Master Génie Logiciel (GL)

Réseaux avancés

Programmation réseau en Java : Sockets

Partie 2

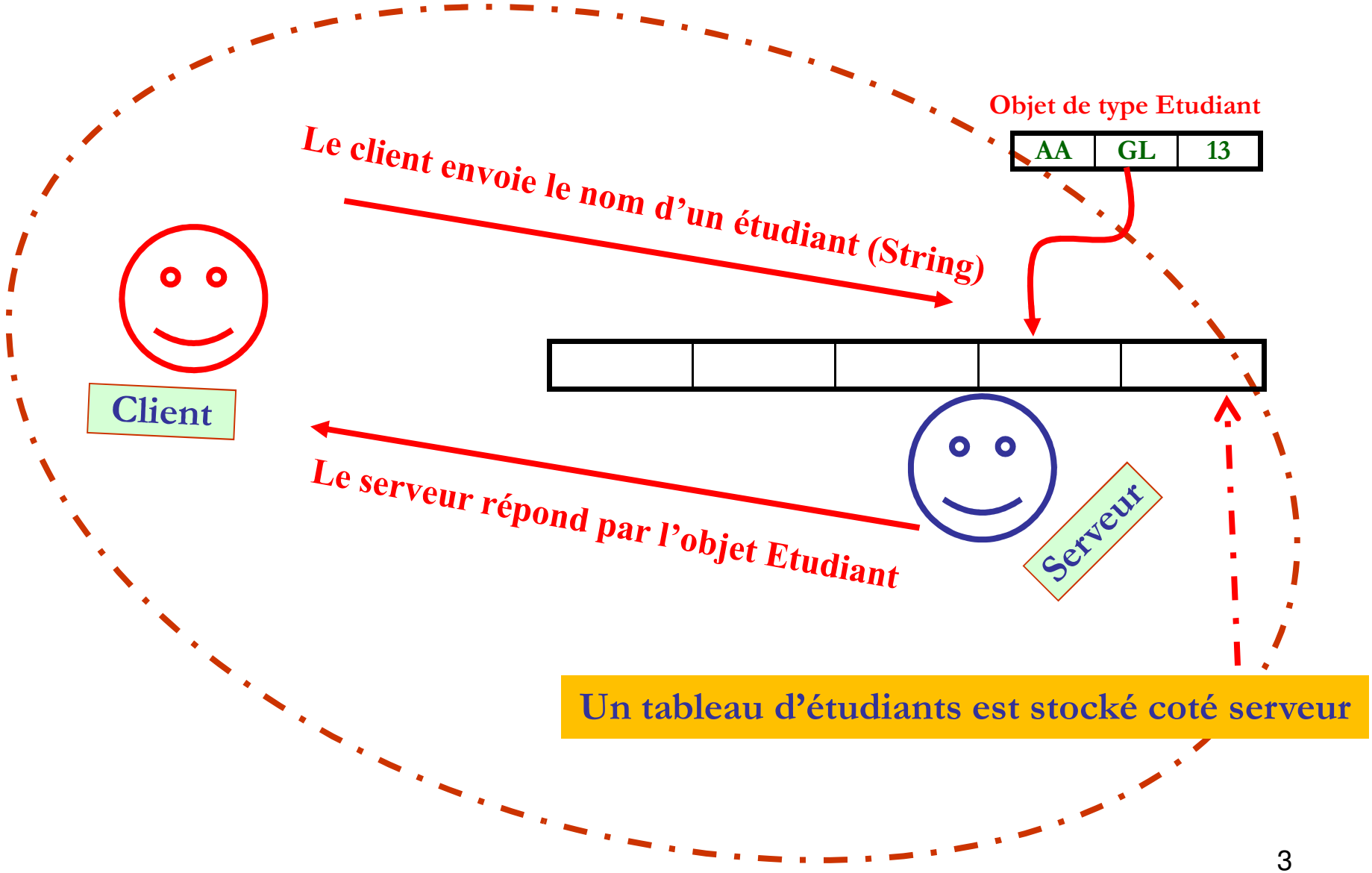
Badr Benmammar

badr.benmammar@gmail.com

Plan

- **Implémentation des Sockets TCP en Java**
  - Transmission d'objet avec les sockets TCP
  - Serveur multi-threads avec les sockets TCP
- **Implémentation des Sockets UDP en Java**
  - Transmission de chaines de caractères
  - Transmission d'objet

Exemple : récupérer un objet Etudiant à partie de son nom



Transmission d'objet par les sockets

```
import java.io.Serializable;
public class Etudiant implements Serializable{
  String nom;
  String specialite;
  int moy;
  Etudiant (String nom, String specialite, int moy) {
    this.nom = nom;
    this.specialite = specialite;
    this.moy = moy;
  }
    String getNom() {
    return nom;
  }
  public String toString() {
    return "Etudiant : "+nom+"   "+specialite+" : "+moy;
  }
}
```

L'objet transmis via le réseau doit être sérialisable

La classe Etudiant est partagée entre le client et le serveur

```
import java.net.*;
import java.io.*;
class ServerEtudiant {
 public static void main(String args[]) {
  Etudiant[ ] tabEtudiant = {
       new Etudiant ("A", "GL", 13),
       new Etudiant ("B", "RSD", 12),
       new Etudiant ("C", "SIC", 14)};
  ServerSocket server = null;
  try {
   server = new ServerSocket(7777);
   while (true) {
    Socket sock = server.accept();
    System.out.println("connecte");
    ObjectOutputStream sockOut = new  ObjectOutputStream(sock.getOutputStream());
    BufferedReader sockIn = new BufferedReader(new InputStreamReader(sock.getInputStream()));
    String recu;  while ((recu = sockIn.readLine()) != null) {
     System.out.println("recu :"+recu);
     String nom = recu.trim();
     for (int i=0; i<tabEtudiant.length; i++)
      if (tabEtudiant[i].getNom().equals(nom)) {sockOut.writeObject(tabEtudiant[i]);
       break;
     }// fin if
    }
    sockOut.close();
    sock.close();
   }
  } catch (IOException e) {
   try {server.close();} catch (IOException e2) {}
  }
 }// fin main
}// fin classe
```

Code du serveur

Lire le nom envoyé par le client

Confirmer la réception

Parcourir le tableau pour trouver l'étudiant

Enregistrer l'objet Etudiant afin de l'envoyer au client

Code du client

```
import java.io.*; import java.net.*;
public class ClientEtudiant {
  public static void main(String[] args) throws IOException {
 String hostName = "localhost";
String NomEtudiant = "A";
    Socket sock = null;
    PrintWriter sockOut = null;
    ObjectInputStream sockIn = null;
    try {
     sock = new Socket(hostName, 7777);
     sockOut = new PrintWriter(sock.getOutputStream(), true);
     sockIn = new ObjectInputStream(sock.getInputStream());
    } catch (UnknownHostException e) {System.err.println("host non atteignable : "+hostName);  System.exit(1); }
       catch (IOException e) {System.err.println("connection impossible avec : "+hostName); System.exit(1);}
    sockOut.println(NomEtudiant);
    try {
     Object recu = sockIn.readObject();
      if (recu == null) System.out.println("erreur de connection");
      else { Etudiant etudiant = (Etudiant) recu;
       System.out.println("serveur -> client : " + etudiant);
      }
    } catch (ClassNotFoundException e) {System.err.println("Classe inconnue : "+hostName); System.exit(1);}
     sockOut.close();
    sockIn.close();
    sock.close();
  }
}
```

Envoyer le nom de l'étudiant au serveur

Restaurer l'objet etudiant reçu du serveur

Afficher les caractéristiques de l'étudiant

Transmission d'objet par réseau

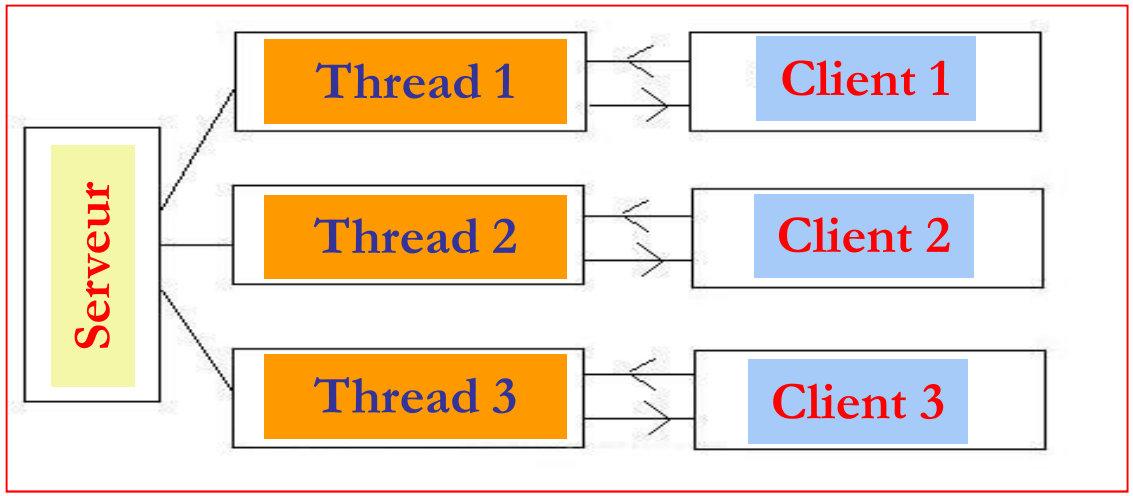
- **Coté serveur :**

  connecte
  recu :A

- **Coté client:**

  serveur -> client : Etudiant : A GL : 13

Serveur multi-threads avec les sockets



Lancer plusieurs clients sur le même numéro de port. Chaque client est pris en charge par un thread.

Client.java

```
import java.io.*;
import java.net.*;
public class Client {
public static void main(String[ ] args){
try {
String recu;
Socket sock;
sock = new Socket("localhost", 2017);
PrintWriter sockOut = new PrintWriter(sock.getOutputStream(), true);
BufferedReader  sockIn = new BufferedReader(new InputStreamReader(sock.getInputStream()));
sockOut.println("C'est BON");
if ((recu = sockIn.readLine())!=null) System.out.println(recu);
sock.close();  } catch (IOException e) { }
}}
```

Classe à exécuter en fonction du nombre de clients

Serveur multi-threads avec les sockets

Serveur.java

```
import java.io.*;
import java.net.*;
public class Serveur {
public static void main(String[] args)
throws IOException{
int nbrclient=0;
ServerSocket socket;
socket = new ServerSocket(2017);
while (true){
try {
Socket s=socket.accept();
nbrclient++;
System.out.println("J'ai "+nbrclient+ " clients");
T thread = new T(s);
thread.start();
} catch (IOException e) {}
} // Fin while
} // Fin main
} // Fin Serveur
```

Le socket de communication est passé comme paramètre au thread

```
class T extends Thread {
private Socket socket;
public T(Socket s){
socket = s;
}
public void run() {
 String recu;
try {
PrintWriter sockOut = new
      PrintWriter(socket.getOutputStream(), true);
BufferedReader  sockIn = new BufferedReader(new
      InputStreamReader(socket.getInputStream()));
if ((recu = sockIn.readLine())!=null){
    System.out.println(recu);
    sockOut.println("OK");
}
socket.close();
} catch (IOException e) { }
} // Fin run
} // Fin class T
```

Le thread doit avoir comme attribut le socket de communication

Le run du Thread est utilisé pour créer le canal de communication.

Exécution coté serveur

J'ai 1 clients

C'est BON

J'ai 2 clients

C'est BON

J'ai 3 clients

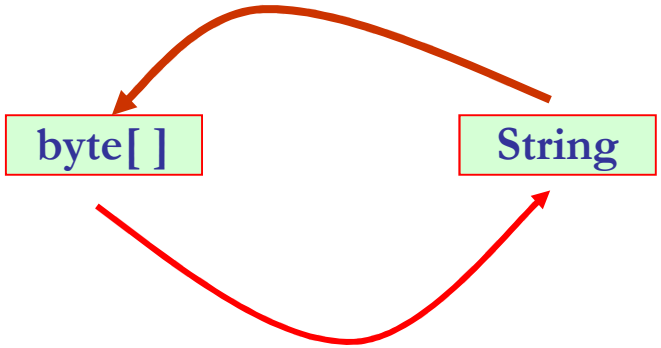
C'est BON

Caractère transmis vs caractère Java (byte vs char)

- **La transmission sur les réseaux est basé sur l'octet : 8 bits.**
- Les méthodes de télécommunication de Java sont donc basées sur le transfert d'octets (byte).

- **Convertir un String en byte[ ] :**
  - String chaine = "ABCD";
  - **byte[ ] message=chaine.getBytes();**

- **Convertir un byte[ ] en String :**
  - **chaine = new String(message);**



byte[ ] tab={65,66,67,68}; // valeurs possibles de -128 à 127.
System.out.println(new String(tab)); → ABCD.

Client UDP : transmission de String

```
import java.net.*;
public class ClientUDP {
  public static void main(String argv[ ]) throws Exception {
     InetAddress serveur = InetAddress.getByName("localhost");
     String mot="BONJOUR";
     int length = mot.length();
     byte buffer[ ] = mot.getBytes();
```

En UDP, il ya un seul type de Socket.

```
receive(DatagramPacket p)
Receives a datagram packet from this socket.
send(DatagramPacket p)
Sends a datagram packet from this socket.
```

```
     // 5 constructeurs pour DatagramSocket (tous pour binder)
     // Créer un socket lié un port quelconque libre
     DatagramSocket socket = new DatagramSocket ( );
     // DatagramPacket utilisé pour envoyer et recevoir les données
     // 6 constructeurs pour DatagramPacket (4 pour envoyer et 2 pour recevoir), aucun avec
     // serveur sous forme de String
     // Création d'un paquet pour envoyer des données (sous forme d'un tableau d'octets)
     DatagramPacket donner_envoyer = new DatagramPacket(buffer, length, serveur, 1000);
     socket.send (donner_envoyer);
     DatagramPacket donner_recu = new DatagramPacket(new byte[1024], 1024);
     socket.receive (donner_recu);
     System.out.println("Données reçues : " + new String (donner_recu.getData()));
     System.out.println("De : " + donner_recu.getAddress() + ":" + donner_recu.getPort()); } }
```

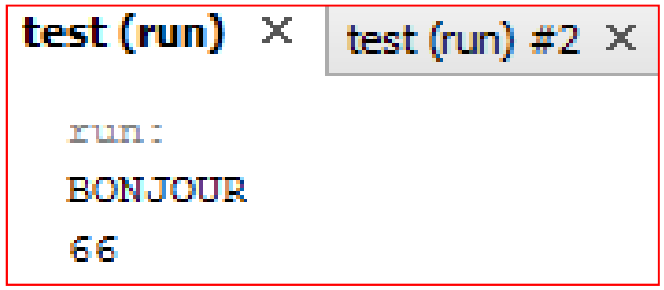
L'adresse de destination à chaque envoi

Serveur UDP : transmission de String

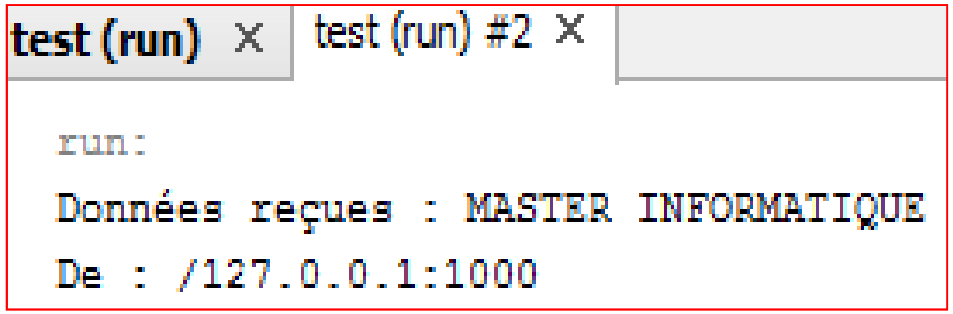
```
import java.net.*;
class ServeurUDP {
 public static void main(String argv [ ]) throws Exception {
 byte buffer [ ] = new byte[1024]; // tampon de réception
     // Créer un socket lié au port 1000.
     DatagramSocket socket = new DatagramSocket(1000);
     while(true){
         // Création d'un paquet pour recevoir les données (sous forme d'un tableau d'octets)
         //  Les données reçues seront placées dans buffer
      DatagramPacket data = new DatagramPacket (buffer,buffer.length);
      socket.receive (data); // data c'est le Paquet
      System.out.println (new String(data.getData ( ))); // data.getData()retourne le buffer
      System.out.println (buffer[0]);
      data.setData ("MASTER INFORMATIQUE".getBytes ( ));
      socket.send (data);
     }
 }
}
```

Exécution

Coté serveur :



Coté client :



Transmission d'objet avec les sockets UDP

```
import java.io.Serializable;
public class Entreprise implements Serializable {
private int id;
private String name;
public Entreprise (int id, String name) {
  this.id = id;
  this.name = name;
}
public int getId() {
 return id;
}
public String getName() {
 return name;
}
public String toString() {
 return "Id = " + getId() + " Name = " + getName();
}
}
```

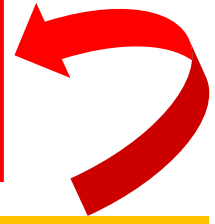
Client UDP

```
import java.io.*;
import java.net.*;
public class UDPSocketClient {
public static void main(String[] args) throws Exception {
Entreprise entreprise = new Entreprise(10, "SOGERHWIT");
DatagramSocket Socket = new DatagramSocket();
InetAddress serveur = InetAddress.getByName("localhost");
byte[ ] buffer = new byte[1024];
// définir une structure dans laquelle les données sont écrites dans un tableau d'octets
ByteArrayOutputStream a = new ByteArrayOutputStream ( ); // classe fille de OutputStream
ObjectOutputStream b = new ObjectOutputStream(a);
b.writeObject (entreprise);
//copier le contenu de a (OutputStream) dans un tableau d'octets pour l'utiliser dans DatagramPacket
byte[ ] data = a.toByteArray();
DatagramPacket envoyer = new DatagramPacket(data, data.length, serveur, 9000);
Socket.send(envoyer);
System.out.println("Objet envoyé par le client");
DatagramPacket recu = new DatagramPacket(buffer,buffer.length);
Socket.receive(recu);
String response = new String(recu.getData());
System.out.println("Réponse du serveur : " + response);
}
}
```

Convertir l'objet entreprise en tableau d'octets

Serveur UDP

```
import java.io.*;
import java.net.*;
public class UDPSocketServer {
public static void main(String[] args) throws Exception {
DatagramSocket socket = new DatagramSocket(9000);
byte[] buffer = new byte[1024];
DatagramPacket recu = new DatagramPacket(buffer, buffer.length);
socket.receive(recu);
byte[ ] data = recu.getData();
//ByteArrayInputStream classe fille de InputStream
ObjectInputStream a = new ObjectInputStream(new ByteArrayInputStream(data));
Entreprise entreprise = (Entreprise) a.readObject();
System.out.println("L'objet entreprise reçu : "+entreprise);
InetAddress IPAddress = recu.getAddress();
int port = recu.getPort();
String reponse = "J'ai bien reçu l'objet";
byte[ ] d = reponse.getBytes();
DatagramPacket envoyer =new DatagramPacket(d, d.length, IPAddress, port);
socket.send(envoyer);
}
}
```



Récupérer l'objet entreprise à partir du tableau d'octets

Exécution

Coté serveur :

L'objet entreprise reçu : Id = 10 Name = SOGERHWIT

Coté client :

Objet envoyé par le client

Réponse du serveur : J'ai bien reçu l'objet

Master Génie Logiciel (GL)

Réseaux avancés
La couche Internet - Partie 1
ARP, ICMP, IGMP et IP

Badr Benmammar

badr.benmammar@gmail.com

Plan

- **Quelques protocoles de la couche internet**
  - **ARP (Address Resolution Protocol)**
    - Connaître l'adresse MAC d'une station à partir de son adresse IP
  - **ICMP (Internet Control Message Protocol)**
    - Gestion des erreurs au niveau IP
  - **IGMP (Internet Group Management Protocol)**
    - Gestion des adresses IP multicast
- **IP (Internet protocol)**
  - IPv4 vs IPv6
  - En-tête IPv4
  - Traceroute : ICMP et TTL

ARP (Address Resolution Protocol)

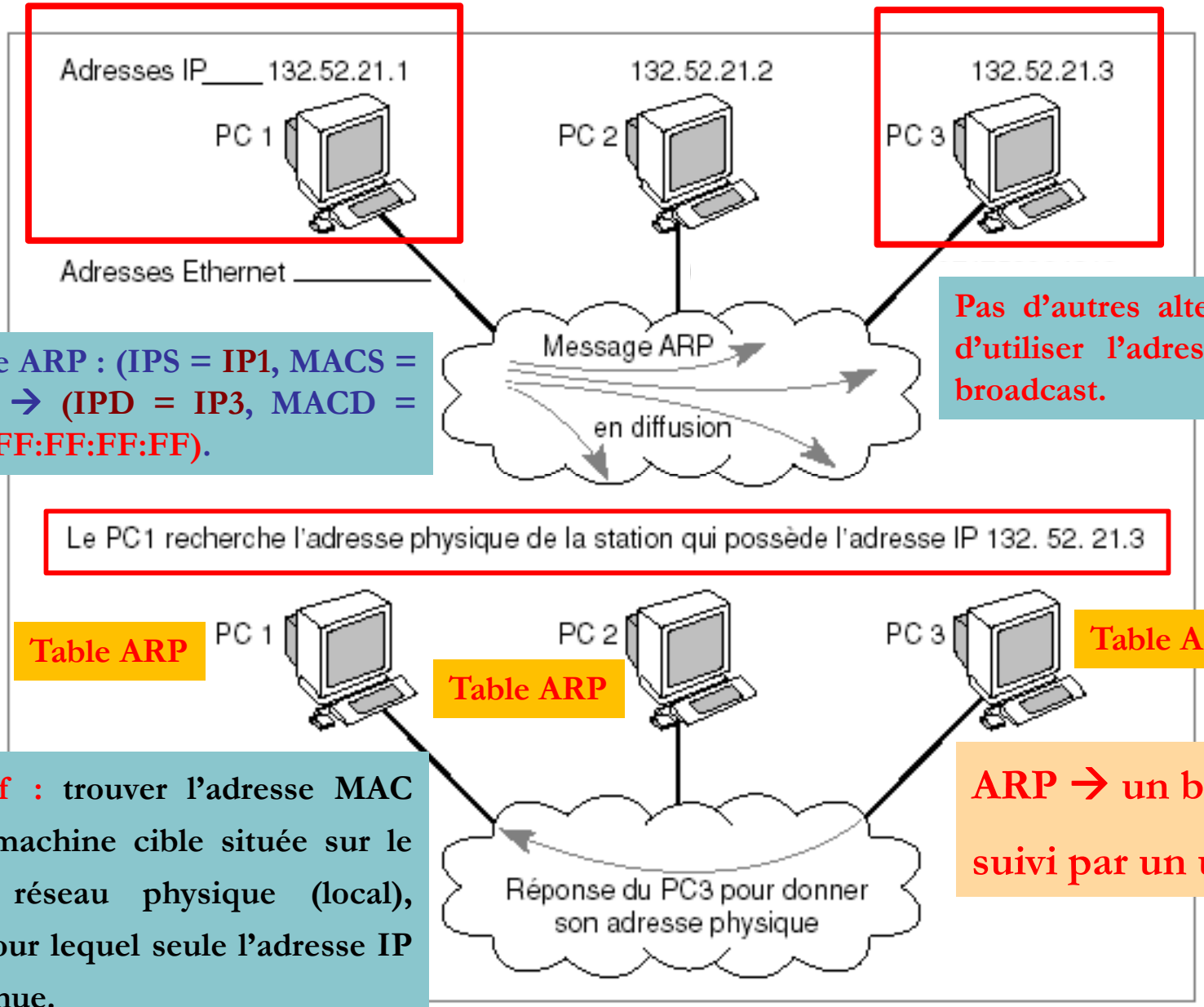
Test d'accessibilité d'un hôte dans un réseau local

- ping 132.52.21.3 (sur PC 1 : 132.52.21.1).

Même réseau sinon routage (voir plus tard)

- Besoin : En-tête IP (IP 1 et IP 2). En-tête MAC (MAC 1 et MAC 2).
- IP 1 (ex. DHCP), MAC 1 (carte réseau) connus par le SE de PC1 (enregistrés et lus à partir de registres).
- IP 2 aussi connue car fournie sur la ligne de commande par l'utilisateur.
- Comment faire pour connaître MAC 2 ?
- Le SE doit lancer en arrière plan (transparent pour l'utilisateur) un protocole afin de découvrir MAC 2 et ainsi découvrir tous les paramètres du ping → Quel est ce protocole ?

ARP (Address Resolution Protocol)



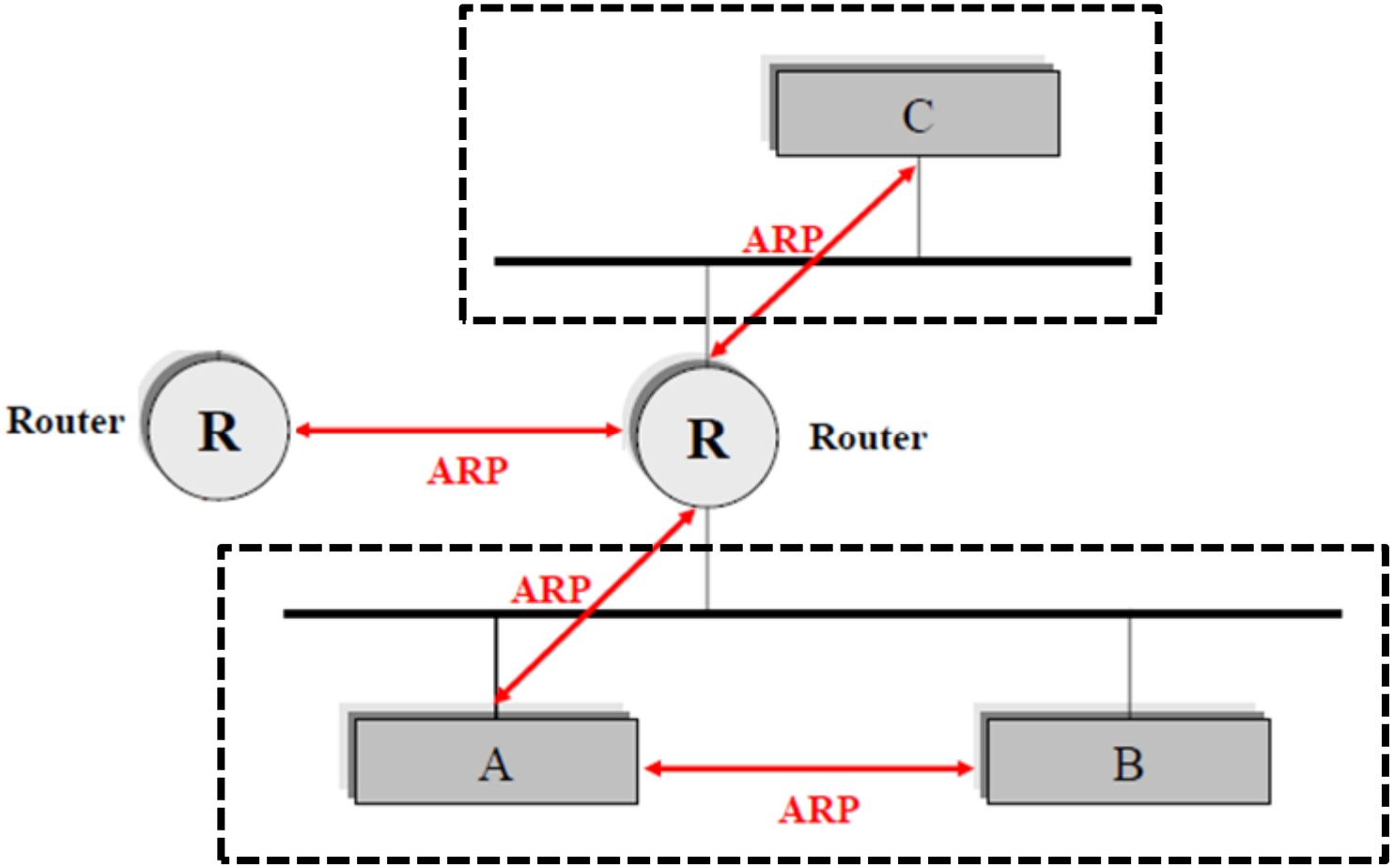
Requête ARP : (IPS = IP1, MACS = MAC1) → (IPD = IP3, MACD = FF:FF:FF:FF:FF:FF).

Pas d'autres alternatives que d'utiliser l'adresse MAC de broadcast.

Objectif : trouver l'adresse MAC d'une machine cible située sur le même réseau physique (local), mais pour lequel seule l'adresse IP est connue.

ARP → un broadcast suivi par un unicast.

ARP (Address Resolution Protocol)



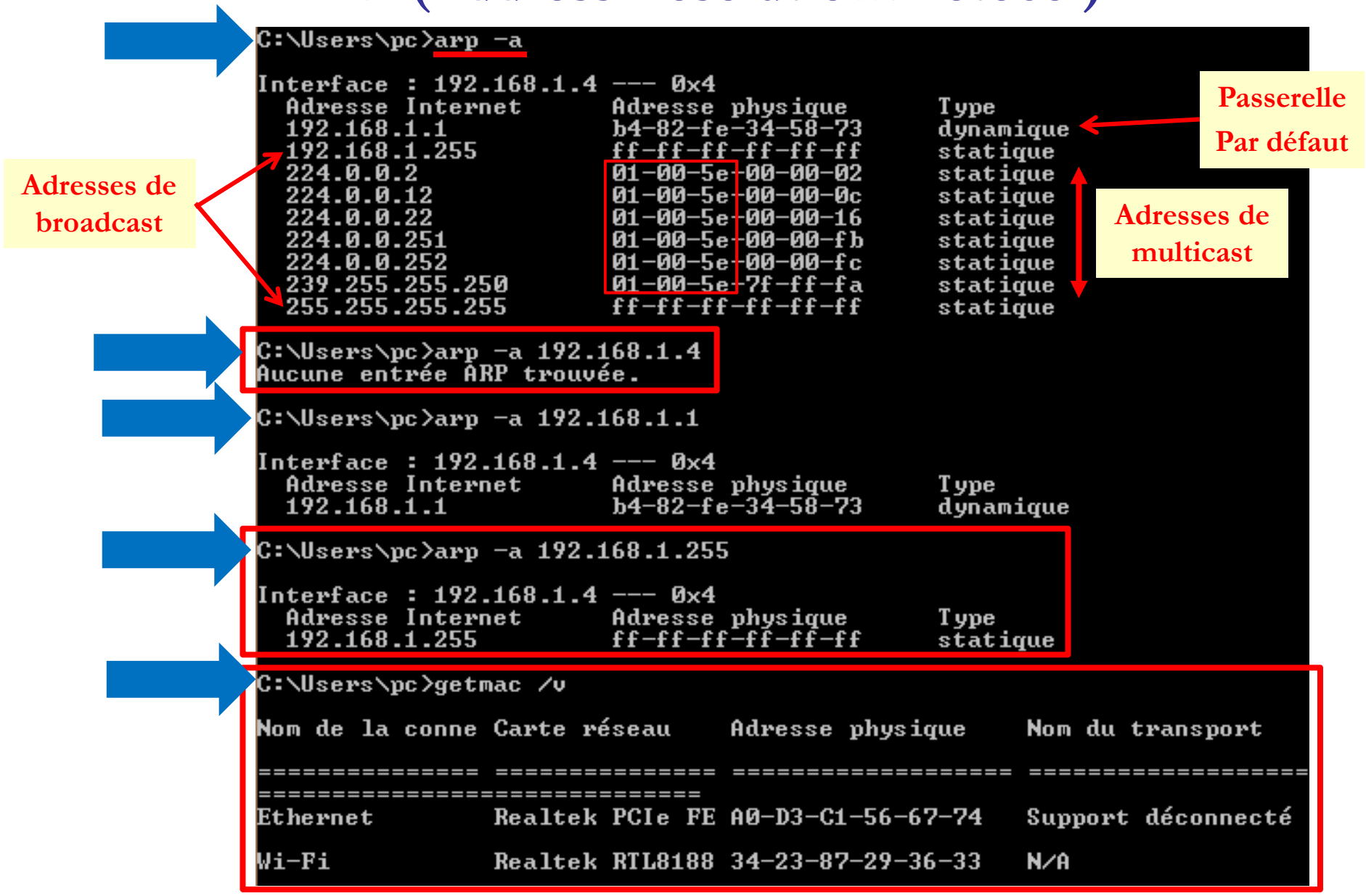
Variante de ARP : Gratuitous ARP

- **Une requête ARP gratuite (Gratuitous ARP)** est une réponse ARP qui n'a pas été sollicitée par une requête ARP.
  - **Objectif :** permettre à un nœud **d'annoncer son (IP, MAC)** (il vient de rejoindre le réseau) ou **de mettre à jour son (IP, MAC)** sur l'ensemble du réseau (souvent lors d'une modification de son adresse IP).

- **Requête ARP gratuite : (IPS, MACS) → (IPS, MAC = ff:ff:ff:ff:ff:ff).**
- **Aucune réponse n'est attendue.**
  - Une réponse à une requête ARP gratuite est reçue → Un conflit d'adresses IP sur le réseau (l'une est statique et l'autre est fournie par DHCP par exemple).

- **Table ARP (dynamique) → dans la mémoire.**
- La commande **arp -a** permet d'avoir le contenu de la table de la machine sur laquelle on se trouve.

ARP (Address Resolution Protocol)



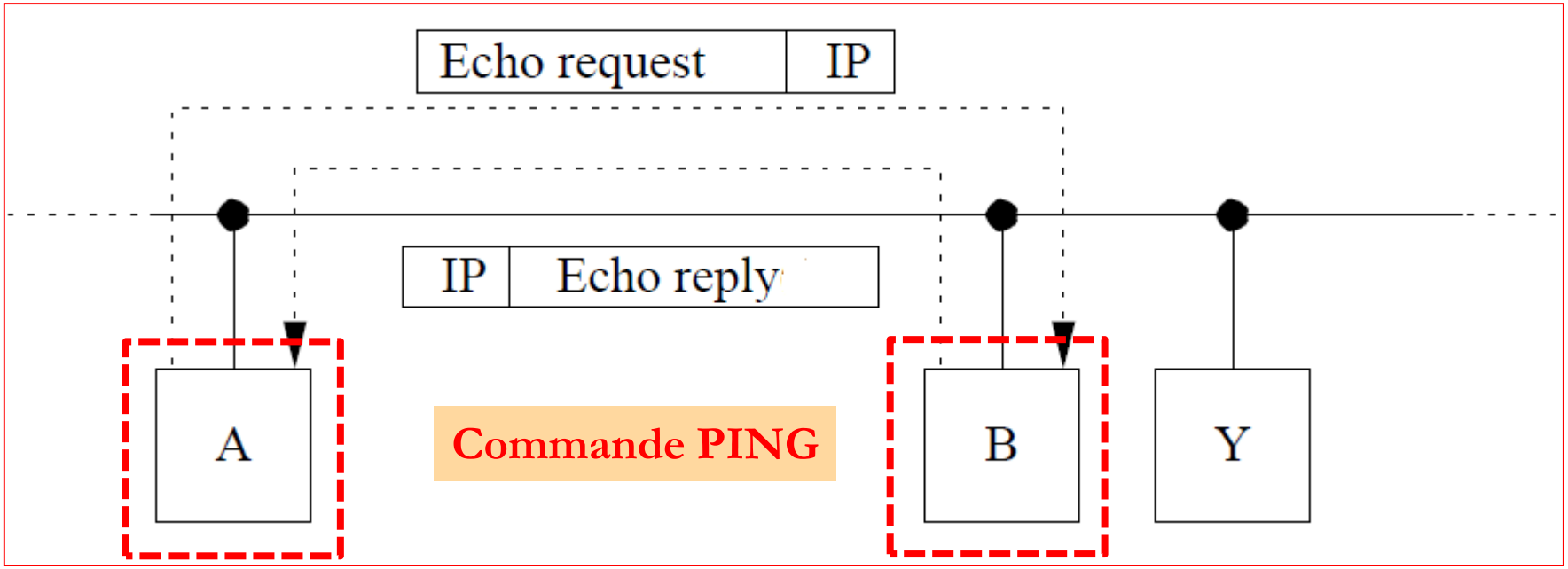
ICMP (Internet Control Message Protocol)

ICMP (Internet Control Message Protocol)

- **Le protocole IP n'est pas prévu pour la gestion des erreurs.**
  - Des paquets IP peuvent ne pas être délivrés à leur destinataire et le protocole IP lui-même ne contient rien qui puisse permettre de détecter cet échec de transmission.
  - **Des machines peuvent être en pannes, déconnectées du réseau ou incapables de router les paquets parce que en surcharge.**
- **ICMP est un mécanisme de contrôle des erreur au niveau IP.**
  - Quand le protocole IP n'arrive pas, dans certains cas, à remplir correctement la tâche qui lui est assignée, il l'indique au protocole ICMP qui émet un paquet à destination de la station source notifiant la nature et l'origine de l'erreur.
- **La fonctionnalité principale de ICMP est de rapporter, à la station émettrice du paquet, les erreurs qui peuvent se produire au niveau IP.**
- **Exemple : Utilisation de ICMP avec le champ TTL pour la commande Traceroute.**

Quelques types de messages ICMP

- **Echo Request :** une machine envoie un message ICMP **"echo request"** pour tester si son destinataire est accessible.
- **Echo Reply :** la machine qui reçoit une telle requête doit formuler un message ICMP **"echo reply"** en retour.
- Ce mécanisme est extrêmement utile, la plupart des implémentations le propose sous forme d'un utilitaire **(ping)**.



TP 1 : Méthode isReachable de InetAddress.

IGMP (Internet Group Management Protocol)

IGMP (Internet Group Management Protocol)

- **IGMP → gestion des adresses multicast (adresses de groupe).**

- L'usage du multicast étant par construction dédié aux applications comme la radio ou la vidéo sur le réseau, donc **consommatrices de bande passante.**

- Les flux ayant une adresse multicast sont à destination d'un groupe d'utilisateurs dont l'émetteur ne connait ni le nombre ni l'emplacement → **analogie avec l'envoi de mail à une boite commune.**

- **IGMP est un protocole de communication entre les routeurs (susceptibles de transmettre des flux multicast) et des machines qui veulent s'enregistrer dans tel ou tel groupe.**

- **Objectif :** permettre aux routeurs de savoir s'il y a des utilisateurs de tel ou tel groupe sur les LANs directement accessibles pour ne pas encombrer les bandes passantes associées avec des flux d'octets que personne n'utilise plus.

IGMP (Internet Group Management Protocol)

- Dans java.net → public class MulticastSocket extends DatagramSocket

`joinGroup(InetAddress mcastaddr)` ←
Joins a multicast group.

`joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)`
Joins the specified multicast group at the specified interface.

`leaveGroup(InetAddress mcastaddr)` ←
Leave a multicast group.

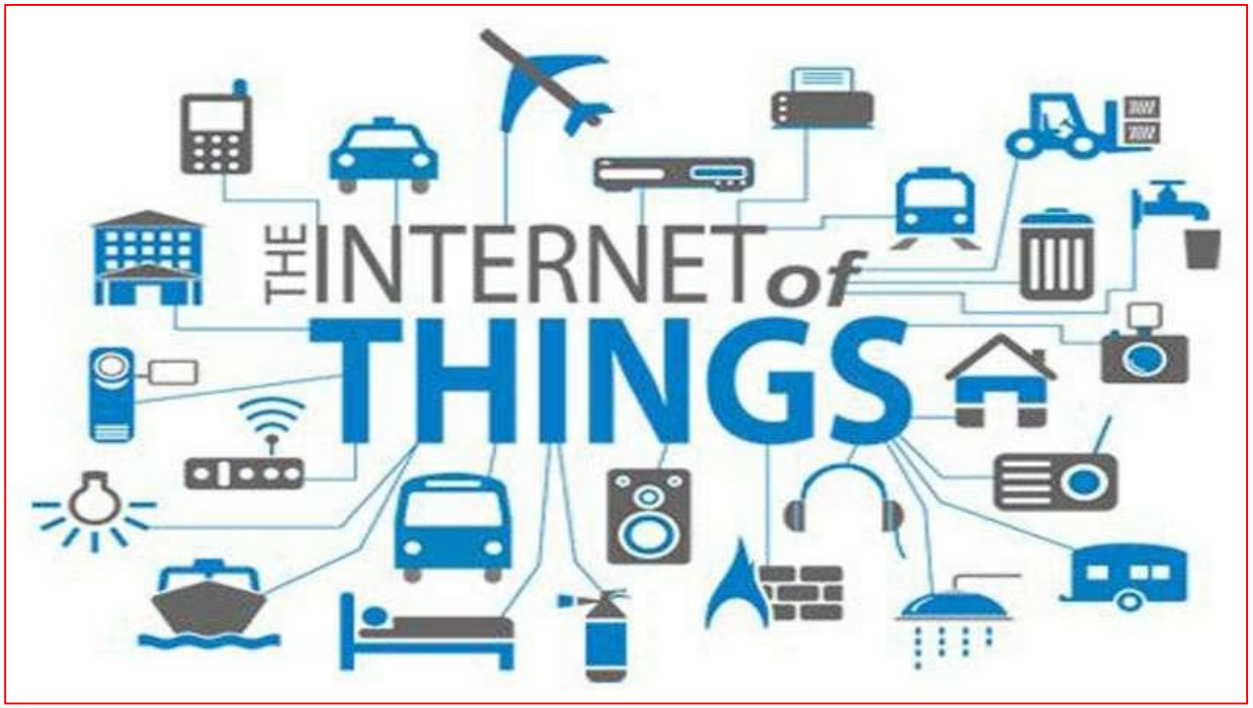
`leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf)`
Leave a multicast group on a specified local interface.

Voir en TP : envoyer un objet à une adresse multicast en utilisant la classe MulticastSocket.

IP (Internet protocol)

IPv4

- **Adresse IP ou adresse logique :** constituée avec 32 bits (IPv4), soit **4 octets** notés en décimal en séparant les octets par les points, dont les octets vont de 0 à 255.
- Capacité d'adressage d'environ **4,3 milliards d'adresses** **(2^{32} = 4 294 967 296)** et donc arrive aujourd'hui à saturation, surtout en Asie.
- **Problématique :** utilisation croissante d'Internet et l'apparition d'objets capables de se connecter à Internet, phénomène autrement désigné par l'Internet des choses **(Internet of Things ou IoT) → 5G → smart cities.**



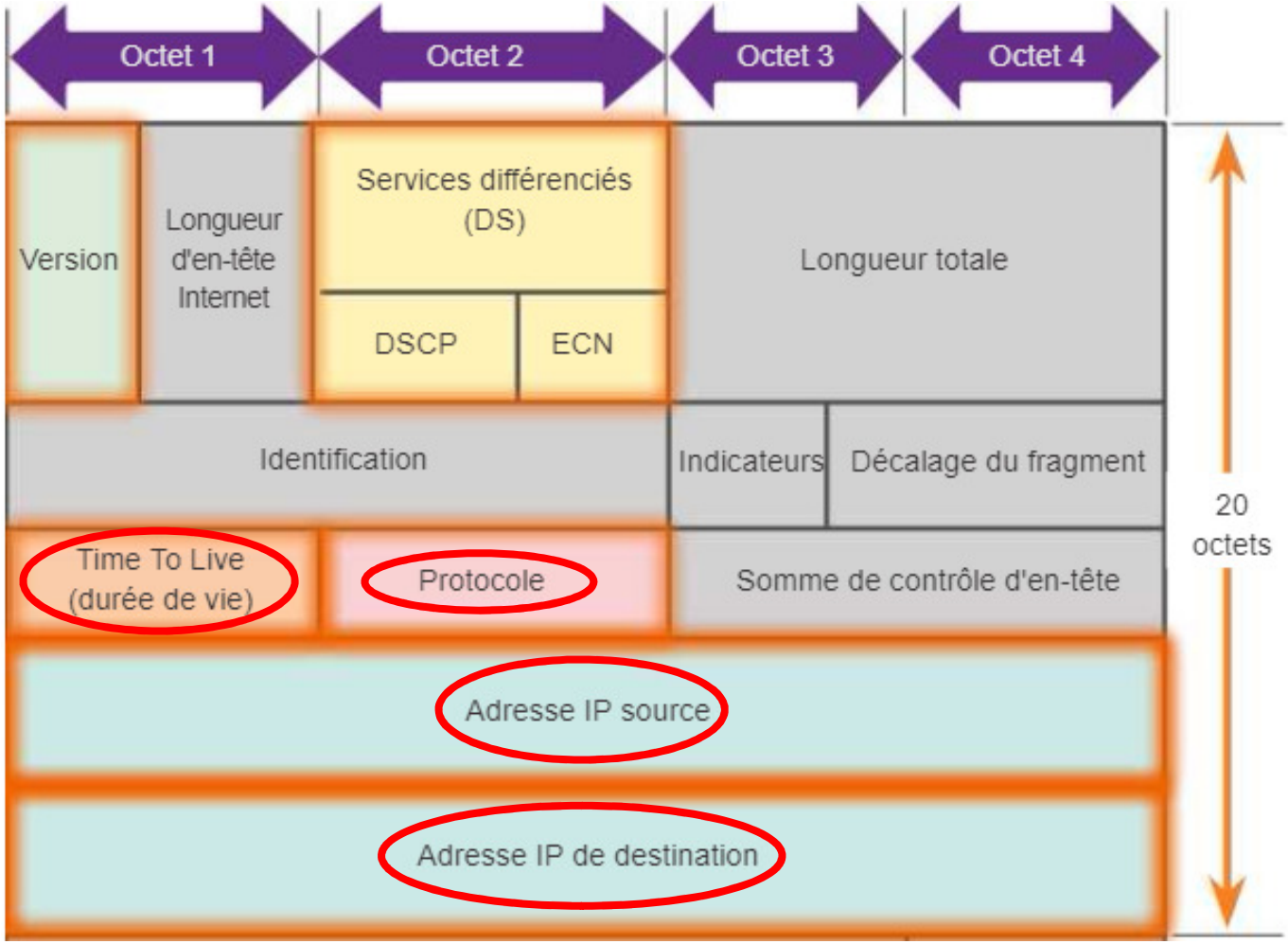
IPv6

- **Solution :** remplacer IPv4 pour satisfaire le nombre croissant de réseaux dans le monde et résoudre le problème d'épuisement.

- **IPv6 :** sous la forme : **"x:x:x:x:x:x:x:x"** où **x représente les valeurs hexadécimales** des **8 portions** de **16 bits** de l'adresse.
  - Exemple : **5f06:b500:89c2:a100:0000:0800:200a:3ff7**.
  - Offre une capacité d'adressage d'environ **(2^{128} = $(2^{32})^4$ = (capacité d'adressage IPv4)^4 → 340 282 366 920 938 463 463 374 607 431 768 211 456.**

- Pour saturer le système, il faudrait placer plus de **667 millions de milliards** d'appareils connectés à internet sur **chaque millimètre carré de surface terrestre.**

En-tête IPv4


Octet 1
Octet 2
Octet 3
Octet 4
Version
Longueur d'en-tête Internet
Services différenciés (DS)
DSCP
ECN
Longueur totale
Identification
Indicateurs
Décalage du fragment
Time To Live (durée de vie)
Protocole
Somme de contrôle d'en-tête
Adresse IP source
Adresse IP de destination
20 octets

En-tête IPv4

- **Champ Protocole :**

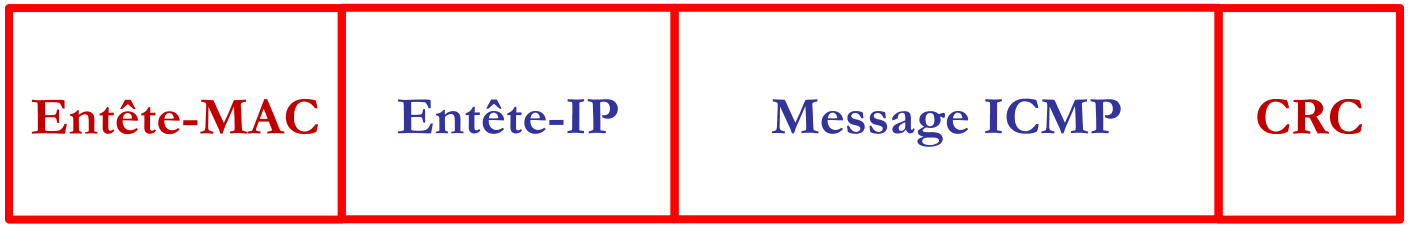
- **Protocole** – cette valeur binaire de **8 bits** indique le type de données utiles transportées par le paquet, ce qui permet à la couche réseau de transmettre les données au protocole de couche supérieure approprié.
  - Les valeurs habituelles sont notamment **ICMP (1), IGMP (2), TCP (6) et UDP (17).**

En-tête IPv4

- **Champ Protocole :**

- **Protocole** – cette valeur binaire de **8 bits** indique le type de données utiles transportées par le paquet, ce qui permet à la couche réseau de transmettre les données au protocole de couche supérieure approprié.
  - Les valeurs habituelles sont notamment **ICMP (1), IGMP (2), TCP (6) et UDP (17).**

- Chaque message ICMP traverse le réseau comme une donnée d'un paquet IP :



- Il est important de bien voir que même si les messages ICMP sont encapsulés dans un paquet IP, **ICMP n'est pas considéré comme un protocole de niveau plus élevé.**

En-tête IPv4

- **Champ TTL :**

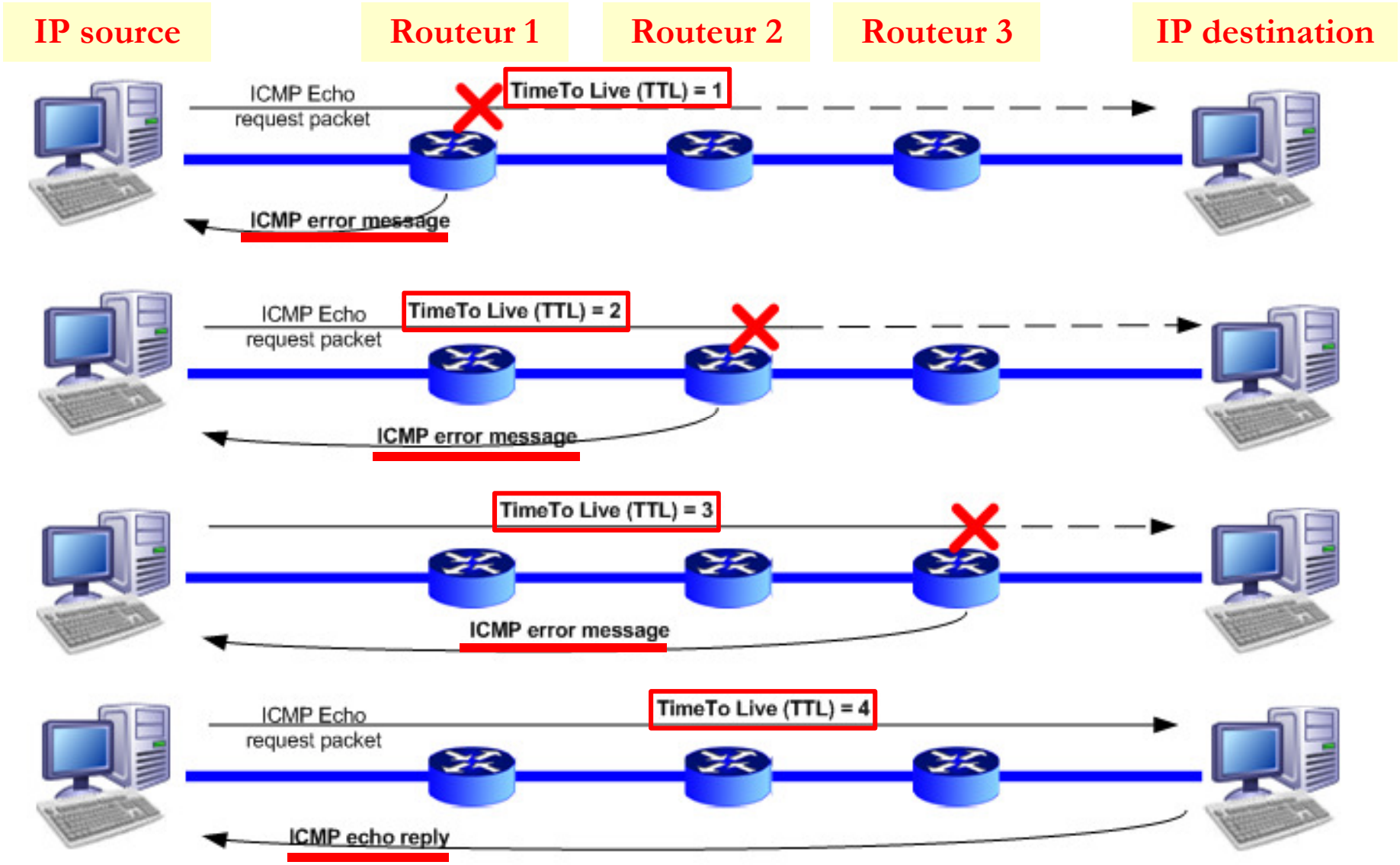
- **Time-to-live (durée de vie, TTL)** – contient une valeur binaire de **8 bits (valeur maximale égale à 255)** utilisée pour limiter la durée de vie d'un paquet → **afin qu'un paquet ne tourne pas d'une manière illimitée sur internet.**
- Cette durée est indiquée en secondes mais est généralement appelée **«nombre de sauts»**.
- L'expéditeur du paquet définit la valeur de durée de vie initiale (couramment 64) et celle-ci **diminue de un** chaque fois que le paquet est traité par un routeur, ou effectue un saut.
- **Si le paquet reste en file d'attente d'un routeur plus d'une seconde, alors la décrémentation sera plus élevée.**

- **Si la valeur du champ TTL (durée de vie) arrive à zéro → le routeur rejette le paquet et envoie un message de dépassement de délai ICMP à l'adresse IP source (ICMP Time exceeded).**

Traceroute : ICMP et TTL

- **Objectif :** suivre les chemins qu'un paquet de données va prendre pour aller d'une machine source à une autre machine connectée au réseau IP.
- **Traceroute** utilise le champ TTL pour identifier les routeurs utilisés entre la source et la destination.
- **Un traceroute consiste à envoyer vers une destination des ICMP echo Request de TTL 1, 2, 3, ... jusqu'à réception du ICMP echo Reply.**
- **La liste des adresses des routeurs intermédiaires qui répondent est alors établie.**
- Chaque routeur qui reçoit un paquet IP en décrémente le TTL avant de le transmettre. Lorsque le TTL atteint 0, le routeur émet un paquet **ICMP Time Exceeded** vers la source. Traceroute découvre ainsi les routeurs de proche en proche.
- Une fois le paquet arrivé à sa destination finale, traceroute cesse de recevoir des paquets time exceeded, et reçoit un paquet ICMP **echo Reply** ayant pour adresse IP source celle de l'interface de l'équipement sondé à travers laquelle est émis le paquet ICMP.

Traceroute : ICMP et TTL



Tracert sous Windows

- 9 routeurs pour atteindre **172.217.19.36** (l'une des adresse IP publique de google).
- **192.168.1.1** est la passerelle par défaut.
- **-d** pour ne pas faire de recherche DNS sur chaque adresse IP.

```
C:\Users\pc>tracert -d www.google.com

Détermination de l'itinéraire vers www.google.com [172.217.19.36]
avec un maximum de 30 sauts :

  1     2 ms     3 ms     2 ms  192.168.1.1
  2    27 ms    30 ms    27 ms  41.98.64.1
  3    27 ms    26 ms    27 ms  10.103.13.45
  4    36 ms    36 ms    35 ms  172.28.16.13
  5    35 ms    37 ms    35 ms  172.28.16.14
  6    38 ms    37 ms    50 ms  172.17.116.16
  7    59 ms    59 ms    58 ms  72.14.205.138
  8    60 ms    56 ms    56 ms  108.170.252.241
  9    57 ms    56 ms    55 ms  72.14.233.67
 10    58 ms    58 ms    60 ms  172.217.19.36

Itinéraire déterminé.
```

- **3 echo Request → 3 echo Replay.**

Réseaux Avancés
M1 GL 2019/2020 - TP N° 2 (Sockets)

PARTIE 1 : Application client/serveur avec les Sockets TCP

Exercice 1 : manipulation d'objets avec TCP

Soit la classe java suivante (**Etudiant.java**), elle représente les caractéristiques d'un étudiant **(son nom, sa spécialité et sa moyenne générale).**

```
import java.io.Serializable;
public class Etudiant implements Serializable{
  String nom;
  String specialite;
  int moy;
  Etudiant (String nom, String specialite, int moy) {
    this.nom = nom;
    this.specialite = specialite;
    this.moy = moy;
  }
    String getNom() {
    return nom;
  }
  public String toString() {
    return "Etudiant : "+nom+"   "+specialite+" : "+moy;
  }
}
```

Le serveur **(ServerEtudiant.java)** contient 3 étudiants, il récupère le nom d'un étudiant (envoyé par le client), cherche cet étudiant dans son tableau et envoie l'objet étudiant correspondant au client.
Pour lire le nom, le serveur utilise **readLine** de **BufferedReader** mais pour envoyer l'objet, il doit passer par **writeObject** de **ObjectOutputStream.**

```
import java.net.*;
import java.io.*;
class ServerEtudiant {
  public static void main(String args[]) {
    Etudiant[ ] tabEtudiant = {new Etudiant ("A", "GL", 13),new Etudiant ("B", "RSD", 12),new Etudiant ("C", "SIC", 14)};
    ServerSocket server = null;
    try {
      server = new ServerSocket(7777);
      while (true) {
        Socket sock = server.accept();
        System.out.println("connecte");
        ObjectOutputStream sockOut = new  ObjectOutputStream(sock.getOutputStream());
        BufferedReader sockIn = new BufferedReader(new InputStreamReader(sock.getInputStream()));
        String recu;  while ((recu = sockIn.readLine()) != null) {
          System.out.println("recu :"+recu);
          String nom = recu.trim();
          for (int i=0; i<tabEtudiant.length; i++)
            if (tabEtudiant[i].getNom().equals(nom)) { sockOut.writeObject(tabEtudiant[i]);break; }
        }
        sockOut.close();
        sock.close();
      } } catch (IOException e) {
      try {server.close();} catch (IOException e2) {}
    }
  }// fin main
  }// fin classe
```

Le client **(ClientEtudiant.java)** utilise **println** de **PrintWriter** pour envoyer le nom de l'étudiant au serveur et récupère l'objet étudiant (envoyé par le serveur) avec **readObject()** de **ObjectInputStream.**

```
import java.io.*; import java.net.*;
public class ClientEtudiant {
  public static void main(String[] args) throws IOException {
 String hostName = "localhost";
String NomEtudiant = "A";
    Socket sock = null;
    PrintWriter sockOut = null;
    ObjectInputStream sockIn = null;
    try {
      sock = new Socket(hostName, 7777);
      sockOut = new PrintWriter(sock.getOutputStream(), true);
      sockIn = new ObjectInputStream(sock.getInputStream());
    } catch (UnknownHostException e) {System.err.println("host non atteignable : "+hostName);  System.exit(1); }
       catch (IOException e) {System.err.println("connection impossible avec : "+hostName); System.exit(1);}
        sockOut.println(NomEtudiant); // envoyer le nom au serveur
    try {
      Object recu = sockIn.readObject(); // récupérer l'objet Etudiant envoyé par le serveur
      if (recu == null) System.out.println("erreur de connection");
      else { Etudiant etudiant = (Etudiant)recu;
        System.out.println("serveur -> client : " + etudiant);
      }
    } catch (ClassNotFoundException e) {System.err.println("Classe inconnue : "+hostName); System.exit(1);}
     sockOut.close();
    sockIn.close();
    sock.close();
  }
}
```

Modifiez le tableau **tabEtudiant** coté serveur afin d'y avoir 12 étudiants. 3 de chaque spécialité (MID, RSD, SIC et GL). Réalisez par la suite les opérations suivantes :

- Le client se connectera au serveur en envoyant une spécialité (exemple, GL). Le serveur répondra par un tableau contenant uniquement les étudiants de cette spécialité (exemple, les étudiants GL).
- Le client se connectera au serveur en envoyant un entier (entre 1 et 19). Le serveur répondra par un tableau contenant tous les étudiants ayant une moyenne supérieure à cet entier.

Exercice 2 : DataOutputStream et DataInputStream

En utilisant **DataOutputStream** et **DataInputStream :**

- Réalisez un modèle client/serveur à l'aide des **sockets TCP** en java, le client devra envoyer un entier **n** au serveur, ce dernier doit répondre par le nième terme de la suite de **Fibonacci.**
  PS : la suite de **Fibonacci** est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont : **0, 1, 1, 2, 3, 5, 8, 13, 21,** etc. Par exemple, si le client envoie **9** au serveur, il récupèrera **21 (le 9ème terme).**
- Réalisez un modèle client/serveur à l'aide des sockets TCP en java, le client envoie un entier au serveur (5 par exemple), ce dernier répond par la table de multiplication de cet entier.
  Le client devra afficher par exemple :

Table de multiplication de 5 :
0 5 10 15 20 25 30 35 40 45 50

Exercice 3 : manipulations bas niveau des sockets TCP

Pour les caractères, le langage JAVA utilise le type « char » basé sur l'Unicode codé sur 16 bits (2 octets). La transmission sur les réseaux est basée sur l'octet (8 bits). Les méthodes de télécommunication de Java sont donc basées sur le transfert d'octets (byte).
Exemple : Pour convertir la chaîne **« ABCD »** en tableau d'octets **byte [ ] tab**, il faut faire ce qui suit :
String s = "ABCD"; byte [ ] tab=s.getBytes(); // tab contient les valeurs 65, 66, 67, 68 respectivement.
Pour convertir **byte [ ] tab** en **String**, il faut faire **chaine = new String (tab);** // utilisation du constructeur de String.

Donc, pour écrire la chaîne « **ABCD** » directement en octet, le client il doit faire :

```
OutputStream sockOut = socket.getOutputStream();  // On récupère le flux d'écriture OutputStream du socket.
// On l'utilise directement pour écrire des bytes.
byte[ ] buffer = new byte[1024]; // déclarer un tableau d'1KO.
buffer = "ABCD".getBytes();
try {
 sockOut.write(buffer); // write(buffer) écrit tous les bytes du buffer.
sockOut.flush();      // forcer l'écriture du contenu du tampon sur le flot, comme true dans PrintWriter
} catch (IOException e) {}
```

Il existe une autre signature de write, **write(buffer, pos, nbre)** écrit **nbre** bytes à partir de la position indiquée. **sockOut.write(buffer, 0, 2);** permet d'écrire uniquement 2 octets à partir de la position 0.

Le serveur peut faire ce qui suit :

```
InputStream sockIn = socket.getInputStream(); // On récupère le flux de lecture InputStream du socket.
byte[ ] buffer = new byte[1024];
int lu;
try {
 lu = sockIn.read(buffer); // lire le flux et le stocker dans buffer. Retourne le nombre de byte lu.
} catch (IOException e) {}
```

Par la suite et en utilisant le constructeur de **String,** le serveur peut convertir **buffer** en **String.**

- Dans ce qui suit et en utilisant les manipulations bas niveau des sockets TCP (**write de OutputStream et read de InputStream**), écrire un modèle client/serveur dans lequel le client envoie au serveur une phrase contenant trois mots (**ex. UNIV GL INFO**). Le serveur répandra par une phrase contenant le premier caractère de chaque mot. Dans l'exemple le serveur répondra par : **UGI.**

PARTIE 2 : Application client/serveur avec les Sockets UDP

Exercice 1 : Multicast et Broadcast avec UDP

Le premier octet d'une adresse IP multicast commence toujours par les 4 bits **'1110'** suivi par 28 bits (adresse du groupe multicast).

11100000 => 224 **(valeur minimale).**

11101111 => 239 **(valeur maximale).**

Donc, le protocole IP utilise les adresses (virtuelles) de : **224.0.0.0 à 239.255.255.255** pour le multicast. Examinez dans ce qui suit le lien suivant :

https://docs.oracle.com/javase/7/docs/api/java/net/MulticastSocket.html

- Quel est le rôle de cette classe ? Examinez en particulier les deux méthodes joinGroup et leaveGroup.
- Soit la classe java suivante. Lancez une exécution. Que fait ce code ?

```
import java.io.*;
import java.net.*;
public class Multicast {
 public static void main(String argv [ ]) throws IOException{
       String msg = "JE SUIS ETUDIANT EN INFORMATIQUE";
       InetAddress group = InetAddress.getByName("230.0.0.0");
       MulticastSocket socket = new MulticastSocket(1000) ;
       socket.joinGroup(group);
       DatagramPacket hi = new DatagramPacket(msg.getBytes(), msg.length(),group, 1000);
       socket.send(hi);
       byte[] buf = new byte[1024];
       DatagramPacket recv = new DatagramPacket(buf, buf.length);
       socket.receive(recv);
       String ch= new String (recv.getData());
       System.out.println(ch);
       socket.leaveGroup(group);
 }
}
```

- Reprendre la classe Entreprise vue en cours, créer un objet de cette classe et envoyer le par MulticastSocket. Bien évidemment il faut aussi le recevoir.

- Examinez le code suivant :

```
import java.net.*;
import java.io.*;
public class Multicast {
      public static void main(String[] args) throws IOException {
       DatagramSocket socket = new DatagramSocket(5000);
       socket.setBroadcast(true);
       InetAddress address = InetAddress.getByName("255.255.255.255");
       byte[] buffer = "RSDGL".getBytes();
       DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, 5000);
       socket.send(packet);
       byte[] buf = new byte[1024];
       DatagramPacket recv = new DatagramPacket(buf, buf.length);
       socket.receive(recv);
       String ch= new String (recv.getData());
       System.out.println(ch);
       socket.close();
   }
}
```

- Quel est le rôle de la méthode setBroadcast ? Testez votre code avec le paramètre false.

- A votre avis, quel est l'avantage du Multicast par rapport au Broadcast ?

Exercice 2 : Manipulation de plusieurs types avec UDP

Soit les deux classes java suivantes :

```
import java.io.*;
import java.net.*;
public class ClientUDPint{
public static void main(String[] args) throws Exception {
    DatagramSocket socket = new DatagramSocket();                    // ligne 5
    InetAddress serveur = InetAddress.getByName("localhost");
        ByteArrayOutputStream a = new ByteArrayOutputStream();
        DataOutputStream b = new DataOutputStream(a);
        // Ecrire : 10 true bonjour 1.2 dans le outputstream
        b.writeInt(10);
        b.writeBoolean(true);
        b.writeUTF("bonjour");
        b.writeDouble(1.2);
       byte[] buffer = a.toByteArray();
       DatagramPacket packet = new DatagramPacket(buffer,buffer.length,serveur ,2000);
       socket.send(packet);
       System.out.println("Client a envoyé 10 true bonjour 1.2 au serveur");
}
}
```

```
import java.io.*;
import java.net.*;
public class ServeurUDPint {
   public static void main(String[] args) throws Exception {
       DatagramSocket socket = new DatagramSocket(2000);
      DatagramPacket packet = new DatagramPacket(new byte[1024] , 1024);
      socket.receive(packet);
      byte[] data = packet.getData();
      ByteArrayInputStream a = new ByteArrayInputStream(data);
       DataInputStream b = new DataInputStream(a);
       System.out.println("Entier recu : "+b.readInt());
       System.out.println("Boolean recu : "+b.readBoolean());
       System.out.println("String recu : "+b.readUTF());
       System.out.println("Double recu : "+b.readDouble());

}
}
```

- Lancez l'exécution des deux classes précédentes. Quel est l'intérêt de DataOutputStream et DataInputStream pour UDP ?
- Examinez la documentation de DatagramSocket dans le lien suivant https://docs.oracle.com/javase/7/docs/api/java/net/DatagramSocket.html et affichez la taille de la zone tampon (buffer) utilisée pour recevoir et pour envoyer les données.
- Déclarez maintenant un tableau d'entier **(ex. int [ ] tab = {1, 6, 8, 9, 13, 10};)** coté client. Ce dernier doit l'envoyer par UDP au serveur. Le serveur affichera par la suite son contenu (les entiers avec des sauts de lignes) → pensez à utiliser ObjectOutputStream.

Exercice 3 : Application client/serveur avec UDP

Dans ce qui suit, nous allons créer les objets suivants coté serveur (qui seront insérés dans un ArrayList) :

```
Joueur a= new  Joueur(10, "MESSI", "FCB");
Joueur b= new  Joueur(9, "BENZEMA", "REAL");
Joueur c= new  Joueur(11, "DEMBELE", "FCB");
Joueur d= new  Joueur(26, "Mahrez","City");
Joueur e= new  Joueur(10, "MODRIC", "REAL");
```

Donc, vous allez commencer par la création de la classe Joueur qui comporte les champs suivants : **numero (int), nom (String) et equipe (String).**

Vous devrez par la suite réaliser les traitements suivants avec UDP :

- Le client envoie un numéro au serveur (ex. 10), le serveur doit répondre par un objet de type ArrayList contenant les joueurs qui portent le numéro 10.
  Le résultat doit être comme ceci : **[numero = 10 nom = MESSI equipe = FCB, numero = 10 nom = MODRIC equipe = REAL].**

- Le client envoie le nom d'une équipe au serveur (ex. FCB), le serveur doit répondre par un objet de type ArrayList contenant les joueurs de cette équipe. Si vous utilisez un buffer de taille supérieur à la taille du mot reçu (coté serveur), utilisez la méthode **trim()** après la conversion du buffer d'octets en **String** pour supprimer les espaces liés aux cases non remplies dans le buffer de réception.
  Le résultat attendu est comme suit : **[numero = 10 nom = MESSI equipe = FCB, numero = 11 nom = DEMBELE equipe = FCB].**

- Affichez coté client les joueurs dont le nom commence par 'M'. Suite à la requête du client le résultat attendu sera comme suit : **[numero = 10 nom = MESSI equipe = FCB, numero = 26 nom = Mahrez equipe = City, numero = 10 nom = MODRIC equipe = REAL].**