Développement d'applications réparties

Techniques de programmation d'applications réparties

Khaled Barbaria
khaled.barbaria@ensta.org

Faculté des Sciences de Bizerte
GLSI3

2021

Plan

1. Motivations
2. Notion de Système distribué
3. L'Internet
4. Modèle client/serveur
5. Programmtion Sockets
   - Sockets TCP
   - Sockets UDP
   - Sockets Multicast

Évolutions technologiques

- Développement de microprocesseurs puissants et peu couteux
  - En 1980 : la machine coûte 10^7 dollars et exécute une instruction par seconde
  - En 2005 : la machine coûte 10^3 dollars et exécute 10^9 instructions par seconde.
- Développement des réseaux locaux et à large échelle
  - Les réseaux locaux permettent à une centaine de machines d'échanger des données de petite taille en quelques microsecondes (millisecondes)
  - Les réseaux à large échelle connectent des millions de machines avec des vitesses variables de 64 Kbps (kilobits par seconde) à plusieus gigabits par seconde.

Définitions I

Un Système distribué est une collection de calculateurs indépendants qui apparaît aux utilisateurs comme une seule machine.

- Calculateurs : des petits noeuds du réseau de capteurs aux super-calculateurs les plus puissants en passant par les PC et les smart-phones.
- indépendants : autonomes
- utilisateurs du système distribué : humains, ou autres applications
- une seule machine :
  - Collaboration : Comment établir cette collaboration ?
  - Transparence : cacher la composition interne du système, les différences entre les composants, la manière avec laquelle ils communiquent, ce qui permet de facilement l'étendre ou remplacer certains de ces éléments

Définitions II

- Les utilisateurs interagissent avec le système d'une manière cohérente et uniforme indépendamment du temps et du lieu de l'interaction

Un système distribué est un système dans lequel votre machine peut être rendue inutilisable à cause d'une autre machine dont vous n'avez jamais entendu parler.

- Les défaillances sont généralement partielles et peuvent être très graves
- Transparences : limitées par la réalité
- Problèmes de complexité
- Sécurité des composants
- Sécurité du médium de communication (réseau)

Internet

- Système d'interconnexion de machines utilisant un ensemble standardisé de protocoles de transfert de données.
- Réseau composé de millions de réseaux aussi bien publics que privés, universitaires, commerciaux et gouvernementaux.
- Applications et services variés : courrier électronique, messagerie instantanée, Web, etc.

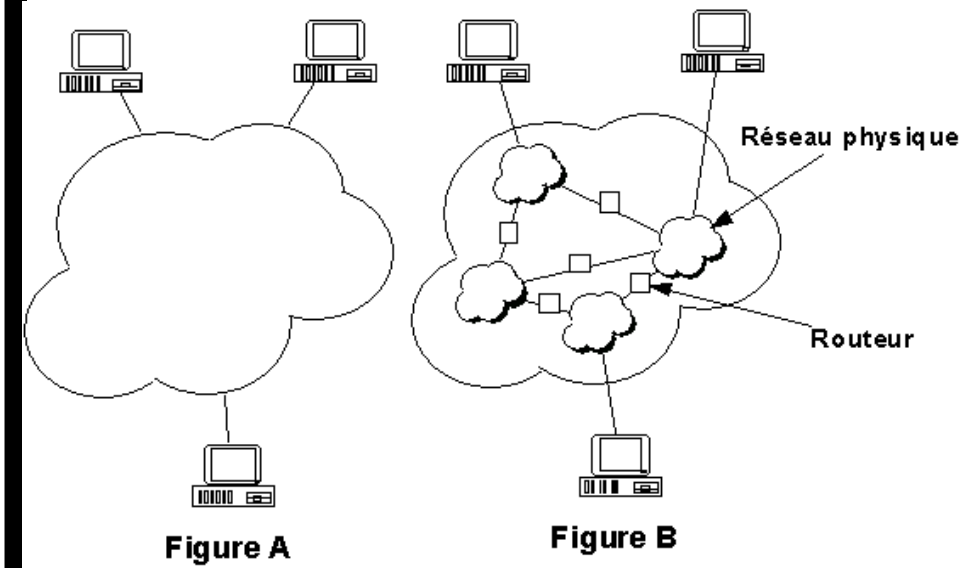


Figure 1 – Internet
[http://www.linux-france.org/]

Protocoles Internet I

IP

- IP se charge de l'acheminement des paquets pour tous les autres protocoles de la famille TCP/IP.
- Somme de contrôle du protocole : confirme l'intégrité de l'en-tête IP.
- Mode non connecté, c'est-à-dire que les paquets émis par le niveau 3 sont acheminés de manière autonome (datagrammes), sans garantie de livraison.

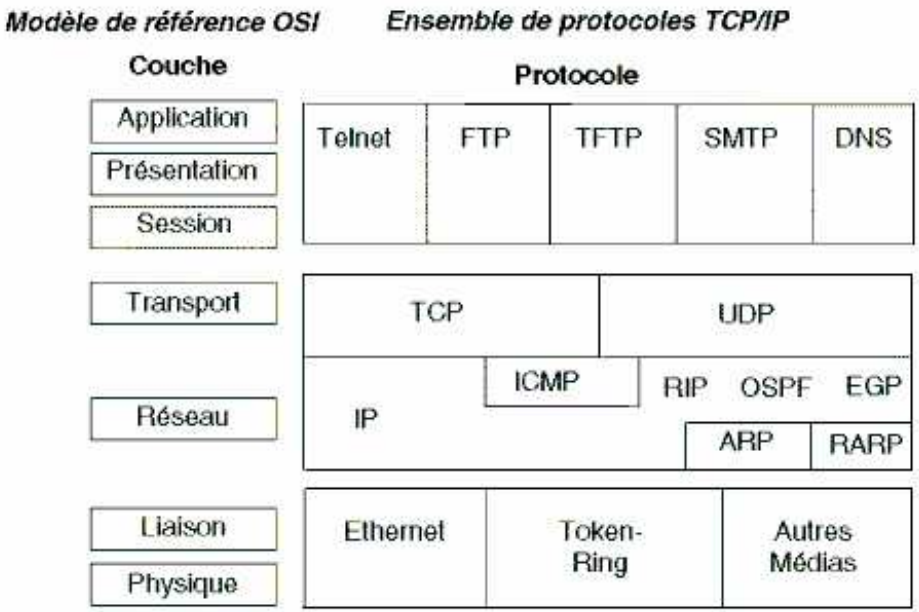


Figure 2 – Protocoles [http://www.linux-france.org/]

Protocoles Internet II

TCP

- TCP fournit un protocole fiable (les paquets sont acquittés et délivrés en séquence, leurs intégrité est vérifiée), orienté connexion, au-dessus d'IP (ou encapsulé à l'intérieur d'IP).
- Cette fiabilité fait de TCP/IP un protocole bien adapté pour la transmission de données basée sur la session, les applications client/serveur et les services critiques tels que le courrier électronique.

Protocoles Internet III

UDP

- Service de datagrammes sans connexion qui ne garantit ni la remise ni l'ordre des paquets délivrés.
- Sommes de contrôle des données facultatives
- Permet d'échanger des données sur des réseaux à fiabilité élevée sans utiliser inutilement des ressources réseau ou du temps de traitement.
- Prend également en charge l'envoi de données vers plusieurs destinataires (multicast).

Protocoles Internet IV

Les protocoles du niveau application les plus connus sont :

- HTTP (Hyper Text Transfer Protocol) permet l'accès aux documents HTML et le transfert de fichiers depuis un site Web
- FTP (File Transfer Protocol) pour le transfert de fichiers
- Telnet pour la connexion à distance en émulation terminal, à un hôte Unix/Linux.
- SMTP (Simple Mail Transfer Protocol) pour la messagerie électronique (sur UDP et TCP)
- SNMP (Simple Network Management Protocol) pour l'administration du réseau
- NFS (Network File System) pour le partage des fichiers Unix/Linux.

Modèle client/serveur I

Modèle C/S

- Se base sur la notion de service et décompose l'application en deux parties : client et serveur
- modèle Client/Serveur largement répandu
  - Serveurs Web, serveurs de temps, serveurs de mail, serveurs de fichiers, serveurs de nommage (e.g. Domain Name Server)
  - ..plein d'autres (transformer une chaîne de caractère en une autre, convertir un fichier, effectuer des calculs exigeants en ressources, etc.)

Serveur

- En attente des demandes de services (Requêtes)
- Les demandes sont traitées et des Réponses sont formulées

Modèle client/serveur II

Client

- Lorsqu'il veut accéder à un service il envoie une requête qui décrit sa demande (càd le service qu'il attend). L'envoi peut être bloquant ou pas (généralement non bloquant).
- Il attend l'arrivée de la réponse. L'attente peut être bloquante ou pas (généralement bloquante).

Modèle client/serveur III

Pour définir une application client/serveur il faut préciser :

- Protocole applicatif (l'ensemble des messages et leurs enchaînements, càd quel message envoyer en réponse à chaque message)
- Protocole du transport (comment les données sont échangées ? exemple TCP, UDP, ou autre) Codes du clients et du serveur
- Code du client
- Code du serveur

Clients lourd et léger I

Une application est dite en client léger si les principaux traitements se font au niveau du serveur. Elle est en client lourd si le client prend en charge l'intégralité du traitement et ne dépend du serveur que pour l'échange des données.

- Le client léger ne se charge que des aspects présentation et potentiellement quelques traitements mineurs (exemple affichage HTML
- Le client lourd prend généralement en charge l'intégralité du traitement (exemples : client avec des capacités d'affichage poussées (3D, GUI), clients utilisant les serveurs uniquement pour stocker les données, Outlook).

Clients lourd et léger II

	Client léger	Client lourd
Autonomie	Dépendant du serveur	Grande autonomie même si serveur indisponible
Eessources nécesaires	Limitées	Bonne à forte exploitation
Fluidité d'exécution	Dépendante du réseau	Dépendante des ressources

État d'un serveur

Un serveur est dit sans état s'il traite chaque requête comme une opération indépendante (aucun lien avec les requêtes précédentes). Il est avec état s'il met à jour, d'une façon durable, des variables internes suite au requêtes des clients.

Sans état	Avec état
Conception simple du serveur	Gestion de l'état nécessaire
Nécessite d'extraire l'information sur le client depuis sa requête	Nécessite de mettre à jour les informations sur les clients
Tolérance aux fautes simple	Nécessite de synchroniser les états entre les répliques

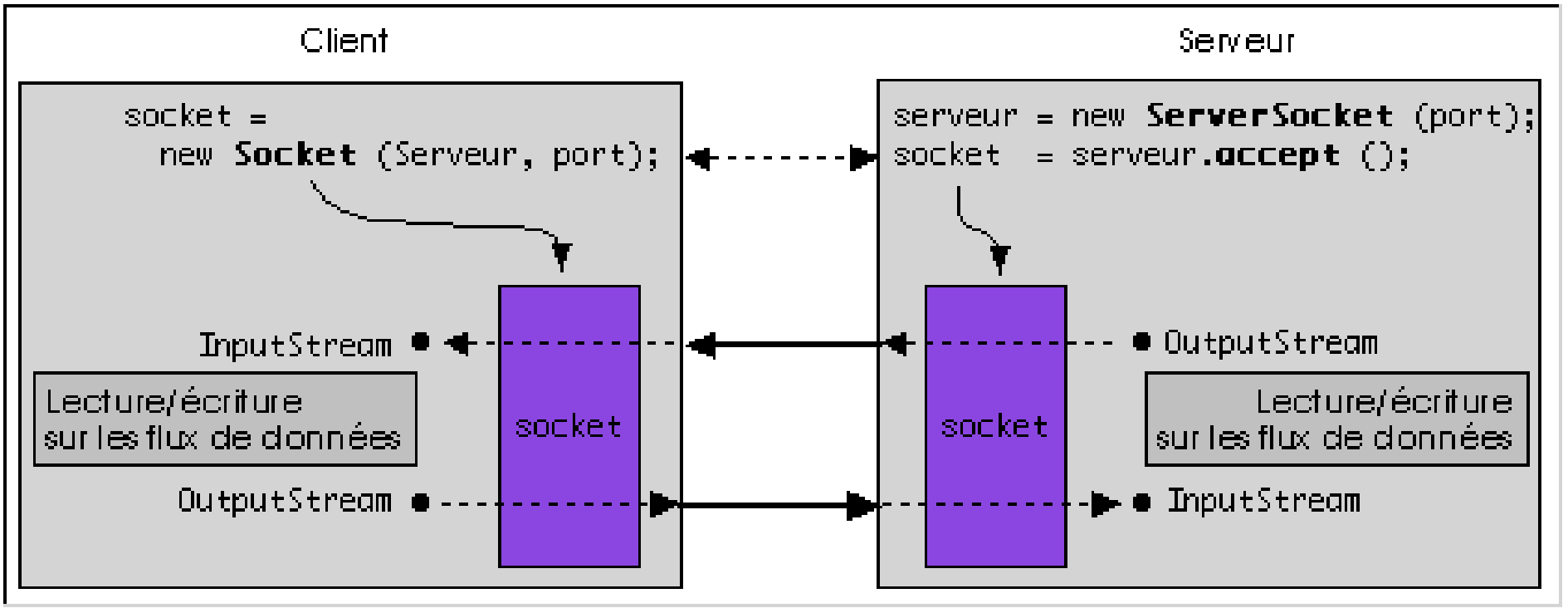
Sockets

- Introduites par BSD (Berkeley Software Distribution)
- Permettent la communication entre les processus
  - Processus sur la même machine
  - Processus connectés par un réseau
- Le protocole réseau (couche 3) utilisé est IP (Internet Protocol)
- Les sockets permettent d'utiliser deux protocoles de transport (couche 4)
  - Transmission Control Protocol (TCP)
  - User Datagram Protocol (UDP) : unicast et multicast

Protocole TCP I

- Orienté connexion (Analogie avec l'appel téléphonique)
  - la première personne compose le numéro, attend que la seconde personne décroche la connexion est alors établie, elle se termine lorsqu'une personne raccroche. La connexion prévue par TCP est durable
  - Remarquer l'utilisation de InputStream et OutputStream
- Protocole fiable (réemission jusqu'à ce que le receveur acquitte l'arrivée des données)
- Contrôle de flux : empêche l'émetteur de submerger le receveur avec les données
- Contrôle de congestion : empêche l'émetteur de saturer le réseau

Protocole TCP II



Protocole TCP III

- Créer une socket Serveur ServerSocket qui attend les connexions. Le numéro de port d'écoute est spécifié ici.
- La demande de connexion du client cause la création d'une nouvelle socket.

- Utiliser le constructeur Socket avec (@IP, port) du serveur.
- Appel bloquant : une fois réussi on peut commencer l'échange des données

=> Il ne reste plus qu'à écrire et lire depuis et vers les flux (streams).

Protocole TCP IV

- Constructeur : ServerSocket (int port) : crée une socket TCP qui attend les connexions au port port.
- Principales méthodes : accept() bloque jusqu'à l'arrivée d'une connexion client, renvoie une nouvelle socket pour dialoguer avec le client. close() ferme la socket

- Constructeur : Socket (string host, int port) : crée une socket TCP connectée à l'hôte spécifié.
- Principales méthodes : getOutputStream() renvoie un flux de sortie permettant d'écrire vers cette socket ; getInputStream() renvoie un flux d'entrée permettant de lire depuis cette socket ; close() ferme la socket.

Flux d'entrée : InputStream

- La méthode read permet de lire le prochain octet du flux.
- L'octet est renvoyé sous forme d'entier entre 0 et 255 (-1 s'il n'existe plus d'octet)
- read bloque jusqu'à ce qu'il y ait des données, que la fin du flux soit détectée ou qu'une exception est levée.

```
1  InputStream istream=
2  new DataInputStream(connectionSocket.getInputStream());
3
4  int data=0;
5  while((data=istream.read())!= 1)
6      System.out.print((char)data);
7  istream.close();
```

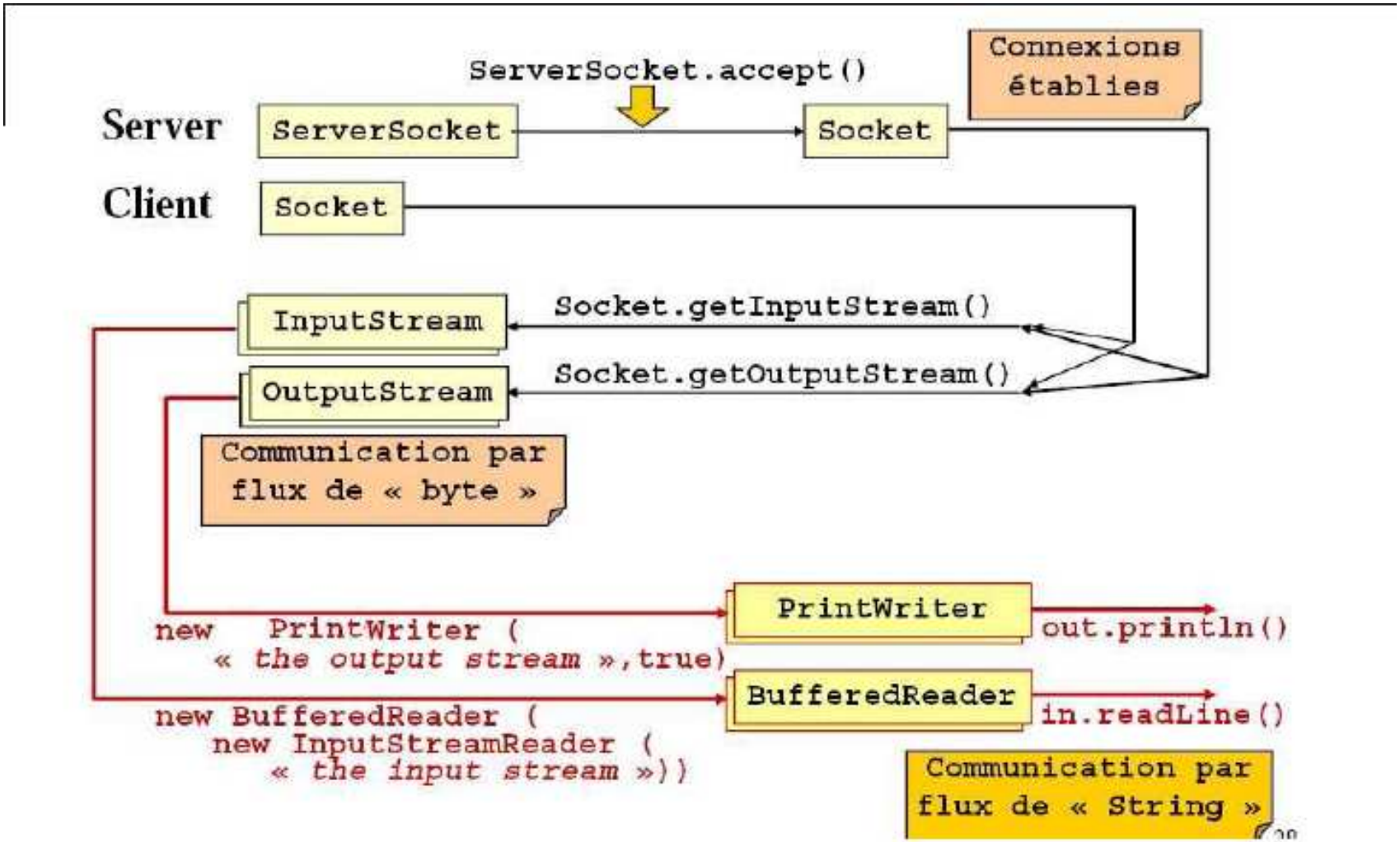
Flux de sortie : OutputStream

OutputStream : flux de sortie

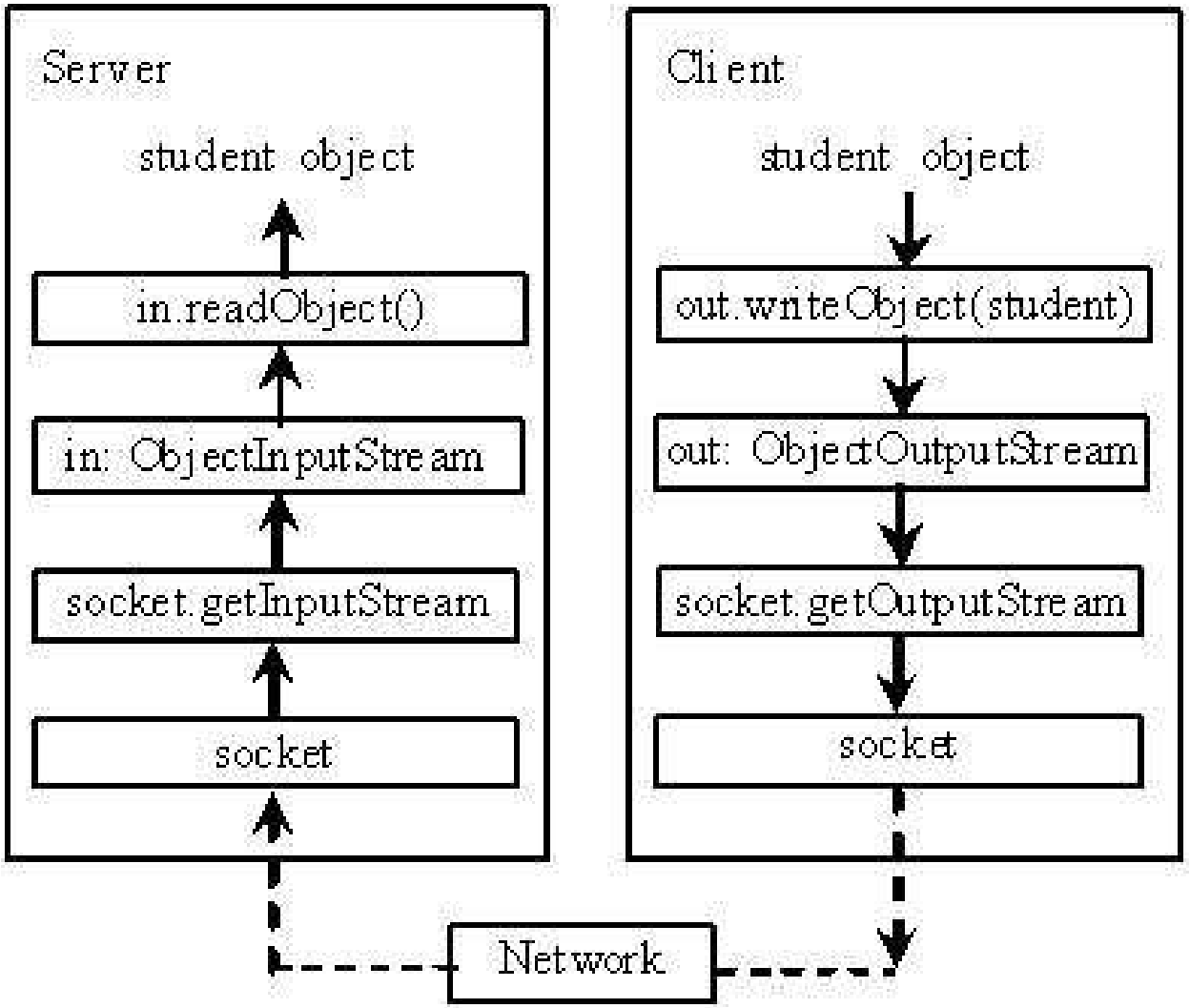
- inscriptible octet par octet grâce à la méthode write.
- La méthode flush permet de forcer l'écriture des octets vers le flux.

```
1  byte [] b =new byte[128]; b=...;
2  Socket s = ....;
3
4  OutputStream theOutput = s.getOutputStream();
5  theOutput.write(b, 0, b.length);
6  //ou encore
7  for(int i=0; i<b.length; i++)
8  theOutput.write (b[i]);
9
10 s.close();
```

Communications par flux de string



Communications basées sur la sérialisation



Encapsuler les descripteurs de flux

```
1  import java.io.*;import java.net.*;
2
3  public class In extends BufferedReader {
4      public In (Socket s) throws IOException{
5    super (new InputStreamReader(s.getInputStream()));
6      }
7
8      public In ()throws IOException{
9    super(new InputStreamReader(System.in));
10     }
11 }
```

```
1  import java.io.*;import java.net.*;
2
3  public class Out extends PrintWriter {
4      public Out (Socket s) throws IOException{
5    super (s.getOutputStream());
6      }
7  }
```

Exemple de serveur TCP

```
1  import java.net.Socket;
2  import java.net.ServerSocket;
3  import java.io.*;
4  import java.util.Date;
5  public class EchoServer {
6      public static void main(String[] args) throws Exception {
7          ServerSocket serverSocket = new ServerSocket(4444);
8    while (true) {
9          Socket clientSocket = serverSocket.accept();
10         BufferedReader in  = new BufferedReader
11       (new InputStreamReader (clientSocket.getInputStream()));
12         PrintWriter out = new PrintWriter(clientSocket.
               getOutputStream());
13         String s = in.readLine();
14         System.out.println("Server received : "+s);
15         out.println("Server replied : "+s);
16         out.println(new Date().toString());
17         out.flush();
18               out.close(); in.close(); clientSocket.close();
19           }
20     }
21 }
```

Exemple de client TCP

```
1
2  import java.net.Socket;
3  import java.io.*;
4  public class EchoClient {
5
6      public static void main(String[] args) throws Exception {
7    String host = "localhost";  int port = 4444;
8          Socket socket = new Socket(host, port);
9          BufferedReader in  = new BufferedReader(new
               InputStreamReader
10           (socket.getInputStream()));
11   PrintWriter out = new PrintWriter(socket.getOutputStream());
12   System.err.println("Connected to " + host + " on port " + port);
13   out.println("I'm the Client"); out.flush();
14   System.out.println(in.readLine());
15   System.out.println(in.readLine());
16         out.close(); in.close(); socket.close();
17     }
18 }
```

Exercices

- Programmer un serveur (TCP) qui renvoie une chaîne de caractères qui renvoie la date et l'heure dès qu'un client se connecte. Les données envoyées par le client ne sont pas traitées. Le serveur ferme la connexion juste après envoi de la chaîne contenant la date. Tester le serveur en utilisant deux méthodes.
- Programmer un client qui permet de trouver les ports occupés par des serveurs TCP sur une machine distante donnée.
- Programmer un serveur qui permet de trouver les ports occupés par des serveurs TCP sur la machine locale.

Serveur de date TCP

```
1  import java.lang.*;import java.io.*;
2  import java.net.*;import java.util.Date;
3  class DateServer {
4      public static void main(String args[]) throws Exception {
5         ServerSocket srvr = new ServerSocket(1235);
6         for(int i=0; i<4;i++){
7        Socket skt = srvr.accept();
8        PrintWriter out = new PrintWriter(skt.getOutputStream(), true)
             ;
9
10       out.println(new Date().toString());
11       out.flush();
12       out.close(); skt.close();
13         }
14         srvr.close();
15     }
16 }
```

Client pour le serveur de date

```
1  import java.lang.*; import java.io.*; import java.net.*;
2
3  class DateClient {
4      public static void main(String args[]) throws Exception{
5    Socket skt = new Socket("localhost", 1234);
6    BufferedReader in = new BufferedReader
7        (new InputStreamReader(skt.getInputStream()));
8          System.out.println(in.readLine()+"\n");
9    in.close();
10     }
11 }
```

Il est également possible d'utiliser telnet

Applications et limites de TCP

Applications

- TCP est utilisé par la majorité des protocoles (applicatifs) tels que HTTP, FTP, SMTP et Telnet qui demandent la fiabilité offerte par ce protocole.
  - Les clients Web utilisent HTTP ou FTP pour télécharger d'une manière fiable depuis un serveur Web.
  - Les utilisateurs du courrier électronique nécessitent la fiabilité de SMTP pour l'emission des emails.

Limites

- Le contrôle le de flux et de congestion causent des problèmes de comportement temporel :
  - performances : temps de transfert
  - déterminisme : distribution des temps de transfert par rapport à la moyenne.
- TCP n'est donc pas adapté aux applications temps réel

Le protocole UDP

- Protocole non connecté (analogue à une communication par courrier). L'adresse de destination est nécessaire à chaque envoi, et aucun accusé de réception (acquittement) n'est donné.
- Pas de garanties de fiabilité (Best effort)
- Meilleur comportement temporel que TCP (performance et déterminisme)
- Bien adapté à la diffusion de vidéo/audio o'u le comportement temporel est important et la la perte de quelques paquets ne pose pas problème

Programmation des sockets UDP en java I

Représente un paquet qui contient toute l'information nécessaire à sa transmission entre les points de connexion.

- Constructeur : DatagramPacket(byte[] buf, int length) construit un paquet pour recevoir des paquets de taille maximale length. DatagramPacket(byte[] buf, int length, InetAddress address, int port) construit un paquet pour envoyer un tableau d'octets.
- Principales méthodes : getLength(), getData(), getPort(), getAddress().

Programmation des sockets UDP en java II

- Constructeur : DatagramSocket(int port) construit une socket et l'attache au port spécifié de la machine locale.
- Principales méthodes : getAddress et getPort renvoient l'adresse de la socket. getLocalPort renvoie le numéro de port d'attache de la socket au niveau du hôte local. send(DatagramPacket p) et receive(DatagramPacket p) pour envoyer et recevoir les paquets datagrammes.

Serveur UDP

```
1  import java.io.*; import java.net.*;
2
3  class UDPServer{
4      public static void main(String args[]) throws Exception{
5    DatagramSocket UDPSock = new DatagramSocket(9876);
6    byte[] rd = new byte[1024];
7    byte[] sd = new byte[1024];
8    while(true){
9        DatagramPacket receivePacket
10     = new DatagramPacket(rd, rd.length);
11       UDPSock.receive(receivePacket);
12       String sentence = new String( receivePacket.getData());
13       System.out.println("RECEIVED: " + sentence);
14       InetAddress IPAddress = receivePacket.getAddress();
15       int port = receivePacket.getPort();
16
17       sd = sentence.toUpperCase().getBytes();
18       DatagramPacket sendPacket =
19     new DatagramPacket(sd, sd.length, IPAddress, port);
20       UDPSock.send(sendPacket);
21   }
22     }
23 }
```

Client UDP

```
1  import java.io.*; import java.net.*;
2
3  public class UDPClient {
4      public static void main(String args[]) throws Exception {
5    BufferedReader inFromUser =
6        new BufferedReader(new InputStreamReader(System.in));
7    DatagramSocket UDPSock = new DatagramSocket();
8    InetAddress IPAddress = InetAddress.getByName("localhost");
9    byte[] sd = new byte[1024];byte[] rd = new byte[1024];
10   String sentence = inFromUser.readLine();
11   sd = sentence.getBytes();
12   DatagramPacket sendPacket =
13       new DatagramPacket(sd, sd.length, IPAddress, 9876);
14   UDPSock.send(sendPacket);
15   DatagramPacket receivePacket =
16       new DatagramPacket(rd, rd.length);
17   UDPSock.receive(receivePacket);
18
19   System.out.println("FROM SERVER:"
20           + new String(receivePacket.getData()));
21   UDPSock.close();
22     }
23 }
```

Sockets Multicast

Multicast

Entre le Broadcast et la communication point à point

- Les membres expriment leur intérêt en joignant un groupe multicast (designé par une adresse IP entre 224.0.0.0 et 239.255.255.255)
- On utilise le TTL pour limiter la portée des packets (protection des bansdes passante)

MulticastSocket

- Constructteur new MulticastSocket(nport)
- Joindre un groupe joinGroup(InetAddress mcastaddr)
- Envoyer les données aux membres du groupe send(DatagramPacket dp, byte ttl)
- Recevoir les données destinées au groupe (idem UDP)
- quitter le groupe leaveGroup(InetAddress mcastaddr)

Serveur Multicast (Subsribers)

```
1  // Le service est répliqué
2  public class MulticastServer{
3      public static void main(String[] args) throws Exception {
4    int port = 5000; String group = "225.4.5.6";
5
6    MulticastSocket s = new MulticastSocket(port);
7    // join the multicast group
8    s.joinGroup(InetAddress.getByName(group));
9
10   // Create a DatagramPacket and do a receive
11   byte [] buf = new  byte[1024];
12   DatagramPacket pack = new DatagramPacket(buf, buf.length);
13   for (int i=0;i<5;i++){
14       s.receive(pack);
15
16       System.out.println("Received data from: " +
17              pack.getAddress().toString() +
18              ":" + pack.getPort()+new String(pack.getData()));
19   }
20   s.leaveGroup(InetAddress.getByName(group));
21   s.close();
22     }
23 }
```

Client Multicast (Publisher)

```
1  // Le client envoie des requêtes à tous les serveurs souscrits
2  public class MulticastClient{
3      public static void main (String[]args) throws Exception {
4          int port = 5000;  String group = "225.4.5.6";
5          int ttl = 1;
6
7    MulticastSocket s = new MulticastSocket();
8
9    // Pas nécessaire de joindre le groupe de
10   // multicast (pas de réception)
11   byte buf[] = new String("Bonjour").getBytes();
12   // Create a DatagramPacket
13   DatagramPacket pack = new DatagramPacket
14       (buf, buf.length, InetAddress.getByName(group), port);
15
16           // Envoi des données, ttl est de type *byte*.
17   s.send(pack,(byte)ttl);
18           s.close();
19     }
20 }
```