Burkhalter

samedi, 6. mars 2010
11:52

Rapport_IFC_Labo1_Li...

Inserted from: <file://D:\jaton\Enseignement\Classes\TR2011\Sockets\Burkhalter\Rapport_IFC_Labo1_Lienhard_Burkhalter.pdf>

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

IFC – LABORATOIRE N°1

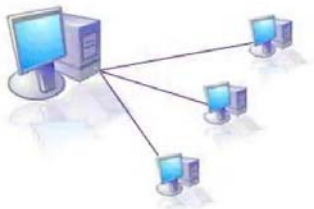
PROGRAMMATION D'UNE APPLICATION JAVA PERMETTANT DE TRANSMETTRE DIFFERENTS TYPES DE DONNEES D'UN CLIENT A UN SERVEUR ET VIS-VERSA.

PROFESSEUR

M. Markus JATON

ASSISTANT

M. Grégory RUCH



Un rapport correct et soigné. Il vous manque (ainsi qu'à d'autres) certaines notions de protocoles d'application qui vous permettraient de mettre sur pied des échanges plus cohérents.

NOTE 5.4

AUTEURS

Steve LIENHARD
Arnaud BURKHALTER

DATES : 27 Février 2010
TEMPS A DISPOSITION : 2 semaines
DATE DE DEBUT: 18 Février 2010
DATE DE FIN : 4 Mars 2010

Table des matières :

Steve LIENHARD
Arnaud BURKHALTER

11.05.2009

Introduction

Dans le monde d'aujourd'hui, les applications qui n'accèdent pas à internet ou qui ne communiquent avec aucune base de donnée externe se font très rares. Le développement d'appareils tels qu'Iphone ou Nexus One ne fait que renforcer cela. Il est donc primordial pour toute personne travaillant dans les télécommunications de comprendre les mécanismes et le fonctionnement permettant de faire communiquer une application avec le monde extérieure. Ce laboratoire va donc d'une part nous faire découvrir le principe des sockets et d'autre part les classes offertes par l'API de Java permettant de les implémenter.

Sockets

Définition

Comme vu en théorie, un socket est un point de connexion à un canal de transmission. Cette transmission peut aussi bien être une transmission locale (inter-processus) qu'une transmission « réseau » vers une application extérieure. Les sockets qui nous intéressent dans ce laboratoire sont les sockets réseaux.

Sockets réseau

Un socket réseau est défini par une adresse IP et un numéro de port. La notion de socket réseau est étroitement liée à l'architecture bien connue de client-serveur. Une fois le serveur lancé, celui-ci écoute sur un port donné les requêtes des éventuels clients. Le client va de son côté envoyer sa requête (par un port local) à l'adresse IP du serveur et sur un le port d'écoute de celui-ci.
Un socket peut fonctionner en mode connecté ou déconnecté. Nous ne nous intéresserons dans ce laboratoire seulement au mode connecté car c'est celui qui nous intéresse. Un socket en mode connecté est appelé « flot » et peut être vu comme un fichier, les deux étant des flux.

On ne peut être plus explicite !

Classes Java

Sockets en Java

L'API Java offre différentes classes permettant d'implémenter la notion de socket. Nous ne citerons seulement celles que nous avons utilisées dans notre code.

Socket : Cette classes permet, une fois la connexion établie entre le serveur et le client, de pouvoir récupérer des flux (par l'intermédiaire de getInputStream() ou getOutputStream()) afin de créer une liaison entre le flux d'écriture du client et le flux de lecture du serveur (par exemple). C'est par la classe socket que la connexion est établie.

ServerSocket : Cette classe permet au serveur d'écouter un port spécifié afin de pouvoir accepter une connexion demandée par le client. L'acceptation de la demande se fait par l'intermédiaire de la méthode *accept()* qui renvoi un objet de la classe socket. Le socket ainsi crée permet de gérer la connexion et les flux entre le client et le serveur

Flux en Java

Les flux en Java permettent l'encapsulation de l'envoi et la réception de données de façon séquentielle.

Toutes les classes misent à disposition par Java, et permettant le traitement sur les données de flux, se trouvent dans le package java.io.

De manière générale, les flux peuvent être séparés en plusieurs catégories. Les flux d'entrée et flux de sortie, ainsi que les flux de traitement d'octets et les flux de traitement de caractères. Il existe donc différentes sous-classes permettant le traitement de ces différentes catégories de flux.

- *Reader* sont des types de flux en lecture sur des caractères
- *Writer* sont des types de flux en écriture sur des caractères
- *InputStream* sont des types de flux en lecture sur des octets
- *OutputStream* sont des types de flux en écriture sur des octets

Fichier en Java

Java permet la gestion et manipulation de fichiers, à l'aide de l'interface FilenameFilter ainsi que la classe File, se trouvant dans le package java.io.

Dans le cadre de ce travail, nous avons principalement eu recours à la classe File, qui est utilisée pour représenter un chemin d'accès à des fichiers ou répertoires et effectuer des opérations sur ces derniers.

Les principales méthodes utilisées dans la gestion de fichier, du moins dans le cadre de notre laboratoire, sont les suivantes :

- File() : permet la création d'une instance de la classe File.
- FileReader(): permet la lecture d'un fichier de caractère via un flux.
- FileWriter() : permet l'écriture d'un fichier de caractères via un flux.

Programmations

Client

Le client de cette application doit être capable de faire les différentes choses mentionnées ci-dessous :

Connexion

La connexion s'effectue une fois que l'utilisateur a entré l'IP du serveur et le port d'accès. Si le serveur est sur écoute à ce moment là, la connexion va être établie et le client va donc afficher une confirmation. Il va également envoyer au serveur le fait que cette connexion est ON. Ceci a été fait au moyen d'une BufferReader et d'un PrintWriter, uniquement pour montrer que la communication sous forme de texte fonctionne. Les deux flux in et out du client ainsi que ceux du serveur ne sont pas indispensables puisque nous aurions pu tout envoyer en tant qu'objet et cela aurait également fonctionné.

Ecriture/Lecture de texte

Le client doit aussi pouvoir envoyer un texte au serveur (sous forme de string) et que le serveur puisse lui renvoyer un echo de ce même texte. Le client doit donc être en mesure d' « écouter » continuellement pour savoir si le serveur lui a renvoyé quelque chose. Nous avons fait en sorte que le serveur puisse également écrire son propre texte à envoyer, à la manière d'un « chat » (ce qui n'est pas forcément demandé dans la donnée).

Ecriture types primitifs

Afin de pouvoir envoyer des types primitifs, nous avons créé un flux ObjectOutputStream qui permet d'envoyer tous types d'objets, pour autant que ceux-ci puissent être envoyés (voir conditions nécessaire dans la suite du rapport). Le client va donc écrire la valeur du type voulu sur le flux d'objet. L'idée est qu'une fois l'objet reçu par le serveur, ce dernier puisse savoir de quelle instance cet objet fait partie par l'intermédiaire d'un « instance of », du fait que ce sont des types primitifs. Ce principe est peut être lourd à première vue, mais il évite de devoir coder un protocole pour l'envoie de types simples. De plus, le flux crée peut également servir à envoyer des types plus compliqué comme des objets, ce qui représente un grand avantage du point de vue de l'extensibilité de l'application.

Vous avez donc utilisé l'autoboxing ? Vous êtes bien conscients que vous tablez sur une spécificité du langage de programmation utilisé ?

Remarque : Nous partons du principe que l'utilisateur du client n'entrera pas un type invalide (string au lieu d'un double, etc.). Par faute de temps, nous n'avons pas traité ce type d'exceptions, du fait que ce n'est pas le but premier de ce laboratoire.

Ecriture fichiers

Afin d'envoyer des fichiers, il est nécessaire cette fois-ci de définir un protocole précis pour que le serveur puisse détecter que c'est un fichier et le « reconstruire » sur son disque. Nous avons donc mis en place le protocole suivant :
Avant d'envoyer le fichier, le client y ajoute l'en-tête « %file* » afin de préciser que c'est un fichier. Après cet en-tête, vient s'ajouter le nom du fichier suivit d'un tiret.
Pour envoyer le fichier, le client ne fait que le lire et l'envoyer sous forme d'un string.

Je constate que vous (ainsi que vos collègues) n'avez pas beaucoup de notions de création de protocoles d'application. Il serait peut-être utile de vous en proposer quelques-unes...

D'un coup ? Ce n'est pas un transfert de fichier...

Déconnexion

La déconnexion du client implique qu'il faut fermer tous les flux ouverts jusqu'à présent. Les flux et sockets sont donc fermés et les autres attributs remis à « 0 ».
Nous avons également ajouté une méthode finalize() dans le cas où l'utilisateur fermerait l'application client à l'aide de la croix rouge. En effet il est indispensable dans ce cas là de fermer également ces flux, pour supprimer proprement la connexion avec le serveur.

Remarque : si le client se déconnecte, il faut que le serveur se déconnecte aussi avant toute nouvelle connexion.

Serveur

Connexion

La connexion du serveur diffère légèrement du client, de par le fait que le serveur doit d'abord créer un « socket serveur » pour pouvoir écouter sur le port spécifié. Une fois le port en écoute, le serveur attend la connexion du client. Une fois ce dernier connecté, le serveur crée la connexion en appelant la méthode « accept() » qui rend le socket de connexion.
La même remarque faite pour le client est applicable ici : les attributs in et out sont uniquement présent dans un but pédagogique afin de montrer que l'application fonctionne avec l'envoie de texte pur (sans passer par des ObjectInputStream ou ObjectOutpuStream).

Ecriture/Lecture de texte

Les mêmes mécanismes que pour le client sont présents ici.

Lecture types primitifs

Comme mentionné précédemment, la lecture se fait par des « instance of » afin de pouvoir « trier » les types entre eux par des if else (se référer au code…).
Nous avons volontairement converti les objets dans leur type respectif afin que l'objet « retrouve » son type initial lors de l'affichage.

Déconnexion

De même que pour le client, les flux sont fermés lors de l'action du bouton déconnexion, et une méthode finalize() a été rajoutée pour les mêmes raisons.

Transmission d'objets

La transmission d'un objet par un canal est tout à fait faisable, moyennant quelques conditions non négligeables :
L'objet doit tout d'abord être sérializable, et donc implémenter l'interface serializable. Si la méthode de sérialisation n'est pas redéfinie, c'est le système qui fera de son mieux pour sérialiser l'objet, sinon, c'est le programmeur qui choisit.
La deuxième condition est que le récepteur de l'objet ait un moyen de le « reconstruire ». En effet, celui-ci va recevoir un objet sérialisé dont il n'aura aucune idée de la signification des données contenues dans l'objet. Il est donc nécessaire de définir un protocole afin que le récepteur puisse sans problème savoir de quoi il s'agit.

L'interface Serializable ne contient aucune méthode !
Comment le programmeur choisit-il ?

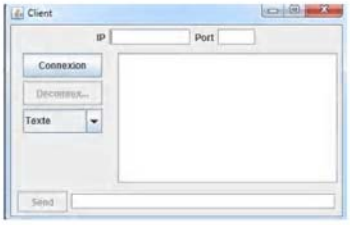
Quel est votre protocole ?

Interface Graphique

Pour la mise en place de la partie graphique du laboratoire, qui consiste à créer une interface, sous forme de fenêtre, permettant à l'utilisateur de gérer le transfère d'informations entre le client et le serveur.
L'implémentation de cette partie, a nécessité l'utilisation des packages ActionListener et ActionEvent toute deux offerts par Java, et mettant à disposition les différentes classes et outils nécessaires à la création de notre interface graphique.

Les interfaces en mode client et serveur se présentent de la manière suivante :

Client :
IP : champ traitant l'adresse IP du serveur.
Port : port du serveur que l'on souhaite écouter.
Connexion : initialise la connexion au serveur.
Deconnexion : mets un terme à la connexion en cours.
Liste déroulante (Texte) : liste permettant à l'utilisateur de choisir le type de données qu'il souhaite echanger avec le serveur. Les valeurs possibles sont *Texte*, *Integer*, *Double*, *Float*, *Byte* et *Fichier*.
Send : envoie la valeur présente dans le champ de texte, présent à sa droite.



Serveur :
Port : port que le serveur doit écouter afin d'autoriser l'accès.
Connexion : initialise la connexion avec le client.
Deconnexion : mets un terme à la connexion en cours.
Send : envoie au client la valeur présente dans le champ de texte, présent à sa droite.

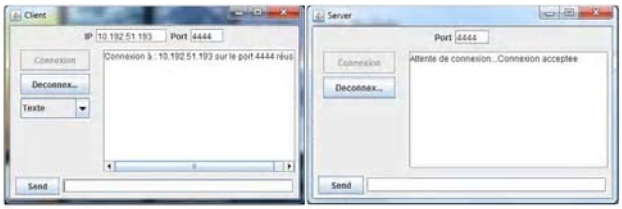
Tests Réalisés

Afin de tester le bon fonctionnement de notre programme, nous avons utilisé une stratégie de tests qui consiste à tester la transmission client-serveur pour chacun des types implémentés.

Aperçu des différents tests réalisés, ainsi que les résultats obtenus :

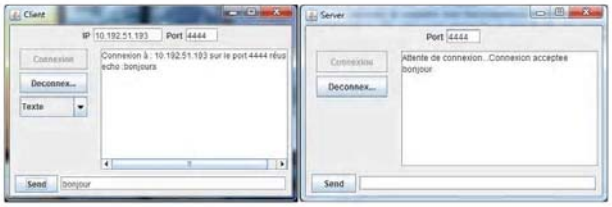
Etablir une connexion entre le client et le serveur
Comme nous pouvons le voir dans les captures d'écrans ci-dessous, la connexion entre le client et le serveur est bien établi. Pour se faire nous devons procéder de la manière suivante :

1) Coté serveur : entrer le port sur lequel il faut écouter, cliquer sur connexion.
2) Coté client : entrer le port ainsi que l'adresse **IP** du serveur, cliquer sur connexion
3) Contrôler que la connexion est bien établie (à l'aide de la confirmation).



Saisies d'écran en JPEG : dommage !

Transmettre du texte
Une fois la connexion établie, nous avons pu effectuer des transferts de texte. Pour se faire nous avons procédé de la manière suivante : sélectionner le type que l'on souhaite transférer, à l'aide de la liste déroulante, dans le cas présent du type Texte, puis en entrant la valeur souhaitée dans la zone de saisie. La transmission est exécutée lorsque nous cliquons sur le bouton Send.
Une confirmation de l'envoie du message est envoyé au client, par le serveur, est affichée, par un echo de la valeur envoyée.



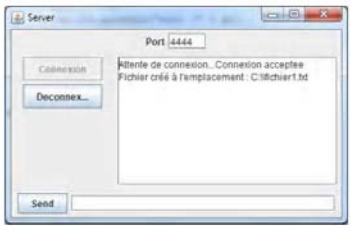
La même stratégie de tests a été mise en place pour tester le transfère de valeurs de type integer, double, float et byte.

Le transfert est-il binaire ou avez-vous restreint le transfert à un type particulier (texte, par exemple)

Transmettre un fichier

Lors du transfert d'un fichier, ce dernier sera transmis et reconstruit sur le serveur à partir du chemin de ce dernier. Une légère contrainte s'impose donc à cela, du fait que si l'on souhaite que le fichier transmis se trouve à la racine du serveur, il faut envoyer un fichier se trouvant lui même à la racine du disque du client. De plus, afin d'indiquer le chemin du fichier, il faut utiliser des doubles back-slashs (\\).

Outre cette mesure à prendre, le transfert de fichier, du client au serveur, se fait de manière correcte. Comme nous pouvons le voir dans la figure ci-dessous, la création du fichier sur le disque du serveur entraine un message sur l'interface de ce dernier.



Conclusion

Ce laboratoire nous a permis de comprendre les fondements même de la programmation réseaux grâce à la manipulation des sockets et des classes mises à disposition par Java. Nous avons vu que grâce à ces classes et en ayant bien compris le principe des flux, il est moins compliqué qu'il y parait de faire intervenir des connections à l'intérieur de nos programmes.

Comme dit précédemment, ces mécanismes sont beaucoup utilisés dans l'architecture web et il est très important de connaître leurs fonctionnements.

Parallèlement à cela, nous avons également pu approfondir nos connaissances sur le codage d'interfaces graphiques, ce qui nous a permis de découvrir différentes classes (notamment JTextField et JTextArea) qui seront très utiles pour la suite.