Algorithmique parallèle et distribuée :
Les bases de la communication réseau en Java

Julien Rossit

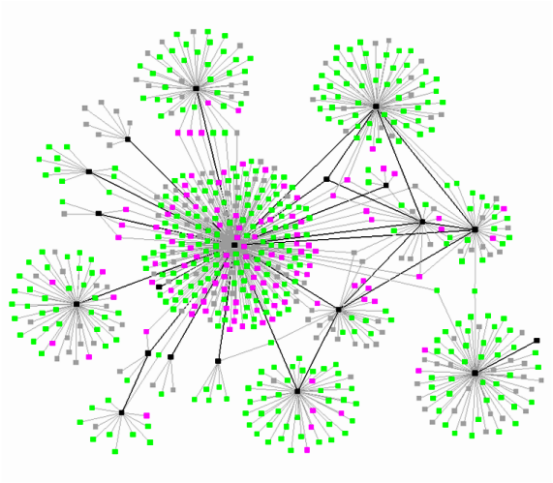
julien.rossit@parisdescartes.fr

IUT Paris Descartes

Motivations :

Problèmes rencontrés par l'algorithmique répartie :

- Comment identifier et contacter une autre machine sur un réseau ?
- Comment transmettre efficacement des données à cette machine ?



Protocoles réseau :

ensemble de contraintes permettant d'établir une communication entre deux machines.

Les protocoles réseaux

Permettent de **transférer des données** d'une machine à l'autre :

- s'appuient sur les protocoles réseau de plus bas niveau (*IP*) pour routage, transfert nœud à nœud, etc ;
- servent de socles pour les protocoles applicatifs (RPC, HTTP, FTP, DNS, etc).

Fournissent des **API** pour envoyer et recevoir des données :

- *TCP* : flux bi-directionnel de communication ;
- *UDP* : mécanisme d'envoi de messages ;
- *Multicast-IP* : envoi de message à un groupe de destinataires ;
- etc.

Caractéristiques des protocoles

Deux primitives de communication :

- *send* : envoi d'un message dans un buffer distant ;
- *receive* : lecture d'un message à partir d'un buffer local.

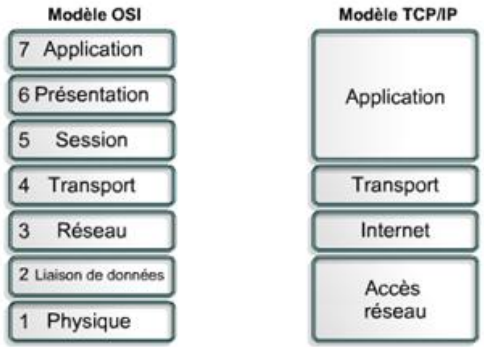
Propriétés associées :

- **Fiabilité** : est-ce que les messages sont garantis sans erreur ?
- **Ordre** : est-ce que les messages arrivent dans le même ordre que celui de leur émission ?
- **Contrôle de flux** : est-ce que la vitesse d'émission est contrôlée ?
- **Connexion** : les échanges de données sont-ils organisés en session ?

Le modèle TCP/IP

TCP/IP a été conçu pour répondre à certain critères :

- fractionnement des données en paquets ;
- utilisation d'un système d'adresses ;
- acheminement des données sur le réseau ;
- contrôle des erreurs de transmission de données.



TCP/IP est un modèle en couche : chacune utilise les services de la couche inférieure et en fournit à celle de niveau supérieur :

- Couche Application : applications standard du réseau ;
- Couche Transport : contrôle de flux et correction des erreurs ;
- Couche Internet : achemine les données à destination ;
- Couche Réseau : spécifie la forme des données.

Les données (paquets d'informations) sont traitées successivement par chaque couche qui vient rajouter un élément d'information (en-tête).

Modes de transfert

Deux modes de transfert :

- *synchrone* : les primitives sont bloquantes ;
- *asynchrone* : les primitives sont non bloquantes.

Exemples :

- send sync. et receive sync. : send reste bloqué jusqu'à l'exécution de receive
- send async. et receive sync. (Java) : send envoie et se termine, receive reste bloqué jusqu'à ce qu'il y ait un message à lire

En résumé : modèle asynchrone plus souple, mais programmes en mode synchrone plus simples à écrire.

Comment combiner les deux approches ?

receive synchrone + multi-threading

Modes de transfert

Deux modes de transfert :

- *synchrone* : les primitives sont bloquantes ;
- *asynchrone* : les primitives sont non bloquantes.

Exemples :

- send sync. et receive sync. : send reste bloqué jusqu'à l'exécution de receive
- send async. et receive sync. (Java) : send envoie et se termine, receive reste bloqué jusqu'à ce qu'il y ait un message à lire

En résumé : modèle asynchrone plus souple, mais programmes en mode synchrone plus simples à écrire.

Comment combiner les deux approches ?

receive synchrone + multi-threading

L'adressage des machines distantes

Le protocole IP *Internet Protocol*

Chaque machine est associée à une adresse unique :

- cette adresse est codable sur 4 octets (32 bits) ;
- elle représentée par 4 entiers séparés par des '.' ;
- 127.0.0.1 (*loopback*) fait référence à la machine locale.

Le protocole IP

enveloppe chaque paquet de données en y ajoutant différentes informations :

- l'adresse IP de l'expéditeur ;
- l'adresse IP du destinataire ;
- etc.

Les *serveurs DNS* assurent la correspondance entre adresses IP et adresses symboliques (ex : www.parisdescartes.fr).

Implémentation en Java

La classe **Java java.net.InetAddress** permet de manipuler les adresses Internet.

Cette classe ne contient pas de constructeur !

Pour obtenir les objets :

- *static getByName(String)* : l'adresse IP de la machine à partir de son nom ;
- *static getAllByName(String)* : toutes les adresses IP de la machine ;
- *static getLocalHost()* : l'adresse IP machine locale ;
- etc.

Méthodes disponibles sur ces objets :

- *String getHostName()* : nom de la machine ;
- *byte[] getHostAddr()* : adresse de la machine (un tableau de 4 octets) ;
- *String toString()* : liste le nom de la machine et son adresse ;
- *boolean isReachable(int timeout)* : teste si une la machine est joignable
- etc.

http ://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html

La sous classe java.net.Inet6Address gère les particularités de IPv6.

Le protocole TCP

Transmission Control Protocol

Le protocole TCP introduit la notion de *port* : permet la communication vers plusieurs applications s'exécutant sur une même machine.

TCP est capable d'établir une communication sûre entre applications :

- découpe les gros paquets de données en paquets plus petits ;
- vérifie que le destinataire soit prêt à recevoir les données.
- numérote les paquets, vérifie la réception, de redemander les paquets manquants, ré-assemble avant de transmettre vers la couche application.

Exemples d'utilisation : HTTP, FTP, Telnet, SMTP, POP, etc.

Propriétés du protocole TCP

Taille des messages :

- quelconque ;
- envoi en général bufferisé ;
- vidage autoritaire des buffers possible.

Ordre des messages :

- garantie que l'ordre d'émission respecte l'ordre de réception ;
- garantie de non duplication des messages.

Perte de messages :

- acquittement des messages envoyés ;
- timeout de ré-émission des messages en cas de non réception de l'acquittement.

Contrôle de flux :

- éviter qu'un émetteur trop rapide fasse "déborder" le buffer du récepteur ;
- blocage de l'émetteur si nécessaire.

Principe de la connexion TCP

Déroulement par étapes d'une connection TCP :

1. Le serveur crée une socket et attend une demande de connexion.
2. Le client envoie une demande de connexion.
3. Acception explicite de la demande par le serveur (*triple poignée de main*).
4. dialogue client/serveur en mode *flux* :
   - échange bi-directionnel ;
   - distinction rôle client/serveur artificielle.
5. fermeture de connexion à l'initiative du client ou du serveur (vis-à-vis notifié de la fermeture).

Les socket actives : la classe java.net.Socket

Constructeurs :

- *Socket(String host, int port)* ;
- *Socket(InetAddress address, int port)* ;
- etc.

Accesseurs :

- *getPort(), getLocalPort()* : ports distant/local ;
- *getInetAddress(), getLocalAddress()* : adresse distante/locale ;
- *getInputStream()* : flux d'entrée ;
- *getOutputStream()* : flux de sortie ;

Autres méthodes :

- *close()* : fermeture de la socket ;
- *setSoTimeout(int)* : fixe le delai d'attente (si 0 receive bloquant) ;
- accès aux options TCP : timeOut, soLinger, tcpNoDelay, keepAlive,.. ;
- etc.

http ://docs.oracle.com/javase/7/docs/api/java/net/Socket.html

Ecriture d'un processus client

Principe général des étapes de mise en oeuvre :

1. Création d'une socket active vers la machine distante.
   new Socket(...)
2. Accès aux flux d'entrée et de sortie de la socket.
   getInputStream(), getOutputStream()
   (Encapsulation suivant les traitement à réaliser.)
3. Dialogue réseau via les flux et réalisation des traitements.
   ...

Exemple de client Java

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

class TCPClient {
  public static void main(String argv[]) throws Exception {
    String request;
    String answer;

    BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

    \\ Création de la socket client, demande de connexion
    Socket clientSocket = new Socket("localhost", 8080);

    \\ Création du flux en sortie
    PrintWriter outToServer = new PrintWriter(
                                  new BufferedWriter(
                                    new OutputStreamWriter(
                                      clientSocket.getOutputStream())),
                                      true);

    \\ Création du flux en entrée
    BufferedReader inFromServer = new BufferedReader(
                                      new InputStreamReader(
                                      clientSocket.getInputStream()));

    sentence = request.readLine();

    \\ Emission des données au serveur
    outToServer.println(request);

    \\ Lecture des données arrivant du serveur
   answer = inFromServer.readLine();

    System.out.println("FROM SERVER: " + answer);

    clientSocket.close();
  }
}
```

Les socket passives : la classe java.net.ServerSocket

Constructeurs :

- *ServerSocket(port)* (si 0, port choisi par TCP) ;
- etc.

Accesseurs :

- *getLocalPort()* : ports local ;
- *getInetAddress()* : adresse locale ;

Autres méthodes :

- *accept()* : attend une demande de connexion, retourne une instance de Socket ;
- *close()* : fermeture de la socket ;
- *setSoTimeout(int)* : fixe le delai d'attente (si 0 accept bloquant) ;
- accès aux options TCP : timeOut, taille des buffers,.. ;
- etc.

`http ://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html`

Ecriture d'un processus serveur

Principe général des étapes de mise en oeuvre :

1. Création d'une socket passive.
   new ServerSocket(...)
2. Attente d'une connexion, obtension d'une socket active.
   accept()
3. Accès aux flux d'entrée et de sortie de la socket.
   getInputStream(), *getOutputStream()*
   (Encapsulation suivant les traitement à réaliser.)
4. Dialogue réseau via la les flux et réalisation des traitements.
   ...
5. (Mise en place d'une boucle de retour à l'étape 2.)

Exemple de serveur Java

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

class TCPServer {
  public static void main(String argv[]) throws Exception {
    String request;

    \\ Création de la socket d'accueil au port 8080
    ServerSocket welcomeSocket = new ServerSocket(8080);

    while(true) {
      \\ Attente d'une demande de connexion sur la socket d'accueil
      Socket connectionSocket = welcomeSocket.accept();

      \\ Création du flux  en entrée attaché à la socket
      BufferedReader inFromClient = new BufferedReader(
                                    new InputStreamReader(
                                      connectionSocket.getInputStream()));

      \\ Création du flux  en sortie attaché à la socket
      PrintWriter outToClient = new PrintWriter(
                                    new BufferedWriter(
                                      new OutputStreamWriter(
                                        connectionSocket.getOutputStream())),true);

      \\ Lecture des données arrivant du client
      request = inFromClient.readLine();

      \\ Emission des données au client
      outToClient.println(request);

      connectionSocket.close();
    } \\ boucle et attend la connexion d'un nouveau client
  }
}
```

Un peu plus de souplesse ?

La méthode accept() est bloquante par défaut (pas nécessairement un inconvénient).

Comment gagner en flexibilité ?

Implantation d' un schéma dispatcheur :

- un thread dispatcheur écoute sur un port (et ne fait que ça) ;
- à chaque nouvelle connexion, le dialogue est délégué à un autre thread ;

Deux solutions pour *accept()* asynchrone :

- timeout
- *ServerSocket.getChannel()* fournit une instance de *ServerSocketChannel* qui implante une méthode *accept()* non bloquante.

Des questions ?