Environnement Client/Serveur

Cours 4
Broadcast UDP
Étude de cas
Java 8

kn@lri.fr
http://www.lri.fr/~kn

Plan

Retour sur les adresses IP

La connectivité IP est données par

- Une adresse IP (**`129.175.28.29`**)
- Un masque (**`255.255.0.0`**)
- Un adresse de réseau (**`129.175.0.0`**), la première adresse de l'intervalle
- Une adresse de diffusion (**`129.175.255.255`**), la dernière adresse de l'intervalle

L'adresse de diffusion est une adresse spéciale qui permet d'envoyer des paquets à toutes les machines du sous-réseau qui écoutent sur un port donné.

En général, envoyer des paquets vers une adresse de broadcast n'a de sens qu'avec UDP (sauf usages avancés). Pourquoi ?

Adresses Multi-cast

Il existe certaines adresses IP réservée (celles de la classe D) appelées adresses multi-cast.
Il est interdit de donner ces adresses à des périphériques réseau. Elles fonctionnent comme les adresses de broadcast mais il n'y a pas besoin de connaître la configuration du réseau sur lequel on est.

Pour le réseau local, ce sont les adresses de l'intervalle : **`224.0.0.0`** à **`224.0.0.255`** (exclus).

Fonctionnement des couches réseau basses

En général, au niveau de la couche réseau matériel (par exemple Ethernet), toutes les machines qui sont sur le même réseau physique voient toutes les informations passer. Les périphériques réseau ignorent simplement les paquets/trames qui ne leurs sont pas destinés.

Note : au niveau Wifi, on peut isoler les nœuds les uns des autres et donc un périphérique réseau ne voit que les informations qui lui sont destinées explicitement

Les routeurs propagent les paquets à destination d'adresses multi-cast/broadcast sur tout le réseau local mais ne les autorisent pas à sortir du sous-réseau

Comment s'utilise l'adresse de diffusion ?

```
//Fonctionne si on est sur le bon sous-réseau
InetAddress i1 = InetAddress.getByName("129.175.255.255");

//Fonctionne sur n'importe quel sous-réseau
InetAddress i = InetAddress.getByName("224.0.0.29");

//Le reste est identique on peut créer des DatagramSocket
//sur ces adresses et envoyer des paquets !
//C'est bien sur pour le client, on ne peut/doit pas
//faire bind sur ces adresses ?
```

Quel est l'utilité de la diffusion multi-cast ?

◆ Ne demande pas de négocier d'adresse IP à l'avance (on n'a pas besoin de connaître l'IP du serveur)
◆ Permet d'envoyer la même information à plusieurs machines sans utiliser beaucoup de ressources

Inconvénient : cela repose sur UDP. Quelques exemples :

◆ Streaming vidéo/audio sur un réseau local (jeu par exemple)
◆ Découverte/négociation d'adresse entre un client et un serveur

Plan

Application de chat

On souhaite réaliser une application de discussion sur un réseau local, avec les propriété suivantes :

◆ Zéro configuration : on démarre l'application et elle découvre toute seule où est le « serveur »
◆ Discussion en salon (façon IRC) : une machine centralise les discussions
◆ On peut joindre à tout moment et quitter à tout moment, même si notre machine fait office de serveur

On va concevoir l'architecture logicielle et les protocoles permettant de faire ça

Une fois l'application lancée...

Quel protocole (transport) utiliser pour l'échange de message ?
⇒ TCP est approprié (messages de longueur arbitraire, pas de perte, ...).
Quel impact sur les choix d'implémentation pour le serveur ?
⇒ Le serveur doit garder une connexion par client et gérer les connexions via du multi-threading
⇒ Le serveur doit aussi, sur réception d'un message d'un client le transmettre à tous les autres clients ?
Quel impact sur les choix d'implémentation pour le client ?
⇒ Les échanges de messages entre client et serveur ne suivent pas un ordre fixé. Le client doit pouvoir envoyer un message dès que l'utilisateur le saisi, mais doit aussi être à l'écoute du serveur et afficher les messages dès leur reception.
⇒ Le client est donc aussi probablement multi-thread

Quelques détails du protocole application

Que doit prendre en compte le protocole application ?
⇒ Toute information pertinente pour le serveur doit être retransmise aux clients (arrivée/départ d'un client, message envoyé, déconnexion, ...). Il est sans doute judicieux d'ajouter un timestamp aux messages d'application. ⇒ Si le serveur décide de quitter, il choisit parmi ces clients un nouveau serveur et diffuse son IP à tous les clients, ouvrent de nouvelles connexions.

Zéro-configuration

Un serveur diffuse à intervalle régulier son adresse IP via des paquets UDP de broadcast
⇒ Lorsque l'application démarre, elle écoute pendant un certain temps les paquets UDP. Si elle reçoit un paquet de serveur, elle s'y connecte (en TCP).
⇒ Au démarage, si aucun paquet UDP n'arrive, l'application décide de jouer aussi le rôle de serveur et se met à diffuser sa propre IP

Plan

Java est enfin un langage fonctionnel

Java 8 apporte des traits fonctionnels au langage. Intérêt de la programmation fonctionnelle :

- Code souvent plus compact
- Meilleure réutilisation du code
- Code potentiellement plus efficace
- Code souvent plus lisible (si on en a l'habitude)

Trois ingrédients :

Les interfaces fonctionnelles
: Une manière de déclarer le type de fonctions

Les lambdas-abstraction
: Une manière de définir des petites fonctions à la volée

La *Stream API*
: Une bibliothèque d'itérateurs efficaces qui vient s'ajouter aux **`Collections`**

Interfaces fonctionnelles

Une interface fonctionnelle est simplement une **`interface`** contenant une seule méthode abstraite. Cette utilisation permet de définir des types de « fonctions ».

`java.util.function.Function<X,Y>` :
    représente le type d'une fonction de **`X`** vers **`Y`**
`java.util.function.Predicate<X>` :
    représente le type d'une fonction de **`X`** vers **`boolean`**
`java.util.function.Consumer<X>` :
    représente le type d'une fonction de **`X`** vers **`void`**

Exemple

```
  class MyIncr implements Function<Integer,Integer> {
  //Doit s'appeler apply

  Integer apply(Integer x) { return x + 1; }
}


  class MyOdd implements Predicate<Integer> {
  //Doit s'appler test

  boolean test(Integer x) { return x % 2 == 1; }
}

  class MyPrint implements Consumer<Integer> {
  //Doit s'appler accept

  void accept(Integer x) { System.out.println(x); }
}
```

Stream API

Nouvelle API de Java 8. Permet de définir de manière déclarative des itérateurs sur des flux. Étant donné un objet qui implémente l'(ancienne) interface **`Collection<E>`** (par exemple **`Vector<E>`**, **`List<E>`**, **`Set<E>`**) :

```
Vector<Integer> v = …;
  …
Stream<Integer> s = v.stream();
```

Que gagne t'on ?

On va pouvoir définir des itérateurs persistants sur les flux, paresseux et parallélisable

Méthodes de l'interface `Stream<T>`

`.count()` :
renvoie le nombre d'éléments dans le flux

`.skip(long n)` :
renvoie un flux où **`n`** éléments ont été sautés

`.limit(long n)` :
renvoie un flux avec uniquement les **`n`** premiers éléments

`.distinct()` :
renvoie un flux où les doublons (au sens de **`.equals(Object o)`** sont retirés

`.sorted()` :
renvoie un flux où les éléments (qui doivent implémenter **`Comparable`**) son triés

Quelle différence avec les opérateurs sur les Collections ?

Les collections sous-jacentes ne sont pas détruites!

Méthodes de l'interface Stream<T> (suite)

`.filter(Predicate< ? super T> p)` :
renvoie le flux de tous les éléments pour lesquels le prédicat **`p`** renvoie vrai (**`p`** peut être défini sur un super-type de T)

`.allMatch(Predicate< ? super T> p)` :
renvoie vrai si tous les éléments satisfont le prédicat **`p`**

`.anyMatch(Predicate< ? super T> p)` :
renvoie vrai si un élément satisfait le prédicat **`p`**

`.map(Function< ? super T, ? extends R> f)` :
renvoie le flux d'éléments de type **`R`** obtenu par application de **`f`** sur tous les éléments du flux de départ.

`.forEach(Consumer< ? super T> c)` :
applique **`c`** tour à tour à tous les éléments du flux.

`.collect(…)` :
transforme le flux en Collection. On peut utiliser **`Collectors.toList()`**, **`Collectors.toSet()`**, ...

Exemple

Supposons que l'on a un vecteur d'**`Integer v`**. On souhaite, tous les incrémenter, les trier, puis afficher ceux qui sont impairs :

```
Vector<Integer> v = … ;
v.stream().map(new MyIncr())
          .sorted()
          .filter(new MyOdd())
          .forEach(new MyPrint());
```

Exercice, faire la même chose sans passer par les flux

λ-abstraction

Il est malcommode de définir les opérations à passer aux itérateurs dans des classes (**MyIncr**, ..., dans l'exemple précédent). Java 8 fournit une syntaxe commode pour définir des fonctions :

```
Vector<Integer> v = … ;
v.stream().map(i -> i+1)
          .sorted()
          .filter(i -> i % 2 == 1)
          .forEach(i -> System.out.println(i));
```

La syntaxe générale est :

$(Type_1\ p_1, \ldots, Type_n\ p_n)$ -> *corps*

Le corps peut être soit une expression simple soit un bloc :

```
(Integer i) -> i+1
(Integer i) -> {
        System.out.println(i);
        return i+1;
}
```

λ-abstraction (suite)

Avantage des λ-abstractions ?

- Syntaxe compacte mais lisible
- Localité de la définition (on n'est pas obligé de mettre le code dans une classe ailleurs)
- Accès à l'environnement local (à l'inverse des classes anonymes)

```
PrintWriter out = …;
…
v.stream().forEach(i -> out.println(i) );
```

Que gagne t'on à utiliser la Stream API ?

- Persistance : les collections sous-jacentes ne sont pas détruites/modifiées, pas besoin de faire des copies
- Paresse : les opérations ne sont faites que si c'est nécessaire :

```
Stream<Integer> vs = v.stream().sorted();
if (do_print)
      vs.forEach(i -> System.out(i));
//le tri n'est fait que si on force (en utilisant .forEach())
```

- Les opérations peuvent se faire le plus possible en parallèle, si on utilise `.parallelStream()` au lieu de `.stream()` pour récupérer le flux
- implémente le même paradigme que Map/Reduce

Plan

Pourquoi ?

Une opération courante en informatique est la sérialisation (et désérialisation) des données, c'est à dire passer d'une représentation «en mémoire» à une représentation « fichier » (par exemple pour enregistrer dans un fichier ou envoyer sur le réseau). Pourquoi est-ce nécessaire ? entre autre, à cause des pointeurs !

Sérialisation en Java

Java propose un mécanisme de sérialisation des objets. Chaque objet que l'on veut sauvegarder doit implémenter l'interface :

`java.io.Serializable`

C'est une interface vide, traitée spécialement par le compilateur Java. Tout objet qui implémente cette interface peut être sauvegardée dans un flux (fichier, réseau, ...) et rechargé ensuite. Les attributs ainsi que le code est sauvegardé.

Beaucoup de classes de la bibliothèque standard implémentent cette interface (tous les types de bases, **`Integer`** ,**`String`**, ..., ainsi que les collections telles que **`Vector`**, ...).

Lecture/Écriture

On utilise deux classes du *package* **`java.io`** pour lire et écrire des objets :

ObjectOutputStream
: possède une méthode **`.writeObject(Object)`** qui permet d'écrire un objet dans le flux

ObjectInputStream
: possède une méthode **`.readObject()`** qui permet d'écrire un objet dans le flux

`.readObject()` renvoie un **`Object`**. On est obligé de faire un *cast* pour récupérer le type de l'Objet. C'est une opération non sûre !

Si le type de l'objet contient des *Generics* alors on ne peut pas utiliser **`instanceof`** et on obtient un warning du compilateur. On peut supprimer ce dernier avec **`@SuppressWarning("unchecked")`**

Exemple

```
  void save(Vector<String> v) {
   ObjectOutputStream out = new ObjectOutputStream(
                              new FileOutputStream("fichier.obj"));
   try {
     out.writeObject(v);
   } catch (IOException e) {
   } finally  { out.close(); };
}

 Vector<String> load() {
   ObjectInputStream in = new ObjectInputStream(
                              new FileInputStream("fichier.obj"));
   try {
      Object o = in.readObject();
      @SuppressWarning("unchecked")
      Vector<String> v = (Vector<String>) o;
      return v;
      } catch (IOException e) {
      } finally  { in.close(); };
      return null;
}
```

Gestion des version

Il est conseillé d'ajouter aux classes sérialisables un attribut

```
static final long serialVersionUID = 12345L;
```

Ce dernier peut être généré automatiquement par Eclipse. S'il n'est pas présent le compilateur Java en insère un automatiquement.

Le runtime Java utilisera cet identifiant pour déterminer qu'un Objet sauvegardé est bien du même type que celui attendu. En particulier il faut changer le numéro quand on ajoute/supprime des champs ou qu'on en change le type.