Département Informatique

Programmation répartie

Programmation clients / serveurs UDP

Durée : 1 séance
M4102 -N4P2
2016–2017

Rappel : Toujours avoir un accès à la doc sous la main (et donc à l'API). Cette documentation est probablement accessible directement dans Eclipse (moins pratique pour avoir une vue globale sur une API) et en locale.

Résumé

Dans ce TP, vous allez mettre en œuvre une communication UDP orientée sans connexion et asynchrone.

Principes généraux

Socket

Un *socket* est l'interface de communication via la pile réseau TCP/IP. Il permet à un processus d'envoyer et/ou de recevoir des données vers d'autres processus distant ou non. Ainsi, deux applications souhaitant communiquer en réseau doivent *ouvrir*[1] un socket pour communiquer ensemble. Un socket est identifiable par un couple "adresse IP / numéro de port". À un instant donné, il est attaché à un seul processus.

En java, c'est l'objet java.net.InetSocketAddress qui permet d'associer au sein d'un seul objet les deux caractéristiques d'un socket. Attention, cette classe ne représente pas un socket. C'est simplement un identifiant couplant l'IP et le port et permettant de simplifier les manipulations.

L'API Java reconnaît deux types de socket. L'un pour les communications orientées sans connexion via des datagrammes UDP (que nous allons voir dans ce TP) et l'autre avec connexion (via des segments TCP que nous verrons dans un prochain TP).

Communication orientée sans connexion

Le mode de communication dit *sans connexion* est le plus simple qui soit. Il n'y a pas de gestion des pertes de paquets. La seule garantie apportée par ce mode de communication est que lorsque les données transférées sont reçues par l'application de destination, alors ces données peuvent être considérées comme valides (le datagramme a été vérifié au niveau Liaison de données, au niveau IP, et également à l'aide d'une somme de contrôle sur le pseudo en-tête UDP au niveau transport).

De même, lors de l'envoi de plusieurs datagrammes consécutifs, rien ne garantie que l'ordre d'arrivée sera identique à celui de départ (indépendances des paquets).

En JAVA

En java, l'API principale pour les communications réseaux se trouve dans le paquetage `java.net`. Pour la programmation d'applications client/serveur en UDP, Java utilise les classes suivantes java.net :

- `java.net.DatagramPacket` : pour la conception des datagrammes (gestion des données, de l'émetteur et de la destination).
- `java.net.DatagramSocket` : pour la gestion de la communication proprement dite (envoie et réception des datagrammes).
- `java.net.MulticastSocket` : pour la gestion de la communication en multi-diffusion (adresse de classe D en IPv4 par exemple).

Pour réaliser une communication entre deux applications, il faudra donc suivre les étapes suivantes :

1. Créer un socket UDP assurant la réservation d'un port de communication de la machine locale avec votre application ;
2. Configurer la connexion (**Attention**, malgré ce que semble indiquer certaines méthodes de l'API, il n'y a pas de connexion en UDP au sens cité plus haut) ;
3. Effectuer l'envoi et/ou la réception des données via ce socket à l'aide de datagrammes UDP.

Remarques

- **Réception** d'un datagramme :
  Un espace mémoire permettant le stockage des données reçues est nécessaire. C'est à vous de réserver cet espace mémoire. Java ne connaissant pas le type des données à recevoir. Ces données seront donc à placer dans un *tableau d'octets* qui est la forme de données la plus générique en Java. Elle devra avoir une taille suffisante, fonction du protocole de communication que vous utilisez.
- **Envoie** d'un datagramme :
  a. Les données utilisateurs sont à préparer pour l'encapsulation dans le datagramme. Ceci se fait avec les même conditions que pour la réception (i.e. : données à placer dans un tableau d'octets) ;

1. obtenir l'accès à

b. UDP étant un mode sans connexion, l'identification du (socket) destinataire est à indiquer pour chaque datagramme à envoyer.

Exercice 1 : Préliminaires

Q 1 Quelle est le volume maximal théorique exacte de données contenues dans un datagramme UDP (justifier) ?
Q 2 Écrivez un programme simple qui utilise une classe du paquetage java.net pour retrouver la liste des adresses IP associées à chacune des interfaces réseau active de votre machine, ainsi que la taille de la MTU [2] qui y est associée.
Q 3 Comparez les valeurs de MTU à la taille maximal d'un datagramme UDP ? Quelle sera la taille optimal d'un datagramme UDP [3] ?

Exercice 2 : Démarrage en douceur

Q 1 Lisez la page du manuel de la commande netcat.
Q 2 Voici le code d'un serveur UDP. Recopiez ce code et testez le avec la commande netcat. Que fait ce service ?

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class ServeurUDP {
  private DatagramSocket dgSocket;

  ServeurUDP(int pSrv) throws IOException {
    dgSocket = new DatagramSocket(pSrv);
  }

  void go() throws IOException {
    DatagramPacket dgPacket = new DatagramPacket(new byte[0], 0);
    String str;

    while (true) {
      dgSocket.receive(dgPacket);
      System.out.println("Datagram␣received␣from␣" + dgPacket.getSocketAddress());

      dgPacket.setSocketAddress(dgPacket.getSocketAddress());
      str = new java.util.Date().toString() + "\n";
      byte[] bufDate = str.getBytes();
      dgPacket.setData(bufDate, 0, bufDate.length);

      dgSocket.send(dgPacket);
    }
  }

  public static void main(String[] args) throws IOException {
    final int DEFAULT_PORT = 9876;
    new ServeurUDP( args.length == 0 ? DEFAULT_PORT : Integer.parseInt(args[0]) ).go();
  }
}
```

Q 3 Réalisez un programme ClientUDP.java *"se connectant"* à ce serveur et affichant en console les informations reçues. L'adresse et le numéro de port du serveur pourront être entrée en ligne de commande.

Exercice 3 : Service de bégaiement

Pour ce service, le client envoi une chaîne de caractères au serveur contenant un caractère représentant un chiffre n de 0 à 9 suivi d'une chaîne de caractère quelconque (une phrase).

Lors de la réception du message client, le serveur affiche dans sa propre console (le terminal à partir duquel il a été lancé) les informations sur le datagramme reçu : l'adresse et le port de l'émetteur, la valeur n du chiffre reçu, ainsi que la phrase reçue.

En condition normale, le serveur renvoie au client une chaîne de caractères contenant le caractère 0 indiquant que qu'il n'y a pas d'erreur, puis la chaîne reçue transformée de telle manière que tous les mots sont écrit n fois.

En cas d'erreur (chiffre manquant au début de la chaîne reçue), le serveur renverra le caractère 1 suivi du message « Erreur : multiplicateur manquant ».

Exemples :

- Si le client envoi : « 2Bonjour le monde », le serveur renvoi : 0Bonjour Bonjour le le monde monde.
- Si le client envoi : « Bonjour le monde », le serveur renvoi : 1Erreur : multiplicateur manquant.
- Si le client envoi : « 0Bonjour le monde », le serveur renvoi : « 0 »

2. Maximum Transfert Unit : volume maximal de données transportable dans une trame
3. Cette taille s'appelle la MSS (Maximum Segment Size) : Taille maximale de données transportable dans un datagramme IP

Q 1 Réalisez le serveur. Testez-le avec la commande netcat.
Q 2 Réalisez un client qui n'affichera en console que le message reçu du serveur sans le code d'information, c'est à dire sans le premier caractère reçu.

Exercice 4 : Service de bégaiement 2

Nous allons améliorer le service précédent. Le message envoyé aura maintenant le format suivant :
Un nombre (et non plus un chiffre), le caractère « : » utilisé comme séparateur, puis une chaîne de caractère.
Exemple : "56 :bla bla bla"

Q 1 Réaliser un serveur qui fonctionne comme le précédent mais avec le nouveau format de message.

Q 2 Que se passe-t-il si votre message est "4000 :Bonjour le monde" ? Proposez une solution.

Exercice 5 : Encore un chat !

Mais celui-là sera en UDP et en multi-diffusion.

Si on souhaite envoyer un même message à un grand nombre de personnes ne se trouvant pas obligatoirement sur le même réseau, il est préférable de diffuser ce message en utilisant un protocole de diffusion multi-points qui prendra en charge la transmission vers les destinataires. Ce type de protocole est associé à un routage multi-destinataire, plus efficace que le simple routage point à point et l'engorgement des réseaux s'en trouve ainsi réduit.

En pratique, on utilise les adresse IP de classe D comme une adresse d'abonnement à une liste de diffusion. Ainsi, toutes machines ayant un accès réseau pourra envoyer un message à destination de tous les abonnés à cette adresse de classe D. De plus, toutes machines ayant effectuée un abonnement sur une adresse IP de classe D pourra recevoir les messages envoyés vers cette adresse. Seul le serveur multi-cast (le routeur en général) connaît donc la liste des abonnés à une adresse de classe D.

Il existe plusieurs types d'adresse IP de classe D. Certaines ne peuvent pas traverser les routeurs (à la manière des adrsse privées en classe A, B et C). Un routeur connaissant des protocoles de diffusion multi-destinataires (multicast), s'il est normalement configuré ne transmet pas les adresses entre 224.0.0.0 à 224.0.0.255 inclus. Ainsi, vous pouvez à loisir utiliser ces adresses pour vos expérimentations, c'est ce que vous ferez dans ce TP.

Mise en oeuvre en Java

En java, c'est la classe `java.net.MulticastSocket` qui étend les possibilités de la classe `DatagramSocket` au multicast en ajoutant les méthodes d'abonnements. La gestion des messages se fera donc par le protocole UDP et la classe `DatagramSocket` dont elle reprend toutes les méthodes.

Les principales opérations supplémentaires sont donc :
- Adhérer à un groupe
  `joinGroup(InetAddress) throws IOException`
- Quitter un groupe
  `leaveGroup(InetAddress) throws SocketException`

Remarque : Afin de limiter la zone atteinte par un message multidestinataire, il est bon de régler le TTL de ces messages à une valeur pas trop élevée. Ce dernier représente le nombre de routeurs que le message peut traverser avant d'être abandonné faute d'avoir atteint sa destination (0 pour l'émetteur, 1 pour le (sous-)réseaux local, 2 pour le suivant …).

Q 1 Rechercher sur le site de l'IANA, la liste des adresses multicasts réservée à l'entreprise Walt Disney.

.....

Q 2 En utilisant un socket multicast, écrivez deux classes :
- l'une permettant de lire les messages d'un groupe multicast donné en argument (couple adresse IP / port) et de les afficher sur la console
- l'autre permettant à l'utilisateur de rentrer des petites lignes de texte et qui les diffuse sur le groupe multicast donné en argument (couple adresse / port).

Exemples :

Fenêtre affichage	Fenêtre saisie
> java ReceptionMsg 224.0.0.24 9876	> java DiffuseMsg 224.0.0.24 9876
	Bonjour, je suis l'afficheur
`Phrase :` *Message en une ligne*	frene05 dit : Message en une ligne
`Phrase :` *FIN*	

Testez vos deux programmes ensembles sur le même groupe multicast que vous choisirez parmi les adresses non assignées (c.f. IANA).

Q 3 Bonus Écrire une petite application qui prend en argument de la ligne de commande : une adresse multicast, un numéro de port et une suite quelconque de mots ou phrases. Cette application devra afficher dans le shell courant tout ce qui est reçu sur l'adresse multicast et y envoyer les mots ou phrases en ligne de commande. En utilisant tous le même numéro de port et la même adresse IP, par exemple 224.0.0.24 : 9876, vous pouvez obtenir une version basique d'un forum de discussion. Il ne reste plus qu'a séparer l'entrée et la sortie via une interface graphique par exemple.