Programmation réseau en Java : sockets UDP

Michel Chilowicz

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

Version du 7/02/2013

Plan

1. Généralités sur IP et ses protocoles de transport
   - Les paquets IP
   - Protocole UDP
2. Les sockets UDP en Java
   - DatagramSocket
   - MulticastSocket

Datagramme IPv4

Bits	Longueur	Champ
0-3	4 bits	Version de protocole IP (4 pour IPv4)
4-7	4 bits	Longueur de l'en-tête (en mots de 32 bits)
8-15	8 bits	Type de service : priorité (bit 0-2), drapeaux de délai (3), de débit (4) et de fiabilité (5)
16-31	16 bits	Longueur totale en octets (en-tête compris) bornée par 65535 octets
32-47	16 bits	Identificateur de paquet (même identificateur pour des sous-paquets fragmentés)
48-50	3 bits	Drapeaux de fragmentation : Don't Fragment (bit 1), More Fragments (bit 2)
51-63	13 bits	Décalage du sous-paquet fragmenté
64-71	8 bits	Durée de vie (TTL) du paquet
72-79	8 bits	Identificateur de protocole (6=TCP, 17=UDP, 1=ICMP)
80-95	16 bits	Somme de contrôle de l'en-tête
96-127	32 bits	Adresse source IPv4
128-163	32 bits	Adresse destination IPv4
164-...	variable	Champs optionnels

Datagramme IPv6

En-tête obligatoire

Bits	Longueur	Champ
0-3	4 bits	Version (6 pour IPv6) : permet d'orienter le paquet vers la pile IPv4 ou IPv6
4-11	8 bits	Classe de trafic : priorité du paquet
12-31	20 bits	Étiquette de flux : pour des applications futures de QoS
32-47	16 bits	Longueur du contenu (limité à 65535 octets, valeur 0 pour des *jumbograms*)
48-55	8 bits	Prochain en-tête : type de protocole encapsulé
56-63	8 bits	Nombre maximal de sauts (décrémenté à chaque passage d'un routeur)
64-191	128 bits	Adresse IPv6 source
192-319	128 bits	Adresse IPv6 destination

En-têtes d'extension chaînés

- En-tête *Hop-by-Hop* : champs libres destinés aux routeurs
- En-tête options de destination : champs libres lus par le destinataire
- En-tête de routage (pour spécifier une route)
- En-tête de fragmentation (uniquement autorisée par la source)
- En-tête d'authentification
- En-tête de chiffrement
- En-tête vide (pour indiquer la fin de liste des en-têtes)

Principales différences entre les datagrammes IPv4 et IPv6

- Somme de contrôle du paquet IP obsolète : généralement déjà implantée par les trames de couches basses (Ethernet) et hautes (UDP et TCP)
- Liste chaînée d'en-têtes optionnels : évite la spécification de données inutiles (informations de fragmentation)

La fragmentation

- Les paquets IPv4 et IPv6 peuvent être transportés par des réseaux physiques imposant une taille maximale de paquet
- Par exemple pour Ethernet : paquets de 1500 octets
- Pour respecter la contrainte de taille de paquet (MTU : Maximum Transfert Unit), deux solutions :
  1. Choisir comme taille de paquet le MTU minimum de tous les réseaux physiques traversés (*Path MTU discovery* : RFC 1191)
  2. Laisser les réseaux physiques fragmenter les paquets (impossible en IPv6)

Routage IP

- Connexion de routeurs interconnectant des réseaux différents communiquant par le protocole IP
- Routage de proche en proche en fonction de l'@IP de destination sans connaissance de la topologie globale
- Routage indépendant de chaque paquet en mode *best effort*
- Utilisation de tables de routage sur un routeur (association de réseau avec interface de sortie et adresse du prochain routeur)
  - Mise à jour manuelle (`route` sous Unix)
  - Mise à jour automatique avec protocoles de routage
    - ★ Protocoles IGP (Interior Gateway Protocol) : RIP, OSPF, EIGRP...
    - ★ Protocole BGP (Border Gateway Protocol)

Machines et ports

- Chaque machine sur le réseau dispose de $2^{16} = 65536$ ports différents
- Chaque port permet de différencier un flux de données applicatif spécifique
- Communication client/serveur : lorsqu'un client initie une communication, il ouvre un port aléatoire source sur lequel le serveur lui répond
- Fichier listant les ports usuels (sous Unix) : `/etc/services`

Quelques port spécifiques

- Port 0 : port réservé, utilisé pour demander l'allocation d'un port libre lors de l'initialisation d'une socket
- Ports < 1024 : ports privilégiés (généralement attachables que par un super-utilisateur)
  - Port TCP/22 : serveur SSH (Secure Shell)
  - Port TCP/25 : serveur mail SMTP
  - Port TCP/80 : serveur HTTP
  - Port TCP/443 : serveur HTTP sécurisé
  - Port UDP/5060 : serveur supportant Session Initiation Protocol (pour téléphonie)
  - ...

Rien n'interdit de ne pas utiliser les ports conventionnellement définis : il faut alors les expliciter

Présentation de UDP (User Datagram Protocol)

- Permet d'envoyer des paquets de données IP indépendants
- La source et la destination d'un paquet sont définis par une adresse IP et un numéro de port (socket)
- Protocole d'implantation simple (latence faible) mais aucune garantie sur la bonne réception du paquet (ainsi que sur le délai de livraison) : pas de contrôle de flux et de congestion
- Vulnérable à l'usurpation d'@IP source
- Supporte le multicast et le broadcast contrairement à TCP
- Utilisé pour les applications nécessitant une latence faible avec une tolérance aux pertes de paquets :
  - Envoi de petits messages informatifs sur un réseau local
  - Streaming audiovisuel
  - Voix sur IP
  - Jeux en réseau
  - ...

Format du datagramme UDP

- Port source sur 16 bits
- Port destination sur 16 bits
- Longueur sur 16 bits (soit 65535 octets au maximum)
- Somme de contrôle sur 16 bits (calculée sur l'en-tête comprenant la fin de l'en-tête IP et les données)
- Champ de données

Sockets

- Les piles IP implantent un mécanisme de socket (association d'adresse IP et de port de communication ouvert sur une machine)
- Réalisation d'une connexion en envoyant un paquet d'une socket vers une autre

Sur les systèmes POSIX

- `int socket(int domain, int type, int protocol)` pour ouvrir une socket
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)` pour lier une socket à une adresse et un port
- `int send(int socket, const void *buf, size_t len, int flags)` pour envoyer un datagramme
- `ssize_t recv(int s, void *buf, size_t len, int flags)` pour extraire un datagramme de la file d'attente de réception
- `man 7 socket, ...` pour en savoir plus

Interface Java pour les sockets IP

Regroupée dans les paquetages :

- java.net.* : sockets pour les communications utilisant UDP et TCP en mode bloquant
- java.nio.* : nouveau pan de l'API pour les communications non bloquantes et asynchrones

Pas d'API pour gérer des messages ICMP
(`InetAddress.isReachable(int timeout)` peut cependant utiliser un ping ICMP)

Les interfaces réseau

- Pour chaque interface réseau d'une machine : au moins une @IP
- `java.net.NetworkInterface` : classe représentant une interface réseau
  - Obtention d'une interface par les méthodes statiques `getByInetAddress(InetAddress addr)` et `getByName(String name)`
  - Récupération de toutes les interfaces : `Enumeration<NetworkInterface> getNetworkInterfaces()`
- Une socket doit se fixer sur une ou plusieurs interfaces réseau (pour s'intéresser uniquement aux données provenant de cette interface) :
  - Choix d'une interface réseau par son @IP (e.g. 127.0.0.1 pour lo en IPv4)
  - Choix de toutes les interfaces réseau avec l'adresse joker (adresse nulle)

Socket réseau UDP : classe DatagramSocket

Association d'une adresse et d'un port pour recevoir ou envoyer des datagrammes UDP
Constructeurs disponibles :

- `DatagramSocket()` : création d'une socket sur l'adresse joker sur un des ports disponibles
- `DatagramSocket(int port)` : permet de choisir le port d'attache, utilisation de l'adresse joker
- `DatagramSocket(int port, InetAddress addr)` : permet de s'attacher sur une adresse spécifique de la machine (n'écoute pas sur toutes les interfaces)

Ces constructeurs peuvent lever une `SocketException` si la socket ne peut être allouée.
Après utilisation, une socket doit être fermée avec `close()` pour libérer des ressources systèmes.

Datagramme UDP : classe DatagramPacket

Cette classe permet de manipuler des datagrammes UDP.

- Ils peuvent être créés par des constructeurs utilisant un tableau d'octets (`byte[]`) soit pour recevoir des données, soit pour en envoyer.
- Pour recevoir des données, le tableau d'octets communiqué doit être de taille suffisante pour recevoir les données (sinon le paquet est tronqué !).
- Une instance peut être utilisée pour envoyer plusieurs paquets ou pour en recevoir. Possibilité de modifier :
  - L'adresse de destination avec `setAddress(InetAddress)` ainsi que le port avec `setPort(int)`.
  - Le tampon de données avec `setData(byte[] buffer, [int offset, int length])`.
- Les données peuvent être consultées avec `byte[] getData()` et `int getLength()` (taille du paquet reçu).

Caractères et octets

- Les données sont échangées sur le réseau par séquence d'octets
- Pour transmettre une chaîne de caractère de A vers B :
  1. Codage de la chaîne en octets par A
  2. Transmission d'un paquet avec le tableau d'octets
  3. Réception du paquet avec tableau d'octets par B
  4. Décodage des octets en chaîne par B

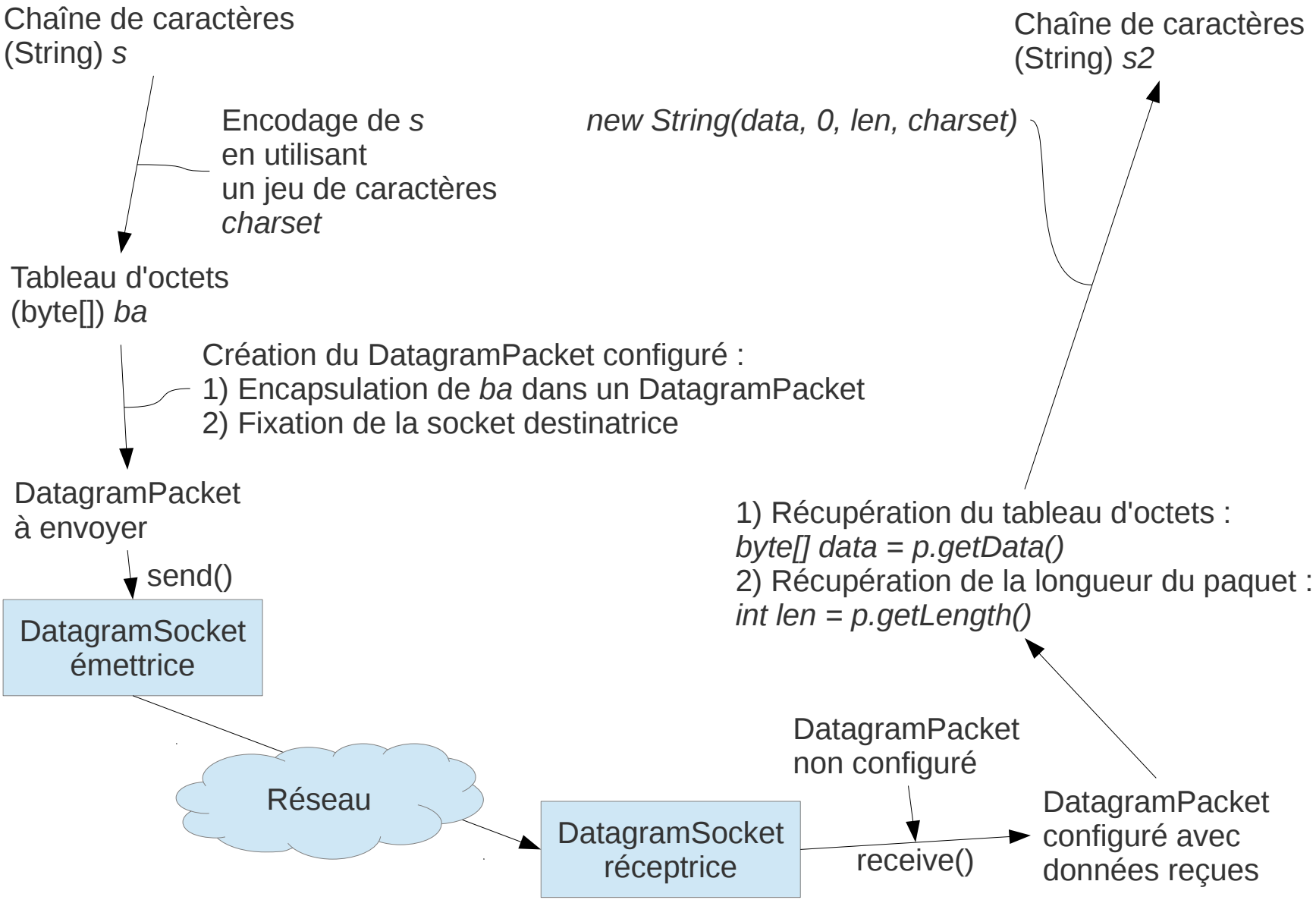
Des caractères vers les octets (et vice-versa)

- Convention de conversion caractère-octet implantée par classe *Charset*
- Quelques jeux de caractères : ASCII, iso-8859-15, UTF-8, UTF-16LE, UTF-16BE, ...
- Un caractère peut être codé par un ou plusieurs octets
- De la chaîne aux octets : `byte[] String.toBytes(String charset)`
- Des octets à la chaîne : `new String(byte[] tableauOctets, int depart, int longueur, String charset)`
- Si le jeu de caractère n'est pas indiqué : utilisation du jeu par défaut (`static Charset Charset.defaultCharset()`)
- Pour obtenir tous les jeux disponibles : `static SortedMap<String,Charset> Charset.availableCharsets()`

Envoyer et recevoir des paquets avec DatagramSocket

- Méthode `send(DatagramPacket)` pour envoyer une paquet : paquet ajouté dans la file d'attente d'émission.
- Méthode `receive(DatagramPacket)` pour recevoir un paquet en file d'attente de réception
  - Appel bloquant jusqu'à la réception d'un paquet
  - `setSoTimeout(int timeoutInMillis)` fixe un délai limite de réception de paquet (au-delà, levée d'une `java.net.SocketTimeoutException`)
- Ces méthodes peuvent lever une `java.io.IOException` qui doit être gérée.
- La taille des files d'attente est consultable et configurable avec :
  - `int get{Send,Receive}BufferSize()`
  - et `void set{Send,Receive}BufferSize()`.

Envoyer des paquets UDP contenant du texte



Exemple Java : envoyer un paquet contenant l'heure à une plage d'adresses

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

public class TimeSender
{
        private final DatagramSocket socket;
        private final DatagramPacket packet;

        public TimeSender(int port) throws SocketException
        {
                this.socket = new DatagramSocket();
                this.packet = new DatagramPacket(new byte[0], 0); // Packet with empty array
                this.packet.setPort(port);
        }

        public void sendTimePacket(InetAddress a) throws IOException
        {
                // We reuse the same datagram packet but modify its content and addressee
                byte[] timeData = new Date().toString().getBytes();
                packet.setData(timeData);
                packet.setAddress(a);
                // this.packet.setPort is useless since the port does not change
                socket.send(packet); // May throw an IOException
        }

        public static InetAddress getSuccessor(InetAddress a)
        {
                // Exercise: write the implementation of getSuccessor(InetAddress) returning the successor of an IP address.
                // Example: the successor of 10.1.2.3 is 10.1.2.4 (for last byte 0 and 255 are forbidden)
        }

        public static void main(String[] args) throws IOException
        {
                if (args.length < 3) throw new IllegalArgumentException("Not enough arguments");
                InetAddress firstA = InetAddress.getByName(args[0]); // Start address
                InetAddress lastA = InetAddress.getByName(args[1]); // Stop address
                int port = Integer.parseInt(args[2]);
                TimeSender ts = new TimeSender(port);
                for (InetAddress currentA = firstA; currentA.equals(lastA); currentA = getSuccessor(currentA))
                        ts.sendTimePacket(currentA);

        }
}
```

Mécanisme de pseudo-connexion

- Possibilité de pseudo-connecter deux sockets UDP avec `void DatagramSocket.connect(InetAddress addr, int port)` : filtrage automatique des paquets émis et reçus.
- Vérification de la connexion avec `boolean isConnected()`.
- Déconnexion avec `void disconnect()`.

Quelques tests avec Netcat

Netcat : application simple pour communiquer en UDP (et aussi en TCP)

1. On lance le serveur sur le port N en écoute en UDP sur toutes les interfaces (adresse joker) : `nc -l -u N`
2. On exécute un client se connectant sur le port N de la machine : `nc -u nomMachine N`
3. Le client envoie chaque ligne de l'entrée standard dans un datagramme UDP : le serveur la reçoit et l'affiche sur sa sortie standard
4. Envoi de datagrammes du serveur vers le client également possible

Limitation pour le serveur netcat : communication avec un seul client

Exemple Java : recevoir des datagrammes sur un port avec un délai limite

```
public class PacketReceiver
{
  public static final int BUFFER_LENGTH = 1024;

  public static void main(String[] args) throws IOException
  {
    if (args.length < 3) throw new IllegalArgumentException("Not enough arguments");
    InetAddress a = InetAddress.getByName(args[0]);
    int port = Integer.parseInt(args[1]);
    int timeout = Integer.parseInt(args[2]);
    DatagramSocket s = new DatagramSocket(port, a);
    try {
      s.setSoTimeout(timeout);
      DatagramPacket p = new DatagramPacket(new byte[BUFFER_LENGTH], BUFFER_LENGTH);
      for (boolean go = true; go; )
      {
        try {
          s.receive(p);
        } catch (SocketTimeoutException e)
        {
          System.err.println("Sorry, no packet received before the timeout of " + timeout + " ms");
          go = false;
        }
        if (go)
        {
          System.out.println("Has received data from " + p.getAddress() + ":" + p.getPort());
          System.out.println(new String(p.getData(), 0, p.getLength(), "UTF-8"));
          p.setLength(BUFFER_LENGTH); // Reset the length to the size of the array
        }
      }
    } finally
    {
      s.close(); // Don't forget to liberate the socket resource
    }
  }
}
```

Multicast UDP

- Quelques rappels sur les adresses de multicast :
  - 224.0.0.0/4 en IPv4 (préfixe 1110)
  - FF00::0/8 en IPv6 (RFC 4291)
  - Certaines adresses à usage prédéfinis (groupe des hôtes, routeurs, uPnP...)

Principe

- Les hôtes s'abonnent/se désabonnent à une adresse de multicast A.
- Une trame émise vers A est adressée à tous les membres du groupe.

Avantages

- En théorie, mutualise les paquets échangés sur des segments de réseau contrairement à l'unicast.
- Évite d'envoyer des paquets à tous les hôtes comme en broadcast (broadcast incompatible avec un réseau étendu).

Protocoles de multicast

- IGMP (Internet Group Management Protocol) permet de signaler à la passerelle du réseau local les membres de chaque groupe. Certains commutateurs de couche 2 l'utilisent.
- Sur Internet, un arbre de distribution multicast est construit avec PIM (Protocol Independent Multicast) en *Sparse Mode*, *Dense Mode*, mixte *Sparse-Dense* ou *Source Specific Mode*.

java.net.MulticastSocket

- Version améliorée de `DatagramSocket` avec support du multicast
- Envoi classique d'un datagramme avec `send(DatagramPacket)` (à une IP unicast ou multicast)
- Réception de datagrammes en multicast requérant un abonnement préalable au groupe de multicast
  - `void joinGroup(InetAddress ia)` pour joindre un groupe
  - `void leaveGroup(InetAddress ia)` pour quitter un groupe
- Quelques paramètres utiles pour le multicast
  - `void setInterface(InetAddress ia)` ou `void setNetworkInterface(NetworkInterface netIf)` pour fixer l'interface utilisée pour la communication multicast
  - `void setTimeToLive(int ttl)` pour indiquer la portée des paquets multicast envoyés