Programmation réseau en Java : les sockets TCP

Michel Chilowicz

Master 2 TTT
Université Paris-Est Marne-la-Vallée

Version du 21/02/2013

Plan

TCP (Transport Control Protocol)

Contrairement à UDP, TCP (RFC 793) présente les caractéristiques suivantes :

- Transmission unicast bidirectionnelle en mode connecté (flux)
- Mécanisme de retransmission des paquets non acquittés par le destinataire

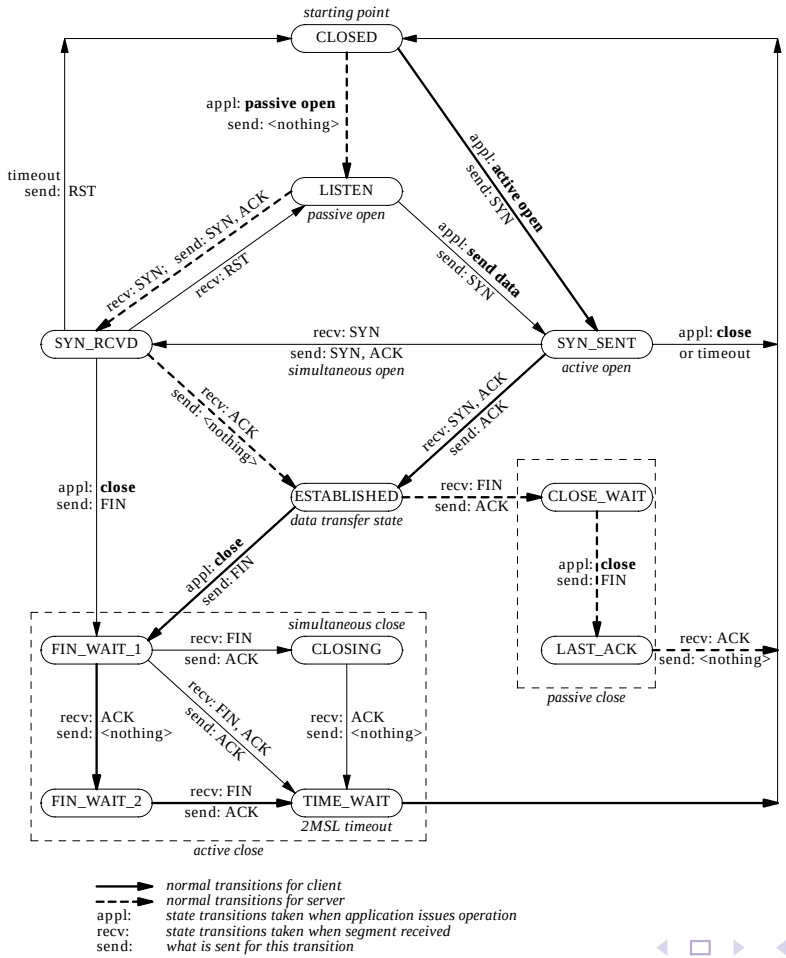
Conséquences :

- Implantation plus lourde avec gestion d'états des correspondants
- Peu adapté au temps réel (décalage de réception dû à des paquets perdus)
- Peu performant sur des réseaux radio (réduction de débit en cas de problèmes transitoires)

A mi-chemin entre UDP et TCP : SCTP (Stream Control Transmission Protocol)

- Transmission de messages (comme UDP) mais avec conservation d'ordre
- Fiabilité de l'acheminement avec gestion de congestion (comme TCP)
- Gestion de multi-flux
- Possibilités de multi-diffusion (comme UDP)
- Implanté en standard avec noyaux Linux et BSD récents
- Implantation Java intégrée depuis le JDK 1.7

Fonctionnement de TCP : les états (TCP/IP Illustrated, Stevens & Wright)



Fonctionnement de TCP : l'envoi et l'acquittement de paquets

- Une fenêtre d'envoi de n segments est utilisée : le correspondant doit envoyer au moins un acquittement tous les n segments reçus
- Acquittement envoyé pour le segment i : tous les segments jusqu'à i ont été reçus
- Compte-à-rebours pour chaque segment envoyé : à l'expiration, le paquet est renvoyé
- Si trop de paquets perdus : congestion probable et réduction de la fenêtre d'envoi
- Problème : un paquet perdu n'est pas obligatoirement lié à la congestion de liens (réseaux sans fils)

Format de segment TCP

- Port source (16 bits)
- Port destination (16 bits)
- Numéro de séquence (32 bits)
- Numéro d'acquittement (32 bits) du dernier segment reçu
- Taille de l'en-tête en mots de 32 bits (32 bits)
- Drapeaux divers (28 bits) : ECN (congestion), URG (données urgentes), ACK (accusé de réception), PSH (push), RST (reset), SYN (établissement de connexion), FIN (terminaison de la connexion)
- Taille de la fenêtre (16 bits) souhaitée en octets (au-delà un acquittement doit être envoyé)
- Somme de contrôle (16 bits), sur la fin de l'en-tête IP, l'en-tête TCP et les données
- Pointeur de données urgentes (16 bits)
- Options (complétées avec un bourrage pour une longueur d'en-tête multiple de 32 bits)

Initialisation d'une socket client

En Java, socket TCP représentée par une instance de `java.net.Socket`

Constructeurs et méthodes peuvent lever une `IOException`

1. On souhaite écouter sur l'adresse joker et laisser le système choisir un port libre d'attache :
   - `Socket(InetAddress addr, int port)` : se connecte sur la socket distante *addr:port*
   - `Socket(String host, int port)` : équivaut à `Socket(InetAddress.getByName(host), port)`
2. On souhaite spécifier l'adresse de l'interface et le port local d'attache :
   - `Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)`
3. On souhaite ne pas connecter la socket pour l'instant :
   1. `Socket()`
   2. ensuite, il faut attacher la socket localement avec `bind(SocketAddress sa)`,
   3. et puis se connecter (avec un temps limite si timeout $\neq 0$) à la socket distante avec `connect(SocketAddress remote, int timeoutInMillis)`

Flots de Socket

Pour communiquer entres sockets TCP (locale et distante), on utilise des flots binaires.

- `InputStream getInputStream()` : flot où lire des données binaires venant du correspondant
- `OutputStream getOutputStream()` : flot où écrire des données à destination du correspondant

Quelques remarques (et rappels) :

- Les opérations d'E/S sur ces flots sont bloquantes.
- Les `IOException` doivent être gérées.
- Les écritures ne provoquent pas nécessairement l'envoi immédiat de paquet TCP (bufferisation possible)
  - Appeler `flush()` permet de demander l'envoi immédiat (utiliser *TCP No delay*))
- Fermer un flot ferme la socket sous-jacente

Configuration de Socket

- `void setKeepAlive(boolean keepAlive)` : envoie régulièrement des paquets pour maintenir en vie la connexion
- `void setOOBInline(boolean oobinline)` : intègre les données urgentes Out-Of-Band dans le flux entrant (sinon elles sont ignorées)
  - `void sendUrgentData(int data)` : permet d'envoyer un octet urgent (8 bits de poids faible)
- `void setReceiveBufferSize(int size)` : fixe la taille du buffer en réception
- `void setSendBufferSize(int size)` : fixe la taille du buffer en émission
- `void setSoLinger(boolean on, int lingerTimeInSeconds)` : configure une fermeture bloquant jusqu'à confirmation de réception des dernières données envoyées ou timeout atteint
- `void setSoTimeout(int timeoutInMillis)` : configure un délai limite pour les lectures (`SocketTimeoutException` peut être levée)
- `void setTcpNoDelay(boolean on)` : permet de désactiver l'algorithme de Nagle (utile pour les protocoles interactifs : envoi immédiat de petits segments)
- `void setTrafficClass(int tc)` : configure l'octet Type-Of-Service pour les paquets (utile pour la QoS)
- `void setReuseAddress(boolean on)` : permet de réutiliser une adresse de socket locale récemment fermée (état TIME_WAIT) ; doit être appelé avant `bind`.

Remarque : existence de getters relatifs à ces méthodes (pour obtenir la configuration courante)

Fermeture de Socket

- `shutdownInput()` : permet d'ignorer toutes les données entrant ultérieurement sur la socket $\longrightarrow$ la socket distante n'est pas informée et peut continuer d'envoyer des données
- `shutdownOutput()` : ferme le flot sortant de la socket
- `close()` : permet de libérer les ressources liées à la socket

Exemple : un client TCP qui envoie un fichier

```
public static void main(String[] args)
{
        if (args.length < 2) throw new IllegalArgumentException("Not enough arguments");
        try ( Socket s = new Socket(args[0], Integer.parseInt(args[1]));
                InputStream is = new FileInputStream(args[2]);
                OutputStream os = new BufferedOutputStream(s.getOutputStream()) )
        {
                s.shutdownInput(); // We are not interested in the answer of the server
                transfer(is, os);
                os.flush();
        }
}
```

Principe de fonctionnement d'une socket serveur

1. La socket serveur écoute les paquets entrants
2. À la réception d'un paquet SYN :
   - retour d'un paquet SYN/ACK à la socket client pour continuer le *3-way handshake*
   - retour d'un paquet RST pour refuser la connexion (si trop de connexions pendantes par exemple)
3. Le socket client confirme l'ouverture par un paquet ACK à la socket serveur
4. La socket serveur créé une socket de service pour communiquer avec la socket client et l'ajoute dans une file d'attente pour être récupérée par le programme serveur (par *accept*)

java.net.ServerSocket

Construction

- `ServerSocket(int port, int backlog, InetAddress addr)` : créé une socket serveur TCP écoutant sur l'IP locale spécifiée, le port indiqué avec le backlog (nombre maximal de connexions en file d'attente) donné (si addr non indiqué utilisation de l'adresse joker, si backlog non indiqué utilisation d'une valeur par défaut)
- `ServerSocket()` : créé une socket serveur localement non attachée
  - appel ultérieur de `bind(SocketAddress sAddr, int backlog)` nécessaire

Acceptation de connexion

`Socket accept()` retourne la plus ancienne socket de service en attente.
Cette méthode est bloquante.
Timeout instaurable avec `setSoTimeout(int timeoutInMillis)`.

Modèle itératif d'utilisation de `ServerSocket`

1. On construit une `ServerSocket`
2. On récupère une socket connectée avec le prochain client en attente avec `accept()`
3. On dialogue en utilisant la socket connectée au client
4. La communication terminée avec le client, on ferme la socket
5. On retourne à l'étape 2 pour récupérer le client suivant

Limitation du modèle : 1 seul client traité à la fois (sinon plusieurs threads nécessaires)

Utilisation d'une socket de service

- Il s'agit d'une instance de `java.net.Socket` : même utilisation que côté client
  - `getInputStream()` pour obtenir un flot pour recevoir des données du client
  - `getOutputStream()` pour obtenir un flot pour envoyer des données au client
  - Utilisation possible des setters de configuration
- Comment connaître les coordonnées du client ?
  - `getInetAddress()` pour obtenir l'adresse IP
  - `getPort()` pour obtenir le port

Exemple : un serveur TCP qui réceptionne des fichiers

```
public class FileReceiverServer
{
        // After one minute with no new-connection, the server stops
        public static final int ACCEPT_TIMEOUT = 60000;

        private final File directory;
        private final int port;

        public FileReceiverServer(File directory, int port)
        {
                this.directory = directory;
                this.port = port;
        }

        public void launch() throws IOException
        {
                try (ServerSocket ss = new ServerSocket(port))
                {
                        ss.setSoTimeout(ACCEPT_TIMEOUT);
                        while (true)
                        {
                                try (Socket s = ss.accept();
                                        OutputStream os = new BufferedOutputStream(new FileOutputStream(
                                                new File(directory, s.getAddress() + "_" + s.getPort())) )
                                {
                                        // Save the data in a file named with the address and port of the client
                                        transfer(s.getInputStream(), os);
                                } catch (SocketTimeoutException e)
                                {
                                        System.err.println("The server will shutdown because no new connections are available");
                                        break;
                                } catch (IOException e)
                                {
                                        e.printStackTrace();
                                        continue; // Treat the next incoming client
                                }
                        }
                }
        }

        public static void main(String[] args)
        {
                new FileReceiverServer(
                        new File(args[0]), Integer.parseInt(args[1])).launch();
        }
}
```