Le blog d'un développeur Ruby on Rails

2dconcept

JXTA

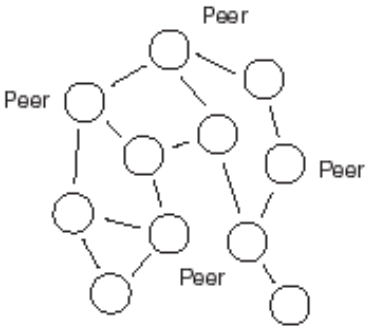
Ce texte a été écrit dans le cadre de mon travail de fin d'études en avril 2006.
Il est le résultat des connaissances que j'ai acquises en développant une application de vidéoconférence et de prise de contrôle à distance sur un réseau JXTA.

Je me suis également basé sur les livres suivants :

- JXTA Java programmers guide
- JXTA in a nutshell & Mastering JXTA

1) Le peer to peer

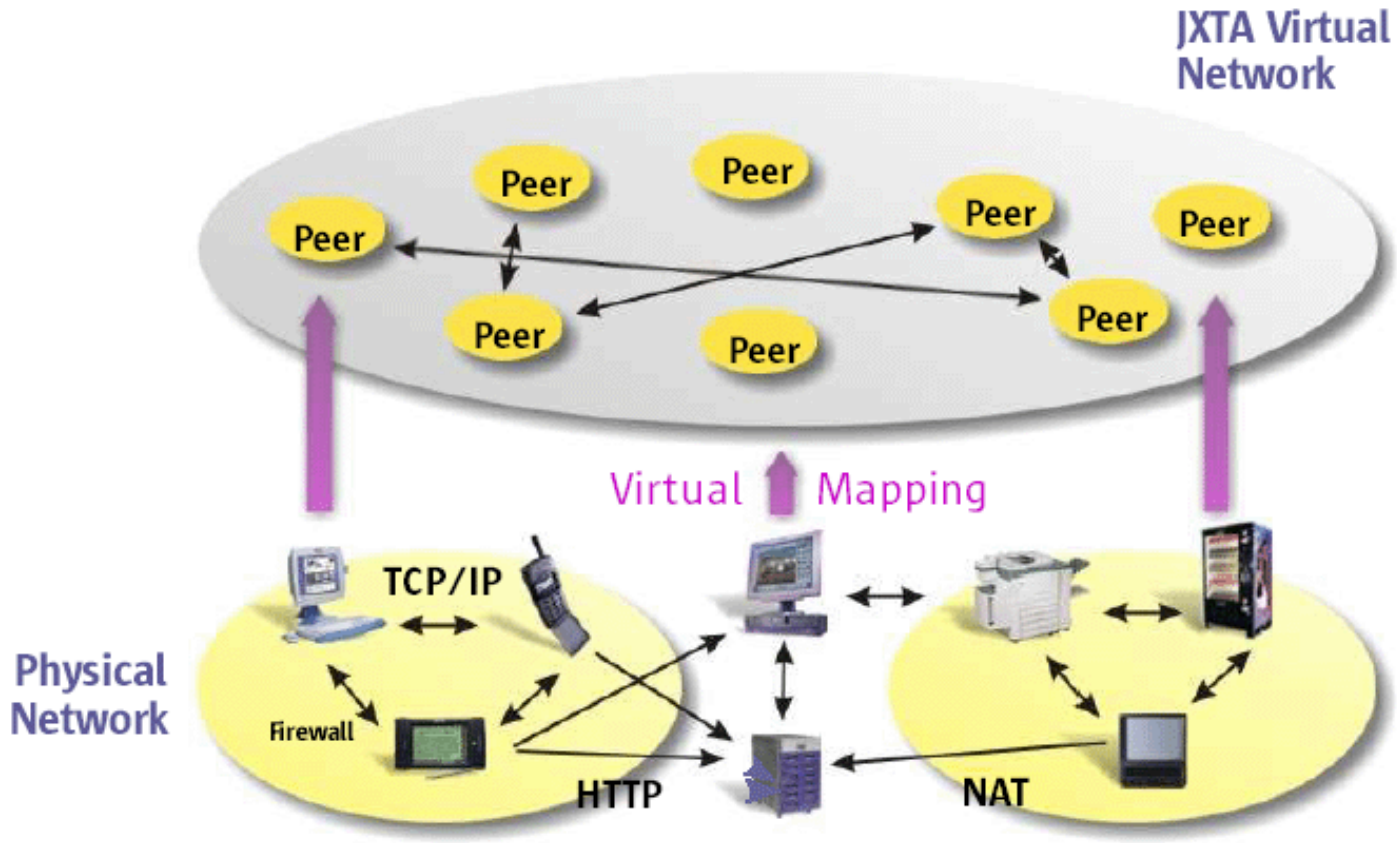
Le peer to peer (point à point ou d'égal à égal en français) est un terme fort connu du grand public. Cette notoriété est due aux nombreux programmes de téléchargement permettant majoritairement d'échanger des fichiers multimédias. Les exemples sont nombreux : un des précurseurs fut Napster mais beaucoup lui ont emboîté le pas (Gnutella, Kazaa ou encore Emule). Si ces programmes ont connu un franc succès, c'est principalement grâce à l'architecture sur laquelle ils reposent : le peer to peer. Si celui-ci est bien connu de nom, il est généralement mal connu dans les faits. En effet, la plupart des gens font une égalité du téléchargement de fichiers et du peer to peer. Ce dernier ne se limite cependant pas à cela, ses possibilités sont en effet très nombreuses ! Il suffit de prendre pour exemple l'application développée dans le cadre de ce stage : des communications textes, audio et vidéos ainsi qu'une fonctionnalité de Remote Desktop ont été implémentées sur un réseau peer to peer. Cette architecture offre donc bien plus que des téléchargements, elle apporte une nouvelle vision d'un réseau dans laquelle chaque ordinateur est considéré à la fois comme un client et comme un serveur.
Elle s'oppose ainsi à l'architecture la plus répandue appelée client/serveur, dans laquelle chaque machine a un rôle bien défini. Ainsi, des ordinateurs sont dédiés à la fourniture de services, ce qui signifie qu'ils sont constamment en attente de connexions des divers clients. Ce modèle permet de centraliser les services et ainsi de les gérer avec facilité mais aussi s'assurer de la sécurité. En contre partie, si le serveur tombe en panne, tous les clients dépendants de lui se retrouvent bloqués et c'est tout le réseau qui peut en être affecté. Dans un réseau peer to peer, tous les peers sont interconnectés et peuvent agir à la fois comme client et comme serveur. Une même machine peut à la fois télécharger un fichier en provenance d'un autre peer tout en partageant ses fichiers avec le reste du réseau.



Ainsi, toute personne du réseau peut offrir des services et en consommer. Une recherche sur un service particulier nous indique les peers qui offrent celui-ci, il suffit alors de se connecter à celui de notre choix. Toute dépendance à un unique serveur est donc éliminée. Le résultat est donc un réseau hautement disponible sur lequel un service n'est pas confiné à un seul endroit, mais dupliqué en de nombreux points du réseau.

2) Généralités sur JXTA

JXTA (à prononcer juxta, en provenance du mot juxtapose) est une suite de protocoles Open Source développée par Sun. Il offre la possibilité à chaque composant d'un réseau de communiquer, de collaborer et de partager des ressources. Les peers JXTA créent un réseau virtuel au dessus du réseau physique, cachant ainsi la complexité de celui-ci. Dans un réseau virtuel JXTA, chaque peer peut interagir avec tous autres, sans se soucier de son emplacement, du type de composant ou de l'environnement d'exécution et même si il est situé derrière un firewall ou qu'il utilise une autre couche de transport réseau. La technologie JXTA fonctionne sur n'importe quel composant, qu'il s'agisse d'un PC ou d'un PDA, …



Basé sur des standards tels que TCP/IP, HTTP et XML, JXTA est indépendant de tout langage de programmation, de tout plateforme réseau et tout système d'exploitation. JXTA a été étudié pour permettre à des peers interconnectés de se découvrir, de communiquer facilement ainsi que s'offrir différents services. La sécurité des communications peut être assurée grâce à différents mécanismes tels que le SSL et les certificats.
JXTA fournit donc aux développeurs une structure solide permettant d'élaborer des applications peer to peer. Jusqu'à présent, les bases de chaque application étaient différentes et celles-ci n'étaient donc pas compatibles entre elles, notamment à cause du format non standard des messages. Le but de JXTA est donc de devenir un standard dans ce domaine évitant de plus, de réinventer la roue à chaque développement.
La communauté JXTA a développé différentes APIs, propres aux différents langages. Celle qui nous intéresse particulièrement est bien sûr celle spécifique à Java.

3) Pourquoi JXTA ?

La société Defimedia était déjà en possession d'un outil de vidéoconférence avant notre arrivée mais il souffrait d'un problème de taille : les données étaient régulièrement bloquées par les firewalls et les communications ne passaient pas sur les réseaux pratiquant le NAT (Network Address Translation). Or, ce dernier et les firewalls étant de plus en plus répandus, il devenait impératif de corriger ce problème dans la nouvelle application.
JXTA offre des solutions efficaces à ces problèmes et c'est une des raisons principales pour laquelle nous avons choisi cette technologie comme base réseau de notre projet. Ces solutions sont expliquées plus en détails au chapitre x. En plus de cela, nous avons vite découvert la puissance de cette technologie ainsi que l'existence d'une multitude de fonctionnalités. JXTA est une fameuse brique et pourrait faire à lui seul le sujet de tout un stage.
Je conseille vivement à tout développeur de découvrir les possibilités de cette architecture car elle devrait, selon moi, entrer en ligne de compte pour la plupart des futurs développements web.

4) L'architecture JXTA

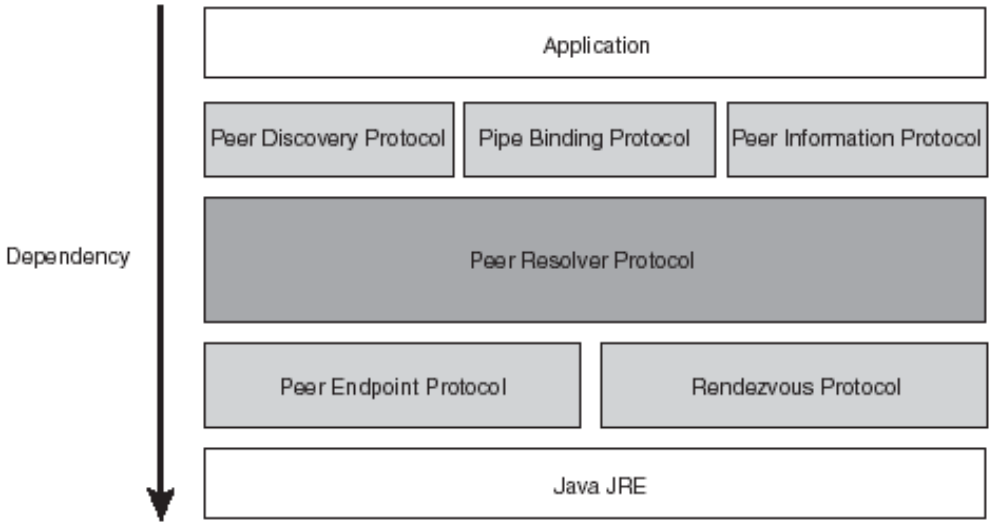
Pour répondre aux nombreux objectifs tels que les standards et l'interopérabilité, l'architecture JXTA a été soigneusement étudiée.
Ainsi, une suite de protocoles a été développée, indépendamment de tout langage, chacun apportant ses services spécifiques.

Parmi ceux-ci, permettre aux peers :

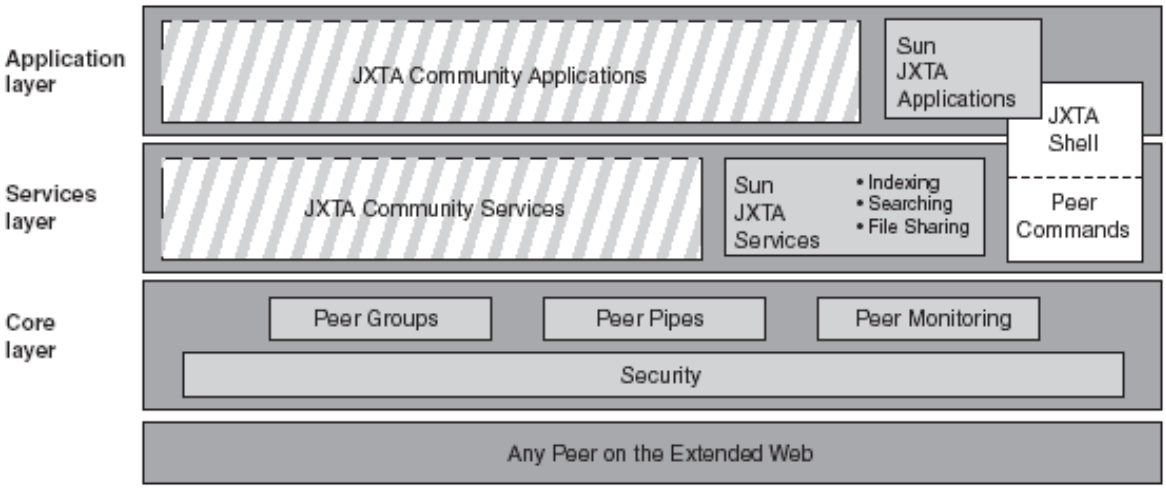
- de se découvrir les uns les autres ;
- de s'auto organiser en groupes ;
- d'annoncer et découvrir des ressources ;
- de communiquer.

Cette suite est actuellement composée de six protocoles :
- Peer Resolver Protocol (PRP) : utilisé pour envoyer une requête à un nombre indéterminé de peers et recevoir une réponse ;
- Peer Discovery Protocol (PDP) : utilisé pour annoncer et découvrir du contenu ;
- Peer Information Protocol (PIP) : utilisé pour obtenir l'état d'un peer ;
- Pipe Binding Protocol (PBP) : utilisé pour créer un chemin de communication ;

- Peer Endpoint Protocol (PEP) : utilise pour trouver une route entre deux peers ;
- Rendezvous Protocol (RVP) : utilise pour propager les messages dans le réseau.



Pour permettre à ces protocoles de travailler ensemble et de former un système complet, une architecture doit être mise en place.
JXTA est basé sur un modèle en couche, ce qui lui apporte de nombreux avantages.
Chaque couche apporte ainsi ses services, permettant aux couches supérieures de les utiliser.
Chacun peut alors se spécialiser dans la couche qui l'intéresse et développer des services pour celle-ci, qu'ils soient Open Source ou commerciaux.



C'est dans le core layer que se situe le code implémentant les différents protocoles.
Le services layer offre différents services qui utilisent les protocoles de la couche inférieure pour accomplir une tâche. Il existe deux types de services, les essentiels ou non. Ainsi, le service permettant à un peer de rejoindre un groupe est considéré comme essentiel tandis que celui permettant de connaître la météo ne l'est pas. Les services essentiels sont donc ceux indispensables au bon fonctionnement du réseau.
Le but de l'architecture en couche et des services est clairement la réutilisation, évitant de réécrire le code à chaque développement.
L'application layer est la couche où viennent se placer les différentes applications, profitant des services de la couche précédente. Ces applications regroupent les peers et leurs offrent différentes fonctionnalités.

5) Terminologie

5.1) IDs

Dans un réseau peer to peer où le nombre de participants et de ressources peut bien sûr être très élevé, il est nécessaire de se fier à une technique fiable de référencement et d'identification.
Un simple nom ne peut pas suffire à identifier une ressource, pour des raisons évidentes de duplication.
JXTA résout ce problème en introduisant un ID JXTA, également appelé URN.
Il s'agit d'une chaîne de caractère unique, utilisée pour identifier six types de ressources : les peers, les groupes de peers, les tubes de communications, les contenus, les spécifications de modules et les classes de modules.
Ces identifiants peuvent en fait être considérés comme étant le système d'adressage de JXTA.

Un identifiant est composé de trois parties :

- un identifiant de namespace = jxta ;
- une spécification de format = urn ;
- un ID = valeur unique.

Exemple : urn:jxta:uuid
59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903

Il est intéressant de noter qu'il existe trois IDs réservés, il s'agit du NULL ID, du World Peer Group ID et du Net Peer Group ID., les deux derniers étant des groupes par défaut.

5.2) Les peers

Le peer est l'élément de base de JXTA. Il peut s'agir de n'importe quel type de composant réseau, depuis le PDA jusqu'au super ordinateur.
Chaque peer opère de manière indépendante par rapport aux autres peers et est identifié d'une manière unique par un peer ID.
Il est indispensable que les peers puissent communiquer entre eux : c'est pourquoi les peers publient une ou plusieurs interfaces réseau à utiliser avec les protocoles JXTA. Chaque interface publiée est annoncée comme étant un peer endpoint, qui permet d'identifier l'interface de manière unique. Les peers endpoints sont utilisés pour créer des connexions directes, point à point, entre deux peers.
Les peers ne sont cependant pas obligés d'avoir des connexions directes avec tous les autres, il existe en effet des peers intermédiaires (appelés relay ou rendezvous peers) qui permettent de router les messages entre des peers séparés. Ceux-ci permettent par exemple de résoudre différents problèmes de configuration réseau tels que le NAT, les firewalls ou les proxys.
Les peers sont automatiquement configurés pour découvrir d'autres nœuds sur le réseau et pour former des relations appelées peer groups.
Les peers n'ont aucune obligation quant à la durée de leur connexion sur le réseau. Un peer ne peut donc pas être garanti que le peer lui offrant un service restera online jusqu'à la fin de celui-ci !
Dans la plupart des cas, le service principal que fournit un peer et le partage de contenu.
Le peer fait connaître au groupe le contenu qu'il est disposé à partager et utilise les services du groupe pour rechercher le contenu par lequel il est intéressé.

Des peers peuvent également être créés pour fournir des services spécialisés que le groupe n'offre pas. Par exemple, un peer pourrait servir d'intermédiaire entre un client et un magasin pour valider une carte de crédit. Le client fournit les informations de sa carte de crédit au peer intermédiaire, le magasin ne s'occupe alors que de la commande (clé duale).

5.3) Les groupes de peers

Un peer group est une collection de peers offrant le même type de services, chacun étant identifié par un peer group ID unique.
Chaque groupe peut établir sa propre politique d'accès, de libre (c'est-à-dire où tout le monde peut rejoindre le groupe), jusqu'à extrêmement sécurisée (authentification par certificat, par exemple). Il existe donc des groupes privés et publics.
Les peers peuvent être présents dans plusieurs groupes simultanément, le groupe par défaut auquel chaque peer appartient étant le Net Peer Group.
Un peer doit obligatoirement faire partie d'un groupe pour pouvoir communiquer avec d'autres peers, la communication étant bien sûr limitée aux autres membres de ce groupe.
Chaque peer group possède un certains nombres de services de base qui ont été définis par une spécification. Il s'agit de :

- Discovery Service : permet de rechercher du contenu dans un groupe ;
- Membership service : permet la création d'un groupe sécurisé ;
- Access Service : permet de valider l'accès d'un peer à un groupe ;
- Pipe Service : permet de créer et utiliser des tubes de communications ;
- Resolver Service : permet les requêtes et les réponses pour les services de peers ;
- Monitoring Service : permet aux peers de surveiller d'autres peers et groupes.

Les groupes ont bien sûr la possibilité d'implémenter autant d'autres services additionnels qu'ils le souhaitent.

5.4) Advertisements

Toutes les ressources JXTA, telles que les peers, les groupes, les tubes de communications et les services, sont représentés par des advertisements. Ces annonces sont des méta-données représentées par des fichiers XML.
Les protocoles JXTA utilisent ces annonces pour décrire et publier les ressources des différents peers. Les peers découvrent les ressources en recherchant les annonces correspondantes et peuvent les sauvegarder localement grâce à un système de cache.
Chaque advertisement est publié avec une durée de vie qui permet de spécifier le temps d'activité de la ressource associée. Une seconde annonce peut être publiée pour augmenter cette durée.

Les annonces suivantes sont définies par les protocoles JXTA :

- Peer Advertisement : Décrit un peer. Son utilité principale est garder les informations principales d'un peer telles que son nom, son ID, ses endpoints disponibles ainsi que d'autres caractéristiques.
- Peer Group Advertisement : donne les informations principales d'un groupe telles que son nom, son ID, sa description et ses services.
- Pipe Advertisement : décrit un tube de communication et est utilisé pour créer les endpoints d'entrée et de sortie correspondants. On retrouve de nouveau un ID, mais aussi un type (point à

point, propagé, sécurisé, …).
- Module Class/Spec/Impl advertisement : offrant toutes les informations nécessaires sur un module.
- Rendezvous Advertisement : décrit un peer qui agit comme un rendezvous pour un peer group donné.
- Peer Info Advertisement : son utilité principale est de fournir des informations spécifiques sur le statut courant d'un peer, telles que son uptime, le nombres de messages reçus et envoyés, l'heure du dernier message reçu et envoyé, etc.

Voici un exemple de Pipe Advertisement :

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
   <Id>
      urn:jxta:uuid-
   59616261646162614E504720503250338E3E786229EA460DADC1A176B69B731504
   </Id>
   <Type>
       JxtaUnicast
   </Type>
   <Name>
      TestPipe.end1
   </Name>
</jxta:PipeAdvertisement>
```

Il est intéressant de noter que les advertisement énoncés ci-dessus peuvent être dérivés pour créer des annonces spécifiques.

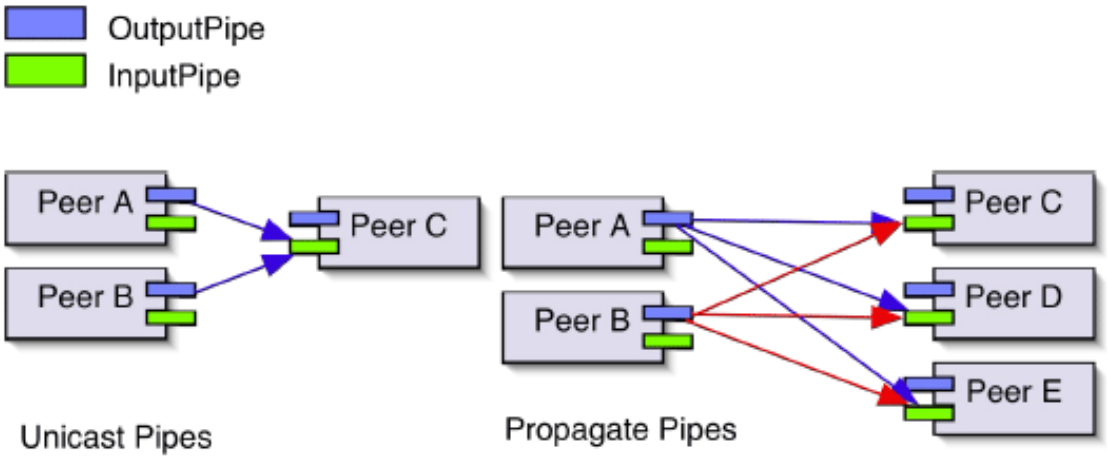
5.5) Les pipes

JXTA utilise les pipes comme mécanisme de communication. Ce concept de tubes de communication découle des systèmes UNIX. L'information entre dans une des extrémités du tube et sort de l'autre côté. Grâce aux pipes, des messages peuvent être envoyés entre les différents peers sans se soucier de l'infrastructure sous-jacente. Les peers ne doivent pas s'inquiéter à propos de la topologie du réseau ou de la localisation d'un autre peer pour pouvoir communiquer.
Les pipes utilisent la notion de endpoint pour désigner les points d'entrée et de sortie d'un canal de communication.

Il existe trois types de pipes :

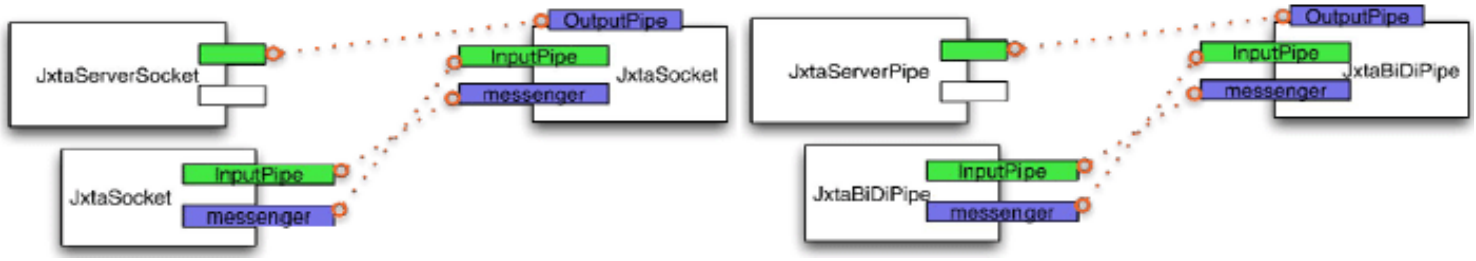
- Unicast : unidirectionnel, non sécurisé et non fiable.
- Unicast secure : unidirectionnel, sécurisé et non fiable.
- Propagating : tube de propagation, non sécurisé et non fiable.

Les pipes unicast connectent un peer à un autre pour une communication dans un sens unique.
Le pipe de propagation un tube de sortie à plusieurs entrées de tubes.



5.6) Sockets et tubes bidirectionnels

Etant donné que les pipes classiques sont unidirectionnels et non fiable, il était nécessaire d'implémenter d'autres canaux de communication, bidirectionnels et fiables.
C'est dans cette optique qu'est apparue la Reliability Library permettant d'assurer le séquencement des messages ainsi que leur livraison.
Des systèmes de communication plus élaborés ont alors été créés sur les pipes classiques, par l'intermédiaire de cette librairie.
Il s'agit des JxtaSocket-JxtaServerSocket et des JxtaBiDiPipe/JxtaServerPipe.



Il sera expliqué plus tard dans ce chapitre que d'autres mécanismes peuvent être utilisés, parmi ceux-ci les JxtaMulticastSocket.

5.7) Les messages

Un message est un objet qui est transmis entre des peers JXTA. Il s'agit de l'unité de base d'échange de données.
Un message est essentiellement représenté par une liste de couples nom/valeur et est composé d'une enveloppe et d'un corps.
L'enveloppe est dans un format standard et contient :

- Un en-tête
- Les informations sur le endpoint source (URI)
- Les informations sur le endpoint destination (URI)
- Un digest (optionnel)

Le corps du message est de taille variable et peut, en plus du contenu, véhiculer des informations de sécurité (credential).

Le format est assez simple pour la bonne et simple raison qu'il doit être flexible et facilement adaptable en fonction de l'environnement. En effet, ces messages peuvent être implémentés au dessus de n'importe quel type de réseau, pour répondre à un des objectifs de base de JXTA.
Il y a deux formats d'encodage possibles pour les messages : le XML et le binaire.
Les avantages étant assez évidents : le XML optimise la portabilité mais pénalisé les performances et inversement pour le binaire.

5.8) Le discovery

Le discovery est le processus permettant à un peer de rechercher des advertisements et donc, trouve les services par lesquels il est intéressé.
Ce processus s'applique à n'importe quel type de ressource JXTA, un peer peut découvrir d'autres peers, mais aussi des groupes, des pipes, ainsi que n'importe quelle autre ressource.
Cette recherche s'effectue à l'intérieur d'un groupe spécifique.

Il existe trois types de discovery :

- Local
- Direct
- Propagé

5.8.1) Local

Chaque peer possède son propre cache, qui peut être comparé à un système de pages jaunes.
Le cache est vide lors de la première connexion du peer mais il se remplit au fur et à mesure en fonction des advertisements reçus.
Lorsqu'un peer lance une recherche, il commence d'abord par fouiller son cache local pour trouver l'information. Si le résultat est positif, le peer obtient directement les données nécessaires pour se connecter à la ressource.
Si aucun résultat ne correspond à la recherche, une découverte distante va être effectuée en envoyant des advertisements spécifiques à la recherche aux peers connus.
L'inconvénient du cache est que l'information s'y trouvant peut être périmée. Soit parce que le peer correspondant est déconnecté du réseau, soit parce qu'il ne possède plus la ressource demandée. Ce désagrément peut être évité en associant une durée de vie aux entrées du cache.

5.8.2) Direct

Grâce au direct discovery, un peer a la possibilité de contacter tous les participants de son réseau et découvrir tous les services et contenus que ceux-ci fournissent.
Cette fonctionnalité est possible grâce aux techniques de multicast et de broadcast, permettant de contacter d'une traite plusieurs peers ou tout le réseau.
Le message émis contiendra les différents critères de recherche et chaque peer pourra répondre de manière appropriée.
Il est évident que cette méthode ne fonctionne que sur son propre réseau/sous-réseau. Ces paquets de broadcast et multicast ne sont en effet pas autorisés à traverser les routeurs pour accéder à Internet.

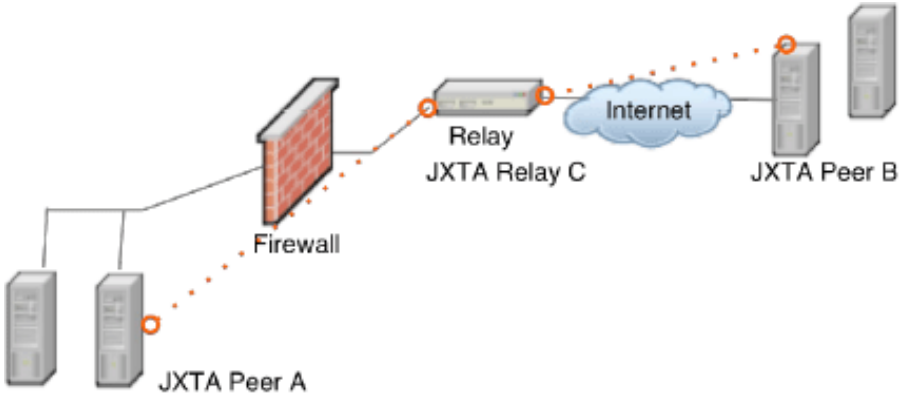
5.8.3) Propagated

Cette dernière technique permet d'effectuer une recherche sur Internet, et donc de passer au-delà des frontières du réseau privé.
Un peer spécialisé est indispensable à son bon fonctionnement, il s'agit d'un rendezvous peer.
Celui-ci porte bien son nom, il s'agit en effet d'un endroit ou les peers peuvent se rendre pour trouver d'autres peers. Le peer de rendezvous garde en cache les advertisements de tous les peers avec lesquels il est entré en contact.
Les peers de rendezvous transmettent également les messages de discovery pour aider les autres peers à découvrir les différentes ressources.
Quand un peer rejoint un groupe, il recherche automatique après un rendezvous pour ce groupe. Si il n'en trouve pas, il devient lui-même rendezvous pour ce groupe.
Chaque rendezvous mémorise une liste d'autres rendezvous ainsi que les peers qui l'utilise comme leur rendezvous.
Un point de rendezvous maintient un index comprenant les advertisements reçus, et fait circuler chaque nouvelle entrée aux autres rendezvous.
Un peer contacte donc le rendezvous et lui soumet les paramètres de sa recherche. Il s'agit d'un avantage considérable pour un peer se trouvant sur un réseau privé, le rendezvous lui permet en effet de sortir sur le réseau public.
Etant donné qu'il existe un nombre élevé de rendezvous sur Internet, une recherche à grande échelle peut être effectuée en très peu de temps.
Sur l'exemple suivant, le peer A est configuré pour utiliser le peer R1 comme rendezvous.
Quand A effectue un discovery, il envoie donc la requête à son rendezvous et aux autres peers de son réseau grâce au mutlicast.
Si le peer de rendezvous trouve un index correspondant à la recherche, il répond directement. Dans le cas contraire il envoie la demande aux autres rendezvous qu'il connaît, qui vont tenter à leur tour de résoudre la requête en cherchant dans leur annuaire.

Pour éviter les boucles infinies entre les rendezvous, c'est-à-dire pour éviter que ceux-ci se renvoient sans cesse la même requête, un système de TTL (Time To Live) a été mis en place.
Il s'agit d'un compteur propre à chaque recherche qui est décrémenté à chaque passage par un point de rendezvous. Si ce compteur devient nul, le paquet est jeté.
Les peers maintiennent également des listes de requêtes déjà traitées ce qui leurs évitent de la retraiter par la suite.

5.9) NAT/Firewall

Il existe un autre type de « super peer » : le relay.
Celui-ci maintient des informations sur les chemins menant à d'autres peers et route les messages entre les différente peers. Un relay peut donc être assimilé à un routeur.
Lorsqu'un peer ne connaît pas la route vers un autre, il peut donc interroger le relay qui va lui transmettre les informations nécessaires.
Le relay transmet également les messages au nom de peers qui ne peuvent pas le faire eux-mêmes directement, par exemple à cause du NAT. Il peut servir de véritable pont entre deux réseaux physiques ou logiques.
Un relay permet également à un peer situé derrière un firewall d'être contacté depuis l'extérieur. En effet, le relay peut garder les différents messages à destination du peer en mémoire, à charge de celui-ci de venir les consulter à intervalle régulier.



N'importe quel peer peut être configuré comme rendezvous ou comme relay, et même implémenter ces deux fonctions en même temps.

6) Le shell JXTA

Le shell JXTA est une application interactive qui fournit un accès direct au réseau JXTA, de la même manière q'un shell UNIX donne accès au système d'exploitation.
Par l'intermédiaire d'une série de commandes, il est possible d'interagir avec le réseau et d'obtenir des informations sur celui-ci.

Les commandes sont très nombreuses, en voici quelques exemples :

- whoami : permet d'obtenir les informations sur l'utilisateur connecté, pseudo, ID, …
- peers : permet de lancer un discovery sur les peers.
- groups : permet de lancer un discovery sur les groupes.
- mkpgrp : permet de créer un nouveau groupe.
- join : permet de rejoindre un groupe.
- search : permet de recherche des advertisements.
- mkpipe : permet de créer un tube de communication.
- mkmsg : permet de créer un message.
- send/recv : permettent l'envoi et la réception de messages.
- Man : permet d'obtenir la documentation d'une commande.

Ces commandes sont bien sûr polymorphes et peuvent prendre en entrée, différentes combinaisons de paramètres.
Le shell est extensible, cela signifie que de nouvelles commandes peuvent lui être ajoutées très simplement. Il suffit pour cela de créer une classe dérivant de ShellApp. Celle-ci apporte les méthodes startApp et stopApp qu'il faut redéfinir. Ces deux méthodes seront respectivement appelées au à l'appel et à la fin de la commande. Le bytecode correspondant devra ensuite être placé dans le répertoire bin du shell.
Le shell JXTA peut s'avérer être très pratique pour dépanner une application.

Pour information, voici la liste complète des commandes de base du shell JXTA :

```
cat         Concatane and display a Shell object
chpgrp      Change the current peer group
clear       Clear the shell's screen
env         Display environment variable
exit        Exit the Shell
exportfile  Export to an external file
get         Get data from a pipe message
grep        Search for matching patterns
groups      Discover peer groups
help        No description available for this ShellApp
history     No description available for this ShellApp
importfile  Import an external file
instjar     Installs jar-files containing additional Shell commands
join        Join a peer group
leave       Leave a peer group
man         An on-line help command that displays information about a
specific Shell command
mkadv       Make an advertisement
mkmsg       Make a pipe message
mkpgrp      Create a new peer group
mkpipe      Create a pipe
more        Page through a Shell object
peerconfig  Peer Configuration
peerinfo    Get information about peers
peers       Discover peers
put         Put data into a pipe message
rdvserver   No description available for this ShellApp
rdvstatus   Display information about rendezvous
recv        Receive a message from a pipe
search      Discover jxta advertisements
send        Send a message into a pipe
set         Set an environment variable
setenv      Set an environment variable
share       Share an advertisement
Shell       JXTA Shell command interpreter
sql         Issue an SQL command (not implemented)
sqlshell    JXTA SQL Shell command interpreter
talk        Talk to another peer
uninstjar   Uninstalls jar-files previously installed with 'instjar'
version     No description available for this ShellApp
wc          Count the number of lines, words, and chars in an object
who         Display credential information
whoami      Display information about a peer or peergroup
```

7) Lancer JXTA

7.1) Pré requis

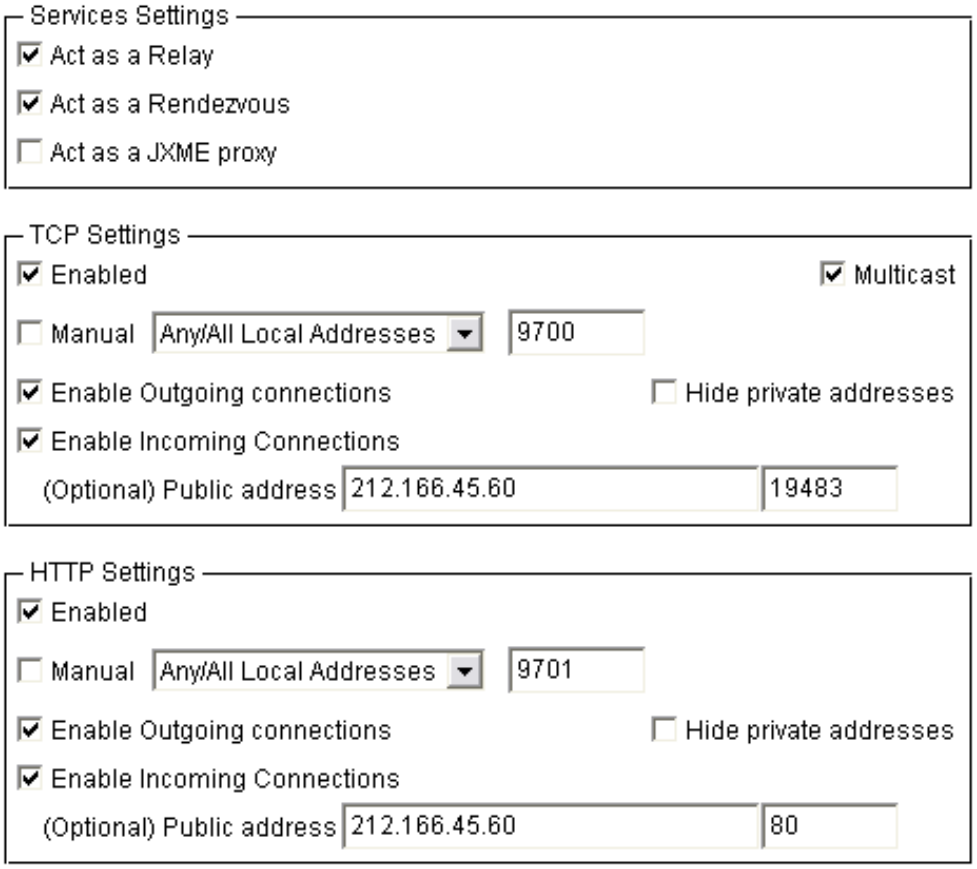
Le développement étant effectué en JAVA, un JRE est évidemment indispensable, dans le cadre de stage il s'agit de sa dernière version, c'est-à-dire la 1.5.
Le site officiel de JXTA (www.jxta.org) propose les sources et JAR en libre téléchargement, il est donc très simple de se les procurer.
Une dizaine de JAR est nécessaire pour pouvoir développer, ceux-ci sont en majorités dédiés à JXTA même, mais d'autres sont plus généraux, notamment ceux spécifiques à SAX et DOM, indispensables pour le traitement du XML omniprésent dans JXTA.
Il suffit alors d'incorporer ces fichiers JAR au projet, ce qui se fait très simplement avec un IDE comme Eclipse.

7.2) Configuration

La première fois qu'une application utilisant la technologie JXTA est lancée, un utilitaire de configuration est exécuté. Il est composé de 3 onglets différents :

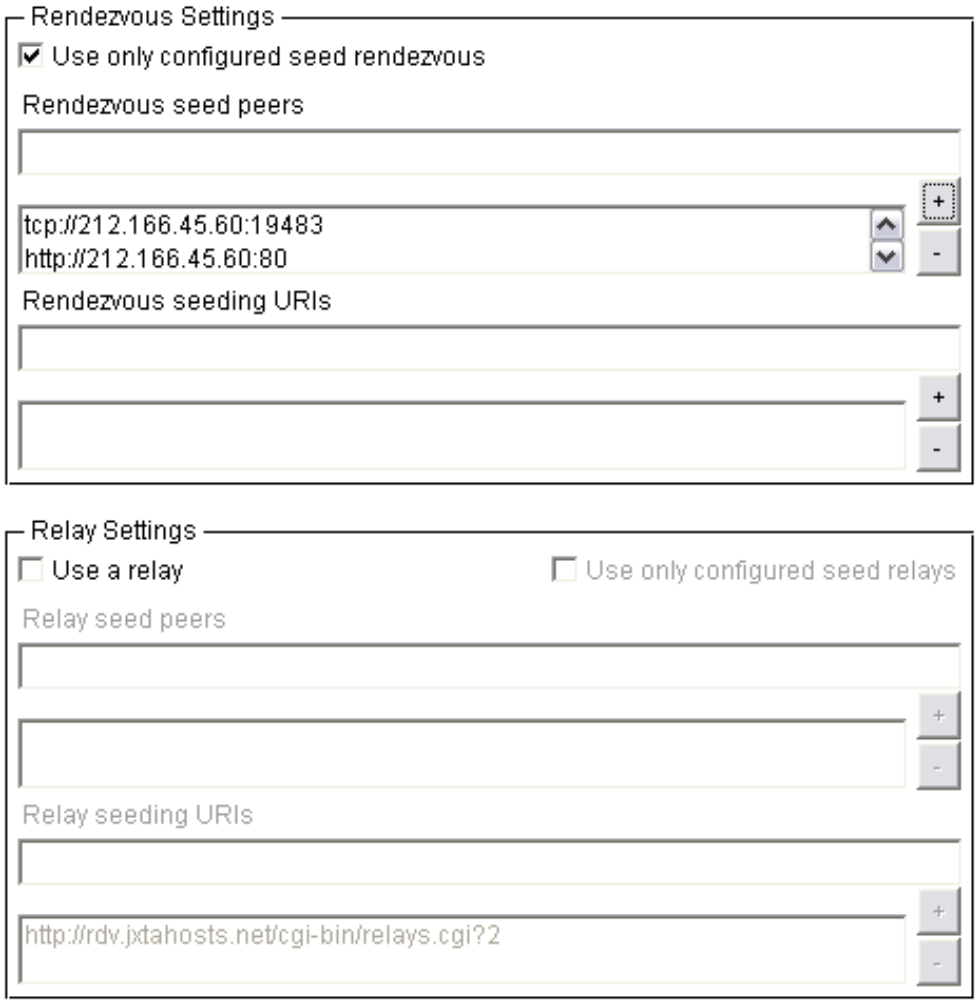
L'onglet basic, permettant d'entrer les informations de bases, c'est-à-dire le nom du peer et son mot de passe. Il offre également la possibilité de fournir un certificat pour ce peer.

L'onglet advanced :



C'est lui qui permet de définir le comportement du peer (relay, rendezvous, …) et de configurer les paramètres TCP et HTTP.
Il est possible, pour chaque protocole, d'autoriser ou non les connexions entrantes et sortantes.
L'outil offre de plus la possibilité de spécifier les adresses/ports publics et privés à utiliser.

L'onglet Rendezvous/Relays permet quant à lui de configurer les différents rendezvous et relays à



utiliser :

Si cet outil est assez pratique pour le développeur, il est bien sûr inacceptable pour un client.
Il serait en effet d'assez mauvais goût de laisser le client compléter cette configuration lui-même, d'autant plus que dans la plupart des cas, il n'y comprendrait pas grand-chose.

JXTA a donc prévu cette difficulté et ce fichier peut être généré automatiquement par programmation.
Il suffit en effet d'obtenir une instance de la classe Configurator (net.jxta.ext.config) laquelle possède une multitude de méthodes permettant d'effectuer la configuration.
Parmi celles-ci : setName, setAddress, setRendezvous, setRelay, … et surtout save prenant en paramètre un objet de type File qui devra être nommé PlatformConfig.

Il y a deux types de configuration possibles :
- Un peer simple (edge peer) : celui-ci peut se trouver ou non derrière un firewall ou du NAT.
Il est recommandé de configurer ce peer pour qu'il utilise TCP en entrée/sortie et HTTP en sortie uniquement. Il devrait également utiliser un relay et un rendezvous.
- Un peer de rendezvous/relay : ce type de peer est prévu pour fournir des services et est directement accessible sur Internet. Il est donc conseillé de le configurer avec TCP en entrée/sortie et HTTP en entrée uniquement. Il faudra de plus lui signaler d'agir en tant que rendezvous et/ou relay.

8) Développement

8.1) HelloWorld

L'exemple classique du développeur, le HelloWorld !
L'application qui va suivre illustre la manière de lancer la plateforme JXTA. Elle va instancier cette plateforme puis afficher le nom et l'ID du peer ainsi que ceux du groupe.
Cette instanciation s'effectue par l'intermédiaire de la méthode newNetPeerGroup() de l'objet statique PeerGroupFactory (net.jxta.peergroup). Celle-ci retourne un objet PeerGroup (net.jxta.peergroup) contenant les informations sur le groupe par défaut : le net peer group.
Cet objet contient les implémentations de différents services de base de JXTA tels que le discovery, le membership ou le service de rendezvous.
Il contient également l'ID et le nom du groupe ainsi que ceux du peer sur laquelle la plateforme est exécutée.
Cette méthode signale toute erreur en lançant une PeerGroupException (net.jxta.exception).
Il ne reste ensuite qu'à afficher les informations souhaitées par l'intermédiaire des méthodes correspondantes de l'objet PeerGroup. Parmi celles-ci : getPeerGroupName(), getPeerName(), getPeerGroupID() et getPeerID().
La plateforme sera enfin arrêtée via la méthode stopApp().

```
PeerGroup netPeerGroup;
try
{
    // Création et démarrage du groupe par défaut de JXTA
    netPeerGroup = PeerGroupFactory.newNetPeerGroup();
}
catch (PeerGroupException e)
{
    // Impossible d'instancier le groupe
    e.printStackTrace();
    System.exit(1);
}

System.out.println("Hello from group "+netPeerGroup.getPeerGroupName());
System.out.println("Group ID = "+netPeerGroup.getPeerGroupID().toString());
System.out.println("Peer name = " +netPeerGroup.getPeerName());
System.out.println("Peer ID = " +netPeerGroup.getPeerID().toString());
netPeerGroup.stopApp();
```

Ce qui pourrait donner :

```
Hello from group NetPeerGroup
Group ID = urn:jxta:jxta-NetGroup
Peer name = suzi
Peer ID = urn:jxta:uuid-59616261646162614A78746150325033F3B
```

Après cette exécution, on peut constater qu'un répertoire .jxta a été créé, contenant le fichier PlatformConfig ainsi qu'un répertoire de cache nommé cm. C'est dans celui-ci que sont sauvés les différentes advertisements et autres fichiers d'index.

8.2) Advertisements

8.2.1) Afficher un advertisement

Les advertisements (voir 5.4) étant omniprésents dans JXTA, il n'est pas rare de devoir les contrôler et de s'assurer de leur contenu, dans un objectif de debuggage.
Cette vérification peut se faire de manière très simple grâce à la méthode getDocument()(présente dans chaque type d'avertisement), renvoyant StructuredTextDocument.
Cette méthode demande un paramètre en entrée qui est en fait le format. Dans notre cas, nous lui spécifierons que nous souhaitons obtenir l'advertisement dans son format d'origine, le XML.
Un objet de type StructuredTextDocument possède quant à lui une méthode sendToStream() qui permet d'envoyer le contenu du document sur le flux spécifiant en paramètre.

```
PipeAdvertisement aPipeAdv = new PipeAdvertisement();
StructuredTextDocument aDoc = (StructuredTextDocument)
aPipeAdv.getDocument(new MIMETYPE("text/xml"));
try
{
    aDoc.sendToStream(System.out);
} catch(Exception e) {}
```

8.2.2) Créer un advertisement depuis un fichier existant

Une tâche courante pour un peer est de créer un advertisement, par exemple pour un pipe ou un service qu'il vient de mettre en place.
Une des possibilités est de créer cet advertisement depuis un fichier XML déjà existant sur le système de fichiers.
Il suffit pour cela de créer un FileInputStream sur ce fichier. Il faut alors utiliser l'objet statique AdvertisementFactory permettant de créer un advertisement depuis ce fichier.

```
PipeAdvertisement myPipeAdvertisement = null;
try
{
nbsp;    FileInputStream is = new FileInputStream("service1.adv");
nbsp;    myPipeAdvertisement = (PipeAdvertisement)
nbsp;    AdvertisementFactory.newAdvertisement(new MimeMediaType("text/xml"), is);
}
catch (Exception e) {}
```

8.3.3) Créer un advertisement à la volée

Le première étape pour cette création est d'appeler la méthode newAdvertisement() de l'objet AdvertisementFactory. Il s'agit d'une de ses versions, celle-ci prenant en paramètre le type d'advertisement à créer.
Une fois l'advertisement créé, les méthodes de celui-ci sont appelées individuellement pour placer les différents attributs aux valeurs passées en paramètres.
Une fois toutes ces méthodes correctement appelées, un objet advertisement valide est obtenu et peut alors être utilisé par le peer.

```
ModuleSpecAdvertisement msa = (ModuleSpecAdvertisement)AdvertisementFactory.
```

```
newAdvertisement(ModuleSpecAdvertisement.getAdvertisementType());

msa.setName("JXTASPEC:JXTA-CH15EX2");
msa.setVersion("Version 1.0");
msa.setCreator("Antho”);
msa.setModuleSpecID(IDFactory.newModuleSpecID(myService1ID));
msa.setSpecURI("");
msa.setPipeAdvertisement(myPipeAdvertisement);
```

8.3) Discovery

L’application consiste à envoyer des requêtes sur le net peer group, à la recherche d’autres peers. Pour chaque réponse reçue, le nom des peers découverts sera affiché.
Suivant la proximité des peers connectés au net peer group, cette recherche peut prendre plus ou moins de temps. Le fichier de configuration de base comprend plusieurs points de rendezvous du groupe par défaut. Le peer va donc les contacter pour obtenir des informations sur les autres membres du groupe.
Des peers présents dans le monde entiers peuvent ainsi être découverts et c’est donc sans réelle surprise que notre premier essai avec un chat JXTA (myJXTA : myjxta.jxta.org) nous a permis de communiquer avec un correspondant chinois !
Si aucun résultat ne venait à arriver, il faudrait vérifier la configuration. Le peer serait en effet probablement derrière un firewall et il faudrait le configurer en conséquence.
Comme déjà expliqué précédemment, un peer JXTA peut interroger son propre cache pour trouver un advertisement déjà reçu. Il doit pour cela utiliser la méthode getLocalAdveretisements().
Si il veut obtenir d’autres advertisements, il envoie une requête à d’autres peers via la méthode getRemoteAdvertisements(). Cette requête peut être envoyée à des peers spécifiques ou à tout le réseau JXTA. Dans la version Java de JXTA, les requêtes sont envoyées en multicast sur le sous-réseau et au peer de rendezvous.
Chaque peer qui reçoit la requête de discovery va l’analyser, et si il trouve une correspondance parmi les advertisements qu’il possède déjà, il va répondre en envoyant cet advertisement.
Il y a deux possibilités pour réceptionner les réponses à ces requêtes.
Soit attendre qu’un peer réponde, puis utiliser la méthode getLocalAdvertisements() pour parcourir les advertisements du cache.
Soit par l’intermédiaire d’un système de notification asynchrone, le Discovery Listener, qui appelle automatiquement la méthode discoveryEvent (recevant l’évènement en paramètre et permettant donc de le traiter) lorsqu’un évènement de type discovery se produit. Il y a pour cela deux solutions, soit enregistrer le listener grâce à la méthode addDiscoveryListener() soit en passant le listener en paramètre à la méthode getRemoteAdvertisements().
Le DiscoveryEvent permet d’obtenir différentes informations. Notamment la réponse associée en utilisant la méthode getResponse() renvoyant un objet de type DiscoveryResponseMessage.
Chacun de ces messages contient les advertisements du peer ayant répondu (un advertisement par peer qu’il connaît), ainsi que d’autres informations comme le nombre de réponses envoyées. Le peer répondant intègre également son propre advertisement dans le message, celui-ci peut être facilement récupéré via la méthode getPeerAdvertisement() du message.
Les autres advertisements peuvent, quant à eux, être récupérés grâce à la méthode getAdvertisement().
Le service de discovery est obtenu en appelant la méthode getDiscoveryService de l’objet PeerGroup.
Sa méthode getRemoteAdvertisements() prend en fait cinq arguments, permettant de créer le paquet de discovery :

- String peer ID : ID du peer à qui doit être envoyé la requête. Si null, elle est propagée sur tout le réseau et est envoyée au rendezvous.

- int type : Type de requête, la valeur doit être une des constantes suivantes : DiscoveryService.Peer, DiscoveryService.Group ou DiscoveryService.Adv (= tout autre type d'advertisements, tels que les pipes advertisements par exemple).
- String attribute : nom d'un attribut pour limiter la recherche.
- String value : valeur de l'attribut.
- int threshold : seuil indiquant le nombre maximum de réponses.

```
<xs:element name="DiscoveryQuery" type="jxta:DiscoveryQuery"/>

<xs:complexType name="DiscoveryQuery">
    <!-- this should be an enumeration -->
    <xs:element name="Type" type="xs:string"/>
    <xs:element name="Threshold" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="PeerAdv" type="jxta:PA" minOccurs="0"/>
    <xs:element name="Attr" type="xs:string" minOccurs="0"/>
    <xs:element name=vValue" type="xs:string" minOccurs="0"/>
</xs:complexType>
```

La portée de la recherche peut être limitée en spécifiant un couple attribut/valeur, seuls les advertisements correspondant à ce couple seront renvoyés. Le nom de l'attribut doit obligatoirement correspondre au nom d'un tag dans le fichier XML correspondant. La chaîne de caractères pour le paramètre value peut utiliser un wildcard (par exemple *) pour déterminer la correspondance.

Exemple : la requête suivante va limiter la recherche aux peers dont le nom contient le mot test :
discovery.getRemoteAdvertisements(null, DiscoveryService.PEER, "Name", "*test*", 5);
Un seuil de cinq est également fourni, limitant ainsi à cinq le nombre de réponses possibles par message. Il n'y a aucune garantie quant au nombre de réponses, elles peuvent varier de 0 à l'infini.
Le format de celles-ci est :

```
<xs:element name="DiscoveryResponse" type="jxta:DiscoveryResponse"/>

<xs:complexType name="DiscoveryResponse">
    <!-- this should be an enumeration -->
    <xs:element name="Type" type="xs:string"/>
    <xs:element name="Count" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="PeerAdv" type="xs:anyType" minOccurs="0">
        <xs:attribute name="Expiration" type="xs:unsignedLong"/>
    </xs:element>
    <xs:element name="Attr" type="xs:string" minOccurs="0"/>
    <xs:element name="Value" type="xs:string" minOccurs="0"/>
    <xs:element name="Response" type="xs:anyType" maxOccurs="unbounded">
        <xs:attribute name=vExpiration" type="xs:unsignedLong"/>
</xs:element>
</xs:complexType>
```

Voici un exemple de code permettant de découvrir les peers en utilisant un listener :

```
import java.util.Enumeration;
import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;
```

```
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.PeerAdvertisement;

public class DiscoveryDemo implements DiscoveryListener
…

// Obtention du service de discovery depuis le net peer group obtenu précédemment
DiscoveryService discovery = netPeerGroup.getDiscoveryService();
discovery.addDiscoveryListener(this);

// Fonction envoyant les requêtes dans une boucle
while (true)
{
    System.out.println(“Envoi d’un message de discovery”);
    discovery.getRemoteAdvertisements(null, DiscoveryService.PEER, null, null, 5);
    try
    {
   nbsp;    Thread.sleep(60 * 1000);
    }
    catch(Exception e) {}
}

// Fonction appelée en cas de Discover Event
public void discoveryEvent(DiscoveryEvent ev)
{
    // On récupère le message correspondent à l’évènement
    DiscoveryResponseMsg res = ev.getResponse();
    String name = "unknown";
    PeerAdvertisement peerAdv = res.getPeerAdvertisement();
    if (peerAdv != null)
    {
       String name = peerAdv.getName();
       System.out.println("Réception de "+res.getResponseCount()+" advertisements du
       peer : "+name) ;

       PeerAdvertisement adv = null;
       Enumeration en = res.getAdvertisements();
       if (en != null)
       {
          // On affiche les peers connus par le peer qui nous a répondu
           while (en.hasMoreElements())
          {
              adv = (PeerAdvertisement) en.nextElement();
             System.out.println (" Peer name = " + adv.getName());
           }
       }
    }
}
```

Le lancement de cet exemple pourrait donner le résultat suivant :
Envoi d'un message de discovery
Réception de 5 réponses du peer : unknown
Peer name = suz
Peer name = jsoto-2K
Peer name = peertopeer
Peer name = JXTA.ORG 237
Peer name = Frog@SF05

Le Group Discovery se déroule exactement de la même façon.
Seules deux légères variations sont nécessaires :

- Il faut spécifier qu'on effectue cette recherche sur les groupes et non sur les peers :
discovery.getRemoteAdvertisements(null, DiscoveryService.GROUP, null, null, 5);

- Il faut également signaler que ce sont maintenant des PeerGroupAdvertisement qui sont reçus : adv = (PeerGroupAdvertisement) en.nextElement();

Comme expliqué ci-dessus, il est également possible d'effectuer la recherche uniquement sur le cache local. Dans ce cas, aucun paquet n'est envoyé sur le réseau. Tous les advertisements se trouvant dans le cache et répondant au type, attribut et valeur passés en paramètres à la méthode getLocalAdvertisements() sont renvoyés dans une énumération. Celle-ci sera ensuite parcourue simplement par l'intermédiaire d'une boucle.

```
Enumeration localEnum;
try
{
    localEnum = DiscoveryService.getLocalAdvertisements(DiscoveryService.PEER,
    "Name", "*test*");
if (localEnum != null)
    {
        while (localEnum.hasMoreElements())
        {
            Element elem = localEnum.nextElement();
        }
    }
} catch (Exception e) {}
```

Il est bien sûr possible de réaliser cette tâche manuellement en vidant le répertoire du cache (cm) mais il est parfois intéressant de le réaliser par programmation.
Les méthodes flushAdvertisements() et flushAdvertisement ont été implémentées à cet effet.

Exemple : flushAdvertisement(aPipeAdvertisement.getID().toString(), DiscoveryService.ADV);

Pour information, voici la structure classique du cache JXTA :

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
      <Id>
       urn:jxta:uuid-
59616261646162614A757874614D504725184FBC4E5D498AA0919F662E400
28B04
      </Id>
      <Type>
       JxtaUnicast
      </Type>
      <Name>
       PipeExample
      </Name>
</jxta:PipeAdvertisement>
```

8.4) Connexion au rendezvous

Un des intérêts principal de JXTA est bien sûr d'utiliser Internet.
Il peut donc être intéressant d'attendre d'être connecté à un point de rendezvous situé sur la toile avant d'envoyer des messages de discovery. Ceux-ci seront en effet propagés sur Internet grâce au rendezvous, les chances de réponses étant alors largement accrues.

8.4.1) Par polling

Pour ajouter cette fonctionnalité, il faut commencer par récupérer le service de rendezvous depuis le net peer group. Celui-ci est donné par la méthode getRendezVousService() de la classe PeerGroup.
Il suffit alors de boucler sur la méthode isConnectedToRendezVous() qui retourne un boolean reflétant le statut de la connexion.

Le code ressemble alors à ceci :

```
RendezVousService rdv = netPeerGroup.getRendezVousService();
System.out.printl("Attente de connexion au rendezvous...");
while (!rdv.isConnectedToRendezVous())
{
    try
    {
       Thread.sleep(2000);
    }
    catch (InterruptedException ex) {}
}
```

8.4.2) Notification asynchrone

Une autre possibilité est d'implémenter l'interface RendezVousListener.
Il suffit alors d'implémenter la méthode correspondante : rendezvousEvent(). Celle-ci sera appelée automatiquement lorsqu'un évènement lié au rendezvous a lieu.
Il peut s'agir d'un évènement de :

- Connexion (RdvConnect) : lorsque le peer est connecté au rendezvous.
- Déconnexion (RdvDisconnect) : le peer est déconnecté du rendezvous.
- Reconnexion (RdvReconnect)
- Echec (RdvFailed)

```
public synchronized void rendezvousEvent(RendezvousEvent event)
{
    if(event.getType() == event.RDVCONNECT || event.getType() == event.RDVRECONNECT)
System.out.println("Connecté au rendezvous");
}
```

8.5) Publication d'un advertisement

Le service de discovery expliqué ci-dessus ne se limite pas aux fonctionnalités énoncées.
En effet, pour pouvoir découvrir des advertisements, il faut bien sûr que ceux-ci aient été préalablement publiés.

Il existe quatre méthodes permettant cette publication :

```
public void publish(Advertisement advertisement, int type);
public void publish(Advertisement adv, int type, long lifetime, long lifetimeForOthers);
public void remotePublish(Advertisement adv, int type);
public void remotePublish(Advertisement adv, int type, long lifetime);
```

Ces méthodes peuvent donc être groupées suivant leur portée : locale ou distante.
Chaque advertisement publié possède une durée de vie par défaut de un an, mais celle-ci peut également être spécifiée grâce à une méthode polymorphe de la méthode.

8.5.1) Publication locale

Dans le cas d'une publication locale, l'advertisement est placé dans le cache du peer émetteur et est envoyé en mutlicast sur le réseau local.

```
DiscoveryService.publish(myAdvertisement, DiscoveryService.ADV);
```

8.5.2) Publication distante

Un advertisement publié de manière distante sera ajouté au cache du peer, publié sur le réseau local et surtout, envoyé aux différents rendezvous.
Cette technique est bien sûr à utiliser pour faire connaître son service sur tout le réseau, Internet y compris.

```
DiscoveryService.remotePublish(myAdvertisement, DiscoveryService.ADV);
```

8.6) Les groupes de peers

8.6.1) Création d'un groupe

Les groupes de peers (5.3) sont un des concepts de base d'un réseau peer to peer.
Ils permettent de regrouper des peers autour d'un même type de services.
L'exemple suivant va décrire la marche à suivre pour créer un tel groupe et publier des advertisements pour celui-ci, afin qu'il soit connu sur le réseau.
Il faut commencer par créer un ModuleImplAdvertisement, celui-ci contient tous les services de base que

chaque groupe se doit d'implémenter. Il est obtenu par l'intermédiaire de la méthode getAllPurposePeerGroupImpleAdvertisement() de l'objet de type PeerGroup, instanciant le groupe de base : le Net Peer Group. Il s'agit du groupe obtenu lors du lancement de la plateforme JXTA (voir 8.1). La méthode newGroup du groupe par défaut est alors appelée pour créer le nouveau groupe.

PeerGroup pg = myGroup.newGroup(null, implAdv, "Groupe 1", "Groupe de test");

Quatre arguments sont passés à cette méthode :

- Un ID (PeerGroup) : il s'agit de l'ID du groupe qui va être créé. Si il est null, cet ID sera généré automatiquement.
- L'advertisement (Advertisement) de ce groupe.
- Le nom (String) du groupe.
- La description (String) du groupe.

L'appel à cette méthode newGroup va publier le groupe fraîchement créé dans le cache local.
Il faut ensuite procéder à une publication distante pour faire connaître ce groupe à l'extérieur du réseau local. Cela est réalisé très simplement grâce à une des méthodes expliquées au chapitre précédent (8.5.2) : remotePublish().

```
PeerGroupAdvertisement adv;
try
{
    // Création du groupe
    ModuleImplAdvertisement implAdv = myGroup.getAllPurposePeerGroupImplAdvertisement();
    PeerGroup pg = myGroup.newGroup(null, implAdv, "PubTest", "testing group adv");
    adv = pg.getPeerGroupAdvertisement();
    PeerGroupID GID = adv.getPeerGroupID();
    System.out.println(" Group = " +adv.getName() + " Group ID = " + GID.toString());
}
catch (Exception e) { }
try
{
    // Publication de l'advertisement
    DiscoveryService.remotePublish(adv);
}
catch (Exception e) { }
```

8.6.2) Rejoindre un groupe

L'exemple suivant décrit la méthode à adopter pour joindre un groupe.
Le Membership Service permet de ''postuler'' pour rejoindre un groupe, de joindre celui-ci et de prolonger son bail dans ce groupe.
Ce service permet également à un peer de s'établir une identité dans le groupe.
Un credential est créé en correspondance avec cette identité, permettant de prouver que le peer possède réellement cette identité. Cette notion d'identité est utilisée par les services pour déterminer les droits de chaque peer.
Ce credential est représenté par un jeton et doit être présent dans chaque message envoyé. L'adresse de l'émetteur est placée dans l'enveloppe du message JXTA et est comparée avec l'identité de l'émetteur, qui se trouve dans le credential.

La séquence pour établir une identité est la suivante :

- Postuler : le peer fournit un credential initial au Membership Service. Celui-ci sera utilisé par le service pour déterminer quelle méthode d'authentification sera nécessaire pour établir l'identité de ce peer. Si le service autorise cette authentification, un objet d'authentificateur adéquat est retourné.
- Joindre : l'authentificateur complété est retourné au service et l'identité de ce peer est ajustée en fonction du nouveau credential disponible avec l'authentificateur.
- Prolonger : quelle que soit l'identité établie pour ce peer, elle revient automatiquement à l'identité « nobody » à intervalle régulier. La séquence doit donc être exécutée à nouveau pour prolonger son adhésion au groupe.

Le credential fournit donc deux pièces importantes : la méthode d'authentification demandée et l'identité qui sera fournie à cette méthode.
Toutes les méthodes d'authentification ne se basent pas sur l'identité.

Pour joindre un groupe donné, il faut donc commencer par générer le credential pour le peer :

AuthenticationCredential authCred = new AuthenticationCredential(grp, null, creds);

Ce constructeur requière trois arguments :

- Le groupe (PeerGroup) : il s'agit du groupe pour lequel le credential est créé.
- La méthode d'authentification (String) : celle-ci sera demandée lorsque le credential sera envoyé au Membership Service du groupe.
- Informations d'identité (Element) : il s'agit d'informations optionnelles sur l'identité du peer, qui seront utilisées par la méthode d'authentification.

Il faudra ensuite récupérer le Membership Service du groupe que l'on souhaite rejoindre via la méthode getMembershipService(). Il suffit alors d'appeler la méthode apply() de ce service en lui passant le credential précédemment généré en paramètre.
Cet appel va retourner un objet de type Authentificator qui va permettre de vérifier le succès de l'opération. La technique pour ensuite compléter cet objet est unique à chaque méthode d'authentification. La seule fonction commune est isReadyForJoin() qui est assez explicite.
Après avoir postulé, l'étape suivante est de rejoindre le groupe. Cela s'effectuer simplement en appelant la méthode join() du service d'adhésion.
Il est important de remarquer que lorsqu'un peer rejoint un groupe, il cherche automatiquement un rendezvous pour ce groupe. Si il n'en trouve pas, il deviendra dynamiquement un rendezvous pour ce groupe.

```
StructuredDocument creds = null;
try
{
    // Génération du credential
    AuthenticationCredential authCred = new AuthenticationCredential( grp, null, creds );

    MembershipService membership = grp.getMembershipService();
    Authenticator auth = membership.apply( authCred );
    if (auth.isReadyForJoin())
    {
       Credential myCred = membership.join(auth);
```

```
        System.out.println("Adhésion au groupe " +grp.getPeerGroupName());

        // Affichage du credential
        System.out.println("Credential : ");
        StructuredTextDocument doc = (StructuredTextDocument) myCred.getDocument
        (new MimeMediaType("text/plain"));
        StringWriter out = new StringWriter();
        doc.sendToWriter(out);
        System.out.println(out.toString());
        out.close();
    }
    else
    {
        System.out.println("Impossible de joindre le groupe");
    }
}
catch (Exception e) { }
```

8.7) Les pipes

8.7.1) Réception

Cet exemple illustre la méthode à utiliser pour établir une communication entre deux peers en utilisant les pipes (5.5). Les pipes fournissent un canal de communication unidirectionnel et asynchrone entre deux peers.
La classe PipeService définit un ensemble d'interfaces pour créer et accéder à des pipes, à l'intérieur d'un groupe de peers.
Une application qui souhaite établir une communication en réception avec d'autres peers doit créer un tube en entrée et lier celui-ci à un pipe advertisement spécifique.
L'application publie ensuite cet advertisement afin que d'autres applications puissent créer un tube en sortie, permettant d'envoyer des messages au tube correspondant en entrée.
Les pipes sont identifiés de manière unique par l'intermédiaire de pipes ID (UUID). Chaque pipe advertisement contient l'identifiant du pipe correspondant. C'est cet ID qui est utilisé pour associer les pipes en entré et en sortie.
Il est important de signaler qu'un pipe est indépendant de tout peer ou emplacement géographique. L'association entre le pipe et sa localisation physique est effectuée par l'intermédiaire du Pipe Binding Protocol (PBP). Il s'agit en fait d'un mécanisme de recherche permettant de trouver à chaque instant les peers qui utilisent un pipe donné.

Ce premier exemple va donc créer un pipe en entrée et écouter sur celui-ci en l'attente de tout message entrant.
La première étape est de créer une classe implémentant l'interface PipeMsgListener.
Il faut ensuite récupérer le Pipe Service par l'intermédiaire du net peer group et sa méthode getPipeService(). Il est nécessaire de créer un pipe advertisement, celui-ci pourra être généré à la volée ou être construit depuis un fichier (voir 8.2).
Le pipe est alors créé grâce à la méthode createInputPipe() du Pipe Service. Celle-ci prend deux paramètres en entrée, il s'agit de l'advertisement du pipe et du listener associé (l'objet qui recevra les évènements liés au pipe).
C'est bien sûr la classe qui vient d'être créée qui sera enregistrée comme listener. Sa méthode pipeMsgEvent() (apportée par l'interface PipeMsgListener) sera automatiquement invoquée pour chaque

évènement de type pipeMsgEvent correspondant à la réception d'un message.
L'évènement possède sa méthode getMessage() permettant de renvoyer le message associé. Ce dernier est composé de plusieurs éléments sous forme de couple clé/valeur. Chaque élément peut être extrait via la méthode getMessageElement() prenant en paramètre le nom du tag à récupérer.

```
public class PipeListener implements PipeMsgListener
…
Lancement de la plateforme JXTA, adhésion au Net Peer Group, …
…
// Récupération du Pipe Service
PipeService pipe = netPeerGroup.getPipeService();
try
{
    // Création du pipe advertisement depuis un fichier
    FileInputStream is = new FileInputStream("pipexample.adv");
    PipeAdvertisement pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement
(MimeMediaType("text/xml"), is);
    is.close();
    InputPipe pipeIn = pipe.createInputPipe(pipeAdv, this);
}
catch (Exception e) { }

public void pipeMsgEvent(PipeMsgEvent event)
{
    Message msg=null;
    // Récupération du message associé à l'évènement
    msg = event.getMessage();

    // Récupération des elements du message
    Message.ElementIterator en = msg.getMessageElements();
    MessageElement msgElement = msg.getMessageElement(null, "PipeListenerMsg");

    if (msgElement.toString() == null)
       System.out.println("Réception d'un message vide");
    else
       System.out.println(msgElement.toString());
}
```

Une autre méthode que la notification asynchrone peut être utilisée, il s'agit du polling.
Il suffit pour cela d'appeler la méthode poll() :
Message newMessage = inputPipe.poll(30000);

Le paramètre est un timeout en millisecondes. Le système restera bloqué pendant la durée du timer ou jusqu'à ce qu'un message soit reçu.

8.7.2) Envoi

Cet exemple crée un pipe en sortie et envoie un message sur celui-ci.
La classe créée implémente l'interface OutputPipeListener.

La méthode de création s'utilise de la même façon que pour un tube en entrée, il s'agit de createOutputPipe() du Pipe Service. Les paramètres nécessaires sont l'advertisement du pipe et le listener.
La méthode outputPipeEvent() est appelée automatiquement lorsque les endpoints du pipe sont reliés. La méthode getOutputPipe() de l'évènement permet de récupérer le pipe qui vient d'être créé.
Il faut alors créer le message à envoyer en appelant le constructeur de la classe Message.
Un message est composé d'éléments, le pipe en entrée et en sortie doivent se mettre d'accord sur le format de ces messages.
Pour ajouter un élément au message, il faut commencer par le créer en appelant le constructeur de StringMessageElement, prenant comme paramètres, le nom du tag, la valeur et une éventuelle signature.
Il suffit alors d'appeler la méthode addMessageElement() de l'objet message en lui passant l'élément à ajouter en paramètre.
Une fois le message assemblé, il ne reste plus qu'à l'envoyer via la méthode send du pipe, puis éventuellement fermer le tube via sa méthode close.

```
public class PipeOutput implements OutputPipeListener
…
Lancement de la plateforme, adhesion au Net Peer Group, …
…
PipeService pipe = netPeerGroup.getPipeService();
pipe.createOutputPipe(pipeAdv, this);

public void outputPipeEvent(OutputPipeEvent event)
{
    OutputPipe op = event.getOutputPipe();
    Message msg = null;
    try
    {
       // Création du message
       msg = new Message();
       Date date = new Date(System.currentTimeMillis());
       StringMessageElement sme = new StringMessageElement
       (SenderMessage, date.toString() , null);
       msg.addMessageElement(null, sme);
       op.send(msg);
    }
    catch (Exception e) { }
    op.close();
}
```

Pour information, voici la structure du fichier d'advertisement utilisé dans ces exemples :

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
      <Id>
       urn:jxta:uuid-
59616261646162614A757874614D504725184FBC4E5D498AA0919F662E400
28B04
      </Id>
      <Type>
       JxtaUnicast
      </Type>
      <Name>
       PipeExample
      </Name>
</jxta:PipeAdvertisement>
```

Aussi bien le pipe en lecture que celui en écriture utilise ce fichier. Il est important qu'ils utilisent le même pipe ID pour être en mesure de communiquer.

8.8) Les pipes bidirectionnels

8.8.1) Serveur

Comme expliqué au point 5.6, les tubes bidirectionnels permettent d'effectuer des échanges fiables et dans les deux sens.
Dans l'exemple suivant, deux applications vont être créées : une va instancier un JxtaServerPipe et attendre pour des connexions tandis que l'autre va s'y connecter.

Les différentes étapes ayant déjà été expliquées précédemment, elles ne seront pas décrites ici. Il s'agit bien sûr du lancement de la plateforme JXTA, de l'adhésion au Net Peer Group et de la création du pipe advertisement (via un fichier ou à la volée).

Le JxtaServerPipe possède notamment les méthodes suivantes :

- bind : permet d'effectuer le lien entre le JxtaServerPipe et un advertisement spécifique. Les paramètres requis sont le groupe et le pipe advertisement.
- accept : permet de se placer en attente de connexions sur la socket, et d'accepter celles-ci.
- setPipeTimeout : spécifie le timeout pour établir une connexion avec un JxtaBidiPipe. Une valeur de 0 bloque indéfiniment.

```
try
{
   JxtaServerPipe serverPipe = new JxtaServerPipe(netPeerGroup, pipeAdv);
   // On souhaite bloquer jusqu'à ce qu'une connexion soit établie
   serverPipe.setPipeTimeout(0);

   // Attente de connexions
   while (true)
   {
      JxtaBiDiPipe bipipe = serverPipe.accept();
      if (bipipe != null )
      {
         // Si une connexion a été établie, on envoie un message de test
         Message msg = new Message();
         String data = "Message de test";
```

```
            msg.addMessageElement("champs_test",
            new StringMessageElement("champs_test", data, null));
            System.out.println("Envoi du message : "+data);
            bipipe.sendMessage(msg);

        }
    }
}
catch(Exception e) { }
```

On peut remarquer qu'il est ainsi très simple de mettre en place un serveur multi-thread, un thread prenant en charge chaque nouvelle connexion.

8.8.2) Client

Le client va quant à lui devoir créer un JxtaBiDsiPipe et se connecter au serveur.
Ce pipe est créé très simplement en appelant le constructeur correspondant : JxtaBiDiPipe().
Il suffit alors d'appeler la méthode connect et de s'enregistrer comme listener pour tout évènement lié à un message. La classe devra pour cela implémenter l'interface PipeMsgListener qui appellera sa méthode pipeMsgEvent automatiquement.

Les méthodes principales du JxtaBidiPipe sont :

- connect : permet de se connecter à un pipe distant. Il en existe plusieurs versions polymorphes, celle utilisée ici prenant en paramètres : le groupe, le peer ID, le pipe advertisement, un timeout et le listener.
- getMessage : lit un message sur le pipe.
- sendMessage : envoie un message sur le pipe.
- setReliable : permet rendre le tube fiable ou non.

```
class Client implements PipeMsgListener
…
…
try
{
    JxtaBiDiPipe pipe = new JxtaBiDiPipe();
    pipe.setReliable(true);
    pipe.connect(eg.netPeerGroup, null, pipeAdv, 180000, this);
}
catch(Exception e) { }
…
public void pipeMsgEvent(PipeMsgEvent event)
{
    Message msg = null;
    msg = event.getMessage();
    // Traitement du message
    …
}
```

8.9) Les sockets JXTA

La JxtaSocket est un tube bidirectionnel présentant la même interface qu'une socket Java classique. Les JxtaSockets s'utilisent donc de manière très similaire aux sockets de base.
Il est cependant important de noter que les sockets JXTA n'utilisent pas l'algorithme de Nagel. C'est donc le rôle des applications de « vider » les sockets lorsque c'est nécessaire, en utilisant la méthode flush().
Comme précédemment, cet exemple utilise deux applications, une créant le serveur et l'autre le client.

```
try
{
   JxtaServerSocket serverSocket = new JxtaServerSocket(netPeerGroup, pipeAdv, 10);
   // Bloque jusqu'à ce qu'une connexion soit disponible
   serverSocket.setSoTimeout(0);

   while (true)
   {
      Socket socket = serverSocket.accept();
      if (socket != null)
      {
         OutputStream os=socket.getOutputStream() ;
         InputStream is=socket.getInputStream();
         …
      }
   }
}
```

On constate que le code n'apporte aucune surprise et est bien connu.
Un avantage considérable est le fait de pouvoir créer des flux sans aucune difficulté. Il ne faut cependant pas oublier d'appeler les méthodes flush pour éviter toute mauvaise surprise.

Du côté du client, le code ressemble à ceci :

```
try
{
   JxtaSocket socket = new JxtaSocket(netPeerGroup, null, pipeAdv, 30000, true);
   socket.setOutputStreamBufferSize(65536); // Optionnel
   OutputStream os=socket.getOutputStream();
   ObjectOutputStream oos=new ObjectOutputStream(os);
   …
   …
   oos.flush();
   oos.close();
} catch(Exception e) { }
```

Le constructeur prenant les mêmes paramètres que la méthode connect des pipes (le groupe, le peer ID, l'advertisement et le timeout). La seule différence étant le boolean passé en dernier permettant de spécifier si la communication est fiable ou non.

8.10) Le multicast

JXTA n'a pas fini de nous étonner et introduit encore un autre moyen de communication.
La JxtaMulticastSocket va ainsi permettre de joindre des groupes multicast et d'envoyer d'une traite un message à tout un groupe.
Les avantages sont considérables, surtout pour une application de conférence comme celle développée dans le cadre de ce stage !
Les sockets utilisent des DatagramPacket comme moyen de communication et le contenu de ceux-ci doit être spécifié sous forme de bytes.
Les datagrammes transportent également un peerID qui est représenté par une InetAddress. Celle-ci est toujours composée du couple hôte/adresse IP, l'hôte étant l'ID du peer et l'adresse IP toujours égale à 0.0.0.0 (étant donné qu'elle n'a aucun sens dans un réseau JXTA).
La JxtaMulticastSocket est construite au dessus des tubes propagés et dérive de la MulticastSocket de base de Java.

Voici un exemple d'utilisation :

```
String msg = "Hello World";
MulticastSocket socket = new JxtaMulticastSocket(peergroup, pipeAdv);

// Envoi
DatagramPacket envoi = new DatagramPacket(msg.getBytes(), msg.length());
socket.send(envoi);
…
// Reception
byte[] buf = new byte[1000];
DatagramPacket recv = new DatagramPacket(buf, buf.length);
socket.receive(recv);
...
socket.close();
```

Une application peut également répondre directement à un message reçu sans envoyer la réponse à tout le groupe :

```
DatagramPacket envoi = new DatagramPacket(msg.getBytes(), msg.length());
envoi.setAddress(recv.getAddress());
socket.send(envoi);
```

8.11) Les modules

8.11.1) Introduction

Les groupes de peers fournissent les fonctionnalités de bases nécessaires pour un système peer to peer. Il est cependant possible de compléter celles-ci en ajoutant des services additionnels, utilisables par tous les peers.
Un module est une des possibilités pour fournir un service. Il s'agit simplement d'une fonctionnalité destinée à être téléchargée, permettant à un peer d'instancier un nouveau comportement.
Par exemple, pour rejoindre un groupe particulier, un peer aurait à apprendre une méthode de recherche spécifique à ce groupe. Il lui faudrait pour cela instancier le module correspondant.
Ces modules sont annoncés par différents advertisements :

- Module Class : cet advertisement est utilisé pour annoncer l'existence d'une fonctionnalité. Chaque module class est identifié par un identifiant unique : le ModuleClassID.

- Module Specification : cette spécification contient toutes les informations nécessaires pour accéder et invoquer le module. Elle peut également contenir un pipe advertisement à utiliser pour communiquer avec le service. Il peut exister plusieurs modules specifications pour un module class donné. Chaque spécification étant identifiée par un ModuleSpecID. Le ModuleSpecID contenant le ModuleClassID indiquant le module class associé.
- Module Implementation : il s'agit de l'implémentation d'une spécification donnée. Il peut exister plusieurs implémentations pour une spécification donnée. Chaque implémentation contient le ModuleSpecID de la spécification associée.

Comme exemple, considérons le discovery service de base du noyau JXTA. Il possède un unique ModuleClassID l'identifiant comme étant un service de découverte. Il peut exister différentes spécifications pour ce service, chacune utilisant une technique de découverte différente, par exemple en fonction de la taille du réseau. Chaque spécification possède son propre ModuleSpecID pointant ver le ModuleClassID du service. Et enfin pour chaque spécification il peut exister de multiples implémentations contenant le ModuleSpecID correspondant. Ce système a été instauré car un des objectifs du peer to peer est que n'importe quel peer (et donc sur n'importe quel environnement, utilisant n'importe quel langage) puisse implémenter la même spécification. Les implémentations peuvent ainsi être écrites dans différents langages, tout en fournissant le même service.

8.11.2) Les advertisements

Le Module Class Advertisement est composé de trois éléments :

- Le MCID : ModuleClassID ;
- Le nom ;
- La description.

```
<xs:complexType name='MCA">
      <xs:element name="MCID" type="JXTAID'/>
      <xs:element name="Name" type="xs:string" minOccurs='0"/>
      <xs:element name="Desc" type="xs:anyType' minOccurs="0"/>
</xs:complexType>
```

Cet advertisement est fort simple, l'élément important étant bien sûr le MCID qui sera utilisé par la spécification et l'implémentation.

Une fois que le Module Class Advertisement a été publié sur le réseau, il doit normalement être suivi par un Module Specification Advertisement. Un de ses objectifs est donc de fournir des informations sur le module class.

Cet advertisement est composé des éléments suivants :

- Le nom ;
- La description ;
- MSID : ModuleSpecID ;
- CRTR : une chaîne de caractères représentant l'auteur de la spécification ;
- SURI : une URI qui pointe vers une spécification actuelle ;
- Vers : une chaîne de caractères représentant la version de la spécification ;
- Parm : les paramètres qui peuvent être recherchés par le receveur de l'advertisement ;
- Pipe : un pipe advertisement qui peut être utilisé pour communiquer avec un peer qui implémente cette spécification ;
- Proxy : un ModuleSpecID optionnel d'un module qui peut être utilisé pour communiquer avec le

module de cette spécification ;
- Auth : le ModuleSpecID d'un module qui peut être requis pour l'authentification avant l'utilisation de cette spécification.

```
<xs:complexType name='MSA">
      <xs:element name="MSID" type="JXTAID'/>
      <xs:element name="Name" type="xs:string" min0ccurs='0"/>
      <xs:element name="Crtr" type="xs:string" min0ccurs='0"/>
      <xs:element name="SURI" type="xs:anyURI" min0ccurs='0"/>
      <xs:element name="Vers" type="xs:string"/>
      <xs:element name="Desc" type="xs:anyType' min0ccurs="0"/>
      <xs:element name="Parm" type="xs:anyType' min0ccurs="0"/>
      <xs:element name="PipeAdvertisement'
  type='jxta:PipeAdvertisement" min0ccurs='0"/>
      <xs:element name="Proxy" type='xs:anyURI' min0ccurs="0"/>
      <xs:element name="Auth" type="JXTAID' min0ccurs="0'/>
</xs:complexType>
```

Une fois que l'implémentation d'une spécification a été créée, il faut publier un Module Implementation Advertisement. Ce dernier est utilisé pour annoncer aux peers où trouver l'implémentation.

Les éléments de cet advertisement sont les suivants :

- Le nom ;
- La description ;
- Proxy : un élément utilisé pour garder une URL à travers laquelle la communication pourra être dirigée ;
- MSID : ModuleSpecID ;
- Comp : un élément avec les informations requises à propos de l'environnement sur lequel cette implémentation peut s'exécuter ;
- PURI : la localisation du package, si il n'est pas trouvé dans le code de la machine du client ;
- Code : informations nécessaires pour qu'un peer puisse charger et exécuter le code. Il peut s'agir du code complet ;
- Parm : paramètres pour l'implémentation ;
- Prov : une chaîne de caractères contenant les informations sur le fournisseur de cette implémentation.

```
<xs:complexType name='MIA">
      <xs:element name="MCID" type="JXTAID'/>
      <xs:element name="Comp" type="xs:anyType'/>
      <xs:element name="Code" type="xs:anyType'/>
      <xs:element name="PURI" type="xs:anyURIv min0ccurs='0"/>
      <xs:element name="Prov" type="string' min0ccurs="0'/>
      <xs:element name="Desc" type="xs:anyType' min0ccurs="0"/>
      <xs:element name="Parm" type="xs:anyType' min0ccurs="0"/>
</xs:complexType>
```

8.11.3) Programmation

Dans cet exemple, un ModuleClassID est créé et publié dans un an ModuleClassAdvertisement. Un ModuleSpecID (basé sur le ModuleClassID) est ensuite créé et ajouté dans un ModuleSpecAdvertisement. Un pipe advertisement est ensuite ajouté dans le ModuleSpecAdvertisement qui est ensuite publié.

Les autres peers peuvent alors découvrir cet advertisement, en extraire le pipe advertisement et communiquer avec le service.

La première partie est celle du serveur, c'est lui qui va créer le service, ouvrir un tube en entrée, effectuer les différentes publications et se mettre à l'écoute des différents messages.

Le ModuleClassAdvertisement est créé par l'intermédiaire de l'objet AdvertisementFactory et sa méthode newAdvertisement() prenant en paramètre le type d'advertisement requis.
Il faut ensuite inititialiser cet advertisement en appelant ses différentes méthodes telles que setName, setDescription, setModuleClassID, etc. Le ModuleClassID est quant à lui généré via la méthode newModuleClassID() de l'objet statique IDFactory.
Il suffit alors de publier cet advertisement grâce aux méthodes maintenant bien connues du Discovery Service : publish() et remotePublish().
La technique est identique pour la création du ModuleSpecAdvertisement.
Il suffira alors de créer le pipe advertisement depuis un fichier, comme expliqué au chapitre 8.2.2.

Le code pourrait donc ressembler à ceci :

Lancement de la plateforme JXTA, adhésion au Net Peer Group, récupération du service de discovery, etc … Voir chapitres précédents.

```
// Création du ModuleClassAdvertisement
ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement)
AdvertisementFactory.newAdvertisement(ModuleClassAdvertisement.getAdvertisementType());
mcadv.setName("JXTAMOD:JXTA-EX1");
mcadv.setDescription("Exemple de service");

// Génération du ModuleClassID
ModuleClassID mcID = IDFactory.newModuleClassID();
mcadv.setModuleClassID(mcID);

// Publication locale et distante de l'advertisement une fois qu'il est créé
discovery.publish(mcadv);
discovery.remotePublish(mcadv);

// Création du ModuleSpecAdvertisement
ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)
AdvertisementFactory.newAdvertisement(ModuleSpecAdvertisement.getAdvertisementType());
mdadv.setName("JXTASPEC:JXTA-EX1");
mdadv.setVersion("Version 1.0");
mdadv.setCreator("Antho");
mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
mdadv.setSpecURI("http://www.jxta.org/Ex1");

// Création du Pipe Advertisement
PipeAdvertisement pipeadv = null;
try
{
    FileInputStream is = new FileInputStream("pipeserver.adv");
    pipeadv = (PipeAdvertisement)
    AdvertisementFactory.newAdvertisement(MimeMediaType.XMLUTF8, is);
```

```
    is.close();
}
catch (Exception e) { }

// Ajout du Pipe Advertisement dans le SpecAdvertisement et publication
mdadv.setPipeAdvertisement(pipeadv);
discovery.publish(mdadv);
discovery.remotePublish(mdadv);
```

Création du pipe et mise en écoute des messages (voir chapitre 8.7)

Le client quant à lui recherche le service nommé JXTA-EX1 et récupère l'advertisement du pipe correspondant. Il crée alors un tube en sortie sur celui-ci et envoie différents messages.
Cela s'effectue par l'intermédiaire du service de Discovery évoqué au chapitre 8.3.

```
discovery.getRemoteAdvertisements(null,DiscoveryService.ADV, "Name", "JXTASPEC:JXTA-EX1", 1, null);
```

Lors de la découverte du service, on récupère le pipe advertisement et on crée un tube en sortie sur celui-ci :

```
ModuleSpecAdvertisement mdsadv = (ModuleSpecAdvertisement) en.nextElement();
PipeAdvertisement pipeadv = mdsadv.getPipeAdvertisement();
```

9) La sécurité

9.1) Introduction

Depuis le début de son élaboration, JXTA a été étudié suivant une certaine politique de sécurité.
Il s'agit d'un gros avantage car il a déjà été prouvé par le passé, qu'une application dont la sécurité n'est pas intégrée dès le départ dans le processus de développement, est beaucoup moins sécurisée et donc plus simple à attaquer.
La sécurité d'un réseau peer to peer n'est bien sûr pas évidente à implémenter et cela du au fait de sa propre architecture. En effet, l'environnement étant distribué, il n'existe pas d'autorité centralisée. Le chemin parcouru pour utiliser un service chez un peer distant n'est pas défini, il pourrait très bien passer par cinq peers intermédiaires dont les intentions pourraient être douteuses.
Les besoins en sécurité de JXTA sont très similaires aux systèmes traditionnels mais sa nature décentralisée rend plus difficile l'implémentation de la confidentialité, l'intégrité ou encore la non répudiation.

Le projet de sécurité JXTA, représenté par le security toolkit a trois objectifs principaux :

- Fournir des pipes sécurisés : cela est rendu possible grâce à l'option JxtaUnicastSecure qui assure que toutes les communications entre les peers seront chiffrées. Il s'agit d'une forme semblable au SSL, fournissant une connexion sécurisée transparente ;
- Assurer l'authenticité et l'intégrité des données : cette fonctionnalité est fournie par les algorithmes de chiffrement et différentes méthodes de checksum ;
- Assurer la non répudiation des transactions : le toolkit fournit pour cela la possibilité de signer les messages.

Le security toolkit (jxta.security) de JXTA fournit des mécanismes de base pour définir des clés secrètes et RSA, effectuer des chiffrements RC4, créer des messages digests au format SHA-1 et MD5, et créer des signatures digitales sécurisées.
Les APIs cryptographiques de JXTA sont fort différentes de celles du J2SE pour la bonne et simple raison que JXTA est basé sur le modèle des Java Card. Cela est bien sûr indispensable étant donné que les applications doivent pouvoir être exécutées sur différentes plateformes, et notamment J2ME (appareils mobiles et sans fils tels que les PDAs ou les smart phones).

9.2) Les clés

Le premier pas pour utiliser l'infrastructure de sécurité JXTA est d'obtenir une paire de clés.
Pour rappel, il en existe deux types :

- Les clés secrètes : ce sont des clés symétriques qui sont utilisées pour crypter et décrypter l'information. Elles sont dites symétriques car c'est la même clé qui est utilisée des deux côtés de la communication. JXTA utilise les clés secrètes pour les ciphers RC4.
- Les clés publiques et privées : il s'agit cette fois de clés asymétriques. Les données qui sont chiffrées (ou signées) avec une clé publique ne peuvent être que déchiffrée la clé privée correspondante et inversement. JXTA supporte les algorithmes basés sur les paires de clés RSA.

La différence principale entre ces deux types de clés est la façon dont elles sont partagées.
Dans le cas du chiffrement asymétrique, il suffit de partager sa clé publique. Ainsi toute personne qui souhaite vous contacter crypte les données avec votre clé publique. Vous serez alors le seul à pouvoir décrypter le message car vous êtes le seul à posséder la clé privée correspondante. De plus, lorsque vous chiffrez un message avec votre clé privée, il est très simple de s'assurer que le message vient bien de vous étant donné qu'il sera décrypté avec votre clé publique.
Les clés secrètes doivent quant à elles, comme indiqué dans leur nom, rester secrètes.
Seules les participants à la communication peuvent posséder cette clé, différents moyens doivent donc être mis en œuvre pour générer cette clé et s'assurer qu'elle est identique des deux côtés.

Toutes les clés utilisées dans JXTA implémentent l'interface Key (jxta.security.cipher.Key). Celle-ci définit les méthodes de base qui sont utilisées par toutes les clés.
Les clés publiques et privées implémentent respectivement les interfaces PublicKey et PrivateKey (jxta.security.publickey). En fonction du type de clés, elles peuvent encore implémenter d'autres interfaces telles que par exemple : RSAPublicKey (jxta.security.publickey.RSAPublicKey).

La classe KeyBuilder permet de créer des objets Key. Les types suivants sont disponibles :

```
public static final byte TYPE_RSA_PUBLIC = 1;
public static final byte TYPE_RSA_PRIVATE = 2;
public static final byte TYPE_RSA = (TYPE_RSA_PUBLIC+TYPE_RSA_PRIVATE);
public static final byte TYPE_DES = 4;
public static final byte TYPE_RC4 = 8;
```

La taille de la clé doit également être précisée : plus la clé est lourde, plus elle sera difficile à découvrir.
L'objet KeyBuilder supporte les tailles de clé suivantes :
public static final short LENGTH_RC4 = 128;

```
public static final short LENGTH_RSA_MIN = 384;
public static final short LENGTH_RSA_512 = 512;
```

Il est important de remarquer que les clés ainsi produites ne sont pas initialisées, le processus pour effectuer cette initialisation va varier en fonction des algorithmes supportés par la clé.

9.2.1) Création d'une paire de clés RSA

La création de la clé s'effectue très simplement grâce au KeyBuilder.
L'initialisation se fait quant à elle par l'intermédiaire de l'objet de type PublicKeyAlgorithm obtenu via le JxtaCryptoSuite (celui-ci sera expliqué au chapitre suivant).
La méthode setPublicKey() va ensuite calculer la clé publique en fonction de la taille spécifiée.
L'appel de setPrivateKey() va quant à lui générer la clé privée en fonction de la clé publique. Il est donc très important de garder cet ordre d'invocation des méthodes.

```
RSAKey rsaKey = (RSAKey)
KeyBuilder.buildKey(KeyBuilder.TYPE_RSA, KeyBuilder.LENGTH_RSA_512, false);
JxtaCryptoSuite jc = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1, rsaKey,
Signature.ALG_RSA_SHA_PKCS1, (byte) 0);
PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm( );
pka.setPublicKey();
pka.setPrivateKey();
```

Il est évidemment intéressant de sauver ces clés sur un support quelconque, il faut pour cela pouvoir récupérer leurs valeurs. Celles-ci sont représentées par la classe RSAxxxkeyData :

```
RSAPublickeyData publicKeyData = (RSAPublickeyData) pka.getPublickey();
RSAPrivatekeyData privateKeyData = (RSAPrivatekeyData) pka.getPrivatekey();
```

Il alors facile de recréer les clés depuis ces valeurs :

```
PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm();
pka.setPublicKey(publicKeyData);
pka.setPrivateKey(privateKeyData);
```

9.2.2) Création d'une clé RC4

Pour initialiser une clé RC4, il suffit d'obtenir un ensemble de bytes aléatoires, ceux-ci seront passés en paramètres à la méthode setKey :

```
SecretKey secretKey = (SecretKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RC4,
KeyBuilder.LENGTH_RC4, false);
JRandom random = new JRandom( );
byte[] keydata = new byte[KeyBuilder.LENGTH_RC4];
random.nextBytes(keydata);
secretKey.setKey(keydata, 0);
```

9.3) La suite JxtaCrypto

Une suite est similaire à une factory.

Elle va donc permettre de récupérer un objet répondant à un profil prédéfini.

Les profils actuellement disponibles sont les suivants :

- PROFILE_RSA_RC4_SHA1
- PROFILE_RSA_RC4_MD5
- PROFILE_RSA_RC4_SHA1_MD5
- MEMBER_RC4

Si le développeur veut utiliser l'authentification des messages pour assurer l'intégrité de ceux-ci, les profils suivants sont également disponibles :

- ALG_RC4_SHA1
- ALG_RC4_MD5

Et si l'on souhaite signer les messages, on peut encore utiliser :

- ALG_RSA_SHA1_PKCS1
- ALG_RSA_MD5_PKCS1

La JxtaCryptoSuite est créée en utilisant un de ses nombreux constructeurs. Le prototype est le suivant :

JxtaCryptoSuite(ENCRYPTIONPROFILE, key, SIGNATUREPROFILE, MACPROFILE);

Il existe donc trois manières pour utiliser les algorithmes dans le security toolkit :

- Instancier un JxtaCryptoSuite avec tous les algorithmes ;
- Instancier un JxtaCryptoSuite avec des algorithmes sélectionnés ;
- Ne pas utiliser la suite et instancier des algorithmes indépendants.

Exemple d'instanciation d'une suite :

```
RSAKey suiteKey = (RSAKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
KeyBuilder.LENGTH_RSA_512, false);

JxtaCrypto encryptionSuite = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_RC4_MD5, suiteKey,
Signature.ALG_RSA_MD5_PKCS1, MAC.ALG_RC4_MD5);
```

La suite permet alors de récupérer des objets qui réalisent les opérations nécessaires :

- getJxtaCipher() ;
- getJxtaHash() ;
- getJxtaMAC() ;
- getJxtaPublicKeyAlgorithm() ;
- getJxtaSignature.

L'algorithme ainsi obtenu peut alors être utilisé pour chiffrer et déchiffrer des données.

9.4) Chiffrement/Déchiffrement

9.4.1) RSA

L'exemple suivant décrit la marche à suivre pour crypter et décrypter des données en utilisant le security toolkit.
Il faut commencer par créer la clé RSA et la JxtaCryptoSuite. Cette dernière prendra comme paramètre le profil RSA tandis que les autres seront nuls, ce qui signifie qu'aucun algorithme d'authentification et de signature n'est nécessaire. La suite permet alors de récupérer l'algorithme qui sera utilisé via la méthode getJxtaPublicKeyAlgorithm().
Les clés, publique et privée, sont alors initialisées avec setPublicKey() et setPrivateKey().
Les valeurs de celles-ci sont alors récupérées dans des objets de type RSAxxxkeyData.
Les tableaux de bytes sont ensuite créés à la taille exacte grâce à la méthode getMaxInputDataBlockLength().
La clé publique ou privée est alors positionnée, suivant qu'il s'agisse d'un cryptage ou d'un décryptage.
Le chiffrement/déchiffrement est enfin effectué via la méthode Algorithm :

rsa.Algorithm(X, 0, X.length, KeyBuilder.TYPE_RSA_PRIVATE, true);

Le premier paramètre étant le tableau de bytes, le second est le byte pour commencer l'encryption, le troisième est la taille du tableau, le quatrième est la clé à utiliser et le dernier est un boolean spécifiant qu'on effectue un chiffrement (true) ou un déchiffrement (false).

Le code ressemble donc à ceci :

```
// Création de la clé RSA
RSAKey rsaKey = (RSAKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
KeyBuilder.LENGTH_RSA_512, false);

// Récupération de la suite
JxtaCrypto suite = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1, rsaKey, (byte)0, (byte)0);

// Récupération de l'objet implémentant l'algorithme RSA et initialisation des clés
PublicKeyAlgorithm rsa = suite.getJxtaPublicKeyAlgorithm();
rsa.setPublicKey();
rsa.setPrivateKey();
RSAPublickeyData rPublicD = (RSAPublickeyData)rsa.getPublickey();
RSAPrivatekeyData rPrivateD = (RSAPrivatekeyData)rsa.getPrivatekey();

// Chiffrement
byte[] X = new byte[rsa.getMaxInputDataBlockLength()];
rsa.setPrivateKey(rPrivateD);
byte[] Y = rsa.Algorithm(X, 0, X.length, KeyBuilder.TYPE_RSA_PRIVATE, true);

// Déchiffrement
rsa.setPublicKey(rPublicD);
byte[] Z = rsa.Algorithm(Y, 0, Y.length, KeyBuilder.TYPE_RSA_PUBLIC, false);
```

9.4.2) RC4

Le code de cet exemple est très similaire au précédent. La différence majeure étant le fait que RC4 peut traiter des tableaux de grande taille, contrairement à RSA qui limite celle-ci.

```
// Récupération de la suite et du cipher
JxtaCrypto suite = new JxtaCryptoSuite(JxtaCrypto.MEMBER_RC4, null, (byte)0, (byte)0);
Cipher rc4 = suite.getJxtaCipher();

// Création du mot de passe et génération de la clé
byte[] password = “Mon password”.getBytes();
SecretKey key = (SecretKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RC4,
KeyBuilder.LENGTH_RC4, false);
// Initialisation
byte[] ibuf = new byte[256];
key.setKey(password, 0);
rc4.init(key, Cipher.MODE_ENCRYPT);

// Chiffrement des données
byte[] obuf = new byte[ibuf.length];
rc4.doFinal(ibuf, 0, ibuf.length, obuf, 0);

// Déchiffrement des données
byte[] clearText = new byte[obuf.length];
rc4.init(key, Cipher.MODE_DECRYPT);
rc4.doFinal(obuf, 0, obuf.length, clearText, 0);
```

9.5) Signer les données

La classe signature (jxta.security.signature.Signature) permet de créer et de vérifier des signatures digitales.
Une telle signature permet de vérifier que les données proviennent bien de la bonne source.
Les signatures digitales sont basées sur les clés RSA. La signature est créée avec une clé privée tandis que la vérification s’effectue avec la clé publique correspondante.
L’objet permettant de traiter les signatures sera obtenu, on s’en doute, par l’intermédiaire de la JxtaCryptoSuite. Il faudra pour cela lui spécifier un comportement de type : ALG_RSA_SHA1_PKCS1 ou ALG_RSA_MD5_PKCS1.
Une fois l’objet Signature obtenu, il faut l’initialiser soit en mode signature ou en mode vérification. On utilisera pour cela sa méthode init() prenant comme paramètre le mode souhaité : MODE_SIGN ou MODE_VERIFY.
A noter que la méthode update peut être utilisée pour ajouter des données à signer ou vérifier.

```
// Création des clés, de la suite et recuperation de l’objet Signature
RSAKey rsaKey = (RSAKey)KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
KeyBuilder.LENGTH_RSA_512, false);
JxtaCrypto suite = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1, rsaKey,
Signature.ALG_RSA_SHA_PKCS1, (byte)0);
Signature sig = suite.getJxtaSignature();

// Signature des données
byte[] ibuf = new byte[1024];
sig.init(Signature.MODE_SIGN);
byte[] sigBuf = sig.sign(ibuf, 0, ibuf.length);

// Vérification de la signature
sig.init(Signature.MODE_VERIFY);
```

```
boolean verified = sig.verify(ibuf, 0, ibuf.length, sigBuf, 0, sigBuf.length);
```

9.6) Le hash

La classe Hash (jxta.security.hash.Hash) calcule le message digest ou hash des données.
Un digest est un petit tableau de bytes qui est calculé sur un ensemble de données. Il n'est pas garanti qu'un hash soit unique, mais la probabilité que deux ensembles de données différents produisent le même hash est vraiment très faible.
Une fois ce hash calculé sur le message, il est joint à celui-ci et envoyé sur le réseau. A la réception, le hash est recalculé sur les données et ensuite comparé avec celui envoyé sur le message. En cas de non égalité, on peut conclure que les données ont été modifiées durant le transport. Le problème est évidemment que si une personne modifie les données en cours de route, elle peut modifier le hash par la même occasion.
Pour éviter ce problème il faut signer ce hash avec une signature digitale ou préférer utiliser un MAC (voir chapitre suivant).
L'API JXTA supporte deux algorithmes de hash : MD5 et SHA1. Les objets Hash sont récupérés grâce à la méthode getJxtaHash() de la JxtaCryptoSuite.
Une fois cet objet Hash récupéré, il suffit d'ajouter les données sur lesquelles le hash sera généré via la méthode update. La génération du hash en lui-même est alors réalisée grâce à la méthode doFinal.

```
JxtaCrypto suite = new JxtaCryptoSuite((byte)0, null, (byte)0, Hash.ALG_RSA_MD5_PKCS1);
Hash hs = suite.getJxtaHash();
byte[] ibuf=”Mon hash”.getBytes();
byte[] obuf=new byte[64];
hs.update(ibuf, 0, ibuf.length);
hs.doFinal(ibuf, 0, ibuf.length, obuf, 0);
```

9.7) Le MAC

La classe MAC (jxta.security.mac.MAC) crée un hash sécurisé en le chiffrant avec RC4.
Un MAC peut donc être comparé à une signature, la différence étant située au niveau des clés utilisées. Ainsi, si une signature digitale utilise une paire de clés RSA (publique et privée), le MAC utilise quant à lui une clé secrète qui doit être partagée entre les correspondants. C'est donc la même clé qui servira pour la création et la vérification du MAC.
Il faut donc commencer par créer une clé secrète et l'initialiser avec un nombre aléatoire.
Les données du MAC sont ensuite ajoutées via la méthode update().
Le MAC est alors généré par l'intermédiaire de la méthode encrypt().

```
// Création de l'objet MAC et de la clé secrète
MAC mac = jc.getJxtaMAC( );
SecretKey secretKey = (SecretKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RC4,
KeyBuilder.LENGTH_RC4, false);
// Initialisation du MAC
JRandom random = new JRandom( );
byte[] keydata = new byte[KeyBuilder.LENGTH_RC4];
random.nextBytes(keydata);
mac.init(MAC.MODE_ENCRYPT, secretKey, keydata);

// Création du MAC
byte[] ibuf=”Mon hash”.getBytes();
```

```
byte[] obuf;
mac.update(ibuf, 0, ibuf.length);
mac.encrypt(ibuf, 0, ibuf.length, obuf, 0);
```

9.8) Des communications sécurisées

Les pipes utilisés jusqu'à présent sont non sécurisés et présentent donc les problèmes suivants :

- Les données transmises sur le réseau circulent en clair, n'importe quelle personne ayant accès au réseau peut donc les lire (confidentialité).
- Les données peuvent passer par d'autres ordinateurs qui peuvent donc en profiter pour les modifier (intégrité).
- On ne peut jamais être certain que l'on communique avec la même personne. Un peer peut en effet très bien se faire passer pour quelqu'un qu'il n'est pas (authentification).

Les tubes sécurisés offrent les solutions en utilisant des données signées permettant d'identifier les peers en communication. Les données sont également cryptées, assurant ainsi leur intégrité et leur confidentialité.
La technologie utilisée est en fait le JXTATLS qui est basée sur SSLv3 et TLSv1. JXTA utilise une paire de clés RSA pour initialiser la connexion, il utilise également le triple-DES pour chiffrer les données et enfin des signatures digitales. Une des caractéristiques du triple-DES est qu'il utilise des clés de 192 bits, ce qui le rend très difficile à pirater.
Pendant l'initialisation, chaque peer sur le pipe présente son certificat attestant de sa clé publique aux autres peers. Ceux-ci vont alors valider le certificat pour s'assurer de l'identité du peer distant.
Par défaut, les certificats utilisés sont issus de la plateforme JXTA elle-même, ce qui signifie que ces certificats sont auto signés et donc peu sûr. Il est cependant possible de modifier cela pour utiliser des certificats basés sur des autorités bien connues.

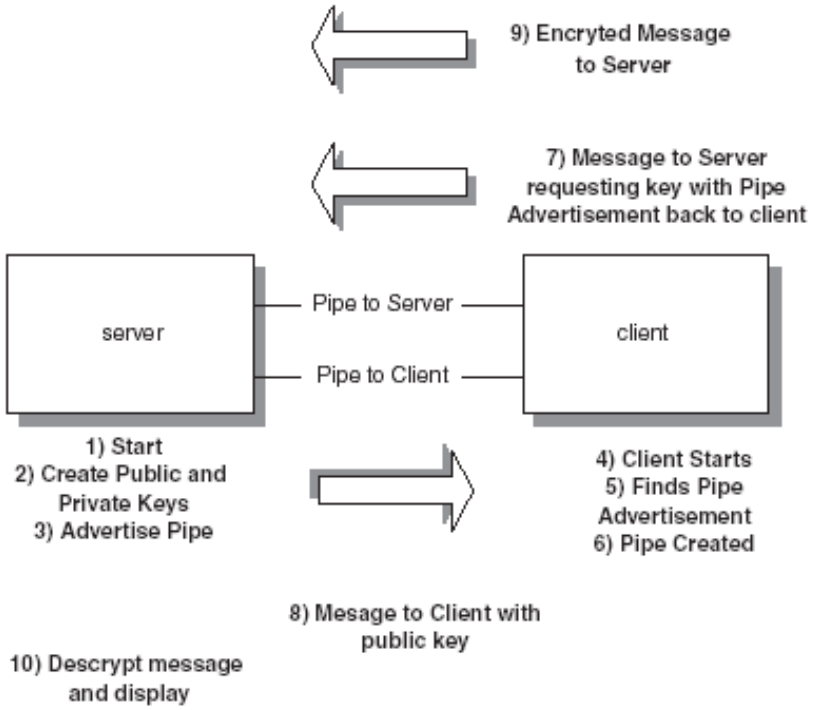
Mais comment tout ce système est-il mis en place ? La réponse est très simple, comme expliqué précédemment il existe plusieurs types de pipes et ce type est précisé dans le pipe advertisement. Pour créer un tube sécurisé il suffit donc d'appeler la méthode adéquate :

```
pipeAdvertisement.setType(PipeService.UnicastSecureType);
```

Rien de très compliqué… Il ne faut cependant pas oublier que l'utilisation de communications sécurisées ralenti évidemment les échanges, et cela du en fait des opérations de chiffrement/déchiffrement.

Avec toutes les techniques expliquées dans ce chapitre consacré à la sécurité, il serait cependant tout à fait possible de créer notre propre système de communication sécurisé. Cela permettrait par exemple d'avoir plus de contrôle sur les algorithmes de chiffrement à utiliser, etc.
Il faudrait donc par exemple commencer par créer une paire de clés RSA. Ensuite créer un pipe, publier son advertisement et se mettre en écoute sur celui-ci.
De l'autre côté, il suffirait de rechercher cet advertisement et de créer le pipe correspondant.
Le client peut alors demander la clé publique au serveur. Ce dernier sérialise donc celle-ci et la fait parvenir au client. Il recrée ainsi la clé RSA avec la valeur reçue et crypte un premier message de test.
Le serveur décrypte se message et vérifie donc que tout est correct.
Il s'agit bien sûr d'un schéma assez simple, il peut cependant facilement être complété avec des certificats, des vérifications d'intégrité, etc.

Les opérations pourraient être représentées comme ceci :



10) Créer un groupe sécurisé

10.1) Introduction

Dans les exemples décrits jusqu'à présent, chaque peer peut rejoindre un groupe sans aucune restriction. Le Membership Service (voir 8.6.2) permet cependant de limiter l'accès à un groupe aux peers qui possèdent un ensemble de credentials suffisant. Le groupe examine donc les credentials d'un peer et détermine alors si celui-ci est autorisé à le rejoindre. Pour pouvoir implémenter un tel système d'authentification, il est nécessaire de créer une classe qui dérive de la class abstraite Membership. Cette classe implémente les politiques d'accès du groupe. Une implémentation existante est la classe PasswdMembershipService. Celle-ci est initialisée avec une liste d'utilisateurs et leurs passwords. Ainsi, chaque peer qui veut joindre un groupe utilisant ce service doit fournir un credential avec le password approprié.

10.2) Création de l'advertisement d'implémentation

Comme expliqué dans le chapitre consacré aux modules (voir 8.11), un groupe est représenté par plusieurs advertisements : module class, specification et implementation. Pour créer un groupe sécurisé, utilisant le PasswdMembershipService il faut créer un nouveau implementation advertisement qui contient les informations nécessaires à l'utilisation de ce service.
Pour effectuer cela, il faut copier l'advertisement par défaut et modifier les champs requis.

Les étapes pour créer son propre implementation advertisement sont les suivantes :

- Copier l'advertisement d'implémentation par défaut d'un groupe ;
- Retrouver le memebership service dans les éléments de cet advertisement ;
- Modifier l'élément membership avec les informations sur le PasswdMembreshipService ;
- Publier le nouvel advertisement.

Il faut donc commencer par créer un objet de type ModuleImplAdvertisement qui va être une copie de l'advertisement par défaut du net peer group. On utilise pour cela la méthode getAllPurposePeerGroupImplAdvertisement() du groupe par défaut.
Un groupe offre de nombreux services de base et pas uniquement celui de membership. Il faut donc d'abord retrouver ce dernier pour pouvoir le modifier. La méthode getParam() du ModuleImplAdvertisement permet de récupérer un PeerGroupParamAdv, il s'agit de la liste des services qui va pouvoir être récupérée dans une hashtable grâce à sa méthode getServices().
Cette hashtable va donc être parcourue, chacun de ses éléments est casté en ModuleClassID et la méthode isOfSameBaseClass() de ce dernier est appelée, prenant refMembershipSpecID en paramètre. Ce dernier est l'identifiant du service d'adhésion et la méthode isOfSameBaseClass() permet donc logiquement de déterminer si le MCID récupéré est celui du Membership Service.
Une fois que le service a été trouvé, il faut donc le modifier pour apporter le mécanisme d'authentification. Il suffit pour cela de modifier le nouvel advertisement en appelant les méthodes adéquates : setModuleSpecID pour spécifier le passwordMembershipSpecID, setCode pour signaler ou se trouve le code correspondant et setDescription pour annoncer qu'il s'agit d'un groupe sécurisé.
L'ancien service peut alors être supprimé de la hashtable par un simple remove() et le nouveau ajouté via put().
Il ne reste alors qu'à publier l'advertisement en utilisant la technique déjà expliquée précédemment.

```
ModuleImplAdvertisement mia = null;
StdPeerGroupParamAdv PeerGroupParamAdv = null;

// Récupère le implementation advertisement par défaut du NetPeerGroup
mia = netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

// Récupère le champs qui contient tous les services associés au groupe
PeerGroupParamAdv = new StdPeerGroupParamAdv(mia.getParam());

// Récupère une hashtable contenant les services
Hashtable allPurposePeerGroupServicesHashtable = PeerGroupParamAdv.getServices();

// Replace the membership service in the hashtable.
Enumeration services = allPurposePeerGroupServicesHashtable.keys();
while (services.hasMoreElements())
{
    Object key = services.nextElement( );
    ModuleClassID mcid;
    mcid = (ModuleClassID) key;
    if (mcid.isOfSameBaseClass(PeerGroup.refMembershipSpecID))
    {
       // On a trouvé le membership service, on crée un nouvel advertisement pour le nouveau
       service d'authentification
       mia.setModuleSpecID( PasswdMembershipService.passwordMembershipSpecID);
       mia.setCode("net.jxta.impl.membership.PasswdMembershipService");
       mia.setDescription("Authenticated Membership Service");
       allPurposePeerGroupServicesHashtable.remove(key);
       allPurposePeerGroupServicesHashtable.put(key, mia);
       break;
    }
}
// On sauvegarde les services dans l'advertisement
```

```
PeerGroupParamAdv.setServices(services);
mia.setParam((Element) PeerGroupParamAdv.getDocument(new MimeMediaType("text/xml")));

// On publie le nouvel advertisement
discovery.publish(mia, PeerGroup.DEFAULT_LIFETIME, PeerGroup.DEFAULT_EXPIRATION);
discovery.remotePublish(mia, PeerGroup.DEFAULT_EXPIRATION);
```

10.3) Création de l'advertisement du groupe

La création de cet advertisement se fait de manière habituelle, avec les nuances suivantes :

- Une nouvelle implémentation ayant été créée pour le groupe, il faut lui préciser la spécification pour celle-ci ;
- Il faut enregistrer la liste des logins et passwords dans la section Param de l'advertisement.

L'advetisement en lui-même est créé simplement via l'objet statique AdvertisementFactory (voir 8.3.3). Sa spécification est spécifiée en appelant la méthode setModuleSpecID, prenant en paramètre le ModuleSpecID de l'advertisement créé au chapitre précédent (mia.getModuleSpecID()).
Il faut alors créer un document structuré contenant les codes d'accès et placer celui-ci dans la section aram de l'advertisement du groupe.
Ce dernier peut alors être publié et il ne reste plus qu'à créer le groupe.

```
String login = ''Antho'';
String passwd = ''stage'';
String groupName = "AnthoSecure'';

// Création d'un nouvel advertisement pour le groupe
PeerGroupAdvertisement pgAdv=(PeerGroupAdvertisement)
AdvertisementFactory.newAdvertisement(PeerGroupAdvertisement.getAdvertisementType());
pgAdv.setPeerGroupID(myID);
pgAdv.setModuleSpecID(mia.getModuleSpecID());
pgAdv.setName(groupName);
pgAdv.setDescription("Groupe sécurisé par password");

// Création du document contenant le login et le password
StructuredTextDocument loginAndPasswd= (StructuredTextDocument)
StructuredDocumentFactory.newStructuredDocument(newMimeMediaType("text/xml"),"Parm");
String loginAndPasswdString= login+":"+PasswdMembershipService.makePsswd(passwd)+":";
TextElement loginElement = loginAndPasswd.createElement("login",loginAndPasswdString);
loginAndPasswd.appendChild(loginElement);

// Le login et password sont places dans la section Param de l'advertisement du groupe
pgAdv.putServiceParam(PeerGroup.membershipClassID,loginAndPasswd);

discovery.publish(pgAdv, PeerGroup.DEFAULT_LIFETIME, PeerGroup.DEFAULT_EXPIRATION);
discovery.remotePublish(pgAdv, PeerGroup.DEFAULT_EXPIRATION);

// Création du groupe
pg=netPeerGroup.newGroup(pgAdv);
```

Il est important de noter que l'algorithme de chiffrement utilise par PasswdMembershipService.makePsswd est très faible et donc insécurisé. Il est donc vivement conseillé d'utiliser un autre algorithme pour crypter ses passwords.

10.4) Joindre le groupe

Une fois que le groupe a été découvert, un peer doit suivre différentes étapes pour y adhérer :

- Récupérer le credential d'authentification du groupe ;
- Obtenir l'authentificateur lié à ce document (voir 8.6.2) ;
- Compléter l'authentificateur avec les informations nécessaires ;
- Utiliser cet objet pour joindre le groupe.

Comme déjà expliqué, le credential est récupéré en passant le groupe au constructeur de AuthentificationCredential. Le MembershipService est quant à lui obtenu via la méthode getMembershipService du groupe.
Il faut alors appeler la méthode apply de ce service pour récupérer l'objet de type authentificator. Ce dernier sera ensuite casté en PasswdAuthentificator.
Le login et le password seront alors spécifiés en appelant les méthodes setAuth1Identity() et setAuth2_Password().
Le groupe peut alors être rejoint en appelant la méthode join du Membership Service.

```
StructuredDocument creds = null;
AuthenticationCredential authCred = new AuthenticationCredential(pg, null, creds);
MembershipService m = (MembershipService) pg.getMembershipService();

Authenticator auth = m.apply(authCred);
PasswdMembershipService.PasswdAuthenticator pwAuth=
(PasswdMembershipService.PasswdAuthenticator) auth;

pwAuth.setAuth1Identity("Antho");
pwAuth.setAuth2_Password("stage");

if (!auth.isReadyForJoin( ))
{
    System.out.println(“Erreur”);
}
else
{
    m.join(auth);
}
```

Il va de soi que les informations demandées par l'authentificator varient en fonction du service d'adhésion utilisé. Dans notre cas il s'agit d'un login et d'un password car nous utilisons le PasswdMembershipService.
Ce dernier est très simple, il ne fait que comparer le password reçu avec ceux présents dans le champs Param de l'advertisement du groupe.
Je rappelle qu'il est cependant possible de créer ses propres services de membership, répondants à des critères d'adhésion très précis.
Un service plus complexe est Envision qui permet d'obliger à crypter les passwords, de signer les credentials ou encore d'interagir avec un annuaire LDAP.

Au niveau de la programmation, n'importe quel service respectera cependant les règles énoncées ci-dessus. La seule différence est les informations passées à l'authentificateur qui varient d'un service à un autre.

11) Un réseau privé

Une technique très simple pour protéger et sécuriser son réseau JXTA est de le rendre privé.
Cela permet d'isoler son réseau du reste du réseau public et donc d'apporter un certain niveau de sécurité.

Pour qu'un tel système puisse fonctionner, il devient bien sûr indispensable d'implémenter ses propres peers de rendezvous et de relay, permettant d'établir la communication entre les différents peers situés sur des extrémités différentes du réseau. Un réseau privé permet donc également d'avoir plus de contrôle sur le fonctionnement de ce lui-ci.
Si le cryptage est de plus implémenté sur le réseau privé, ce dernier pourrait alors être comparé à un VPN.
Pour éviter que chaque peer ne rejoignent le net peer group par défaut mais plutôt le réseau privé, il suffit de placer un fichier config.properties dans le dossier .jxta de chaque peer. Ce fichier contient les informations sur le groupe principal du réseau privé, à savoir : son ID, son nom et sa description.
Le code permettant de lancer la plateforme JXTA doit alors être adapté pour prendre en compte ce fichier. L'objet statique PeerGroupFactory déjà utilisé pour rejoindre le net peer group possède différentes méthodes permettant de le configurer. Parmi celles-ci, on remarque notamment : setNetPGID(), setNetPGName() et setNetPGDesc(). C'est donc grâce à elles qu'on va pouvoir configurer le groupe, en récupérant les informations du fichier properties.
Le code pourrait alors ressembler à ceci :

```
ClassLoader cl = getClass().getClassLoader();
InputStream is = cl.getResourceAsStream("config.properties");
Properties p = new Properties();
p.load(is);
String pg = p.getProperty("NetPeerGroupID");
PeerGroupID pgid = (PeerGroupID )IDFactory.fromURI(new URI(pg));
PeerGroupFactory.setNetPGID(pgid);
PeerGroupFactory.setNetPGName(p.getProperty("NetPeerGroupName"));
PeerGroupFactory.setNetPGDesc(p.getProperty("NetPeerGroupDesc"));
netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

Une question qui peut venir à l'esprit est : comment générer l'ID de notre groupe privé?
Il suffit pour cela de créer un digest sur le nom et la description du groupe puis de passer celui-ci à la méthode newPeerGroupID() de l'objet IDFactory :

```
byte[] buffer = id.getBytes();
byte[] digest;
MessageDigest algorithm = null;

try
{
algorithm = MessageDigest.getInstance("MD5");
// Génération du digest
algorithm.reset();
```

```
algorithm.update(buffer);
digest = algorithm.digest();
}
catch (Exception e) { }

PeerGroupID pgID = IDFactory.newPeerGroupID(digest);
```

Si JXTA est encore peu connu, plus que probablement suite à la mauvaise réputation du peer to peer, il ne fait aucun doute qu'il s'agit d'une technologie très évoluée aux multiples avantages.
Cette suite de protocoles permet d'élaborer une couche réseau solide sur laquelle peut reposer tout type d'application destiné à fonctionner sur Internet. C'est pourquoi je ne serai pas surpris de voir JXTA intégré dans de nombreux projets futurs, commerciaux ou non.
Après un premier développement avec l'API Java de JXTA, une constatation s'impose : le modèle de cette technologie a été soigneusement étudié, offrant de nombreuses possibilités avec une grande facilité de développement.
JXTA a de plus l'énorme avantage d'être une solution Open Source, et donc en perpétuelle évolution …

Votre langue : **FR** | EN

Qui suis-je?

- # Catégories

- Ruby (1)
- Ruby on Rails (9)

- # Archives

- Août (1)
- Mai (1)
- Avril (3)
- Mars (3)
- Février (2)

- # Liens

Twitter

- I'm ready to jump into Ruby 1.9.1 thanks to the excellent Ruby Switcher : http://bit.ly/xSZ7V about an hour ago
- I really like the concept of http://99designs.com 3 days ago

- Rails + Compass (Sass + Blueprint) == WIN : http://bit.ly/36eWCM 3 days ago
- RT @ merbjedi: Went through gemcutter.org and subscribed to all my favorite RubyGems. Now I have an RSS feed for any gem updates. Awesome 3 days ago
- @ jlecour Très bon article et merci pour le lien! 3 days ago

Suivez-moi sur Twitter