Programmation Réseau

API Java TCP

Jean-Baptiste.Yunes@univ-paris-diderot.fr

UFR Informatique

2012-2013

Les flux réseau en Java

- pré-requis : utiliser correctement les classes d'entrées-sorties Java (`package java.io`)
- le premier package à connaître est `java.net`
- deux classes importantes :
  - `ServerSocket`
  - `Socket`

- `ServerSocket`
  - `ServerSocket(int port);` créé une socket attachée au port spécifié et permettant de contrôler les demandes de connexion entrantes
  - la méthode `Socket accept();` permet d'attendre une demande de connexion entrante, de l'accepter lorsqu'elle arrive et de fournir en retour une `Socket` de service autorisant la communication

```
// squelette de serveur
ServerSocket socketAttente = new ServerSocket(PORT);
do {
    Socket service = socketAttente.accept();
    // connexion établie
    // la communication est désormais possible
    // bla bla bla bla...
    service.close();
} while (true);
socketAttente.close();
```

- `Socket`
  - `Socket(String nom,int port);` permet de créer une `Socket` attachée à un port libre local depuis laquelle une demande de connexion est lancée vers la `ServerSocket` de la machine de nom donné et attachée sur le `port` spécifié, lorsque la connexion est acceptée le constructeur termine normalement et la `Socket` est utilisable pour communiquer

```
// squelette de la partie cliente
Socket service =
        new Socket("be.bop.a.lu.la",PORT);
// connexion établie
// la communication est désormais possible
// bla bla bla bla...
service.close();
```

- `Socket`
  - une socket TCP étant par nature un objet de communication bi-directionnel, il est possible de récupérer de chaque côté les flux de lecture et d'écriture

- `InputStream getInputStream();` permet d'obtenir le flux de lecture correspondant aux écritures du pair
- `OutputStream getOutputStream();` permet de récupérer le flux d'écriture correspondant au flux de lecture du pair

```
// squelette permettant de récupérer les flux d'entrées/sorties
InputStream is = service.getInputStream();
OutputStream os = service.getOutputStream();
// bla bla bla bla...
```

- On remarquera qu'une fois la connexion établie, il n'y a plus de distinction dans les rôles :
  - les sockets sont rigoureusement symétriques
  - la communication est
- C'est la partie applicative qui est désormais en charge du protocole d'échange
  - qui envoie quoi, quand, etc.

API Java

```
import java.net.*;
import java.io.*;

public class Serveur {
  public static final int PORT = 11111;
  public static void main(String []arguments) {
    try {
      ServerSocket socketAttente = new ServerSocket(PORT);
      do {
        Socket service = socketAttente.accept();
        BufferedReader bf = new BufferedReader(new InputStreamReader(service.getInputStream()));
        String qui = bf.readLine();
        System.out.println(qui+" : vient de se connecter");
        Thread.sleep(5000);
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(service.getOutputStream()));
        pw.println("j'ai bien reçu ton message "+qui);
        pw.close();
        bf.close();
        service.close();
      } while (true);
    } catch(Exception e) {
      System.err.println("Erreur sérieuse : "+e);
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

```
import java.net.*;
import java.io.*;

public class Client {
  public static final int PORT = 11111;
  public static void main(String []arguments) {
    try {
      Socket service = new Socket("localhost",PORT);
      PrintWriter pw = new PrintWriter(new OutputStreamWriter(service.getOutputStream()));
      pw.println(arguments[0]);
      pw.flush();
      BufferedReader bf = new BufferedReader(new InputStreamReader(service.getInputStream()));
      String message = bf.readLine();
      System.out.println("Je viens de recevoir le message : "+message);
      pw.close();
      bf.close();
      service.close();
    } catch(Exception e) {
      System.err.println("Erreur sérieuse : "+e);
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

- Un problème important se pose dans avec le squelette de serveur employé

  - en effet, pendant que le service se déroule la `ServerSocket` n'est pas dans un état acceptant (i.e. pas dans l'appel à `accept()`) et donc les communications entrantes sont bloquées

  - le serveur tel qu'il est conçu sérialise les communications, ce mode n'est en général utile que dans le cas où les communications sont (très) courtes

  - c'est un serveur

- Sinon (et donc en général), il est nécessaire de paralléliser les traitements de service, en déléguant le service :
  - à un autre processus
  - à un autre `Thread`

Les threads Java

- la machine Java (JVM, aka Java Virtual Machine) possède son propre système de gestion d'exécution

  - chaque instance de JVM possède un ordonnanceur capable gérer dans le même espace mémoire différentes exécutions concurrentes

    - chacune de ces exécutions s'appelle un

- Pour l'instant, il est utile de savoir :
  - Construire un `Thread` :
    - soit par instanciation d'une sous-classe qui surcharge la méthode `void run();`
    - soit par association avec une classe qui implémente `Runnable` et donc la méthode `void run();`

- Pour qu'un `Thread` commence son exécution il faut utiliser sa méthode `void start();`
  - à cet instant une nouvelle exécution concurrente démarre
  - il est possible à tout `Thread` d'attendre la terminaison d'un autre `Thread` par appel à `void join();` sur le `Thread` à attendre
    - équivalent pour les `Threads` de `wait()` pour les processus…
    - on notera que la terminaison d'un `Thread` est obtenue par terminaison de l'appel à `run();`

- Il est possible de forcer un `Thread` à suspendre son exécution par appel à la méthode statique `sleep(int ms);`

  - qui a pour effet de suspendre le `Thread` qui fait appel à cette méthode pour la durée exprimée

```
class DuCode implements Runnable {
  private String nom;
  private Random alea;
  public DuCode(String nom) {
    this.nom = nom;
    alea = new Random();
  }
  public void run() {
    for (int i=0; i<10; i++) {
      try {
        Thread.sleep(alea.nextInt()%5*1000);
      } catch(Exception e) {
      } finally {
        System.out.println("Je suis "+nom+" i="+i);
      }
    }
  }
}
```

```
public class Essai {
  public static void main(String []argument) {
    DuCode code1 = new DuCode("jacques");
    DuCode code2 = new DuCode("émile");
    Thread t1 = new Thread(code1);
    Thread t2 = new Thread(code1);
    Thread t3 = new Thread(code2);
    t1.start();
    t2.start();
    t3.start();
    try {
      t1.join();
      t2.join();
      t3.join();
    } catch(Exception e) {
    }
  }
}
```

Serveur multithreadé

- Il suffit de déléguer l'exécution du service à un `Thread` dédié...

```
class Service implements Runnable {
  private Socket maChaussette;
  Service(Socket s) { maChaussette = s; }
  void run() {
    // bla bla bla bla...
    maChaussette.close();
  }
}
...
// squelette de serveur
ServerSocket socketAttente = new ServerSocket(PORT);
do {
    Socket s = socketAttente.accept();
    // connexion établie
    // la communication est désormais possible
    Thread t = new Thread(new Service(s));
    t.start(); // on démarre l'exécution concurrente du service
} while (true);
socketAttente.close();
```