Telecommunication Services Engineering (TSE) Lab

Programmation Socket

Fatna Belqasmi, PhD

Research Associate, Concordia University

Objectif:

- Présenter les bases de la programmation des sockets
- Démontrer concrètement comment cela fonctionne avec Java

Agenda:

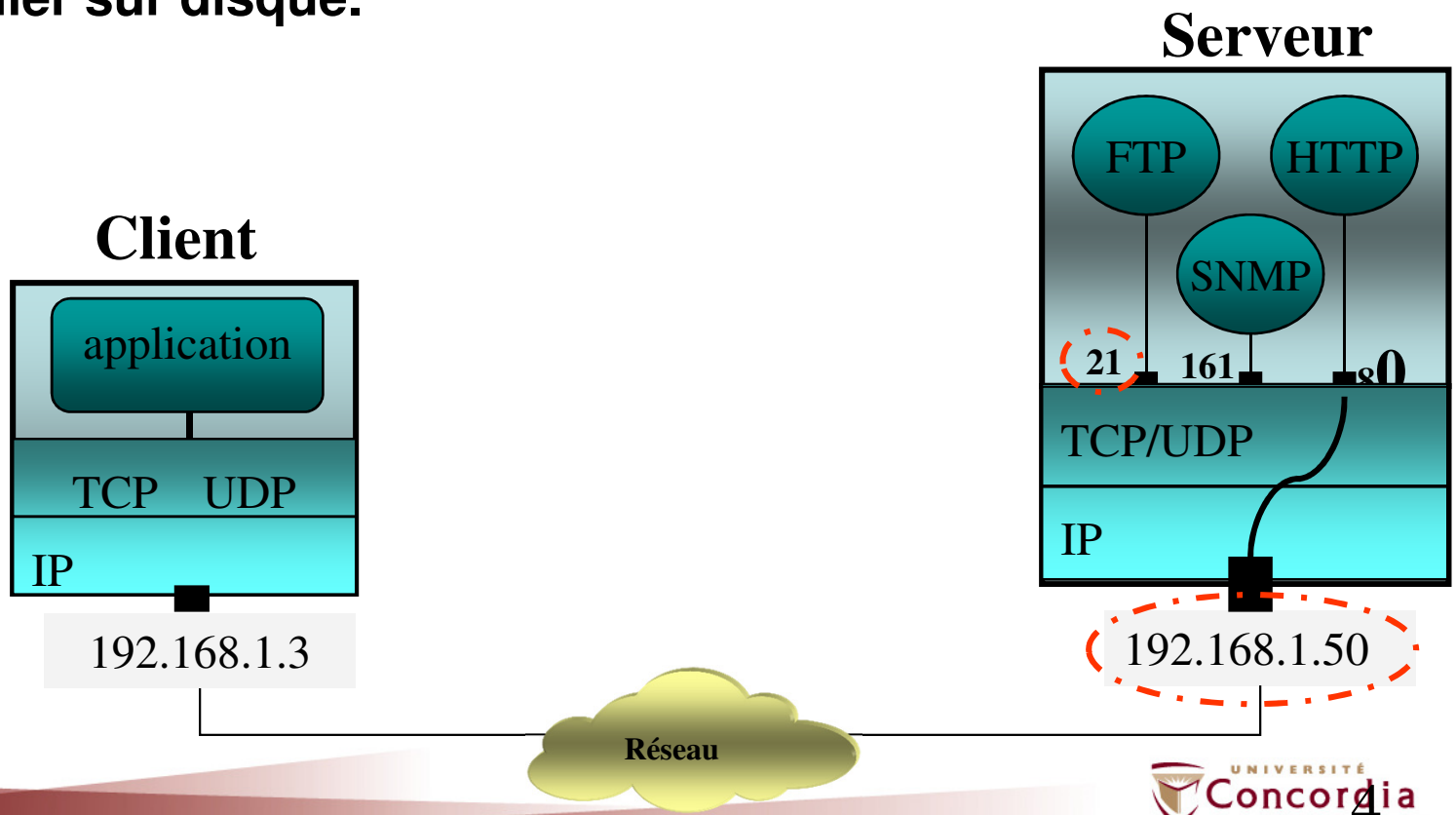
- Principes de base
- Implémentation du serveur et du client
- Envoi et réception de données
- Programmation avancée des sockets

Principes de base

- **Qu'est-ce qu'une socket?**
- **Communication par socket**

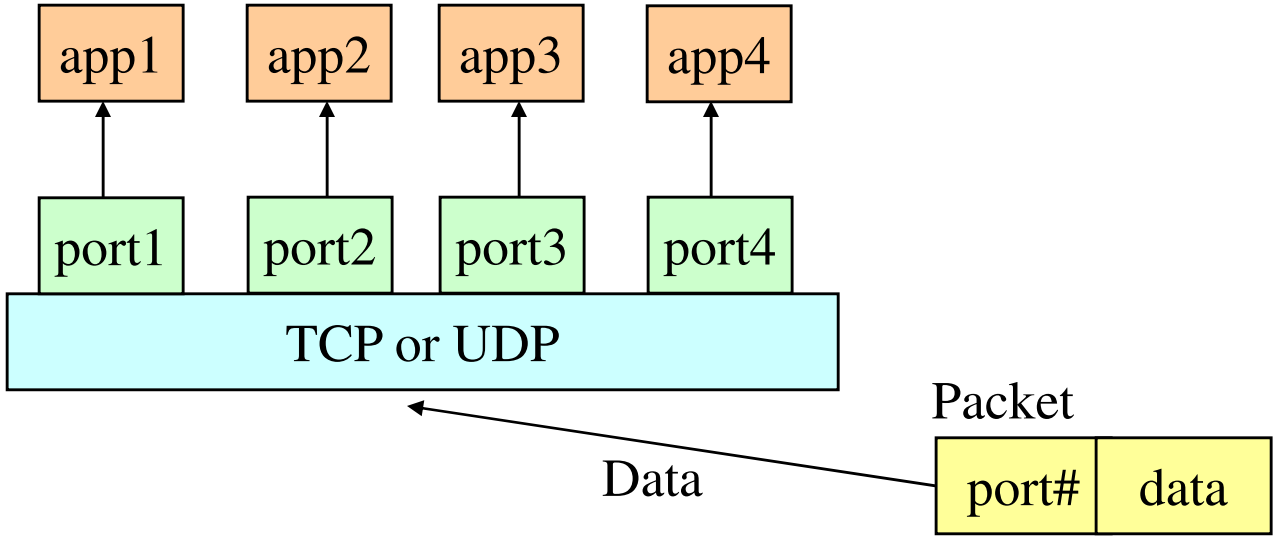
Qu'est-ce qu'une socket?

Une socket est une abstraction à travers laquelle une application peut envoyer et recevoir des données, de la même manière qu'un gestionnaire de fichier (file handler) permet à une application de lire et écrire des données dans un fichier sur disque.



Qu'est-ce qu'une socket?

- Les numéros de *ports* sont utilisés par les protocoles TCP et UDP pour identifier le programme de destination (l'application) des données entrantes



Qu'est-ce qu'une socket?

- **Les numéros de port compris entre 0 et 1023 sont réservés (ils sont utilisés par des services bien connus)**
  - FTP: 21/tcp, 21/udp
  - HTTP: 80/tcp, 80/udp
  - HTTPS: 443/tcp, 443/udp
  - SNMP: 161/tcp, 161/udp

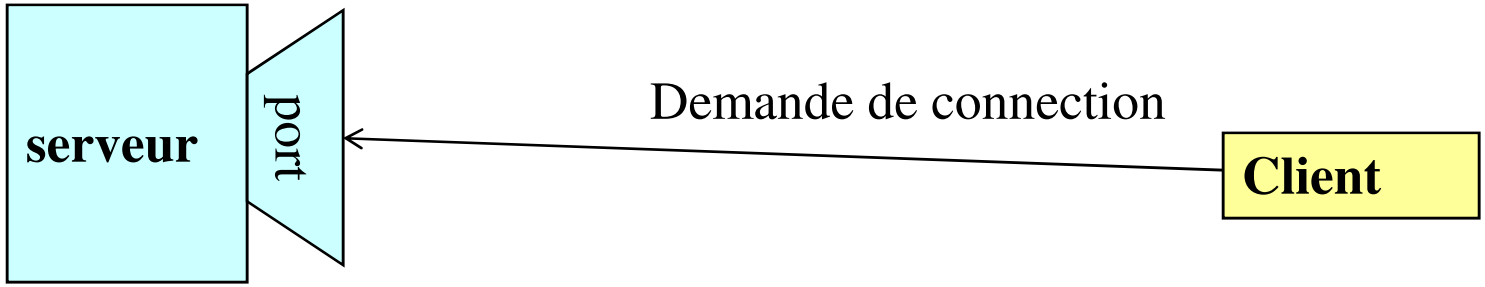
* Gérés par l'Internet Assigned Numbers Authority (IANA)

- **Lorsque vous choisissez un numéro de port pour votre serveur, sélectionnez en un qui est supérieur à 1023**

Communication par socket

- **Un serveur (application) s'exécute sur un ordinateur spécifique et a une socket qui est lié à un port spécifique.**

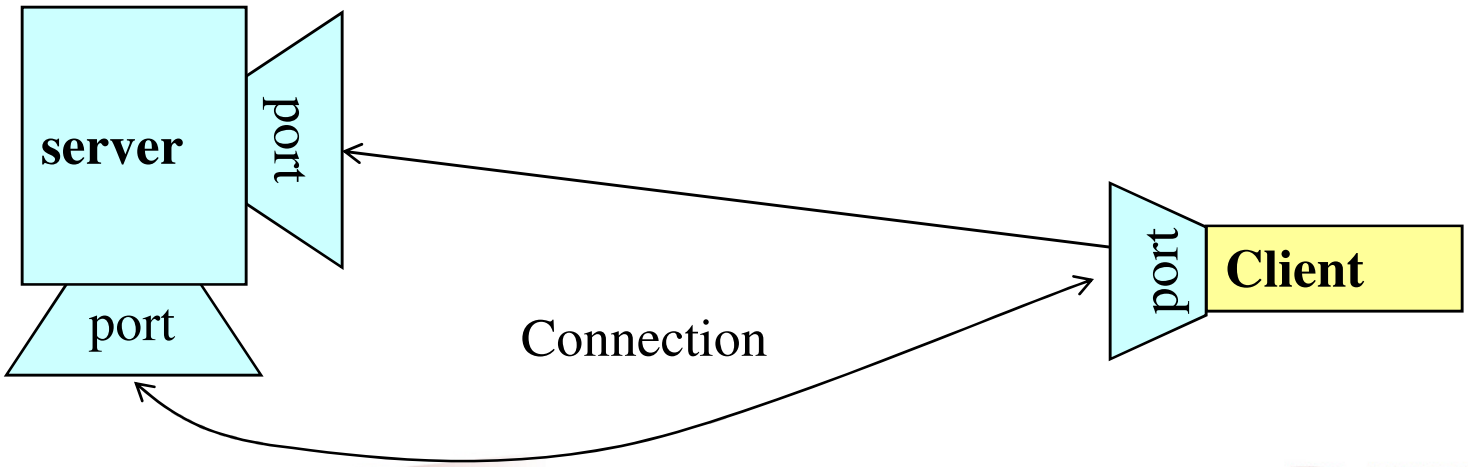
- **Le serveur écoute sur la socket et attend qu'un client fasse une demande de connexion.**



Communication par socket

- **Le serveur accepte la demande de connexion entrante.**

- **Le serveur crée une nouvelle socket liée a un numéro de port différent de celui sur lequel il a reçu la demande.**



Communication par socket

- **Deux principaux protocoles de communication peuvent être utilisés pour la programmation des sockets**
  - UDP
    - Communication par datagramme
    - datagram sockets
  - TCP
    - Communication par flux (de données) continu (ou stream)
    - stream sockets

Communication par socket

- **Communication par datagramme**
  - UDP est un protocole sans connexion (connectionless)
    - Pour chaque datagramme, nous devons envoyer le descripteur de la socket locale et l'adresse de la socket réceptrice
    - Il ya une limite de taille de 65,500 octets sur chaque datagramme
    - Aucune garantie que les datagrammes envoyés seront reçus dans le même ordre

==> UDP est souvent utilisé pour l'implémentation des applications client/serveur dans un réseau local

Communication par socket

- Communication par datagramme
  - Structure de l'entête UDP

0 … 15	16 … 31
Source Port Number(16 bits)	Destination Port Number(16 bits)
Length(UDP Header + Data)16 bits	UDP Checksum(16 bits)
Application Data (Message)	

Entête IP

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version				IHL				Differentiated Services								Total length															
Identification																Flags			Fragment offset												
TTL								Protocol								Header checksum															
Source IP address																															
Destination IP address																															
Options and padding :::																															

Communication par socket

- **Communication par flux continu**
  - TCP est un protocole orienté connexion
    - Une connexion doit d'abord être établie entre le client et le serveur.
    - Aucune limite sur les données à envoyer
    - Les paquets envoyés sont reçus dans l'ordre dans lequel ils ont été envoyés

==> TCP est utile pour la mise en place de services réseaux tel que la connexion à distance (rlogin, telnet) et le transfert de fichiers (FTP)

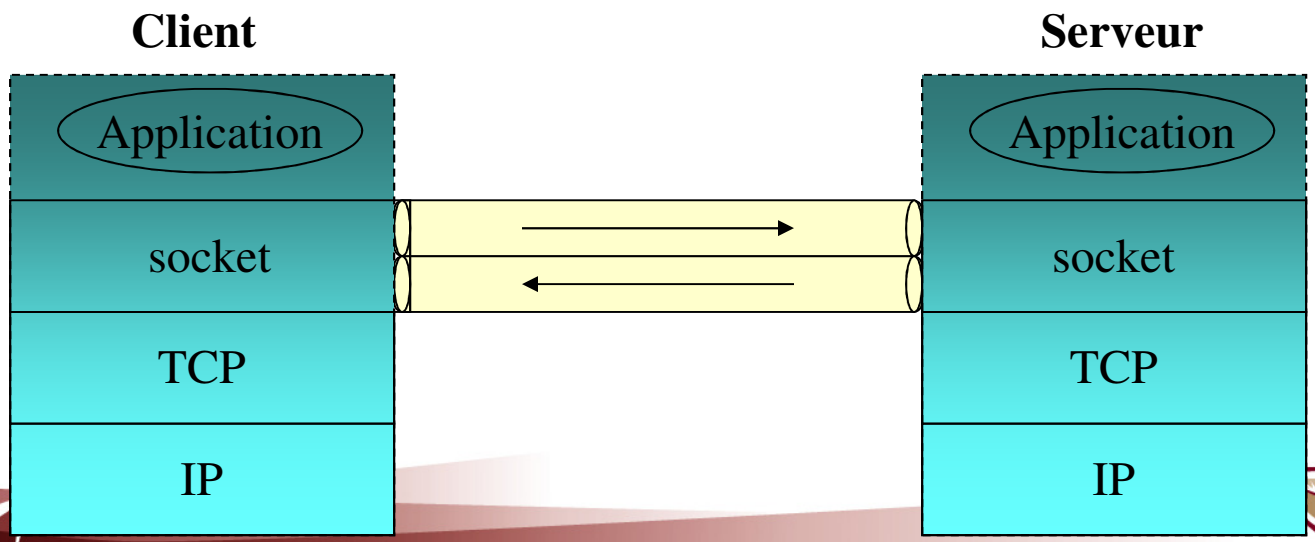
Implémentation du client et du serveur

- Les principales classes
- Implémentation d'un client/serveur TCP
- Implémentation d'un client/serveur UDP

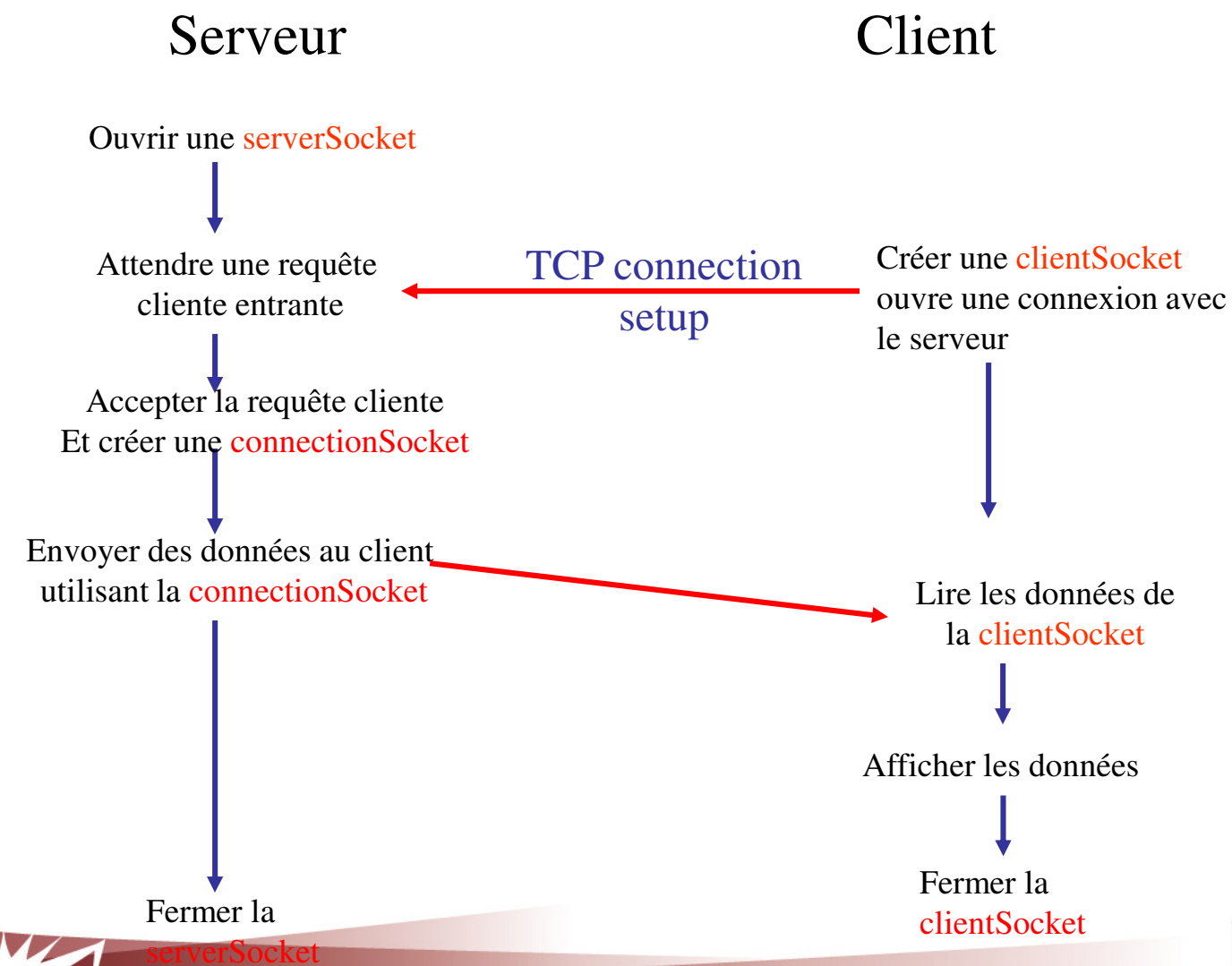
Les principales classes

- Le package java.net fourni trois classes principales:
  - Socket – pour l'implémentation d'un client TCP
  - ServerSocket – pour l'implémentation d'un serveur TCP
  - DatagramSocket – pour l'implémentation à la fois d'un client et d'un serveur UDP

- L'échange de données
  - TCP: InputStream and OutputStream
  - UDP: DatagramPacket



Implémentation d'un client/serveur TCP



Implémentation d'un serveur TCP

Ouvrir une socket serveur

```
ServerSocket server;
    try {
        server = new ServerSocket(portNumber);
    } catch (IOException e) { System.out.println(e); }
```

Créer un objet pour la socket de connexion

```
try {
    Socket connectSocket =
    server.accept();
} catch (IOException e) { .. }
```

Ecrire des données dans la socket de connexion

```
String data = "Hello from server";
try {
    OutputStream out = connectSocket.getOutputStream();
    out.write(data.getBytes() );
} catch (IOException e) {…}
```

Implémentation d'un serveur TCP

Fermer les sockets

```
try {
        connectSocket.close(); // Close the socket. We
    are
                                    // done with this client!
        server.close();   // close the server socket
} catch (IOException e) { System.out.println(e); }
```

Implémentation d'un client TCP

Ouvrir une socket client

```
try {
    Socket  clientSocket = new Socket(serverIP/Name, serverPort);
} catch (IOException e) { System.out.println(e); }
```

Lire des données de la socket

```
int  MAXLENGTH= 256;
byte[ ] buff = new byte[MAXLENGTH];
try {
        InputStream in = clientSocket.getInputStream();
        in.read(buff);
    } catch (IOException e) {System.out.println(e); }
```

Fermer la socket

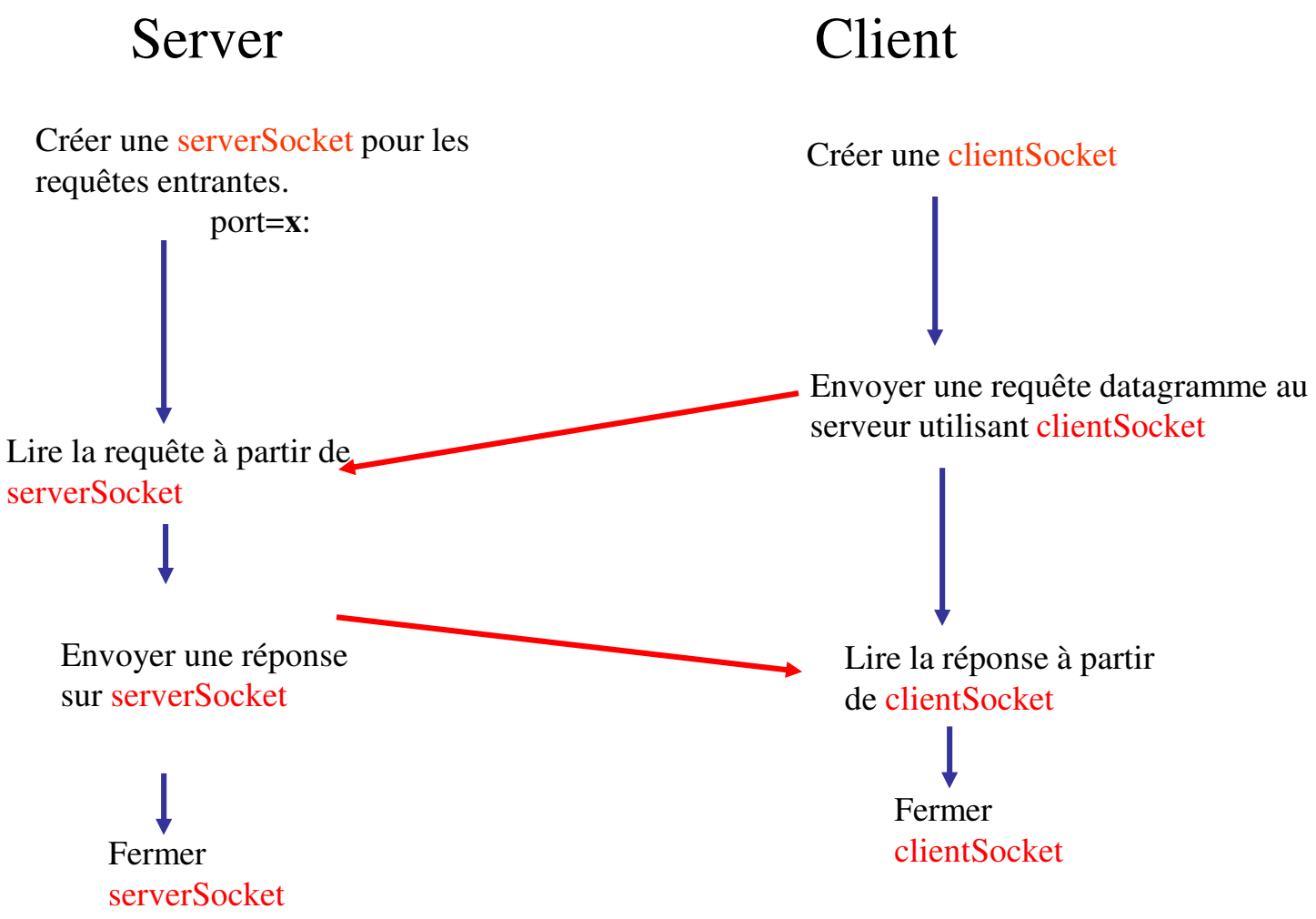
```
try {
 clientSocket.close(); // Close the socket and its streams
 } catch (IOException e) {…}
```

Implémentation d'un serveur/client TCP

- **OutputStream:**
  - abstract void write(int data)
  - void write(byte[ ] data)
  - void write(byte[ ] data, int offset, int length)
  - void flush()
  - void close()

- **InputStream:**
  - abstract int read()
  - int read(byte[ ] data)
  - int read(byte[ ] data, int offset, int length)
  - int available()
  - void close()

Implémentation d'un client/server UDP



Implémentation d'un client UDP

Créer une clientSocket

```
try {
   DatagramSocket clientSocket = new DatagramSocket();
} catch (IOException e) { System.out.println(e); }
```

Envoyer un datagramme au serveur

```
int PACKETLENGTH= 256;
byte[ ] data = new byte[PACKETLENGTH];
try {
      DatagramPacket packet = new DatagramPacket(data,
         data.length, serverIP, serverPort);
      clientSocket.send(packet);
} catch (IOException e) {System.out.println(e); }
```

Implémentation d'un client UDP

Lire la réponse du serveur

```
byte[ ] rcvData = new byte[PACKETLENGTH];
try {
    DatagramPacket receivePacket =
        new DatagramPacket (rcvData,
    rcvData.length);
    clientSocket.receive(rcvPacket);

    String rcvString = new String(rcvPacket.getData());
    System.out.println("The received packet is: "+rcvString);
} catch (IOException e) {System.out.println(e); }
```

Fermer la socket

```
try {
    clientSocket.close(); // Close the socket
} catch (IOException e) {…}
```

Implémentation d'un serveur UDP

Ouvrir une socket

```
DatagramSocket server;
    try {
        serverSocket = new DatagramSocket (portNumber);
    } catch (IOException e) { System.out.println(e); }
```

Recevoir un datagramme du client

```
byte[ ] buff = new byte[PACKETLENGTH];
try {
        DatagramPacket rcvPacket = new DatagramPacket
                (buff, buff.length);
        server.receive(rcvPacket );
} catch (IOException e) { .. }
```

Implémentation d'un serveur UDP

Obtenir l'adresse IP et le numéro du port du client

```
InetAddress clientIP = rcvPacket.getAddress();
int clientPort = rcvPacket.getPort();
```

Envoyer un datagramme au client

```
String data = "Hello from server";
try {
        DatagramPacket sendPacket = new DatagramPacket
                (sendData, sendData.length,  clientIP, clientPort );
        serverSocket.send(sendPacket);
  } catch (IOException e) {…}
```

Autres classes

- **DatagramSocket:**
  - DatagramSocket()
  - DatagramSocket(int localPort)
  - DatagramSocket(int localPort, InetAddress localAddr)

- **DatagramPacket: Creation**
  - DatagramPacket(byte[ ] data, int length)
  - DatagramPacket(byte[ ] data, int offset, int length)
  - DatagramPacket(byte[ ] data, int length, InetAddress remoteAddr, int remotePort)
  - DatagramPacket(byte[ ] data, int offset, int length, InetAddress remoteAddr, int remotePort)
  - DatagramPacket(byte[ ] data, int length, SocketAddress sockAddr)
  - DatagramPacket(byte[ ] data, int offset, int length, SocketAddress sockAddr)

Sockets UDP vs. sockets TCP

- **Sockets UDP : Analogue a une communication par mail**
  - Pas de connexion requise
  - L'adresse de destination doit être spécifiée pour chaque message a envoyer
  - Il y'a une limite sur les données (datagramme) à envoyer
  - Aucune garantie sur l'ordre de réception des messages
  - Les messages peuvent

- **Sockets TCP : Analogue a une communication téléphonique**
  - Une connexion est requise
  - L'adresse de destination est spécifiée au moment de la création de la connexion
  - Aucune limite sur les données à envoyer
  - Les paquets envoyés sont reçus dans l'ordre dans lequel ils ont été envoyés
  - La même socket peut être utiliser pour

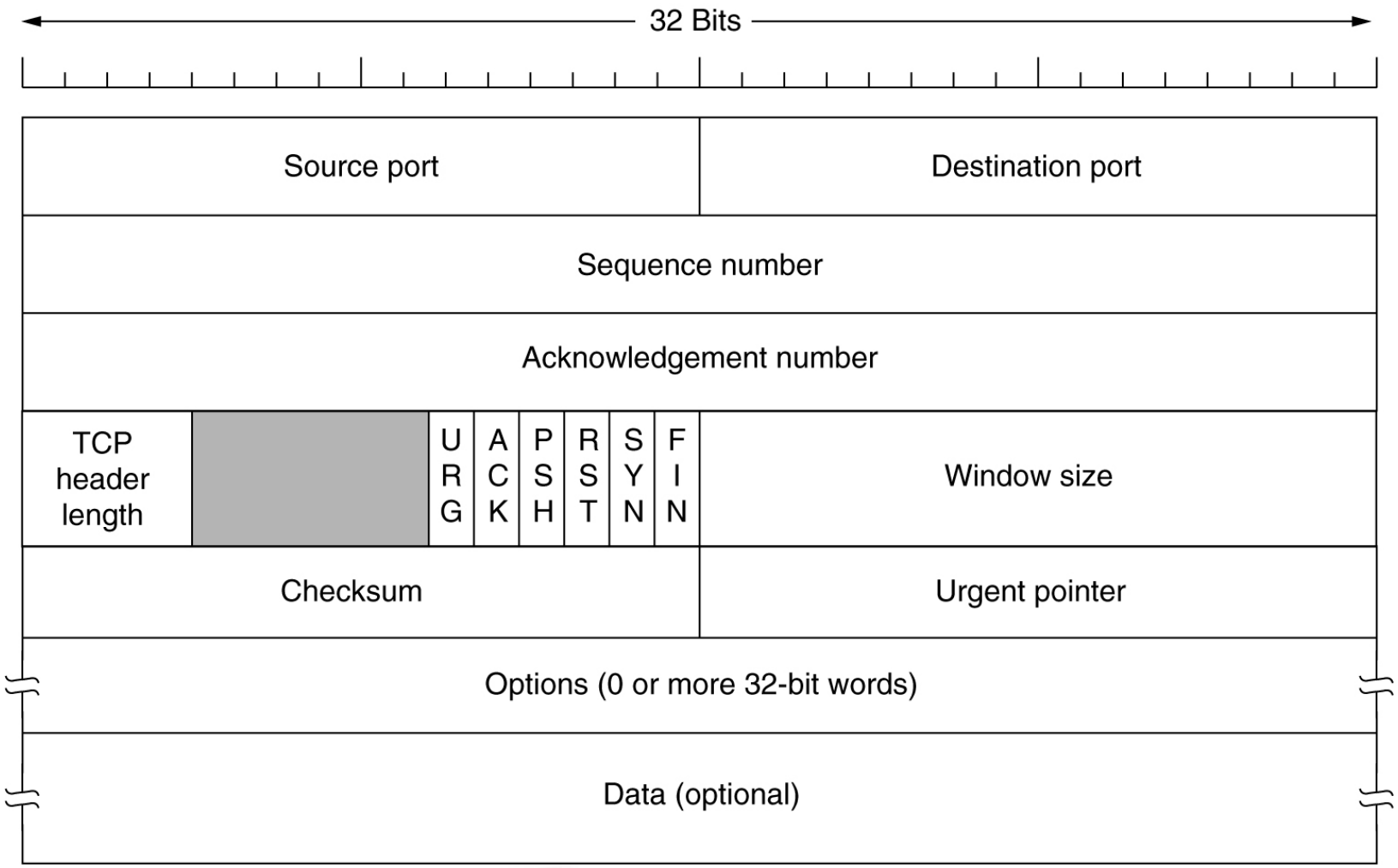
Envoi et réception de données

- **Protocole de Communication**
  - Codage et décodage des message
  - Cadrage (framing)
- **Exemple**

Envoi et réception de données

- **L'expéditeur et le destinataire doivent s'entendre sur le protocole de communication**
  - Comment les informations échangées seront codées (représenté sous forme de séquence de bits)
    - Comment la séquence de bits est organisé par l'expéditeur et interprété ou analysé par le récepteur?
    - Framing: fait reference au fait de permettre au récepteur de localiser le début et la fin d'un message.
  - Quel program envoie quel information et quant?
  - Comment les informations reçues affectent le comportement de l'application réceptrice?

L'en-tête d'un packet TCP

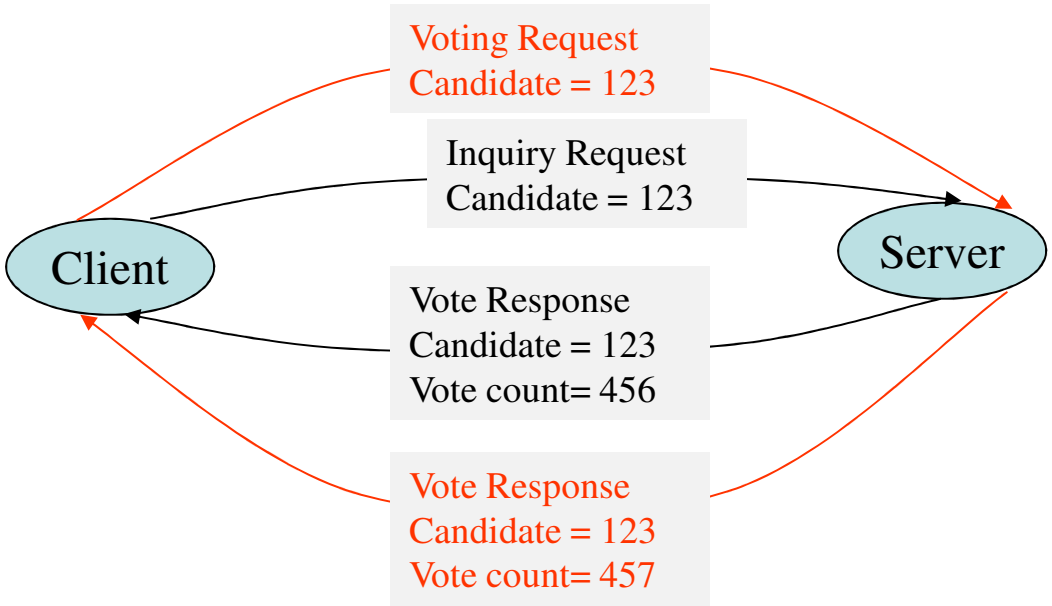


Envoi et réception de données

- **Framing:**
  - Problèmes possibles:
    - *Deadlock*
    - Erreurs de protocole
  - Deux types de techniques peuvent permettre à un récepteur de déterminer la fin d'un message sans ambiguïté:
    - *Utilisant un délimiteur*
      - E.g. un indicateur de fin de flux de données, un caractère particulier
      - Et si le message contient déjà le délimiteur?
    - *Longueur explicite*
      - *Plus simple, mais on a besoin de savoir la limite supérieure de la taille du message*
- **Les mêmes considérations s'appliquent à trouver les limites des différents *champs* d'un message donné**

Envoi et réception de données

- **Exemple**
  - **Considérons le protocole de vote ci-après**



Envoi et réception de données

1. **Représentation du message :**

```
public class VoteMsg {
        private boolean isInquiry; // true if inquiry; false if vote
        private boolean isResponse;// true if response from server
        private int candidateID; // in [0,1000]
        private long voteCount; // nonzero only in response
        …..
}
```

2. **Codage et décodage du message**

Magic-string	Type [‘v’,’i’]	RespFlag [‘R’]	Candidate ID	Vote Count

Envoi et réception de données

```
public class VoteMsgEncoder implements VoteMsgCoder {
    public static final String MAGIC = "Voting";
    public static final String VOTESTR = "v";
    public static final String INQSTR = "i";
    public static final String RESPONSESTR = "R";

    public static final String FIELDELIMSTR = " ";
    public static final int MAX_MSG_LENGTH = 2000;

    public byte[ ] encode(VoteMsg msg) throws IOException {
    String msgString = MAGIC + FIELDELIMSTR +
        (msg.isInquiry() ? INQSTR : VOTESTR)
        + FIELDELIMSTR + (msg.isResponse() ? RESPONSESTR +
```

Envoi et réception de données

```
public class VoteMsgDecoder implements VoteMsgCoder {
    boolean isInquiry;
        boolean isResponse;
        int candidateID;
        long voteCount;
    public VoteMsg decode(byte[ ] message) throws IOException
    {
      ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
       Scanner s = new Scanner(new InputStreamReader(msgStream));
       String token;
                try {
                 token = s.next();
                  if (!token.equals(MAGIC)) {
                     throw new IOException("Bad magic string: " + token); }
                 token = s.next();
                  if (token.equals(VOTESTR)) {
                      isInquiry = false;
                  } else if (!token.equals(INQSTR)) {
                        throw new IOException("Bad vote/inq indicator: " + token);
                     } else {
                              isInquiry = true; }
```

Vérifier si le message commence par le mot magique

Vérifier si c'est un message de vote

Magic-string	Type ['v','i']	RespFlag ['R']	Candidate ID	Vote Count

Envoi et réception de données

Vérifier si c'est une réponse

Extraire l'information du message

Créer et retourner un message de vote

```
token = s.next();
if (token.equals(RESPONSESTR)) {
        isResponse = true;
        token = s.next();
} else {
        isResponse = false;
}
candidateID = Integer.parseInt(token);
if (isResponse) {
        token = s.next();
        voteCount = Long.parseLong(token);
} else {
        voteCount = 0;
}
} catch (IOException ioe) {
  throw new IOException("Parse error...");
}
return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
}

}
```

Magic-string	Type ['v','i']	RespFlag ['R']	Candidate ID	Vote Count

Envoi et réception de données

3. Framing

Définir une classe de cadrage (framing) du message. Cette classe implémente un cadrage utilisant le caractère spéciale de "nouvelle ligne" ("\ n") comme délimiteur

```
public class DelimFramer {
        private InputStream in; // data source
        private static final byte DELIMITER = "\n";
                                        // message delimiter

        public void frameMsg(byte[] message, OutputStream out) throws IOException {

            for (byte b : message) {
            if (b == DELIMITER) {
                throw new IOException("Message contains delimiter"); }
            }
            out.write(message);
            out.write(DELIMITER);
            out.flush();
    }
```

S'assurer que le message ne contient pas le caractère délimiteur

Envoi et réception de données

Regarder les octets jusqu'à ce que le délimiteur est trouvé

Ajouter l'octet au buffer

Retourner le message

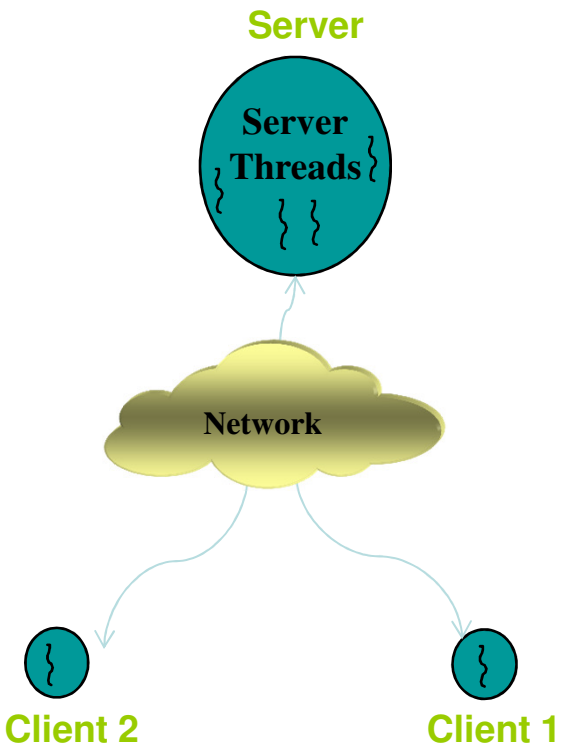
```
public byte[] readNextMsg() throws IOException {
  ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
   int nextByte;

   while ((nextByte = in.read()) != DELIMITER) {
       if (nextByte == -1) { // end of stream?
              if (messageBuffer.size() == 0) { // if no   byte read
                          return null;
          } else { // if bytes followed by end of stream:  framing error
             throw new EOFException("Non-empty message without
                                                 delimiter");}
       }
   messageBuffer.write(nextByte);
   }
   return messageBuffer.toByteArray();
   ……..
}
```

Autres fonctions

- **Codage de valeurs simples**
  - *ByteArrayOutputStream* buf = new ByteArrayOutputStream();
  - *DataOutputStream* out = new DataOutputStream(buf);
  - out.writeByte(byteVal);
  - out.writeShort(shortVal);
  - out.writeInt(intVal);
  - out.writeLong(longVal);
  - out.flush();
  - byte[] msg = buf.toByteArray();
  - ….
- **Décodage de de valeurs simples**
  - ByteArrayInputStream
  - DataInputStream in;
  - in.readByte()
  - In.readdUnsignedByte()
  - In.readFloat()
  - In.readShort()
  - In.readUnsignedShort()
  - In.readLong()
  - …

Programmation avancée des sockets



- Multithreading
- Blocage et Timeouts
- Destinataires multiple

Multithreading

- Les serveurs itératifs traitent les demandes des clients de manière séquentielle, en terminant avec un client avant de passer a l'autre.
  - Le temps d'attente pour les clients suivants peut être inacceptable
  - Marche mieux pour les applications où le temps de traitement de chaque client est faible
- Le multithreading permet au serveur de gérer les clients en parallèle

Multithreading

- **Java fournit deux méthodes pour exécuter une tâche dans un nouveau thread:**

1) La définition d'une sous-classe de la classe ***Thread***

2) La définition d'une classe qui implémente l'interface ***Runnable***, et en passant une instance de cette classe au constructeur Thread.

```
public class ThreadExample implements Runnable {
        ......
        void run(){ .....}
}
new Thread(new ThreadExample()).start();
```

Multithreading

- **Exemple de thread**

```
public class OneClientHandler implements Runnable {
    private Socket clntSock; // Socket connect to client
    public OneClientHandler(Socket clntSock) {
                this.clntSock = clntSock;
    }
    public static void handleClient(Socket clntSock) {
      try {
            // Get the input and output I/O streams from socket
            InputStream in = clntSock.getInputStream();
            OutputStream out = clntSock.getOutputStream();
            .........
              clntSock.close();
        } catch (IOException e) { }
      }
   public void run() {
          handleClient(clntSock);
      }
}
```

Multithreading

- **Example de serveur multithreading**

```
public class MultithreadingServer {
    public static void main(String[] args) throws IOException {
                ...
     int echoServPort = Integer.parseInt(args[0]); // Server port
     ServerSocket servSock = new ServerSocket(echoServPort);

    // Run forever, accepting and spawning a new client thread for each connection
    while (true) {
                Socket clntSock = servSock.accept();
                // Spawn thread to handle new connection
                Thread thread = new Thread(new
                                        OneClientHandler(clntSock));
                thread.start();
    }
  }
}
```

Blocage et Timeouts

- Des méthodes de socket peuvent être bloquantes
  - La méthode accept() de *ServerSocket* bloque jusqu'à ce que une connexion est établie
  - Le constructeur de *Socket* bloque jusqu'à ce qu'une connexion est établie
  - Les méthodes read() et receive() bloquent si il n'ya pas de données disponibles
  - La méthode write() bloque si il n'a pas assez d'espace dans la mémoire tampon (buffer) de sortie
- Un appel de méthode bloquée rend le thread qui l'exécute inutiles
  - E.g. En attente de datagrammes perdus

Blocage et Timeouts

- Comment contourner les appels bloquants?
  - Définir une limite supérieure pour le temps de blocage
    - Marche pour accept(), read() et receive()

```
try{
        sock.setSoTimeout(timeBoundMillis);
        //serverSocket.setSoTimeout(timeBoundMillis);
        //datagramSocket.setSoTimeout(timeBoundMillis);

} catch (InterruptedIOException ex) { //blocking timeout is reached   }
```

  - Utiliser la méthode available()
    - Vérifiez la disponibilité des données avant d'appeler read ()

```
InputStream in = clntSock.getInputStream();
if (in.available()){
        in.read(…);}
```

Blocage et Timeouts

- Connecter une socket

```
Try{
        InetAddress addr = InetAddress.getByName("java.sun.com");
        int port = 80;
        SocketAddress sockaddr = new InetSocketAddress(addr, port);
        //Create an unbound socket
        Socket sock = new Socket();
        int timeoutMillis = 2000; // 2 seconds
        sock.connect(sockaddr, timeoutMillis);
}catch (SocketTimeoutException ex){….}
```

- Ecrire dans une socket
  - La quantité de temps que write () peut bloquer est contrôlée par l'application réceptrice
  - Actuellement, Java ne fournit pas de moyen de provoquer le timeout de la méthode write ()

Destinataires multiple

- Les informations fournies par le serveur peuvent être d'intérêt à plusieurs destinataires
  - Envoyer une copie des données à chaque destinataire via Unicast
    - Inefficace (gaspille la bande passante)
    - E.g,
      - Le serveur envoi des flux à 1Mbps
      - La connexion réseau est à 3Mbps
      - Seulement 3 clients simultanés peuvent être supportés
  - Les réseaux offrent un moyen d'utiliser la bande passante plus efficacement
    - Les paquets sont dupliqués par le réseau (et non par l'application) seulement quant c'est nécessaire
    - 2 manière:
      - Broadcast
      - Multicast

Destinataires multiple

- Radiodiffusion (Broadcast)
  - La radiodiffusion de datagrammes UDP est similaire à leur monodiffusion (unicast), sauf que l'adresse de *diffusion* est utilisée au lieu d'une adresse IP régulière (unicast)
    - IPv4: 255.255.255.255
    - IPv6: FFO2::1
  - Tous les hôtes sur le même réseau (local) de diffusion reçoivent une copie du message.

Destinataires multiple

- Multidifusion (multicast)
  - Une adresse multicast identifie un ensemble de destinataires
    - IPv4: adresses entre 224.0.0.0 et 239.255.255.255
    - IPv6: n'importe quelle adresse commençant par FF

Destinataires multiple

- Exemple de multicast

Expéditeur d'un message multicast

```
import java.net.MulticastSocket;
public class MulticastSender {
  public void sendMulticastMessage(String msg) {
      try{
            MulticastSocket mSocket = new MulticastSocket();
            mSocket.setTimeToLive(TTL); // Set TTL for all datagrams
            ….
            DatagramPacket message = new DatagramPacket(msg, msg.length,
                                                        multicastDestAddr, destPort);
            mSocket.send(message);
            mSocket.close();
      }catch (IOException ex){….}
} }
```

Destinataires multiple

- Example de multicast

Destinataire d'un message multicast

```
Try{
        MulticastSocket  mSock = new MulticastSocket(port); // for receiving
        mSock.joinGroup(multicastAddress); // Join the multicast group

        // Receive a datagram
        DatagramPacket packet = new DatagramPacket(new
        byte[MAX_MSG_LENGTH],
        VoteMsgTextCoder.MAX_WIRE_LENGTH);

        sock.receive(packet);

        sock.close();
}
```

Broadcast ou Multicast?

- **Broadcast**
  - Portée limitée au réseau local
  - Tous les hôtes (sur le réseau local) peuvent recevoir les message broadcast par défaut
- **Multicast**
  - Peut inclure des récepteurs n'importe où sur le réseau
  - Les récepteurs doivent connaître (et joindre) l'adresse d'un groupe de multicast

- **Références**
  - TCP/IP Sockets in Java: Practical Guide for Programmers, Second Edition, Kenneth L. Calvert and Michael J. Donahoo, ISBN: 978-0-12-374255-1
  - "All About Sockets" http://java.sun.com/docs/books/tutorial/networking/sockets/

Telecommunication Services Engineering (TSE) Lab

UNIVERSITÉ
Concordia
UNIVERSITY

www.ciise.concordia.ca