Valentin Brémond
Gaylord Cherencey
Florian Massacret

INVERSION OF WEB SERVICE INVOCATION USING PUBLISH/SUBSCRIBE PUSH-BASED ARCHITECTURE

Projet de session – Avril 2013

Développement avancé de Patrons et Modèles – 8INF956 – Hamid Mcheick

Table des matières

INTRODUCTION

Introduction

RAPPEL DE L'ARTICLE

Notre idée de projet provient de l'article « Inversion of Web Service Invocation using Publish/Subscribe Push-Based Architecture », par Thanisa Numnonda et Rattakorn Poonsuph. Cet article explique comment optimiser les services Web basés sur l'architecture Web Service Invocation (WSI) fonctionnant selon le principe de publication / abonnement au travers d'un système « push » pour diminuer la latence entre le moment où un contenu est disponible et le moment où le client le reçoit.

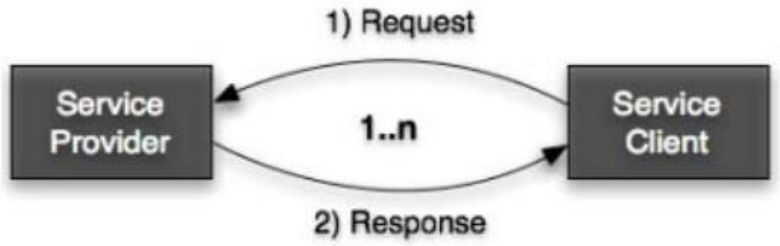

Figure 1 : architecture publisher / subscriber classique

Une architecture publication / abonnement (« publisher / subscriber ») est une architecture dans laquelle un client accède à un contenu proposé par un fournisseur. Pour cela, il existe 2 types de systèmes permettant l'échange d'information :

- Les systèmes dits « pull » (« tirer ») : le client doit régulièrement demander au fournisseur si de nouvelles informations sont disponibles (dans l'affirmative, le fournisseur envoie les informations au client)
- Les systèmes dits « push » (« pousser ») : le client, une fois qu'il a souscrit à un contenu, ne fait plus aucune requête auprès du fournisseur (c'est à ce dernier d'envoyer les informations directement au client)

Afin de limiter les problèmes pouvant survenir au niveau du fournisseur (surcharge), il est mis en place un intermédiaire, appelé « broker ». Le broker a pour objectif de transmettre des notifications du fournisseur au client lorsqu'un nouveau contenu est disponible. Le fournisseur est ainsi moins sollicité (il est uniquement sollicité lors de l'apparition d'un nouveau contenu), permettant aux développeurs d'alléger le code des fournisseurs. Toutefois, le broker doit être développé avec pour

principale caractéristique la robustesse. En effet, un crash du broker entraînerait la fin des communications entre le ou les fournisseur(s) et le ou les client(s).

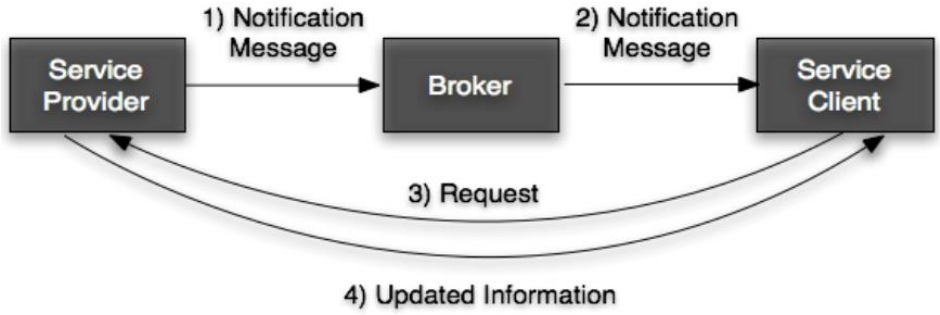

Figure 2 : architecture publisher / subscriber de type pull

Toutefois, cette architecture n'est pas totalement optimisée : en effet, lorsque le fournisseur envoie une notification au broker comme quoi un nouveau contenu est disponible, ce dernier fait suivre cette notification aux clients. Lorsque les clients reçoivent cette notification, ils vont chercher l'information sur le fournisseur. Si de nombreux clients font cette requête au même moment (ce qui est vraisemblablement le cas étant donné que le broker envoie la notification à tous les clients simultanément), on retombe sur des problèmes de goulot d'étranglement et de surcharge.

Les auteurs proposent donc une solution alternative : au lieu d'envoyer des notifications de nouveau contenu au broker puis au client, il serait plus efficace d'envoyer directement l'information au broker, qui la diffusera aux clients. Ainsi, le client n'effectue aucune requête sur le fournisseur, et la seule requête qu'il effectue sur le broker est son abonnement. Ceci a pour effet de grandement limiter les requêtes, de recevoir les informations en temps réel et de supprimer le goulot d'étranglement au niveau du fournisseur.

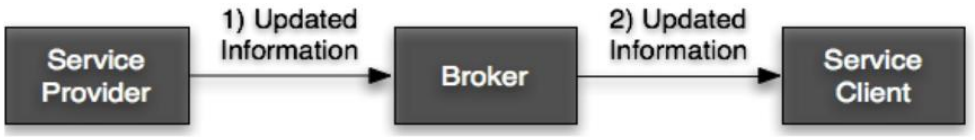

Figure 3 : architecture publisher / subscriber de type push

Afin de permettre au broker de contacter un client et au fournisseur de contacter le broker, il faut inverser le sens des services Web : en effet, en temps normal, le sens de « discussion » va du client vers le broker, puis du broker vers le fournisseur. Il faut donc que le fournisseur propose une interface au broker et que le broker propose une interface au client. Or ici, il faut que le client propose une interface au broker (pour qu'il puisse le contacter à tout moment) et que le broker

propose une interface au fournisseur. Les interfaces sont inversées par rapport à un service Web traditionnel.

Les auteurs étudient cette architecture d'un point de vue théorique et font des considérations de performances qui démontrent que les temps de latence entre le moment où une nouvelle information est publiée et le moment où le client la reçoit sont diminués de plus de 50%, de même que les goulots d'étranglement sont éliminés. Malheureusement, les auteurs ne proposent aucune implémentation concrète.

OBJECTIF DU PROJET

Etant donné l'absence d'implémentation, nous avons décidé de mettre en place notre propre système de publication / abonnement de type push.

Le but du projet était d'implémenter en Java le système qui était présenté dans l'article.

Nous avons donc créé le fournisseur, le broker et le client, puis nous avons mis en place les services Web à l'aide de technologies disponibles par défaut dans Java. L'avantage d'utiliser des services Web étant de permettre à des clients non Java d'utiliser les services offerts par notre implémentation.

Conception du logiciel

TECHNOLOGIES UTILISEES

Nous avons utilisé plusieurs technologies pour mener à bien notre projet.

Patron de conception Observateur

Nous avons utilisé ce patron de conception dans l'interface graphique. En effet, étant donné que nous avons séparé le client « fonctionnel » du client « graphique », il fallait que la partie graphique soit actualisée lors de la réception de nouvelles informations.

Nous avons donc utilisé ce patron, qui est en quelque sorte le pendant « objet » d'une architecture de type « push ».

JAX-WS

Java API for XML Web Services (JAX-WS) est l'implémentation Java EE de référence. Cette implémentation permet de créer très simplement des services Web à partir d'interfaces et d'annotations.

JAX-WS génère des fichiers WSDL (« Web Services Description Language »), qui sont des fichiers XML spéciaux décrivant les services d'un serveur (ses méthodes) ainsi que les types acceptés pour ses paramètres. Il utilise le protocole SOAP (« Simple Object Access Protocol ») afin de permettre à des systèmes hétérogènes de communiquer via un langage commun.

En plus de permettre la génération automatique de fichiers WSDL, JAX-WS permet de récupérer un fichier WSDL puis de le convertir en un objet du même type qu'une interface. Son fonctionnement est donc similaire à RMI et il permet de manipuler des objets distants avec la même simplicité.

AspectJ

AspectJ est une extension de Java permettant de faire de la programmation par aspects. Pour rappel, la programmation par aspects permet de modifier le comportement des classes sans les modifier, en utilisant des classes spéciales appelées « aspects ». Un aspect peut « capturer » des évènements (tel que l'appel d'une méthode, l'instanciation d'un objet, la modification d'un attribut, …) et effectuer des actions avant, pendant ou autour de ces évènements (il est même possible de remplacer les actions que la classe devrait effectuer par des actions définies dans les aspects).

La programmation par aspects permet de séparer les préoccupations transversales des préoccupations métiers. Dans notre application, nous avons principalement utilisé AspectJ pour

logger des informations. Aucune des classes que nous avons écrites n'affichent des informations console, tout se passe dans les aspects.

Ceci pourrait être utilisé, comme l'expliquent les auteurs, pour logger des statistiques et ainsi optimiser le fonctionnement du broker en analysant les comportements des fournisseurs et des clients.

ARCHITECTURE

Nous avons logiquement séparé notre application en 3 parties :

- Un package « fournisseur »
- Un package « broker »
- Un package « client »

De plus, nous avons créé deux interfaces : une sur le broker qui permettra à la fois au fournisseur de fournir du contenu et au client de s'abonner, et une sur le client qui permettra au broker de lui envoyer l'information.

Enfin, les informations envoyées par le fournisseur sont encapsulées dans une classe nommée « Information » qui contient à la fois le type de l'information (cuisine, cinéma, sciences, …) ainsi que l'information associée.

Le package « fournisseur »

Ce package contient simplement le fournisseur. Il est très léger et son fonctionnement l'est tout autant : dans l'état actuel, il se contente de générer des messages séquentiels avec un type aléatoire à chaque fois, puis d'envoyer ces messages au broker.

Il est possible de lancer plusieurs instances du même fournisseur, tout comme il serait possible de lancer plusieurs instances de différents fournisseurs.

Un fournisseur ne possède aucune interface ni ne génère aucun WSDL étant donné qu'il n'est jamais contacté directement.

Le package « broker »

Ce package contient le plus gros de l'implémentation : le broker. Son objectif est de pouvoir faire suivre des messages en provenance de fournisseurs à des clients, en filtrant ces messages en fonction des types auxquels les clients ont demandé à souscrire.

Pour cela, le broker garde une liste de clients connectés et mémorise les types auxquels ces clients ont souscrit. Lorsqu'un message est reçu d'un fournisseur, le broker parcourt la liste des clients abonnés (ou connectés) puis, pour chacun d'eux, vérifie s'il a demandé à recevoir ce type. Dans l'affirmative, le message est envoyé au client, sinon le broker passe au client suivant.

Afin que l'architecture globale du système ne soit pas compromise, il faut que le broker soit le plus résilient possible. Nous n'avons pas particulièrement pris en compte ce critère dans la réalisation de notre implémentation ; toutefois, dans une implémentation réelle, il faudrait considérer cette caractéristique.

Le broker possède une interface définissant les méthodes accessibles par les fournisseurs et une autre définissant les méthodes accessibles par les clients. Afin de simplifier le code, mais surtout de permettre à un fournisseur d'être aussi client, nous avons regroupé ces deux interfaces en une seule.

Le broker est une sorte de middleware dans notre architecture ; c'est un connecteur entre le fournisseur et le client.

Le package « client »

Ce package contient une ébauche de client graphique permettant de souscrire à un service et de recevoir les informations.

Le client possède une interface qui permettra au broker de lui envoyer les informations.

Grâce à JAX-WS et aux protocoles de services Web utilisés (SOAP avec WSDL), il est tout à fait possible de développer des clients dans d'autres langages que Java. Toutefois, pour des raisons de praticité, nous avons développé un client graphique Swing avec Java.

Nous avons également implémenté les méthodes nécessaires au client pour que la fermeture de la fenêtre entraîne le désabonnement automatique du broker.

Les interfaces

Comme expliqué précédemment, le broker possède une interface permettant aux fournisseurs de lui envoyer des informations et aux clients de se connecter. Cette interface possède 5 méthodes :

- `boolean ajouterTypeInformation (String type)` : permet au fournisseur d'envoyer les types disponibles au broker (un par un). Cette méthode peut être appelée n'importe quand pour rajouter des types en cours d'exécution.
- `boolean envoyerInformation (Information info)` : permet au fournisseur d'envoyer une information au broker.

- `String[] recupererTypesInformations ()` : permet à un client de récupérer la liste des types disponibles sur le broker.
- `boolean sAbonner (String[] listeTypesInformations)` : permet à un client de s'abonner à un certain nombre de types (le client peut choisir les types d'information qu'il va reçevoir).
- `boolean seDesabonner ()` : permet à un client de se désabonner du broker.

Le client quant à lui possède une interface comprenant une seule méthode : `boolean envoyerInformation (Information info)`. Cette méthode permet à un broker de lui envoyer une information.

Toutes les méthodes, à l'exception de `recupererTypesInformations ()`, renvoient un booléen : il vaut « vrai » si l'opération a été effectuée avec succès, « faux » sinon.

Fonctionnement

Le fonctionnement de l'application se fait en 3 grandes étapes :

- Démarrage du broker
- Connexion du (des) fournisseur(s)
- Connexion du (des) client(s)

La première étape, fondamentale, est le démarrage du broker. En effet, ce dernier étant le point central de l'architecture, aucun fournisseur ne peut envoyer d'information et aucun client ne peut souscrire à des services sans la présence du broker.

Les deux étapes suivantes, qui peuvent être effectuées dans le désordre, sont explicitées dans les prochains paragraphes.

CONNEXION D'UN FOURNISSEUR

Lors de la connexion d'un fournisseur, ce dernier effectue deux actions : tout d'abord, il va envoyer au broker la liste des types qu'il propose (cuisine, cinéma, sciences, …). Une fois qu'il aura notifié au broker son contenu, il pourra commencer à lui envoyer les informations. Dans notre implémentation, une nouvelle information aléatoire est envoyée chaque seconde. Chaque information est encapsulée dans un objet « Information », comprenant le type de l'information et l'information en elle-même.

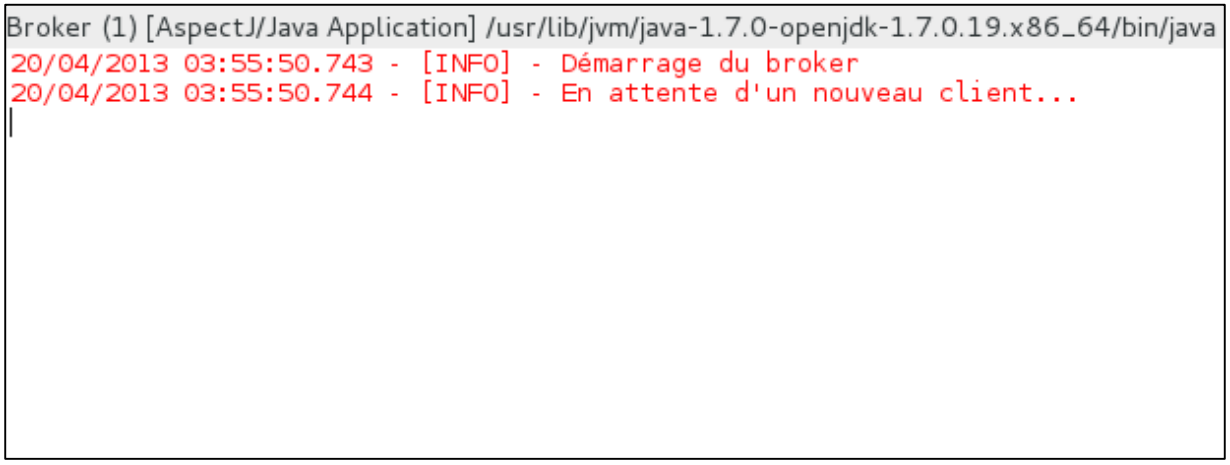

Figure 4 : broker démarré, aucune connexion

FONCTIONNEMENT

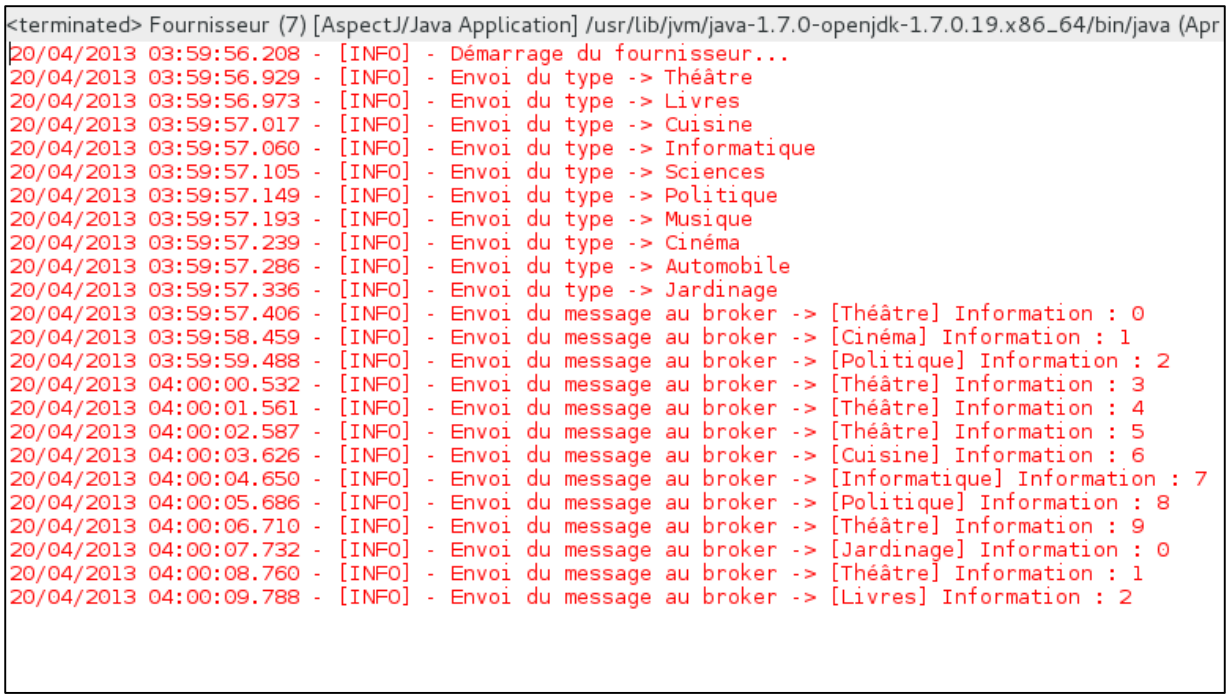

Figure 5 : fournisseur démarré et connecté au broker

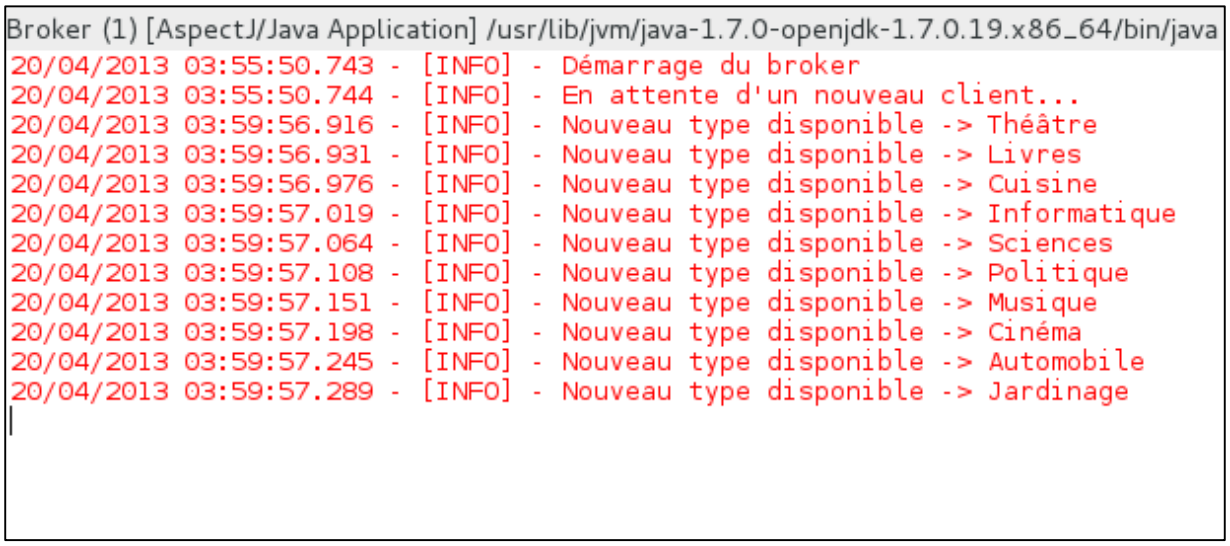

Figure 6 : broker après la connexion d'un fournisseur

Si un fournisseur se connecte à un broker et envoie, parmi sa liste des types, un type que le broker possède déjà, la méthode `ajouterTypeInformation ()` retournera « faux » (ce qui résultera par un « warning » dans les logs).

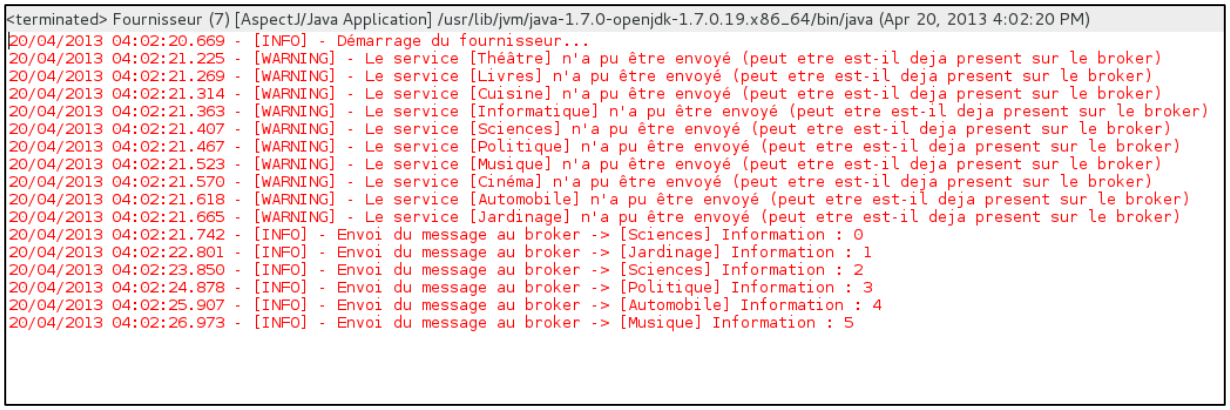

Figure 7 : fournisseur qui envoie des types déjà connus par le broker

CONNEXION D'UN CLIENT

La première étape à effectuer par un client lors de sa connexion est la récupération des types disponibles sur le broker. Ceci permet à l'utilisateur de choisir les types auxquels il veut souscrire.

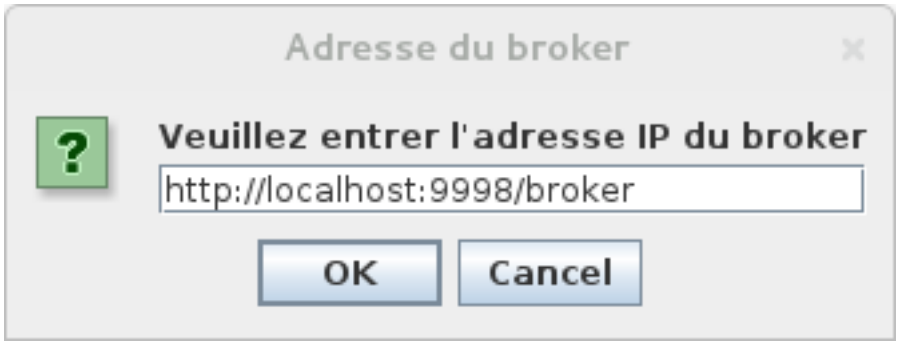

Figure 8 : fenêtre client de connexion au broker

FONCTIONNEMENT

Figure 9 : fenêtre client de choix des types

Une fois les types choisis par l'utilisateur, le client fait une requête d'abonnement sur le broker en précisant les types auxquels il veut s'abonner. Le broker mémorise alors l'adresse IP du client et les types auxquels il a souscrit. A partir de ce moment, le client n'a plus à communiquer avec le broker. Il se contente d'attendre les informations envoyées par le broker.

FONCTIONNEMENT

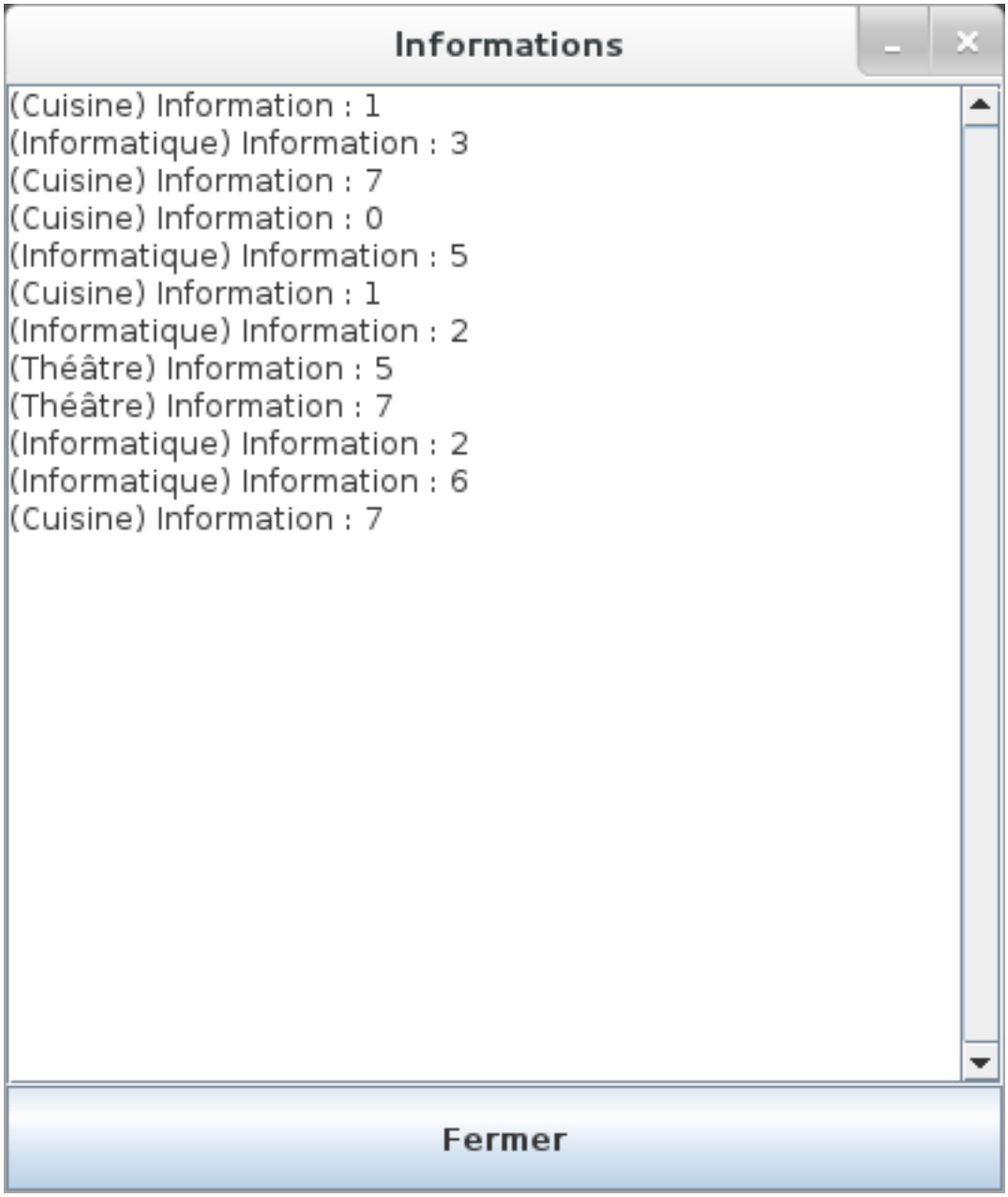

Figure 10 : client graphique connecté au broker

FONCTIONNEMENT

FONCTIONNEMENT DU BROKER

A chaque réception d'une information par un fournisseur, le broker va parcourir la liste des clients connectés et vérifier si le client a souscrit au type de l'information reçue. Si le client a souscrit à ce type, le broker va aller récupérer le WSDL de ce client sur l'adresse IP stockée, appeler la méthode `envoyerInformation ()` du client puis passer au client suivant. Si le client n'a pas souscrit à ce type, le broker passe directement au client suivant.

Figure 11 : broker après la connexion d'un fournisseur et d'un client

TOLERANCE AUX PANNES

Du fait de l'architecture du système, une panne d'un client ou d'un fournisseur n'affecterait pas le broker, et donc le système global. En revanche, en cas de panne du broker, ni le fournisseur ni le client ne pourraient utiliser le système.

Afin de sécuriser le système, il serait judicieux de renforcer le broker ; voire mieux : de rajouter une couche de cloud par-dessus le broker (ou de virtualisation). En effet, en mettant en place un système de load-balancing autour du broker (ou de redondance), on pourrait avoir un système extrêmement résilient sans en modifier l'architecture. De plus, étant donné que les fournisseurs peuvent être « n'importe qui » (et donc ne pas nous appartenir – de même que les clients), la seule partie du système qu'on peut renforcer est le broker.

FONCTIONNEMENT

Le broker faisant office de middleware, c'est la première chose à renforcer et / ou à dédoubler afin de garantir une haute disponibilité du service.

Conclusion

En se basant sur l'article précédemment cité, nous avons voulu proposer une implémentation de l'architecture publisher / suscriber de type push, l'objectif étant de vérifier les arguments avancés par les auteurs. Nous avons donc réalisé une implémentation fonctionnelle de cette architecture, celle-ci passant par trois packages : le fournisseur, le broker et le client.

Comme il était annoncé, le transfert des messages est grandement amélioré. La latence est en effet supprimée, chaque message étant envoyé dès sa création. Cet envoi immédiat permet également une réduction significative du nombre de demandes, le client n'ayant plus à faire de requêtes inutiles.

Ce projet nous a également permis d'appliquer plusieurs technologies vues en cours telles le patron de conception Observateur et l'AspectJ, et d'autres non vues en cours telle que JAX-WS. On a apprécié la facilité d'utilisation de JAX-WS passant, notamment, par une documentation très complète ; on regrettera que cette documentation n'ait pas été de la même qualité pour AspectJ.

En termes de perspective pour ce projet, il reste toujours un point d'ombre concernant la redondance des brokers afin de permettre une disponibilité accrue des services. On pourrait ainsi imaginer une sorte de cloud de brokers pouvant être démarrés dynamiquement si un broker tombe en panne.

L'autre point qui n'est pas adressé concerne la sécurité de l'architecture puisque, pour le moment, il est possible de se connecter librement au broker et de lui envoyer des informations. On peut alors facilement imaginer un utilisateur malveillant qui enverrait de fausses informations via le broker. Afin de sécuriser l'architecture, on pourrait imaginer un système d'authentification pour se connecter au broker (qui pourrait passer par un certificat pour les fournisseurs permettant de valider leur légitimité).

Références

Inversion of Web Service Invocation using Publish/Subscribe Push-Based Architecture, Rattakorn Punsuph, Thanisa Numnonda

Inversion of Web Service Invocation using Publish/Subscribe Push-Based Architecture, Thanisa Numnonda

http://skebir.developpez.com/tutoriels/java/aspectj/ : programmation orientée aspect en Java avec AspectJ (developpez.com)

http://www.eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html : pointcuts en AspectJ (eclipse.org)

https://jax-ws.java.net/ : JAX-WS, tutorial sur la technologie (java.net)

http://www.mkyong.com/tutorials/jax-ws-tutorials/ : tutorial JAX-WS (mkyong.com).

http://blog.espenberntsen.net/2010/03/18/aspectj-examples-with-pointcuts-based-on-annotations/ : tutorial AspectJ sur les pointcuts avec annotations (espenberntsen.net)

http://objis.developpez.com/tutoriels/aspectj/log/ : tutoriel AspectJ, création aspect log (developpez.com)

http://thinkitdifferently.wordpress.com/2009/02/12/aopwithaspectj/ : logging avec AspectJ (Wordpress thinkitdifferently)

http://www.morgan-design.com/2011/08/aspectj-and-custom-logging-annotation.html : tutoriel logging (morgan-design.com)

http://msdn.microsoft.com/en-us/library/ms996486.aspx : comprendre WSDL (microsoft.com)