Messagerie asynchrone et Services Web

SOAP, WSDL SONT DES STANDARDS EMERGEANT DES SERVICES WEB, LES IMPLEMENTATIONS DE CEUX-CI SONT ENCORE EN COURS D'ELABORATION. LES INFRASTRUCTURES BASES SUR CES NOUVELLES TECHNOLOGIES SUPPORTENT 2 TYPES DE MISE EN ŒUVRE : SYNCHRONE ET ASYNCHRONE

L'ASPECT ASYNCHRONE A ETE JUSQU'ICI PEU MIS EN VALEUR, IL DEVRAIT CEPENDANT ETRE AUSSI PRESENT QUE L'ASPECT SYNCHRONE CAR IL APPORTE DES ATOUTS MAJEURS NOTAMMENT EN TERME DE FIABILITE

DANS CET ARTICLE NOUS PRESENTERONS CE QUE SONT LES SERVICES WEB ASYNCHRONES ET PROPOSERONS UNE SOLUTION DE BASE SUR DE LA MESSAGERIE ASYNCHRONE.

Bio

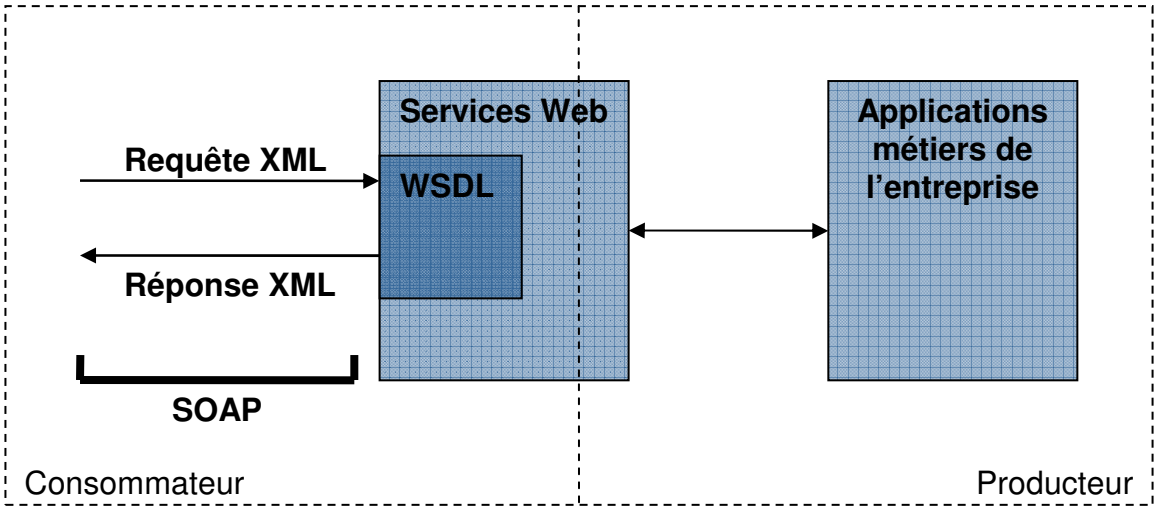
Sébastien Letélié *est ingénieur d'études et de développement au sein de la société Improve, spécialiste en programmation objet depuis 1998, il réalise actuellement un Diplôme de Recherche Technologique sur le thème de l'informatique mobile d'entreprise, il anime des formations sur XML et les Services Web, ainsi que sur l'informatique embarqué dans les universités, écoles d'ingénieurs et formations professionnelles.*

Introduction

Les services Web sont des applications modulaires accessibles de façon standard via les protocoles standard du Web. Le principe est d'utiliser des standards basés sur XML pour échanger de l'information entre systèmes hétérogènes ou exécuter des traitements spécifiques liés à un métier via l'internet.
Les Services Web s'associent à deux spécifications XML:

SOAP définissant le transport de donnée et l'interopérabilité

WSDL pour la description des services (sémantique d'appels, protocoles utilisés, serveurs d'hébergements)



Les enjeux de l'asynchrone

Aucune spécification liée aux services Web n'exprime comment assurer le bon acheminement des messages. Comment maîtriser des problèmes comme des pannes réseaux ou machine dans une architecture de Services Web ? Une des réponses est apportée par le mode asynchrone, qui par définition n'implique pas d'être connecté en permanence, ni de recevoir immédiatement une réponse. Un Service Web asynchrone aura donc l'avantage d'être plus fiable et de permettre la récupération et la centralisation administrative des erreurs. Par ailleurs, dans un environnement englobant plusieurs applications et services interagissant entre eux, le nombre d'interfaces décrivant les informations d'accès aux services se multiplie ; leur maintenance ainsi que leur administration en devient plus complexe. Le mode asynchrone, lui, préconise une architecture lâchement couplée (loosely-coupled), c'est-à-dire que chaque service ou application n'impose pas à l'autre une interface de communication, c'est au message lui-même d'encapsuler les données et le contexte, devenant ainsi une unité autonome.

Contexte général

Quel modèle choisir pour mettre en œuvre un service Web asynchrone ? En effet, jusqu'ici, les spécifications ne préconisent rien en ce qui concerne les aspects asynchrones ; cependant dans sa spécification WSDL définit 2 styles d'invocation des services :

- document (pour échanger un format XML)
- rpc (pour effectuer une commande RPC en bout de chaîne)

ainsi que 4 modes d'opérations :

- one way : la cible reçoit simplement un message

- request/response : la cible reçoit un message et renvoie une réponse
- solicit/response : la cible envoie un message et reçoit une réponse
- notification : la cible envoie un message

A partir de cela Holt Adams définit 4 modèles pour aborder l'aspect asynchrone : nous verrons dans cet article un de ces modèles. Dans tous les cas, un numéro unique est associé au message et à sa réponse, et un objet est instancié dans un processus séparé pour la récupération de la réponse.

Bien que HTTP soit souvent mis en évidence, la spécification SOAP n'impose pas de protocole. Des démonstrations de Services Web sur SMTP ont déjà été réalisées. Pour la couche de transport d'un service Web asynchrone, il existe différentes solutions dont :

- HTTPR : le protocole HTTP d'IBM, le R pour Reliable c'est-à-dire assurer l'acheminement des messages.
- Et la messagerie asynchrone (MQSeries, Sonic MQ, MSMQ, API JMS)

Dans cette étude, nous nous intéresserons plus particulièrement à la messagerie asynchrone.

Le principe de messagerie asynchrone apporte aux systèmes d'information d'entreprise une solution robuste pour communiquer entre applications. En utilisant une queue de messages, une messagerie asynchrone est un système indépendant qui permet à chaque application de traiter une information quand elle le désire : une application envoie un message à une autre application, ce message patiente dans une queue en attendant que l'application destinataire vienne le consommer. Ce système est conçu pour assurer que le message sera délivré.

La solution présentée dans cet article est basée sur le langage JAVA, nous utiliserons donc 2 outils JAVA pour la mise en œuvre :

- AXIS, framework d'Apache pour déployer des Services Web
- JMS, API de Sun pour s'interfacer aux outils de messageries asynchrones

Présentation d'AXIS

AXIS est la continuation du projet Apache SOAP, son but est d'apporter plus de souplesse à l'élaboration de services Web avec SOAP. AXIS se compose de plusieurs sous-systèmes fonctionnant ensemble.

Gestionnaire d'évènements (*Handler*) et chemin des messages

Le principe de fonctionnement d'AXIS consiste en une suite de gestionnaire d'événements appelés à la suite et dans l'ordre. Cet ordre est défini par deux facteurs : la configuration de déploiement et le fait que le moteur de traitement soit serveur ou client. L'objet qui transite entre ces gestionnaires d'événements est nommé *MessageContext*. Cet objet contient le message requête, le message réponse, et des propriétés sous forme de clé/valeur.

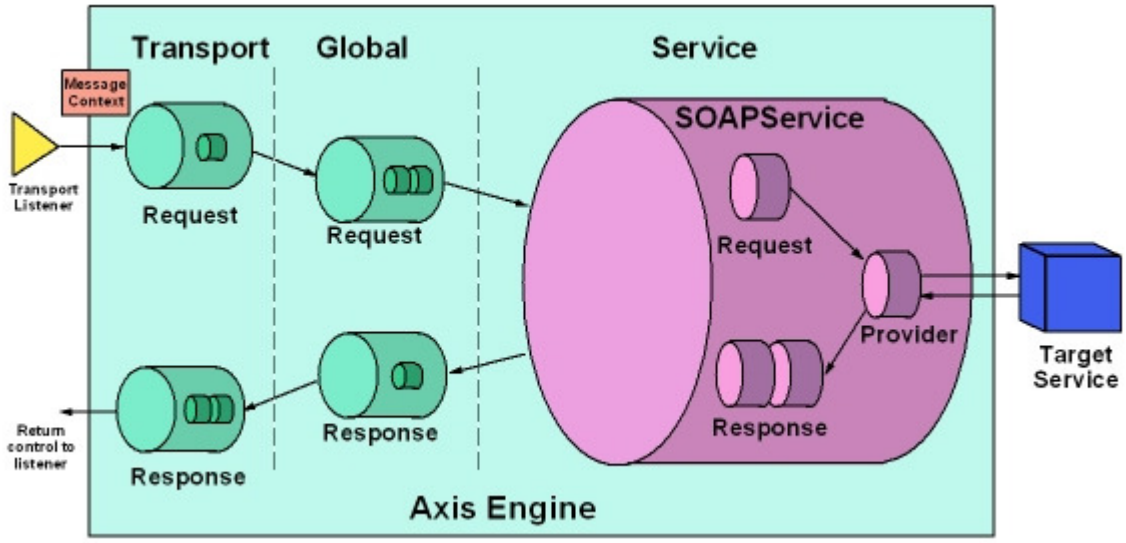
Il y a 2 manières d'invoquer AXIS :

- comme un serveur, selon le protocole de transport choisi : un *listener* doit récupérer les requêtes, créer un *MessageContext* et appeler le moteur AXIS
- comme un client : une application génère un *MessageContext* et appelle le moteur AXIS

Dans tous les cas, le rôle du moteur d'AXIS (*AxisEngine*) est de faire passer le *MessageContext* via un ensemble de gestionnaire d'événements (configurable), chacun d'eux effectuant un traitement bien défini.

Chemin du message sur le serveur

Le chemin est décrit dans le schéma ci-dessous, les gestionnaires d'événements sont représentés sous forme de petits cylindres, les gros cylindres représentant des chaînes de gestionnaire d'événements :



Quel que soit le transport choisi, un message arrive et il est récupéré par un programme à l'écoute sur un port prédéfini, nommé *listener*. Le rôle de celui-ci est d'extraire le message reçu dans un objet *Message* AXIS et de l'insérer dans un *MessageContext*. Le *MessageContext* est créé et associé au transport. Le *listener* passe alors la main à l'AxisEngine

La première tâche de l'AxisEngine est de récupérer le nom du transport. La couche transport dans AXIS est représentée par un objet qui contient une chaîne requête, ou une chaîne réponse, ou les deux. Une chaîne est elle-même un gestionnaire d'événements contenant un ensemble de gestionnaire d'événements s'appelant entre eux. Si une chaîne requête, correspondant au nom du transport, existe, elle passe l'objet *MessageContext* dans sa méthode *invoke(),* ce qui résulte d'un appel à la suite des gestionnaires d'événements de la chaîne requête.

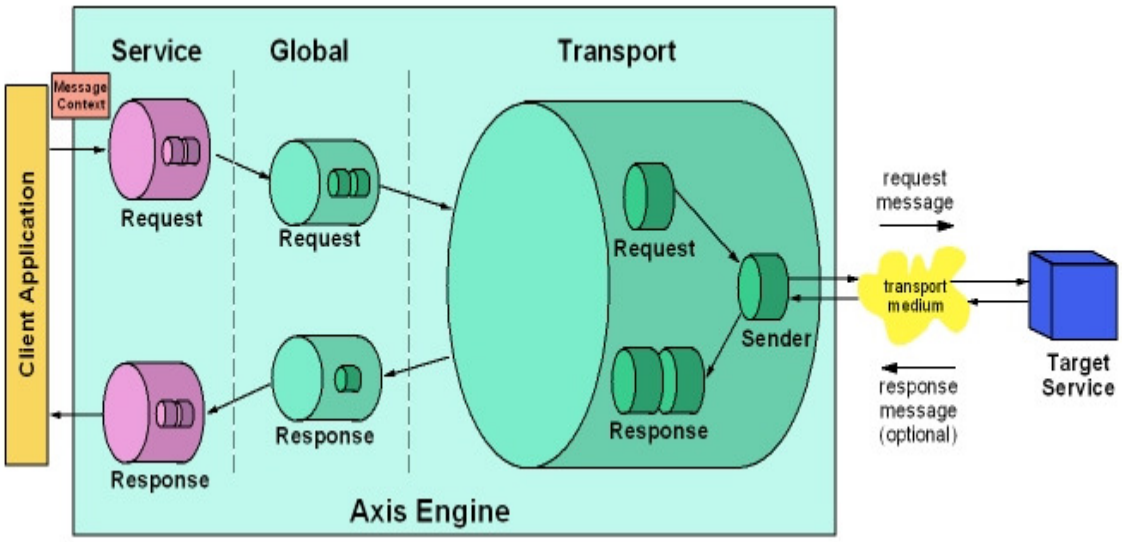
Ensuite AXIS offre la possibilité à l'AxisEngine d'effectuer un traitement spécifique à la requête en précisant une classe *Handler* propriétaire dans le fichier de description de déploiement du serveur (WSDD : Web Service Deployment Descriptor).C'est la couche nommée Global sur le schéma. Il en va de même pour la réponse.

Enfin la couche Service représente une instance de la classe *SOAPService* d'AXIS qui contient la chaîne requête et réponse (vues ci-dessus), ainsi qu'un *Provider*, le dernier *Handler* de la chaîne *SOAPService*, implémentant la logique du service.

Ce *Provider* est, la plupart du temps, une instance de la classe *RPCProvider* : son rôle est d'appeler la classe définie dans le WSDD (<classname>) et d'exécuter la méthode avec les arguments demandés par la requête. Il utilise la convention SOAP-RPC pour déterminer la méthode à appeler et identifier les types des paramètres.

Chemin du message sur le client

Le chemin est similaire côté client, comme le montre le schéma ci-dessous :

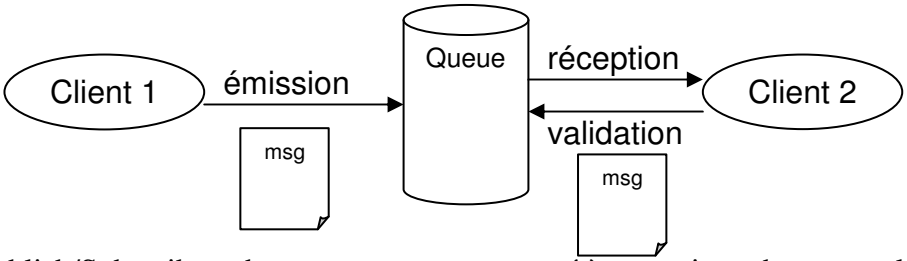


Le gestionnaire d'événements *Service* est appelé en premier lieu ; côté client, il n'y a pas de notion de *Provider*, mais la chaîne requête et la chaîne réponse sont présentes.
Comme pour le serveur, on retrouve la couche Global permettant d'insérer son propre gestionnaire d'événements, pour des traitements spécifiques sur la requête ou la réponse.
Enfin la couche transport contient un gestionnaire d'événements spécifique à l'envoi et la réception du service : le *Sender*. C'est cet objet qui, selon le protocole de transport choisi, va encapsuler le message SOAP pour l'envoyer vers le serveur SOAP cible. La réponse est ensuite véhiculée via la chaîne réponse dans le champ *responseMessage* de l'objet *MessageContext*.

Présentation de JMS

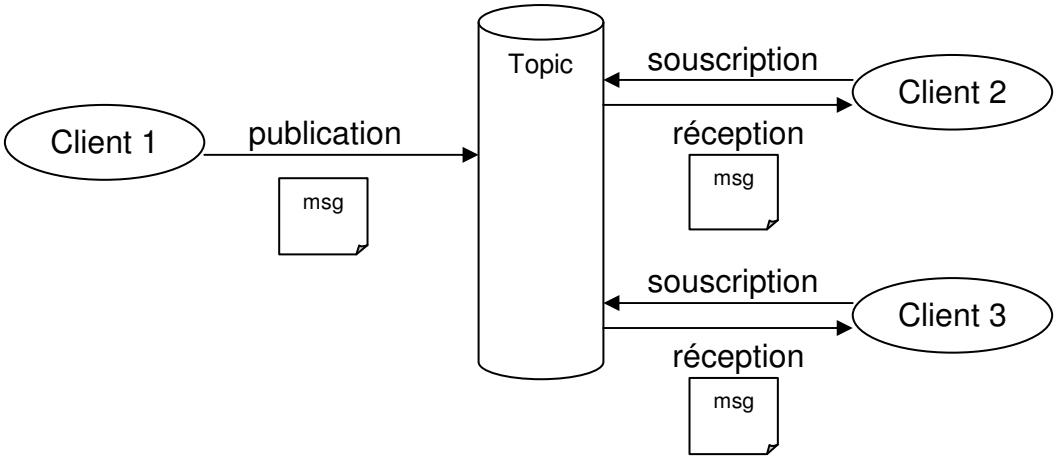
Les produits de messagerie pour l'entreprise, plus communément appelée MOM pour Messaging Oriented Middleware, deviennent un composant essentiel pour intégrer des opérations internes à l'entreprise. Ils permettent de combiner de manière fiable des composants métiers. En plus des fournisseurs traditionnels de MOM, des produits de messageries d'entreprise sont également fournis par plusieurs fournisseurs de base de données et un certain nombre de compagnies Internet. Les clients Java doivent être capables d'employer ces systèmes de transmission de messages. JMS fournit une API commune aux programmes Java pour accéder à ces systèmes. JMS est un ensemble d'interfaces et de sémantiques associées qui définissent comment un client accède à un MOM. JMS est une spécification de Sun, actuellement en version 1.1.
Il existe deux types d'approche :

- la messagerie point à point : chaque message est envoyé à une queue et les clients associés à celle-ci récupèrent le message. La queue retient le message tant qu'il n'a pas été récupéré ou qu'il n'a pas expiré. Un message ne peut être récupéré qu'une seule fois, il n'a donc qu'un seul destinataire. Il n'y a pas de contraintes de temps entre l'émetteur et le récepteur du message, le récepteur peut récupérer le message même si l'émetteur n'est plus actif. Le récepteur valide la bonne réception du message.



- Le Publish/Subscribe : chaque message est envoyé à un topic sachant que plusieurs clients peuvent publier et souscrire à un même topic. Le topic retient le message tant que tous les

souscripteurs ne l'ont pas consommé : un message a donc plusieurs destinataires. De plus, il y a une contrainte de temps entre les publicateurs et les souscripteurs, car un souscripteur ne peut récupérer que les messages publiés après sa souscription et doit rester actif pour recevoir les messages. Cependant l'API JMS permet par une technique de « souscription durable » de ne pas être obligé de rester actif.



Bien que le principe de base d'une messagerie soit d'être asynchrone, il faut préciser qu'avec JMS il y a deux façons d'appréhender la réception d'un message :

- synchrone : un récepteur ou un souscripteur réceptionne le message en appelant la méthode *receive().* Cette méthode bloque le processus en attendant le message ou rend la main au bout d'un temps défini.
- asynchrone : c'est le principe du *listener* présenté ci-dessus. Un programme démon, implémentant l'interface MessageListener, est notifié de l'arrivée du message dans la queue ou le topic et fait appel à la méthode *onMessage().*

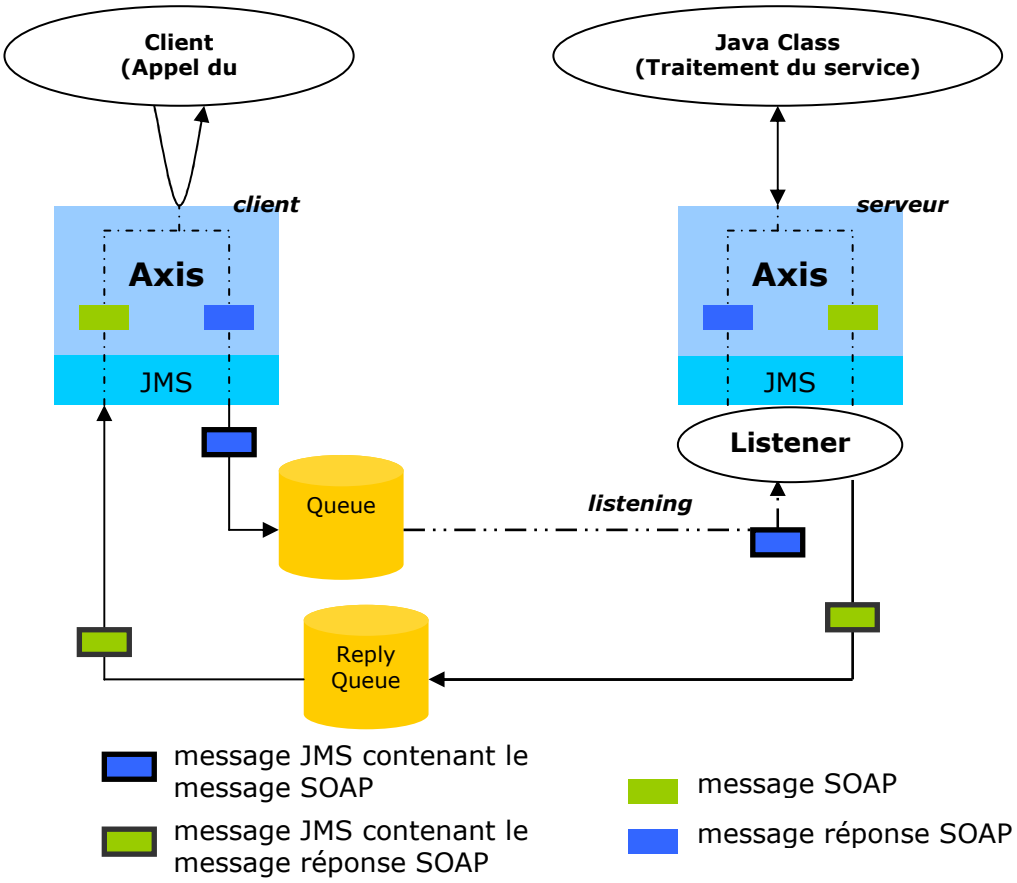
Etude

Nous utiliserons un modèle de requête/réponse asynchrone. Un message SOAP sera véhiculé entre le client et le serveur sous forme de message JMS envoyé à une queue (jms/Queue) et il faudra récupérer une réponse sur une autre queue (jms/ReplyQueue) (une queue n'a qu'un destinataire unique).
AXIS se présente comme un framework indépendant de la couche de transport véhiculant les données SOAP, examinons alors le modèle prévu à cet effet. Comme nous l'avons vu dans la présentation de AXIS, la couche Transport est un élément à part entière de l'architecture fonctionnant avec le système de séquence de *Handler*.

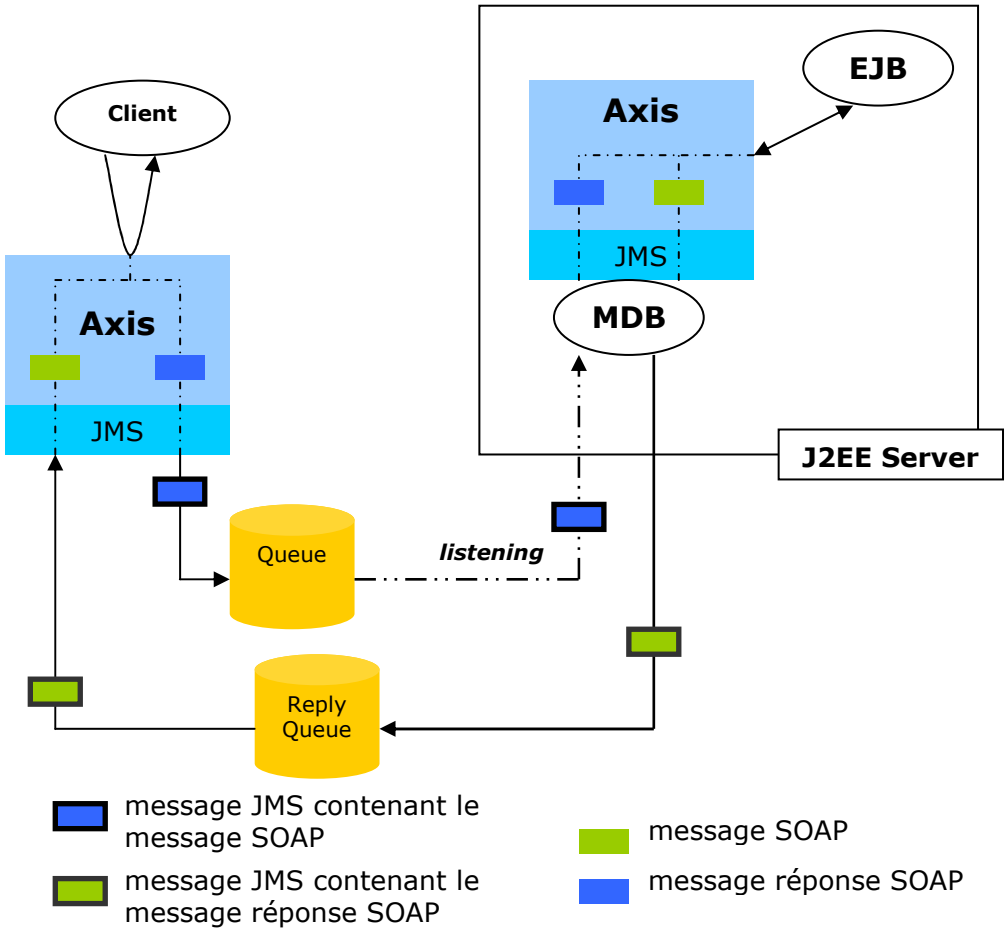
- **Configuration cliente** : La couche transport, caractérisée par la classe org.apache.axis.client.Transport, doit être configurée pour faire appel à un gestionnaire d'événements spécifique destiné à envoyer le message et à le recevoir : le *Sender*. Par défaut c'est la classe HTTPSender (héritant de la classe BasicHandler) qui, comme son nom l'indique, est appelée pour envoyer un message via le protocole HTTP. Ici nous allons créer une classe JMSSender qui, elle, utilisera JMS comme protocole pour envoyer un message.
- **Configuration serveur** : Pour recevoir les requêtes venant d'un client et en renvoyer les réponses, AXIS utilise par défaut un serveur Web : nous devons donc procéder à une modification et intégrer

une classe qui sera à l'écoute des messages JMS comme un serveur Web à l'écoute de requêtes HTTP, JMS fournit une interface *MessageListener* qui permet cela. Nous utiliserons deux types de configurations, dont l'une utilise la spécification EJB 2.0 :

- **Architecture de base** : il faut implémenter l'interface *MessageListener* dans une classe et la déclarer auprès de la queue de message, pour récupérer les messages via la méthode *onMessage()*. C'est dans cette méthode que l'on extrait du message JMS reçu le message SOAP qu'il contient. Il reste à faire appel au service serveur d'AXIS via la classe *AxisEngine* pour interpréter le message SOAP. L'appel de l'*AxisEngine* nécessite de lui associer un descripteur (WSDD : Web Service Deployment Descriptor) nécessaire pour l'interprétation du message SOAP (description des classes et méthodes)



- **Architecture avec EJB 2.0** : la spécification EJB2.0 intègre JMS, en définissant un nouveau type d'EJB destinée à récupérer des messages JMS : le MDB (Message Driven Bean). De la même façon que pour la configuration simple, cet objet implémente l'interface *MessageListener*. On le déclare auprès d'une queue spécifique via le fichier de description du déploiement de l'application J2EE. On y retrouve ainsi le même code que pour la classe précédente. Cependant, dans un environnement J2EE, on préférera associer au message SOAP des EJB ; AXIS permet de le faire en le précisant dans le descripteur WSDD.



Notre modèle de fonctionnement asynchrone bloque le processus client qui se positionne en attente de la réponse. Pour permettre au client de continuer son processus, un autre modèle propose d'instancier, dans une thread séparée, une classe dédiée uniquement au traitement de la réponse. Reste au serveur d'ajouter des propriétés au message de réponse pour permettre au client de l'interpréter correctement.

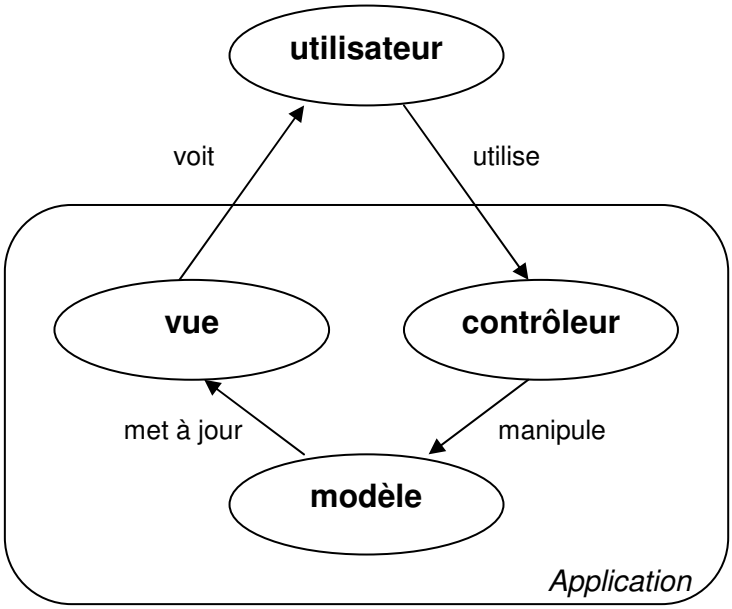
Couplage lâche

Il nous faut maintenant encapsuler au sein du message assez d'information pour qu'il soit plus autonome. Dans un premier temps il nous suffit d'ajouter au message des propriétés liées au contexte comme par exemple le numéro unique que l'on retrouve dans les propriétés de tout message JMS sous le nom *JMSCorrelationID*.
Dans un deuxième temps, face à une architecture contenant un grand nombre services et donc d'interfaces pour y accéder, il devient difficile de rendre notre message autonome sans le rendre complexe. Une solution serait de s'interfacer avec un service unique et de stocker dans le message les informations liées au service final. C'est typiquement un modèle du type MVC2.
L'avantage est donc de pouvoir encapsuler dans un même message des informations d'authentification, des références au client (JMSCorrelationID, replyTo), ainsi que l'interface d'accès au service.

Présentation du modèle MVC2

À l'origine, MVC (pour **M**odel **V**iew **C**ontroller) a été créé pour structurer les applications écrites en langage SmallTalk. Depuis, ce modèle a été réutilisé avec d'autres langages objet. Ce modèle présente une structure à 3 niveaux :

- Le modèle : noyau de l'application, contenant l'ensemble des données utilisées par l'application, indépendamment de leur représentation graphique et de leur interaction avec l'utilisateur
- La vue : les différentes façons de présenter à l'écran les données du modèle
- Le contrôleur : les différentes façons d'agir sur les données du modèle



C'est une technique de conception fréquemment employée par les programmes orientés objets pour faciliter la modularité, la flexibilité et la réutilisation. Une application MVC définit un modèle, des vues et des contrôleurs : par exemple, pour une application J2EE, le modèle est défini par un EJB, les contrôleurs sont les servlet, les vues sont les pages JSP. Le modèle MVC est une avancée importante en terme d'architecture d'applications Web. Elle n'est cependant pas encore idéale : elle oblige à écrire une multitude de servlets, qui sont autant de points d'entrée dans l'application. Pour pallier cet inconvénient, des frameworks ont été développés. Ces frameworks, composés d'un seul servlet (ie un seul contrôleur), sont regroupés sous l'étiquette "Model 2" encore appelé "MVC2".

Implémentation : mvc4ws

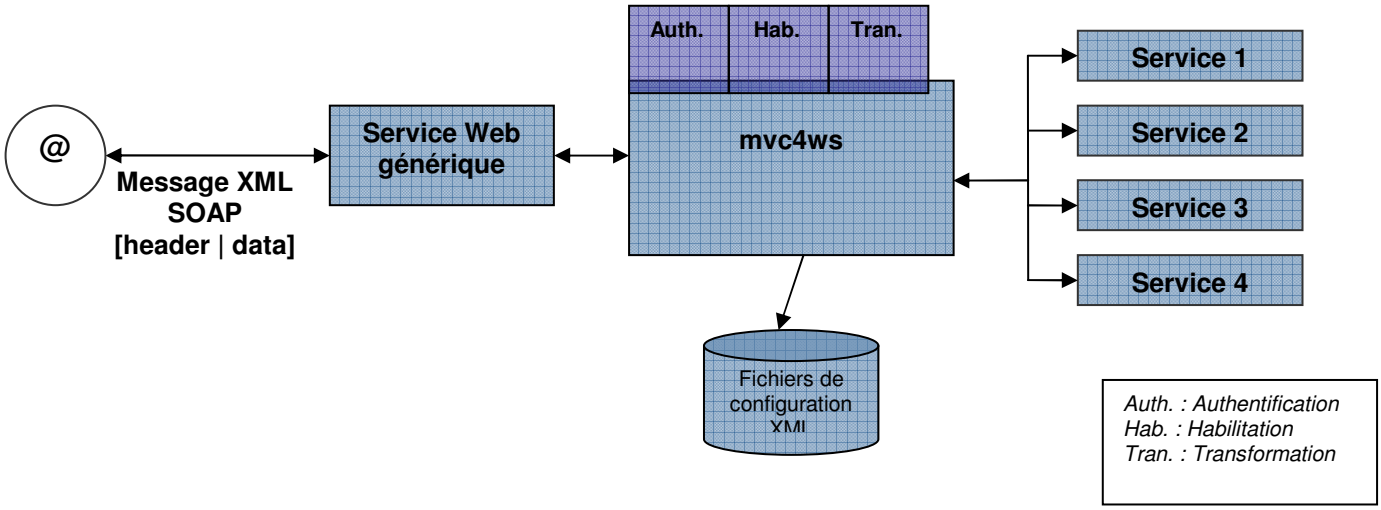
Le framework Struts d'Apache est une implémentation du modèle MVC2 pour déployer des applications Web en Servlets/JSP. Ce framework étant en open source, il nous est possible d'adapter le code pour les Services Web. Il est de plus en plus utilisé par la communauté des développeurs d'applications Web : il est par conséquent un bon choix pour l'implémentation de notre solution.

L'architecture consiste en un Service Web générique définissant une méthode contenant 2 paramètres :

- Une chaîne de caractères contenant un en-tête formaté en XML
- Une chaîne de caractères contenant les données, elles aussi formatées en XML

L'en-tête contient les informations d'authentification, le nom du Service Web final ainsi que le domaine (la notion de domaine est importante car elle permet de différencier la mise en œuvre d'un même service, par exemple un service d'achat n'est pas similaire selon le pays pour une même entreprise). Ces informations sont ensuite utilisées pour accéder à l'objet représentant le service final et lui transmettre les données.

Cependant il faut considérer que l'entreprise utilisant une telle architecture sera forcée d'imposer un format d'entrée des données selon les services proposées, ce qui peut poser des problèmes suivant les relations que celle-ci a avec certains de ses clients. Pour apporter plus de souplesse à ce modèle, il faut prévoir un système de transformations des messages de façon à s'adapter à tous types de format XML en entrée.
Le schéma suivant résume l'architecture, la partie mvc4ws représente un framework MVC2 de Services Web basé sur Struts :



Conclusion

Si l'on ne cherche pas à recevoir une réponse, ou du moins pas une réponse instantanée, l'approche asynchrone s'avère très robuste et très intéressante en terme de fiabilité. Elle implique de ne pas être dépendant d'un serveur surchargé ou de fluctuations du réseau. Par ailleurs, une architecture de Service Web basée sur MVC2 se révèle nécessaire pour rendre le message plus autonome.

Il apparaît clairement que ce type de configuration correspond bien à une architecture système pour l'informatique mobile où la fiabilité du réseau reste incertaine, et les problèmes de charges serveur seront plus présents vu la multiplicité des terminaux.

REFERENCES

1. http://www.sys-con.com/webservices/articleprint.cfm?id=213
2. http://www.webservices.org/index.php/article/articleview/352/1/24/
3. http://www-106.ibm.com/developerworks/webservices/library/ws-asynch1.html
4. http://www.improve-technologies.com/alpha/mvc4ws