Mini-projet Sécurité et réseaux

L3 Informatique, 2017-2018

0.1 Rappels

Chaque application communiquant sur un réseau est identifiée par le triplet:

- @IP de l'hôte
- Le protocole de transport à utiliser
- Le numéro de port sur lequel l'application s'attend à recevoir les communications

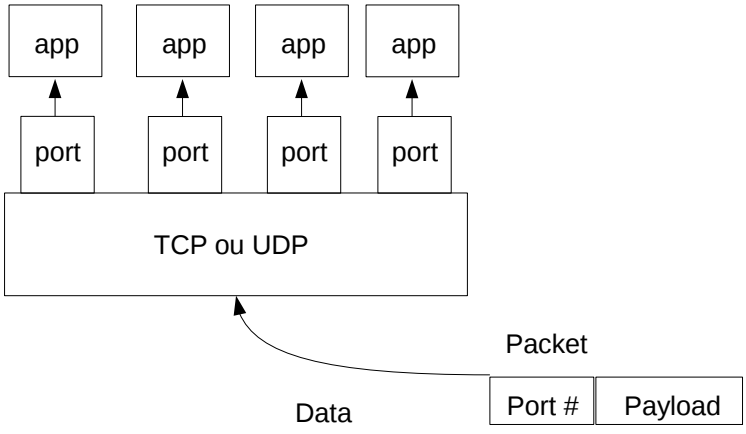

Figure 1

0.2 Les sockets de communication

Les sockets vont servir d'interface entre les applications et la couche transport

- Elles permettent de masquer l'interface et les mécanismes de transport des données

Une socket est un moyen simple de désigner l'émetteur ou le destinataire d'une communication en l'associant à un port

- La connexion à travers des sockets va être bidirectionnelle
- Elles vont désigner une communication entre deux processus communicants

Une fois la connexion établie, les processus client et serveur vont pouvoir communiquer en utilisant des primitives READ et WRITE

- Similaire aux primitives utilisées pour écrire ou lire un fichier
- Formellement une socket va désigner une communication entre deux processus. Elle est caractérisée par

  Le protocole de transport utilisé

  L'adresse de la machine A

  Le numéro de port de la machine A

  L'adresse de la machine B

  Le numéro de port de la machine B

0.3 Les sockets en JAVA

Classes et Interfaces du paquetage java.net

- Adresses IP

  InetAddress

- TCP

  Socket

  ServerSocket

- UDP

  DatagramSocket

  DatagramPacket

- Multicast

  MulticastSocket

  DatagramPacket

0.4 La résolution d'adresse en JAVA

La résolution d'adresse est utile aussi bien pour TCP que pour UDP car elle permet d'identifier une machine à partir de son nom.
Ex de nom: win2k8.castres.univ-jfc.fr

```
package reseau;

import java.net.InetAddress;
import java.net.UnknownHostException;

public class ResolutionDeNom{
        public static void main (String[] args) {
                InetAddress address;
                try{
                        address = InetAddress.getByName(args[0]);
                        System.out.println(args[0] + ":" + address.getHostAddress());
                }
                catch(UnknownHostException e){
                        e.printStackTrace();
                }
        }
}
```

0.5 Les sockets avec TCP

0.5.1 Requête de connexion

Le serveur ouvre une socket en écoute sur son numéro de port local et attend. Le client connait l'adresse IP du serveur (ou son nom) ainsi que le numéro de port sur lequel celui-ci est en attente. Le client va également utiliser un numéro de port de son coté pour pouvoir s'identifier auprès du serveur. Ce numéro est en général choisit au hasard par le système.

0.5.2 Etablissement de connexion

Si tout se passe bien lors de la requête de connexion, le serveur dispose maintenant d'une nouvelle socket identifiant le client. Le serveurs dispose donc d'au moins deux sockets:

- L'une pour attendre les requêtes de connexion
- L'autre pour communiquer avec le client

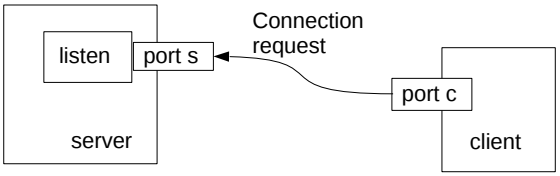

Figure 2

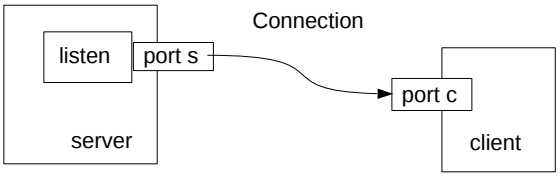

Figure 3

0.6 Un serveur TCP

Premièrement le serveur doit construire la socket sur laquelle il va attendre les connexions.

```
try{
        serverSocket = new ServerSocket(4444);
}
catch(IOException e){
        System.out.println("Could not listen on port 4444");
        System.exit(-1);
}
```

Si la création se passe bien (si le numéro de port est libre), le serveur se met en attente d'un client à l'aide de la méthode bloquante accept().

```
Socket clientSocket = null;
try{
        clientSocket = serverSocket.accept();
}
catch(IOException e){
        System.out.println("Accept failed on port 4444");
        System.exit(-1);
}
```

Lors d'une connexion, la méthode accept() permet d'obtenir la socket identifiant le client (i.e l'autre coté de la connexion).

0.7 Un client TCP

Le client TCP va effectuer sa demande de connexion lors de la création de la socket

```
try{
        echotSocket = new Socket(nom machine distante, numero port distant);
```

```
        out = new PrintWriter(echoSocket.getOutputStream(), true);
        int = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
}
catch(UnknownHostException e){
        System.out.println("Destination unknown");
        System.exit(-1);
}
catch(IOException e){
        System.out.println("now to investigate this IO issue");
        System.exit(-1);
}
```

A partir de la socket il est encore nécessaire de construire deux flux d'entrées/sorties Ces flux seront ensuite utilisés pour pouvoir communiquer avec le serveur.

0.8 Les entrées/sorties et les sockets

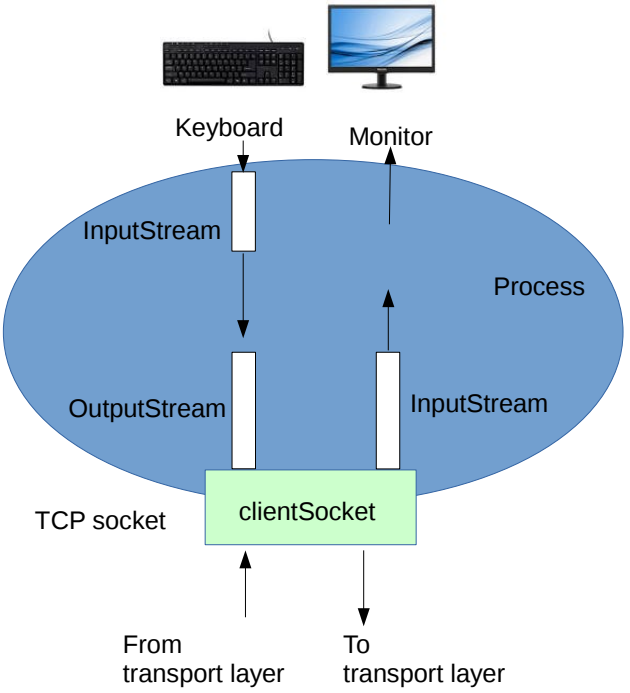

Figure 4

Deux méthodes sont fournies dans la classe Socket pour obtenir ces flux

- getOutputStream(): Le flux de sortie va permettre d'envoyer des informations vers le processus distant (ici le serveur)
- getInputStream(): Le flux d'entrée va permettre de récupérer des informations depuis le processus distant (ici le serveur)

Une fois les flux obtenus, il est possible comme pour les fichiers de les combiner avec des *Reader et de *Writer pour formatter les échanges de données.
Ex: BufferedReader / BufferedWriter

```
        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("taranis", 7);
            out = new PrintWriter(echoSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(
                                        echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: taranis.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                               + "the connection to: taranis.");
            System.exit(1);
        }

        BufferedReader stdIn = new BufferedReader(
                                    new InputStreamReader(System.in));
        String userInput;

        while ((userInput = stdIn.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
        }

        out.close();
        in.close();
        stdIn.close();
        echoSocket.close();
    }
}
```

Figure 5

0.9 Les datagrammes en UDP

Un datagramme est un message d'information indépendant émis sur le réseau dont l'acheminement n'est pas garanti.

Le langage Java fournit trois classes pour manipuler les datagrammes: DatagramSocket, DatagramPacket et MulticastSocket.

Un programme basé sur UDP écrira donc des DatagramPaquet dans des DatagramSocket pour faire une communication point à point et dans des MulticastSocket pour faire des communications multi-points.

0.10 Les communications UDP

Création d'une DatagramSocket

```
socket = new DatagramSocket(4445);
```

Création d'un datagramme pour stocker l'information reçue Le datagramme est vide et sera rempli par la méthode receive()

```
byte[] buf = new byte[256];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

Création d'un datagramme pour émettre de l'information L'envoi s'effectue avec la méthode send()

```
InetAddress addr = packet.getAddress();
int port = packet.getPort();
packet = new DatagramPacket(buf, buf.length, addr, port);
socket.send(packet);
```

0.11 Les communications multicast

- Multicast: groupe de diffusion.

  Un émetteur, plusieurs récepteurs

  Tous les récepteurs appartiennent au même groupe multicast

  L'émetteur n'en fait pas obligatoirement partie

- TTL Time to Live: Intervalle dans lequel les récepteurs peuvent recevoir des paquets. Le TTL est décrémenté en fonction du nombre de routeurs traversés Par défaut en JAVA le TTL vaut 1 ce qui veut dire qu'il ne peut traverser qu'un routeur

- Attention à ne pas confondre multicast et broadcast: Le broadcast est un envoi de 1 vers tous le monde, il n'y a pas de sélection possible des récepteurs contrairement au multicast.

0.11.1 Récepteur Multicast

Utilisation de la classe MulticastSocket qui propose notamment la méthode join(InetAddress)

```
MulticastSocket socket = new MulticastSocket(4446);
InetAddress group = InetAddress.getByName("230.0.0.1");
socket.joinGroup(group);

DatagramPacket packet;
for (int i = 0; i < 5; i++) {
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);

    String received = new String(packet.getData());
    System.out.println("Quote of the Moment: " + received);
}
socket.leaveGroup(group);
socket.close();
```

Figure 6

0.11.2 Emetteur Multicast

```
public void run() {
    while (moreQuotes) {
        try {
            byte[] buf new byte[256];
            // don't wait for request...just send a quote

            String dString = null;
            if (in == null)
                dString = new Date().toString();
            else
                dString = getNextQuote();
            buf = dString.getBytes();

            InetAddress group = InetAddress.getByName(
                                            "230.0.0.1");
            DatagramPacket packet;
            packet = new DatagramPacket(buf, buf.length,
                                        group, 4446);
            socket.send(packet);

            try {
                sleep((long)Math.random() * FIVE_SECONDS);
            } catch (InterruptedException e) { }
        } catch (IOException e) {
            e.printStackTrace();
            moreQuotes = false;
        }
    }
    socket.close();
}
```

Figure 7

0.12 Java et encryption

Java supporte, en plus des communications, diverses opération d'encryption. Le mini-programme suivant en est un exemple:

```
package testtp;

import javax.crypto.*;
import java.security.*;

public class Test {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                byte[] data ;
                byte[] result;
                byte[] original;

                try {
                    KeyGenerator kg = KeyGenerator.getInstance("DES");
                    Key key = kg.generateKey();
                    Cipher cipher = Cipher.getInstance("DES");

                    cipher.init(Cipher.ENCRYPT_MODE, key);
                    data = "Hello World!".getBytes();
                    result = cipher.doFinal(data);
                    cipher.init(Cipher.DECRYPT_MODE, key);
                    original = cipher.doFinal(result);
                    System.out.println("Decrypted data: " + new String(original));
                }
```

```
                catch (NoSuchAlgorithmException e) {
                    e.printStackTrace();
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }
}
```

Activités

Vous allez réaliser en binôme une application client-serveur dont l'objectif est le chat sécurisé.

— Créez dans un premier temps un client et un serveur capables de s'échanger des chaînes de caratères. Vous utiliserez pour ceci les sockets TCP.

— Modifiez votre application de manière à ce que la réception de la chaîne "bye" mette fin à la connexion.

— Créez un module capable de générer une clé AES, de l'exporter dans un fichier et de charger une clé existante depuis un fichier

— Intégrez le code responsable du chargement de la clé depuis un fichier au processus de démarrage du client et du serveur.

— Utilisez cette clé partagée pour chiffrer les messages envoyés et les déchiffrer à la réception. Affichez, pour chaque message, la version chiffrée et la version en clair.

— Comment pourriez-vous améliorer la sécurité de ces échanges ?

Pistes

— Commentez votre code et justifiez vos choix (numéro de port, algorithmes, type de crytpographie...)

— Votre code pouvoir s'exécuter sur n'importe quelle machine sans demander de créer une arborescence de fichiers particulière ou des IP spécifiques

— Votre rendu consistera en votre code source et un bref rapport décrivant les machines d'état de vos outils, les ports choisis etc...

— Les classes à voir : KeyGenerator, SecretKey, Cipher...