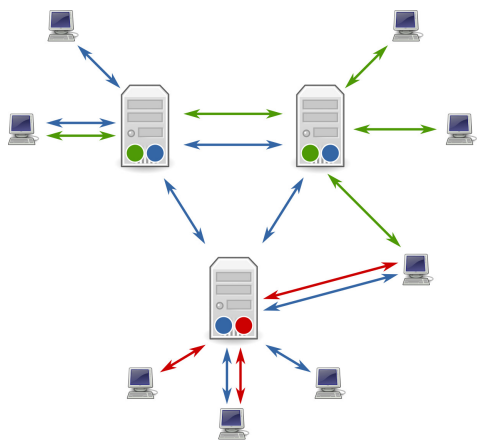
Programmation Répartie : Programmation client/serveur Socket

Problématique

- Créer une application **répartie** en utilisant les protocoles de transport
- Application client-serveur :
  - un **serveur** fournit un service
  - un **client** utilise ce service
- Exemples : Serveur pop, imap, ftp, www, ssh, jeux en ligne, chat ...



- Les communications entre les deux machines doivent être sûres, les données ne doivent pas se perdre, et arriver dans le même ordre d'émission.
- **TCP** fournit ce canal de communication

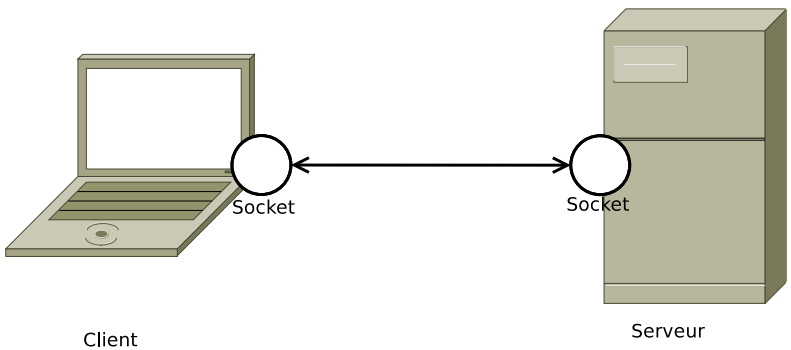
Pas de manipulation directe du protocole TCP

- Le client et le serveur doivent établir une connexion entre eux, ils utilisent des **sockets**
- Chaque programme client ou serveur, va lire et écrire dans un socket

Socket

Mécanisme de communication permettant d'utiliser une interface de transport

- Apparu dans les systèmes UNIX dans les années 80, devenu maintenant un standard
- Présent dans de nombreux langages (C, C++,python. . .)



Comment communiquer via un socket ?

- Comment établir la **connexion initiale** entre le client et le serveur ?
- Comment **envoyer** des messages ?
- Comment **recevoir** des messages ?

Client

Etablir un socket

- **identité du serveur** : adresse IP, ou nom de la machine
- numero **port** TCP

Creation socket

Socket socket = **new** Socket(identiteServeur, numPort)

Pour tester son programme, le serveur peut tourner sur la même machine que le client, donc l'identité du serveur est 127.0.0.1

Lire des données depuis un Socket

Le socket fournit un flux d'entrée avec la méthode socket.getInputStream()

Utilisation des classes de streams I/O classiques pour faciliter la lecture

Lecture de chaînes de caractères

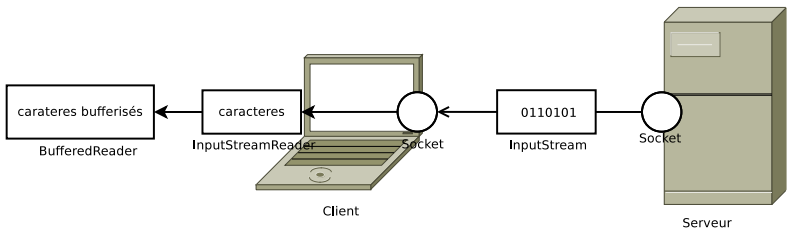
```
InputStreamReader stream =
new InputStreamReader(socket.getInputStream());
```

- getInputStream() : gère le flux d'entrée du socket
- InputStreamReader : convertit un flux binaire en flux de caractères

Lecture de chaînes de caractères (suite)

```
BufferedReader reader = new BufferedReader(stream);
String message = reader.readLine();
```

- BufferedReader : permet de gérer efficacement la lecture de caractères, de lignes . . .



Fermeture du socket

```
socket.close();
```

- Plus de communication à travers un socket après fermeture
- Pas de possibilité de le réouvrir
- Ferme les InputStream et OutputStream associés

Recapitulatif Lecture

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

public class Client {

  public static void main(String[] args) {
   try {
     Socket socket = new Socket("127.0.0.1",4444);
     InputStreamReader stream =
          new InputStreamReader(socket.getInputStream());
     BufferedReader reader = new BufferedReader(stream);
     String message = reader.readLine();
     socket.close();
   }catch (Exception e) {
        e.printStackTrace();
   }
  }

}
```

Ecrire sur le socket

Le socket fournit un flux de sortie avec la méthode socket.getOutputStream()

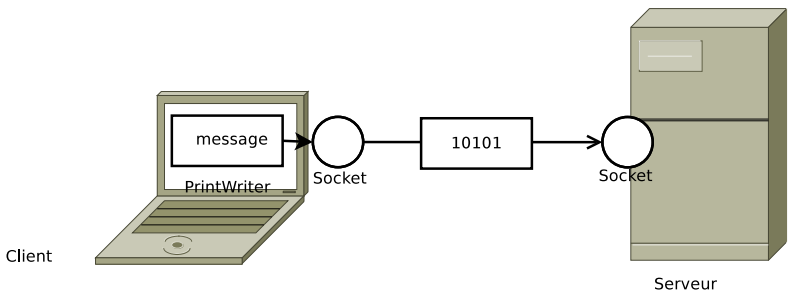
Initialiser le flux d'écriture

```
PrintWriter writer =
            new PrintWriter(socket.getOutputStream());
```

- getOutputStream() : gère les flux sortant du socket
- PrintWriter : Conversion entre flux de caractères et flux binaire

Ecrire

```
writer.print("message");
writer.println("message");
```



Recapitulatif

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

public class ClientEcriture {

  public static void main(String[] args) {
   try {
     Socket socket = new Socket("127.0.0.1",4444);
     PrintWriter writer =
          new PrintWriter(socket.getOutputStream());
     writer.println("Hellow world!");
     writer.flush();

     socket.close();
   }catch (Exception e) {
        e.printStackTrace();
   }
  }

}
```

Flux

Autres flux

- **BufferedReader, PrintWriter** permettent d'échanger des chaînes de caractères, mais ce ne sont pas les seules classes permettant d'utiliser les flux avec les sockets.
- **DataInputStream, DataOutputStream** permettent de manipuler les types primitifs de Java
- ...

Serveur

Deux sockets :

- ServerSocket : écouter et accepter les connexions
- Socket : pour communiquer avec le client

Creation d'un ServerSocket sur un port specifique

```
ServerSocket serverSock = new ServerSocket(numeroPort);
```

- Si le numero de port est 0, le port est alloué automatiquement. getLocalPort permet de retrouver ce port.
- La file d'attente pour des demandes de connexions est à 50. Si une demande arrive et que la file est pleine, la connexion est refusée.
- On peut définir ce nombre maximum, en le fournissant en paramètre au constructeur.

Serveur (suite)

Le serveur attend un demande de connexion d'un client

Creation du socket

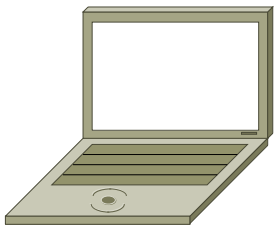
```
Socket socket = serverSock.accept();
```

- La méthode accept est bloquante (possibilité de fixer un délai maximum)
- Une fois la demande acceptée, un socket est créé
- Le socket se manipule ensuite comme dans la partie client

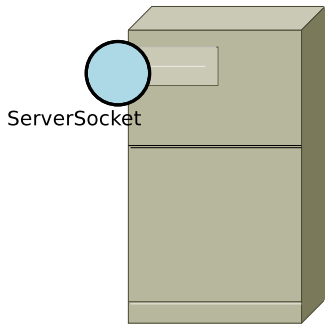
Recapitulatif

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Serveur {

 public static void main(String[] args) {
    try {
       ServerSocket socketServer = new ServerSocket(4444);
       Socket socketClient = socketServer.accept();
       System.out.println("connexion d'un client");
       socketClient.close();
       socketServer.close();
    }catch (IOException e) {
          e.printStackTrace();
    }
 }

}
```

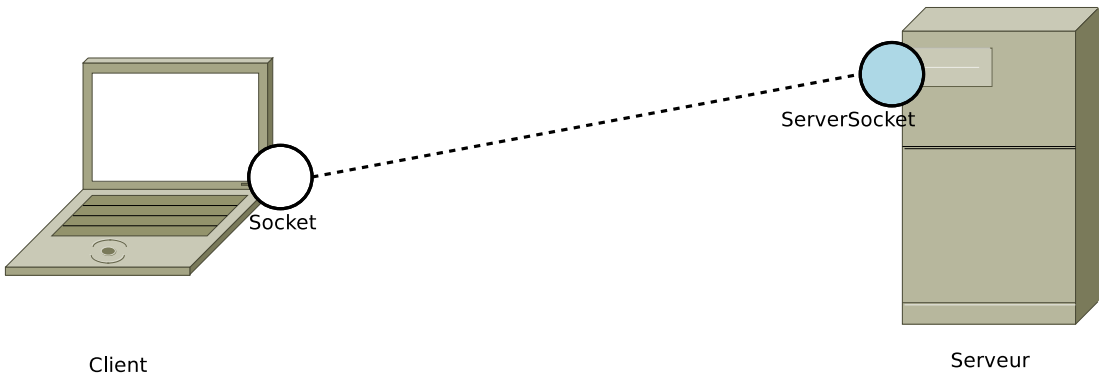
Serveur

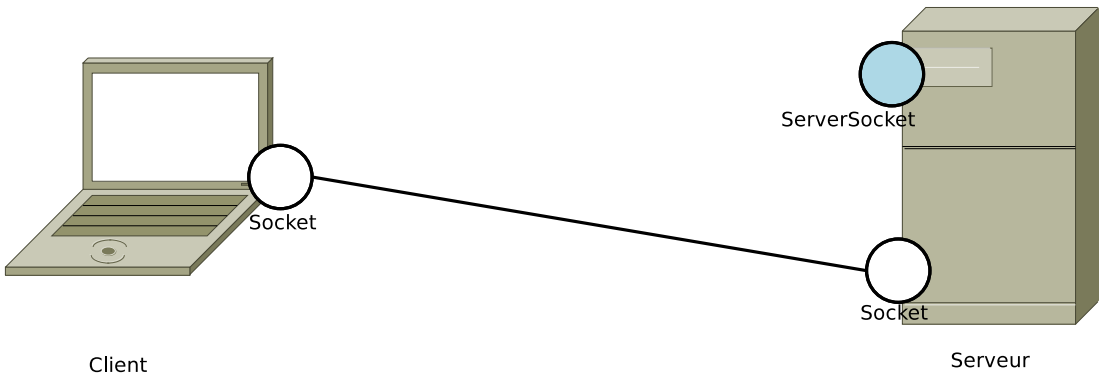


Client



Serveur


ServerSocket
Socket
Client
Serveur


ServerSocket
Socket
Socket
Client
Serveur

Problème

- Lorsque le serveur interagit avec le client, il n'est plus à l'écoute des autres clients
- Il ne peut interagir qu'avec un seul client à la fois
- Ce n'est pas le comportement attendu d'un serveur

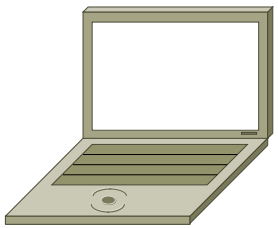
$\Rightarrow$ Utilisation des threads pour traiter chaque client

Serveur

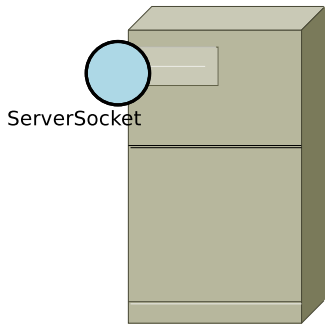
```
ServerSocket serverSock = new ServerSocket(5000);
while(true){
 Socket clientSocket = serverSock.accept();
 Thread t = new Thread(new ClientHandler(clientSocket));
 t.start();
}
```

Thread communiquant avec chaque client

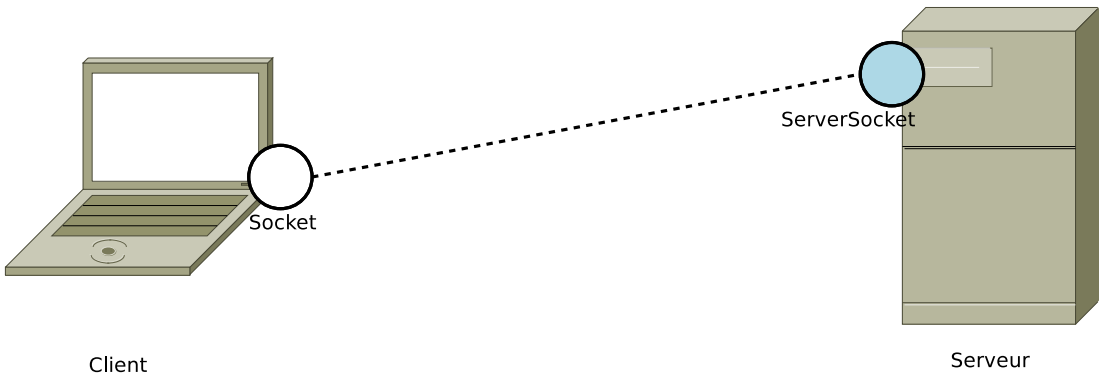
```
public class ClientHandler implements Runnable{

public ClientHandler(Socket socket){
 // initialisation
}

public void run(){
   // Interaction avec le client
}
```

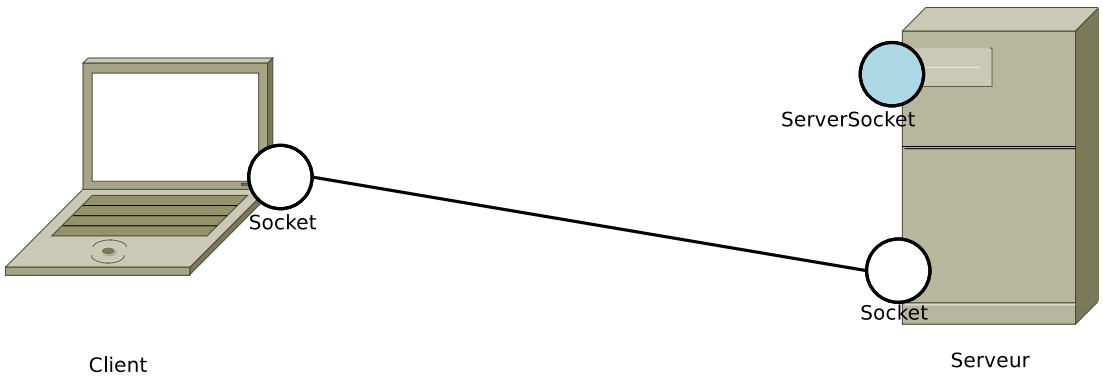


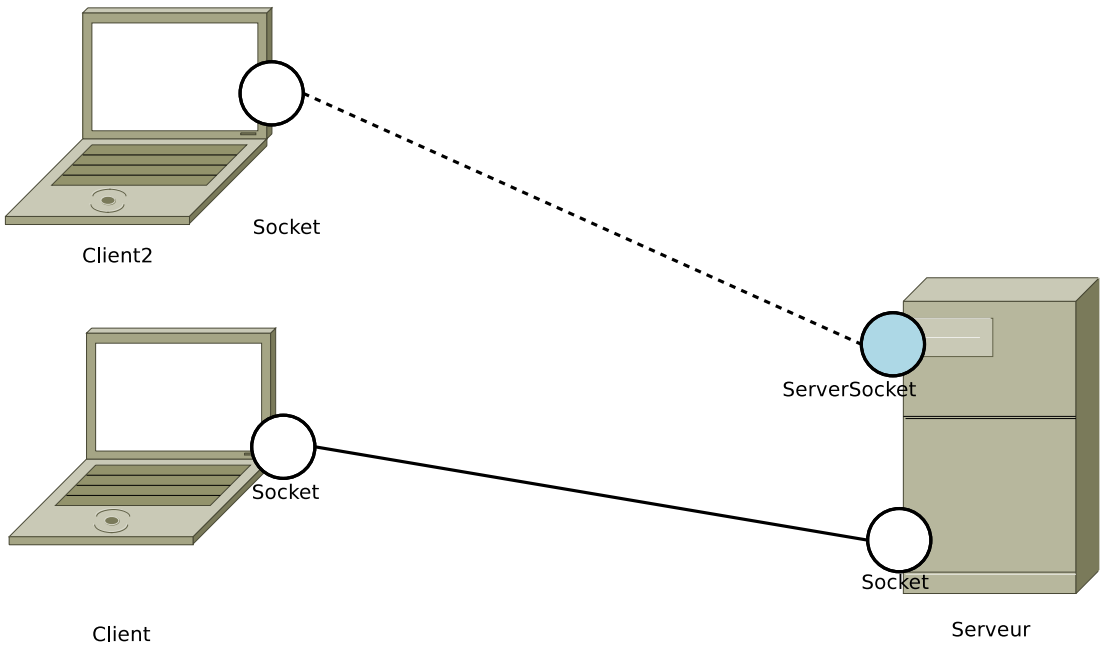
Client

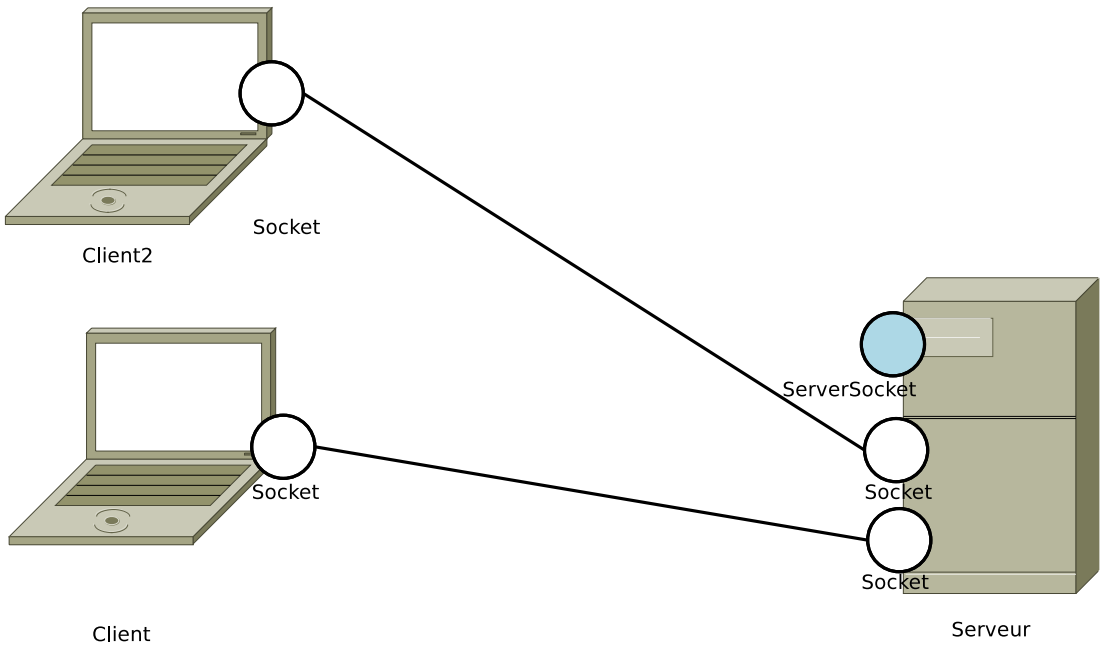


Serveur


ServerSocket
Socket
Client
Serveur


ServerSocket
Socket
Socket
Client
Serveur


Client2
Socket
ServerSocket
Socket
Socket
Client
Serveur


Client2
Socket
Socket
Client
ServerSocket
Socket
Socket
Serveur

Résumé

Client

- Initialiser le socket
- Initialiser les flux d'entrée et de sortie
- Lire et écrire à l'aide de ces flux suivant le protocole mis en place avec le serveur
- Fermer le socket

Serveur

- Initialiser un socket server et attendre la connexion d'un client
- *Communiquer via le socket* (cf client)
- Fermer le socket server

Conclusion

- Socket : extrémité d'une liaison réseau
- Facilite la lecture et l'écriture de données sur le réseau
- En Java classe Socket et ServerSocket
- Nous avons vu le mode connecté en TCP permettant une transmission sûre de données, mais il existe un mode déconnecté en UDP

Liens

- https://docs.oracle.com/javase/tutorial/networking/sockets/
- http://www.javaworld.com/article/2077322/core-java/core-java-sockets-programming-in-java-a-tutorial.html