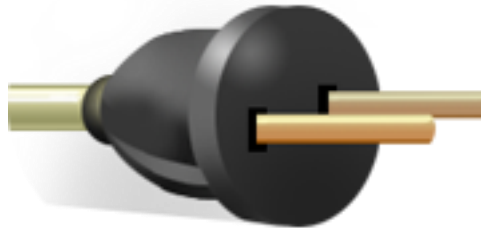


COURS - MRI 4 : Les sockets

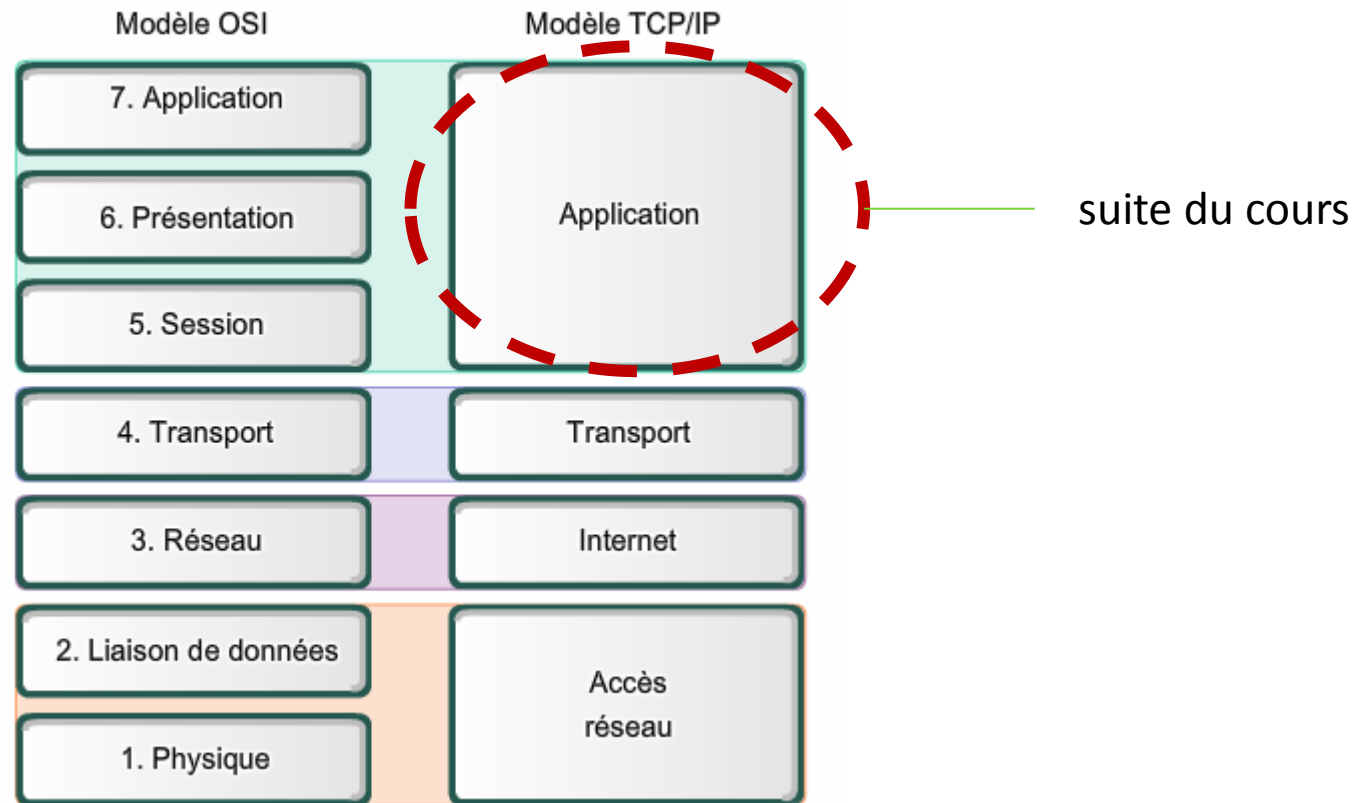
yoann.maurel@irisa.fr





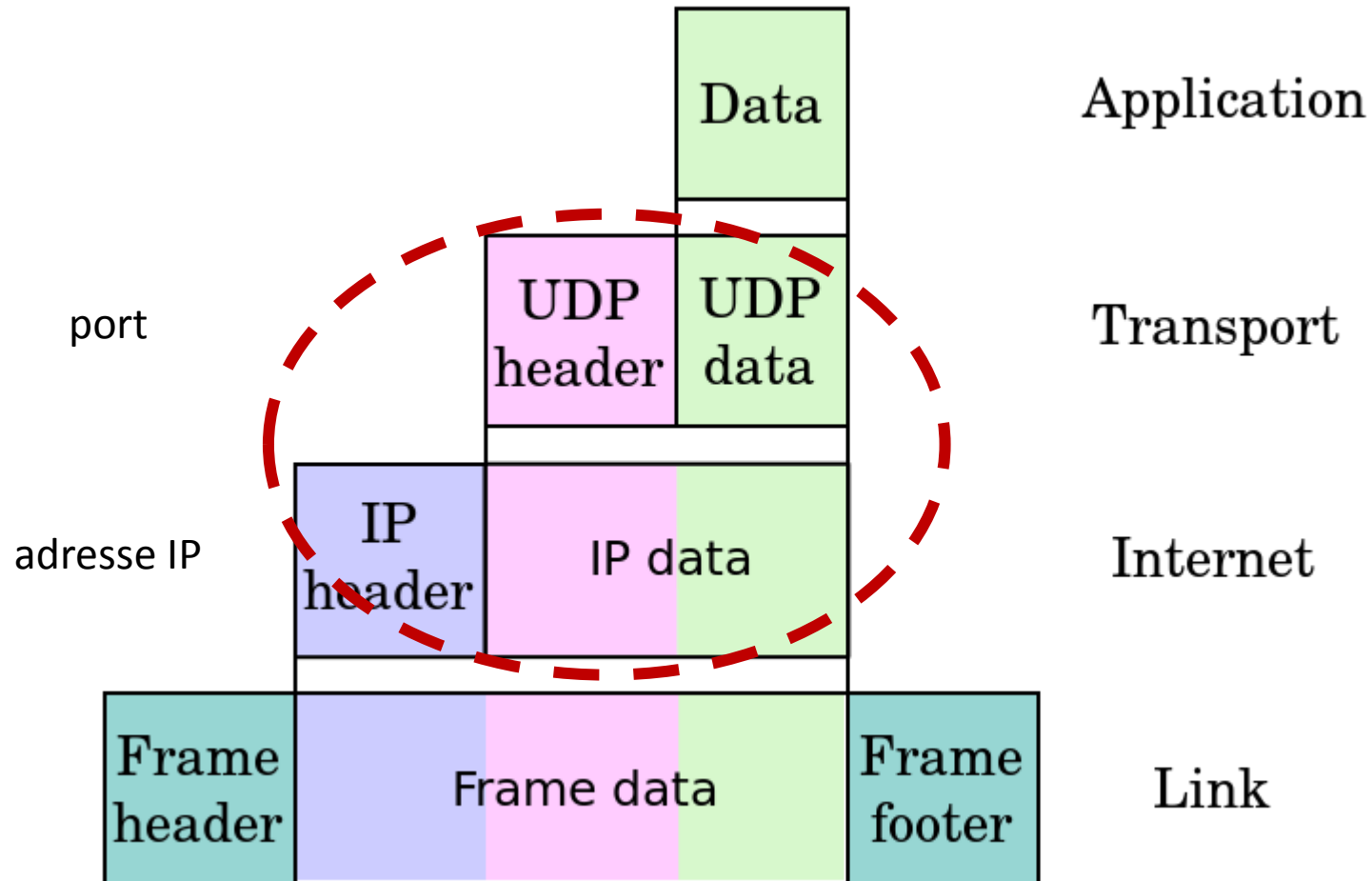
Rappels

Rappel - Les couches



source wikipedia

Rappel : Encapsulation

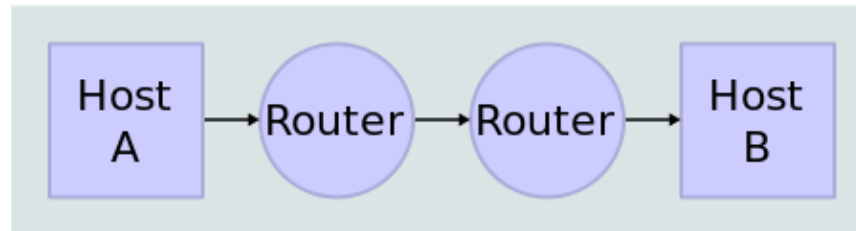


Exemple pour le protocole UDP

source wikipedia

Rappel - Les couches

Network Topology

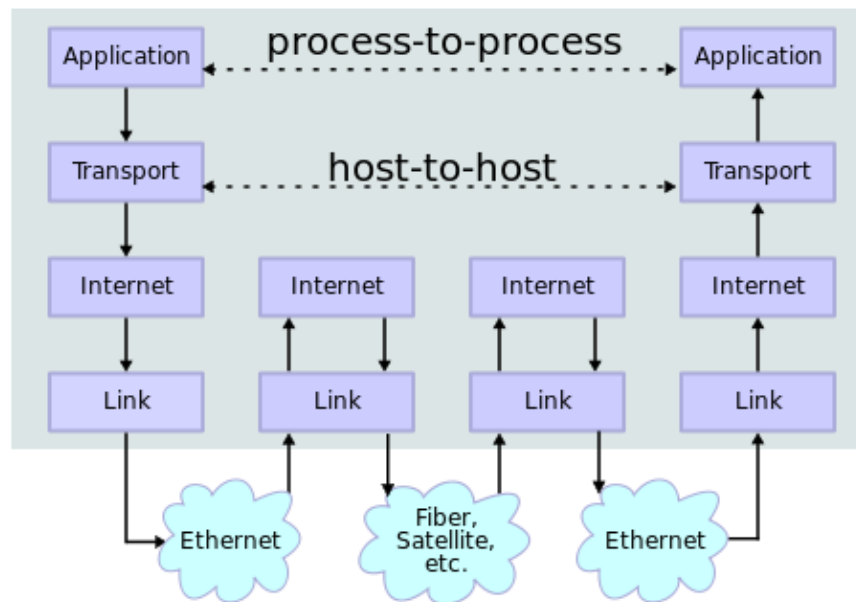


@ip:port

88.77.66.23:6400

173.194.23.43:80

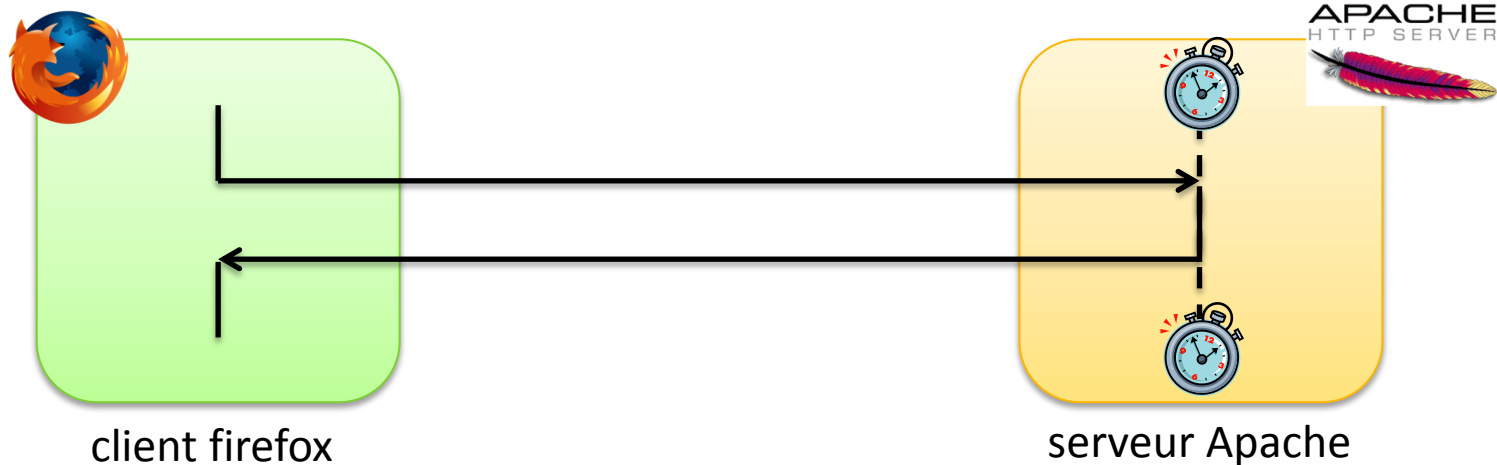
Data Flow



source wikipedia

Rappel : Client/Serveur

- ▶ Le serveur, il fournit un service
 - ▶ passif, il attend les connexions sur un port donnée
 - ▶ "serveur" désigne l'ordinateur ou l'application/processus
- ▶ Le ou les client utilisent des services
 - ▶ actifs, il prennent l'initiative de la connexion
 - ▶ "client" désigne l'ordinateur ou l'application/processus



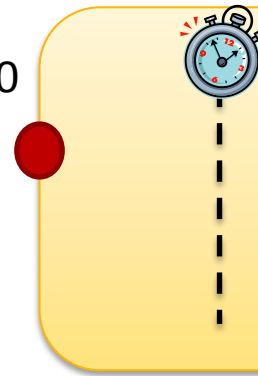
Rappel : Client/Serveur

- Le serveur écoute sur un port donné



client

80

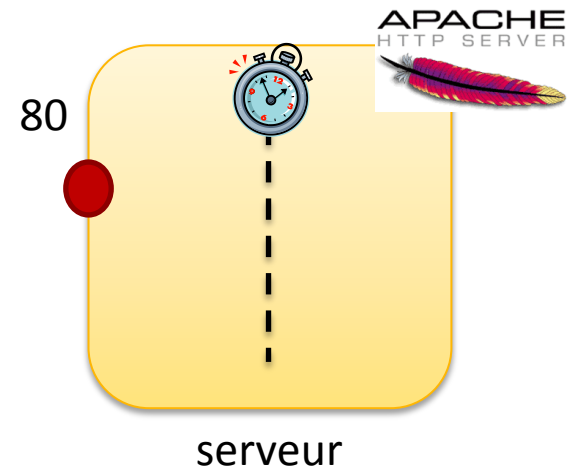
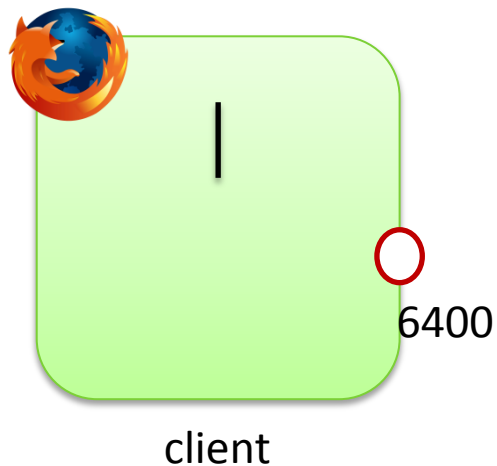


serveur



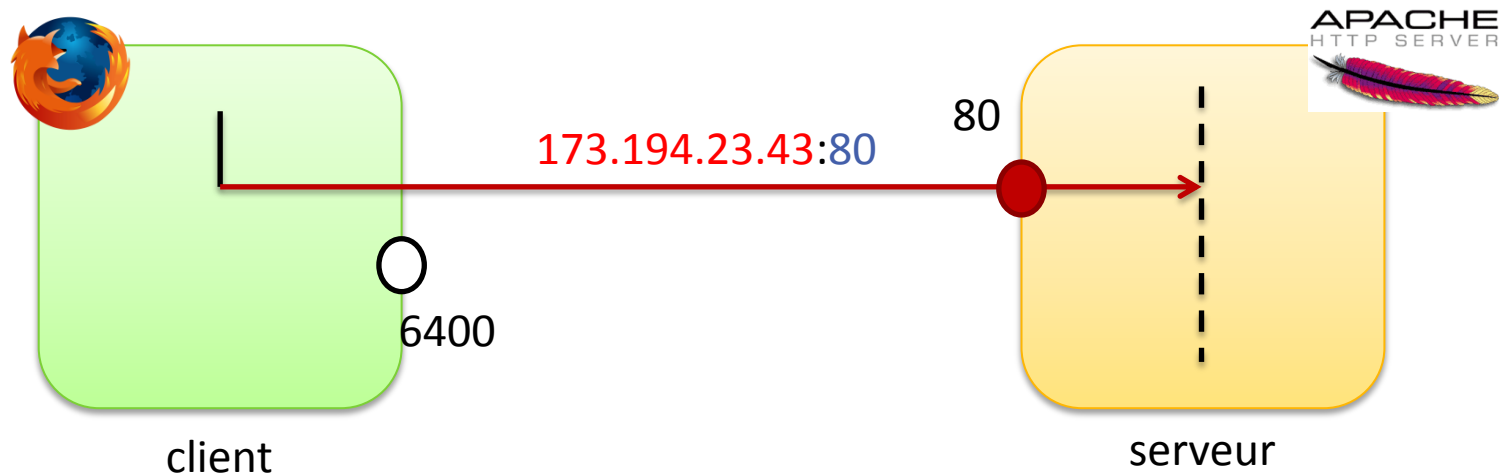
Rappel : Client/Serveur

- ▶ Le serveur écoute sur un port donné
- ▶ Le client choisit un port sur lequel recevoir la réponse



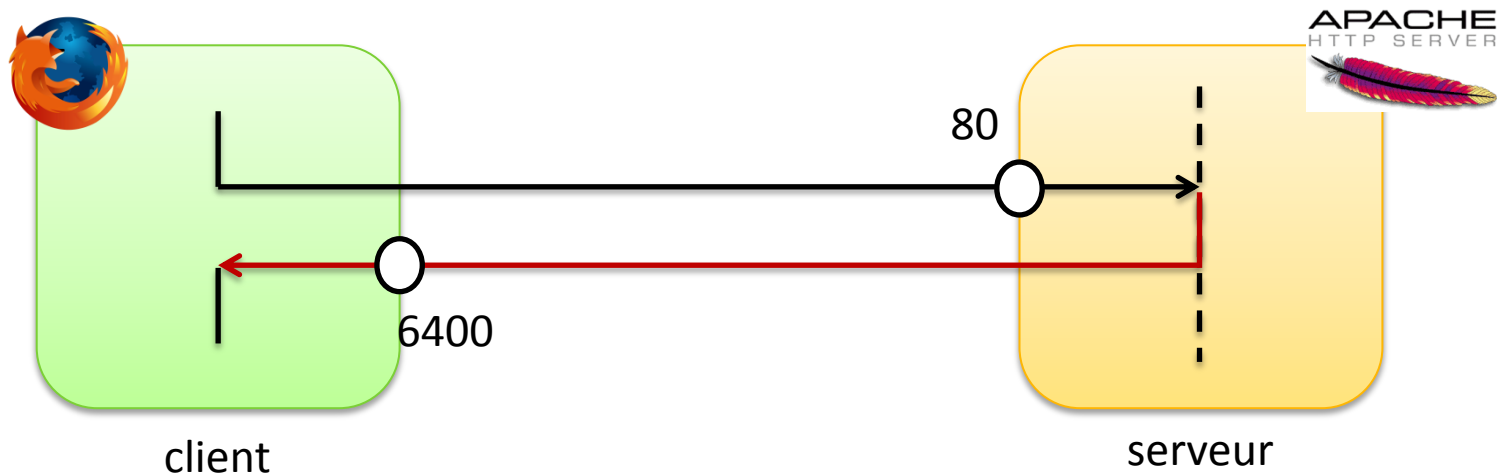
Rappel : Client/Serveur

- ▶ Le serveur écoute sur un port donné
- ▶ Le client choisit un port sur lequel recevoir la réponse
- ▶ Le client se connecte au serveur
 - ▶ utilise @IP+port qu'il connaît à l'avance

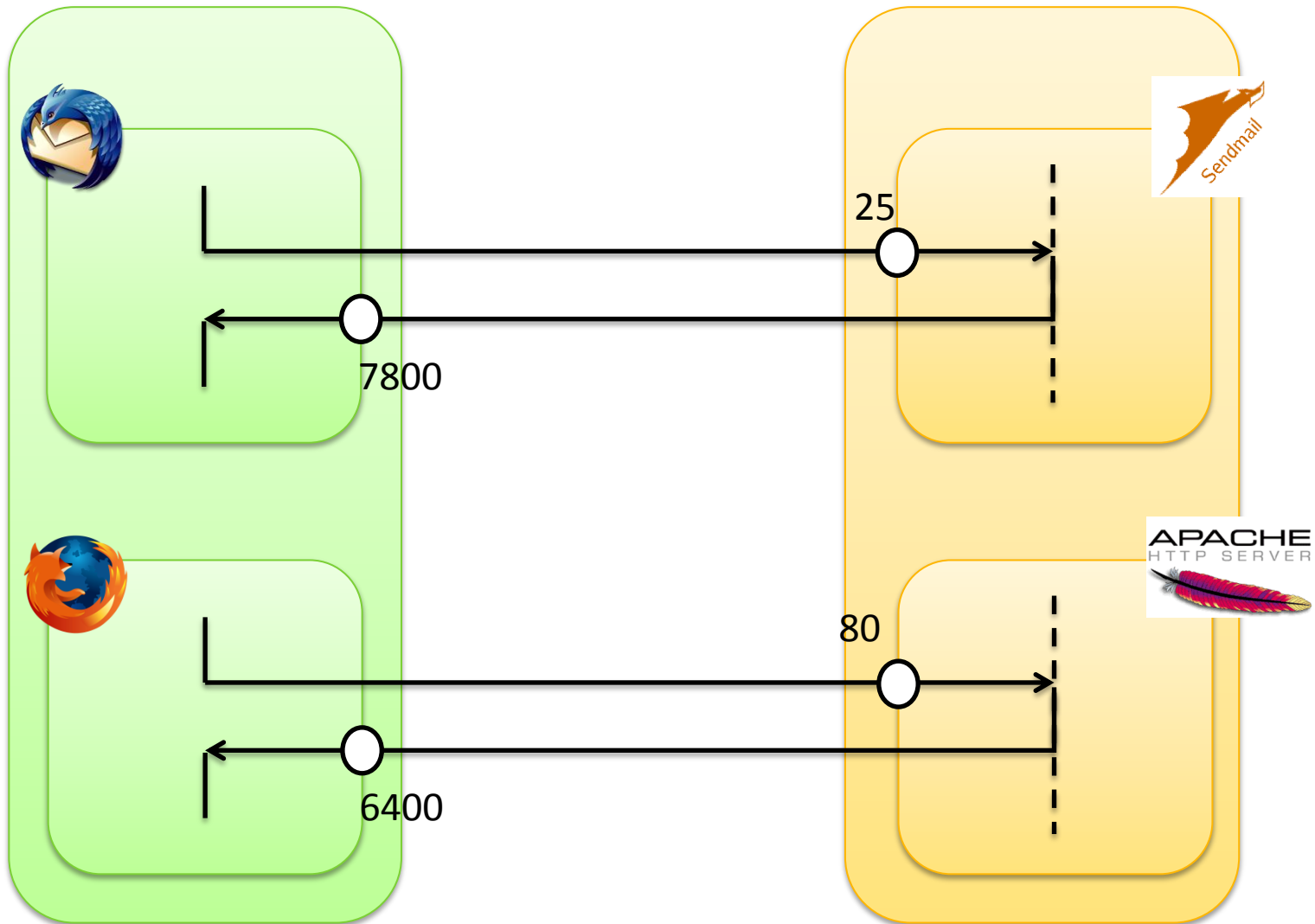


Rappel : Client/Serveur

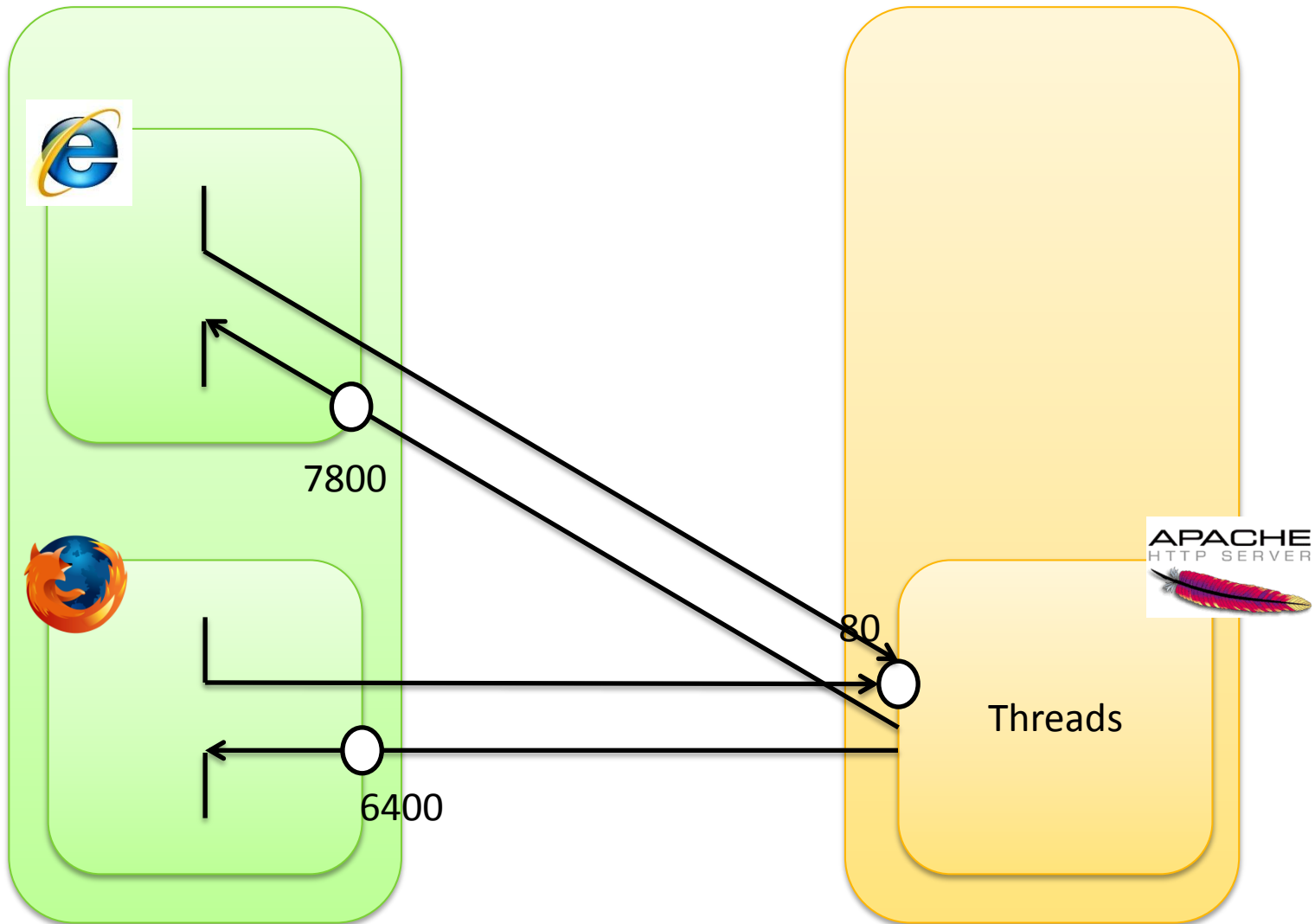
- ▶ Le serveur écoute sur un port donné
- ▶ Le client choisit un port sur lequel recevoir la réponse
- ▶ Le client se connecte au serveur
 - ▶ utilise @IP+port qu'il connaît à l'avance
- ▶ Le serveur répond sur le port du client



Plusieurs clients/serveurs par ordinateurs



Plusieurs clients supportés par serveur



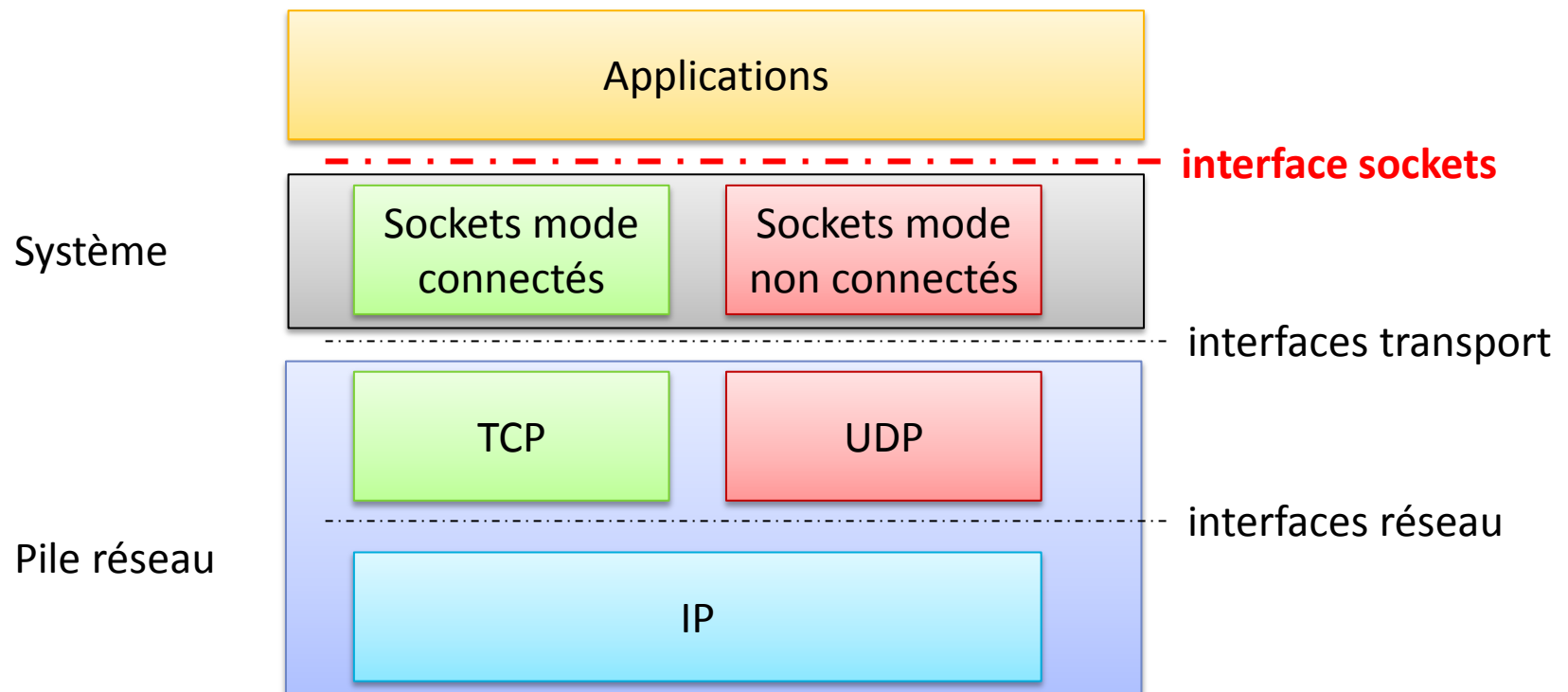
Les sockets



Les sockets (package java.net)

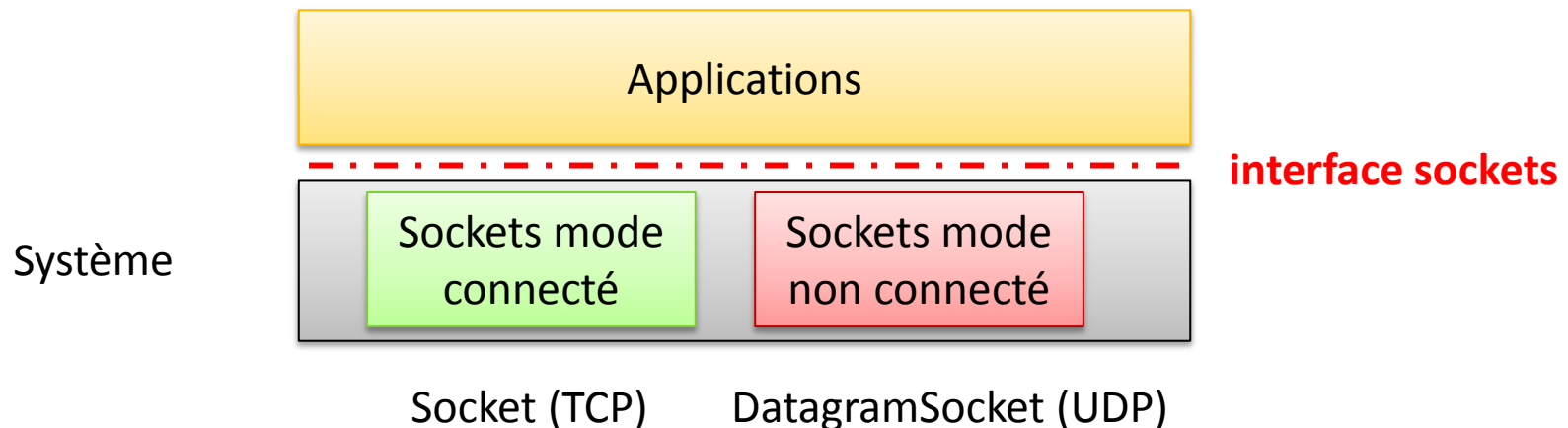


- ▶ Les sockets = API du package java.net
 - ▶ font l'interface entre les programmes d'applications et les couches réseaux



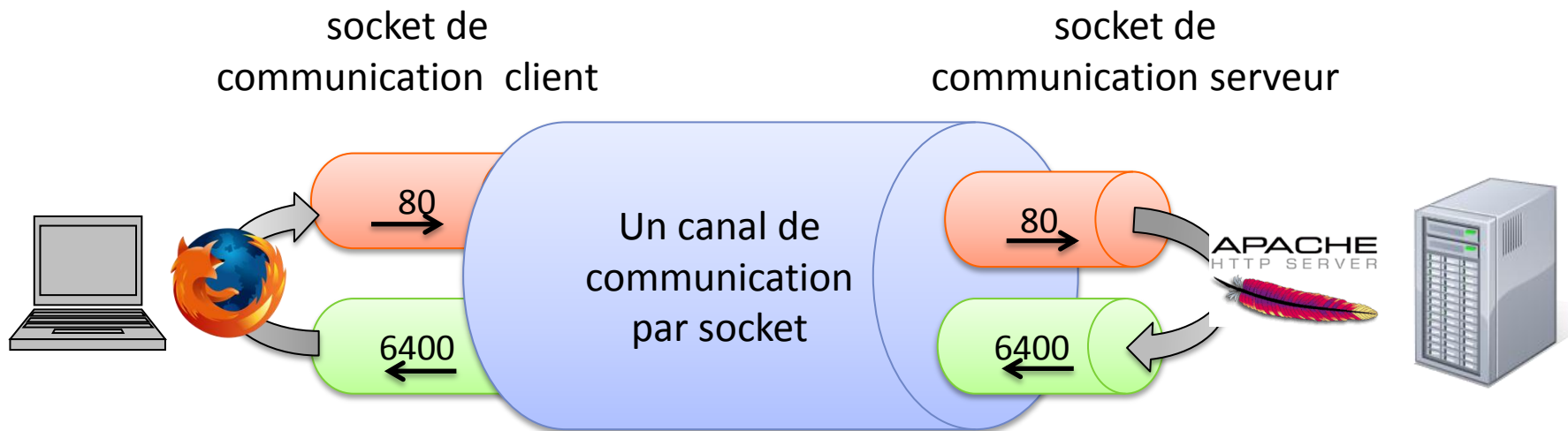
Les sockets (package java.net)

- ▶ Une socket désigne un point d'attache à une machine. On distingue deux types principaux :
 - ▶ socket en mode connecté TCP (`java.net.Socket`)
 - ▶ socket en mode non connecté UDP (`java.net.DatagramSocket`)



Les sockets (package java.net)

- ▶ La connexion se fait point à point en utilisant les **couples (adresse IP + PORT)** pour désigner les points de connections
- ▶ 2 types de sockets sont impliquées dans une communication
 - ▶ au moins deux *sockets de communications* de la classe Socket (une pour le client, une pour le serveur)
 - ▶ au moins une *socket serveur* de la classe SocketServeur sur le serveur



88.77.66.23:6400

173.194.23.43:80

Adresses et ports

Généralités : Ports

- ▶ Un port est utilisé par un seul processus à la fois
- ▶ Plage de ports:
 - ▶ **ports reconnus de 1 à 1023** (il faut les droits administrateur pour les utiliser), services généraux et communs
 - ▶ **ports réservés de 1024 à 49151**, utilisable par n'importe quelle application.
 - ▶ **ports libres 49152 à 65535**, normalement utilisés pour une durée limitée.
- ▶ utiliser un port en dehors de ces plages génère une exception
- ▶ le port 0 est utilisé par l'API Socket pour désigner n'importe quel port libre

Généralités : Adresses IP particulières

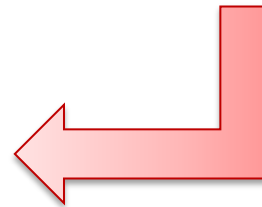
- ▶ Une adresse IP désigne une machine.
- ▶ Adresses **locales** :
 - ▶ généralement 127.0.0.1, "localhost"
- ▶ Adresse de **diffusion** :
 - ▶ 255.255.255.255, xxx.255.255.255, xxx.yyy.255.255, xxx.yyy.zzz.255
- ▶ Adresse **multicast**
 - ▶ de 224.0.0.0 à 239.255.255.255
 - ▶ de 224.0.0.1 à 224.0.0.255 utilisées par des services réseaux.
- ▶ Adresses **privées non routables**
 - ▶ de 10.0.0.1 à 10.255.255.254
 - ▶ de 172.16.0.1 à 172.31.255.254
 - ▶ 192.168.0.1 à 192.168.255.254

Classe `java.net.InetAddress`

- ▶ permet de représenter une adresse IP
- ▶ deux sous classes `Inet4Address` (IPv4), `Inet6Address` (IPv6)
- ▶ Méthodes utiles:
 - ▶ `static InetAddress getByName(String host)` : Retourne l'adresse du host.
 - ▶ host peut être une IP ou un nom de domaine.
 - ▶ `static InetAddress getLocalHost()` : Retourne l'adresse locale

```
System.out.println(InetAddress.getByName("192.168.0.1"));  
System.out.println(InetAddress.getByName("www.google.fr"));
```

```
/192.168.0.1  
www.google.fr/74.125.132.94
```



Classe `java.net.NetworkInterface`

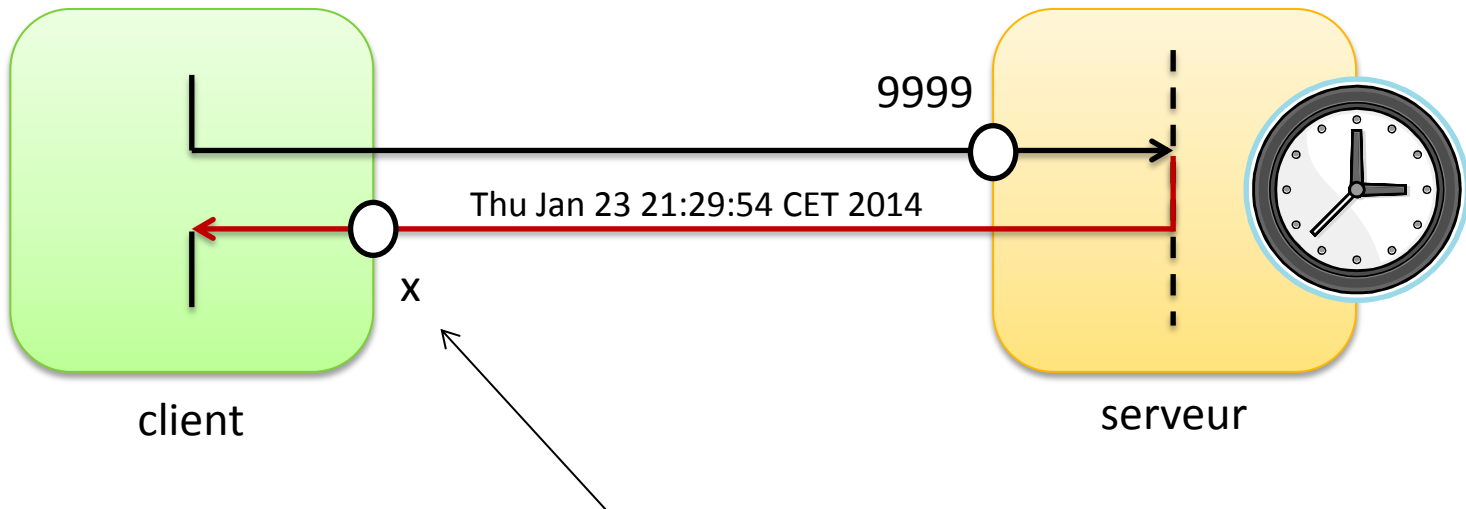
- ▶ La classe `NetworkInterface` permet de représenter une interface réseau
 - ▶ `NetworkInterface.getNetworkInterfaces()` : permet d'obtenir une énumération des interfaces.
 - ▶ `getInetAddresses()` permet d'obtenir la liste des adresses associées à l'interface
 - ▶ `String getName()` : avoir le nom
 - ▶ ex : "eth0"
 - ▶ `String getDisplayName()` : avoir le nom affiché à l'utilisateur
 - ▶ ex : "Software Loopback Interface 1"

```
Enumeration<NetworkInterface> interfaceList = NetworkInterface.getNetworkInterfaces();
while (interfaceList.hasMoreElements()) {
    NetworkInterface itf = interfaceList.nextElement();
    System.out.println(itf);
}
```

Socket TCP : la base

Exemple : un serveur de temps en TCP

- ▶ Le serveur attends sur le port 9999
- ▶ Le client se connecte en TCP (pas de message envoyé)
- ▶ Le serveur envoie la date sur le port X choisit par le client

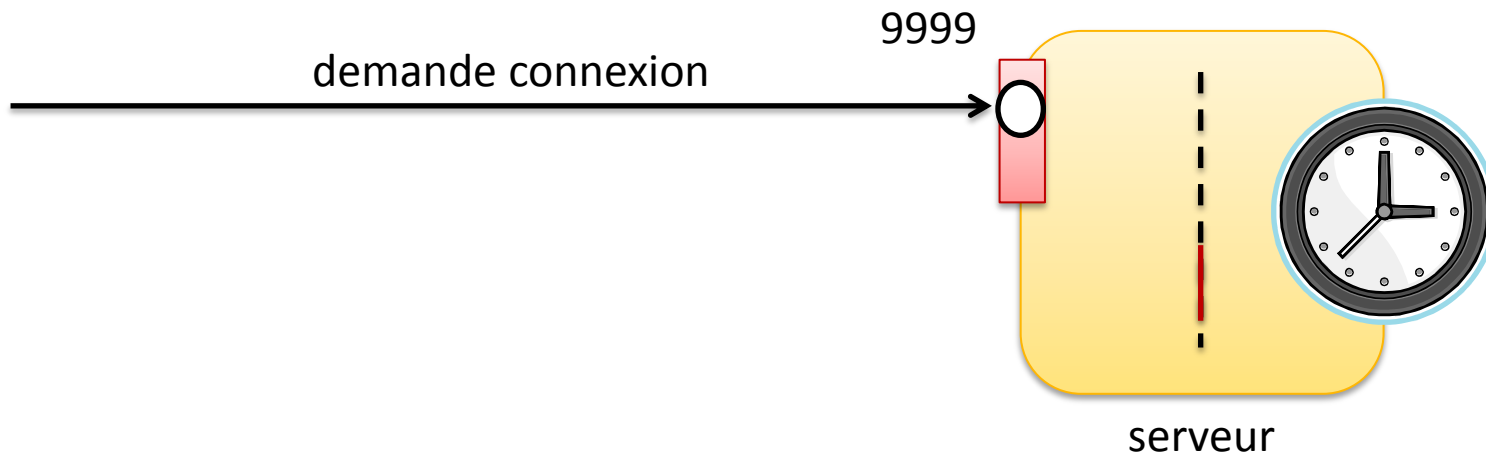


port de retour choisit au hasard parmi les disponibles
(automatiquement par l'API)

Le serveur : 2 types de sockets

► Deux types de sockets :

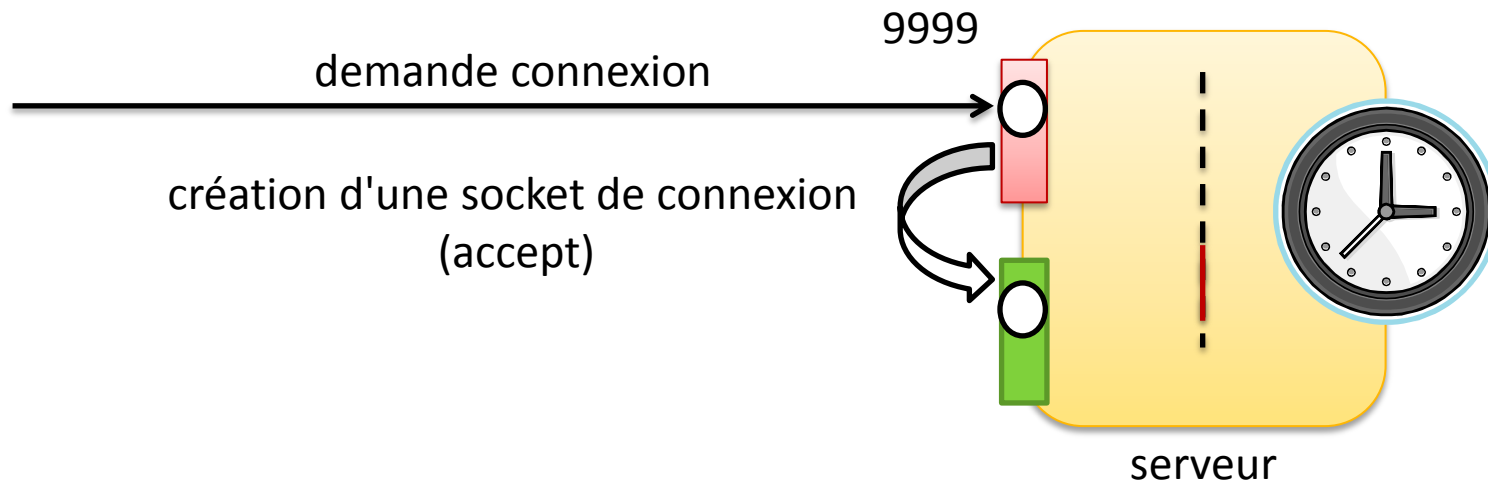
- une **socket d'attente dite serveur** (`java.net.SocketServer`) qui attends la connexion au serveur
- une ou plusieurs **sockets de communication** (`java.net.Socket`) qui lie le serveur aux clients



Le serveur : 2 types de sockets

► Deux types de sockets :

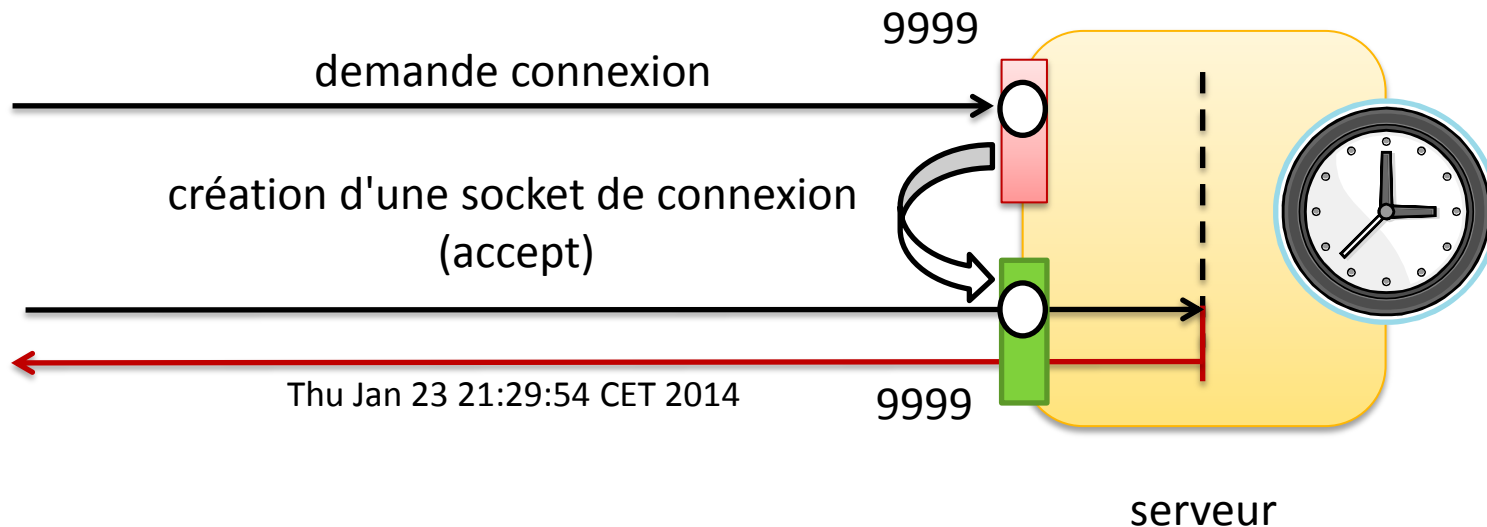
- une **socket d'attente dite serveur** (`java.net.SocketServeur`) qui attends la connexion au serveur
- une ou plusieurs **sockets de communication** (`java.net.Socket`) qui lie le serveur aux clients



Le serveur : 2 types de sockets

► Deux types de sockets :

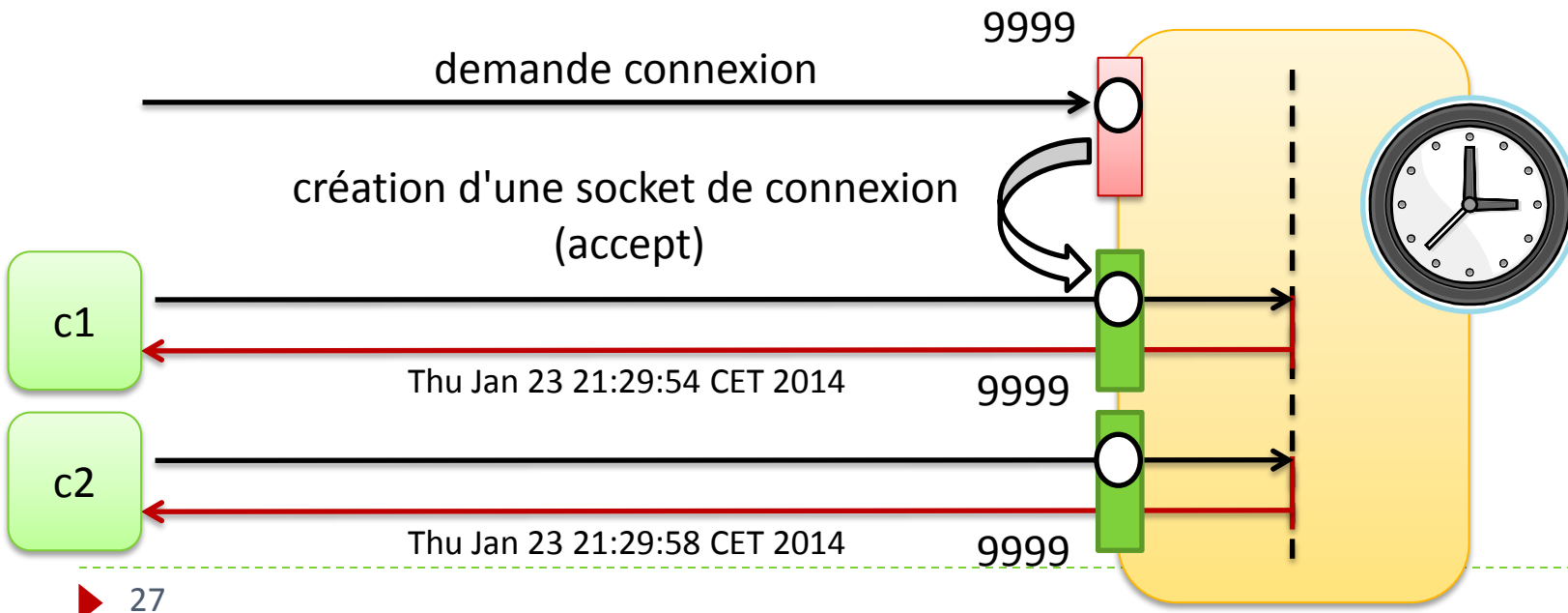
- une **socket d'attente dite serveur** (`java.net.SocketServeur`) qui attends la connexion au serveur
- une ou plusieurs **sockets de communication** (`java.net.Socket`) qui lie le serveur aux clients



Le serveur : 2 types de sockets

► Deux types de sockets :

- une **socket d'attente dite serveur** (`java.net.SocketServeur`) qui attends la connexion au serveur
- une ou plusieurs **sockets de communication** (`java.net.Socket`) qui lie le serveur aux clients



Socket serveur (java.net.ServerSocket)

Serveur : La socket Serveur (attente)

- ▶ La socket du type **java.net.ServerSocket** :
 - ▶ généralement une par serveur (le port qu'il écoute)
 - ▶ attend la connexion sur le port
 - ▶ crée les sockets de communication vers clients

- ▶ Plusieurs étapes pour la socket serveur:
 1. **créer** ServerSocket (appel à un constructeur)
 2. **lier** à un **port** (méthode bind ou constructeur)
 3. **écouter** sur ce **port**
 4. **accepter** la connexion (méthode accept)
 5. **créer** Socket de communication vers client
 6. **fermeture** (méthode close)



Créer et lier une `java.net.ServerSocket`

► Constructeurs :

- ▶ `ServerSocket()` : créé la socket sans la lier
- ▶ `ServerSocket(int port)` : créé la socket et la lie au port, si 0 alors le port est arbitraire.
- ▶ ...

► Liaison :

- ▶ `void bind(int port)` : permet de se lier à un port donné

► Vérifier l'attachement :

- ▶ `getInetAddress()`, `getLocalPort()` et `getLocalSocketAddress()`

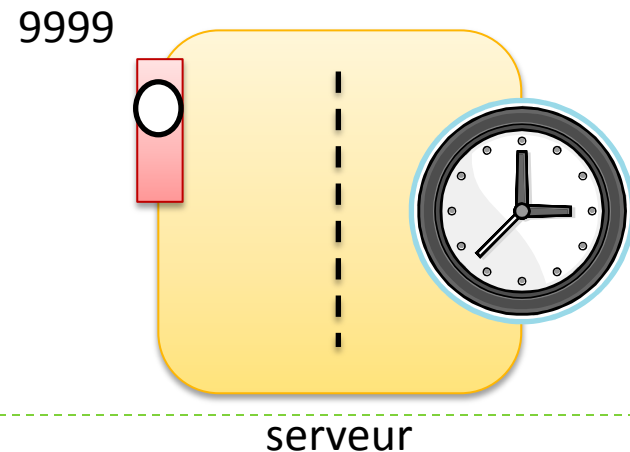
Exemple de création pour notre serveur

► Soit

```
int portEcouté = 9999;  
ServerSocket socketServeur = new ServerSocket(portEcouté);
```

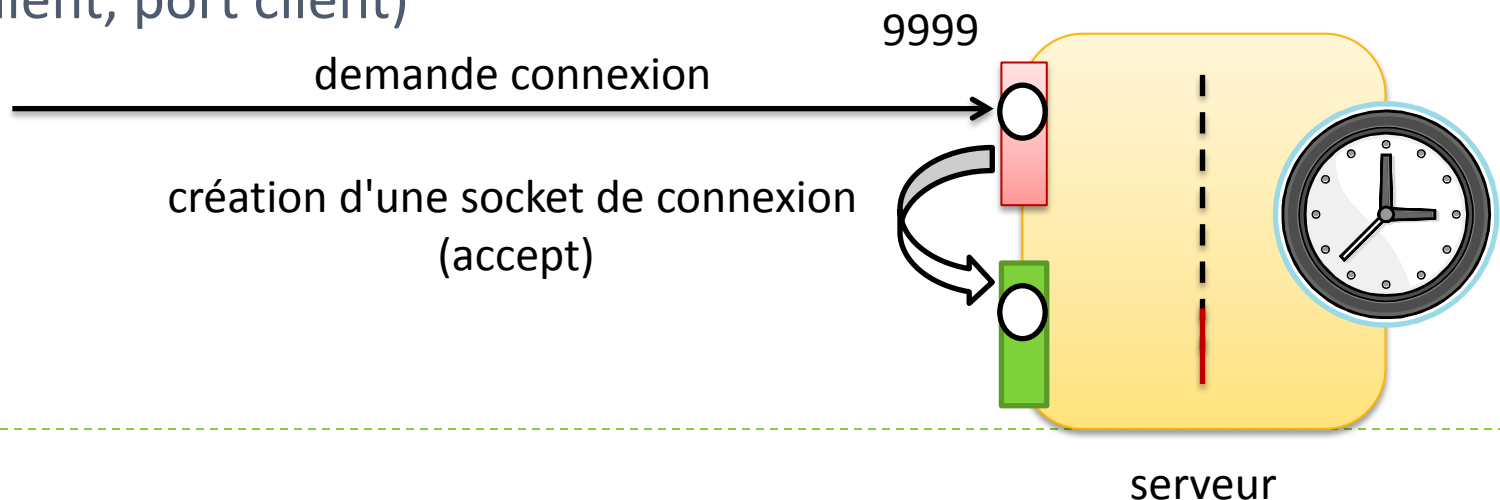
► Soit

```
int portEcouté = 9999;  
ServerSocket socketServeur = new ServerSocket();  
socketServeur.bind(portEcouté);
```



Ecouter , accepter, créer la socket de com ...

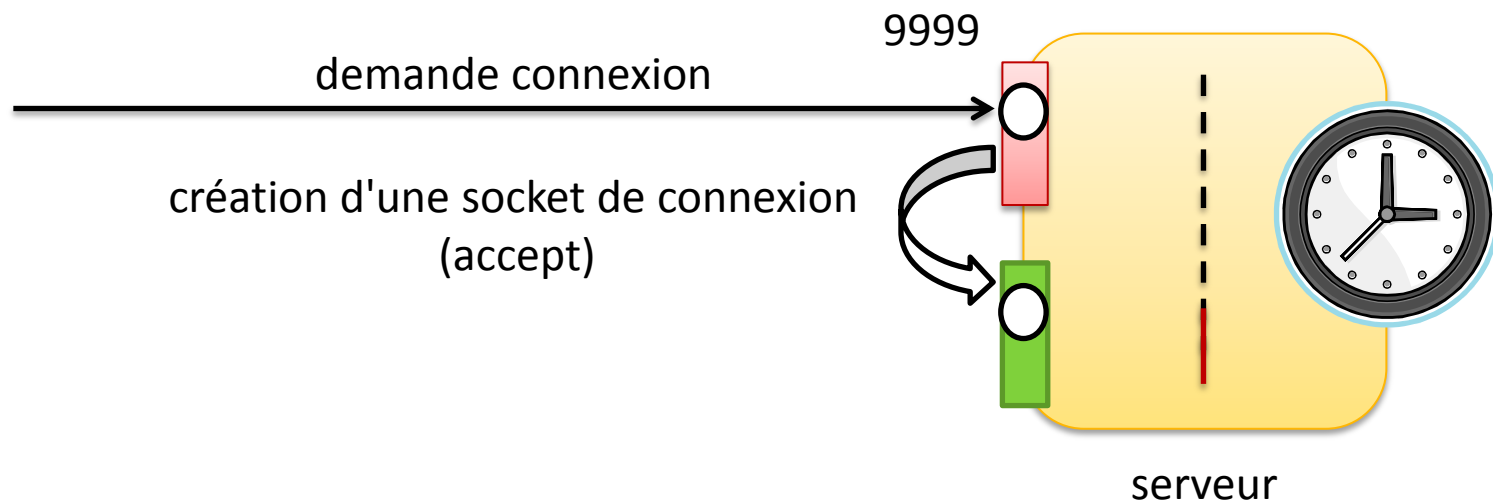
- ▶ Méthode : **Socket accept()**
 - ▶ **bloquante par défaut**, c'est à dire arrête l'exécution du thread en cours tant qu'il n'y a pas de connexion
 - ▶ **non bloquante** si `setSoTimeout(int millis)` a été spécifié
- ▶ L'objet retourné est une **socket de communication** (appelée aussi socket de service).
 - ▶ elle lie le serveur (@ip serveur, port serveur) à un client (@ip client, port client)



Exemple de code d'accept

- On traite les clients séquentiellement (while)

```
while (true) {  
    System.out.println("Attends les clients");  
    Socket socketVersUnClient = socketServeur.accept();  
    System.out.println("Le client " + socketVersUnClient.getInetAddress() + " est connecté");  
    //....  
}
```



Fermeture de ServerSocket: close

- ▶ On doit fermer la socket de serveur lorsque l'on a terminé d'écouter les clients.
 - ▶ permet de libérer les ressources.

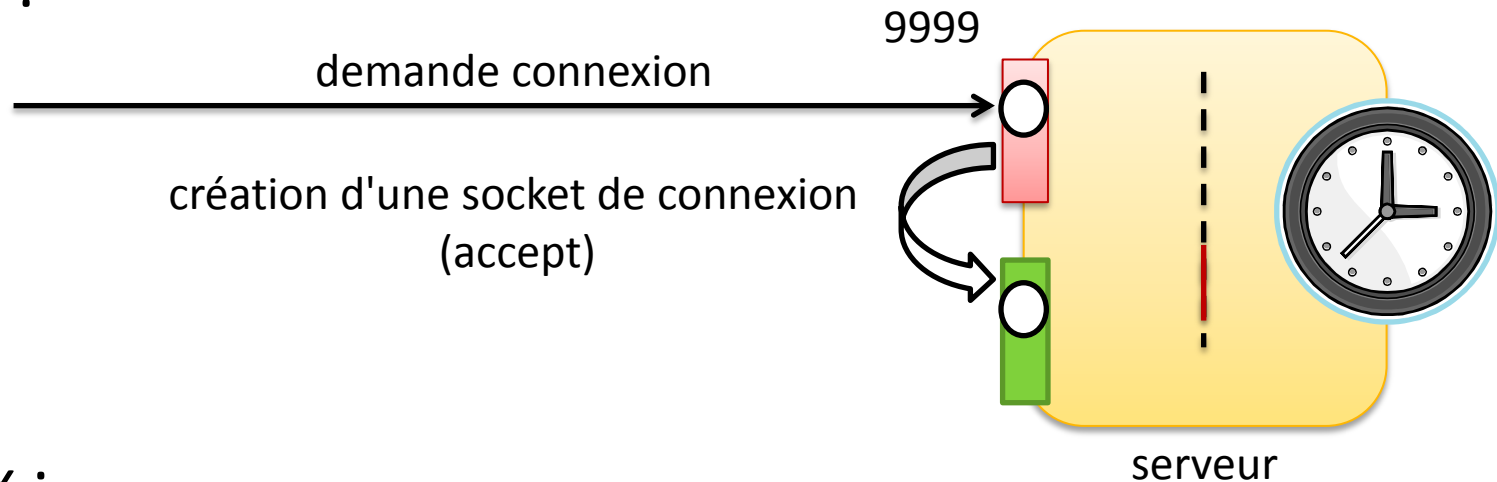
```
socketServeur.close();
```



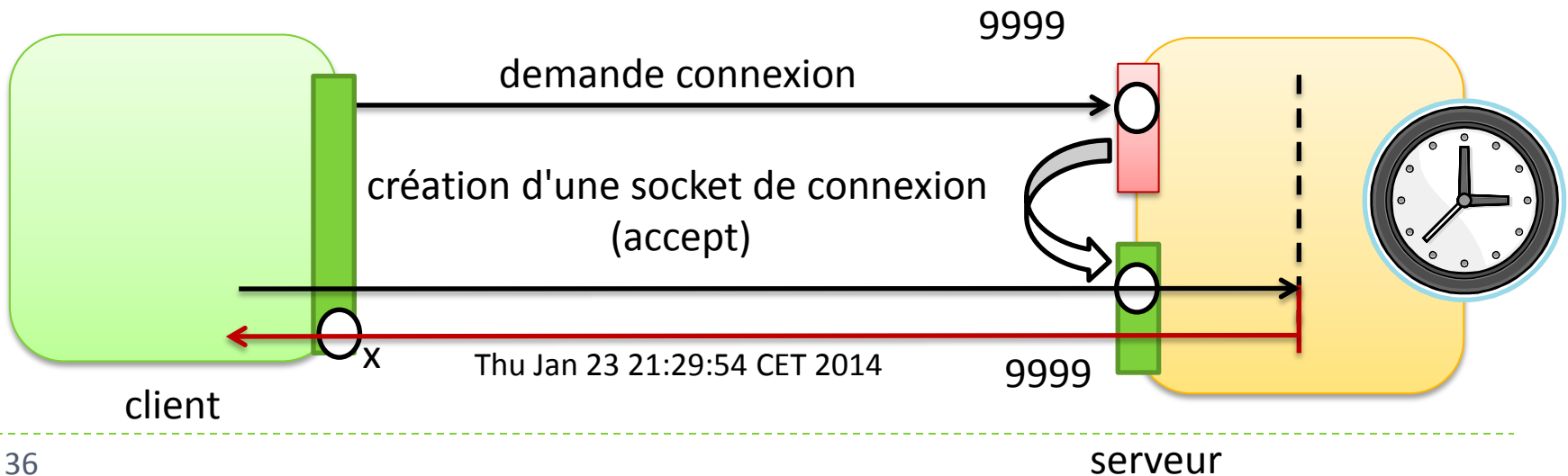
Sockets de communication (java.net.Socket)

Les sockets de communication

► On a vu :



► On veut :



Sockets de communication (java.net.Socket)

- ▶ Utilisées lorsque la connexion est établie entre les deux parties (client/serveur).
 - ▶ la relation est alors symétrique
- ▶ Plusieurs étapes :
 1. **Créer** la socket de communication (constructeur ou fait par le accept de la socket serveur)
 2. **Lier** la socket au serveur distant et port local souvent arbitraire (bind)
 3. **Envoyer** (getOutputStream) des octets
 4. ou/et **Recevoir** (getInputStream) des octets
 5. **Fermer** la connexion (close)



Envoyer recevoir des données et fermeture

- ▶ Méthodes utilisées par le serveur et par le client
- ▶ Envoyer recevoir :
 - ▶ `InputStream getInputStream()` : permet de lire les informations reçues par la socket (en octets)
 - ▶ `OutputStream getOutputStream()` : permet d'écrire les informations reçues par la socket (en octets)
 - ▶ `close()` : fermeture de la connexion
- ▶ Fermeture
 - ▶ méthode `close()`

Coté serveur (java.net.Socket)

```
public class ServeurTemps {  
    public static void main(String[] args) throws IOException {  
        int portEcoule = 9999;  
        ServerSocket socketServeur = new ServerSocket(portEcoule);  
        try {  
            while (true) {  
                System.out.println("Attends les clients");  
                Socket socketVersUnClient = socketServeur.accept();  
                System.out.println("Le client " + socketVersUnClient.getInetAddress() + " est connecté");  
                OutputStream out = socketVersUnClient.getOutputStream();  
                Date date = new Date();  
                out.write(date.toString().getBytes());  
                socketVersUnClient.close();  
            }  
        } catch (IOException exception) {  
            System.out.println("Erreur " + exception.getMessage());  
        } finally {  
            socketServeur.close();  
        }  
    }  
}
```

← La socket serveur

← La socket cliente

← on convertit la date en chaîne puis en byte getByte(); et on l'écrit

Créer et lier (java.net.Socket)

► Construction et Liaison

► Socket()

- puis bind(SocketAddress bindPoint) pour attachement local, et
- connect(SocketAddress) ou connect(SocketAddress, int) pour établir la connexion (int = timeout éventuel)

► Socket(InetAddress addr, int port)

► Socket(String host, int port)

► Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)

► Socket(String host, int port, InetAddress localAddr, int localPort)

Coté client (java.net.Socket)

```
public class ClientTemps {
```

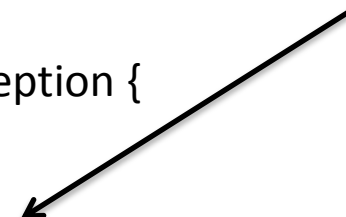
```
    public static void main(String[] args) throws IOException {
```

```
        int portDuServeur = 9999;
```

```
        String adresseDuServeur = "192.168.0.34";
```

```
        Socket socketVersLeServeur = new Socket(adresseDuServeur, 9999);
```

créé la socket cliente



```
        System.out.println("Connecté à " + socketVersLeServeur.getInetAddress());
```

```
        InputStream in = socketVersLeServeur.getInputStream();
```

```
        byte buffer[] = new byte[100];
```

```
        in.read(buffer);
```

```
        String date = new String(buffer);
```

```
        System.out.println("Date : " + date);
```


← on lit la date en octet
on convertit le buffer en String
on affiche la date

```
        socketVersLeServeur.close();
```

← fermeture

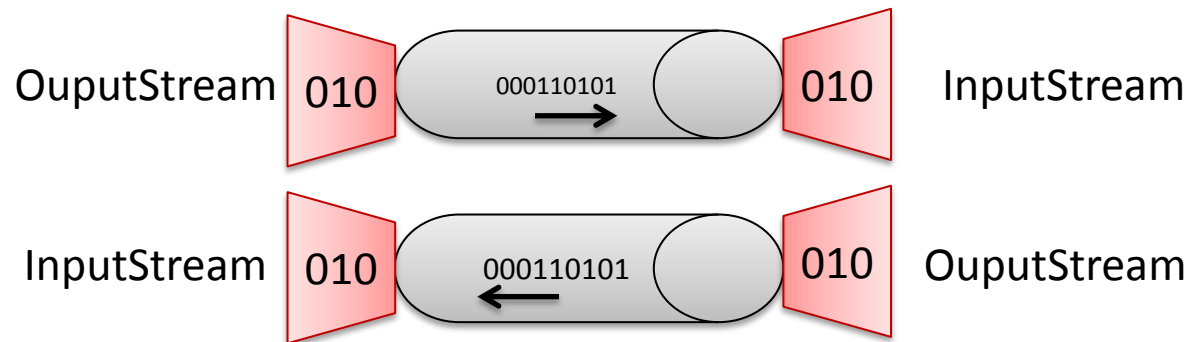
```
    }
```

```
}
```



Rappel sur les streams et les Reader/Writers du package java.io

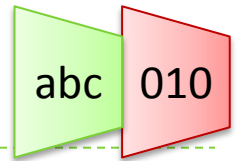
- ▶ En Java, les stream sont une abstraction qui permettent d'écrire et lire des informations en provenance d'un fichier, de la console, ... et **des sockets**.
- ▶ Les stream lisent et écrivent des **données binaires**.
- ▶ Deux grands types de stream :
 - ▶ les InputStream : lire des octets
 - ▶ les OutputStream : écrire des octets



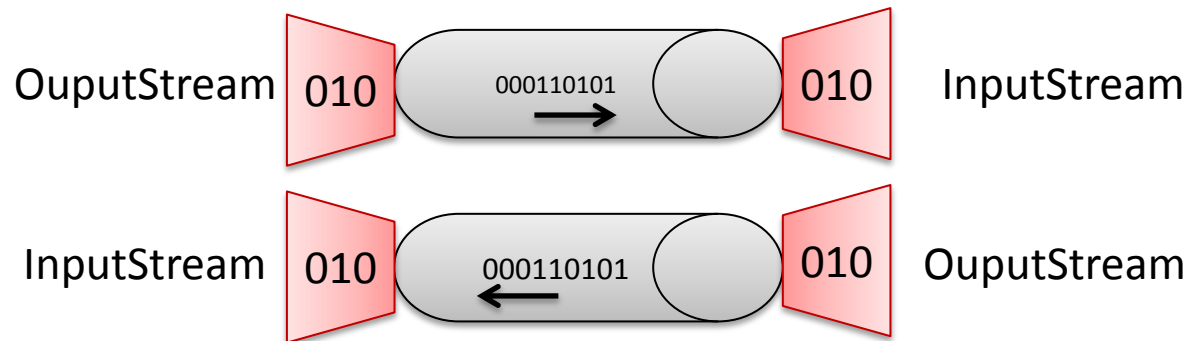
- ▶ Lecture **bloquante** d'octets en entrée (input)
- ▶ Lire un octet et renvoie ce byte ou -1 si c'est la fin du flux
 - ▶ `int read()`
- ▶ Lire un tableau de byte (plus efficace car limite les accès)
 - ▶ `int read(byte[] buffer)`
 - ▶ `int read(byte[] buffer, int off, int len)`
- ▶ Ignorer un nombre n de d'octets
 - ▶ `long skip(long n)`
- ▶ Fermer le flux
 - ▶ `void close()`

- ▶ Ecrire un octet (sous la forme d'un int)
 - ▶ void **write**(int b)
- ▶ Ecrire un tableau de byte (limite nombre d'accès)
 - ▶ void **write**(byte[] buffer)
 - ▶ void **write**(byte[] buffer, int off, int len)
- ▶ Demande d'écrire ce qu'il y a dans le buffer
 - ▶ void **flush**()
- ▶ Ferme le flux
 - ▶ void **close**()

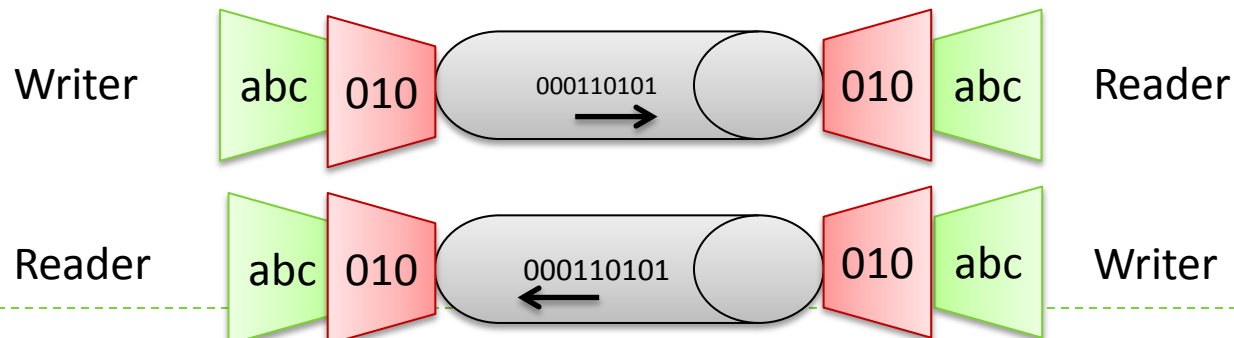
Deux grandes catégories de flux



- Les stream (InputStream/OutputStream) et leur sous-classes qui permettent de lire et écrire des octets.



- Les Reader/Writer et sous-classes qui permettent d'écrire des caractères.

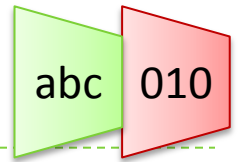


Reader : Lecture de flux de caractères

- ▶ Les lectures sont **bloquantes**
- ▶ Lire un char et renvoie celui-ci ou -1 si c'est la fin du flux
 - ▶ `int read()`
- ▶ Lire un tableau de char
 - ▶ `int read(char[] b)`
 - ▶ `int read(char[] b, int off, int len)`
- ▶ Ignorer un nombre n de de caractères
 - ▶ `long skip(long n)`
- ▶ Fermer le flux
 - ▶ `void close()`
- ▶ Savoir si il reste des caractères à lire
 - ▶ `boolean ready();`

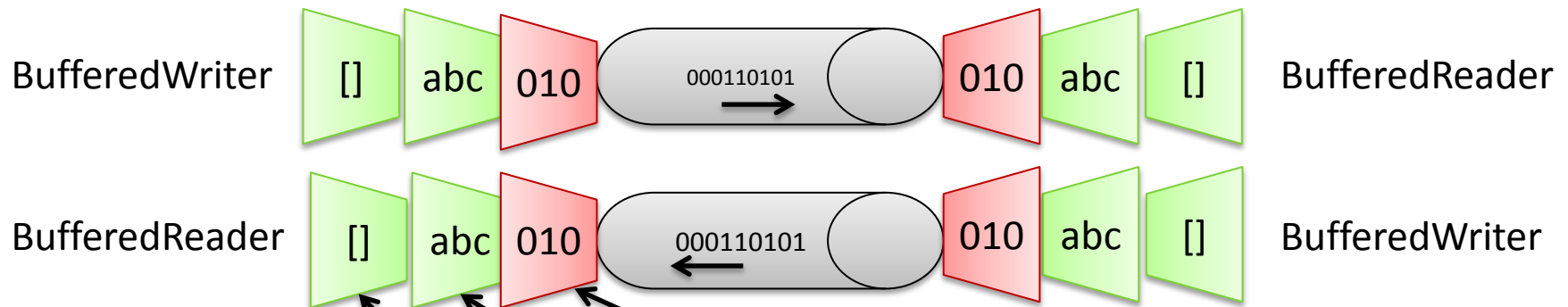
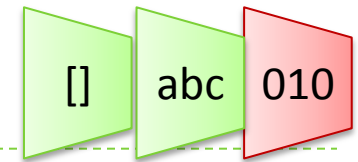
Writer: Ecriture de flux de caractères

- ▶ Ecriture de caractère en sortie
- ▶ Ecrire un caractère
 - ▶ void **write**(char c)
- ▶ Ecrire un tableau de caractère
 - ▶ void **write**(char[] b)
 - ▶ void **write**(char[] b, int off, int len)
- ▶ Demande d'écrire ce qu'il y a dans le buffer
 - ▶ void **flush**()
- ▶ Ferme le flux
 - ▶ void **close**()



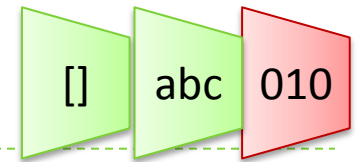
- ▶ Les flots d'octets ou de caractères ne sont pas bufferisés en Java par défaut :
 - ▶ problème de performance car accès fréquent aux couches sous jacentes plus lente (fichiers, réseau, ...)
- ▶ Il faut utiliser les méthodes `read(bytes[])` ou `read(char[])` particulièrement en lecture.
- ▶ Les sous classes `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` et `BufferedWriter` permettent de le faire automatiquement

Construction (Décorator Pattern)



```
InputStream in = System.in;  
BufferedReader inBuffer = new BufferedReader(new InputStreamReader(in));  
  
OutputStream out = System.out;  
BufferedWriter outBuffer = new BufferedWriter(new OutputStreamWriter(out));
```

Méthodes utiles des BufferedReader



- ▶ Lire ligne par ligne :

- ▶ `String readLine();`

```
while ((thisLine = br.readLine()) != null) {  
    System.out.println(thisLine);  
}
```

- ▶ Marquer une position pour y revenir :

- ▶ `mark(int readLimit)` : la limite est le nombre de caractère pouvant être lu avant d'effacer la marque

- ▶ Revenir à la position :

- ▶ `reset()` : revient à la marque

- ▶ Penser à utiliser `flush()` pour envoyer le contenu !

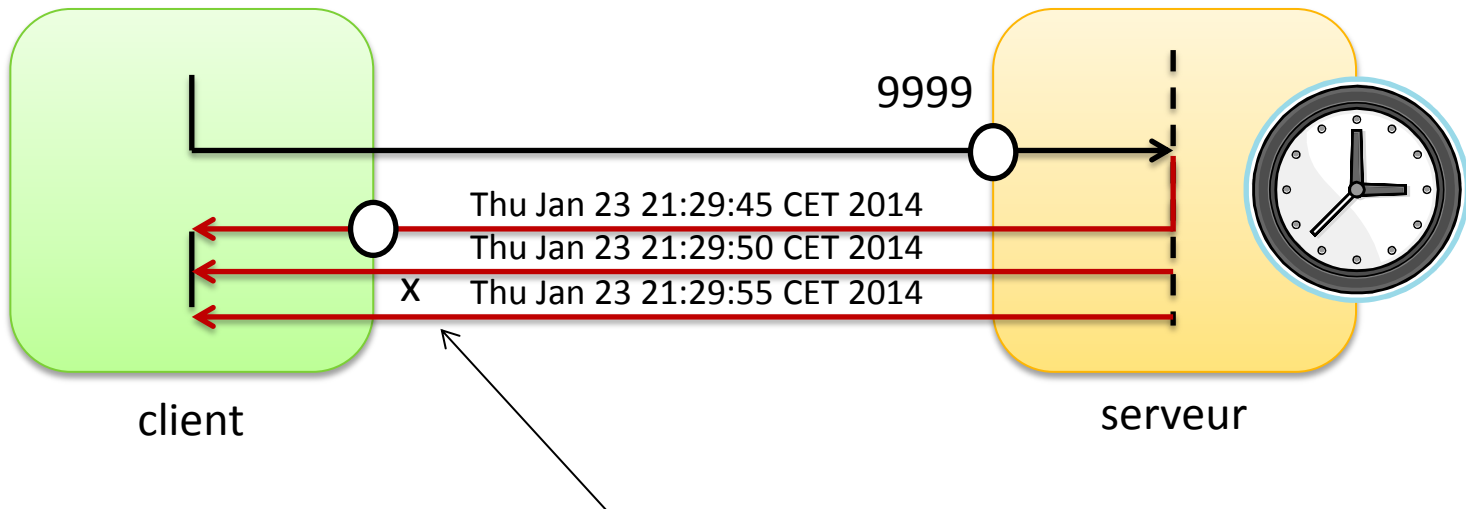
Méthodes utiles pour `PrintWriter`

- ▶ `System.out` est un `PrintWriter` :
 - ▶ `println(String)`
 - ▶
- ▶ `flush()`; <= important dans le cas où l'on souhaite envoyer le contenu avant que le tampon soit plein ...

Exemple avec Writer/Reader

Exemple : un serveur de temps en TCP

- ▶ Le serveur attends sur le port 9999
- ▶ Le client se connecte en TCP (pas de message envoyé)
- ▶ Le serveur envoie 100 fois la date sur le port X choisit par le client toute les 5 s



port de retour choisit au hasard parmi les disponibles
(automatiquement par l'API)

Code serveur

```
public class ServeurTemps {  
    public static void main(String[] args) throws IOException, InterruptedException {  
        int portEcoule = 9999;  
        ServerSocket socketServeur = new ServerSocket(portEcoule);  
        try {  
            while (true) {  
                Socket socketVersUnClient = socketServeur.accept();  
                PrintWriter out = new PrintWriter(new OutputStreamWriter(socketVersUnClient.getOutputStream()));  
                for (int i = 0; i < 100; i++) {  
                    Date date = new Date();  
                    out.println(date.toString());  
                    out.flush();  
                    Thread.sleep(5000);  
                }  
                socketVersUnClient.close();  
            }  
        } catch (IOException exception) {  
            System.out.println("Erreur " + exception.getMessage());  
        } finally {  
            socketServeur.close();  
        }  
    }  
}
```

on crée un PrintWriter

on donne directement une chaîne (plus de conversion en octets)

▶ 55

Code client

```
public class ClientTemps {
```

```
    public static void main(String[] args) throws IOException {
```

```
        int portDuServeur = 9999;
```

```
        String adresseDuServeur = "192.168.0.34";
```

```
        Socket socketVersLeServeur = new Socket(adresseDuServeur, 9999);
```

on crée un BufferedReader

```
        BufferedReader in = new BufferedReader(new InputStreamReader(socketVersLeServeur.getInputStream()))
```

```
        String date;
```

```
        while ((date = in.readLine()) != null) {
```

```
            System.out.println(date);
```

```
        }
```

```
        socketVersLeServeur.close();
```

```
    }
```

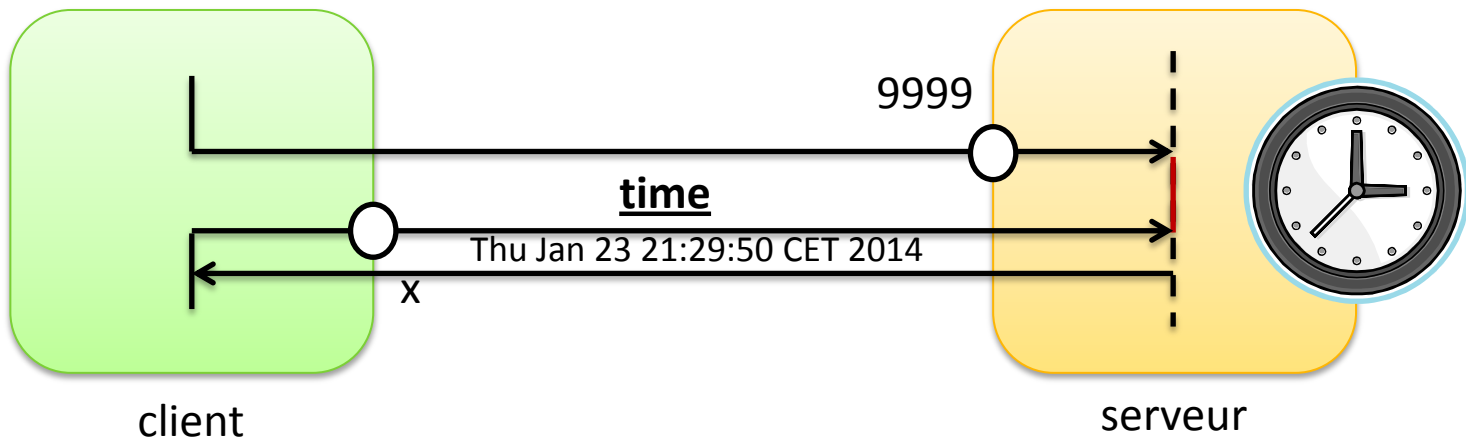
```
}
```

On lit ligne par ligne les réponses du serveur

Avec écriture/lecture des deux cotés

Exemple : un serveur de temps en TCP

- ▶ Le serveur attends sur le port 9999
- ▶ Le client se connecte en TCP
- ▶ Le client demande "time", le serveur répond



Coté client

```
public ClientTemps{  
    //....
```

```
    PrintWriter out = new PrintWriter(new OutputStreamWriter(socketVersLeServeur.getOutputStream()));  
    out.println("time");  
    out.flush();
```

```
    BufferedReader in = new BufferedReader(new InputStreamReader(socketVersLeServeur.getInputStream()));  
    String date;  
    while ((date = in.readLine()) != null) {  
        System.out.println(date);  
    }
```

```
    //...
```



Coté serveur

```
public ServeurTemps{  
    //...
```

```
    BufferedReader in = new BufferedReader(new InputStreamReader(socketVersUnClient.getInputStream()));  
    String requete = in.readLine();
```

```
    System.out.println("requete is " + requete);
```

```
    PrintWriter out = new PrintWriter(new OutputStreamWriter(socketVersUnClient.getOutputStream()));
```

```
    if (requete.equals("time")) {  
        for (int i = 0; i < 100; i++) {  
            Date date = new Date();  
            out.println(date.toString());  
            out.flush();  
            Thread.sleep(5000);  
        }  
    } else {  
        out.println("Bad request");  
    }  
}
```



Gestion des exceptions

Attraper les exceptions en Java

- ▶ Les classes de `java.net` et `java.io` génèrent des `IOException`
- ▶ Il faut les attraper pour programmer correctement.
 - ▶ penser à fermer les ressources dans le `finally`

```
try{  
    //...  
} catch (IOException exception) {  
    System.out.println("Erreur " + exception.getMessage())  
} finally {  
    socketServeur.close();  
}
```

JAVA 7 : try-with-resources

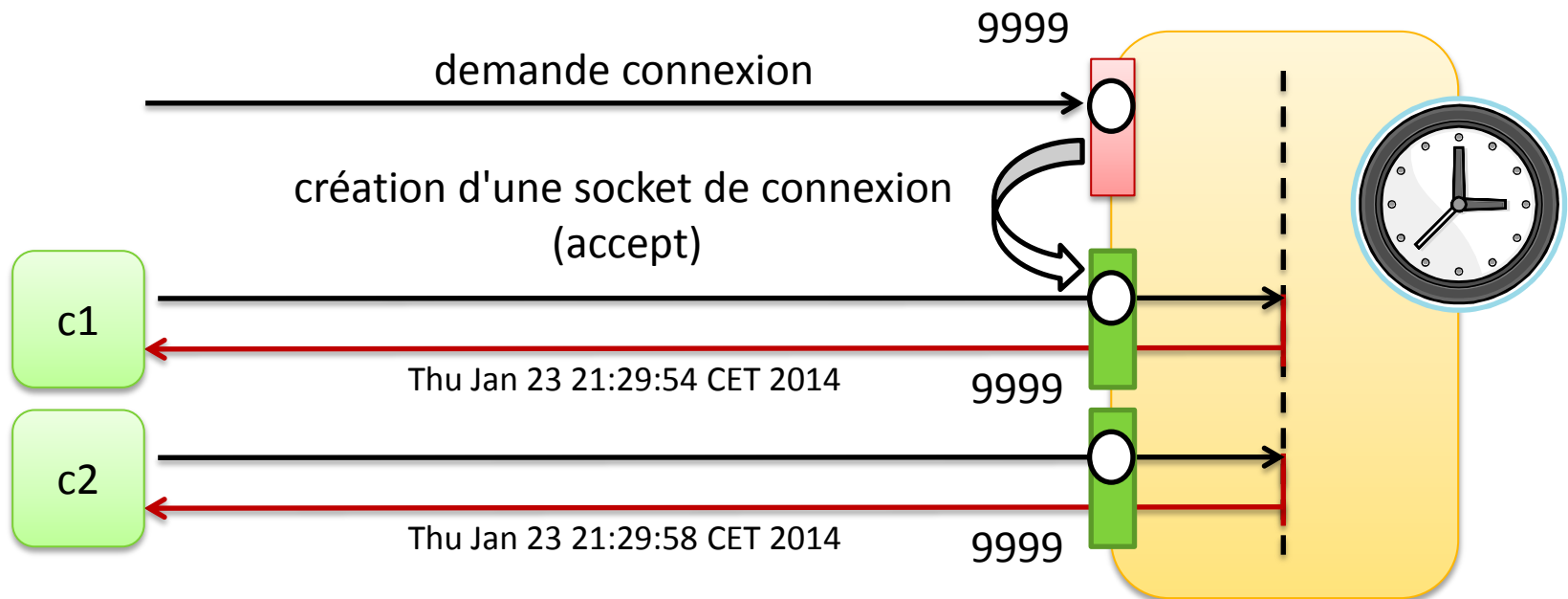
- ▶ Permet de déclarer des ressources (qui étendent **Closable**) dans le bloc **try**
- ▶ Les ressources sont automatiquement fermées en cas d'exception

```
try (Socket socketVersUnClient = socketServeur.accept()) {  
    //...  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

Gestion de plusieurs clients à la fois

De serveur itératif à serveur parallèle

- ▶ Le serveur actuel ne peut pas traiter en parallèle deux clients, il est itératif :
 - ▶ à tel point qu'il faut attendre la déconnexion d'un client pour pouvoir traiter le client suivant



Les Threads en Java, La classe Thread

- ▶ Par défaut 1 thread, celui de main
- ▶ Classe Thread :
 - ▶ `start()` : démarre un Thread
 - ▶ `run()` : méthode exécutée lorsqu'on démarre le Thread
 - ▶ `stop()` : stop un Thread
 - ▶ `join()` : fait attendre le thread appelant jusqu'à la fin du thread sur lequel il appelle `join`.
- ▶ Méthode statique
 - ▶ `static Thread.sleep(int ms)` : fait attendre le Thread appelant pendant `n ms`
- ▶ C'est généralement une mauvaise idée d'étendre Thread directement

L'interface Runnable

► Une seule méthode :

- `run()` : la méthode qui sera exécutée lorsque le Thread est démarré

```
public class MonRunnable implements Runnable {  
    public void run() {  
        while (true) {  
            try {  
                System.out.println("Affiche toute les s");  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Créer et lancer un Thread à partir d'un Runnable

► Constructeur Thread(Runnable r)

```
Thread t = new Thread(new MonRunnable());  
Thread t2 = new Thread(new MonRunnable());  
  
t.start();  
t2.start();
```

- Créé 2 thread à partir de MonRunnable et les démarre.
On a donc 3 threads :
 - un pour le main
 - un pour t
 - un pour t2

Version Multithreadée du serveur

- ▶ On va déléguer à un Thread le traitement des socket de communications pour chaque client

```
public class ServeurTemps {  
    public static void main(String[] args) throws IOException, InterruptedException {  
        int portEcoute = 9999;  
        try (ServerSocket socketServeur = new ServerSocket(portEcoute)) {  
            while (true) {  
                Socket socketVersUnClient = socketServeur.accept();  
                Thread t = new Thread(new TraiteUnClient(socketVersUnClient));  
                t.start();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Classe TraiteUnClient, constructeur

- Il faut qu'on récupère la socket dans le constructeur

```
public class TraiteUnClient implements Runnable {
```

```
    private Socket socketVersUnClient;
```

```
    TraiteUnClient(Socket socket) {  
        socketVersUnClient = socket;  
    }
```

```
    @Override  
    public void run() {  
        //....  
    }
```

Classe TraiteUnClient, méthode run

```
@Override
public void run() {
    try {
        BufferedReader in = new BufferedReader(new InputStreamReader(socketVersUnClient.getInputStream()));
        String requete = in.readLine();
        PrintWriter out = new PrintWriter(new OutputStreamWriter(socketVersUnClient.getOutputStream()));
        if (requete.equals("time")) {
            for (int i = 0; i < 100; i++) {
                Date date = new Date();
                out.println(date.toString());
                out.flush();
                Thread.sleep(5000);
            }
        } else {
            out.println("Bad request");
        }
        socketVersUnClient.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



Problèmes de la mémoire partagée

- ▶ les threads peuvent avoir accès à des ressources partagées :
 - ▶ listes, collections, variables, ...
 - ▶ il faut maintenir la cohérence du système (race condition, ...)
- ▶ Le mot-clef ***synchronized***, empêche l'accès simultané à une méthode par deux threads :

```
public synchronized void retirerArgent() {  
    // retire de l'argent  
}  
  
public synchronized void afficherSoldeDeCompte() {  
    // affiche le solde  
}
```


Gestion des threads, notion de pool

- ▶ Les threads sont consommateurs de ressources :
 - ▶ mémoire, processeur, ...
 - ▶ la création d'un Thread est couteuse.
- ▶ Un serveur doit pouvoir limiter le nombre de client qu'il sert en même temps.
- ▶ Solution : **Pool de Threads**
 - ▶ on met à disposition un nombre maximal N de Thread qui peuvent répondre aux clients
 - ▶ on réutilise les Threads existant
 - ▶ on met en attente les clients si le nombre maximal est atteint

java.util.concurrent.ExecutorService

- ▶ Java fournit de nombreuses API pour la concurrence dans **java.util.concurrent**
- ▶ Les executors permettent (entre autre) de limiter le nb de Threads et de contrôler leur réutilisation.
- ▶ Executor service = `Executors.newFixedThreadPool(n)`
 - ▶ créé un pool de n thread max
- ▶ `service.execute(Runnable)`
 - ▶ demande l'exécution du Runnable dans un des Threads du pool
 - ▶ si thread dispo > exécution immédiate
 - ▶ sinon attendre la dispo d'un thread

En pratique

- ▶ On réécrit main pour n'avoir que 4 clients à la fois :

```
public static void main(String[] args) throws IOException, InterruptedException {  
    int portEcoule = 9999;  
    try (ServerSocket socketServeur = new ServerSocket(portEcoule)) {  
  
        Executor service = Executors.newFixedThreadPool(4);  
        while (true) {  
            Socket socketVersUnClient = socketServeur.accept();  
            service.execute(new TraiteUnClient(socketVersUnClient));  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

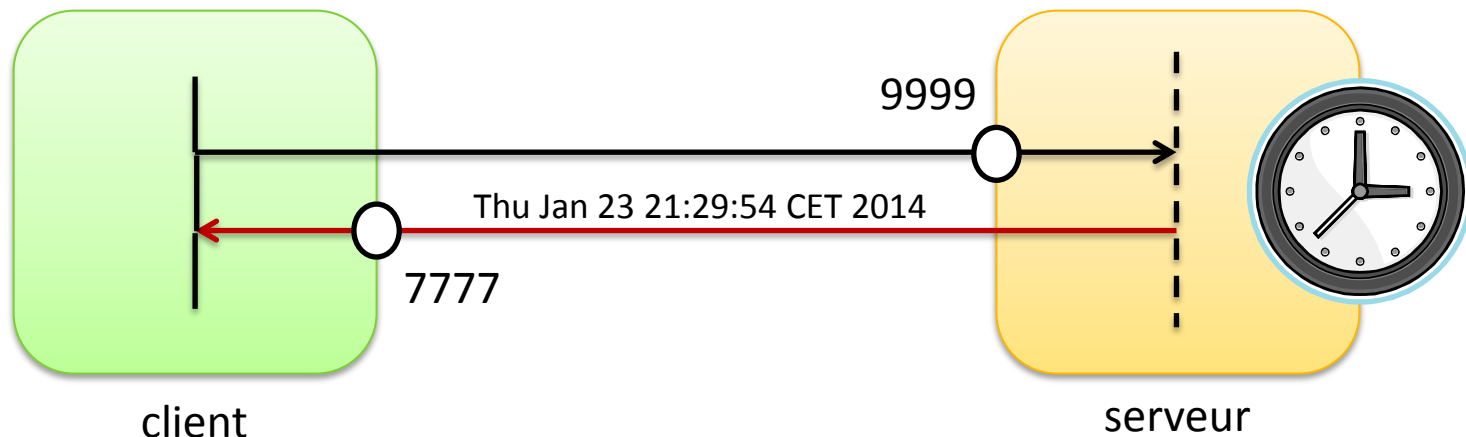
Les DatagramSockets et UDP

Rappels sur UDP over IP

- ▶ Même adressage que TCP :
 - ▶ notion de port + adresse IP
- ▶ Service **non fiabilisé** et **non connecté**
 - ▶ mécanisme d'acquittement à mettre en œuvre si nécessaire (comme pour TFTP)
- ▶ **Pas de garantie sur l'ordre** des messages (datagrammes)
- ▶ Les **datagrammes** sont considérés comme **indépendants** les uns des autres.
- ▶ Utilisé pour la diffusion (broadcast ou multicast)

UDP : sockets non connectées

- ▶ On peut également utiliser des **sockets non connectés basée sur UDP**
- ▶ Un seul type de socket :
 - ▶ classe `java.net.DatagramSocket`
- ▶ Echange des datagrammes :
 - ▶ classe `java.net.DatagramPacket`



java.net.DatagramSocket

- ▶ Utilisée pour la réception et l'envoi
- ▶ **Pour la réception**, il faut attacher la socket à un port donné (et écouter sur le port)
- ▶ Constructeurs :
 - ▶ `DatagramSocket()` : on peut l'attacher plus tard avec `bind` mais pas obligatoire
 - ▶ `DatagramSocket(int port)` : attachée au port `port`
 - ▶ `DatagramSocket(int port, InetAddress laddr)`
- ▶ Liaison :
 - ▶ `bind(SocketAddress addr)`
- ▶ Vérification :
 - ▶ `getLocalPort()`, `getLocalAddress()` et `getLocalSocketAddress()`
- ▶ Fermeture :
 - ▶ `close()`

Méthodes d'envoi/ réception

- ▶ Envoi : void send(DatagramPacket p)
 - ▶ le paquet doit désigner la destination du message si socket non attachée.
 - ▶ Le DatagramPacket doit spécifier les données à envoyer et la machine à qui envoyer
 - ▶ Aucune garantie de bonne délivrance
- ▶ Réception : void receive(DatagramPacket p)
 - ▶ méthode bloquante, attend la réception d'un message p
 - ▶ setTimeout permet de fixer la durée max de l'attente.
 - ▶ Le DatagramPacket doit spécifier la zone où copier les données à recevoir
 - ▶ machine et port sont mis à jour à la réception.

java.net.DatagramPacket

- ▶ Objets qui transitent sur le réseau
- ▶ Constructeurs :
 - ▶ DatagramPacket(byte[] buf, int length)
 - ▶ DatagramPacket(byte[] buf, int length, InetAddress address, int port)
 - ▶ DatagramPacket(byte[] buf, int offset, int length)
 - ▶ DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)


java.net.DatagramPacket

- ▶ Informations sur la machine distante (récepteur ou émetteur selon qui regarde le paquet):
 - ▶ `getSocketAddress()`, `setAddress()`, `getAddress()`, `setPort()`, `getPort()`
- ▶ Les données échangées :
 - ▶ `setData()`, `getData()`, `getOffset()`, `setLength()`
 - ▶ `getLength()` :
 - ▶ en émission :
 - taille des données à envoyer
 - ▶ en réception :
 - pendant l'attente, taille de la zone de stockage
 - une fois reçu, taille des données reçues

Exemple client simple

```
public class ClientTemps {  
  
    public static void main(String[] args) throws IOException {  
        int portDuServeur = 9999;  
        int portDuClient = 8888;  
  
        String adresseDuServeur = "192.168.0.34";  
        DatagramSocket socketVersLeServeur = new DatagramSocket(portDuClient);  
  
        byte[] messageEnBytes = "time".getBytes();  
        DatagramPacket paquetEnvoie = new DatagramPacket(messageEnBytes, messageEnBytes.length, new  
InetSocketAddress(adresseDuServeur, portDuServeur));  
        socketVersLeServeur.send(paquetEnvoie);  
  
        byte buffer[] = new byte[100];  
        DatagramPacket reception = new DatagramPacket(buffer, buffer.length);  
        socketVersLeServeur.receive(reception);  
  
        String date = new String(buffer);  
        System.out.println("Date : " + date);  
  
        socketVersLeServeur.close();  
    }  
}
```

création d'un paquet pour la
réception



Exemple de serveur simple

```
public class ServeurTemps {  
    public static void main(String[] args) throws IOException {  
        int portDuServeur = 9999;  
  
        DatagramSocket socketVersLeServeur = new DatagramSocket(portDuServeur);  
  
        byte buffer[] = new byte[100];  
        DatagramPacket reception = new DatagramPacket(buffer, buffer.length);  
        socketVersLeServeur.receive(reception);  
  
        String requete = new String(buffer);  
        System.out.println("Requete : " + requete);  
  
        byte[] messageEnBytes = (new Date()).toString().getBytes();  
        DatagramPacket paquetEnvoie = new DatagramPacket(messageEnBytes, messageEnBytes.length, new  
        InetSocketAddress(reception.getAddress(), reception.getPort()));  
        socketVersLeServeur.send(paquetEnvoie);  
  
        socketVersLeServeur.close();  
    }  
}
```

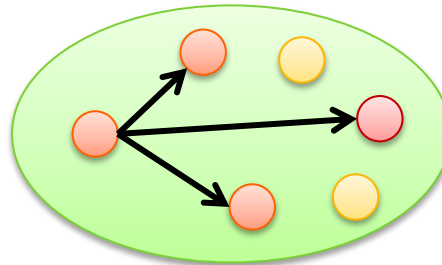




Multicast

Rappel sur le multicast

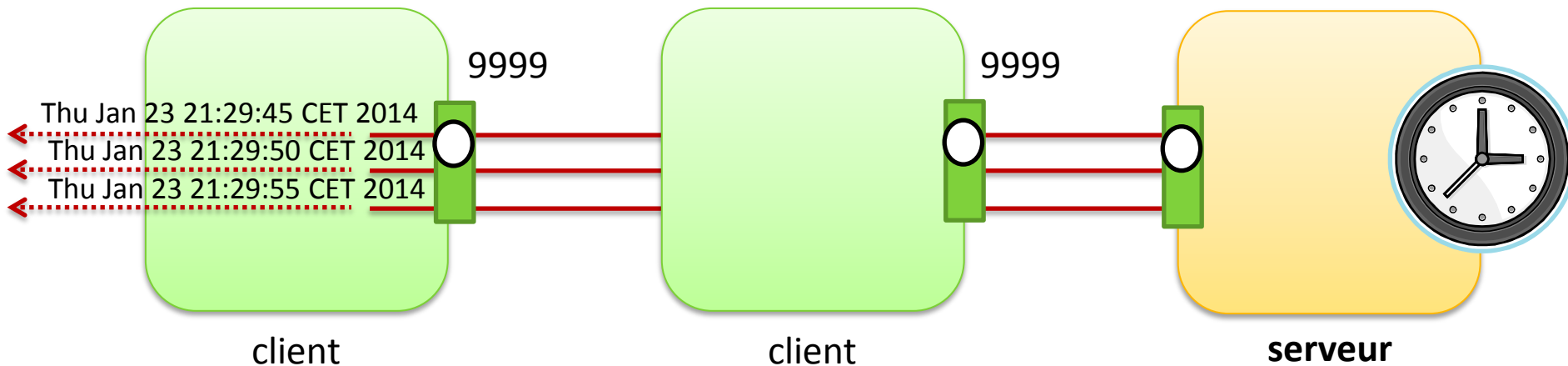
- ▶ Envoie d'un datagramme à plusieurs destinataires
 - ▶ via une adresse un point de connexion multicast :
 - ▶ couple (ip multicast, port)
 - ▶ les utilisateurs doivent écouter cette adresse + point via une socket



- ▶ Utilise le protocole UDP
 - aucune garantie de bonne réception
 - aucune garantie sur l'ordre des messages
 - aucune garantie sur la présence de destinataires en écoute

Exemple : serveur de temps en multicast

- ▶ Le serveur envoie régulièrement (5s), un message sur une adresse multicast 225.0.3.2, port 9999
- ▶ Les clients écoutent ce message en utilisant une **MulticastSocket**
- ▶ Les messages sont envoyés dans des **DatagramPacket**



Les sockets `java.net.MulticastSocket`

- ▶ Sous-classe de `DatagramSocket` qui sert à écouter et envoyer sur un canal multicast.
- ▶ Ne change rien pour l'envoi de données
- ▶ Ajoute des méthodes pour la **réception** :
 - ▶ `void joinGroup(InetAddress groupAddress)` : rejoindre le groupe communicant via l'adresse `groupAdresse`
 - ▶ doit être fait explicitement en multicast
 - ▶ `void leaveGroup(InetAddress groupAddress)` : indique que l'on ne souhaite plus écouter cette adresse

Exemple code du serveur basique

```
public class ServeurTemps {  
  
    public static void main(String[] args) throws IOException, InterruptedException {  
        InetAddress groupeIP = InetAddress.getByName("225.0.3.2");  
        int port = 9999;  
        MulticastSocket socketEmission = new MulticastSocket(port);  
  
        while (true) {  
            byte[] contenuMessage = (new Date()).toString().getBytes();  
            DatagramPacket message;  
            message = new DatagramPacket(contenuMessage, contenuMessage.length, groupeIP, port);  
            socketEmission.send(message);  
            Thread.sleep(1000);  
        }  
    }  
}
```



Exemple code du client basique

```
public class ClientTemps {  
  
    public static void main(String[] args) throws IOException {  
        InetAddress groupeIP = InetAddress.getByName("225.0.3.2");  
        int port = 9999;  
  
        MulticastSocket socketReception = new MulticastSocket(port);  
        socketReception.joinGroup(groupeIP);  
  
        DatagramPacket message;  
        byte[] contenuMessage;  
        String date;  
  
        while (true) {  
            contenuMessage = new byte[1024];  
            message = new DatagramPacket(contenuMessage, contenuMessage.length);  
            socketReception.receive(message);  
            date = new String(contenuMessage);  
            System.out.println(date);  
        }  
    }  
}
```



Codage de caractères

Pb du codage des caractères

- ▶ Il existe un grand nombre de codages utilisés pour les caractères en fonction des symboles utilisés.
- ▶ Java requiert au minimum :

US-ASCII	Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
ISO-8859-1	ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8	Eight-bit UCS Transformation Format
UTF-16BE	Sixteen-bit UCS Transformation Format, big-endian byte order
UTF-16LE	Sixteen-bit UCS Transformation Format, little-endian byte order
UTF-16	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

- ▶ Java utilise Unicode en interne sur 2 octets



Et beaucoup d'autres ...

► Sur ma machine 168 :

Big5,Big5-HKSCS,EUC-JP,EUC-KR,GB18030,GB2312,GBK,IBM-Thai,IBM00858,IBM01140,IBM01141,IBM01142,IBM01143,IBM01144,IBM01145,IBM01146,IBM01147,IBM01148,IBM01149,IBM037,IBM1026,IBM1047,IBM273,IBM277,IBM278,IBM280,IBM284,IBM285,IBM290,IBM297,IBM420,IBM424,IBM437,IBM500,IBM775,IBM850,IBM852,IBM855,IBM857,IBM860,IBM861,IBM862,IBM863,IBM864,IBM865,IBM866,IBM868,IBM869,IBM870,IBM871,IBM918,ISO-2022-CN,ISO-2022-JP,ISO-2022-JP-2,ISO-2022-KR,ISO-8859-1,ISO-8859-13,ISO-8859-15,ISO-8859-2,ISO-8859-3,ISO-8859-4,ISO-8859-5,ISO-8859-6,ISO-8859-7,ISO-8859-8,ISO-8859-9,JIS_X0201,JIS_X0212-1990,KOI8-R,KOI8-U,Shift_JIS,TIS-620,US-ASCII,UTF-16,UTF-16BE,UTF-16LE,UTF-32,UTF-32BE,UTF-32LE,UTF-8,windows-1250,windows-1251,windows-1252,windows-1253,windows-1254,windows-1255,windows-1256,windows-1257,windows-1258,windows-31j,x-Big5-HKSCS-2001,x-Big5-Solaris,x-euc-jp-linux,x-EUC-TW,x-eucJP-Open,x-IBM1006,x-IBM1025,x-IBM1046,x-IBM1097,x-IBM1098,x-IBM1112,x-IBM1122,x-IBM1123,x-IBM1124,x-IBM1364,x-IBM1381,x-IBM1383,x-IBM300,x-IBM33722,x-IBM737,x-IBM833,x-IBM834,x-IBM856,x-IBM874,x-IBM875,x-IBM921,x-IBM922,x-IBM930,x-IBM933,x-IBM935,x-IBM937,x-IBM939,x-IBM942,x-IBM942C,x-IBM943,x-IBM943C,x-IBM948,x-IBM949,x-IBM949C,x-IBM950,x-IBM964,x-IBM970,x-ISCII91,x-ISO-2022-CN-CNS,x-ISO-2022-CN-GB,x-iso-8859-11,x-JIS0208,x-JISAutoDetect,x-Johab,x-MacArabic,x-MacCentralEurope,x-MacCroatian,x-MacCyrillic,x-MacDingbat,x-MacGreek,x-MacHebrew,x-MacIceland,x-MacRoman,x-MacRomania,x-MacSymbol,x-MacThai,x-MacTurkish,x-MacUkraine,x-MS932_0213,x-MS950-HKSCS,x-MS950-HKSCS-XP,x-mswin-936,x-PCK,x-SJIS_0213,x-UTF-16LE-BOM,x-UTF-32BE-BOM,x-UTF-32LE-BOM,x-windows-50220,x-windows-50221,x-windows-874,x-windows-949,x-windows-950,x-windows-iso2022jp

Exemple de problèmes

► Sur ma machine (windows-1252):

```
String unicode = "Heureux soient les fêlés, car ils laisseront passer la lumière.";
System.out.println("Default charset " + Charset.defaultCharset());
for (String charsetName : new String[] { "UTF-8", "US-ASCII", "UTF-16", "ISO-8859-3", Charset.defaultCharset().name() })
{
    InputStream in = new ByteArrayInputStream(unicode.getBytes());
    BufferedReader reader = new BufferedReader(new InputStreamReader(in, charsetName));
    System.out.println(charsetName + ": " + reader.readLine());
}
```

Default charset windows-1252

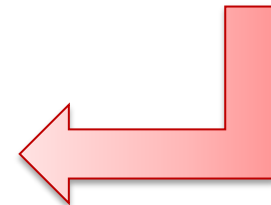
UTF-8: Heureux soient les f?l?s, car ils laisseront passer la lumi?re.

US-ASCII: Heureux soient les f?l?s, car ils laisseront passer la lumi?re.

UTF-16: ?????????????????????????????????

ISO-8859-3: Heureux soient les fêlés, car ils laisseront passer la lumière.

windows-1252: Heureux soient les fêlés, car ils laisseront passer la lumière.



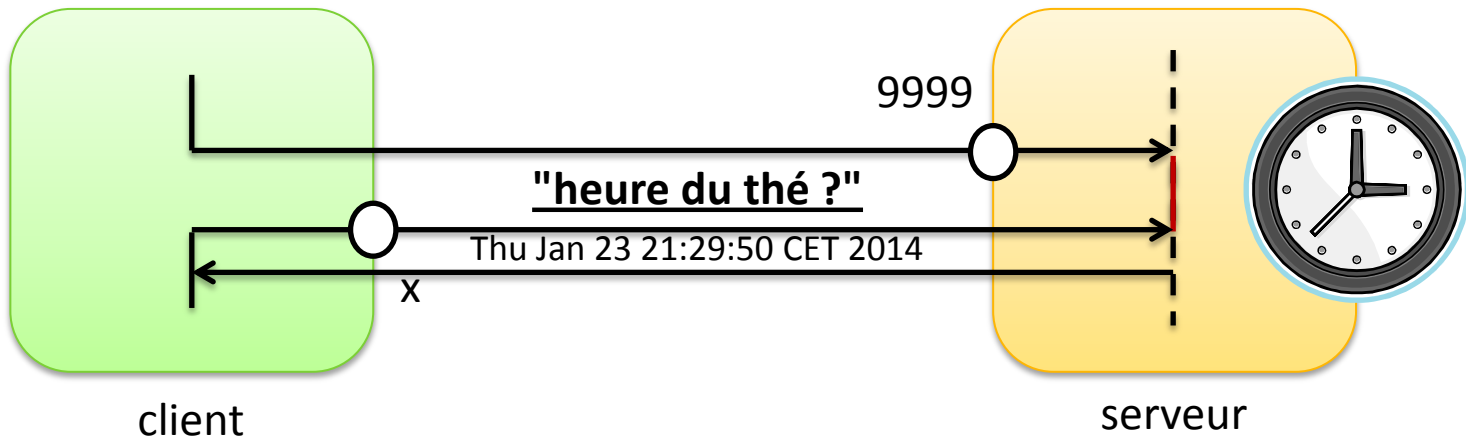
Reader/Writer

- ▶ On peut déclarer un charset sur les Reader et les Writers :
 - ▶ **InputStreamReader**(InputStream, "charset");
 - ▶ **OutputStreamWriter**(OutputStream, "charset");

```
String unicode = "Heureux soient les fêlés, car ils laisseront passer la lumière.";
System.out.println("Default charset " + Charset.defaultCharset());
for (String charsetName : new String[] { "UTF-8", "US-ASCII", "UTF-16", "ISO-8859-3", Charset.defaultCharset().name() })
{
    InputStream in = new ByteArrayInputStream(unicode.getBytes());
    BufferedReader reader = new BufferedReader(new InputStreamReader(in, charsetName));
    System.out.println(charsetName + ": " + reader.readLine());
}
```

Exemple Client/Serveur

- Le client et le serveur utilise UTF-8



Code client

```
public static void main(String[] args) throws IOException {
    int portDuServeur = 8787;
    String adresseDuServeur = "localhost";
    Socket socketVersLeServeur = new Socket(adresseDuServeur, portDuServeur);

    PrintWriter out = new PrintWriter(new OutputStreamWriter(socketVersLeServeur.getOutputStream(), "UTF-8"));
    out.println("heure du thé ?");
    out.flush();

    BufferedReader in = new BufferedReader(new InputStreamReader(socketVersLeServeur.getInputStream(), "UTF-8"));
    String date;
    while ((date = in.readLine()) != null) {
        System.out.println(date);
    }

    socketVersLeServeur.close();
}
```



Code serveur

```
public class ServeurTemps {
    public static void main(String[] args) throws IOException, InterruptedException {
        int portEcoule = 8787;
        ServerSocket socketServeur = new ServerSocket(portEcoule);
        try {
            while (true) {

                try (Socket socketVersUnClient = socketServeur.accept()) {

                    BufferedReader in = new BufferedReader(new InputStreamReader(socketVersUnClient.getInputStream(), "UTF-8"));
                    String requete = in.readLine();

                    System.out.println("requete is " + requete);

                    PrintWriter out = new PrintWriter(new OutputStreamWriter(socketVersUnClient.getOutputStream(), "UTF-8"));

                    if (requete.equals("heure du thé ?")) {
                        for (int i = 0; i < 100; i++) {
                            Date date = new Date();
                            out.println(date.toString());
                            out.flush();
                            Thread.sleep(5000);
                        }
                    } else {
                        out.println("Bad request");
                        out.flush();
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        } finally {
            socketServeur.close();
        }
    }
}
```

