Comment gérer la répartition de charge et la tolérance aux erreurs lors de l'invocation d'un service web ? Nous allons étudier le problème et proposer une solution élégante, s'appuyant sur la création d'un plugin du framework Axis, généralement utilisé pour les applications Java.

Les services web permettent d'invoquer des services publiés sur des serveurs HTTP, JMS ou autres. Un service web est identifié par un URL, appelé « port » dans le jargon. Cela identifie un serveur, un port de socket et un chemin. Le client doit générer une requête en XML et l'envoyer en mode POST sur l'URL du service web. Celui-ci analyse la requête, invoque le service et la méthode correspondante, puis retourne une réponse ou une exception à l'appelant. Tout cela au format XML. Comment les serveurs peuvent-ils garantir leur fonctionnement ? Que faire si un serveur tombe ? Est-ce qu'un serveur de secours est disponible ? Etudions les différentes techniques à notre disposition pour garantir l'exécution d'un service web dans le cadre d'une publication HTTP.

1) Répartition de charge par DNS

Tout d'abord, un serveur HTTP est identifié par un nom. En interrogeant un serveur de nom (DNS), il est possible de retrouver l'adresse IP du serveur. Cela permet ensuite d'ouvrir une socket vers l'adresse IP retournée et de demander l'exécution du service web.
Mais, un serveur de nom peut associer plusieurs adresses IP pour le même nom de site. Cela permet de déclarer plusieurs serveurs pouvant répondre à des requêtes HTTP par exemple (Figure 1). Le client est libre de choisir une adresse IP au hasard ou bien le serveur modifie l'ordre des IP pour chaque client. La première adresse est généralement privilégiée par le client. En général, une fois le choix fait, le client continue à utiliser la même adresse IP pour toutes les requêtes vers le même domaine. Plusieurs clients différents peuvent utiliser des adresses IP différentes. La distribution aléatoire des adresses IP utilisées parmi la liste permet de garantir une certaine distribution de charge vers les serveurs web. En tant que client, si le serveur choisi tombe, il n'est pas possible d'en changer tant que le cache client DNS n'est pas rafraîchi (Figure 1).

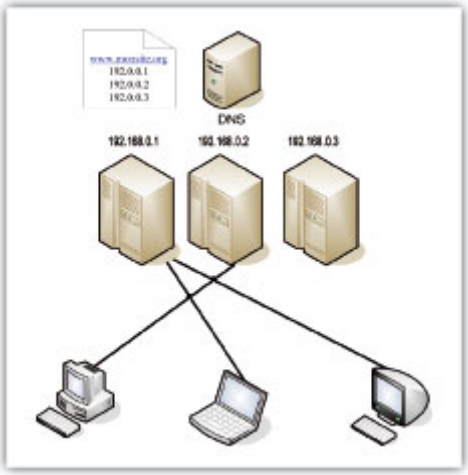


Fig. 1 : Répartition de charge par DNS

Cela est intéressant avec plusieurs clients différents. Lors de communications de serveurs à serveurs, cette technique n'apporte généralement pas grand-chose. En effet, les services sont généralement invoqués par un seul serveur, lui-même très sollicité.

2) Répartition de charge par http

2) Répartition de charge par http Redirect

Certains composants répondent aux requêtes HTTP utilisateurs, sélectionnent un serveur cible et retournent un statut HTTP Redirect en indiquant l'adresse IP du serveur sélectionné. Il s'agit d'un code d'erreur particulier du protocole HTTP, indiquant qu'un serveur a déménagé temporairement ou définitivement vers une nouvelle adresse. En réaction, le client envoie à nouveau sa requête vers le serveur cible. Cette redirection peut être intelligente et s'appuyer sur la localisation géographique du client pour lui proposer un serveur proche de lui.
Cette approche fonctionne pour les requêtes de type GET, mais n'est généralement pas applicable sur des requêtes de type POST. En effet, les navigateurs ne savent pas ré-émettre des requêtes POST. Tous les clients web services ne sont pas capables de traiter une demande de redirection HTTP.

3) Répartition de charge par boîtier

Une autre approche consiste à utiliser un boîtier s'occupant d'effectuer la répartition de charge entre différents serveurs. Ils travaillent au niveau des couches 4 à 7 du modèle OSI. Ils sont placés avant les serveurs web et distribuent les traitements sur les différents serveurs, en utilisant différents algorithmes de répartition. Le plus simple consiste à envoyer les requêtes sur chaque serveur, l'un après l'autre, et de recommencer. D'autres algorithmes plus complexes analysent la charge des serveurs pour envoyer les demandes vers les serveurs les moins sollicités. En cas de défaillance d'un serveur, celui-ci est enlevé de la liste des serveurs cibles. Périodiquement, une requête est envoyée sur les serveurs en échec, pour voir s'ils ne se sont pas réveillés entre temps.
Ces technologies permettent de déployer des clusters verticaux et horizontaux. On appelle « cluster horizontal » un ensemble de machines, toutes similaires. Chacune exécute la même application dans les mêmes conditions (même port). Un cluster vertical est un empilement de la même application dans le même serveur. Chacune utilise un port ou une carte réseau différents. Cela permet, dans les architectures multiprocesseurs, de mieux répartir la charge entre les processus, en évitant les contentions. Cela permet également d'allouer moins de mémoire pour chaque application, allégeant par là le travail d'un ramasse-miettes (Figure 2).

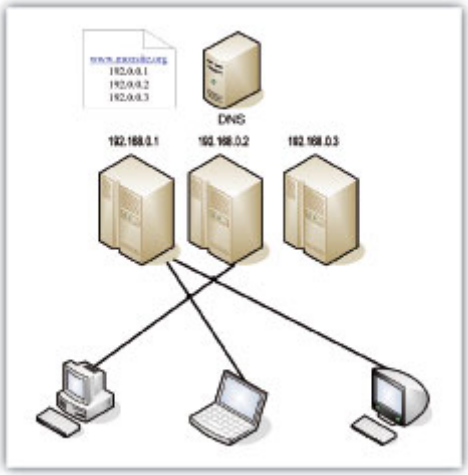


Fig. 2 : Clusters verticaux et horizontaux

4) Solutions logicielles

Ces matériels étant très coûteux, des solutions purement logicielles sont proposées. Par exemple, des logiciels permettent de distribuer les sockets vers différents serveurs. Pour fonctionner, il faut un serveur actif et un serveur passif (Figure 3).

Le serveur actif joue le rôle de routeur NAT. Toutes les requêtes vers les serveurs cibles passent par lui. Une adresse IP flottante ou virtuelle lui est associée (VIP). Cette adresse IP correspond à n'importe quel serveur de la grappe de serveurs. Lorsqu'un client demande à ouvrir une connexion vers cette adresse VIP, il envoie des paquets vers le serveur actif. Celui-ci manipule les en-têtes des paquets et les renvoie vers un des serveurs cibles. Les réponses suivent un chemin inverse.
Elles repassent par le serveur actif avant de retourner vers le client. Il existe donc un point sensible d'interruption : le serveur actif. Si celui-ci tombe, plus rien ne fonctionne.

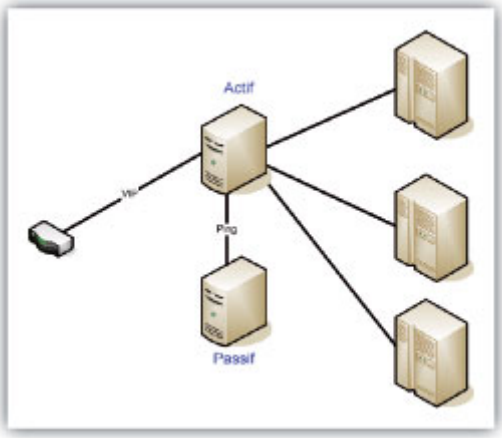


Fig. 3 : Répartition de charge, actif fonctionnel

Le serveur passif est là pour pallier la défaillance du serveur actif. Il envoie très régulièrement des paquets vers le serveur actif. Celui-ci a peu de temps pour y répondre. En l'absence de réaction, le serveur passif en déduit que le serveur actif ne fonctionne plus.
Il souhaite alors rediriger les requêtes destinées au serveur actif vers lui, pour devenir actif à son tour. Pour capturer les paquets destinés au serveur actif, le serveur passif envoie gratuitement un paquet ARP d'association IP/MAC. Ainsi, le client ou le routeur en amont, redirige ensuite tous les paquets vers le serveur passif (Figure 4).

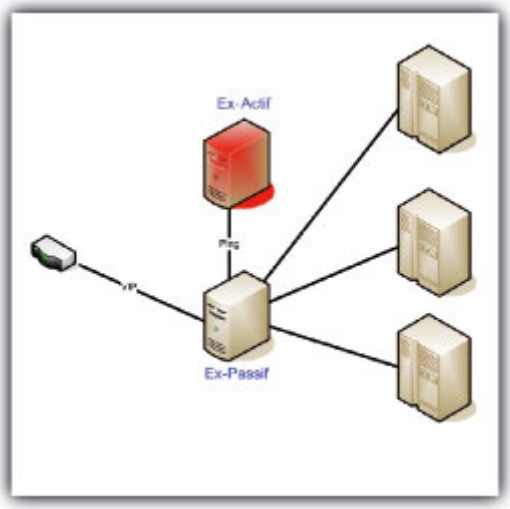


Fig. 4 : Répartition de charge, passif fonctionnel

Fig. 4 : Répartition de charge, passif fonctionnel

Cette technique présente plusieurs inconvénients :
Elle est utilisée par les pirates pour détourner les flux. Les routeurs ou les pare-feu peuvent ne pas apprécier les paquets ARP non sollicités et prendre cela pour une attaque.
Les deux serveurs, l'actif et le passif, doivent être présents sur le même brin réseau. Il n'est pas possible de les placer dans des lieux différents, permettant de pallier un incendie ou une inondation d'un immeuble.
D'autres approches permettent de s'affranchir du passage systématique par un serveur. Une adresse IP virtuelle est déclarée. Un ou plusieurs serveurs écoutent le réseau afin de capturer les requêtes destinées à cette adresse.
Les premiers paquets de l'ouverture d'une socket sont analysés par un serveur. Un serveur cible est désigné. Les premiers paquets lui sont renvoyés. Celui-ci termine alors l'ouverture de la session et traite ensuite toute la communication sur cette socket (Figure 5).

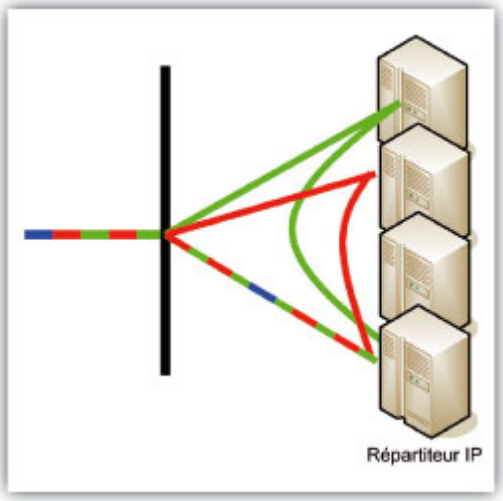


Fig. 5 : Répartition de charge

En cas de défaillance d'un serveur, les sockets courantes sont abandonnées. Les prochaines connexions sont valides.

5) La faiblesse à corriger

Lors de la perte du serveur actif ou d'un des serveurs cibles de la grappe, les connexions en cours sont interrompues et retournent un échec au client. Il y a donc, malgré tous les efforts déployés, un échec de connexion.
Le client peut répéter sa demande, les mécanismes de répartition de charge et de correction d'erreurs prendront le relais pour honorer la nouvelle requête. Le système ne tombe pas complètement, mais l'application cliente risque de ne pas apprécier le refus d'exécuter ponctuellement un service web.
Il faut donc répéter plusieurs fois la requête en cas d'échec de connexion, avant de considérer que la grappe de serveur est hors circuit. Répéter la requête est une bonne idée, mais il existe quand même un risque. Rien ne garantit que la requête précédente n'a pas été exécutée correctement. Il est possible que le plantage du serveur se soit produit lors de l'écriture de la réponse. Par exemple, une application de commerce doit tolérer ce genre de situation aux limites, au risque d'avoir deux commandes enregistrées pour le même client.
Fonctionnellement, il est relativement facile de gérer cela. Est-il préférable de perdre une commande ou de l'honorer deux fois, quitte à gérer un retour client ? Un numéro unique, généré convenablement, permet également de détecter les doublons.

6) Les stratégies

Comment proposer la répartition de charge et la tolérance aux pannes, avec un client de services web ? Il y a plusieurs stratégies. La première, la plus simple en apparence et la plus évidente, consiste à revoir toutes les invocations des services web pour ajouter un traitement d'erreur, demandant de tester plusieurs fois la même requête avant de renvoyer l'erreur à l'application.
Si l'application possède trois cents invocations de services web, il faut modifier les trois cents invocations, avec un code proche, mais à chaque fois différent. Chaque service possède en effet des types et des paramètres qui lui sont propres.
Pour invoquer des services web, il est possible de générer un code client à partir de la description du service au format WSDL. Un code java est produit, appelé « stub », simulant le service équivalent sur le serveur. L'utilitaire WSDL2Java d'Axis s'occupe de générer les stubs clients.
Une stratégie consiste à modifier la génération des stubs des services web. En régénérant tous les stubs avec une version enrichie au fluor, il est facile d'intégrer la répartition de charge et la tolérance aux pannes (Figure 6).

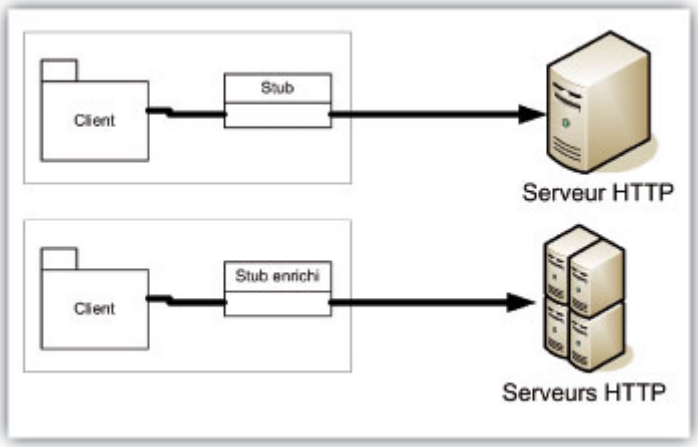


Fig. 6 : Stub enrichi

A condition d'avoir accès aux sources du client et aux WSDL du serveur. Il faut également que tous les services web soient invoqués par des Stubs. Ce n'est pas toujours le cas. Les API permettent des invocations dynamiques.

7) Implémentation

La plupart des applications Java utilisent l'implémentation open source d'Axis (http://ws.apache.org/axis/).
Le code de cette version est particulièrement bien écrit et structuré. La documentation indique que les invocations peuvent s'effectuer suivant différentes technologies : HTTP, JMS, email, etc. Il existe donc une couche de transport, s'occupant d'aiguiller les requêtes vers différentes technologies.
En effet, en étudiant le code, on trouve la classe ~~HTTPSender~~ ayant pour rôle d'envoyer la requête SOAP et de récupérer le résultat, en utilisant le protocole HTTP.
La méthode ~~invoke()~~ attend un paramètre MessageContext, agrégeant toutes les informations de la requête.
Il est tentant de surcharger cette classe, et de redéfinir la méthode ~~invoke()~~ pour exécuter plusieurs fois ~~super.invoke()~~, en modifiant à chaque fois le nom du serveur cible. Il ne s'agit pas de modifier le code, mais d'étendre la couche de transport HTTP.
Il faut ensuite trouver une technique d'intégration du nouveau code. En cherchant un peu, on trouve le fichier ~~client-config.wsdd~~ qui définit les différents drivers à utiliser

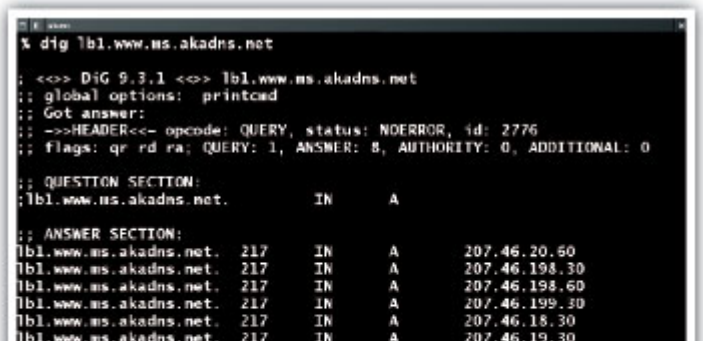
pour les différentes techniques de transport. En modifiant légèrement ce fichier, il est possible d'enrichir la couche transport HTTP.

```
<?xml version=»1.0» encoding=»UTF-8»?>
<deployment name=»defaultClientConfig»
            xmlns=http://xml.apache.org/axis/wsdd/
            xmlns:java=»http://xml.apache.org/axis/wsdd/providers/java»>
 <globalConfiguration>
   <parameter name=»disablePrettyXML» value=»true»/>
 </globalConfiguration>
 <transport name=»http”
    pivot=”java:name.prados.philippe.http.LoadBalancingAxis»/>
 <transport name=»local»
    pivot=»java:org.apache.axis.transport.local.LocalSender»/>
 <transport name=»java»
   pivot=»java:org.apache.axis.transport.java.JavaSender»/>
</deployment>
```

Il n'est pas nécessaire de modifier l'archive ~~axis.jar~~ pour cela. En créant une archive avec notre extension et le fichier de paramétrage d'axis dans le répertoire racine, c'est notre version qui sera prise en compte.
Il est également possible de valoriser la propriété système ~~axis.ClientConfigFile~~ pour indiquer où chercher le fichier. Sans modifier l'application client, quelles que soient les techniques d'invocation ou les spécificités des messages SOAP, en ajoutant une simple archive, il est possible de bénéficier de la distribution de charge et de la tolérance aux pannes.

8) Algorithme

Nous devons maintenant réfléchir à l'algorithme à mettre en place pour bénéficier de cela. Nous devons gérer plusieurs situations, et comme toujours, nous souhaitons réduire le nombre de paramètres. « Moins il y a de paramètres, moins il y a de problèmes de déploiement ».
La première situation à gérer est le cas d'un client d'un service web dont le serveur utilise une technologie de répartition de charge décrite plus haut ou un matériel spécialisé. Dans ce cas, il n'existe qu'une seule adresse IP virtuelle pour représenter tous les serveurs web de la grappe de serveurs. Comme nous l'avons vu, ce n'est pas parce qu'une requête a échoué, qu'il faut en déduire que la grappe entière est hors circuit.
Cela peut être le cas quand un des serveurs tombe pendant la requête. Les autres prennent immédiatement le relais, mais le flux en cours est interrompu.
Cela peut être également le cas si le serveur actif tombe, le temps que le serveur passif prenne le relais. Pour gérer cela, nous allons tenter plusieurs fois la même requête sur le même serveur virtuel avant de le considérer comme inopérant.
La deuxième situation à gérer est le cas d'un ensemble de serveurs avec des adresses IP différentes, déclarées sous le même nom dans la base de données DNS (Figure 7).



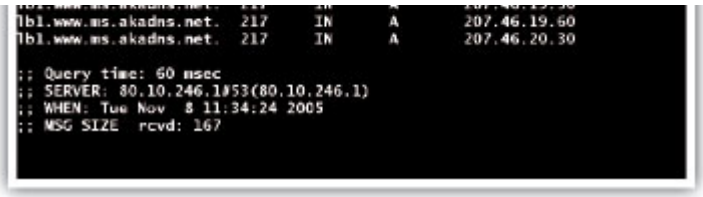


Fig. 7 : Plusieurs IP pour le même nom de domaine

Nous pouvons récupérer la liste des adresses IP, et modifier les invocations du service web pour utiliser l'une après l'autre les différentes adresses. Il ne faut plus utiliser le nom du site, sinon un seul serveur sera sollicité.
Il suffit d'ajouter un nouveau serveur dans la déclaration DNS pour que celui-ci soit pris en compte par les clients. Attention, les serveurs cibles doivent accepter une invocation à l'aide de l'adresse IP.
Il n'est pas possible, dans cette configuration, d'utiliser des serveurs virtuels, différenciés par le nom de domaine utilisé dans le paramètre Host de la requête HTTP. Une petite modification d'Axis permettrait de corriger cela, mais ce n'est pas notre objectif.
La troisième situation est le cas où plusieurs serveurs d'adresses, de ports ou de noms différents, participent à la même application. C'est le cas des grappes verticales par exemple.
Dans ce cas, il faut obligatoirement paramétrer le client. Celui-ci doit pouvoir associer un nom de domaine et un port avec une liste de noms de domaine pouvant servir de remplacement. Pour gérer le problème de répartition de charge et de tolérance aux pannes, il est indispensable que le client connaisse l'architecture de la grappe à invoquer.
Nous avons alors deux boucles imbriquées : l'une s'occupe de répéter les requêtes vers la même adresse IP, l'autre s'occupe de parcourir les différentes adresses pour le même service. Un index est mémorisé pour pouvoir solliciter successivement tous les serveurs. L'algorithme round-robin est utilisé.

Attention:

Il y a un cas limite. Imaginons que tous les serveurs d'une grappe sont hors circuit. Lors des requêtes futures, plus aucun serveur n'est testé. Dans ce cas, il est préférable, à tout hasard, d'en utiliser un pour tenter quand même une requête.

Que faire si un serveur ne répond pas ou ne répond pas correctement ? Après plusieurs échecs, le serveur est déclaré hors circuit. Il ne sera plus sollicité. Pourquoi ne pourrait-il pas revenir à la vie et participer à nouveau à l'application ?
Pour vérifier cela, une tâche doit s'occuper de vérifier périodiquement la présence des serveurs HS. Un mécanisme de battement de cœur permet de vérifier si un serveur est réveillé. Si c'est le cas, son statut retourne à l'état valide. Le serveur peut de nouveau être utilisé. Une page HTTP doit être accessible. En effet, c'est une contrainte de cette approche, mais le seul moyen de détecter le réveil d'un serveur. Si un serveur n'est pas sollicité depuis un moment, il n'est pas possible de savoir s'il est toujours vivant. Il serait intéressant de détecter cela avant que la prochaine requête n'arrive.
Ainsi, le serveur tombé entre deux requêtes ne sera pas essayé. Le client demandera l'exécution du service, directement à ses collègues. Nous allons ajouter un autre battement de cœur, avec une période plus rapide que pour détecter le réveil d'un serveur. En cas d'absence d'un battement, le serveur est déclaré hors circuit.
La tâche de fond a donc deux fonctions :

- Détecter le réveil des serveurs en échec ;
- Détecter par anticipation les serveurs hors circuit.

Pour vérifier qu'un serveur est actif, une simple demande d'URL est suffisante. Par défaut, la page de garde du serveur est demandée. Un paramètre permet de modifier cela.

cela.
Par exemple, il est pertinent de vérifier la présence de la servlet Axis par une demande du WSDL du service web Version : ~~/axis/services/Version?wsdl~~. Vous pouvez, bien entendu, vérifier l'ensemble des composants de votre application dans une page. Un code d'erreur HTTP 500 permet alors de signaler un problème sur la base de données par exemple.
Dans le cas de l'utilisation d'une adresse virtuelle pour le serveur cible, le plugin n'identifie qu'un seul serveur virtuel. Dans ce cas, le battement de cœur n'est pas nécessaire.
Nous avons alors quatre paramètres principaux à valoriser :

- Le nombre de tentative sur le même serveur ;
- Le délai entre chaque battement de cœur pour vérifier la présence d'un serveur ;
- Le délai, plus long, avant de tester le retour d'un serveur en échec ;
- L'URI à utiliser pour le ping.

Les valeurs par défaut de ces paramètres sont suffisantes pour la majorité des situations. Il n'est pas nécessaire de les modifier. Si, et seulement si, vous êtes contraint de déclarer l'architecture de la grappe de serveurs (vous ne voulez pas paramétrer le serveur de nom pour associer plusieurs serveurs sous le même nom, vous utilisez une grappe verticale...), vous pouvez avoir besoin de plus de paramètres. Ceux-ci vont permettre d'associer un couple ~~host:port~~ avec plusieurs couples ~~host:port~~.
Par exemple, vous avez installé plusieurs serveurs d'applications sur la même machine, en utilisant des ports différents ou sur plusieurs machines. Le fichier ~~LoadBalancingAxis.xml~~ devra indiquer comment associer un nom de grappe, avec une liste de serveurs cibles.

```
<?xml version=»1.0» encoding=»UTF-8»?>
<!DOCTYPE loadbalancing
  SYSTEM «http://www.philippe.prados.name/dtd/LoadBalancingAxis.dtd»>
<loadbalancing
  nbretry=»3»
  heartbeat=»5»
  rescue=»30»
  defaultping=»/ping.jsp»
>
<!-- Grappe verticale -->
<host name=»verticale:8080»>
  <alias>localhost:80</alias>
  <alias>127.0.0.1:8080</alias>
</host>
<!-- Grappe horizontale -->
<host name=”horizontale”>
  <alias>serveur1</alias>
  <alias>serveur2</alias>
  <alias>serveur3:8080</alias>
</host>
</loadbalancing>
```

Architecture	IP	Ping	Paramètres client
En amont d'un répartiteur de charge et de tolérance aux pannes	Une seule adresse IP virtuelle pour la grappe de serveurs	Pas nécessaire	Pas nécessaires
En amont de serveurs déclarés dans le DNS	Plusieurs adresses IP délivrées par le DNS.	Nécessaire pour identifier les serveurs HS	Pas nécessaires
En amont de serveurs non			Nécessaires pour déclarer les serveurs d'applications

déclarés dans le DNS ou en présence de grappes verticales	Plusieurs couples d'adresses IP : port	Nécessaire pour identifier les serveurs HS	les serveurs d'applications. Les paramètres peuvent être déportés sur le serveur.

Tab. 1 : Les différentes architectures

Cela oblige les clients à connaître l'architecture du serveur. Que se passe-t-il si elle est réorganisée ? Il faut modifier les fichiers de paramètres de tous les clients. Pour éviter cela, le serveur peut publier son fichier de paramétrage. Les clients déclarent alors uniquement où chercher les paramètres.

```
<?xml version=»1.0» encoding=»UTF-8»?>
<!DOCTYPE loadbalancing
  SYSTEM «http://www.philippe.prados.name/dtd/LoadBalancingAxis.dtd»>
<loadbalancing>
  <config url="http://server1:8080/axis/LoadBalancingAxis.xml"/>
</loadbalancing>
```

Utilisant un URL, tous les protocoles compatibles sont possibles (file, FTP, etc.). La syntaxe est vérifiée par une DTD livrée par le code.

```
<?xml version=»1.0» encoding=»UTF-8»?>
<!-- DTD for LoadBalancingAxis -->
<!ELEMENT loadbalancing ( comment?, (host*|config+) ) >
<!ATTLIST loadbalancing nbretry CDATA «3»>
<!ATTLIST loadbalancing heartbeat CDATA «5»>
<!ATTLIST loadbalancing rescue CDATA «30»>
<!ATTLIST loadbalancing defaultping CDATA «/»>
<!ELEMENT host (alias+) >
<!ATTLIST host name CDATA #REQUIRED >
<!ATTLIST host ping CDATA «»>
<!ELEMENT config EMPTY >
<!ATTLIST config url CDATA #REQUIRED>
<!ELEMENT alias (#PCDATA) >
```

Plusieurs architectures sont alors possibles (Tableau 1). Pour suivre le processus, ~~commons-logging.jar~~ est utilisé. En paramétrant la trace comme il faut, des messages plus ou moins importants permettent de suivre l'état des serveurs.

```
log4j.logger.name.prados.philippe.axis.http=DEBUG
```

Les sources sont disponibles sur mon site : http://www.philippe.prados.name
Avec ce petit code (la boucle principale fait 80 lignes), les clients de services web utilisant l'API d'Axis sont maintenant capables de participer à la répartition de charge et tolèrent le plantage d'un serveur. Avec quelques paramètres ou aucun, sans modifier une application existante, un client SOAP peut bénéficier de ces améliorations.
Pour faciliter la maintenance et le déploiement, il est conseillé de se limiter aux architectures ne nécessitant pas de paramètres.