Changement dynamique de modèle de communication dans une plate-forme logicielle pour composants

Victor Budau, Guy Bernard

Institut National des Télécommunications
9 rue Charles Fourier
F-91011 Evry Cedex
{Victor.Budau, Guy.Bernard}@int-evry.fr

RÉSUMÉ. La programmation des systèmes répartis étant complexe, de nouveaux modèles de programmation ont été mis en place pour la conception des applications. Les modèles d'invocation des méthodes distantes et ceux basés sur des événements sont les modèles de programmation majeurs utilisés aussi par les techniques composants. Ils s'appuient, pour la communication entre les différents composants, sur deux principaux modèles de communication : synchrone et asynchrone. Bien que les modèles de programmation et de communication soient orthogonaux, les plates-formes composants actuelles font l'association "modèle de programmation - modèle de communication" implicite : les invocations distantes sont synchrones et les applications programmées dans un paradigme basé sur des événements communiquent d'une manière asynchrone. Dans cet article, nous proposons un mécanisme de changement dynamique et transparent de modèle de communication entre les composants d'un système logiciel réparti. Ce mécanisme, visant surtout les applications à grande échelle et le systèmes mobiles, offre la possibilité d'adapter la communication aux conditions d'exécution des applications et à celles de l'environnement de communication.

MOTS-CLÉS : Communication par messages, RMI, Composants logiciels, Service d'événements

1. Introduction

La programmation des systèmes qui s'exécutent et communiquent à travers l'Internet devenant très complexe, de nouveaux *modèles de programmation* ont été mis en place pour faciliter la compréhension, la conception et l'implantation des systèmes répartis. Parmi ces modèles, les appels de procédures distantes et les structures basées sur la génération et la notification d'événements se sont imposés comme des solutions viables pour la programmation distribuée. L'apparition des plates-formes middleware a beaucoup facilité le dialogue entre les systèmes hétérogènes, et les modèles de composants logiciels aident à gérer la complexité, l'hétérogénéité et le besoin d'adaptabilité et reconfiguration de ces systèmes. Aussi bien les invocations de méthodes distantes que la communication à base de messages ont tiré profit de ce développement, qui les rend plus fiables et plus faciles à utiliser.

Les plates-formes logicielles pour les composants font une association implicite, néanmoins naturelle, entre ces modèles de programmation et les *modèles de communication* (synchrone et asynchrone) qu'elles supportent : les invocations distantes sont synchrones (disponibilité simultanée des entités communicantes) et les applications programmées dans un paradigme basé sur des événements communiquent d'une manière asynchrone (avec des messages). Pourtant, un croisement hybride entre les deux types de modèles peut être utile dans certains cas d'utilisation. La combinaison des deux modèles de communication, rarement prise en compte pour des raisons de difficulté de programmation, reste un sujet de recherche ouvert.

Dans cet article, nous proposons un mécanisme de changement dynamique et transparent de modèle de communication entre les constituants d'un système logiciel réparti. Les applications à grande échelle sont fortement exposées aux changements, parfois aléatoires, de leur contexte d'exécution (indisponibilité temporaire des serveurs, coupure des canaux de transmission, reconfigurations architecturales, etc.). L'adaptation de la communication à ces nouvelles situations est souhaitable à cause des gains de performance ou des soucis de fiabilité. De plus, si cette adaptation est effectuée d'une manière transparente, elle réalise une protection élégante du code applicatif face à la reconfiguration de la communication système.

Après une description de l'interaction "modèles de programmation - modèles de communications" dans la section 2, la nature et l'utilisation de cette alternance de type de communication sont présentées dans la section 3. Dans la quatrième section nous décrivons le schéma du commutateur synchrone/asynchrone qui réalise ce changement. À la lumière de cette approche, les limites dictées par les modèles de programmation ou l'aspect composant sont discutés dans la section 5. Nous continuons dans la section 6 avec l'implantation en cours de réalisation sur une plate-forme à base de composants. Les conclusions et la présentation des travaux futurs clôturent cet article.

2. Modèles de programmation et modèles de communication

Un important domaine de recherche pour la diffusion des applications réparties est l'identification du modèle de programmation qui convient le plus aux demandes concrètes des différentes applications. La plupart des styles architecturaux exploitent le paradigme de communication “client-serveur” pour construire les systèmes répartis. Dans cette approche, bien desservie par le modèle de programmation par *invocations distantes*, les interactions entre les différentes composants apparaissent quand une entité demande à une autre d'exécuter une opération à sa place. Dans une approche *basée sur des événements*, les composants du système sont des entités autonomes qui informent "le monde extérieur" des changements de leur état. La notification d'un événement est vue comme un stimulateur externe qui détermine un changement de son état interne. Ainsi, la communication est indirecte.

Dans le premier cas, celui des *invocations distantes*, l'entité appelante (client), avant de demander un service à une autre entité (serveur), doit connaître l'existence d'un tel serveur capable de satisfaire/exécuter sa requête et doit obtenir sa référence (Fig. 1). Bien que le besoin de connaître l'identité du partenaire de dialogue soit retardé jusqu'au moment de l'exécution, on a toujours besoin de cette référence pour invoquer une méthode du serveur. Dans le deuxième cas, celui d'une architecture *basée sur des événements*, ses composants coopèrent par l'envoi et la réception d'événements, un type spécial de messages. L'émetteur du message (la source) transmet un message

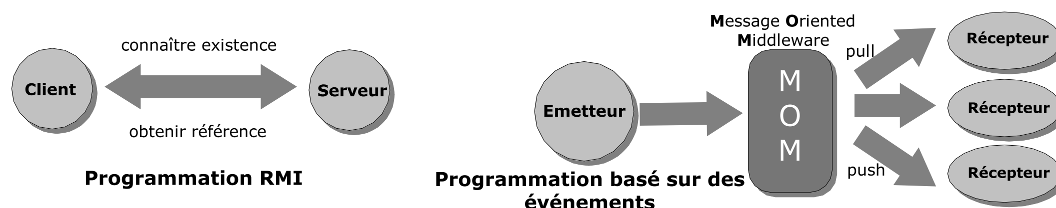

Figure 1. *Modèles de programmation*

à un aiguilleur de message qui se charge de le distribuer à tous les composants ayant déclaré leur intérêt dans la réception de ce message. La source d'une communication n'est pas obligée de spécifier la destination de ses messages et le destinataire ne connaît pas nécessairement l'origine du message. Ce type d'infrastructure s'appelle souvent MOM (Message Oriented Middleware).

Il est habituel d'associer, au modèle de programmation choisi pour coder une application, le modèle de communication qui s'est imposé comme le type privilégié de communication pour ce modèle de programmation. Ainsi, lorsqu'on pratique une programmation dans un paradigme d'invocation distante, on adopte généralement comme type de communication l'*invocation synchrone* (RPC, RMI), tandis que le modèle de programmation basé sur des événements dispose nativement d'une *communication asynchrone*, avec des messages. Le choix d'un type de communication ou d'un autre est fait dès la conception de l'application par le choix même de l'infrastructure de communication. Dans le premier cas (invocations de méthodes distantes + modèle de communication synchrone) la communication est réalisée avec un mécanisme *stub/skeleton* qui se charge du transfert de données. En utilisant une communication synchrone qui bloque le client jusqu'au retour du résultat, nous obtenons une sémantique de l'invocation distante similaire à celle d'une invocation locale. Cette ressemblance sémantique et sa simplicité d'utilisation font de l'invocation de procédures distantes le modèle de communication privilégié dans les systèmes répartis. On le rencontre dans des infrastructures middleware comme CORBA [OMG 97], JavaRMI [Sun 98].

Dans l'autre cas, la communication basée sur des événements s'appuie sur une infrastructure MOM qui fournit des facilités pour la création, l'envoi, la réception et la lecture des messages. Pour communiquer, un composant crée et envoi un message à une tierce entité qui le stocke dans une file d'attente. Tous les autres composants qui ont besoin de ce message, soit contactent cet agent intermédiaire pour obtenir le message (*pull*), soit s'inscrivent auprès du même agent pour se faire livrer lorsque le message attendu est arrivé dans la file d'attente (*push*).

En analysant cette synergie entre les deux modèles de programmation et communication, on peut se demander dans quelle mesure l'adoption d'un modèle de programmation influence-t-elle le choix du modèle de communication. Est-ce qu'on ne pourrait pas utiliser des messages pour communiquer avec un serveur qui affiche une interface et attend des invocations ? Ou bien, ne pourrait-on pas faire en sorte que dans un paradigme de programmation basé sur des événements on puisse contacter ces entités par le biais des invocations directes, synchrones ? Enfin, ce changement de type de communication pourrait-il être réalisé pendant l'exécution du programme (dynamiquement) et non pas être prédéterminé à la programmation/conception ? Avant d'analyser ces possibilités croisées, l'intérêt d'une telle démarche doit être justifié. Ceci est l'objet de la section suivante.

3. Justification d'un régime hybride de communication

Pour mieux comprendre la motivation de changer de modèle de communication pendant la communication elle-même, prenons l'exemple banalisé de la communication téléphonique. Le réseau téléphonique propose deux types de communication : le service téléphonique de base et celui par messages (vocaux ou SMS). Le premier est synchrone car nécessite la présence simultanée de deux interlocuteurs. Si le destinataire de l'appel n'est pas disponible (appareil éteint, par exemple) la connexion ne pourra pas être établie et le système propose à l'entité appelante de laisser un message sur un support stable (boîte vocale du protagoniste absent). Celui-ci pourra le lire plus tard et éventuellement répondre. Ainsi entre les deux utilisateurs une communication asynchrone est établie en cas d'échec de la communication synchrone. Le choix d'une communication basée sur des messages permet d'éviter le rappel systématique par une stratégie de polling continu.

La transposition de cet exemple dans le domaine des applications réparties consiste à fournir le choix dynamique du type de communication : *invocation* ou *message*. C'est le choix d'avoir à sa disposition des moyens de communications *synchrones*, respectivement *asynchrones*. L'utilisation de l'un ou l'autre de ces moyens, ou des deux alternativement, reste une décision du système en fonction des opportunités concrètes de communication.

Dans un modèle de programmation *basé sur des invocations*, lorsqu'un client invoque une méthode d'un serveur et que celui-ci n'est pas disponible immédiatement (panne, surcharge, etc.), il est intéressant d'emballer son invocation sous la forme d'un message et de la stocker dans une file d'attente propre au serveur. Lorsqu'il est prêt à effectuer de nouveaux traitements, le serveur contacte cette file (étant ou non averti de l'existence d'un message en attente) et récupère ainsi l'invocation (empaquetée) du client. Le résultat est renvoyé par le même moyen.

La communication par message étant pénalisée par le temps de transfert entre les deux composants (il y a toujours un tiers interposé entre eux), le scénario situé à l'autre extrême pourrait se montrer intéressant : deux applications développées dans un modèle de programmation *basé sur des événements* peuvent essayer d'établir un dialogue direct, par des invocations distantes, et seulement dans le cas où cette tentative échoue seront elles amenées à s'envoyer des messages comme prévu. Les applications mobiles peuvent trouver une opportunité de passer d'une communication asynchrone à une communication synchrone lors du basculement du mode déconnecté au mode connecté. Si dans le mode déconnecté l'utilisation du serveur de messages permet de continuer le travail en local (en envoyant les messages dans une file d'attente de ce serveur), ce dernier devient un *extra hop* quand les messages peuvent être envoyés directement aux destinataires accessibles. Un service de monitoring détectant le changement de mode pourra faire changer le modèle de communication. Les gains de performance dans ce cas restent liés à l'élimination du surcoût du serveur de messages.

Le mécanisme de changement du type de communication peut bénéficier aux applications qui sont confrontées aux situations changeantes de leur contexte d'exécution (nouvelles exigences des utilisateurs, différentes contraintes à respecter) ou de leur environnement de communication (déconnexions des nœuds, problèmes fréquents de communication dans les applications à grande échelle). Dans le cas de la réplication des composants logiciels avec état, l'utilisation des deux modèles de communication peut servir à une plus judicieuse utilisation de ressources pour assurer la cohérence de ces copies. Une cohérence stricte (*eager replication*) maintient un état identique pour tous les répliques d'un tel composant. À la fin d'une transaction qui a changé l'état d'un composant, ce type de réplication met à jour toutes les copies dans le cadre de cette même transaction. La communication synchrone, lorsqu'elle est achevée avec succès, garantit la disponibilité des tous les nœuds participants. De ce fait, son utilisation est bien adaptée à ce cas. Un nœud en panne est rapidement détecté et la transaction annulée. La durée augmentée de la transaction à cause des mises à jour reste prohibitive pour la majorité des applications distribuées. De plus, dans les applications impliquant des nœuds mobiles, où il est difficile d'avoir tous les nœuds connectés en même temps, ce type de cohérence est inutilisable. Ces systèmes demandent une cohérence allégée (*lazy replication*) dans laquelle les mises à jour sont propagées d'une manière asynchrone après la validation de la transaction. Le délai de transmission n'étant pas crucial dans ce cas, et la communication par messages ne demandant pas une disponibilité immédiate des destinataires, ce type de communication est bien adapté à la tâche de mise à jour.

L'utilisation d'un commutateur dynamique de communication synchrone/asynchrone rend la programmation d'une infrastructure qui devrait assurer une cohérence stricte (de l'état des différents composants) identique à celle d'une infrastructure qui pourrait se contenter d'une cohérence plus faible. Une seule interface de programmation est nécessaire, le choix de la politique de réplication se métamorphosant en des options de configuration de ce commutateur. Cette configuration est faite par le déployeur de l'application (qui peut être aussi un outil automatique basé sur un service de monitoring) offrant ainsi une transparence vis-à-vis du programmeur de l'application.

4. Le commutateur dynamique Synchrone/Asynchrone (S/A)

Afin de permettre le changement transparent de type de communication entre deux ou plusieurs systèmes communicants, un modèle d'intercepteur doit être mis en place. Sur chaque nœud dialoguant avec d'autres nœuds il

existe un élément qui intercepte les messages ou les invocations à la fois entrantes et sortantes. Dans la Figure 2 on observe le positionnement de ces "taupes" vis-à-vis du sens de dialogue pour le cas simple de deux nœuds. Le rôle de ces commutateurs est de changer le type de communication. Supposons qu'un composant essaie de

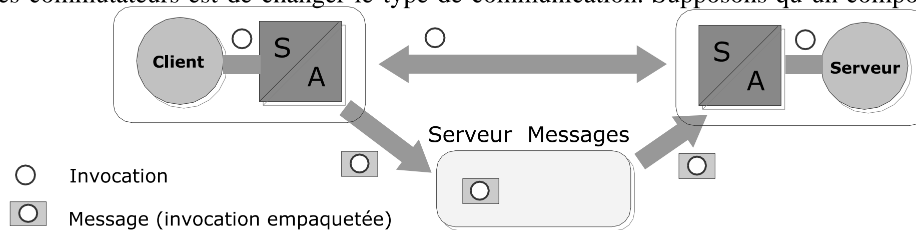

Figure 2. *Les commutateurs S/A dans la chaîne de communication*

communiquer avec un autre et effectue pour cela une invocation. Dans ce cas, le commutateur laisse passer celle-ci sans la modifier. Si la connexion avec le composant distant n'est pas possible, l'infrastructure sous-jacente va retourner un message d'exception en indiquant une erreur de contact. Mais ce message va être capté par le commutateur local et ne va le livrer pas à l'émetteur. À ce moment, l'intercepteur peut jouer son rôle de commutateur Synchrone/Asynchrone (S/A) et initier une communication par messages avec le serveur distant. Il emballe l'invocation originale dans un message qu'il poste dans un serveur de messages. Quand le serveur est à nouveau disponible, le message lui est remis, sauf que l'intercepteur de l'autre côté va capter le message avant qu'il n'arrive au destinataire final (qui d'ailleurs n'attend pas un message mais une invocation !). Maintenant il joue le rôle inverse de déballeur et extrait l'invocation qui est finalement adressée au serveur. Pour le retour des résultats le même schéma peut être utilisé. L'usage des messages pour contourner l'impossibilité d'établir un dialogue direct est dans ce cas totalement transparent aux deux entités communicantes. Le commutateur est aussi utilisé dans le scénario inverse quand deux composants logiciels programmés dans un paradigme basé sur des événements communiquent via des messages. Dans ce cas, en interceptant les messages échangés, le commutateur S/A essaie d'établir une communication directe synchrone pour éliminer le surcoût introduit par le MOM.

5. Synchronisme et Asynchronisme dans la programmation

Dans cette section, nous faisons une analyse des implications liées au choix dynamique du modèle de communication sur le modèle de programmation et donc sur la façon dont une application est écrite. Selon le modèle de communication utilisé et le modèle de programmation on distingue quatre situations, correspondant aux quatre combinaisons possible des deux valeurs de chaque modèle (Fig. 3).

1) **programmation par invocations + communication synchrone**

2) **programmation par événements + communication asynchrone**

La première combinaison représente le mariage du modèle de programmation de type RMI avec la communication par invocations distantes. L'autre montre un modèle de programmation basé sur des événements et s'appuyant pour la communication sur un service de messages. Ce sont les deux situations les plus répandues et qui constituent toutes les deux des vrais "standards" de travail dans les systèmes répartis. Notre discussion n'est pas centrée sur ces deux points mais sur les combinaisons exploratoires no. 3 et 4.

3) **programmation par événements + communication synchrone**

Ce cas offre la possibilité à une application programmée dans un paradigme basé sur des événements de bénéficier de la rapidité relative de la communication avec des requêtes RPC par rapport à la communication avec des messages. Les fonctionnalités usuelles du service de message garantissant la livraison fiable des messages ou le respect de l'ordre de cette livraison obligent à faire de sauvegardes régulières sur un support stable qui ralentissent le transfert de messages. L'intérêt de passer par les invocations directes est d'éviter l'*overhead* impliqué par l'usage de ce service. Le surcoût induit à la communication par les commutateurs S/A est minime car ils se trouvent sur les même nœuds que les entités communicantes, et donc les échanges entre celles-ci et les commutateurs se font localement.

4) **programmation par invocations + communication asynchrone**

Déjà évoqué, ce cas combine la facilité de la programmation qui utilise les appels distants avec la flexibilité du dialogue par messages. Le programmeur utilise toujours des interfaces des serveurs pour invoquer leurs méthodes mais la communication est faite par des messages asynchrones. Mais dans ce cas, est-ce qu'on tire assez profit de l'asynchronisme de la communication ? Pourrait-il continuer son exécution une fois l'invocation lancée et faire ainsi que le système gagne en parallélisme d'exécution ? La réponse est que cela dépend du modèle de programmation utilisé et la façon dont les applications sont écrites, comme présenté dans la discussion suivante.

Modèle prog. \ Modèle comm.	Asynchrone	Synchrone
Invocations	4	1
Evénements	2	3

Figure 3. *Associations modèle programmation - modèle communication*

Les langages de programmation proposent généralement des appels bloquants quand il s'agit des invocations distantes. Un appel de ce type ne retourne que lorsque l'invocation a été achevée avec succès ou une erreur est intervenue. Une amélioration est représentée par le type de méthode *oneway* dans le langage IDL CORBA. Sa sémantique "best-effort" ne garantit pas la livraison de l'invocation car la méthode n'attend pas une réponse de confirmation du serveur. Seules les méthodes qui n'attendent pas de résultat, par contre, peuvent être étiquetées avec le label *oneway*. Si la méthode doit retourner un résultat, des nouvelles structures syntaxiques ont été développés pour contourner ce problème, comme les `Futures` [WAL 90], `Promises` [LIS 88], ou les *delegates* de la plate-forme .NET[Mic 01]. Une structure de type `Future` par exemple, retarde le blocage d'une invocation jusqu'à ce que le résultat soit demandé par le client. Lorsque le client invoque une méthode, il reçoit un objet de type `Future` qui tient le rôle de résultat jusqu'au moment ou celui-ci est nécessaire aux futurs calculs du client. À ce moment, le client fait un appel bloquant sur l'objet `Future` pour obtenir sa valeur. Ainsi, le client et le serveur peuvent continuer leur exécution jusqu'au moment où l'application cliente demande la connaissance du résultat.

En plus de modifier la syntaxe des langages employées, pour rendre les appels non-bloquants, la solution courante est d'utiliser un thread pour chaque invocation (.NET, ARMI [FAL 98]). Bien qu'elle soit gérée par le serveur (le standard EJB interdisant, par exemple, les threads dans les beans), la gestion des threads reste lourde et leur synchronisation, souvent une source d'erreurs. Leur utilisation serait éliminée par le commutateur S/A.

Toutes ces solutions trouvées pour rendre les invocations plus performantes ont des "effets secondaires". Si cette optimisation des performances passe par des compilateurs particuliers et des protocoles de sérialisation et emballage/déballage propriétaires, ces solutions ne sont plus compatibles avec les systèmes actuels. Par exemple, les projets sur Java-RMI asynchrone comme Manta [MAA 99] ou NinjaRMI [WEL 99] rendent impossible la communication avec les objets développés avec RMI standard. De l'autre côté, les modifications dans la syntaxe de la programmation [HIR 98], les nouveaux mots clés [FAL 98], ou des nouvelles structures ou attributs éloignent ces solutions des modèles de programmation standardisés en leur conférant un usage limité.

Le plus important aspect du commutateur S/A est que ce changement dynamique de modèle de communication n'altère pas le modèle de programmation choisi pour concevoir l'application. Il utilise simplement d'une manière transparente différents moyens de communication. Par exemple, il combine le JavaRMI synchrone avec l'asynchronisme du JMS ou il emploie différentes capacités - en terme de protocoles de communication, HTTP ou JMS - de la même spécification (SOAP [W3C]). Bien sûr que les systèmes répartis sont plus efficaces lorsqu'on les conçoit avec un modèle de programmation basé sur des événements en tête. Mais la difficulté de ce type de programmation fait que la large majorité du code aujourd'hui est écrit dans le paradigme RMI. Et c'est ici que le commutateur S/A trouve son utilité. Tout en laissant le programmeur utiliser son modèle de programmation "favorite", il choisit le meilleur modèle de communication dans un certain contexte (avec les limitations mentionnées plus haut).

Lorsqu'il est utilisé avec des modèles de programmation basés sur des événements, ce mécanisme a comme but d'optimiser les performances en essayant de contacter directement l'entité distante, éliminant ainsi son surcoût. Quand on l'utilise avec des modèles RMI, il représente une solution élégante et peu consommatrice de ressources pour une communication fiable en éliminant le polling continu du serveur fait par le côté client. Pour une meilleure utilisation de notre mécanisme, des modèles de programmation améliorés, telles que ceux présentés, doivent être employés. Si ces modèles sont "conscients" de l'aspect réparti de l'application, néanmoins ils permettent de gagner en parallélisme d'exécution du côté client. Masquer cette adaptation du modèle de programmation pourrait se faire par l'intermédiaire des objets répartis transparents comme dans ProActive (ObjectWeb) [Obj 02].

6. Implémentation

Nous avons choisi d'implanter le commutateur S/A sur une plate-forme J2EE à base de composant logiciels (JBoss [JBo 01]) qui se sert du JavaRMI pour la communication synchrone et du JMS pour celle asynchrone. Le choix de la plate-forme orientée composants n'est pas fortuit. Le conteneur réalise toutes ses fonctions en interceptant, tout comme le commutateur S/A, les invocations faites sur les méthodes des composants qu'il abrite. Le *pattern* de l'intercepteur rend le conteneur extensible en permettant l'ajout de fonctionnalités. Le commutateur S/A est ajouté avant même le conteneur et intercepte les invocations/messages dans tous les deux sens. Il existe un seul commutateur pour chaque serveur. Afin de permettre la communication asynchrone, les plates-formes com-

posants affichent toutes des services d'événements : Event Service pour CORBA [OMG 97], JMS [Sun 99b](Java Messaging Service) pour J2EE [Sun 99a] et aussi des spécifications pour des composants programmables dans un paradigme basé sur événements : MDB [Sun 01](Message Driven Beans). Cela nous offre la possibilité de tester les différentes combinaisons entre les modèles de programmation et de communication sur une plate-forme concrète pour effectuer les tests de performances nécessaires à la validation des idées présentés.

L'implantation doit résoudre en principe un problème central : la traduction "message-invocation" dans les deux sens. Pour "emballer" une invocation dans un message le mécanisme de sérialisation du Java permet de transformer l'image mémoire de l'objet `Invocation` dans une représentation binaire qui peut être stockée sur un support stable ou envoyée à travers le réseau. Le processus inverse transforme ce code binaire en l'objet original, dans l'espace mémoire de réception. Le code binaire circule entre les deux commutateurs S/A encapsulé dans un objet de type *message* qui est reconnu, pris en charge et délivré par le service de messages.

Un autre problème apparaît lorsqu'on réalise le passage de l'asynchrone au synchrone. Dans les applications programmées avec un modèle basé sur des événements, le service de messagerie est nécessaire parce que la source ne spécifie pas le(s) destinataire(s). A leur tour, ceux-ci ne connaissent pas l'origine des messages qu'ils consument. Comme la communication synchrone nécessite la connaissance du destinataire, contourner le service de messages à l'aide d'une communication directe/synchrone pose un problème d'identification de récepteurs. Pour résoudre ce problème inhérent à l'élimination de l'extra hop, sans pour autant modifier le serveur de messages, nous prévoyons de mettre en place un protocole d'échange de *stubs*. Les messages échangés contiendront dans des champs additionnels les *stubs* ajoutés par ces entités. Après un seul tour d'échanges de messages (via le serveur), l'émetteur connaîtra l'identité des destinataires de ses messages et les destinataires apprendront la source des messages qu'il consomment. Ils auront les moyens, après cette phase de découverte, de communiquer directement.

Les travaux concerneront la gestion des erreurs, des ressources et la prise en compte des problèmes inhérents à l'élimination du service de messages (filtrage des messages, préservation du contexte transactionnel) . Cette implantation va nous permettre d'effectuer des mesures de performance (gain de l'élimination du serveur de messages, vitesse relative de la communication synchrone) nécessaires à la validation des hypothèses et concepts présentés.

7. Conclusions

Dans cet article nous avons introduit un mécanisme de changement dynamique et transparent de modèle de communication entre les composants d'un système réparti. Le choix d'utiliser un type différent de communication, tout en gardant le même modèle de programmation, est intéressant de plusieurs points de vue correspondant aux différentes applications concrètes : gain de performance, adaptation aux changements contextuels du milieu communicant, faible consommation de ressources pour une communication fiable. Les applications mobiles et les applications à grande échelle où la communication joue un rôle capital dans le besoin continu d'assurer les performances, trouvent dans le commutateur Synchrone/Asynchrone un moyen d'améliorer le temps de réponse et la fiabilité à faible coût. Enfin, l'utilisation d'un modèle de communication avancé (tel que fourni par le `Futures`, `Promises`) devrait permettre d'exploiter pleinement le mécanisme que nous proposons.

8. Bibliographie

[FAL 98]FALKNER K. E. K., CODDINGTON PAUL D.AND OUDSHOORN M. J., « Implementing Asynchronous Remote Method Invocation in Java », *Technical Report DHPC-072, University of Adelaide*, , 1998.

[HIR 98]HIRANO S., YASU Y., IRAGASHI H., « Performance Evaluation of Popular Distributed Object Technologies for Java », *In Worldwide Computing and Its Application*, , February 1998.

[JBo 01]JBOSS GROUP, « JBoss - J2EE application server 2.4.3 », *http ://www.jboss.org*, , 2001.

[LIS 88]LISKOV B., SHIRA L., « Promises : Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems », *In Proc. of SIGPLAN'88 Conf. Programming Language Design and Implementation*, , June 1988.

[MAA 99]MAASSEN J., VELDEMA R., VAN NIEUWPOORT R., BAL H. E., « An efficient implementation of Java's Remote Method Invocation », *In Proc. ACM Symposium on Principles and practice of parallel programming*, , May 1999.

[Mic 01]MICROSOFT CORPORATION, « Microsoft Developer Network », *http ://msdn.microsoft.com*, , 2001.

[Obj 02]OBJECTWEB CONSORTIUM, « Projet ProActive », *http ://www.objectweb.org*, , 2002.

[OMG 97]OMG, « Common Object Request Broker : Architecture and specification 2.1 », *OMG Document 7-08*, , 1997.

[Sun 98]SUN MICROSYSTEMS INC, « Java Remote Metod Invocation Specification », *http ://java.sun.com/rmi*, , 1998.

[Sun 99a]SUN MICROSYSTEMS INC, « Java 2 Platform Enterprise Edition », *http ://java.sun.com/j2ee*, , 1999.

[Sun 99b]SUN MICROSYSTEMS INC, « Java Messaging Service Specification », *http ://java.sun.com/jms*, , 1999.

[Sun 01]SUN MICROSYSTEMS INC, « Enterprise Java Beans Specification 2.0 », *http ://java.sun.com/products/ejb*, , 2001.

[W3C]W3C CONSORTIUM, « Simple Object Access Protocol - SOAP 1.1 », *http ://www.w3.org/TR/SOAP*.

[WAL 90]WALKER E. F., FLOYD R., NEVES P., « Asynchronous Remote Operation Execution In Distributed Systems », *In Proc. of the Tenth International Conference on Distributed Computing Systems*, , May/June 1990.

[WEL 99]WELSH M., « Ninja RMI : A Free Java RMI », *http ://www.cs.berkeley.edu/˜mdw/proj/ninja/ninjarmi.html*, , 1999.