas où un agent attend en moyenne une durée inférieure à celle d'une communication réseau ($\nu - \gamma > 0$) avant d'effectuer une migration, alors il est possible que le temps pour le joindre soit infini.

4.2.4.2 Conclusion

Nous avons, dans cette section construit, un modèle d'un mécanisme de communication à base de répéteurs. Celui-ci utilise quatre paramètres (λ, ν, δ, γ), les deux premiers étant dépendants de l'application étudiée et les deux derniers dépendants de l'infrastructure d'exécution utilisée. Nous avons mis en évidence la sensibilité du modèle aux erreurs commises sur le délai inter-sites ($1/\gamma$) et sur la durée de migration ($1/\delta$).

Nous allons maintenant passer à l'étude du deuxième mécanisme de communication qui utilise un serveur centralisé.

4.3 Analyse Markovienne du serveur centralisé

4.3.1 Modèle

Nous allons modéliser le serveur avec priorité de traitement décrit dans la section 3.5.2 en considérant une unique file d'attente où seront mises les requêtes envoyées par une unique source et un unique agent. Les requêtes sont mises en attente dans cette queue en attendant de pouvoir être servies. Nous avons donc trois entités intervenant dans ce modèle : une source, un agent et un serveur de localisation.

Proposition 4.3.1 *Le mécanisme de communication à base de serveur est entièrement représenté par l'état de la source, de l'agent et du serveur*

Les messages reçus par un serveur sont de deux types

update location request : nouvelle localisation envoyée par l'agent

location request : demande de la position d'un agent par une source

De la description du serveur détaillée de la section 3.5.2 et en gardant à l'esprit que nous n'étudions qu'une source et qu'un unique agent, nous pouvons obtenir facilement les propriétés suivantes :

Propriété 4.3.1 *il ne peut y avoir au maximum qu'une unique* `location request` *dans la queue.*

Propriété 4.3.2 *il ne peut y avoir au maximum qu'une unique* `update location request` *dans la queue.*

Nous supposerons pour des raisons de simplification que le nombre maximum de requêtes dans le serveur est de 2 ce qui revient à supposer que lorsqu'un message de l'agent arrive alors que le serveur est en train de traiter le précédent, celui-ci le remplace. En d'autres termes le serveur est préemptif. Il est possible de s'affranchir de cette hypothèse au prix d'une augmentation de 30% du nombre des états de la chaîne de Markov, cependant d'après nos expérimentations le modèle n'y gagne que très peu en précision.

Nous supposons toujours l'hypothèse **H1** vérifiée (cf Section 4.1). Cependant, dans ce contexte p_j représentera la somme du j-ème temps de voyage de l'agent vers son nouveau site et le temps de voyage de la requête associée vers le serveur (`update location request`). Nous allons supposer que les temps de communication entre la source et la localisation présumée de l'agent (resp. entre la source et le serveur de localisation) sont des variables aléatoires indépendantes et identiquement distribuées (i.i.d.) suivant une loi exponentielle de paramètre $\gamma_1 > 0$ (resp. $\gamma_2 > 0$). Finalement nous supposons que les temps de service (quelque soit le type de requête) sont des variables aléatoires (i.i.d.) suivant une loi expoenentielle de paramètre $\mu > 0$ (dans le cas où il n'y a qu'une source et qu'un agent, μ est la vitesse de traitement du serveur). Toutes ces variables sont considérées être mutuellement indépendantes.

Nous allons modéliser le comportement du système en utilisant une chaîne de Markov finie dont le diagramme et les taux de transition sont donnés dans la figure 4.6. Un état est représenté par un triplet $(\mathbf{i},j,k)$ avec $\mathbf{i} \in \{A,B,\ldots,G\}$, $j \in \{0,0^*,1,1^*\}$ et $k \in \{0,1,1^*,2\}$, où $A,B,\ldots,G$ sont définis dans la figure 4.6. Pour tenir compte du fait que le serveur envoie une réponse à la source, nous avons dû séparer ses états en deux groupes, le premier contenant $A,\ldots,E$ indique un serveur en fonctionnement normal, alors que le deuxième contenant F,G dénote un serveur bloqué car en train d'envoyer une réponse.

Plus précisément, le vecteur $\mathbf{i} = (i_1,i_2)$ représente l'état de la queue du serveur, i.e. le type des messages (`update location request` ou `location request`) dans la queue, avec $i_l \in \{0,\lambda,\nu\}$ le type du message qui occupe la l-ème position dans la queue. Par convention, $i_l = 0$ (resp. $i_l = \lambda$, $i_l = \nu$) indique que la l-ème position n'est pas occupée (resp. la l-ème position est occupée par une `location request`, la l-ème position est occupée par une `update location request`).

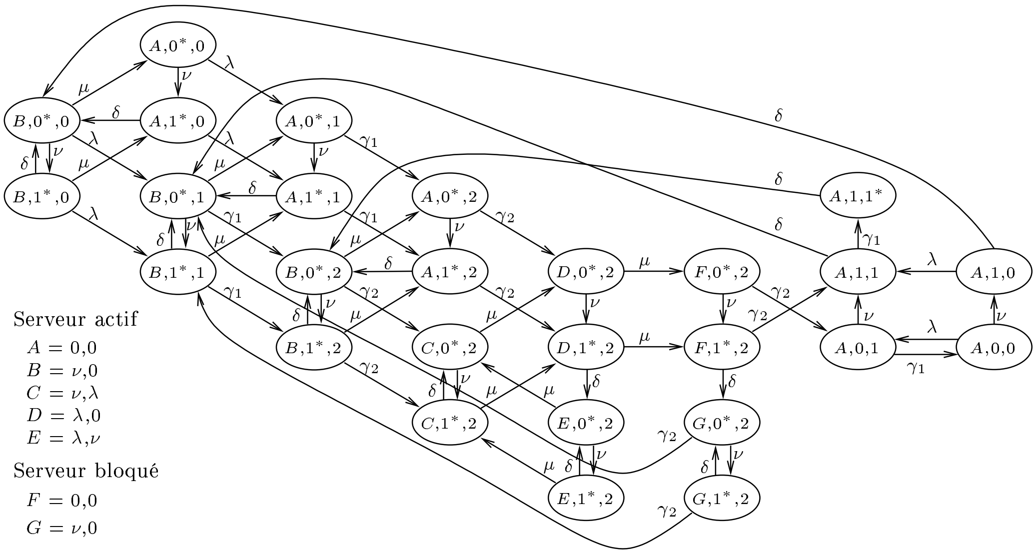


FIG. 4.6 – *État du système et taux de transitions dans le mécanisme du serveur.*

La composante $j \in \{0{,}0^*{,}1{,}1^*\}$ dans la description de l'état $(\mathbf{i},j,k)$ représente l'état de l'objet : $j = 1$ (resp. $j = 1^*$) indique que l'objet migre et que la source connaît (resp. ne connaît pas) la localisation de l'hôte que l'objet quitte; de manière similaire, $j = 0$ (resp. $j = 0^*$) indique que l'objet ne migre pas et que la source connaît (resp. ne connaît pas) sa localisation.

Finalement, la composante $k \in \{0{,}1{,}1^*{,}2\}$ dans la description de l'état $(\mathbf{i},j,k)$ représente l'état de la source : $k = 0$ si la source n'a pas d'activité, $k = 1$ si elle a envoyé un message à l'agent, $k = 1^*$ si le message envoyé par la source a atteint un site que l'agent est en train de quitter, et $k = 2$ si elle a envoyé un message au serveur pour localiser l'agent. Ce dernier cas arrive si la position supposée de l'agent n'est plus valide.

Le système entre dans l'état $(A,0,0)$ juste après la fin d'une communication (définie comme l'instant où un message atteint l'agent). Il reste dans cet état pendant une durée de distribution exponentielle et de moyenne $1/\lambda$ puis un nouveau message est généré par la source. Le temps qui s'écoule entre la génération d'un nouveau message et la visite suivante à l'état $(A,0,0)$ est le *temps de communication* (i.e. les quantités τ_i introduites dans la Section 4.1). En d'autres termes, les temps de communication successifs $\{\tau_i\}_i$ sont initialisés quand k passe de 0 à 1, et sont arrêtés quand k passe de 1 à 0. A chaque fois qu'une communication échoue, un message est envoyé au serveur et k passe de 1 à 2. Dès que le serveur répond, k repasse à 1 et le message est ré-émis par la source vers la nouvelle position de l'objet telle que donnée par le serveur (qui peut être ou ne pas être la bonne localisation).

D'après les hypothèses et la description précédentes, le processus de la figure 4.6 est une chaîne de Markov à espace d'etats fini et irréductible sur l'espace des états $\mathcal{F}$, où

$\mathcal{F}$ est l'ensemble de tous les états décrits dans la figure 4.6 ($\mathcal{F}$ contient 27 éléments). Soit $\mathbf{p} = \{p_{\mathbf{i},j,k}, (\mathbf{i},j,k) \in \mathcal{F}\}$ la probabilité stationnaire de cette chaîne de Markov ($\mathbf{p}$ existe car une chaîne de Markov finie et irréductible est ergodique). Si nous notons $\mathbf{Q}$ le générateur infinitésimal de ce processus Markovien (dont les éléments peuvent être facilement identifiés grâce à la figure 4.6), alors nous savons (voir e.g. [38]) que $\mathbf{p}$ est l'unique solution du système d'équations linéaire $\mathbf{p}.\mathbf{Q} = 0$, $\mathbf{p}.\mathbf{1} = 1$, qui peut être résolu en utilisant une analyse numérique.

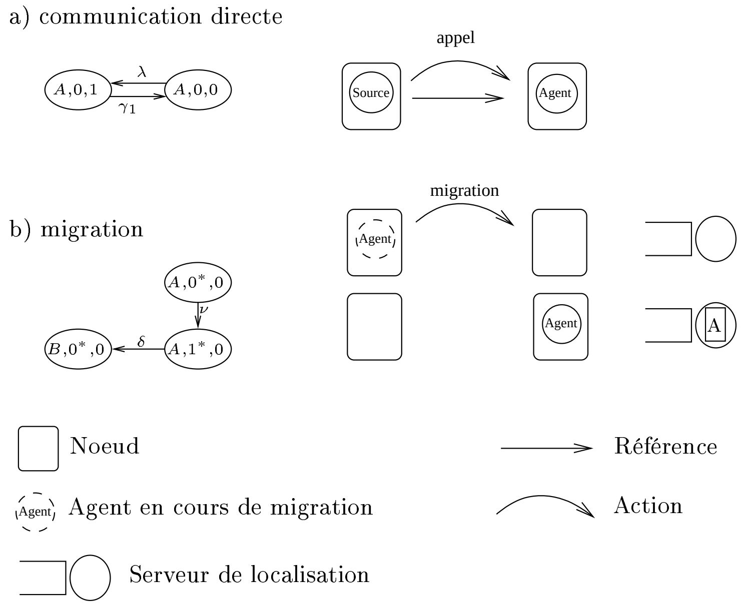


FIG. 4.7 – *Détails des transitions du modèle du serveur - source et agent*

Comme pour l'analyse des répéteurs, nous allons décrire en détail certaines transitions de notre modèle. Tout d'abord intéressons-nous à la figure 4.7 qui traite de la source et de l'agent.

figure 4.7 a) Le premier cas concerne une communication directe comme dans le cas des répéteurs. La source attend un temps moyen $1/\lambda$ avant de commencer sa communication et la finit après un temps moyen $1/\gamma_1$.

figure 4.7 b) Après avoir attendu un temps moyen $1/\nu$, l'agent décide de migrer ce qui lui prend un temps moyen $1/\delta$. À la fin de ce temps, un message a été envoyé au serveur qui commence à la servir immédiatement.

La figure 4.8 détaille quant à elle des spécificités du traitement des requêtes par le serveur.

figure 4.8 a) Lorsque la source a une ancienne référence vers un agent (indice $*$) et essaie de communiquer avec lui, elle est notifiée du problème. Dans cette situation, elle contacte un serveur de localisation (durée moyenne $1/\gamma_2$) et attend la réponse. Dans le cas où le serveur n'a aucune autre requête à traiter, il s'occupe de la requête de la source ce qui lui prend en moyenne $1/\mu$. Une fois ce traitement effectué, le serveur envoie la nouvelle localisation à la source (durée moyenne de $1/\gamma 2$), qui peut de nouveau essayer de contacter l'agent.

figure 4.8 b) Nous avons choisi d'implémenter le serveur avec une politique spéciale de façon à maximiser les performances (section 3.5.2). Cela se traduit notamment par le fait que si à la fin du service d'une requête de la source, avant de lui envoyer la réponse, le serveur a en queue une requête de l'agent, alors il remet celle de la source après celle de l'agent. Le serveur qui était dans l'état E passe alors dans l'état C.

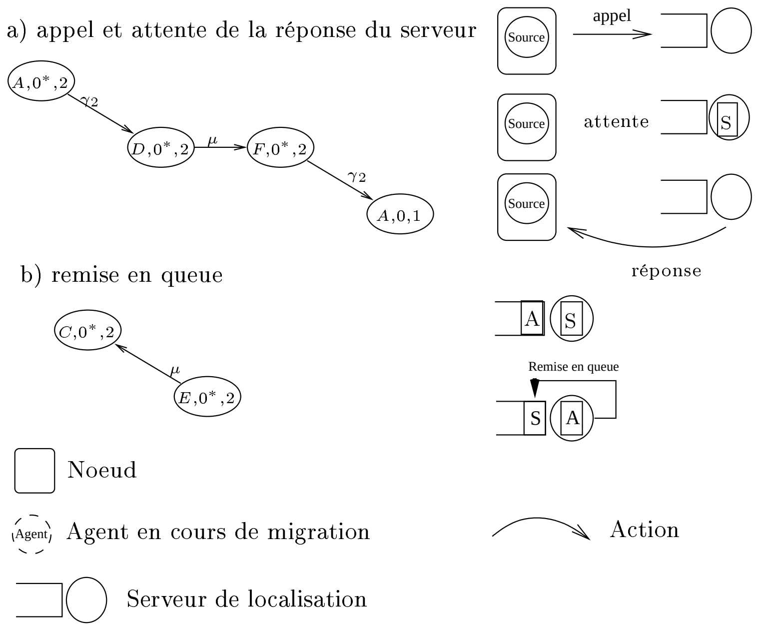


FIG. 4.8 – *Détails des transitions du modèle du serveur - le serveur*

Pour résumer nous avons dans notre modèle les paramètres décrits dans la table 4.5.

4.3.2 Temps moyen de communication

Nous nous intéressons au temps moyen de communication noté T_S (l'indice S dénote le serveur).

Définition 4.3.1 *Le temps moyen de communication pour le mécanisme de localisation utilisant un serveur est le temps pris par un message pour rejoindre un agent qui inclut le temps éventuel passé à demander la position de celui-ci à un serveur de localisation.*

Paramètre	Description
λ	Attente de la source
ν	Attente de l'agent
δ	Inverse de la durée de migration
γ_1	Inverse de la latence vers l'agent
γ_2	Inverse de la latence vers le serveur
μ	Taux de service

TAB. 4.5 – *Description des paramètres de la modélisation du serveur de localisation*

Une communication commence quand la source (après une période de repos de $1/\lambda$) envoie un message à l'objet mobile et se termine quand il l'atteint. En regardant la figure 4.6, il est facile de voir qu'un message peut seulement être généré quand le système est dans un des 6 états où $k = 0$. Dés qu'un message est généré k passe à la valeur 1^*. Ainsi, une communication peut seulement débuter quand le système est dans un des états suivants: $(A,0,1)$, $(A,1,1)$, $(A,0^*,1)$, $(A,1^*,1)$, $(B,0^*,1)$ ou $(B,1^*,1)$, et elle se termine quand le système atteint l'état $(A,0,0)$.

En notant $T_{\mathbf{i},j,k}$ le temps moyen pour atteindre l'état $(A,0,0)$ en partant de l'état $(\mathbf{i},j,k)$, nous avons la définition suivante

Définition 4.3.2 *Le temps moyen de réponse T_S du système est donné par*

$$\begin{aligned} T_S = {} & q_S(A,0,1)\, T_{A,0,1} + q_S(A,0^*,1)\, T_{A,0^*,1} + q_S(B,0^*,1)\, T_{B,0^*,1} \\ + {} & q_S(A,1,1)\, T_{A,1,1} + q_S(A,1^*,1)\, T_{A,1^*,1} + q_S(B,1^*,1)\, T_{B,1^*,1} \end{aligned} \tag{4.66}$$

où $q_S(\mathbf{i},j,1)$ est la probabilité qu'une communication commence quand le système est dans l'état $(\mathbf{i},j,1)$.

Cette formule s'obtient en suivant le même raisonnement que dans le cas des répéteurs. Nous allons dans un premier temps exprimer les probabilités $q_S(i,j,k)$ en fonction de nos paramètres.

Proposition 4.3.2

$$\begin{aligned}
q_S(A,0,1) &= \frac{\lambda}{\lambda+\nu} && (4.67)\\
q_S(A,1,1) &= \frac{\nu\lambda}{(\lambda+\delta)(\lambda+\nu)} && (4.68)\\
q_S(A,0^*,1) &= \frac{\nu\mu\delta}{(\lambda+\nu)(\lambda+\nu+\mu)(\lambda+\delta+\nu)} && (4.69)\\
q_S(A,1^*,1) &= \frac{\nu^2\mu\delta(2\lambda+\delta+\mu+\nu)}{(\lambda+\delta)(\lambda+\nu)(\lambda+\nu+\mu)(\lambda+\delta+\nu)(\mu+\lambda+\delta)} && (4.70)\\
q_S(B,0^*,1) &= \frac{\nu\delta}{(\lambda+\nu+\mu)(\lambda+\delta+\nu)} && (4.71)\\
q_S(B,1^*,1) &= \frac{\nu^2\delta}{(\lambda+\nu+\mu)(\lambda+\delta+\nu)(\mu+\lambda+\delta)}. && (4.72)
\end{aligned}$$

Preuve : Nous allons calculer $q_S(v)$, la probabilité qu'une transmission commence dans l'état $v \in \mathcal{V}$, où

$$\mathcal{V} := \{(A,0^*,1),(A,1^*,1),(B,0^*,1),(B,1^*,1)\}.$$

Soit $p(w;v)$ la probabilité qu'une communication commence dans l'état $v \in \mathcal{V}$ sachant que le système est dans l'état $w \in \mathcal{V} \cup \{(A,0^*,0),(A,1^*,0),(B,0^*,0),(B,1^*,0)\}$. Nous pouvons voir grâce à la figure 4.6 que

$$q_S(v) = \frac{\delta\nu}{(\lambda+\delta)(\lambda+\nu)}\, p((B,0^*,0);v) \tag{4.73}$$

pour $v \in \mathcal{V}$. Il reste à calculer $p((B,0^*,0);v)$ pour $v \in \mathcal{V}$. Les relations suivantes dérivent directement de la figure 4.6:

$$\begin{aligned}
p((A,0^*,0);v) &= \frac{\lambda}{\lambda+\nu}\, p((A,0^*,1);v) + \frac{\nu}{\lambda+\nu}\, p((A,1^*,0);v)\\
p((A,1^*,0);v) &= \frac{\lambda}{\lambda+\delta}\, p((A,1^*,1);v) + \frac{\delta}{\lambda+\delta}\, p((B,0^*,0);v)\\
p((B,0^*,0);v) &= \frac{\lambda}{\lambda+\nu+\mu}\, p((B,0^*,1);v) + \frac{\nu}{\lambda+\nu+\mu}\, p((B,1^*,0);v)\\
&\quad + \frac{\mu}{\lambda+\nu+\mu}\, p((A,0^*,0);v)\\
p((B,1^*,0);v) &= \frac{\lambda}{\lambda+\delta+\mu}\, p((B,1^*,1);v) + \frac{\delta}{\lambda+\delta+\mu}\, p((B,0^*,0);v)\\
&\quad + \frac{\mu}{\lambda+\delta+\mu}\, p((A,1^*,0);v)
\end{aligned}$$

pour tout $v \in \mathcal{V}$. La ré-écriture de ces équations sous forme matricielle donne

$$\begin{bmatrix} \lambda+\nu & -\nu & 0 & 0 \\ 0 & \lambda+\delta & -\delta & 0 \\ -\mu & 0 & \lambda+\nu+\mu & -\nu \\ 0 & -\mu & -\delta & \lambda+\delta+\mu \end{bmatrix} \begin{bmatrix} p((A,0^*,0);v) \\ p((A,1^*,0);v) \\ p((B,0^*,0);v) \\ p((B,1^*,0);v) \end{bmatrix} = \lambda \begin{bmatrix} p((A,0^*,1);v) \\ p((A,1^*,1);v) \\ p((B,0^*,1);v) \\ p((B,1^*,1);v) \end{bmatrix} \tag{4.74}$$

pour tout $v \in \mathcal{V}$. Tous les termes du coté droit des équations de (4.74) sont égaux soit à

v	$(A,0^*,1)$	$(A,1^*,1)$	$(B,0^*,1)$	$(B,1^*,1)$
$p((A,0^*,1);v)$	1	0	0	0
$p((A,1^*,1);v)$	0	1	0	0
$p((B,0^*,1);v)$	0	0	1	0
$p((B,1^*,1);v)$	0	0	0	1
$p((B,0^*,0);v)$	$\frac{(\lambda+\delta)\mu}{K}$	$\frac{(2\lambda+\delta+\mu+\nu)\nu\mu}{(\mu+\lambda+\delta)K}$	$\frac{(\lambda+\nu)(\lambda+\delta)}{K}$	$\frac{(\lambda+\nu)(\lambda+\delta)\nu}{(\mu+\lambda+\delta)K}$

TAB. 4.6 – *Valeur des probabilités (avec $K := (\lambda+\nu+\mu)(\lambda+\delta+\nu)$)*

0, soit à 1 comme indiqué dans la table 4.6 ci-dessous (lignes 1-4). En utilisant les lignes 1-4 de la table 4.6 et (4.74), nous pouvons calculer $p((B,0^*,0),v)$ pour tout $v \in \mathcal{V}$ (voir la dernière ligne de la table 4.6). Il suffit d'introduire les valeurs de $p((B,0^*,0);v)$ dans (4.73) pour obtenir (4.69)-(4.72) et conclure la preuve. ■

Il reste à calculer les temps $T_{\mathbf{i},j,k}$. Ils sont obtenus simplement en utilisant le générateur infinitésimal de la matrice $\mathbf{Q}$ comme suit (voir Thm 3.3.3 pp. 113-114 dans [49])

$$\begin{aligned} T_{A,0,0} &= 0 \\ \sum_{\mathbf{i},j,k} q_{(\mathbf{i}',j',k'),(\mathbf{i},j,k)} T_{\mathbf{i},j,k} &= -1 \quad \text{for } (\mathbf{i}',j',k') \neq (A,0,0) \end{aligned} \tag{4.75}$$

où $q_{(\mathbf{i}',j',k'),(\mathbf{i},j,k)}$ sont éléments de la matrice génératrice $\mathbf{Q}$.

Pour simplifier l'écriture de (4.75), nous introduisons $\mathbf{M}_{A,0,0}$ comme mineur (comatrice) de la matrice $\mathbf{Q}$ obtenue en supprimant la ligne et la colonne correspondant à l'état $(A,0,0)$. Soit $\mathbf{T} = \{T_{\mathbf{i},j,k}, (\mathbf{i},j,k) \in \mathcal{F} - \{(A,0,0)\}\}$ le vecteur du temps, sauf pour $T_{A,0,0}$ (qui est nul par définition). L'équation (4.75) devient

$$\mathbf{M}_{A,0,0} \cdot \mathbf{T} = -\mathbf{1}. \tag{4.76}$$

En résolvant pour (4.76) et en utilisant (4.67)-(4.72) nous obtenons finalement T_S. La forme finale de T_S a été obtenue à l'aide d'outils de traitement symbolique [71, 74] mais ne présente pas d'intéret particulier c'est pourquoi elle n'est pas indiquée ici.

4.3.3 Influence des paramètres

Nous procédons maintenant à la même analyse que dans la section 4.2.4, c'est à dire étudier la sensibilité du modèle aux incertitudes sur les paramètres. La figure 4.9 détaille les résultats pour quatre couples (λ,ν). Plusieurs observations peuvent être faites. La première est que le taux de service (μ) a une très faible influence sur le modèle. Il faut une erreur relative négative importante pour influer sur le temps de réponse. Dans le cas d'une erreur positive il n'y a aucun changement quelle que soit celle-ci. En effet, notre serveur n'est absolument pas saturé et donc ne constitue pas le facteur limitant dans notre modèle. Nous adresserons dans la section 4.4.2 le cas plus général d'un serveur avec un taux d'utilisation élevé. La deuxième est que ν a une influence linéaire et de même signe (une erreur négative initiale donne une erreur négative au final) contrairement à λ, δ, γ_1 et γ_2. La table 4.7 donne la marge d'incertitude tolérée en entrée pour avoir en sortie entre

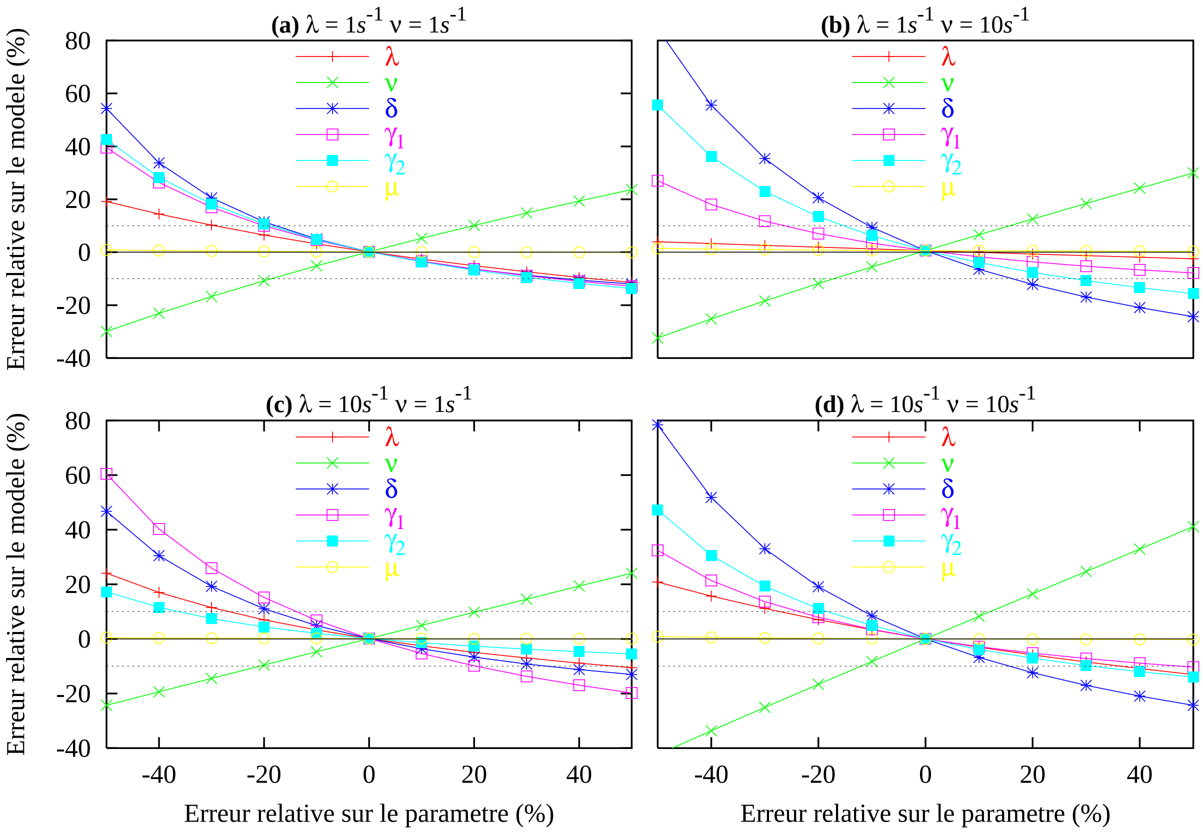


FIG. 4.9 – *Propagation des erreurs dans le modèle du serveur* $\delta = 10.9$, $\gamma_1 = 115.6$, $\gamma_2 = 76.3$, $\mu = 2325$, $t = 10000$

-10% et 10% d'erreur. Remarquons que le modèle tolère une incertitude très importante sur λ, en particulier quand $\lambda = 1, \nu = 10$ alors que ce n'est pas le cas pour ν et δ.

	$\lambda = 1, \nu = 1$	$\lambda = 1, \nu = 10$	$\lambda = 10, \nu = 1$	$\lambda = 10, \nu = 10$
λ	$[-29\%, 42\%]$	$[-50\%, 50\%]$	$[-27\%, 47\%]$	$[-28\%, 36\%]$
ν	$[-19\%, 20\%]$	$[-17\%, 16\%]$	$[-21\%, 21\%]$	$[-12\%, 12\%]$
δ	$[-12\%, 37\%]$	$[-11\%, 16\%]$	$[-19\%, 34\%]$	$[-12\%, 16\%]$
γ_1	$[-20\%, 35\%]$	$[-27\%, 72\%]$	$[-16\%, 20\%]$	$[-24\%, 48\%]$
γ_2	$[-19\%, 32\%]$	$[-16\%, 28\%]$	$[-34\%, 152\%]$	$[-22\%, 31\%]$
μ	$[-93\%, \infty[$	$[-91\%, \infty[$	$[-77\%, \infty[$	$[-91\%; \infty[$

TAB. 4.7 – *Marge d'erreur des paramètres en entrée pour une erreur relative de plus-moins 10%*

Comme dans le cas des répéteurs, la Table 4.8 indique qu'une augmentation du taux de migration de l'agent ν entraîne une augmentation du temps de réponse. Dans tous les autres cas (λ, δ, γ_1, γ_2 et μ), celui-ci varie d'une façon opposée : une augmentation d'un paramètre entraîne une diminution du temps de réponse. Comme dans le cas des répéteurs, une diminution de la latence (augmentation de γ_1 et γ_2) avantage le temps de réponse qui diminue. Si le serveur traite les requêtes rapidement (μ élevé), alors la source pourra tenter une nouvelle communication plus tôt ce qui aura aussi pour effet de diminuer le temps de réponse. Finalement, dans le cas d'une augmentation de δ (diminution du temps de migration), il y aura une diminution du temps de réponse car le système se trouvera moins souvent dans l'état $(A,1,1^*)$ où la source doit attendre la fin de la migration avant de pouvoir continuer.

Paramètre	Sens de variation	Sens de variation de T_S
Attente de la source (λ)	$\nearrow$	$\searrow$
Attente de l'agent (ν)	$\nearrow$	$\nearrow$
Inverse de la durée de migration (δ)	$\nearrow$	$\searrow$
Inverse de la latence vers l'agent (γ_1)	$\nearrow$	$\searrow$
Inverse de la latence vers le serveur (γ_2)	$\nearrow$	$\searrow$
Taux de service (μ)	$\nearrow$	$\searrow$

TAB. 4.8 – *Évolution du temps de réponse en fonction de l'évolution des paramètres pour le serveur*

4.3.3.1 Conclusion

Le modèle du serveur centralisé obtenu dans cette section utilise six paramètres (λ, ν, δ, γ_1, γ_2, μ). Deux d'entre eux (λ et ν) sont dépendants de l'application considérée tandis que les autres sont dépendants des conditions réseaux et matérielles. Nous avons montré que le modèle est particulièrement sensible aux erreurs commises sur la durée de migration ($1/\delta$) et sur le temps de communication avec le serveur ($1/\gamma_2$). Par contre, la vitesse de traitement du serveur ($1/\mu$) semble n'avoir que peu d'influence sur le résultat final.

4.4 Extension au cas multi-sources multi-agents

L'étude précédente a comme principale limitation qu'elle ne concerne que des applications constituées d'une source communicant avec un agent. Nous allons dans cette partie essayer d'étendre nos résultats aux cas multi-sources multi-agents. Plus précisément nous allons étudier des applications composées de n couples source-agent et voir les conséquences que cela entraîne sur nos modèles.

4.4.1 Répéteurs

Le modèle des répéteurs peut être très facilement étendu en remarquant que dans le cas de n couples source-agent, il y a indépendance des chaînes de répéteurs. En effet, la chaîne ne peut augmenter ou diminuer que sur action de la source ou de l'agent à ses extrémités. Donc une application composée de n couples sera donc composée de n chaînes de répéteurs indépendantes. Notre modèle reste donc valide dans le cas général.

4.4.2 Serveur

Nous allons considérer n couples source-agent et un serveur composé de n files d'attentes. A chacun des couples est attribué une file d'attente pour éviter un mélange des requêtes et nous permettre d'appliquer la politique de service définie précédemment comme indiqué dans la figure 4.10.

Chaque queue peut donc avoir jusqu'à deux requêtes, une unique `update request` venant d'un agent et une unique `location request` d'une source essayant de communiquer avec l'agent. L'unique serveur passe d'une queue à l'autre de manière cyclique, servant une unique requête dans chaque queue. Un tel système est bien modélisé par un unique serveur avec des services cycliques non exhaustifs, c'est à dire un serveur qui sert une requête par queue et passe à la queue suivante, par opposition à un serveur qui viderait complètement une queue avant de continuer. Pour avoir de bons résultats, il faut représenter le fait que le serveur reste bloqué quand il doit envoyer une réponse à une source. Dans une telle situation, le serveur n'est pas disponible alors que des clients pourraient être en attente dans une ou plusieurs queues. Pour modéliser cela, nous avons deux possibilités : soit considérer un serveur avec vacances [67], ce qui est difficile à modéliser, soit considérer que le temps de "blocage" fait partie du temps de service d'une `location request`. Dans cette dernière situation, il faudrait considérer deux temps de services distincts (μ_1 pour

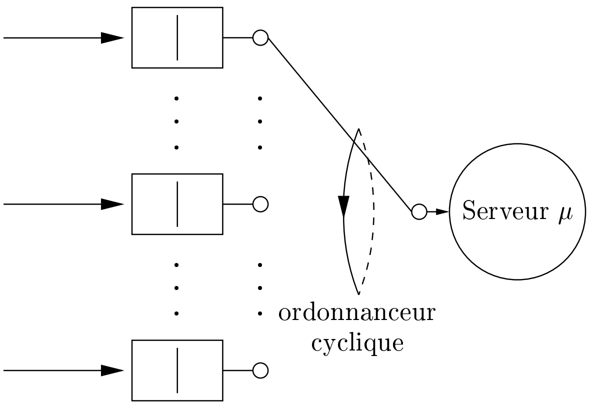


FIG. 4.10 – *Serveur pour n couples.*

les requêtes de l'agent, μ_2 pour les requêtes de la source). Nous allons plutôt considérer un modèle alternatif dans lequel nous avons plusieurs serveurs avec chacun une queue unique.

Un serveur avec vacances décrit un serveur avec une unique queue dans lequel le serveur peut être indisponible alors même qu'il y a des clients en attente. Dans le cas du serveur multi-queue, le temps passé à ne pas servir une queue particulière peut être vu comme une vacation pour celle-ci. Cette remarque nous permet de simplifier grandement l'analyse d'un tel serveur. La décomposition du serveur multi-queues en plusieurs serveurs à queue unique est donné dans la figure 4.11.

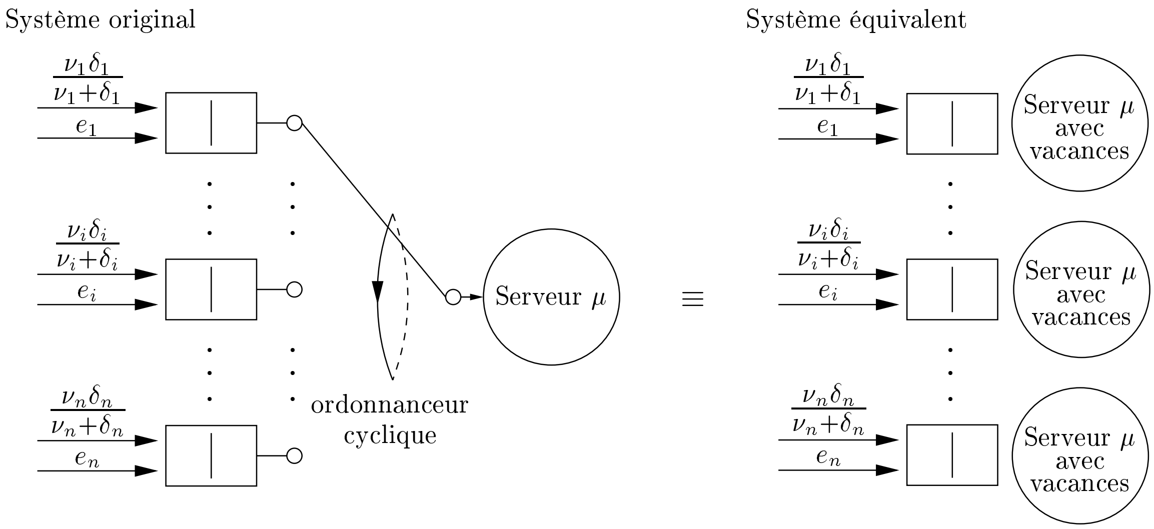


FIG. 4.11 – *Équivalence entre un serveur à n queues et n serveurs.*

La figure 4.11 montre à gauche le système réel avec un service cyclique et son équivalent composé de queues indépendantes et de serveurs avec vacances à droite. Il y a deux flots en entrée, un venant de l'agent et l'autre de la source. Les taux d'arrivée dans la queue i ($i = 1,\ldots,n$ avec n nombre total de queues) sont alors de $\nu_i\delta_i/(\nu_i + \delta_i)$ pour l'agent i (`update requests`) et e_i (`location requests`) pour la source i. Le taux e_i et les deux premiers moments des périodes de vacances qui sont nécessaires pour analyser un tel serveur sont malheureusement difficiles à calculer. Le modèle développé dans la

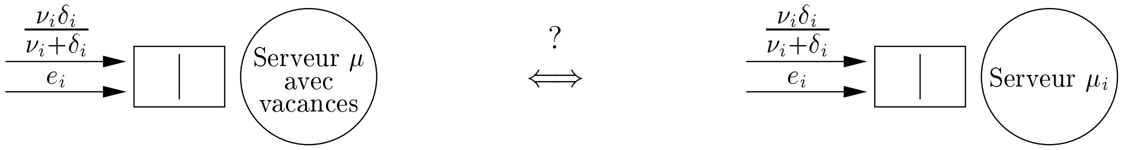


FIG. 4.12 – *Équivalence entre les deux systèmes.*

section 4.3.1 peut être utilisé si nous arrivons à trouver l'équivalence indiquée dans la figure 4.12. Dans la partie gauche de la figure 4.12 est dessiné un serveur à queue unique avec vacances tandis que dans la partie droite est indiqué le système équivalent avec un serveur travaillant à vitesse réduite μ_i. Le point important ici est qu'une telle équivalence est possible si la vitesse de traitement μ_i de la queue i dans le système équivalent peut être estimée. Nous aurons nécessairement $\mu_i \leq \mu$ pour compenser les périodes de vacance. L'égalité $\mu_i = \mu$ est obtenue quand il y a un unique couple source-agent dans le système. Remarquons que dans le cas où le nombre de couples source-agent n'est pas très élevé, il y aura souvent une unique queue active à cause d'un phénomène de multiplexage. Dans ce cas, il suffit de considérer $\mu_i \approx \mu$.

Les systèmes dessinés dans la figure 4.12 sont supposés se comporter de manière équivalente en ce qui concerne les temps d'attente et les temps de réponse. Soit $T_{vacance}$ (resp. $T_{réduit}$) le temps de réponse du système avec vacances (resp. le système avec service réduit). La durée $T_{vacance}$ dépend des paramètres ν_i,δ_i,e_i,μ et des deux premiers moments de la période de vacance. Le temps $T_{réduit}$ dépend des paramètres ν_i,δ_i,e_i et μ_i qui doivent être calculés. En écrivant $T_{vacance} = T_{réduit}$ nous pouvons calculer μ_i en fonction des termes ν_i,δ_i,e_i,μ et des deux premiers moments de la période de vacances. Malheureusement il est difficile d'exprimer e_i et les deux premiers moments de la période de vacance et rien ne peut être fait sans ces paramètres.

Il est possible d'estimer la valeur de μ_i en mesurant le temps de réponse du serveur au niveau de la source. Si la source note le moment st d'envoi d'une requête `location request` et le moment de la réception de la réponse rt elle peut alors mesurer le temps de réponse avec $rt - st - RTT$ où RTT est le temps d'aller retour d'un message dans le réseau. Le taux de serveur est alors approximativement de $1/(rt - st - RTT)$. Dans le cas d'un système avec un unique couple source-agent, nous avons $1/(rt - st - RTT) \lesssim \mu = \mu_i$. Cette approximation ne fonctionne que dans le cas d'un réseau à haut débit, par exemple un

LAN. En effet, quand un serveur envoie une réponse à une source, il reste bloqué le temps que celle-ci atteigne sa destination (rendez-vous entre les objets). Donc quand le ratio $\frac{RTT}{temps\ de\ service}$ augmente, l'approximation devient pire car le temps de blocage est considéré comme du temps de service. Nous vérifierons le comportement de nos approximations dans la section 5.4.

4.5 Conclusion

Nous avons dans cette section développé des modèles formels pour le mécanisme de communication à base de répéteurs et celui à base de serveurs. Dans le premier cas, nous avons réussi à obtenir une formule close pour le temps de réponse. Obtenir une expression similaire pour le serveur implique une inversion de matrice de taille 27 ce qui peut se faire en utilisant des outils de traitement symbolique [71, 74]. Nous avons étudié l'influence des paramètres dans nos deux modèles ce qui nous a permis d'une part de mettre en évidence ceux qui ont le plus d'influence sur le résultat, et d'autre part de définir des intervalles de tolérance.

Finalement les résultats obtenus nous permettent d'envisager plusieurs applications. Tout d'abord, et c'est ce que nous recherchions, nous avons deux expressions des temps de réponse ce qui devrait nous permettre de prédire les performances, et ainsi choisir l'un ou l'autre des mécanismes en fonction de nos besoins. Dans le cas des répéteurs, nous avons trouvé deux expressions pour la longueur de la chaîne, la première indiquant le nombre moyen de répéteurs entre une source et un agent et la seconde le nombre moyen de répéteurs traversé par un message. Il est ainsi possible d'avoir une estimation de la robustesse de la chaîne de répéteurs qui dépend directement de sa longueur, comme indiqué dans [11]. Finalement, nous pouvons envisager une troisième utilisation en essayant de répondre à la question suivante: quel est le temps moyen que peut attendre une source entre deux communications si elle veut pouvoir joindre un agent avec un temps moyen de t millisecondes?

Chapitre 5

Validation par simulations et expérimentations

Nous avons dans la partie précédente développé deux modèles dont nous pensons qu'ils reflètent bien les mécanismes que nous étudions. Cependant, pour en être certain, nous avons besoin d'effectuer une comparaison de ces modèles avec des résultats réels. La théorie développée dans les sections 3.5.1 et 3.5.2 repose sur plusieurs hypothèses. Nous avions supposé que le temps d'attente de la source et de l'agent, le temps de migration, le temps de voyage des messages dans le réseau et le temps de service (approche centralisée seulement) étaient des variables aléatoires exponentielles indépendantes entre elles. Il est bien évident que la plupart de ces hypothèses ne seront pas vérifiées en pratique et que nous devons donc nous attendre à une déviation entre la théorie et la pratique. Cependant, nous devons être certain que, si déviation il y a, elle est due non pas à une erreur dans nos modèles mais bien au non respect des hypothèses. Pour vérifier cela nous avons entrepris un long travail de validation en deux étapes. Un simulateur des systèmes étudiés a été écrit en Java et utilisé pour simuler les deux mécanismes dans diverses conditions. Nous avons ensuite procédé à des expérimentations en grandeur réelle. Tout au long de cette section nous considérons deux conditions de test, un réseau local et un réseau régional. Nous présentons à chaque fois les résultats pour un LAN mais par souci de lisibilité nous avons préféré mettre les résultats pour le MAN dans l'annexe C page 145 sauf quand ils apportaient une information importante.

5.1 Simulation événementielle

Nous allons procéder à une validation de nos modèles en utilisant des résultats obtenus grâce à un simulateur à événements discrets. Nous insistons sur le fait que ce simulateur ne simule en aucune façon les chaînes de Markov décrites précédemment mais les systèmes de communication étudiés dans la section 3.5. Ainsi, il s'agit vraiment d'un simulateur de ProActive, ou pour être plus exact, des mécanismes de migration et de communication (la partie rendez-vous uniquement). Celui-ci a été construit spécifiquement pour nos besoins et nous allons tout d'abord étudier son architecture.

5.1.1 Architecture du simulateur

Le simulateur a été entièrement écrit en Java [30] et est composé des objets suivants :

Source	Agent
Forwarder	ForwarderChain
Simulator	RandomNumberGenerator

Ces objets ne sont pas tous présents à l'exécution, leur présence dépendant du mécanisme simulé. La figure 5.1 montre deux configurations de fonctionnement. Nous avons

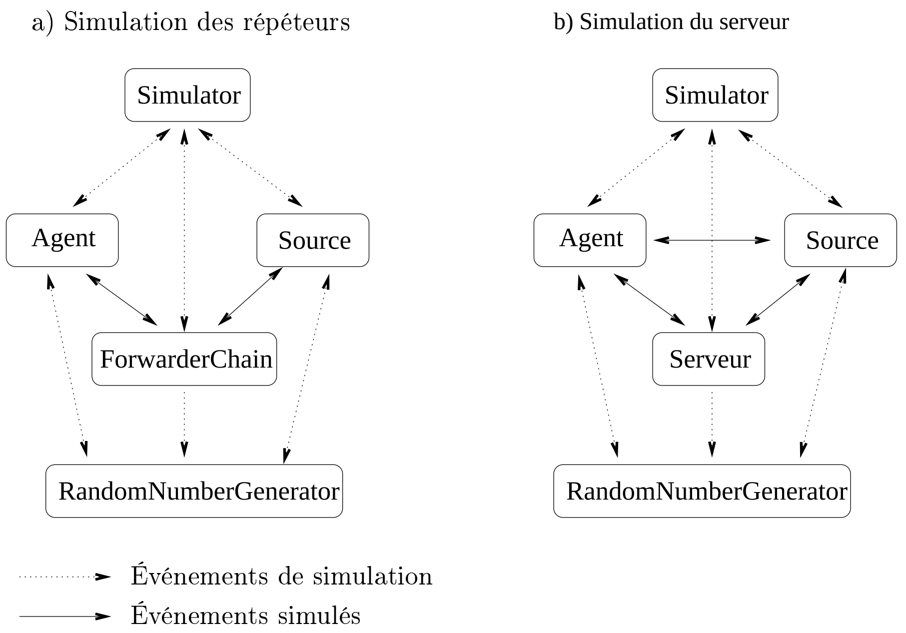


FIG. 5.1 – *Architecture du simulateur*

indiqué deux types de relation entre objets. La première, notée *Événements de simulation*, concerne le fonctionnement interne du simulateur. Tous les objets du simulateur ont ainsi besoin du *RandomNumberGenerator* qui leur fournit les valeurs aléatoires leur permettant d'affecter une durée à chacun de leurs états. Le *Simulator* fonctionne comme le cœur, demandant la mise à jour des objets à la fin d'une période de temps. Chaque état d'un objet dure un temps décidé par le générateur aléatoire. Le rôle du *Simulator* est de vérifier parmi tous les objets qu'il gère celui qui doit changer d'état le plus tôt et de faire "avancer" le temps courant jusqu'à cet instant. Une fois la mise à jour de tous les objets effectuée, le temps est de nouveau avancé jusqu'au prochain événement.

Au dessus du moteur de simulation se trouve le mécanisme simulé qui utilise ce que nous avons noté les *Événements simulés*. Chaque objet maintient son propre état et, quand le moteur de simulation demande une mise à jour à la suite d'un événement, le nouvel état est décidé en fonction de l'état courant de l'objet et de ceux dont il dépend. Par exemple dans le cas des répéteurs l'état de la source dépend de l'état de la chaîne des répéteurs (*ForwarderChain*) qui elle-même dépend de l'état de l'agent.

Nous avons choisi de laisser chaque objet gérer son état au lieu que ce soit au simulateur de le faire, ce qui nous a permis une simplification du code et une meilleure lisibilité ; en effet le code écrit ressemble énormément à celui de ProActive sans la partie fonctionnelle. Il nous est ainsi possible de vérifier visuellement que l'enchaînement des états d'un objet était conforme à celui de l'objet réel.

Finalement, cette approche nous permet d'étendre rapidement le simulateur pour, par exemple, modifier le comportement des objets.

5.1.2 Vérification du simulateur et des modèles

La figure 5.2 montre le temps de réponse mesuré dans la simulation du mécanisme centralisé (resp. répéteurs) comparé à celui donné par le modèle du serveur dans la section 4.3.2. Nous avons simulé une application qui s'exécute pendant 10 000 secondes en utilisant les valeurs obtenues lors d'expériences sur un LAN pour nos paramètres (cf. section 5.2). Les deux variables considérées sont le taux de communication de la source (λ) et le taux de migration de l'agent (ν). Une des variables avait une valeur fixe, soit 1 soit 100 (figures 5.2 (a) et (b) pour λ et 5.2 (c) et (d) pour ν) tandis que l'autre variait entre 1 et 100 afin d'étudier le comportement de notre simulateur et du modèle avec des valeurs "normales" ($\lambda = 1$, $\nu = 1$) et "extrêmes" ($\lambda = 100$, $\nu = 100$). La figure 5.3 donne en plus du temps de réponse le nombre moyen de répéteurs pour chacun des couples λ, ν considérés.
Les résultat obtenus sont excellents dans pratiquement tous les cas, aussi bien pour le serveur que pour les répéteurs. Seul le temps de réponse pour le serveur avec $\nu = 100$ présente une erreur relative plus élevée (figure 5.2 (d)).

5.1.3 Robustesse des modèles

Maintenant que nous avons vérifié que nos modèles et notre simulateur concordaient, nous allons tester la robustesse de nos modèles aux violations de nos hypothèses. Pour cela il nous a fallu déterminer la distribution des temps dans le cas d'applications réelles. Grâce aux expérimentations que nous présenterons dans la section 5.2 nous avons obtenu un ensemble de mesures dont nous avons tenté de déterminer la distribution. Pour chacun des paramètres mesurés (latence, temps de migration, ...) nous avons effectué un test de "goodness of fit" pour déterminer sa distribution probable. Le taux de communication (inverse de λ) et le taux de migration (inverse de ν) ne sont pas considérés ici car ils sont dépendants de l'application. Les résultats sont présentés dans la table 5.1. À part

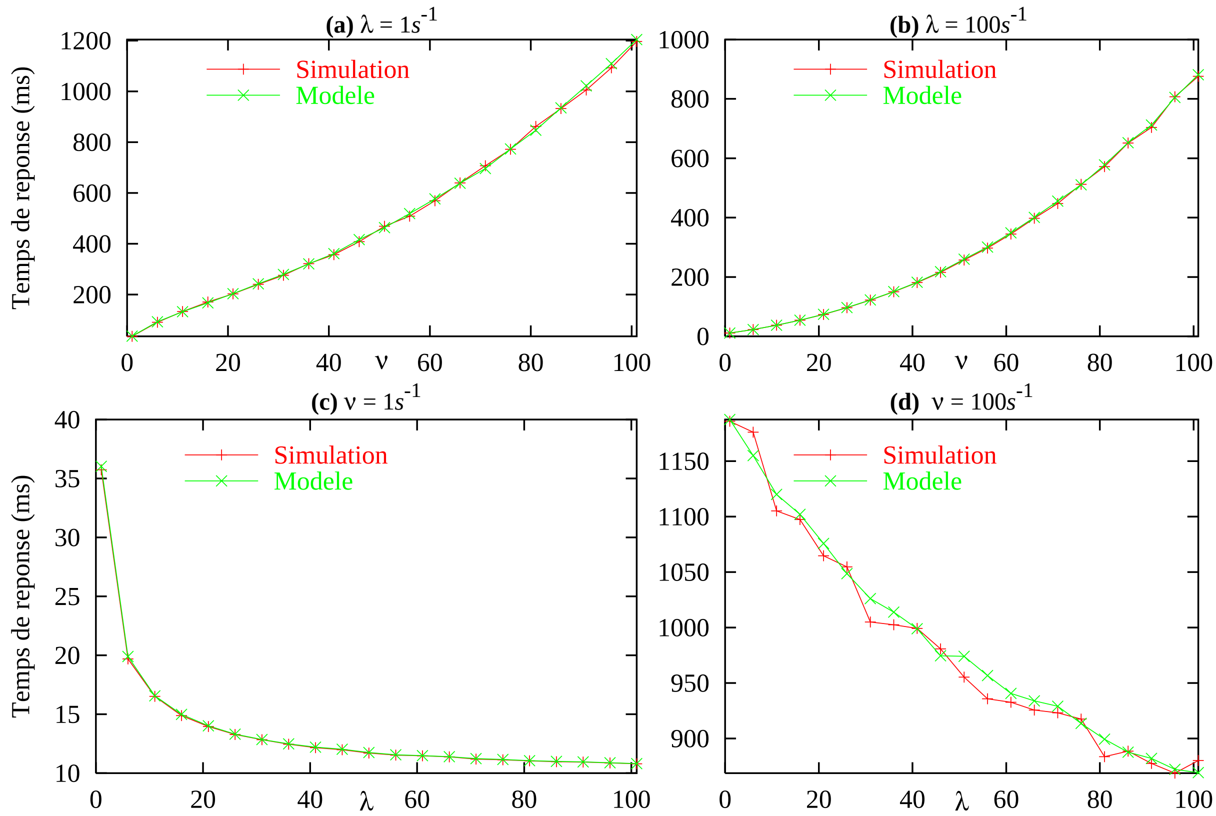


Fig. 5.2 – *Simulateur et modèle du serveur* ($\delta_S = 14.5$, $\gamma_1 = 115.6$, $\gamma_2 = 76.3$,$\mu = 2325$, durée $= 10\ 000s$)

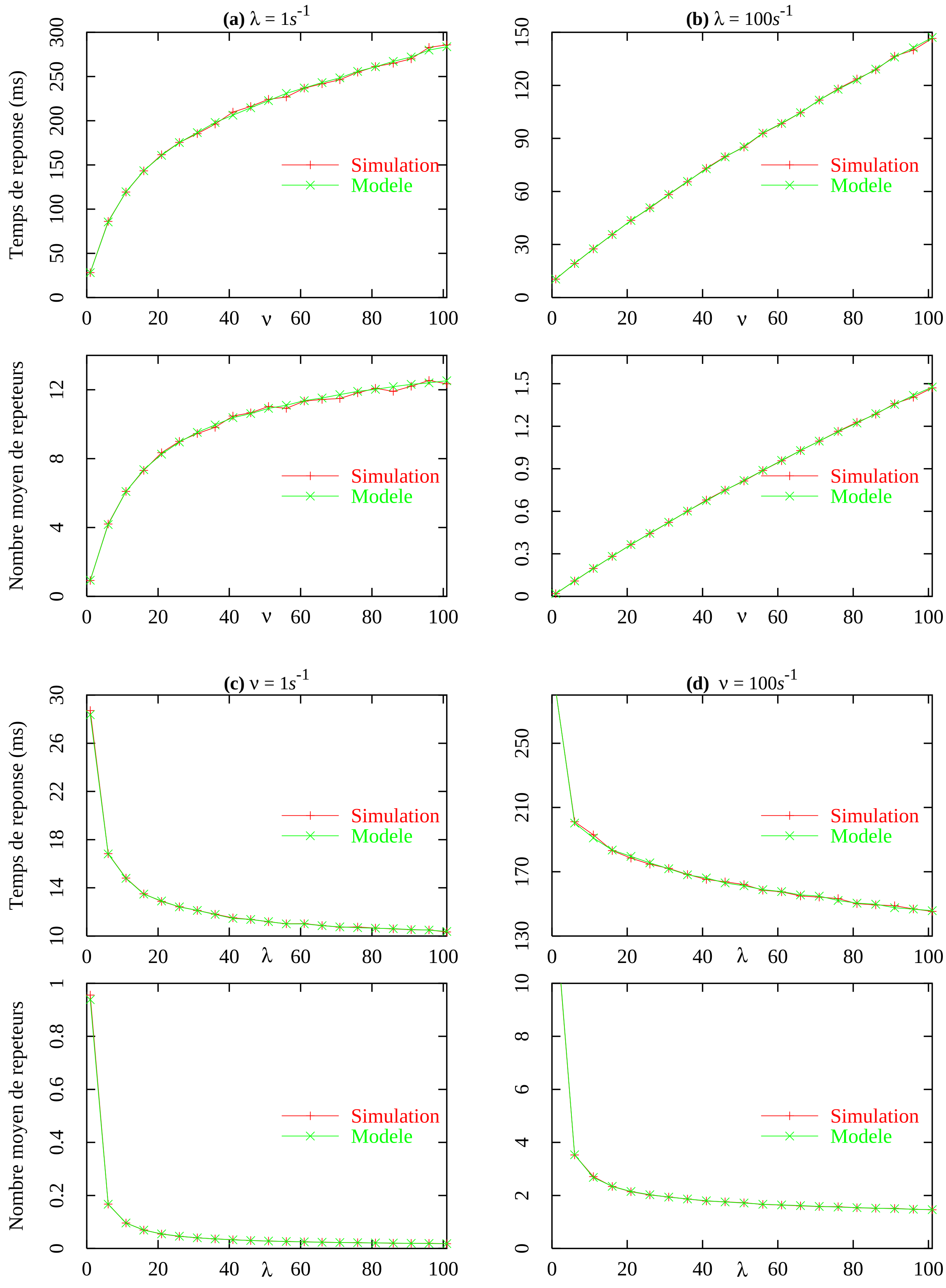


FIG. 5.3 – *Simulateur et modèle des répéteurs* ($\delta_F = 15.16$, $\gamma = 45.6$, durée $= 10\ 000s$)

les temps de service qui sont constants, tous les autres paramètres dépendant du réseau sont bien représentés par une distribution de Weibull. Maintenant que la distribution a été identifiée, nous pouvons tester la robustesse du modèle en utilisant le simulateur. Nous avons effectué des simulations où toutes les variables considérées sont exponentielles sauf une dont la distribution est une Weibull (ou constante pour les temps de service). Finalement, nous avons simulé un système où toutes les variables sont non-exponentielles. Dans ces essais, la seule hypothèse initiale encore respectée concerne l'indépendance des processus considérés.

La table 5.2 donne la moyenne de l'échantillon et les pourcentiles d'erreur relative (exprimés en pourcentage) entre les résultats simulés et les résultats théoriques pour le temps de réponse. Nos deux modèles apparaissent comme très robustes aux durées de communication de distribution Weibull. Dans 95% des simulations menées, l'erreur relative sur le temps de communication est restée au dessous de 7.1% (resp. 2.3% et 2.7%) dans le mécanisme des répéteurs (resp. serveur) (voir ligne 4 (resp. lignes 9 et 10) dans la table 5.2).

Les deux modèles ne sont pas sensibles à la distribution des temps d'attente de l'agent. Dans 95% des simulations menées l'erreur relative sur le temps de communication reste inférieure à 4.2% (resp. 4.8%) des valeurs obtenues par simulation dans le mécanisme des répéteurs (resp. serveur centralisé) (voir ligne 2 (resp. ligne 7) dans la table 5.2). Le modèle du serveur de localisation est très robuste aux temps de service déterministes. L'erreur la plus grande observée durant les simulations est égale à 3.7%.

Toutefois, les modèles sont plus sensibles à la distribution des durées de migration et des temps d'attente de la source. Néanmoins, dans la simulation des répéteurs l'erreur moyenne relative est égale à 7,7% (resp. 8.3%) dans le cas où les temps d'attente de la source sont déterministes (resp. les durées de migration sont Weibull) (voir la première colonne dans la Table 5.2 ligne 1 (resp. ligne 3)). Dans la simulation du serveur, nous observons approximativement la même erreur relative quand les temps d'attente de la source sont déterministes (resp. les durées de migration sont Weibull), l'erreur la plus grande étant 17.2% (resp. 16%).

	100Mb/s LAN commuté	7Mb/s MAN
Répéteurs		
durée de migration	Weibull forme 4, échelle 100	Weibull forme 2.5, échelle 392
latence	Weibull forme 11, échelle 23	Weibull forme 6, échelle 88
Serveur		
durée de migration	Weibull forme 2.5, échelle 75.5	Weibull forme 3, échelle 1010
latence source-agent	Weibull forme 1.8, échelle 9.7	Weibull forme 10, échelle 28.6
latence source-serveur	Weibull forme 1.8, échelle 14.7	Weibull forme 1.8, échelle 93
temps de service	≈ constant	≈ constant

TAB. 5.1 – *Distribution des paramètres des modèles.*

	Moyenne	25%	50%	75%	90%	95%
Répéteurs valeurs : $\lambda = 1, \nu = 10, \delta = 11, \gamma = 45$						
attente déterministe source $\lambda \in [1,10]$	7.7	6.5	8.8	9.6	10.3	10.5
attente déterministe agent $\nu \in [1,10]$	1.6	0.3	1.4	2.4	3.5	4.2
durée migration Weibull $\delta \in [8,25]$ forme 4	8.3	4.3	8.2	10.4	14.7	16.3
durée comm. Weibull $\gamma \in [33.8,69.8]$ forme 11	2.7	1.0	2.2	3.9	6.4	7.1
toutes ensembles $\lambda,\nu \in \{1,3,5,7,9\}$	14.1	4.9	10.0	20.6	32.4	38.3
Serveur valeurs : $\lambda = 1, \nu = 10, \delta = 15, \gamma_1 = 115, \gamma_2 = 75, \mu = 2325$						
attente déterministe source $\lambda \in [1,10]$	14.9	15.1	16.3	17.1	17.1	17.2
attente déterministe agent $\nu \in [1,10]$	2.4	2.2	2.2	2.6	4.1	4.8
durée migration Weibull $\delta \in [12,19]$ forme 2.5	12.3	11.7	12.2	13.5	14.8	15.5
durée comm. Weibull $\gamma_1 \in [90,131]$ forme 1.8	1.3	0.6	1.5	1.8	2.1	2.3
durée comm. Weibull $\gamma_2 \in [56,94]$ forme 1.8	0.9	0.3	0.6	1.3	2.2	2.7
temps service déterministe $\mu \in [500,2500]$	1.5	0.6	1.5	2.2	2.9	3.5
toutes ensembles$\lambda,\nu \in \{1,3,5,7,9\}$	14.1	6.1	10.5	17.0	30.0	40.1

TAB. 5.2 – *Moyennes et pourcentiles des erreurs relatives données par les modèles.*

Quand toutes les hypothèses concernant la distribution des variables aléatoires ne sont pas vérifiées, les performances de nos modèles est satisfaisante. Dans la moitié des simulations l'erreur relative sur le temps de réponse est inférieure à 10.5% et sa moyenne est égale à 14.1% pour les deux modèles (voir lignes 5 et 12 dans la table 5.2).

Pour résumer nous pouvons dire que la sensibilité de nos modèles se situe essentiellement au niveau de l'attente de la source et des durées de migration. Quand leurs distributions ne sont pas exponentielles, l'erreur relative sur le temps de réponse n'est pas négligeable.

5.1.4 Vitesse de convergence

Nous avons effectué jusqu'à présent nos simulations sur une durée arbitrairement choisie de 10 000 secondes. Nous pouvons cependant légitimement nous demander quelle est l'influence de ce choix sur les résultats obtenus dans nos simulations et nos modèles. Pour vérifier cela, nous avons effectué des simulations avec des paramètres fixes mais pendant des durées variables. Pour chacune des simulations nous avons évalué les paramètres que nous avons ensuite utilisé dans nos modèles. Le temps de simulation influence directement l'évaluation des paramètres car il conditionne le nombre de valeurs dont nous disposons pour calculer nos moyennes. Ainsi, si pendant une session l'agent n'effectue qu'une migration, nous n'aurons qu'une unique valeur pour évaluer δ.

5.1.4.1 Cas idéal

Nous nous plaçons ici dans le cas idéal pour nos modèles c'est à dire quand les lois de distributions des paramètres sont exponentielles. Les résultats de nos simulations sont donnés dans la figure 5.5 (resp. 5.4) pour le serveur (resp. les répéteurs). L'échelle utilisée pour représenter le temps est logarithmique; nous avons ajouté une courbe représentant le temps obtenu grâce aux modèles quand les paramètres entrés correspondent exactement à ceux utilisés pour les simulations afin de représenter le résultat idéal à l'infini.

Il est intéressant de noter que dans les deux cas (répéteur et serveur) la simulation présente une variance beaucoup plus importante que les modèles mais que tous convergent vers la valeur optimale assez rapidement, ce qui est confirmé par l'examen de la table 5.3. L'erreur relative après 10 secondes de simulation est au maximum de 26.5% pour le temps de réponse du serveur ($\lambda = 10$, $\nu = 10$ ligne 2 de la table 5.3), 28% ($\lambda = 1$, $\nu = 1$, ligne 5) pour celui des répéteurs et 51.5% ($\lambda = 10$, $\nu = 10$, ligne 10) pour le nombre moyen de répéteurs. Si la durée simulée est de 1000 secondes, les erreurs maximales tombent à respectivement 1.82% (ligne 4), 1.76% (ligne 6) et 1.68% (ligne 9).

	Durée simulée en secondes						
	10	50	100	500	1000	5000	10000
Temps de réponse serveur							
$\lambda = 1$, $\nu = 1$	16.99	10.43	15.99	1.50	1.41	1.56	1.27
$\lambda = 1$, $\nu = 10$	26.50	14.18	4.02	3.20	1.61	0.11	1.17
$\lambda = 10$, $\nu = 1$	17.51	8.92	5.93	0.81	1.75	1.34	0.45
$\lambda = 10$, $\nu = 10$	20.96	1.63	0.13	0.96	1.82	0.34	0.72
Temps de réponse répéteurs							
$\lambda = 1$, $\nu = 1$	27.65	11.97	18.37	2.55	0.76	1.76	0.79
$\lambda = 1$, $\nu = 10$	18.75	0.22	8.38	0.51	7.97	0.07	1.27
$\lambda = 10$, $\nu = 1$	15.46	1.40	1.90	1.60	0.06	0.10	0.31
$\lambda = 10$, $\nu = 10$	15.65	4.99	5.74	0.44	1.13	0.72	0.78
Nombre moyen de répéteurs							
$\lambda = 1$, $\nu = 1$	3.40	21.46	13.83	3.03	9.02	1.28	0.12
$\lambda = 1$, $\nu = 10$	51.43	15.78	1.21	1.83	1.73	1.68	0.58
$\lambda = 10$, $\nu = 1$	0.02	3.50	3.10	1.81	0.89	0.46	0.15
$\lambda = 10$, $\nu = 10$	21.50	2.87	4.53	0.79	0.74	0.29	0.57

TAB. 5.3 – *Erreur relative (en %) sur le temps de réponse et le nombre de répéteurs en fonction de la durée de simulation*

Nous pouvons visualiser sur les figures 5.5 (resp. 5.4) la variance des temps théorique et simulé pour le serveur (resp. temps théorique, simulé et nombre moyen de répéteurs). Dans le cas où la simulation dure plus de 1000 secondes, nous constatons que la variance est très faible pour les valeurs mesurées grâce aux simulations. Les modèles se comportent

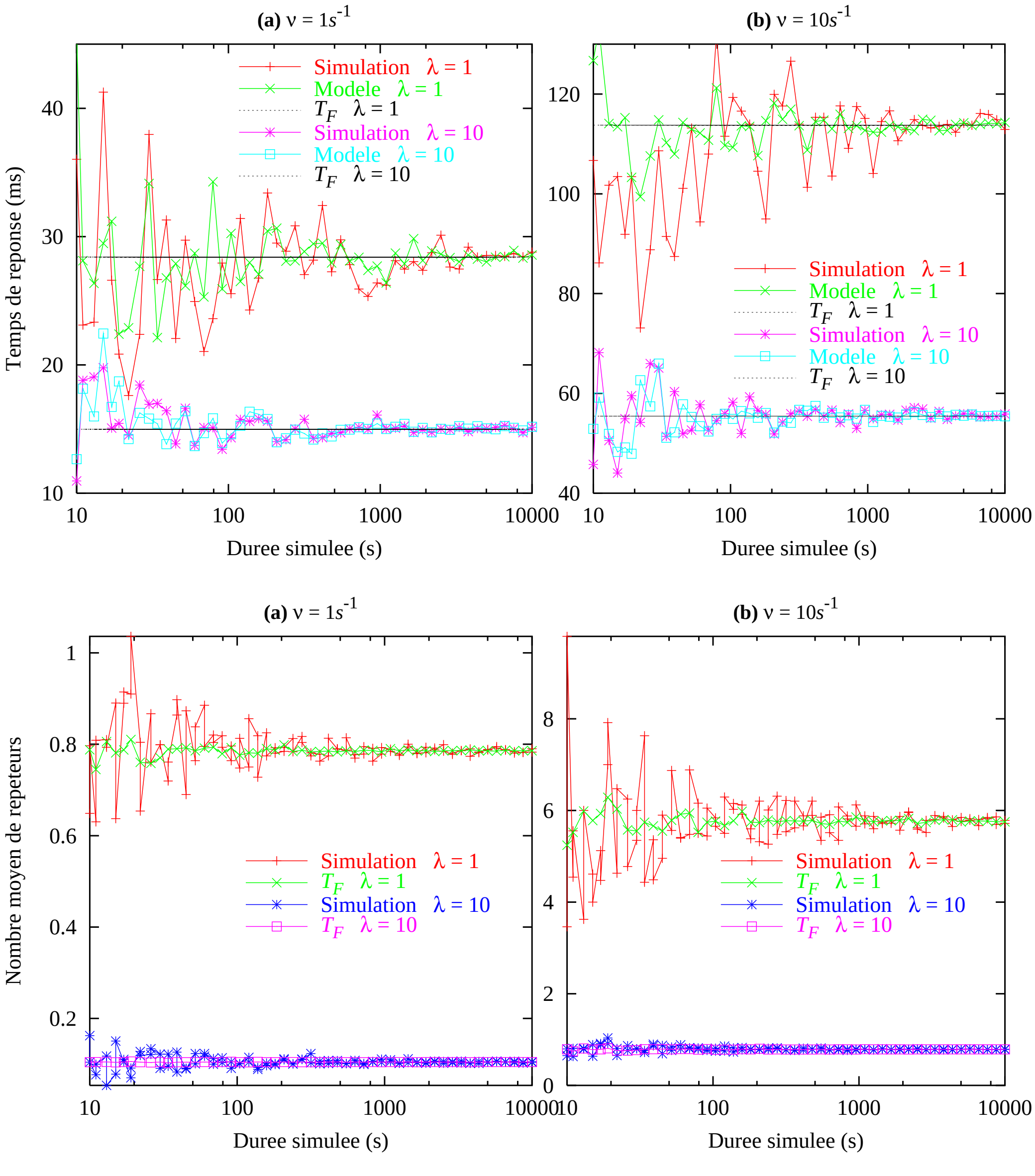


FIG. 5.4 – *Convergence du simulateur et du modèle des répéteurs dans le cadre d'un LAN ($\delta_F = 15.16$, $\gamma = 45.6$)*

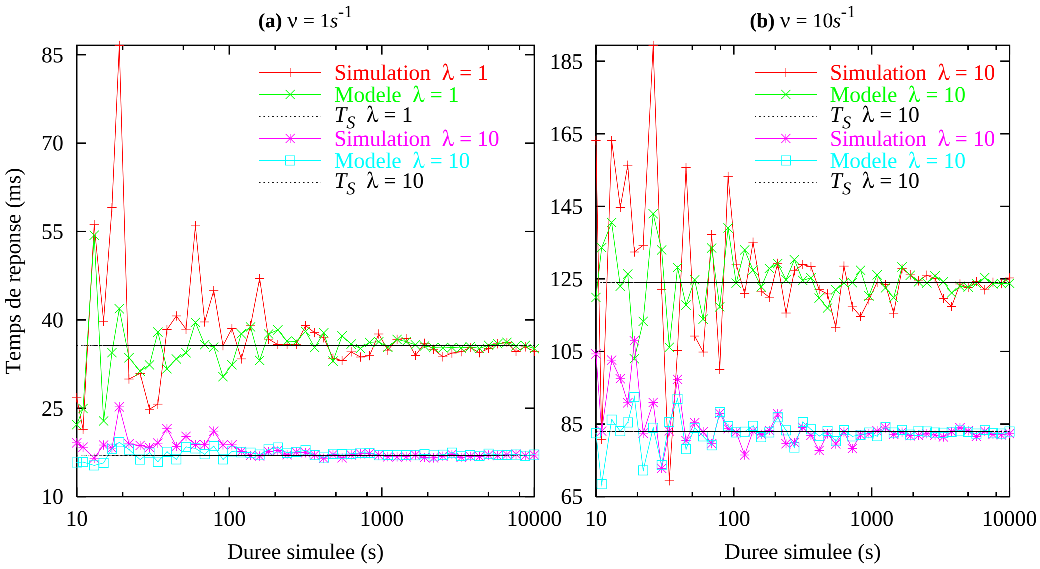


FIG. 5.5 – *Convergence du simulateur et du modèle du serveur dans le cadre d'un LAN ($\delta_S = 14.5$, $\gamma_1 = 115.6$, $\gamma_2 = 76.3$, $\mu = 2325$)*

quant à eux encore mieux car ils présentent une variance inférieure aux simulations.

5.1.4.2 Cas réaliste

Nous étudions maintenant la vitesse de convergence dans le cas où les distributions de nos variables ne sont plus exponentielles mais correspondent à celles obtenues durant nos expérimentations sur un réseau local. Les résultats sont présentés dans les figures 5.7 pour le serveur et 5.6 pour les répéteurs. L'erreur relative est pour sa part reportée dans la table 5.4.

Nous obtenons à peu près les mêmes résultats que lors de nos simulations précédentes, i.e. après 1000 secondes la variance devient faible et le résultat obtenu est très proche du résultat final. Nous voyons qu'il existe un décalage entre le temps mesuré dans le simulateur et celui donné par les modèles et que ces deux temps ne semblent pas converger. Quand $\lambda = 1$, $\nu = 10$ l'erreur relative pour le serveur (figure 5.7 (b)) se stabilise autour de 11% et celle pour les répéteurs (figure 5.6 (b)) autour de 13%.

Notons que les erreurs relatives finales sont légèrement inférieures à celles obtenues précédemment lors du test de la robustesse des modèles (table 5.2) car nous avons maintenu les lois exponentielles pour λ et ν. Cela ne change cependant pas la conclusion concernant la vitesse de convergence.

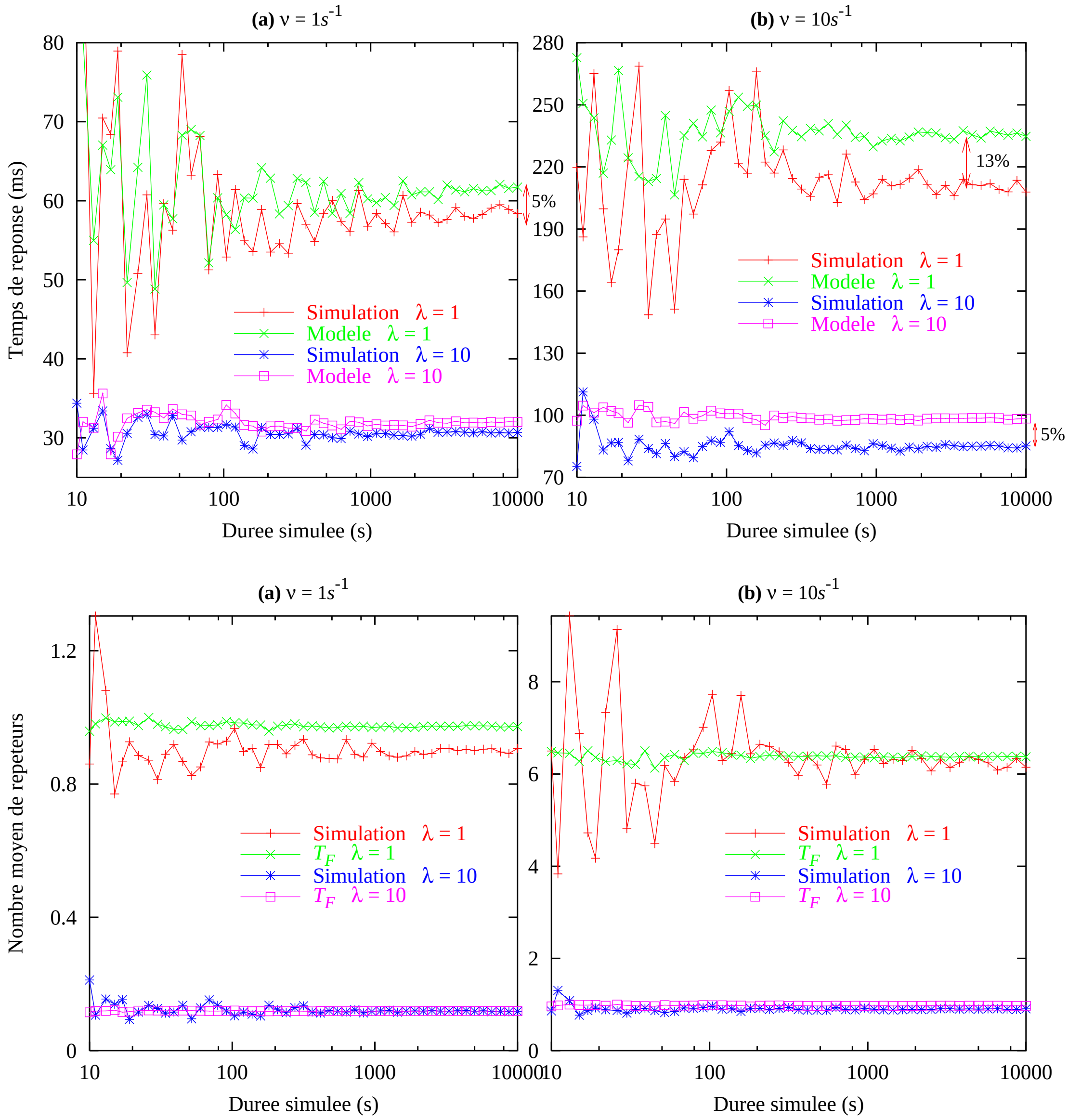


FIG. 5.6 – *Convergence du simulateur et du modèle des répéteurs dans le cadre d'un LAN ($\delta_F = 15.16$, $\gamma = 45.6$) avec des distributions réalistes*

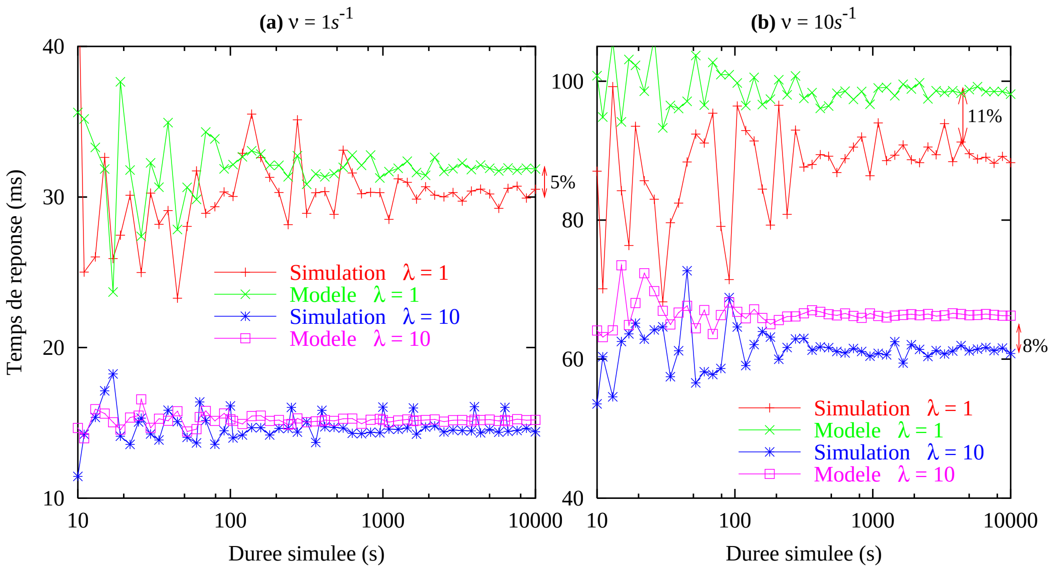


FIG. 5.7 – *Convergence du simulateur et du modèle du serveur dans le cadre d'un LAN ($\delta_S = 14.5$, $\gamma_1 = 115.6$, $\gamma_2 = 76.3$, $\mu = 2325$) avec des distributions réalistes*

	Durée simulée en secondes						
	10	50	100	500	1000	5000	10000
Temps de réponse serveur							
$\lambda = 1, \nu = 1$	24.95	9.24	7.04	9.33	11.12	5.67	4.41
$\lambda = 1, \nu = 10$	15.81	12.24	3.44	8.12	5.34	10.30	11.17
$\lambda = 10, \nu = 1$	28.09	2.80	8.07	2.89	3.20	4.49	5.46
$\lambda = 10, \nu = 10$	19.74	13.91	3.39	7.81	8.86	8.40	8.99
Temps de réponse répéteurs							
$\lambda = 1, \nu = 1$	7.05	13.03	10.14	6.94	2.37	6.49	5.75
$\lambda = 1, \nu = 10$	24.16	9.90	3.92	11.44	8.67	10.87	12.92
$\lambda = 10, \nu = 1$	18.80	10.94	7.90	4.85	3.69	4.29	4.34
$\lambda = 10, \nu = 10$	29.25	23.24	9.44	17.31	14.81	15.90	15.45
Nombre moyen de répéteurs							
$\lambda = 1, \nu = 1$	15.61	20.59	4.64	1.69	3.08	3.50	1.84
$\lambda = 1, \nu = 10$	0.21	2.83	16.07	3.26	2.80	0.87	3.70
$\lambda = 10, \nu = 1$	45.76	25.83	15.57	1.04	0.48	0.28	0.88
$\lambda = 10, \nu = 10$	11.43	19.55	1.73	10.42	8.14	8.16	7.22

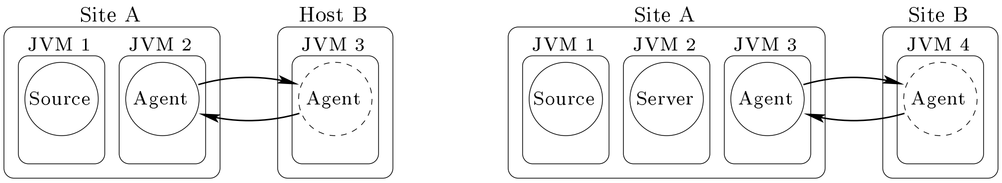
TAB. 5.4 – *Erreur relative (en %) sur le temps de réponse et le nombre de répéteurs en fonction de la durée de simulation pour des distributions réalistes*

5.2 Validation grâce aux expérimentations

Pour valider nos modèles, nous avons conduit des expérimentations intensives sur un réseau local (LAN) et un réseau régional (MAN). Notre LAN était composé de PCs Pentium-2 et Pentium-3 sous Linux 2.2.18 et reliés entre eux par du 100Mb/s commuté. Le MAN avait en plus des Sun Sparc sous Solaris reliés par du 7Mb/s. Dans toutes nos expérimentations nous avons utilisé Java 1.2 [30].

5.2.1 Description

Pour le test des deux modèles, nous avons utilisé une application composée d'un agent mobile et d'un objet fixe (source) qui périodiquement essaie de communiquer avec lui. Plus précisément, le temps moyen entre deux communications de la source vers l'agent suit une loi exponentielle de paramètre λ (i.e. la source attend en moyenne $1/\lambda$ secondes avant de tenter de joindre l'agent). De son côté, l'agent a un taux de migration qui lui aussi suit une loi exponentielle mais de paramètre ν. Ces deux paramètres pouvaient être changés entre deux expérimentations. Tous les autres paramètres introduits lors de la présentation de nos modèles étaient dépendants de l'infrastructure utilisée dans nos expériences. Ainsi, des hypothèses formulées dans notre modélisation, seulement deux sont toujours valides : les temps moyens d'inactivité de la source et de l'agent suivent des lois exponentielles (see Section 4.1)).



(a) Mécanisme des répéteurs **(b) Serveur de localisation**

FIG. 5.8 – *Infrastructure d'expérimentations*

Les applications utilisées dans nos expérimentations ont été écrites en utilisant *ProActive*[52] et les primitives présentées précédemment (voir section 3.1).

5.2.2 Points de mesures

Nous avons introduit pour les besoins de nos modèles de nombreux paramètres que nous allons devoir mesurer ou évaluer lors de nos expérimentations. Dans le cas des mesures, nous avons directement introduit dans *ProActive* des sondes pour nous permettre de collecter des informations pendant l'exécution de l'application. Cela a été fait en modifiant les méta-objets associés à un objet actif. Ainsi le *MigrationManager* a été remplacé par un *TimedMigrationManager* qui en plus de la migration mesure le temps pris pour les opérations. Nous voyons immédiatement du découpage présenté dans la figure 3.4 les objets à modifier pour introduire les points de mesure. Il s'agit du *Proxy*, *RequestSender*, du *Service* et du *MigrationManager* pour respectivement mesurer la durée totale de communication, les temps de communication inter-sites, la durée de traitement des messages au serveur, et la durée de migration.

5.2.3 Évaluation des paramètres

Pour évaluer un paramètre nous avons fait une analyse post-mortem à partir des informations générées par nos sondes de mesure. Pour la plupart d'entre eux (durée de migration par exemple), obtenir sa valeur consistait simplement à faire une moyenne sur l'ensemble des mesures. Le point le plus critique a consisté à obtenir le délai inter-sites. En effet, celui-ci ne consiste pas seulement à mesurer la durée passée dans le *RequestSender*; une communication réseau peut être ralentie parce que la source trouve l'agent en train de migrer et doit donc attendre la fin de sa migration. Ce temps sera donc inclus dans le délai inter-sites mesuré et il faudra le retrancher pour obtenir la véritable valeur.

Proposition 5.2.1 (Délai inter-sites pour le mécanisme du serveur) *Le délai inter-sites quand le mécanisme de localisation du serveur est utilisé est égal au temps effectivement mesuré lors de la communication réseau, moins le temps passé à attendre la fin de l'agent.*

Dans le cas où les répéteurs sont utilisés la situation se complique. En effet, en plus de l'attente déjà notée précédemment, nous devons tenir compte du fait qu'une communication inter-sites peut en fait en cacher plusieurs. Si le message passe à travers plusieurs répéteurs alors il y aura en fait plusieurs sauts. Nous avons donc introduit un point de mesure qui nous indique combien de répéteurs un message a traversé, ce qui nous permet d'introduire la proposition suivante :

Proposition 5.2.2 (Délai inter-sites pour le mécanisme des répéteurs) *Soit t le temps mesuré pour une communication avec un agent par une source, n le nombre de répéteurs traversés par ce message et t_b le temps perdu à attendre la fin des migrations de l'agent. Le temps réel t_r d'une communication réseau (délai inter-sites) est donné par*

$$t_r = \begin{cases} t - t_b & \textit{pour } n = 0, \\ \\ \dfrac{t - t_b}{n+2} & \textit{pour } n > 0, \end{cases} \tag{5.1}$$

Preuve : Dans le cas où $n = 0$ ce résultat est le même que dans le cas du serveur. Pour obtenir la version pour $n > 0$, il faut remarquer qu'un message qui traverse un répéteur effectue en fait deux sauts (source-répéteur, répéteur-agent). Donc quand il y a n répéteurs entre une source et un agent, le message devra effectuer $n+1$ sauts pour atteindre l'agent. Il faut aussi noter qu'un message qui arrive à un agent après avoir traversé un répéteur donne lieu au raccourcissement de la chaîne ce qui fait donc une communication supplémentaire et nous donne donc $n+2$ sauts. ■

Nous voyons ici pourquoi le choix d'une analyse post-mortem a été fait : le délai inter-sites nécessite de connaître des informations distribuées qu'il est difficile d'avoir à l'exécution.

5.2.4 Résultats

Nous avons tracé dans les figures 5.9 et 5.10 les résultats mesurés lors de nos expérimentations ainsi que les valeurs obtenues grâce à nos modèles. Les six premiers graphiques concernent le réseau local, et les six suivants un réseau régional. Les courbes sur la partie gauche indiquent le temps de réponse et le nombre moyen de répéteurs en fonction du taux de communication λ pour des valeurs allant de $1s^{-1}$ à $10s^{-1}$ et ce pour trois valeurs différentes du taux de migration ν ($\nu = 1{,}5{,}10$). De la même manière, la partie droite contient les résultats en fonction du taux de migration ν variant de $1s^{-1}$ à $10s^{-1}$ pour trois valeurs du taux de communication λ ($\lambda = 1{,}5{,}10$). Pour chaque couple (λ,ν), les valeurs empiriques de δ et γ (resp. δ,γ_1,γ_2 et μ) ont été injectées dans les formules de T_F et N (resp. T_S) données en (4.29) (resp. (4.66)) pour le mécanisme des répéteurs (resp. mécanisme du serveur).

Nous pouvons remarquer que les résultats pratiques et théoriques sont relativement proches sur pratiquement toutes les expérimentations. Pour chacune des configurations réseau (LAN ou MAN), la table 5.5 donne les performances des modèles avec la moyenne (1ère colonne) et les pourcentiles, la deuxième colonne donnant le 25ème pourcentile de l'erreur relative, la troisième la valeur médiane...

		Moyenne	25%	50%	75%	90%	95%
Répéteurs	100Mb/s LAN	7.3	2.1	5.7	10.8	17.0	20.3
	7Mb/s MAN	12.1	2.3	7.8	19.0	25.9	35.0
	global	9.7	2.2	7.1	15.4	22.1	26.3
Serveur	100Mb/s LAN	4.6	2.3	4.3	6.2	8.5	10.4
	7Mb/s MAN	13.9	6.2	11.3	21.5	28.3	33.0
	global	9.6	3.7	6.5	13.4	23.4	28.3

TAB. 5.5 – *Moyenne et pourcentiles de l'erreur relative pour les deux mécanismes*

Les résultats de la table 5.5 indiquent que les modèles théoriques ont des comportement acceptables. Leurs performances globales sont très proches pour l'un et pour l'autre (lignes 3 et 6 de la table) : la première colonne indique que l'erreur relative moyenne est égale à 9.7% (resp. 9.6%) pour le modèle des répéteurs (resp. serveur). Dans 90% des expérimentations avec les répéteurs (resp. serveur) l'erreur relative est inférieure à 22.1%

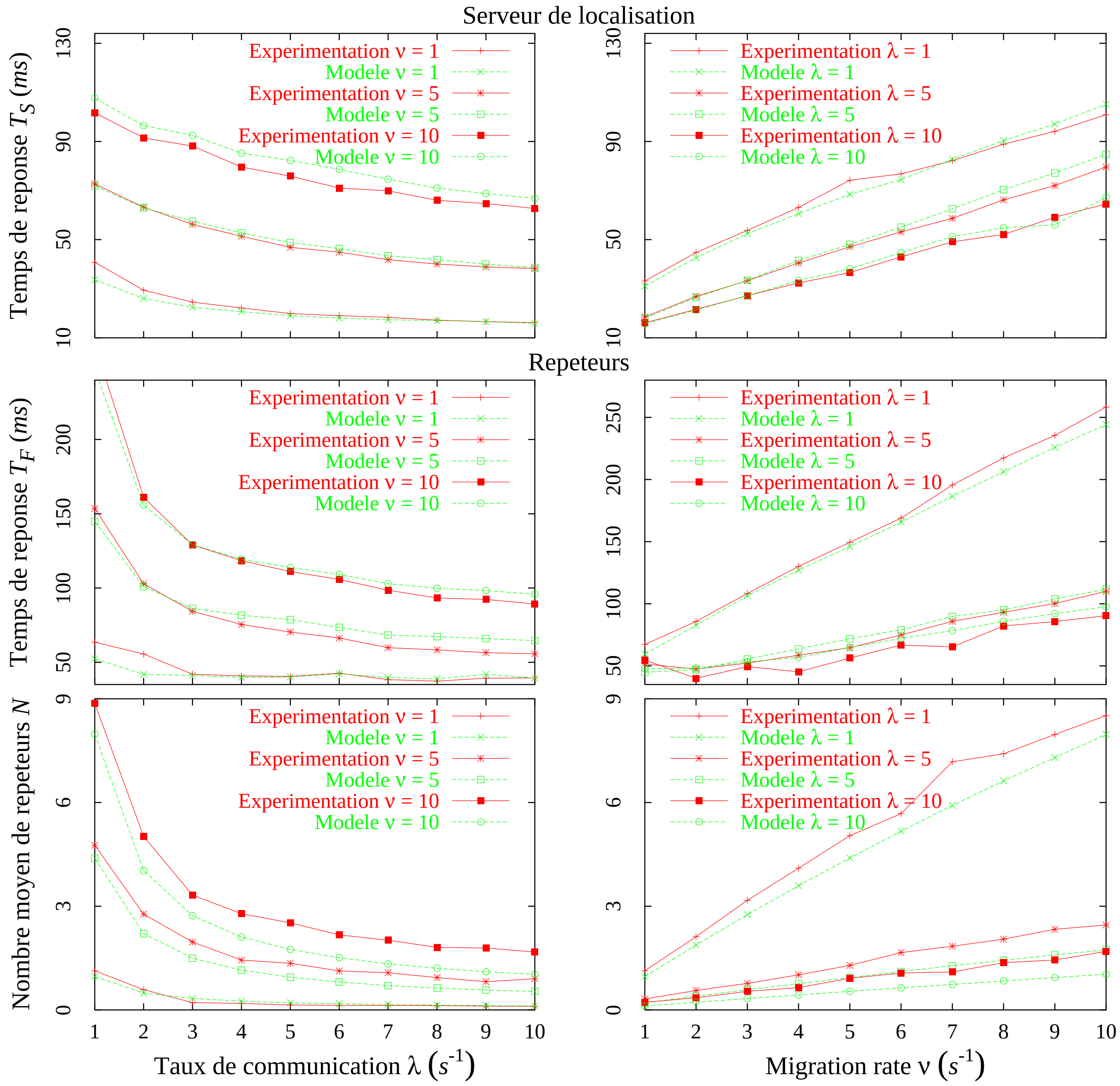


FIG. 5.9 – *Validation et expérimentations sur un LAN*

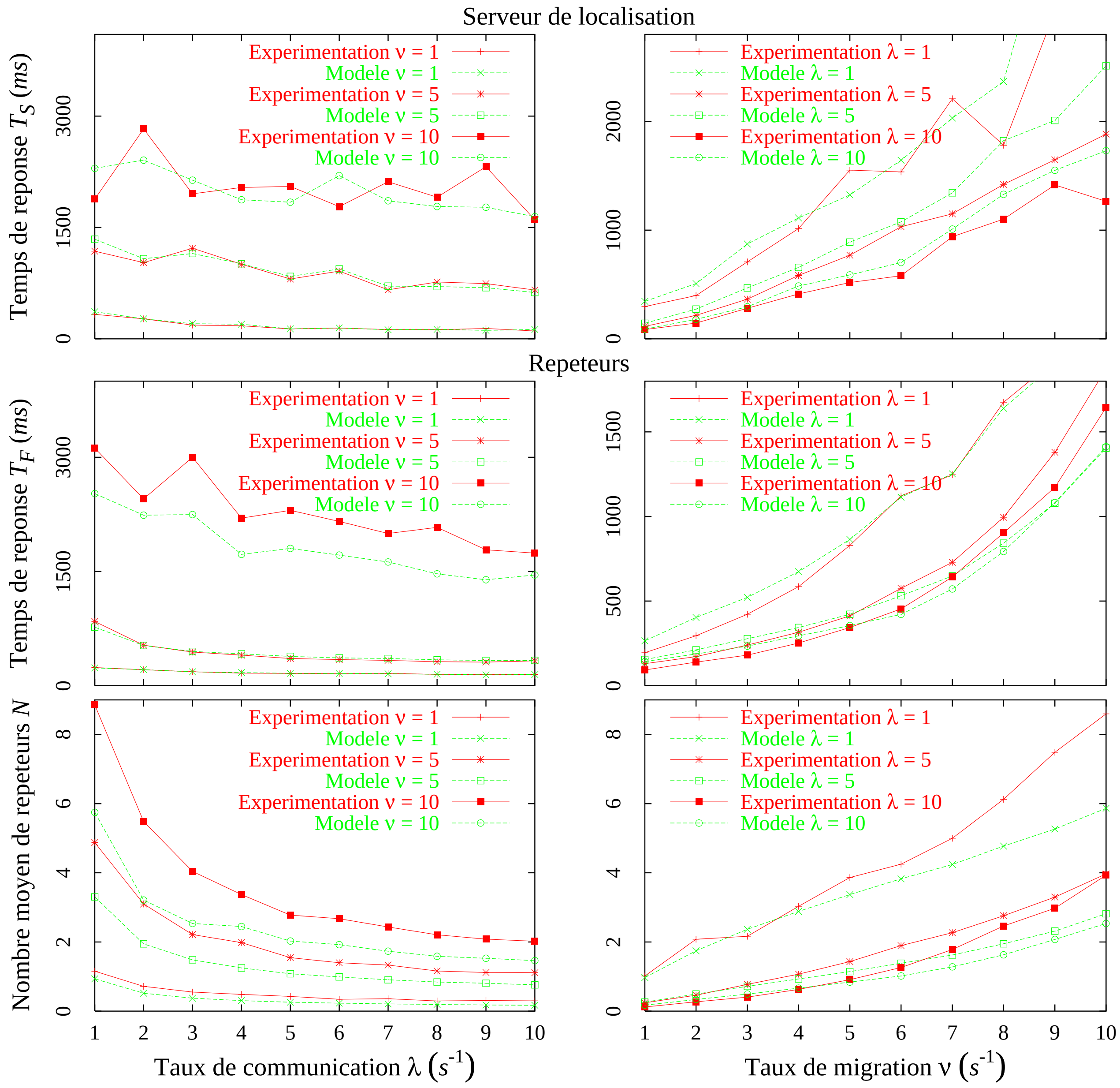


FIG. 5.10 – *Validation et expérimentations sur un MAN*

(resp. 23.4%) comme indiqué dans la 5^{eme} colonne. Cependant, nous voyons que les deux modèles se comportent mieux quand ils sont appliqués sur un LAN (lignes 1 et 4) plutôt que sur un MAN (lignes 2 et 5).

En plus d'une vérification du comportement de nos modèles en conditions réelles, nous pouvons utiliser les résultats des expérimentations pour comparer les performances de notre implémentation des deux mécanismes. Il est facile de voir que sur un LAN, le serveur de localisation offre de meilleures performances que le mécanisme des répéteurs. Mais nous observons le résultat opposé quand nous sommes sur un MAN. Dans ce cas, les répéteurs offrent de bien meilleurs temps de réponse. Il semble donc que le serveur de localisation soit plus adapté à des réseaux à haut débit. Quand la durée de communication augmente (et par là même la durée de migration), les répéteurs surpassent le serveur en termes de performances.

5.2.5 Remarque sur la condition de stabilité dans le cas des répéteurs

La condition de stabilité du modèle des répéteurs qui est $1/\gamma < 1/\nu + 1/\delta$ indique la limite à partir de laquelle il n'est plus possible de joindre un agent parce que le réseau est trop lent ou parce que celui-ci se déplace trop vite. L'important est de savoir si dans la réalité, cette condition est toujours vérifiée ou si au contraire il est possible qu'un agent se déplace tellement rapidement qu'il n'est plus possible de le joindre [47]. Il faut tout d'abord noter qu'en pratique, nous avons $taille(agent) > taille(message)$ et que donc le temps de communication pour envoyer un agent sur un site distant sera plus grand que celui nécessaire pour envoyer un message. Nous aurons donc $1/\gamma < 1/\delta$ et ainsi la condition de stabilité sera toujours vérifiée, i.e. il sera toujours possible de joindre un agent. Nous pouvons donc récrire la condition de stabilité de la manière suivante :

Proposition 5.2.3 *Soit un système à agents mobiles où un agent et un message empruntent les même chemins. Si la taille de l'agent est supérieure à celle du message, alors quelque soit le taux de migration de l'agent, tout message envoyé finira par l'atteindre.*

5.3 Comparaison formelle des performances de localisation

Comme déjà mentionné, certains des paramètres de nos expérimentation sont dépendants des conditions réseau ou matérielles et nécessairement, une comparaison expérimentale sera limitée à quelques scénarios. Il est possible de s'affranchir de cette limitation en comparant directement les résultats théoriques obtenus dans les sections 3.5.1 et 3.5.2. Nous allons donc nous intéresser aux temps de réponse donnés par chacun des deux mécanismes, et précisement à la différence ΔT donnée par

$$\Delta T = T_F - T_S \tag{5.2}$$

où T_F représente le temps de réponse moyen obtenu en utilisant le mécanisme des répéteurs (équation (4.29)) et T_S celui en utilisant un serveur de localisation (équation (4.66)) respectivement. Nous avons étudié quatre cas correspondant à différentes valeurs des paramètres des modèles. Sauf mention contraire, les valeurs utilisées pour ces paramètres sont celles indiquées dans la table 5.6 où chacune des entrées indique la valeur moyenne mesurée dans l'ensemble de nos expérimentations. Dans chacun des cas, nous avons identifié la région où $\Delta T = 0$, ce qui correspond à la situation où les deux mécanismes offrent les mêmes performances. Quand $\Delta T > 0$ (resp. $\Delta T < 0$) le serveur de localisation (resp. les répéteurs) donne de meilleures performances. La figure 5.11 donne le signe de ΔT dans les quatre cas que nous avons étudié.

	$\delta_F(s^{-1})$	$\delta_S(s^{-1})$	$\gamma(s^{-1})$	$\gamma_1(s^{-1})$	$\gamma_2(s^{-1})$	$\mu(s^{-1})$
LAN	15.16	15.12	45.6	115.6	76.3	2325
MAN	2.10	1.3	12.3	36.7	12.1	938

TAB. 5.6 – *Valeurs utilisées pour les comparaisons théoriques*

La figure 5.11(a) indique que dans des conditions de réseau local, la technique à base de serveur se comporte pratiquement toujours mieux, mais que dans le cas d'un MAN ce sont les répéteurs qui prennent le dessus. Il s'agit là d'une simple vérification de ce que nous avions déja observé dans la section 5.2. Cependant, ce que nous ne pouvions pas déduire des expérimentations seules est que pour certains taux de migration (e.g. $\nu = 35$) les répéteurs sont meilleurs seulement pour certaines valeurs du taux de communication ($\lambda \in [9,19]$ par exemple).

Dans la figure 5.11(b), les latences réseau correspondent à un LAN. Nous observons que pour des taux de service extrèmement faibles (μ inférieur à $16s^{-1}$), les répéteurs deviennent meilleurs que le serveur. La frontière entre les régions où chacune des approches est meilleure, i.e. la ligne $\Delta T = 0$, dépend des valeurs de λ et ν. Étonnamment, augmenter le taux de communication tout en maintenant le taux de migration inchangé n'a pas le même effet sur la frontière : pour $\nu = 1$, faire varier λ de 1 à 5 déplace la ligne $\Delta T = 0$ vers la gauche (taux de service plus faible) alors que la même augmentation pour λ quand $\nu = 5$ déplace cette même frontière vers la droite (taux de service plus élevé). Un déplacement de la frontière vers la gauche augmente la région où le serveur se comporte mieux alors qu'un déplacement vers la droite indique une plus grande domination des répéteurs. Remarquons que les conditions de nos expérimentations avec une source et un agent étaient loin des cas que nous étudions actuellement. En effent, nous avions alors $\mu = 2325$ sur notre LAN alors que nous étudions dans notre comparaison un μ variant de 0 à 100. Cependant, dans

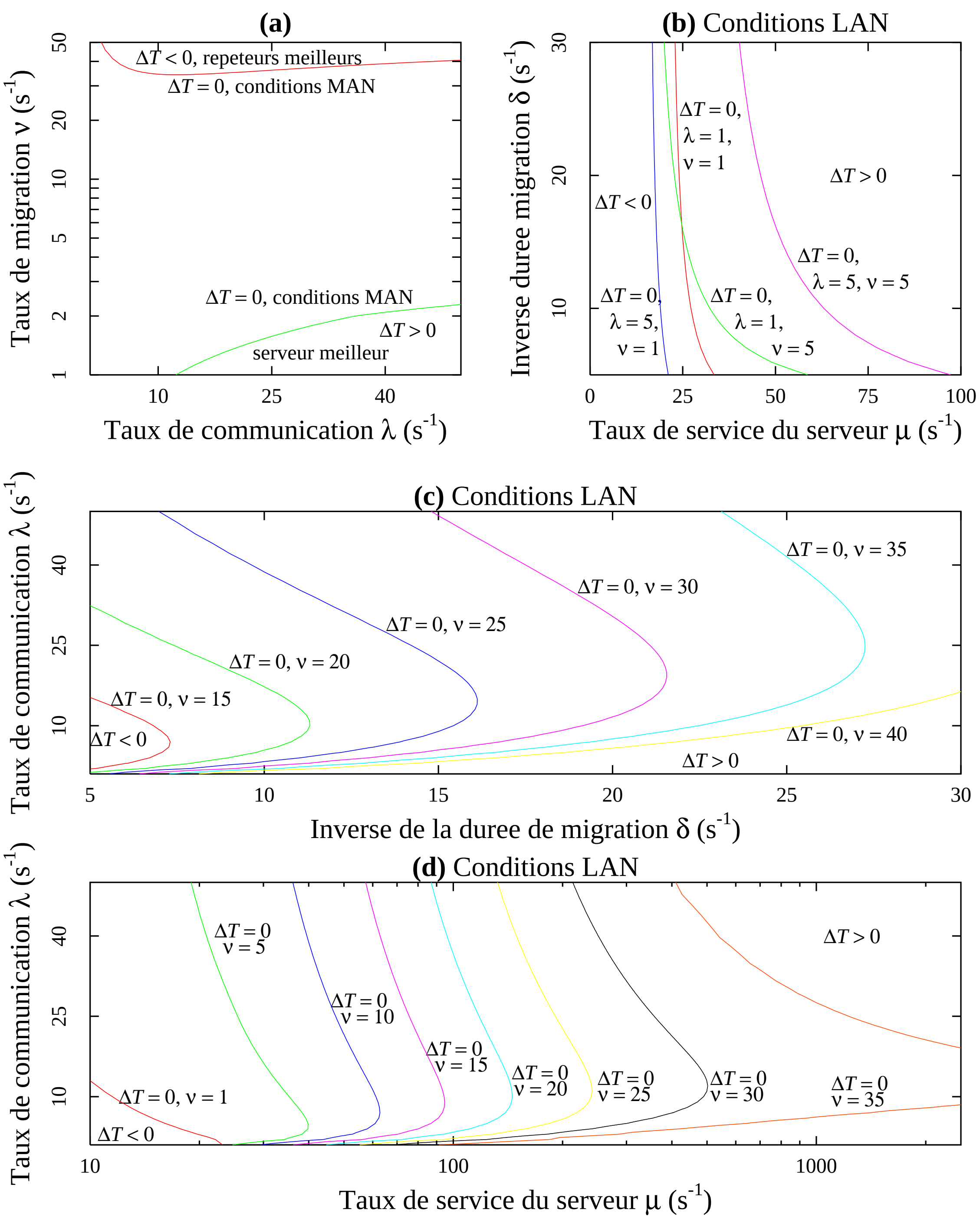


FIG. 5.11 – *Signe de la différence entre les temps de réponse $\Delta T = T_F - T_S$.*

le cas où nous aurions plusieurs sources et plusieurs agents, nous pouvions nous attendre à avoir un serveur qui semble travailler beaucoup plus lentement. L'influence du nombre de couples source-agent sur la vitesse du serveur varie suivant les paramètres λ et ν. Par exemple, dans nos expériences (voir section 5.4) nous avions besoin pour $\lambda = 1, \nu = 1$ de 100 couples pour que le taux de service soit de 30 alors qu'avec $\lambda = 1, \nu = 10$ il n'en fallait que 60. Ce n'est qu'à partir de ce taux que les répéteurs donneront de meilleures performances que le serveur. En regardant 5.11(c) nous voyons que pour chaque valeur de δ (pour rappel $1/\delta$ est le temps moyen de migration), il existe des valeurs de ν pour lesquelles les répéteurs ne sont meilleurs que pour quelques valeurs du taux de communication λ. Par exemple, si $\delta = 10$ et $\nu = 20$, le serveur surpasse les répéteurs pour $\lambda \in [1,5] \cap [18,50]$ alors que nous avons le résultat opposé quand $\lambda \in [6,17]$. Ce comportement peut être observé pour n'importe quelle valeur de λ et ν dans un LAN.

Finalement, la figure 5.11(d) indique la frontière $\Delta T = 0$ pour différents taux de migration ($\nu \in \{1,5,10,15,20,25,30,35\}$) et pour $\mu \in [10,2500]$ et $\lambda \in [1,50]$; la durée de migration et la latence correspondent encore une fois à celles observées sur un LAN. Quand ν augmente, la frontière se déplace vers la droite (taux de service plus élevé) élargissant la région où les répéteurs sont meilleurs. En fait, quand le taux de migration augmente, les performances des deux approches se dégradent : dans le cas des répéteurs, la chaîne de répéteurs entre l'agent et la source s'allonge, amenant par là même un temps de communication plus élevé; dans le cas du serveur, cette augmentation du taux de migration a trois effets. Tout d'abord, la probabilité que la position utilisée par la source ne soit plus valide augmente. Ensuite, la probabilité d'avoir une mauvaise localisation donnée par le serveur augmente aussi. Finalement, il y aura plus de `update location requests` générées. Comme ces requêtes sont prioritaires, elles ralentiront le traitement d'une requête de la source. Tout cela mis ensemble fait que quand ν augmente, les performances du serveur se dégradent plus rapidement que celles des répéteurs. Remarquons que nous observons dans ce dernier graphique les même résultats que ceux des figures 5.11(a) et 5.11(b) : pour certaines valeurs du taux de migration ν les répéteurs se comportent mieux seulement pour quelques valeurs intermédiaires du taux de communication λ. Par exemple si $\mu = 2325$ et $\nu = 35$, les répéteurs sont meilleurs pour $\lambda \in [9,19]$. Ce résultat n'etant pas du tout intuitif, nous avons entrepris des simulations pour essayer de le comprendre. La figure 5.12 montre l'évolution du temps de réponse en fonction du taux de communication d'une source. Nous avons, dans le cas du serveur, fixé son taux de service à 1000 requêtes par seconde, ce qui nous place dans la situation précédente où le signe de ΔT changeait plusieurs fois. Cela apparait sur notre graphique où la courbe du temps de réponse des répéteurs est au dessus de celle du serveur au début, puis passe au dessous pour $\lambda = 6.2$ avant de repasser au dessus pour $\lambda = 27.5$. Cependant, la différence n'est pas très significative, il s'agit plutôt d'une zone où les deux mécanismes ont des performances équivalentes.

Pour conclure cette section nous aimerions insister sur le fait que choisir le meilleur mécanisme de communication n'est pas aussi trivial et intuitif qu'il n'y parait au premier

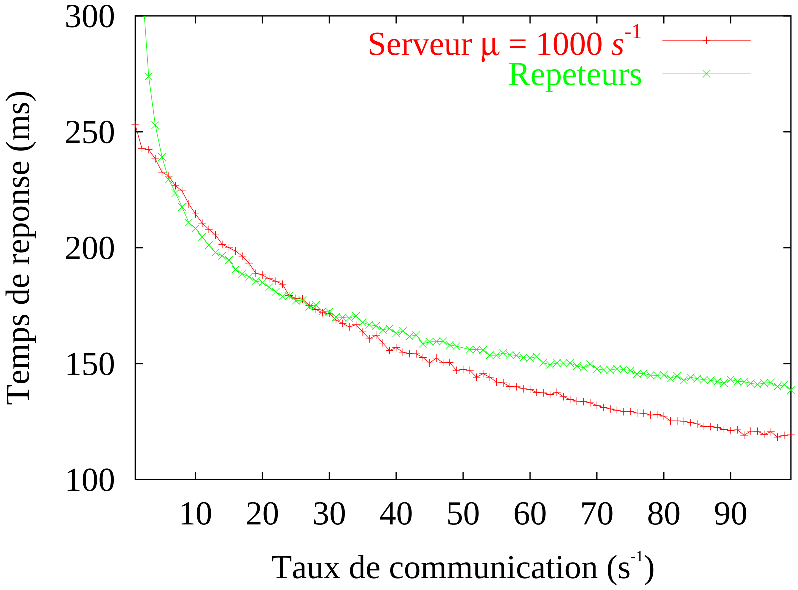


FIG. 5.12 – *Temps de réponse des répéteurs et d'un serveur à taux 1000 s^{-1} en fonction du taux de communication de la source*

abord. Notre étude du signe de ΔT a mis en relief des effets inattendus suivant la valeur de certains paramètres. Bien que nous ayons trouvé des explications pour ces comportements, il était difficile sinon impossible de les prévoir par une simple étude des algorithmes. Grâce aux modèles obtenus et à leur comparaison formelle, nous sommes maintenant en mesure de dégager les domaines où chacun des deux mécanismes donnera de meilleures performances.

5.4 Extension aux cas multi-sources multi-agents

5.4.1 Définitions

Nous avons vu que le mécanisme de localisation qui présente les meilleures performances sur un MAN est celui à base de répéteurs. Sachant qu'un serveur avec plusieurs queues est plus lent qu'un serveur avec une unique queue, il est évident que quelque soient les résultats, le meilleur mécanisme sur un MAN restera les répéteurs. Nous allons donc dans cette section donner les résultats de l'extension de notre modèle dans des conditions de réseau local. Par soucis de complétude, les résultats pour d'autres conditions sont donnés dans l'annexe C.2.1. Pour évaluer le travail fourni par le serveur nous introduisons la définition du taux d'utilisation :

Définition 1 *L'utilisation du serveur est le pourcentage moyen du temps pendant lequel* *il est effectivement utilisé.*

Une façon d'estimer l'utilisation du serveur dans nos expérimentations est de faire le ratio du temps pendant lequel il travaille sur le temps total de l'expérimentation.

5.4.2 Limites du modèle

Nous allons dans un premier temps essayer de définir précisément la zone de validité de notre modèle, i.e. les paramètres pour lesquels l'erreur relative entre celui-ci et notre simulateur reste faible. En plus des variables λ et ν introduites précédemment, nous ajoutons maintenant n, le nombre de couples considérés.

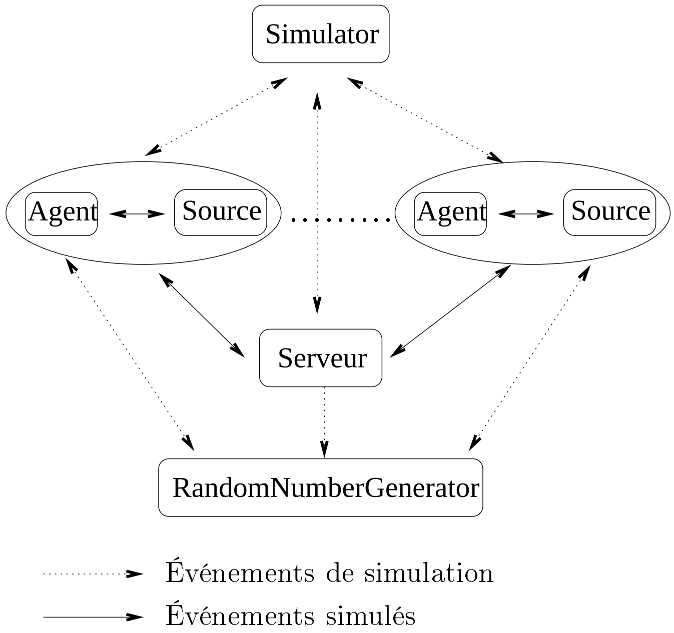


FIG. 5.13 – *Architecture du simulateur Multi source-agent*

Nous avons conduit une série de simulations avec plusieurs sources et plusieurs agents en utilisant une version modifiée de notre simulateur (figure 5.13. Dans chacune, les variables aléatoires utilisées suivaient des lois exponentielles de manière à être le plus près possible des hypothèses utilisées dans notre approximation du serveur de la section 4.4.2. Tous les couples source-agent avaient les même taux de communication et de migration $\lambda_i = \lambda$ et $\nu_i = \nu$ pour $i = 1,\ldots,n$ avec n nombre de couples considérés durant la simulation. Nous avons choisi de prendre les valeurs pour λ et ν dans l'ensemble 1,10 ce qui nous a donné quatre couples possibles pour lesquels nous avons effectué 100 simulations correspondant au nombre de paires source-agent possibles. Nous avons donc au total effectué 400 simulations. Pour chacune d'entre elles, n était fixé et la latence, les temps de service et la durée de migration étaient des variables aléatoires de taux γ_1,γ_2,μ et δ respectivement. Les valeurs retenues pour ces derniers paramètres reflétaient les conditions rencontrées sur un réseau local (voir la table 5.6).

Pour chaque simulation nous avons calculé le temps de réponse estimé T_S en utilisant la formule (4.66) et l'approximation de μ_i introduite dans la section 4.4.2 ; nous avons également mesuré l'utilisation du serveur multi-queues. Le temps de réponse moyen donné par la simulation et le temps théorique T_S sont tracés en fonction du nombre de couples source-agent dans la figure 5.14. Nous avons ajouté l'utilisation du serveur. Il y a donc huit graphiques dans la figure correspondant à quatre séries de simulation. Dans tous les cas, le temps analytique T_S est plus grand que le temps mesuré dans la simulation. Ce résultat était prévisible parce que la valeur estimée de μ_i (figure 4.12) est inférieure à la vraie valeur et la différence entre ces deux valeurs augmente à mesure que le nombre de queues n augmente.

Nous avons vu dans la section 5.3 que les performance du mécanisme de localisation à base de serveur se dégradent rapidement quand le taux de migration ν augmente. Cela est encore plus vrai dans le cas multi-queues en particulier parce que le taux de migration augmente pour tous les agents en même temps. En étudiant la figure 5.14 nous pouvons voir que notre approximation sur μ_i amène une erreur plus faible sur le temps de réponse quand le taux de migration ν est plus faible pour un même taux de communication λ. À l'inverse, faire varier le taux de communication λ de 1 à 10 tout en gardant un taux de migration ν constant n'a que peu d'effets sur la qualité de l'approximation (voir figures 5.14 (a) et (c) et figures 5.14 (b) et (d)).

TAB. 5.7 – *Utilisation et nombre de couples source-agent amenant une erreur de 10% et 15%*

	10% d'erreur		15% d'erreur	
Paramètres	Utilisation	Nombre de couples	Utilisation	Nombre de couples
$\lambda = 1$,$\nu = 1$	0.70	92	N. D. (> 0.77)	N. D. (> 100)
$\lambda = 1$,$\nu = 10$	0.48	21	0.55	25
$\lambda = 10$,$\nu = 1$	0.73	59	0.81	66
$\lambda = 10$,$\nu = 10$	0.50	8	0.61	10

La table 5.7 donne les informations suivantes: avec une tolérance de 10% d'erreur (resp. 15% d'erreur) sur le temps de réponse l'approximation du modèle peut être utilisée tant que l'utilisation du serveur ne dépasse pas 0.73 (resp. 0.81) ce qui est atteint avec 59 (resp. 66) couples source-agent pour $\lambda = 10$ et $\nu = 1$ (voir ligne 3 de la table 5.7). Notons que dans le cas où $\lambda = 1$ et $\nu = 1$, l'erreur maximale obtenue est de 14.3%. Nous n'avons donc pas l'utilisation et le nombre de couples qui amène à une erreur de 15% mais nous savons que pour une telle erreur, l'utilisation du serveur doit être supérieure à 0.77 et que le nombre de couples source-agent doit être supérieur à 100 (voir ligne 1 dans la table 5.7).

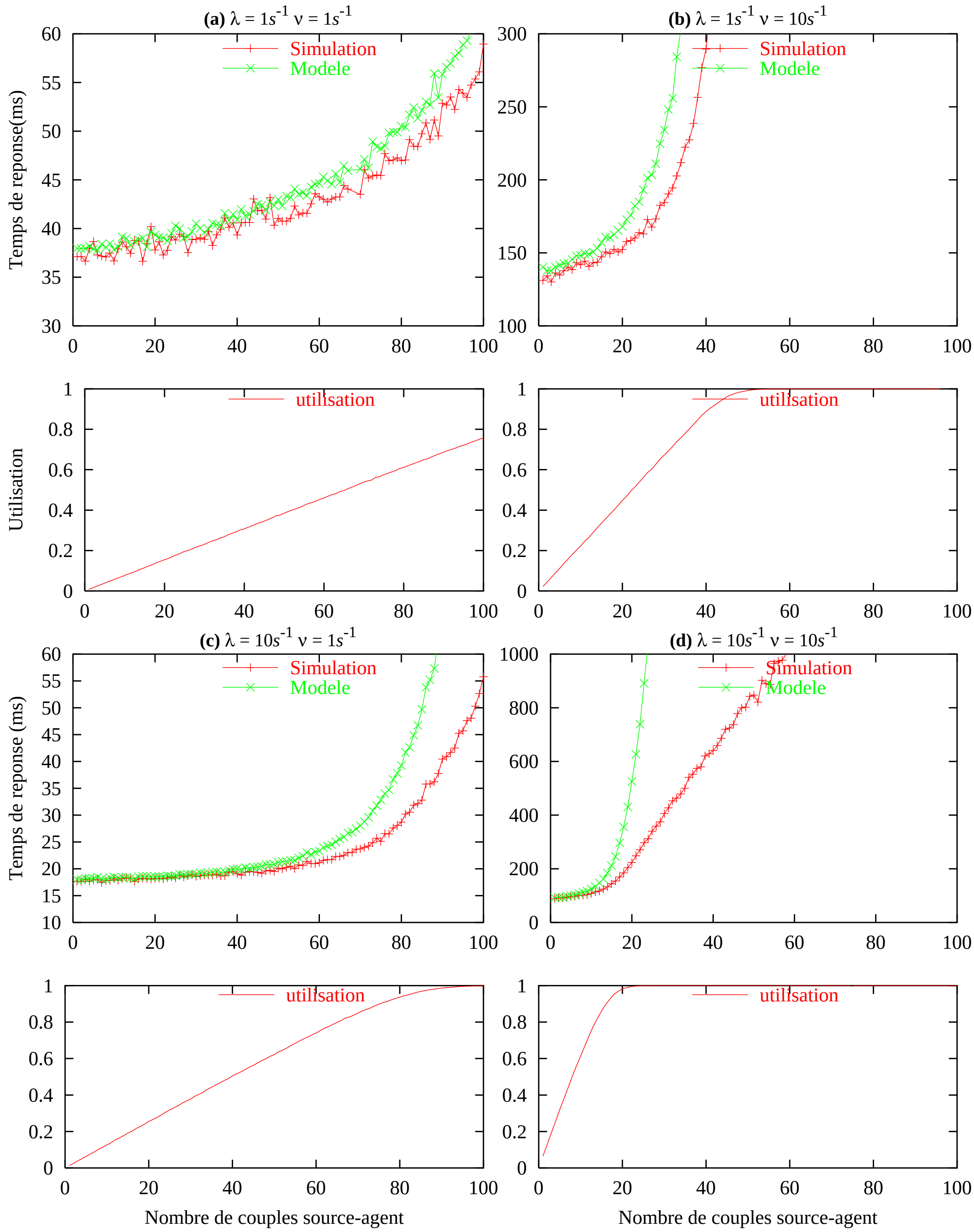


FIG. 5.14 – *Temps de réponse simulé et analytique et taux d'utilisation du serveur sur un LAN.*

Pour conclure cette section nous pouvons dire que, dans tous les cas, notre modèle approximé retourne une bonne estimation du temps de réponse tant que l'utilisation du serveur ne dépasse pas 50%. Il est ainsi possible sans changer notre modélisation d'obtenir des résultats acceptables dans le cas où plusieurs sources et plusieurs agents utilisent le même serveur.

5.5 Conclusion

Nous avons dans ce chapitre entrepris la validation des modèles décrits dans le chapitre 4. Cette validation s'est effectuée en plusieurs étapes. Tout d'abord un simulateur a été construit pour nous permettre de vérifier nos résultats dans le cas idéal où les hypothèses formulées pour permettre la résolution mathématique sont vérifiées. Utilisant le simulateur, nous avons pu vérifier la robustesse de nos modèles quand ces hypothèses ne sont pas vérifiées, nous permettant ainsi d'entreprendre des expérimentations grandeur nature. Les résultats prédis par les formules théoriques se sont révélés très proches de ceux mesurés sur un réseau local, alors que l'erreur relative sur un réseau régional, bien que plus importante, restait tout à fait acceptable. Nous avons donc montré que les modèles développés sous la contrainte d'hypothèses fortes (indépendance des paramètres et lois exponentielles) sont parfaitement utilisables lorsque seulement deux hypothèses sont valides : le temps d'attente de la source et celui de l'agent suivent des lois exponentielles.

Confiant dans la robustesse de nos modèles, nous somme passés d'une phase de validation à une phase prédictive. La comparaison théorique nous a permis de dégager les plages de paramètres donnant de meilleures performances pour l'un ou l'autre des deux mécanismes.

Finalement, nous avons proposé et vérifié dans le cas d'un réseau local une approximation du modèle du serveur centralisé quand plusieurs sources et plusieurs agents utilisent un même serveur, qui devient donc un goulet d'étranglement.

Chapitre 6

Perspective : un moyen de communication mixte

L'étude théorique que nous avons développée dans la Section 4 nous a permis de définir lequel des deux mécanismes parmi les répéteurs ou le serveur se montrait le plus performant en terme de temps de réponse. Dans le cas d'un réseau local, le serveur donne de meilleures performances alors que le contraire se produit sur un réseau régional. Nous allons donc dans cette partie nous placer dans le cadre d'un réseau régional et introduire un nouveau protocole de communication que nous appellerons protocole mixte. Idéalement, nous voudrions avoir les même performances qu'avec des répéteurs tout en évitant les inconvénients que sont la résistance aux pannes et la consommation de ressources inutilement. L'idée derrière notre protocole est d'utiliser des répéteurs seulement pendant une durée limitée, et ensuite de s'en remettre à un serveur de localisation.

6.1 Principes et algorithme

Nous nous plaçons à nouveau dans le cadre d'une application composée d'une source et d'un agent. Lorsqu'un objet migre, il se crée une chaîne de répéteurs entre la source et celui-ci. Cette chaîne restera présente tant qu'aucun message ne l'aura pas traversée et donc, si la source communique peu, consommera des ressources inutilement. Nous allons donc introduire une durée maximale de vie pour un répéteur. Quand cette durée arrive à expiration, le répéteur cesse son activité et libère les ressources qu'il utilise. Cela permettra au ramasse-miettes de le supprimer rapidement. Cependant, cette durée de vie limitée exacerbe le problème de la résistance aux pannes. Alors que jusqu'à présent, le seul événement amenant à la rupture de la chaîne était une panne de machine ou un problème réseau, nous devons maintenant tenir compte de la disparition naturelle des répéteurs. Pour cela, nous introduisons un mécanisme qui consiste à utiliser un serveur centralisé quand la chaîne de répéteurs ne conduit plus à l'agent. Il faut donc que l'agent, en plus de laisser sur chacun des sites un répéteur, contacte le serveur de localisation. Nous avons vu que le temps pris par un agent pour contacter un serveur était loin d'être négligeable sur un réseau régional. Nous avons mesuré lors de nos expérimentations (table 5.6 du chapitre 5) qu'un agent utilisant les répéteurs prend en moyenne 1/2.10 millisecondes pour migrer alors que s'il utilise un serveur il lui faut 1/1.3 millisecondes. Or la durée de migration a une influence importante sur le temps de réponse et il est donc important d'essayer de la minimiser. Partant de cette remarque et en ajoutant que le serveur est dans notre protocole un mécanisme de secours, nous avons décidé que l'agent ne devait pas contacter le serveur après chaque migration. Pour des raisons pratiques, nous avons choisi de faire la mise à jour après un certain nombre de migrations plutôt qu'après une durée.

Nous introduisons donc deux paramètres pour la localisation mixte :

Paramètre	Signification
TTL	Durée de vie d'un répéteur
TTU	Nombre de migrations avant contact du serveur

TAB. 6.1 – *Paramètres de la localisation mixte*

Nous avons choisi de considérer un TTU comme étant un nombre de migration plutôt qu'une durée à cause des difficultés que présenterait cette dernière solution. La mesure du temps écoulé au niveau d'un agent est problématique du fait de la migration. L'agent se déplace sur des sites qui n'ont à priori pas d'horloges synchronisés et ne peut donc mesurer que des durée sur chacun des sites. La durée de migration peut difficilement être prise en compte ce qui introduit une incertitude sur le temps effectivement écoulé depuis la dernière mise à jour.

Les protocoles de migration et de communications sont maintenant légèrement modifiés. Dans le cas de l'agent, il devient :

1. Début de migration
2. Copie sur le site distant
3. Création d'un répéteur
4. Si $nbMigration \geq TTU$ alors contacter serveur
5. Fin de migration

La figure 6.1 montre un agent effectuant deux migrations successives. Les numéros des étapes sur celle-ci correspondent à ceux de notre protocole. L'agent a un TTU de 2 et donc ne contacte le serveur qu'à la fin de sa deuxième migration. À l'étape 1, la première migration s'achève, un répéteur est créé et le serveur qui avait une référence vers l'agent a maintenant une référence vers un répéteur. L'étape suivante qui représente l'étape 4 du protocole de migration montre comment l'agent contacte le serveur lorsqu'il a effectué suffisamment de migrations. Une fois cette mise à jour effectuée, la migration s'achève

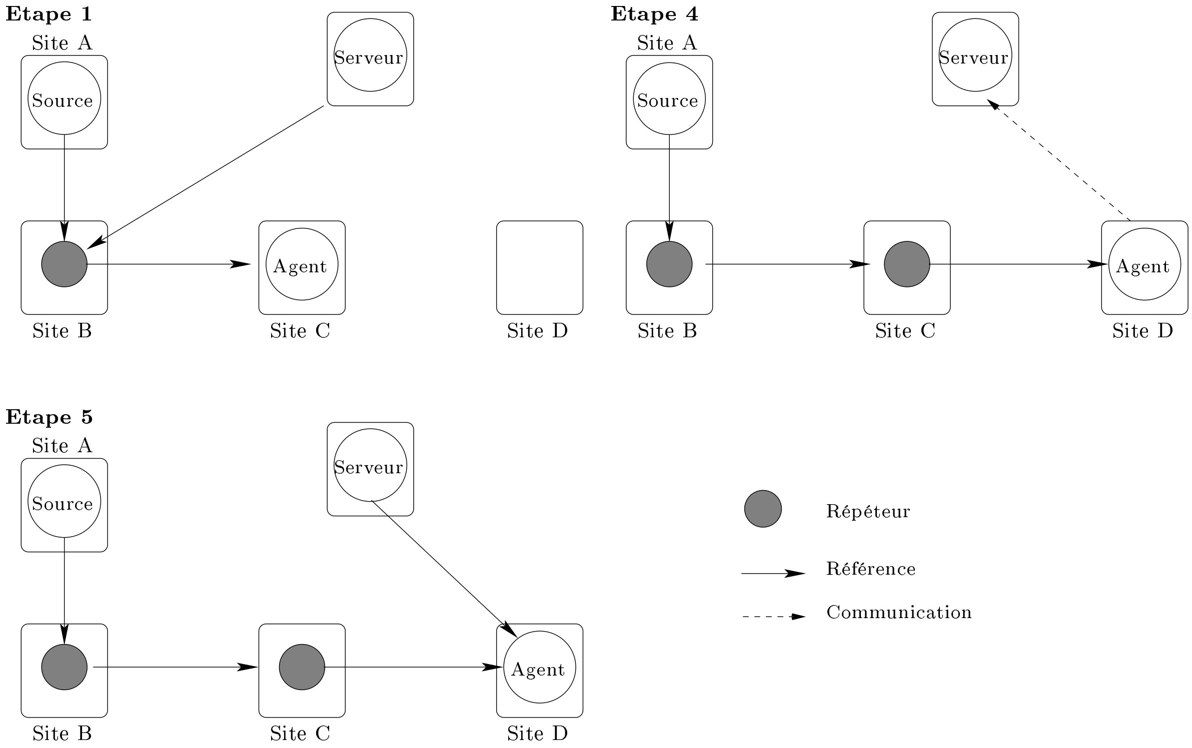


FIG. 6.1 – *Mise à jour du serveur par l'agent après 2 migrations*

et le serveur a la bonne localisation. Notons qu'il y a maintenant deux façons de joindre l'agent, soit passer par la chaîne de répéteurs, soit demander une référence explicite au serveur.

Au niveau de la source, il faut maintenant tenir compte du fait qu'une communication peut échouer, comme dans le cas d'un mécanisme à base de serveur seul. Nous avons donc le protocole suivant :

1. Début de communication
2. Contact avec le site distant
3. Si agent, aller à l'étape 6
4. Si répéteur aller en 2
5. Si erreur contacter serveur et aller en 2.
6. Fin de communication

Une illustration du fonctionnement de ce protocole est donnée dans la figure 6.2. Une source a une référence sur un répéteur qui a cessé son activité. La communication directe va donc échouer (étape 2) et la source va devoir contacter le serveur (étape 5). Notons qu'à ce moment le répéteur n'est plus référencé et peut donc être supprimé par le ramasse-miettes. La référence retournée par le serveur étant la bonne, la communication suivante sera réussie.

6.2 Implémentation

L'introduction de ce mécanisme s'est faite en modifiant les méta objets utilisés dans les mécanismes précédents (sections 3.5.1 et 3.5.2). Étant donné que ce mécanisme est un mélange des précédents, nous avons utilisé chacun des méta-objets décrits précédemment. Ainsi, nous utilisons pour l'appelant *RequestSender* capable de faire une recherche de l'agent auprès d'un serveur, et pour l'appelé un *RequestReceiver* capable d'envoyer sa nouvelle position si une requête est passée à travers des répéteurs.

6.2.1 Répéteurs à durée de vie limitée

Jusqu'à présent les répéteurs n'avaient comme limite de vie que celle décidée par le système et le ramasse miettes. Contrairement à d'autres langages orientés objets comme C++ [60], il n'est pas possible en Java d'enlever explicitement de la mémoire un objet, cette tâche reposant entièrement sur les épaules du ramasse miettes. Notre travail a donc consisté à nous assurer que le répéteur arrivé en fin de vie pouvait être ramassé le plus rapidement possible. Les répéteurs contiennent maintenant un thread qui dort pendant que le répéteur est actif. Au bout d'un temps TTL, ce thread se réveille et fait trois choses : il ferme les connexions entrantes du répéteur pour qu'il ne transmette plus les messages mais envoie à l'appelant une Exception; il contacte le serveur pour lui envoyer sa référence

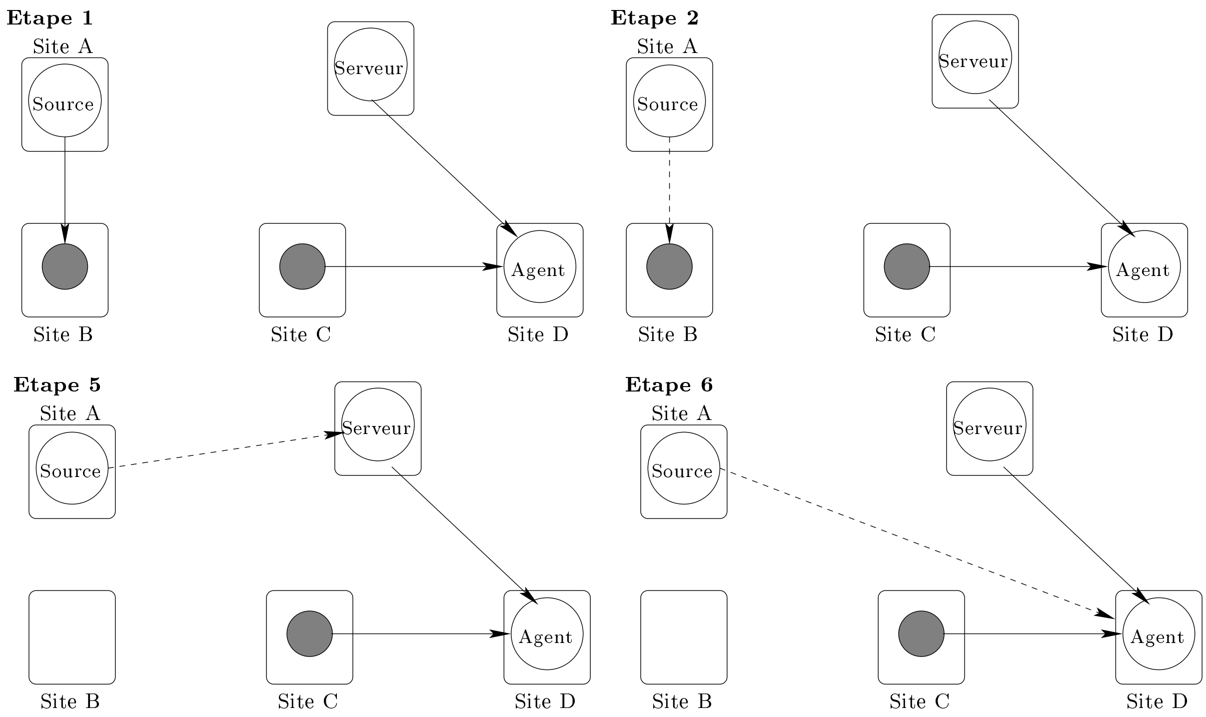


FIG. 6.2 – *Utilisation du serveur par la source quand la chaîne est inactive*

sortante qui le relie à l'agent ou à un autre répéteur; il coupe cette connexion sortante. Il convient de noter que le répéteur ne pourra pas être enlevé par le ramasse-miettes tant que d'autres objets auront des références sur lui. Ainsi, si il est au milieu d'une chaîne il faut que le répéteur le précédant soit lui aussi devenu inactif. En appelant $ttl_{i,t}$ la durée de vie restante du i-ème répéteur à l'instant t il est souhaitable d'avoir :

$$ttl_{i-1,t} < ttl_{i,t}$$

ce qui indique simplement que le temps de vie restant à chaque répéteur dans une chaîne doit être croissant, celui se trouvant le plus près de la source ayant la durée de vie la plus faible. En pratique si les répéteurs ont une durée de vie identique fixée lors de leur création, alors nous aurons bien cette propriété car le répéteur $i-1$ a été crée avant le répéteur i.

6.3 Valeurs des paramètres: éléments de choix

Nous avons dans notre mécanisme introduit deux nouveaux paramètres, la durée de vie d'un répéteur (TTL) et le nombre de migrations effectuées (TTU) par un agent avant de mettre à jour le serveur. Le but de cette section est de donner des éléments permettant, sinon de choisir de manière optimale des valeurs pour ces paramètres, de diminuer les plages de valeurs possibles. Nous introduisons tout d'abord une définition pour décrire l'état d'une chaîne de répéteurs.

Définition 6.3.1 *Une chaîne de répéteurs sera dite active ou vivante si tous les répéteurs la composant sont actifs ou vivants.*

Le meilleur temps de réponse est obtenu sur un MAN en utilisant les répéteurs. Notre nouveau mécanisme, du fait même de son implémentation (répéteurs et serveur si nécessaire), devrait donc avoir un temps de réponse supérieur à celui de la chaîne de répéteurs. Notons cependant que la limite supérieure ne sera pas forcément le temps donné par le serveur centralisé. En effet, si le nombre de migrations avant mis à jour (TTU) est très élevé et que la durée de vie des répéteurs (TTL) est très faible, il est possible que l'agent trouve une chaîne inactive et que le serveur n'ait pas la bonne localisation. Nous serons donc dans une situation où l'agent fait plusieurs tentatives de communication et doit demander plusieurs fois la nouvelle localisation au serveur. Mais celui-ci n'étant mis à jour que très rarement, la source n'aura pas immédiatement la bonne référence. Idéalement, il faudrait que notre nouveau mécanisme ait un temps aussi proche que possible de celui des répéteurs tout en n'ayant pas leurs inconvénients. Pour arriver à ce résultat, il faut que la majorité des communications trouve la chaîne des répéteurs active. Nous allons donc essayer de maximiser le taux de succès à la première tentative, ce qui devrait rapprocher le temps de réponse de celui idéal des répéteurs.

Pour confirmer cette hypothèse nous avons effectué plusieurs simulations dont les résultats sont détaillés dans la figure 6.3. Pour chacune des simulations nous avons fixé une durée de vie aux répéteurs (TTL) et fait varier le nombre de migrations avant d'appeler le serveur (TTU) de 1 à 30. La figure se décompose en quatre parties, notées (a), (b), (c) et (d) correspondant à quatre couples de taux de migration et de taux de communications. Pour chacun d'entre eux, nous avons deux graphiques, le premier indiquant le taux de réponse en fonction du TTU, le deuxième le taux de réussite au premier essai en fonction du même paramètre.

Intéressons-nous tout d'abord au temps de réponse. Trois comportements sont visibles ; pour certaines valeurs du TTL, le temps diverge très rapidement pour dépasser le temps de réponse obtenu avec un serveur seul (TTL de 1000 millisecondes pour $\lambda = 1$, $\nu = 1$ par exemple). Pour d'autres valeurs du TTL, le temps croît légèrement avec l'augmentation du TTU (TTL de 300 ms pour $\lambda = 10$, $\nu = 1$). Finalement, dans un troisième cas le temps de réponse reste pratiquement constant pour n'importe quelle valeur du TTU (TTL de 5000 ms pour $\lambda = 1$, $\nu = 10$). Nous n'allons dans la suite considérer que les TTL donnant un temps de réponse légèrement croissant ou constant. Pour ceux-ci, nous pouvons observer que le temps de réponse diminue légèrement quand le TTU passe de 1 à 3 puis soit reste constant soit augmente. Ce phénomène s'explique facilement en rappelant que le TTU influe directement sur la durée de migration, qui elle-même conditionne le temps de réponse. Une durée de migration plus longue augmentera le temps moyen de réponse parce que lorsque la source essaiera de communiquer avec un agent en cours de migration, elle restera bloquée jusqu'à la fin de celle-ci. Intéressons nous maintenant au taux de succès lors du premier essai. Nous voyons que dans tous les couples, les temps de réponse qui divergent sont associés à un taux de réussite plus faible. Quand $\lambda = 1$ et $\nu = 1$, un TTL de 1000 millisecondes donne un taux de succès d'environ 85%, alors que pour un TTL de 5000ms, le taux de succès est de plus de 98% en moyenne. Cependant, nous pouvons voir qu'une différence très faible dans le taux de réussite ($\lambda = 10$, $\nu = 1$) peut avoir une conséquence très importante sur le temps de réponse. Dans toutes nos simulations, nous avons pu observer que le taux de succès au premier essai augmente quand le TTU passe de 1 à 3. Nous n'avons pas trouvé d'explication concernant ce phénomène.

Il semble donc que le taux de succès soit un indicateur du temps de réponse moyen que le mécanisme va permettre d'obtenir.

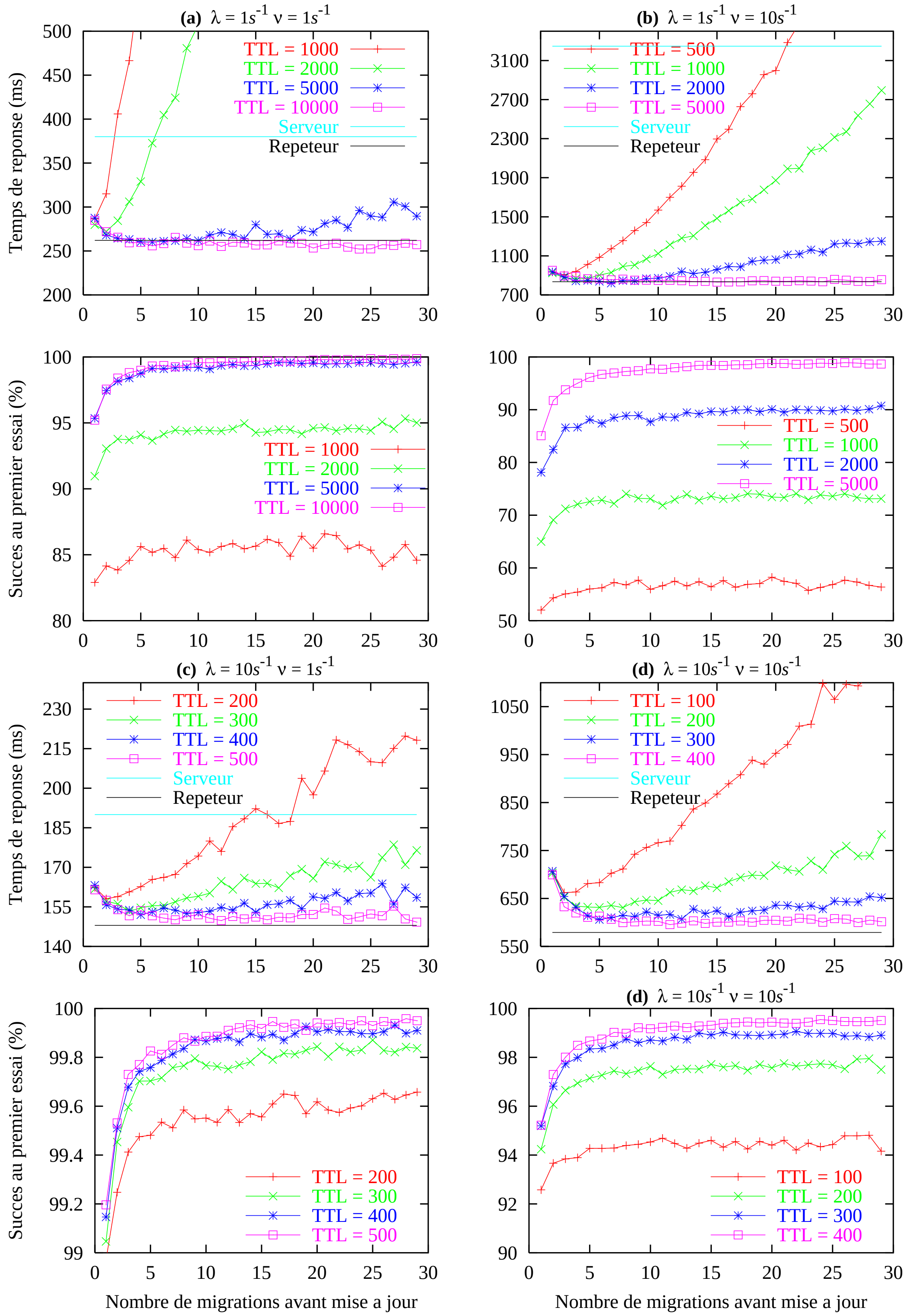


FIG. 6.3 – *Simulation de la communication mixte sur un MAN.*

6.4 Extension : mise à jour du serveur par les répéteurs

Lorsqu'une source demande au serveur la localisation d'un agent parce que la chaîne de répéteurs n'est plus active, il peut arriver que l'information contenue dans celui-ci ne soit pas à jour si par exemple le TTU de l'agent est élevé. Dans ce cas, la source ne pourra rien faire d'autre que faire des tentatives de communications et des demandes au serveur jusqu'à réussir à communiquer. Pour résoudre ce problème, nous allons donner aux répéteurs la possibilité de mettre à jour le serveur. Quand ils arrivent en fin d'activité, ils contactent le serveur pour lui donner la référence vers le suivant dans la chaîne qui est éventuellement l'agent. Cela permet de maintenir à jour la table de localisation du serveur même en l'absence de migration, comme cela est illustré dans la figure 6.4.

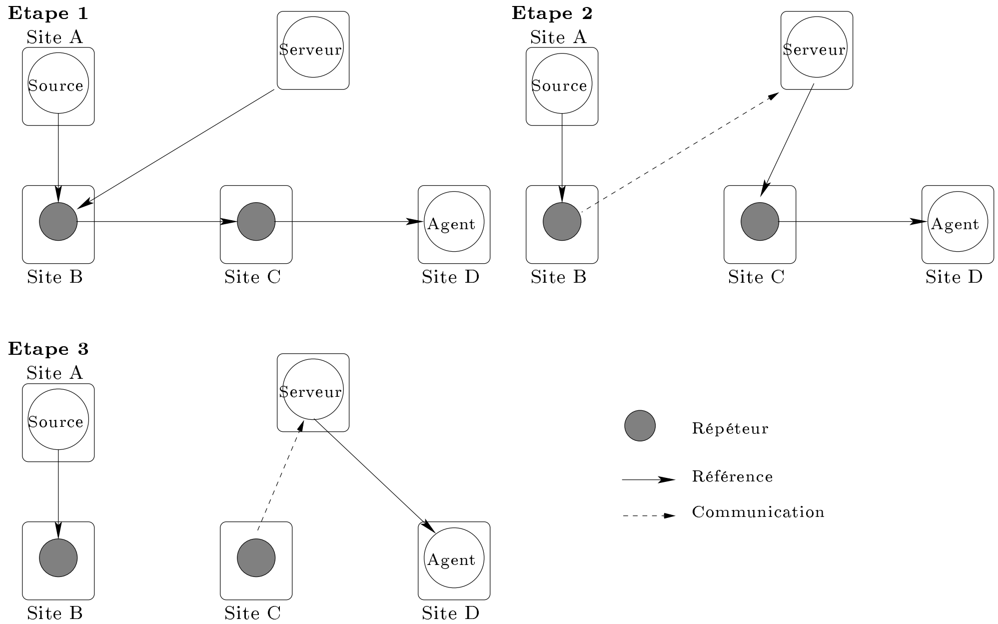


FIG. 6.4 – *Mise à jour des référence à la fin de l'activité d'un répéteur*

6.4.1 Serveur

Le serveur de localisation reçoit maintenant des requêtes de trois entités : la source, l'agent et le répéteur. Le traitement pour les requêtes de la source ne change pas mais, il faut apporter un soin particulier aux deux autres. Il faut en effet être capable de faire la différence entre les requêtes d'un agent et d'un répéteur d'une part, et entre deux requêtes de répéteurs d'autre part, afin de toujours prendre celle qui indique la meilleur position de l'agent. La requête venant d'un agent est forcément la meilleur car elle indique que celui-ci vient juste de finir une migration, il suffit donc juste de les différencier. Dans le cas de deux requêtes de répéteurs, il faut prendre celle envoyée par le répéteur situé au plus prés de l'agent. Pour obtenir cette information nous ajoutons à l'agent un compteur de migration strictement croissant. La valeur de ce compteur est laissée dans le répéteur quand l'agent quitte un site. Lorsqu'un répéteur contacte le serveur, il envoie cette valeur dans la requête. Il ne reste au serveur qu'à traiter la requête ayant le compteur de migration le plus élevé.

6.4.2 Simulation

Nous avons effectué le même type de simulations que précédemment pour vérifier l'impact qu'a la mise à jour du serveur par les répéteurs. Les résultats sont indiqués dans la figure 6.5.

Nous voyons que cette fois il n'y a pas de phénomène d'explosion du temps de réponse quand le TTU augmente mais qu'au contraire, celui-ci reste relativement constant. La durée de vie des répéteurs (TTL) a une influence positive sur le temps de réponse qui diminue lorsque ceux-ci restent actifs plus longtemps. Globalement, nous obtenons de meilleures performances lorsque les répéteurs mettent à jour le serveur. Il semble donc que la source perdait précédemment un temps significatif à attendre que l'agent mette à jour sa localisation auprès du serveur. Notons qu'avec ce nouveau système, il n'est plus forcément nécessaire pour un agent de mettre à jour le serveur car ce sont les répéteurs qui le feront pour lui. En cas de rupture de la chaîne à cause d'une panne de machine, la mise à jour sera effectuée par le prochain répéteur de la chaîne et donc au final il y aura bien une mise à jour de faite. Un problème peut surgir quand c'est le dernier répéteur de la chaîne qui n'est plus fonctionnel et qu'il n'a pas pu contacter le serveur. Dans ce cas il n'y aura pas de mise à jour avant qu'un des deux événements suivants ne se produise : l'agent migre et atteint son TTU, ce qui provoque une mise à jour du serveur ; l'agent migre au moins une fois et un des répéteurs atteint son TTL et met à jour le serveur. Dans le cas où l'agent ne migre pas alors il est possible qu'il ne soit plus joignable. Il est possible de palier à ce problème en spécifiant un temps maximal passé sur un site avant de contacter le serveur. Ainsi, un agent qui ne migrerait plus atteindrait cette limite et appellerait le serveur de localisation alors qu'il n'a pas migré. Notons que cette nouvelle variable ne dépend que du site en cours et donc ne pose pas les même problèmes que la version continue du TTU évoquée auparavant.

Il est possible d'avoir une estimation du taux d'arrivée des requêtes en reprenant les notations introduites lors de la modélisation et en considérant que l'agent ne met jamais à jour le serveur de localisation. Nous avons dessiné dans la figure 6.6 les étapes successives d'une migration d'un agent lorsqu'il utilise un serveur de localisation (haut) et lorsqu'il utilise notre approche mixte (bas).

Nous voyons en haut qu'après avoir attendu un temps moyen de $1/\nu$ il commence un migration, qui ne se finit qu'après un temps $1/\delta + 1/\gamma_2$ correspondant à la durée de la migration plus le contact du serveur. Après cette durée, le serveur a effectivement reçu la

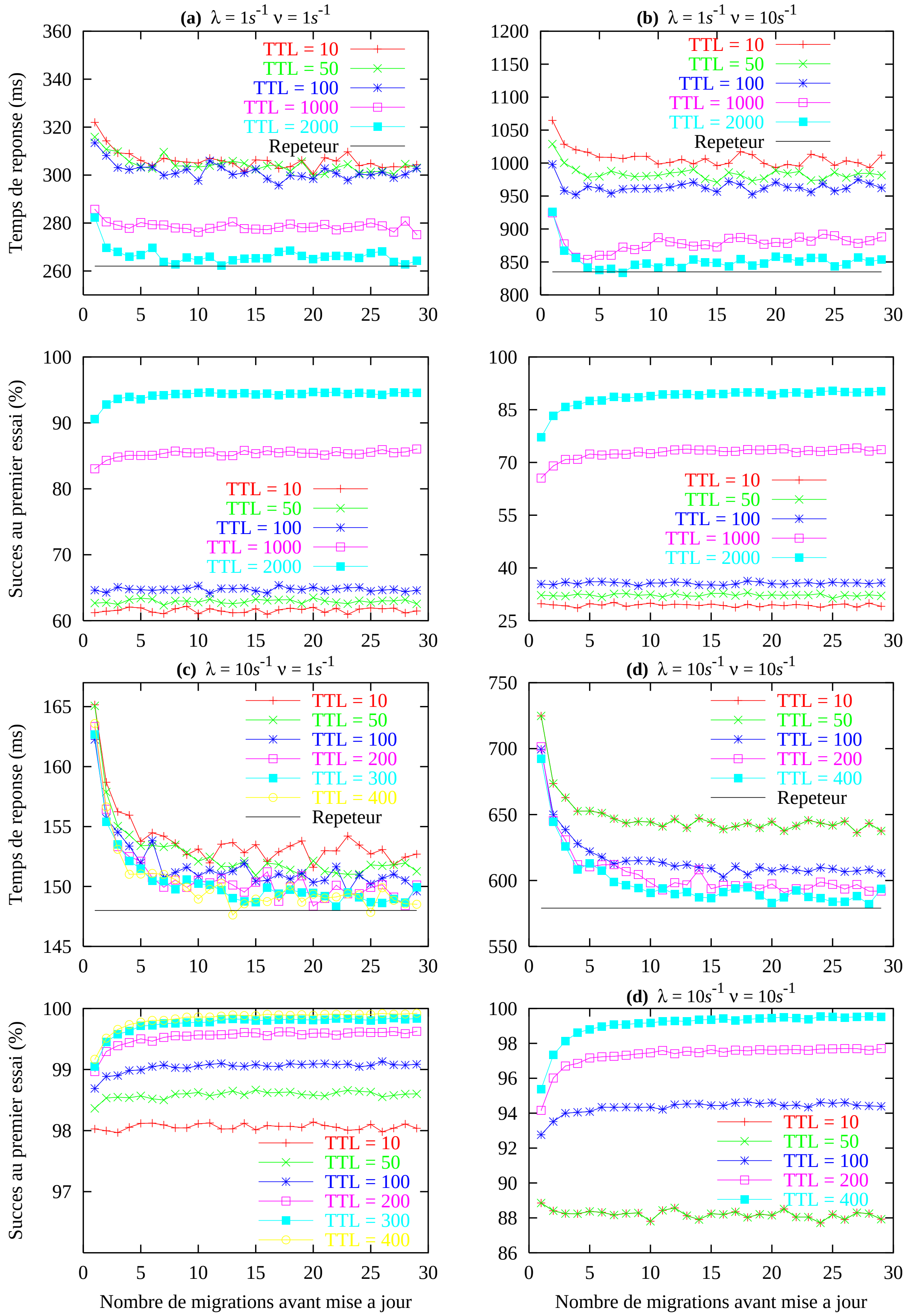


FIG. 6.5 – *Temps de réponse sur un MAN quand les répéteurs mettent à jour le serveur de localisation*

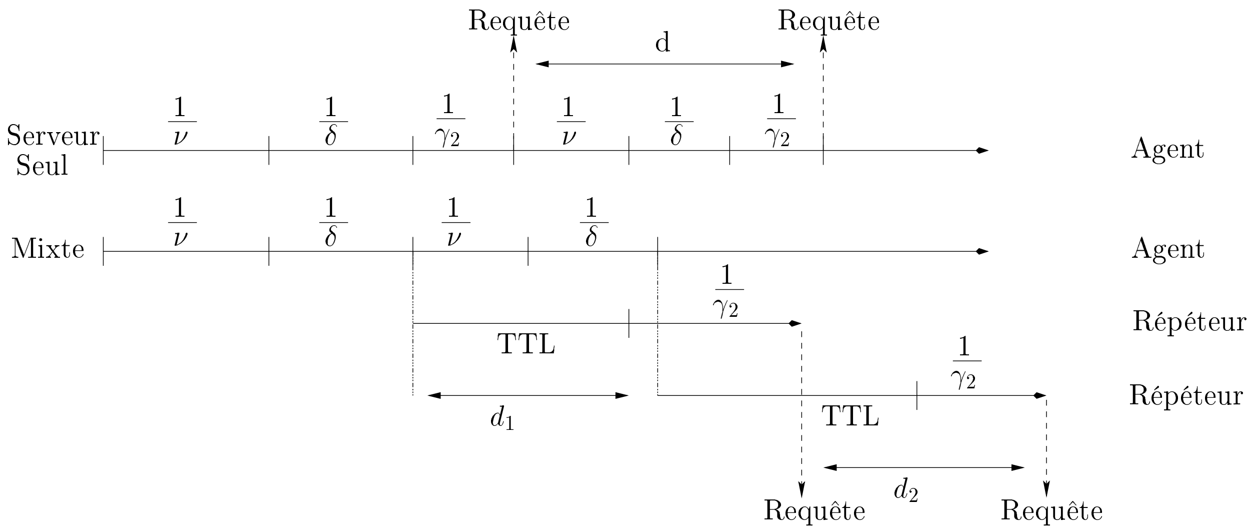


FIG. 6.6 – *Comparaison du taux d'arrivée des requêtes au serveur entre l'approche mixte et le serveur pur*

requête. Nous avons donc le résultat suivant :

Proposition 6.4.1 (Temps moyen entre deux mises à jour du serveur, serveur seulement) *Le temps moyen entre deux mises à jour du serveur de localisation par l'agent est de*

$$\frac{1}{\nu} + \frac{1}{\delta} + \frac{1}{\gamma_2}$$

Nous pouvons obtenir un résultat similaire en analysant la partie inférieur de la figure. Au bout d'un temps $1/\nu + 1/\delta$ en moyenne un répéteur est crée. L'agent n'appelle pas le serveur lui-même dans ce cas, nous avons choisi un TTU très élevé. Ce répéteur aura une durée de vie égale à TTL, après quoi il contactera le serveur. Pendant ce temps, l'agent continue de migrer et crée un deuxième répéteur qui lui aussi restera actif un temps TTL en moyenne avant de contacter le serveur. Nous avons donc un temps moyen de création entre deux répéteur $d_1 = 1/\nu + 1/\delta$. De même, il y a un temps d_2 séparant deux réceptions de requêtes au serveur, et il est clair que $d_1 = d_2$. Nous avons donc :

Proposition 6.4.2 (Temps moyen entre deux mises à jour du serveur, approche mixte) *Le temps moyen entre deux mises à jour du serveur de localisation par les répéteurs est de*

$$\frac{1}{\nu} + \frac{1}{\delta}$$

En comparant les deux situations, nous voyons que l'approche mixte avec mise à jour du serveur par les répéteurs devrait augmenter légèrement la charge générale du serveur induite par l'agent. En effet, le temps moyen entre deux messages de mise à jour sera moins long. Cependant, la source l'utilisera moins et donc on peut s'attendre à une baisse générale de l'utilisation car le traitement d'une demande d'une source est beaucoup plus coûteux que celle d'un agent, à cause de la réponse à envoyer.

6.4.3 Cas multi-sources multi-agents

Il faut vérifier notre hypothèse selon laquelle même si notre dernier mécanisme introduit une charge plus importante sur le serveur à cause des mises à jour des répéteurs, nous allons *in fine* obtenir un meilleur temps de réponse parce qu'une source aura moins recours au serveur de localisation. Pour cela, nous avons effectué de simulations dont les résultats sont donnés dans la figure 6.7.

Nous pouvons observer le même comportement que dans le cas multi-queue avec le serveur seul, c'est à dire qu'il existe une corrélation entre le temps de réponse et l'utilisation du serveur. Par exemple dans le cas $\lambda = 1, \nu = 1$ (figure 6.7 (a)) si le TTL vaut 10 000ms alors l'utilisation du serveur est inférieure à 1% et le temps de réponse est excellent. Si nous avons le TTL qui ne vaut plus que 5000ms alors l'utilisation augmente rapidement (50% pour 25 couples, 70% pour 50 et 93% pour 100). Cette charge du serveur a pour conséquence un temps de réponse très variable et relativement élevé. Notons cependant que les résultats sont bien meilleurs que ceux obtenus dans les simulations multi-queues du serveur dans des conditions MAN (figure C.7 page 149), ce qui montre bien l'intérêt de ce mécanisme en terme de performances et de montée en charge.

6.5 Conclusion

À travers ce chapitre nous avons étudié un protocole de communication mélangeant les répéteurs et le serveur centralisé afin d'obtenir la fiabilité associée à de bonnes performances. Cette approche ajoute deux nouveaux paramètres, la durée de vie d'un répéteur (TTL) et le nombre de migrations de l'agent avant une mise à jour du serveur (TTU). En choisissant judicieusement leurs valeurs, il est possible d'avoir des performances très proches de celles obtenues avec des répéteurs seuls, tout en limitant leur prolifération et en palliant à toute rupture de la chaîne.

Une amélioration de ce protocole est possible si les répéteurs arrivant en fin d'activité mettent à jour le serveur de localisation. Cela nécessite une modification du serveur pour qu'il puisse discriminer entre les requêtes des répéteurs et de l'agent, mais permet au final d'avoir un TTU très élevé ce qui diminue la durée de migration.

Ce protocole permet aussi une meilleur montée en charge dans le cas où un même serveur est utilisé par plusieurs sources et plusieurs agents.

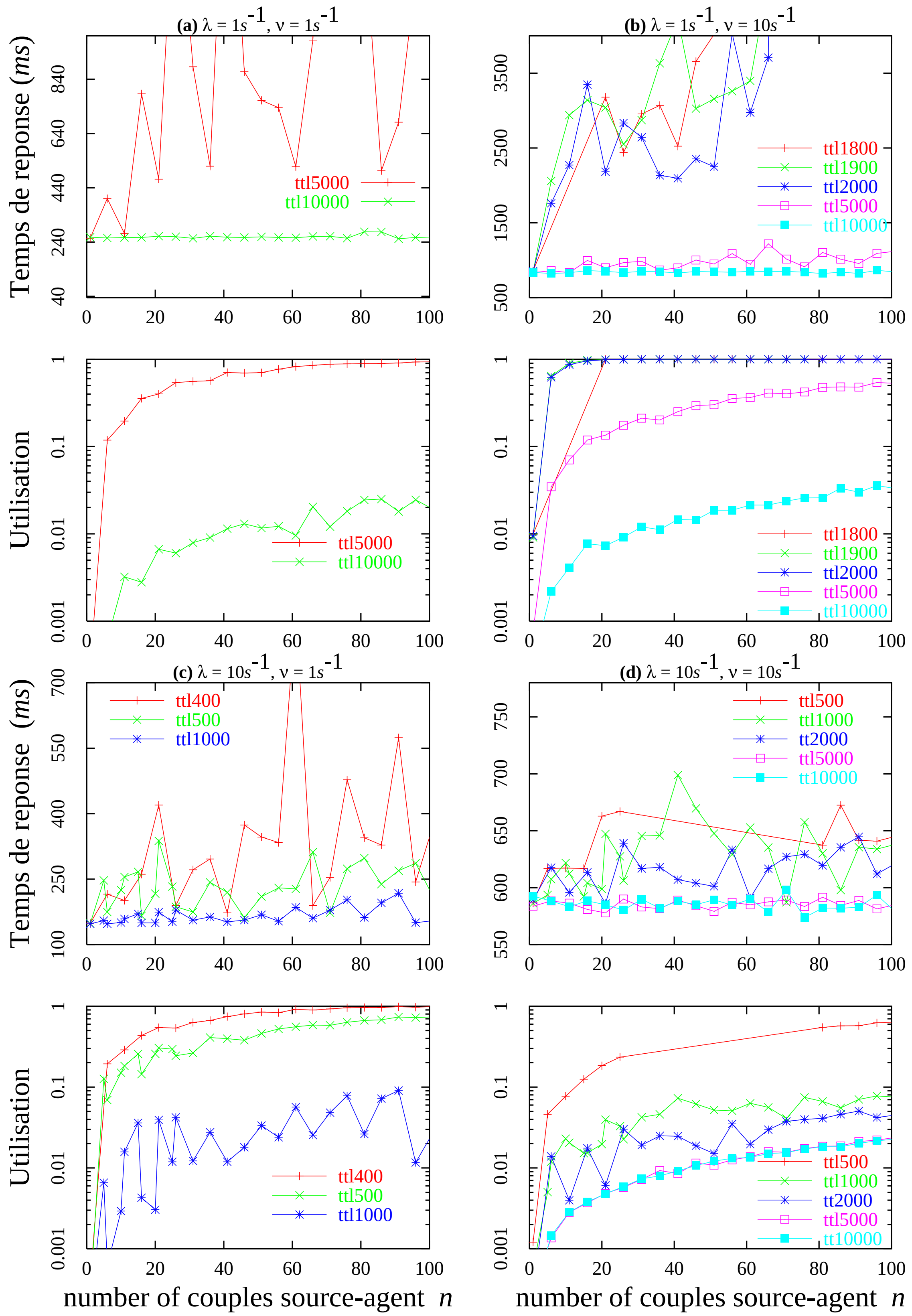


FIG. 6.7 – *Simulation multi-sources multi-agents du protocole mixte dans le cadre d'un MAN* ($\delta = 2.87$, $\gamma_1 = 12.3$, $\gamma_2 = 12.1$, $\mu = 938$, $TTU = 50$)

Chapitre 7

Conclusion

7.1 Bilan, contributions

Nous nous sommes dans cette thèse intéressés à deux problèmes. Le premier concerne la conception d'une bibliothèque de migration faible dans le cadre des objets actifs. Nous avons montré qu'il est possible d'utiliser les caractéristiques de ce modèle, en particulier l'activité des objets pour avoir facilement une sauvegarde de l'état d'un objet. En utilisant les techniques de réflexion, nous avons pu implémenter la migration et les mécanismes de communication associés de manière élégante. Dans notre étude des mécanismes de communication, nous avons vu que deux mécanismes, les répéteurs et le serveur de localisation, étaient très souvent utilisés, sans qu'aucune justification formelle concernant leurs performances ne soit donnée.

Nous avons donc entrepris dans un deuxième temps une étude de ces deux mécanismes en utilisant des chaînes de Markov. Pour chacun d'entre eux, nous avons obtenu une formulation du temps de réponse et, pour les répéteurs, une métrique concernant le nombre moyen de répéteurs. Il est apparu au cours de notre étude que la durée de migration et la durée de communication inter-sites avaient une influence importante sur les performances. Concernant la durée de migration, nous avons montré, dans le modèle des répéteurs, que son influence dépend étroitement du taux de migration et de la latence inter-sites.

La validation de ces deux modèles a été effectuée à travers des simulations et des expérimentations sur un réseau local et régional. Malgré les hypothèses fortes utilisées pour construite nos modèles, nous avons montré que les performances étaient tout à fait acceptables dans des cas réalistes. Nous avons alors entrepris une comparaison théorique des performances afin de choisir suivant les conditions le mécanisme donnant les meilleures performances.

Finalement, dans une dernière partie nous nous somme intéressés à une approche mixte consistant à utiliser des répéteurs à durée de vie limitée et un serveur de localisation. Nous nous sommes en particulier attaché à dégager des pistes permettant de choisir une durée de vie pour les répéteurs et un taux de mise à jour du serveur par un agent. L'idée étant encore une fois de minimiser le temps de réponse. Nous avons étudié deux variantes de ce mécanisme et nous avons montré comment la deuxième version, où les répéteurs mettent à jour le serveur, permet d'obtenir de bonnes performances dans les cas multi-soures multi-agents.

En plus des résultats obtenus, une contribution de cette thèse réside dans les méthodes employées. Nous avons montré comment il était possible d'utiliser des techniques simples d'évaluation de performances pour caractériser les performances de mécanismes relativement complexes. L'ordre de présentation (modèle-simulation-expérience) reflète les trois étapes qui nous semblent indispensables à toute évaluation de performances. Au cours de notre travail nous avons envisagé plusieurs modèles plus simples pour chacun des deux mécanismes de communication, mais ceux-ci n'ont pas passé l'épreuve de la validation.

7.2 Perspectives

7.2.1 Auto adaptation

Sur un plan pratique tout d'abord, nous pouvons utiliser les modèles développés pour avoir un comportement adaptatif. Une application pourrait ainsi mesurer les différents paramètres au cours de l'exécution et choisir d'utiliser un serveur plutôt que des répéteurs pour assurer les communications. Par exemple, si un agent se déplace sur un réseau local, il pourrait utiliser le serveur qui donne de meilleures performances, et plus tard, lorsqu'il quittera le réseau local pourrait utiliser des répéteurs.

Grâce à l'approche mixte développée dans le dernier chapitre de cette thèse, nous pouvons faire de l'adaptation beaucoup plus fine que l'utilisation de l'un ou l'autre des deux mécanismes. En effet, en jouant sur les deux paramètres que sont la durée de vie des répéteurs (TTL) et le nombre de migrations à effectuer avant d'appeler le serveur (TTU) nous avons toute une plage de comportements accessibles. Il est ainsi possible d'avoir le mécanisme des répéteurs ou celui du serveur en choisissant certaines valeurs pour le TTL et le TTU comme indiqué dans la table 7.1. En pratique pour obtenir le mécanisme des

Paramètres	Mécanisme
$TTL = \text{inf} , TTU = \text{inf}$	Répéteurs
$TTL = 0, TTU = 1$	Serveur

TAB. 7.1 – *Paramètres du mécanisme mixte pour obtenir le serveur ou les répéteurs*

répéteurs, il suffit d'avoir un TTL suffisamment grand pour que toute communication tentée par une source trouve la chaîne active. Dans ce cas, l'agent n'a pas besoin de contacter le serveur de localisation pour mettre à jour sa position. Dans le cas où nous considérons l'approche mixte avec mise à jour par les répéteurs, alors fixer un TTL à 0 pour les répéteurs induira une charge plus importante pour le serveur qui devra ignorer ces requêtes car redondantes avec celles de l'agent.

En reprenant le scénario d'un agent travaillant d'abord sur un réseau local puis allant finir sa tâche sur Internet, nous pouvons imaginer le comportement suivant. L'agent commence par utiliser un serveur de localisation quand il se trouve sur le réseau local. En quittant celui-ci, il pourrait utiliser des répéteurs mais nous avons vu que cela pose un problème de résistance aux pannes. Il décide plutôt de laisser des répéteurs à durée de vie limitée et augmente légèrement son TTU pour ne pas appeler le serveur à chaque migration. Il fait ensuite évoluer ces deux paramètres suivant les informations qu'il observe. Si par exemple la source le contacte toutes les n secondes, alors il peut être efficace d'avoir un TTL de $2 * n$ pour avoir un bon taux de réussite au premier essai, et ne pas laisser les répéteurs en vie trop longtemps. Mais à ce moment se pose la question de la mise à jour du serveur par les répéteurs : ce mécanisme est-il nécessaire étant donné les paramètres choisis ? Induit-il une charge excessive sur le serveur ?

Toutes ces questions ne peuvent être répondues simplement. Dans les cas plus simples du serveur seul ou des répéteurs seuls, nous avons vu que la dynamique de ces mécanismes est loin d'être triviale. Il semble donc clair que choisir des paramètres proches de l'optimal suivant les conditions va nécessiter d'avoir des modèles formels semblables à ceux développés dans cette thèse.

7.2.2 Modélisation de l'approche mixte

Intéressons-nous tout d'abord au cas où les répéteurs ne mettent pas à jour le serveur de localisation. Initialement nous nous trouvons dans le cas des répéteurs. Un message est envoyé à un agent qui migre en laissant des répéteurs, mais cette fois avec une durée de vie limitée. Si un message en transit arrive à un répéteur dont la durée de vie est dépassée, alors une erreur est envoyée à la source qui doit contacter le serveur. Nous nous trouvons alors dans le cas de la modélisation du serveur de localisation. La meilleure façon de procéder nous semble être de partir de la chaîne de Markov pour les répéteurs, en ajoutant des transitions pour tenir compte de la durée de vie des répéteurs. Il nous faut donc aussi avoir un état pour caractériser la chaîne des répéteurs, i.e. savoir si elle est active. Notons au passage que la probabilité de trouver une chaîne brisée dépend de la longueur de celle-ci. Une chaîne avec n répéteurs a plus de chance de se briser qu'une chaîne de longueur m inférieur à n. Si un message arrive sur une chaîne inactive, alors la source doit contacter le serveur ; nous nous retrouvons alors dans la modélisation du mécanisme du serveur. Mais Il s'agit ici d'une approximation grossière car nous ne tenons pas compte de l'endroit où la chaîne se brise. Si un message se trouve à n répéteurs de l'agent et que le n-ème répéteur devient inactif, alors la source n'en sera avertie qu'après n communications ce qui introduira un délai non négligeable sur le temps de communication.

À travers cette courte réflexion nous voyons que la modélisation du mécanisme mixte dans le cas le plus simple où les répéteurs ne contactent pas le serveur risque de se révéler assez difficile. De cette constatation est née l'idée d'utiliser les résultats obtenus avec le mécanisme des répéteurs pour obtenir des indications pour choisir des valeurs acceptables pour le TTL et le TTU.

Nos simulations nous ont montré que le taux de réussite de la première communication, c'est-à-dire celle qui traverse complètement la chaîne des répéteurs, est en relation avec un faible temps de réponse. L'idée est de se dire qu'il faut choisir un TTL et un TTU permettant de se rapprocher au maximum du comportement des répéteurs, ou plus simplement, que traverser la chaîne soit la norme et contacter le serveur l'exception.

En étudiant la figure 7.1 nous voyons que trois scénarios sont possibles pour la première étape de la communication vers l'agent. La première ligne concerne l'agent, qui commence une migration après un temps $1/\nu$ et la finit après un temps $1/\delta$ en moyenne. Donc le premier répéteur est créé après un temps $1/\nu + 1/\delta$ en moyenne. La source commence une communication au bout d'un temps $1/\lambda$ et atteint soit l'agent, soit le premier répéteur au bout d'un temps $1/\gamma$. Nous avons donc trois possibilités :

1 $\frac{1}{\lambda} + \frac{1}{\gamma} < \frac{1}{\nu}$ l'agent est joint directement,

2 $\frac{1}{\nu} < \frac{1}{\lambda} + \frac{1}{\gamma} < \frac{1}{\nu} + \frac{1}{\delta}$ l'agent est en train de migrer, la source doit attendre,

3 $\frac{1}{\lambda} + \frac{1}{\gamma} > \frac{1}{\nu} + \frac{1}{\delta}$ la source passe par le premier répéteur.

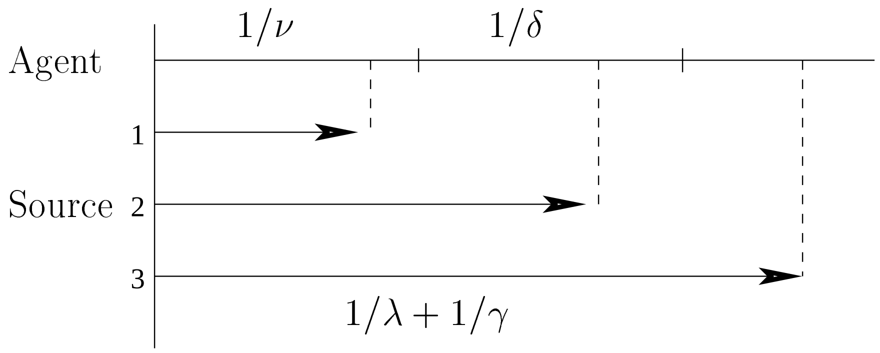


FIG. 7.1 – *Scénarios possibles pour la première communication.*

Dans le troisième cas, nous pouvons savoir en moyenne depuis combien de temps le répéteur est en vie :

Proposition 7.2.1 *Lorsque le message de la source atteint le premier répéteur (si celui-ci existe), il est en vie depuis une durée*

$$\frac{1}{\lambda} + \frac{1}{\gamma} - \left(\frac{1}{\nu} + \frac{1}{\delta}\right) \tag{7.1}$$

Pour que la source le trouve actif, il faut donc que sa durée de vie soit supérieure à la valeur donnée par l'équation (7.1), soit :

Proposition 7.2.2

$$TTL > \frac{1}{\lambda} + \frac{1}{\gamma} - \left(\frac{1}{\nu} + \frac{1}{\delta}\right) \tag{7.2}$$

Nous voyons ainsi comment il est possible d'obtenir des informations sur les paramètres de l'approche mixte en utilisant le modèle des répéteurs. Nous pensons qu'il s'agit d'une piste intéressante et prometteuse qu'il conviendrait d'explorer plus avant.

Annexe A

Rappel sur les chaînes de Markov en temps continue

Une chaîne de Markov est une séquence de valeurs aléatoires dont la probabilité durant un intervalle de temps dépend des valeurs de l'intervalle immédiatement précédent.

Il s'agit en premier lieu de définir un *espace d'états* qui représente l'ensemble des états possible pour le processus considéré. Sous certaines conditions, il est alors possible de calculer les *probabilité d'états stationnaires* qui représentent les probabilités que le système se trouve dans un état donné à un moment pris au hasard dans le futur. Elles peuvent aussi être interprétées comme la proportion de temps passé dans chacun des états au cours d'une très longue observation.

Nous n'allons pas ici donner de définition formelle mais nous concentrer essentiellement sur les propriétés et résultats de base des chaînes de Markov.

A.1 Définitions et propriétés

Définition A.1.1 (Chaîne de Markov irréductible) *Une chaîne de Markov est dite irréductible si tout état est atteignable à partir de tout autre état par des transitions de probabilités strictement positives.*

Définition A.1.2 (Chaîne de Markov apériodique) *Une chaîne de Markov est dite apériodique si le PGCD de la longueur de tous les cycles de probabilité non nulle est égal à 1.*

Définition A.1.3 (Ergodicité) *Une chaîne de Markov qui est irréductible et apériodique est dite ergodique.*

Définition A.1.4 (Générateur infinitésimal) *Le générateur infinitésimal d'un processus, noté Q, est la matrix dont les coefficients tels que :*

- $q(i,j)$ *avec* $i \neq j$ *taux de transition de l'état* i *et l'état* j.

- $q(i,i) + \sum_{i \neq j} q(i,j) = 0$

Le fait qu'une chaîne soit ergodique nous permet d'obtenir un résultat important concernant l'existance des probabilités stationnaires.

Propriété A.1.1 *Si une chaîne est ergodique et si le système d'équations*

$$\pi Q = 0$$

admet une solution strictement positive telle que $\sum_{i \in \mathcal{E}} \pi(i) = 1$, *alors pour tout* $i \in E$

$$\lim_{t \to \infty} p_t(i) = \pi(i)$$

Une fois l'existence des probabilités stationnaires établie, nous avons le résultat suivant :

Propriété A.1.2 (Conservation des flux) *Quand une chaîne de Markov est dans l'état stationnaire, le flux entrant dans un état donné est égal au flux sortant.*

Une utilisation détaillée de cette propriété est donnée dans la section 4.2.

A.2 Exemple

Nous allons considérer en exemple une chaîne de Markov à trois états, $\mathcal{E} = \{1,2,3\}$ représentée dans la figure A.1

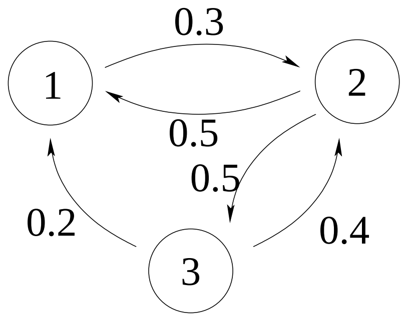


FIG. A.1 – *Diagramme de transition d'une chaîne de Markov*

Notons tout d'abord que cette chaîne est irréductible et apériodique et donc ergodique. Son générateur infinitésimal Q est

$$Q = \begin{pmatrix} -0.3 & 0.3 & 0.0 \\ 0.5 & -1.0 & 0.5 \\ 0.2 & 0.4 & -0.6 \end{pmatrix}$$

Le vecteur des probabilités stationnaires π s'obtient en résolvant $\pi Q = 0$. Les solutions de ce système sont des multiples du vecteur :

$$\left(\frac{8}{3},\frac{5}{6},1\right)$$

et en utilisant la condition de normalisation

$$\pi = \left(\frac{40}{73},\frac{18}{73},\frac{15}{73}\right)$$

Annexe B

Éléments de probabilité et de statistique

B.1 Variable aléatoire

Définition B.1.1 (Espace de probabilités) *Un espace de probabilité est un triplet (Ω,F,P) où*

- *Ω est l'ensemble de toutes les réalisations possibles associées à une expérience,*
- *F est formé de certains sous-ensembles de Ω,*
- *P est une application de F dans $[0,1]$ telle que $P(\Phi) = 0$ et $P(\Omega) = 1$.*

et F ensemble de sous-ensembles de Ω tel que :

- *$\Phi \in F$ et $\Omega \in F$*
- *si $A \in F$ alors l'ensemble complémentaire de A, noté A^c est aussi dans F*
- *si $A_n \in F$ pour $n = 1,2,\ldots$ alors $A_1 \cup A_2 \ldots \cup A_n \cup A_{\ldots}$ appartient à F.*

est appelé tribu.

Prenons un exemple simple pour bien comprendre cette définition. Considérons un dés à six faces. S'il est lancé, il y a six résultats possibles, autrement dit six réalisations

$$\Omega = \{1,2,3,4,5,6\}$$

Si nous nous intéressons à un événement particulier, par exemple “le tirage est impair”, nous aurons alors

$$A = \{1,3,5\}$$

Nous avons alors la tribu suivante

$$F = \{\Phi,\Omega,1,3,5\}$$

La mesure de probabilité sur (Ω,F) est alors donnée par

$$\begin{array}{rcl} P(\Phi) & = & 0 \\ P(\Omega) & = & 1 \\ P(A) & = & 1/2 \\ P(A^c) & = & 1/2 \end{array}$$

Définition B.1.2 (Variable aléatoire) *Une variable aléatoire X est une application de Ω dans $\mathbb{R}$ telle que*

$$\{\omega \in \Omega : X(\omega) \leq x\} \in F$$

pour tout $x \in \mathbb{R}$.

Notons que $\omega \in \Omega$ est simplement le résultat d'une expérience et que donc une variable aléatoire est une fonction qui associe un nombre à une expérience. Elle est utilisée dans le cas où on s'intéresse à un certain nombre dépendent du résultat d'une expérience plutôt qu'au résultat lui-même.

Définition B.1.3 (Fonction de répartition) *Pour toute variable aléatoire X, la fonction de répartition, ou loi, est définie par*

$$F(x) = P(X \leq x)$$

pour tout $x \in \mathbb{R}$.

B.2 Distribution de deux lois de probabilités classiques

Dans cette thèse nous utilisons deux lois de probabilités classiques dont nous allons maintenant donner les définitions.

Définition B.2.1 (Loi exponentielle) *Une variable aléatoire X est dire variable aléatoire exponentielle de paramètre α ($\alpha > 0$), ou encore a une distribution exponentielle de paramètre α si sa fonction de répartition F satisfait*

$$F(x) = \begin{cases} 1 - e^{-\alpha x} & si \quad x > 0 \\ 0 & si \quad x \leq 0 \end{cases} \tag{B.1}$$

Définition B.2.2 (Loi de Weibull généralisée) *Une variable aléatoire X est dire variable aléatoire de Weibull généralisée de forme k et d'échelle b si sa fonction de répartition F satisfait*

$$F(x) = \begin{cases} 1 - e^{-(x/b)^k} & si \quad x > 0 \\ 0 & si \quad x \leq 0 \end{cases} \tag{B.2}$$

Remarquons que dans le cas où la forme vaut 1 nous obtenons une loi exponentielle de paramètre $1/b$. La valeur de la forme donne une indication sur la représentation graphique de la fonction de distribution.

- si $0 < k < 1$ alors la courbe est convexe
- si $k = 1$ alors la courbe est décroissante
- si $k > 1$ alors la courbe est unimodale, avec son maximum en $(\frac{k-1}{k})^{\frac{1}{k}}$.

B.3 Statistique

Pour caractériser les résultats obtenus lors de nos expériences nous utilisons des pourcentiles dont la définition est la suivante :

Définition B.3.1 (Pourcentile) *Un pourcentile indique le pourcentage d'une distribution inférieur ou égal à sa valeur.*

Ainsi, le 90$^{\text{ème}}$ pourcentile indique le pourcentage de notre distribution qui est inférieur a 90%.

Annexe C

Résultats de simulation sur un MAN

C.1 Vitesse de convergence

C.1.1 Cas idéal

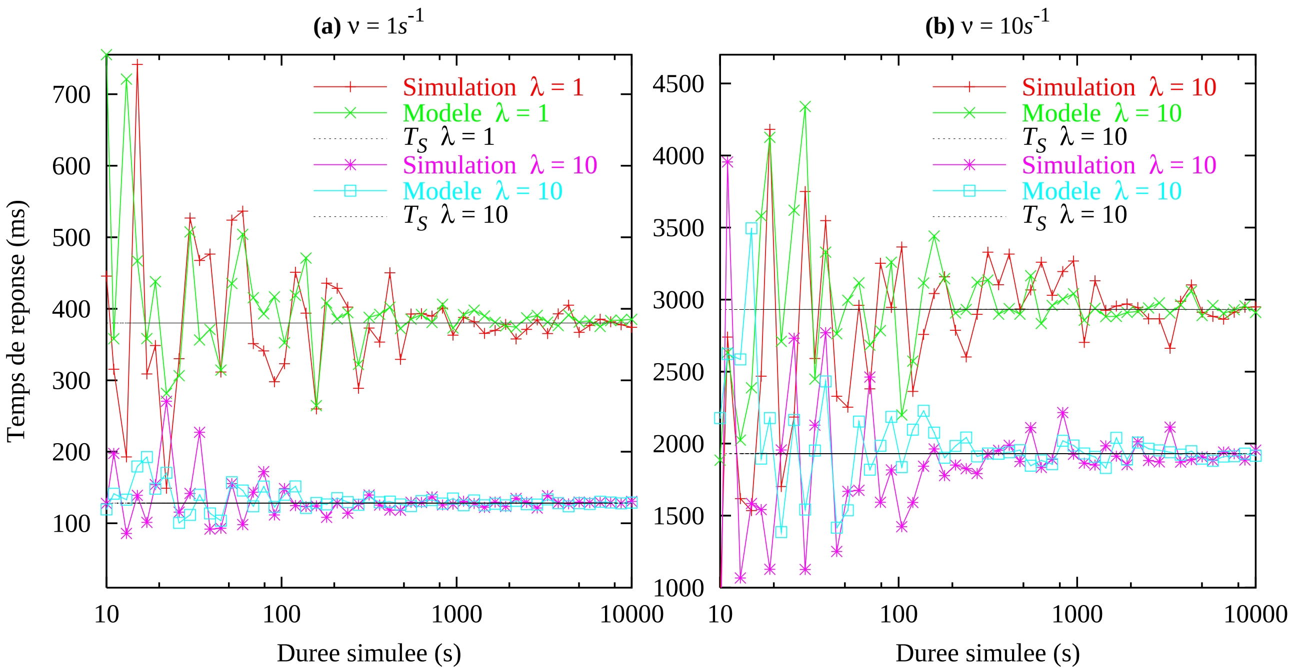


FIG. C.1 – *Convergence du simulateur et du modèle du serveur dans le cadre d'un MAN* ($\delta_S = 1.1$, $\gamma_1 = 36.7$, $\gamma_2 = 12.1$,$\mu = 938$)

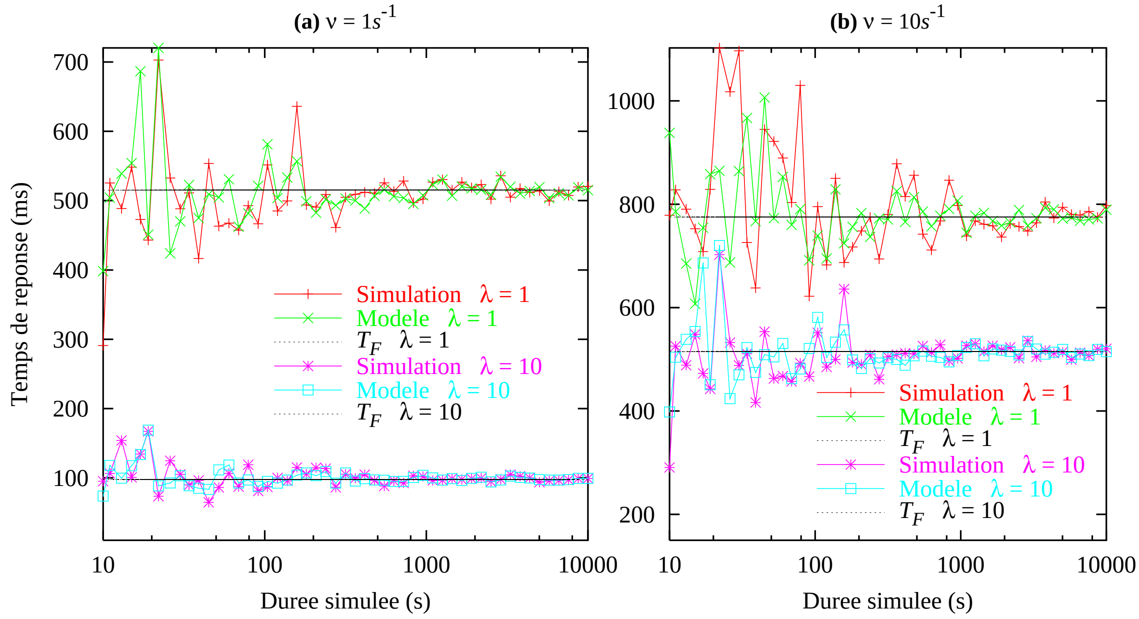


FIG. C.2 – *Convergence du simulateur et du modèle des répéteurs dans le cadre d'un MAN ($\delta_F = 2.87$, $\gamma = 12.3$)*

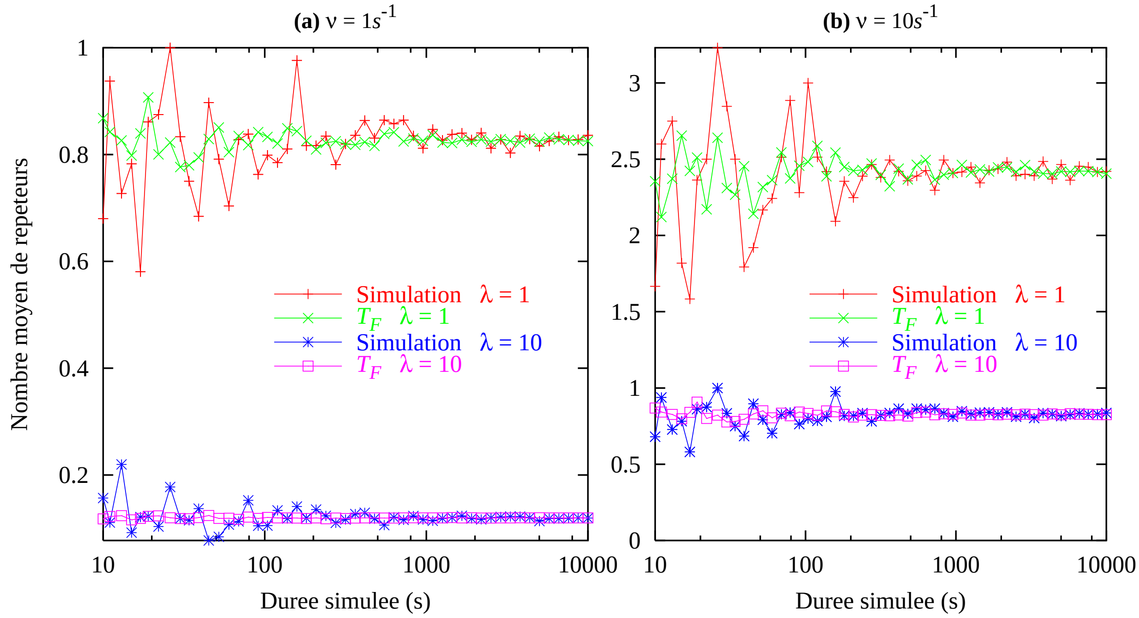


FIG. C.3 – *Convergence du simulateur et du modèle du nombre de répéteurs dans le cadre d'un MAN ($\delta_F = 2.87$, $\gamma = 12.3$)*

C.1.2 Cas réaliste

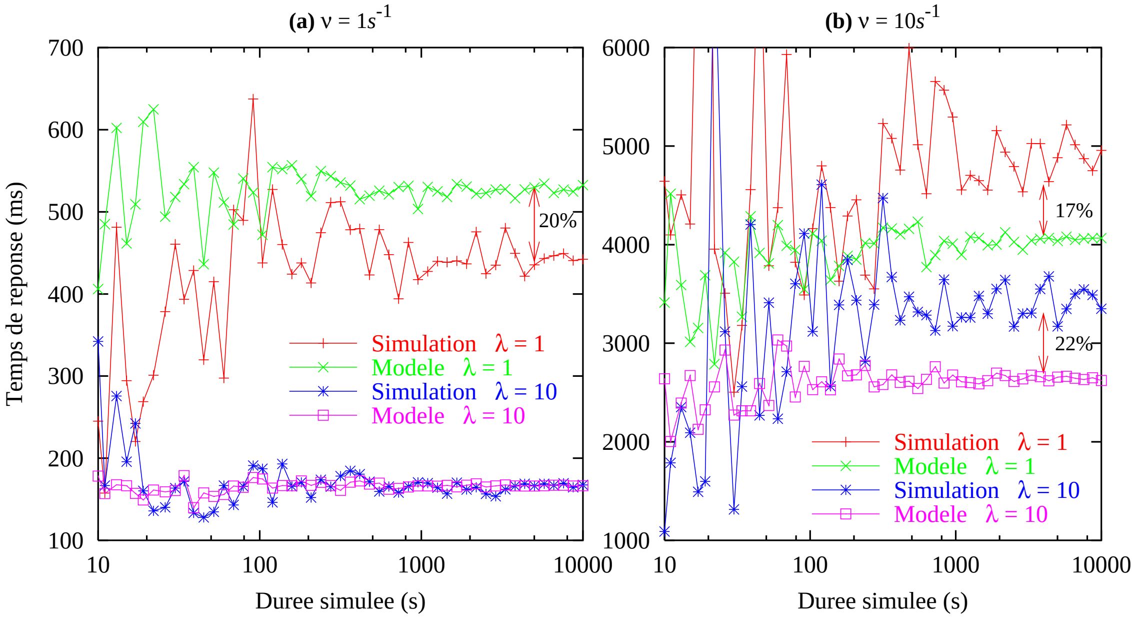


FIG. C.4 – *Convergence du simulateur et du modèle du serveur dans le cadre d'un MAN* ($\delta_S = 1.1$, $\gamma_1 = 36.7$, $\gamma_2 = 12.1$,$\mu = 938$) *avec des distributions réalistes*

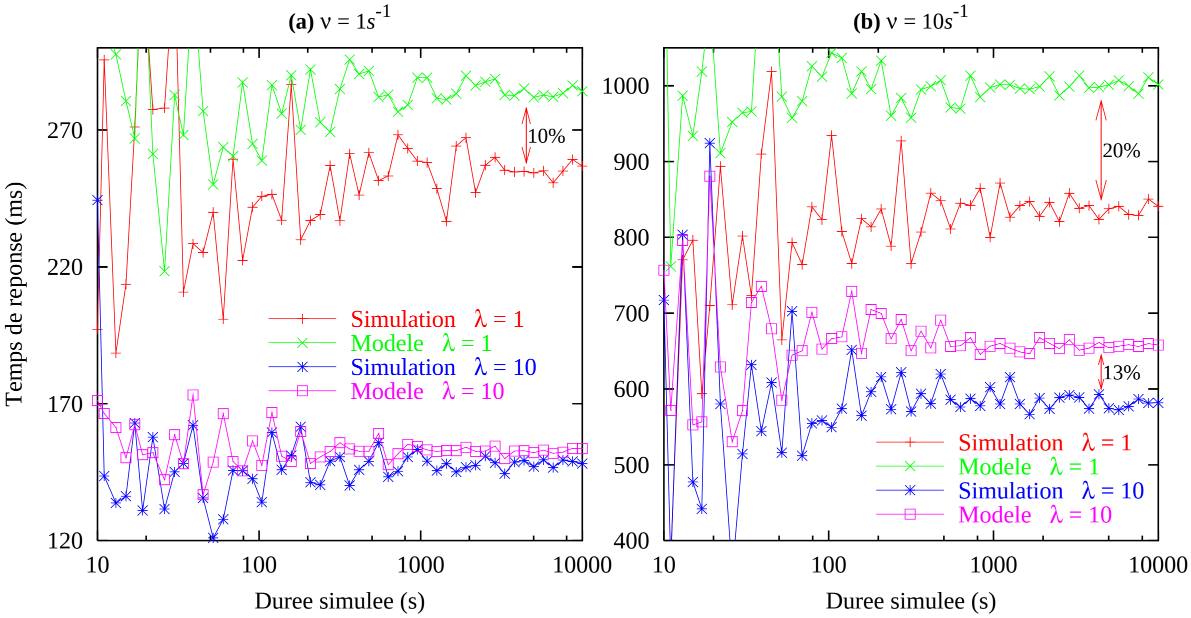


FIG. C.5 – *Convergence du simulateur et du modèle des répéteurs dans le cadre d'un MAN* ($\delta_F = 2.87$, $\gamma = 12.3$) *avec des distributions réalistes*

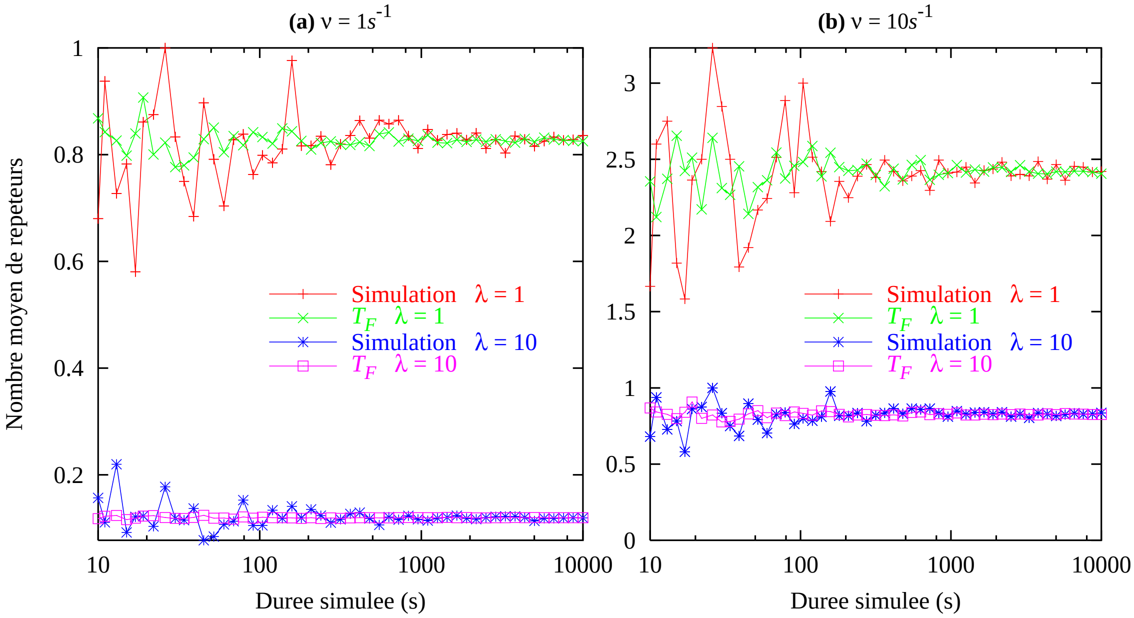


FIG. C.6 – *Convergence du simulateur et du modèle du nombre de répéteurs dans le cadre d'un MAN ($\delta_F = 2.87$, $\gamma = 12.3$)*

C.2 Simulations dans le cas multi-queues

C.2.1 Serveur sur un MAN

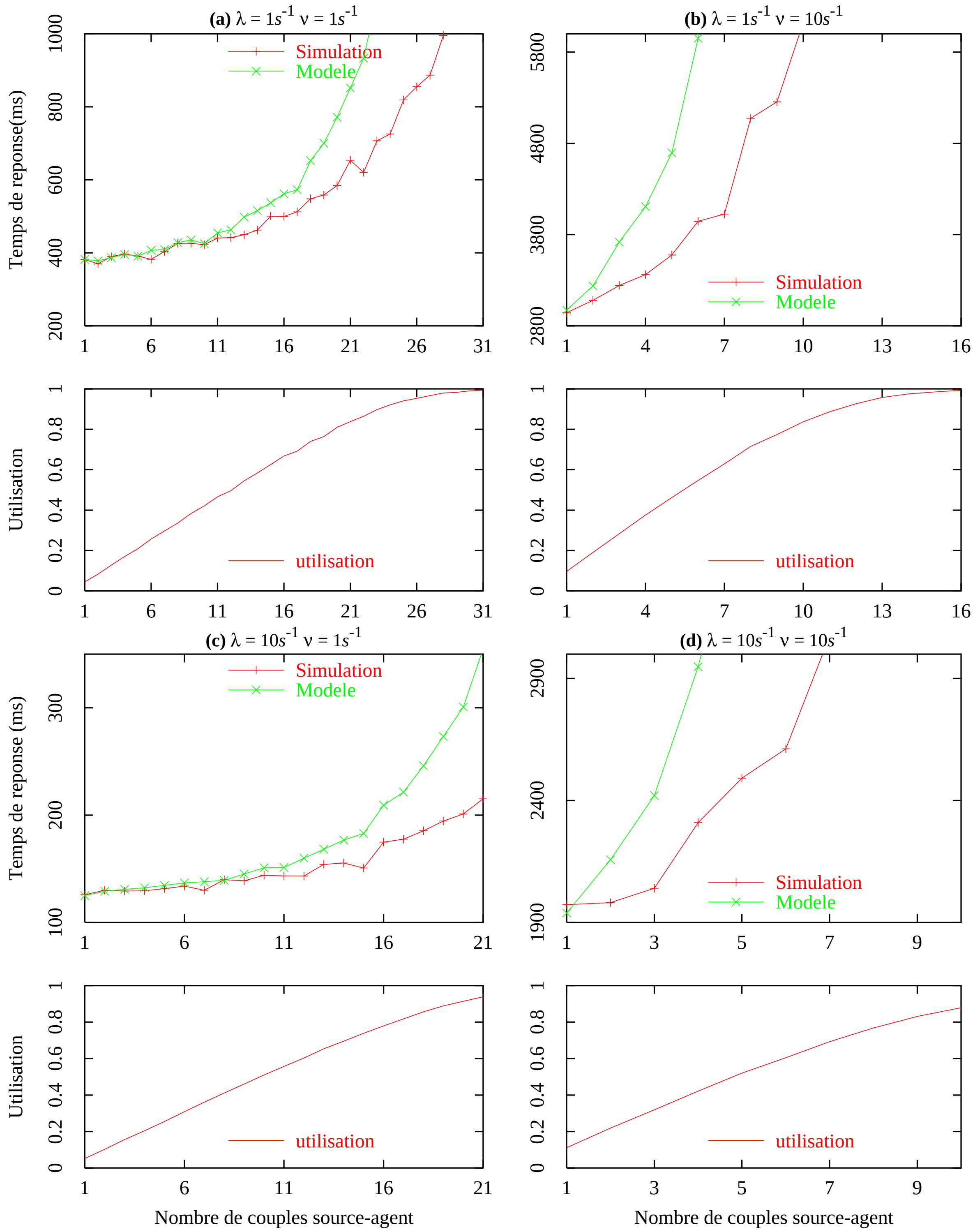


FIG. C.7 – *Validation multi-queues dans le cadre d'un MAN*

Bibliographie

[1] Thin planet. http://www.thinplanet.com.

[2] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[3] Aglets Software Development Kit. IBM, 1999. http://www.trl.ibm.com/aglets/.

[4] S. Alouf, F. Huet, and P. Nain. Forwarders vs. centralized server: An evaluation of two approaches for locating mobile agents. *Proc. of Performance '02, paru dans Performance Evaluation*, 49:299–319, september 2002.

[5] S. Alouf, F. Huet, and P. Nain. Forwarders vs. centralized server: An evaluation of two approaches for locating mobile agents (extended abstract). In *Proc. of ACM SIGMETRICS '02, Marina Del Rey, California, numéro spécial de Performance Evaluation Review*, volume 30(1):278–279, June 2002.

[6] Yeshayahu Artsy and Raphael A. Finkel. Designing a process migration facility: The charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[7] A. Bakker, E. Amade, G. Ballintijn, I. Kuz, P. Verkaik, I. van der Wijk, M. van Steen, and A. S. Tanenbaum. The globe distribution network. In *Proc. of USENIX '00 (FREENIX Track), San Diego, California*, pages 141–152, June 2000.

[8] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Objets actifs mobiles et communicants. *Technique et Science Informatique*, 21(6), 2002.

[9] Françoise Baude, Alexandre Bergel, Denis Caromel, Fabrice Huet, Olivier Nano, and Julien Vayssière. Ic2d: Interactive control and debugging of distribution. In S. Margenov, J. Wasiewski, and P. Yalamov, editors, *Proceedings of the Third International Conference, LSSC 2001*, volume 2179 of *LNCS*, pages 193–200, Sozopol, Bulgaria, June 2001. Springer-Verlag.

[10] Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssière. Communicating mobile active objects in java. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hetrzberger, editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *LNCS*, pages 633–643. Springer, May 2000.

[11] J. Baumann. *Control algorithms for mobile agents*. PhD thesis, University of Stuttgart, IPVR Department, 1999.

[12] Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole - concepts of a mobile agent system. Technical Report TR-1997-15, 1997.

[13] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, 1995.

[14] S. Bouchenak. Pickling threads state in the java system. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal*, 1999.

[15] Jean-Pierre Briot and Rachid Guerraoui. Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. *Technique et Science Informatiques (TSI)*, 15(6):765–800, June 1996.

[16] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive tuple spaces for mobile agent coordination. *Lecture Notes in Computer Science*, 1477, 1998.

[17] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[18] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, November 1998.

[19] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, sep 1993.

[20] W. Chen and C. Leng. A novel mobile agent search algorithm. In *Sixth International Conference on Computer Communications and Networks (ICCCN '97)*. AAAI/MIT Press, 1997.

[21] D. M. Chess, C. G. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM T.J. Watson Research Division, March 1995.

[22] P. Ciancarini and D. Rossi. Jada: a coordination toolkit for Java. Technical Report UBLCS-96-15, 1996.

[23] Paolo Ciancarini. Distributed programming with logic tuple spaces. Technical Report UBLCS-93-7, Piazza di Porta S. Donato,5, 40127 Bologna, Italy, April 1993.

[24] Aramira Corporation. Jumping beans, 1999. http://www.jumpingbeans.com/.

[25] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries (extended abstract). In *Generic Programming. Proceedings*, volume 1766 of *LNCS*, pages 25–39. Springer-Verlag, 2000.

[26] Robert Gray David. Mobile agents: Motivations and state-of-the-art systems. In *Handbook of Agent Technology*, 2001.

[27] Stefan Fünfrocken and Friedemann Mattern. Mobile agents as an architectural concept for internet-based distributed applications - the wasp project approach.

[28] R. J. Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proc. of PODC '86, New York*, pages 108–120. ACM Press, August 1986.

[29] Robert Joseph Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, University of Washington, 1985.

[30] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* GOTOP Information Inc., 1996.

[31] R. Gray, G. Cybenko, D. Kotz, and R. Peterson. and d. rus. d'agents: Applications and performance of a mobile-agent system, 2001.

[32] R. S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proc. of the 4th Annual Tcl/Tk Workshop, Monterey, California*, pages 9–23, July 1996.

[33] Leila Ismail and Daniel Hagimont. A performance evaluation of the mobile agent paradigm. In *Proceedings of OOPSLA'99*, pages 306–313, 1999.

[34] M. ITA. Concordia: An infrastructure for collaborating mobile agents.

[35] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[36] Gregor Kiczales. Beyond the black box: Open implementations. *IEEE Software*, 13(1):8–11, jan 1996.

[37] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97*, volume 1241 of *LNCS*. Springer-Verlag, jun 1997.

[38] L. Kleinrock. *Queueing Systems: Theory*, volume 1. John Wiley and Sons, 1975.

[39] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS '00, Boston, Massachusetts*, pages 190–201, November 2000.

[40] D. R. Cheriton M. M. Theimer, K. A. Lantz. Preemptable remote execution facilities for the v-system. In *Proceedings of the 10th Symposium on Operating System Principles*, 1995.

[41] V. A. Malyshev. An analytical method in the theory of two-dimensional positive random walks. *Mathematicheskii Zhurnal*, 13(6):1314–1329, 1972.

[42] G. Di Marzo, M. Muhugusa, C. F. Tschudin, and J. Harms. The Messenger Paradigm and its Impact on Distributed Systems. In Claus Unger and Ioan Alfred Letia, editors, *ICC'95 International Workshop on Intelligent Computer Communication*, pages 79–94, 1995.

[43] J. McAffer. Meta level programming with CodA. In *Proceedings of ECOOP'95, Aarhus, Denmark*, LNCS 952, pages 190–214. Springer-Verlag, Berlin, 1995.

[44] D. S. Milojičić, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *Proc. of USENIX COOTS '98, Santa Fe, New Mexico*, April 1998.

[45] P. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *Proc. of SIGCOMM '88, Stanford, California*, pages 123–133. ACM Press, August 1988.

[46] Luc Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *The 17th ACM Symposium on Applied Computing (SAC'2002)*, 2002.

[47] Amy L. Murphy and Gian Pietro Picco. Reliable communication for highly mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[48] B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mobile Networks and Applications*, 4(4):245–254, 1999.

[49] J. Norris. *Markov chains*. Cambridge University Press, 1997.

[50] H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. In *First International Workshop on Mobile Agents (MA'97)*, LNCS, pages 50–61. Springer-Verlag, Berlin Germany, 1997.

[51] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proc. of SOSP '83, Bretton Woods, New Hampshire*, October 1983. Published as Operating Systems Review, 17(5):110-119.

[52] ProActive. INRIA, 1999. http://www-sop.inria.fr/oasis/ProActive.

[53] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, USA, 1997.

[54] V. Roth. Scalable and secure global name services for mobile agents. In *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages*, volume 1964 of *LNCS*. Springer Verlag, 2000.

[55] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA*, pages 16–28, 2000.

[56] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *Proceedings of In Coordination'99*.

[57] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, inria, rocquencourt, November 1992.

[58] Chris Steketee, Weiping Zhu, and Philip Moseley. Implementation of process migration in amoeba. In *International Conference on Distributed Computing Systems*, pages 194–201, 1994.

[59] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM '01, San Diego, California*, pages 149–160. ACM Press, August 2001.

[60] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, May 1994.

[61] Sun. Java 2 platform, micro edition, 2001. http://java.sun.com/j2me/.

[62] Sun Microsystems. The Java Virtual Machine Specification, 1998.

[63] Sun Microsystems. Java Object Serialization Specification, 1999.

[64] Sun Microsystems. Java Remote Method Invocation Specification, 1999.

[65] Inc. Sun Microsystems. Rpc: Remote procedure call protocol specification version 2. RFC 1057, Network Working Group, June 1988.

[66] Chantal Taconet and Guy Bernard. A localization service for large scale distributed systems based on microkernel technology. In *Proceedings of the ROSE'94*, 1994.

[67] H. Takagi. *Queueing Analysis: A Foundation of Performance Evaluation*, volume 1. North-Holland, 1991.

[68] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA*, pages 29–43, 2000.

[69] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, 1998.

[70] Voyager. ObjectSpace, Inc., 1999. http://www.objectspace.com.

[71] Waterloo Maple Software. *Maple V*. Waterloo Maple Software, Waterloo, Ontario, Canada, 2001.

[72] Pawel Wojciechowski. Algorithms for location-independent communication between mobile agents. In *AISB '01 Symposium on Software Mobility and Adaptive Behaviour*, 2001.

[73] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[74] Wolfram Research, Inc., 100 Trade Center Drive, Champaign, IL 61820-7237, USA. *Mathematica a system for doing Mathematics by computer*, version 2.2 edition, 2001.

Résumé

Cette thèse a pour sujet la mobilité faible des applications et en particulier la communcation entre entités mobiles. Nous nous sommes tout d'abord intéressés aux relations existant entre le paradigme des objets actifs et celui des applications mobiles. De nombreux protocoles pour assurer les communications entre objets mobiles ont été décrits dans la littérature mais leurs performances n'ont jamais été étudiées formellement. Nous avons isolé des propriétés permettant de les classer en trois familles : la poste restante, la recherche et le routage. Après avoir choisi deux protocoles utilisés dans des bibliothèques Java, nous avons entrepris leur étude à l'aide de chaînes de Markov, le but étant de pouvoir déterminer le temps moyen nécessaire pour communiquer avec un agent mobile. Le mécanisme des répéteurs est représenté à l'aide d'une chaîne à espace d'états infini. Nous avons pu exprimer deux métriques: le temps moyen de réponse du système et le nombre moyen de répéteurs. Le cas du serveur nécessite l'analyse d'une chaîne à espace d'états fini qui est résolue numériquement. Pour valider nos modèles nous avons utilisé un simulateur à événements discrets puis, nous avons mené des expérimentations sur un réseau local et sur un réseau régional. Les résultats ont été comparés à ceux obtenus théoriquement ce qui nous a permis de montrer que les performances de nos modèles sont tout à fait acceptables. Il est donc possible de les utiliser pour prédire les performances. Enfin, nous terminons ce travail par la présentation d'un nouveau protocole de communications utilisant des répéteurs à durée de vie limitée et un serveur. Nous montrons qu'il est possible d'obtenir de bonnes performances sans les aspects négatifs des deux protocoles précédents.

Mots-clés — Code mobile, migration, répéteurs, serveur de localisation, évaluation de performances, chaînes de Markov, simulations, expérimentations.

Abstract

This thesis deals with weak mobility of applications and communication between mobile objects. We have first looked at the relations between active objects and mobile applications paradigms. Many protocols have been proposed to ensure reliable communications between mobile entities but their performance has never been studied. By carefully studying them, we were able to devise a classification of the existing protocols into three families : mailbox, search and routing. Then, we have studied two widely used protocols using Markov chains in order to estimate the average time taken to send a message to a mobile object. The forwarding mechanism is represented by an infinite state-space chain. Using z-transform, we were able to express two metrics : the average response time and the average number of forwarders. In the case of the centralized server, a finite state-space chain was created and solved numerically. In order to validate our models, we have used a discrete event simulator and then conducted intensive experiments on a LAN and a MAN. The results were shown to be close to the theoretical one, allowing us to use the models to predict the performance of the protocols. Finally, the end of this thesis is devoted to the study of a new protocol based on a hybrid approach using forwarders with limited lifetime and a server. We have shown that it is possible to obtain a good performance while avoiding the major drawbacks of the previously studied mechanism.

Keywords — Mobile code, migration, forwarders, location server, performance evaluation, Markov chains, simulations, experiments.