Le modèle client-serveur

Introduction

Christian Bulfone
Jean Michel Adam

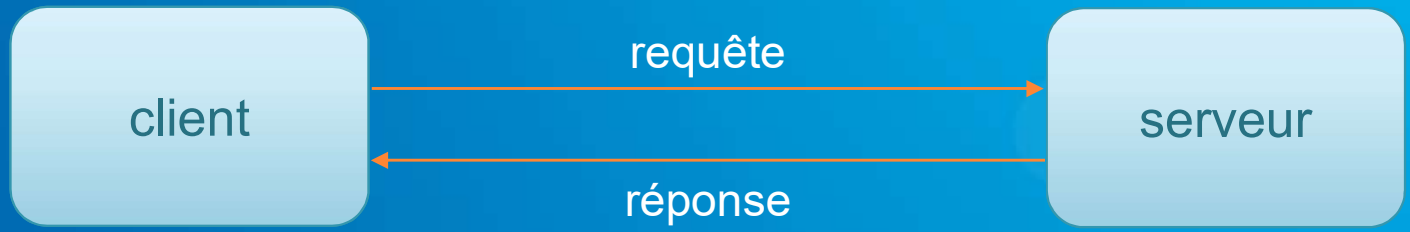
L3 MIASHS

Principe du client / serveur

- Repose sur une communication d'égal à égal entre les applications
  - Communication réalisée par dialogue entre processus deux à deux
    - un processus client
    - un processus serveur
- Les processus ne sont pas identiques mais forment plutôt un système coopératif se traduisant par un échange de données
  - le client réceptionne les résultats finaux délivrés par le serveur
- Le client initie l'échange
- Le serveur est à l'écoute d'une requête cliente éventuelle
- Le service rendu = traitement effectué par le serveur

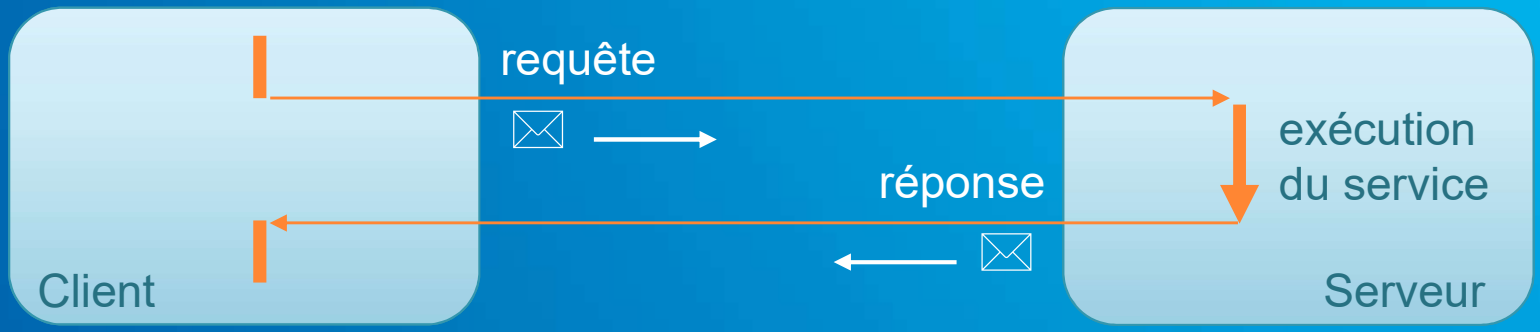
Le modèle client / serveur

- Le *client* demande l'exécution d'un service
- Le *serveur* réalise le service
- Client et serveur sont généralement localisés sur deux machines distinctes (il s'agit parfois de la même machine)



Le modèle client / serveur

- Communication par **messages**
  - Requête : paramètres d'appel, spécification du service requis
  - Réponse : résultats, indicateur éventuel d'exécution ou d'erreur
  - Communication synchrone (dans le modèle de base) : le client est bloqué en attente de la réponse



Gestion des processus

- Client et serveur exécutent des processus distincts
  - Le client est suspendu lors de l'exécution de la requête (appel synchrone)
  - Plusieurs requêtes peuvent être traitées par le serveur
    - → mode itératif
    - → mode concurrent

Gestion des processus

- Mode de gestion des requêtes
  - → itératif
    - le processus serveur traite les requêtes les unes après les autres
  - → concurrent basé sur
    - parallélisme réel
      - système multiprocesseurs par exemple
    - pseudo-parallélisme
      - schéma veilleur-exécutants
    - la concurrence peut prendre plusieurs formes
      - plusieurs processus (une mémoire virtuelle par processus)
      - plusieurs processus légers (*threads*) dans le même espace virtuel

Gestion des processus dans le serveur

- Processus serveur unique

```
while (true) {
  receive (client_id, message)
  extract (message, service_id, params)
  do_service[service_id] (params, results)
  send (client_id, results)
}
```

Gestion des processus dans le serveur

- Schéma veilleur-exécutants

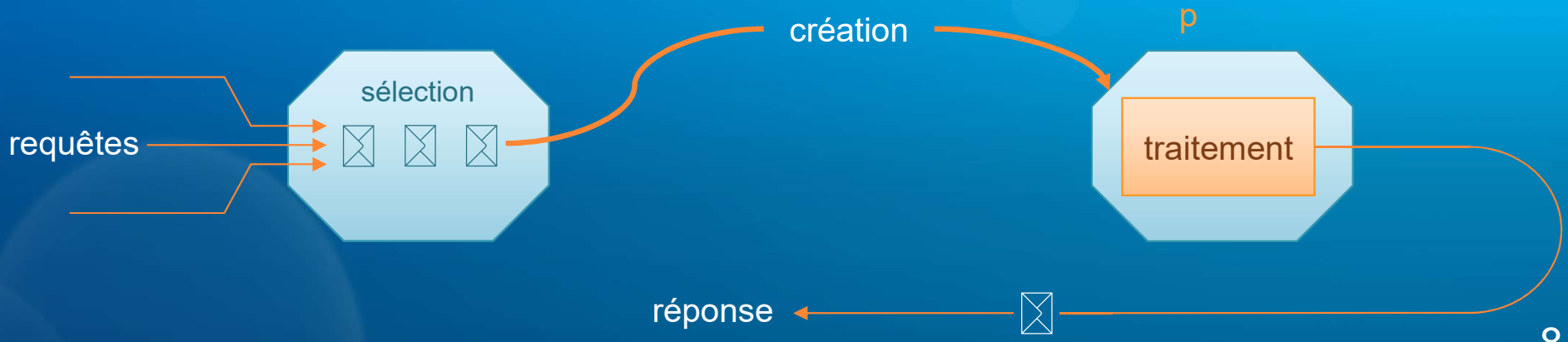
Processus veilleur

```
while (true) {
  receive (client_id, message)
  extract (message, service_id, params)
  p = create_thread (client_id, service_id, params)
}
```

Création dynamique des exécutants

programme de p

```
do_service[service_id] (params, results)
send (client_id, results)
exit
```



Mise en œuvre du modèle client / serveur

- Besoin d'un support pour transporter les informations entre le client et le serveur
  - Bas niveau
    - Utilisation directe du transport : *sockets* (construits sur TCP ou UDP)
  - Haut niveau
    - Intégration dans le langage de programmation : RPC ou *Remote Procedure Call* (construits sur sockets)
- Nécessité d'établir un protocole entre le client et le serveur pour qu'ils se comprennent

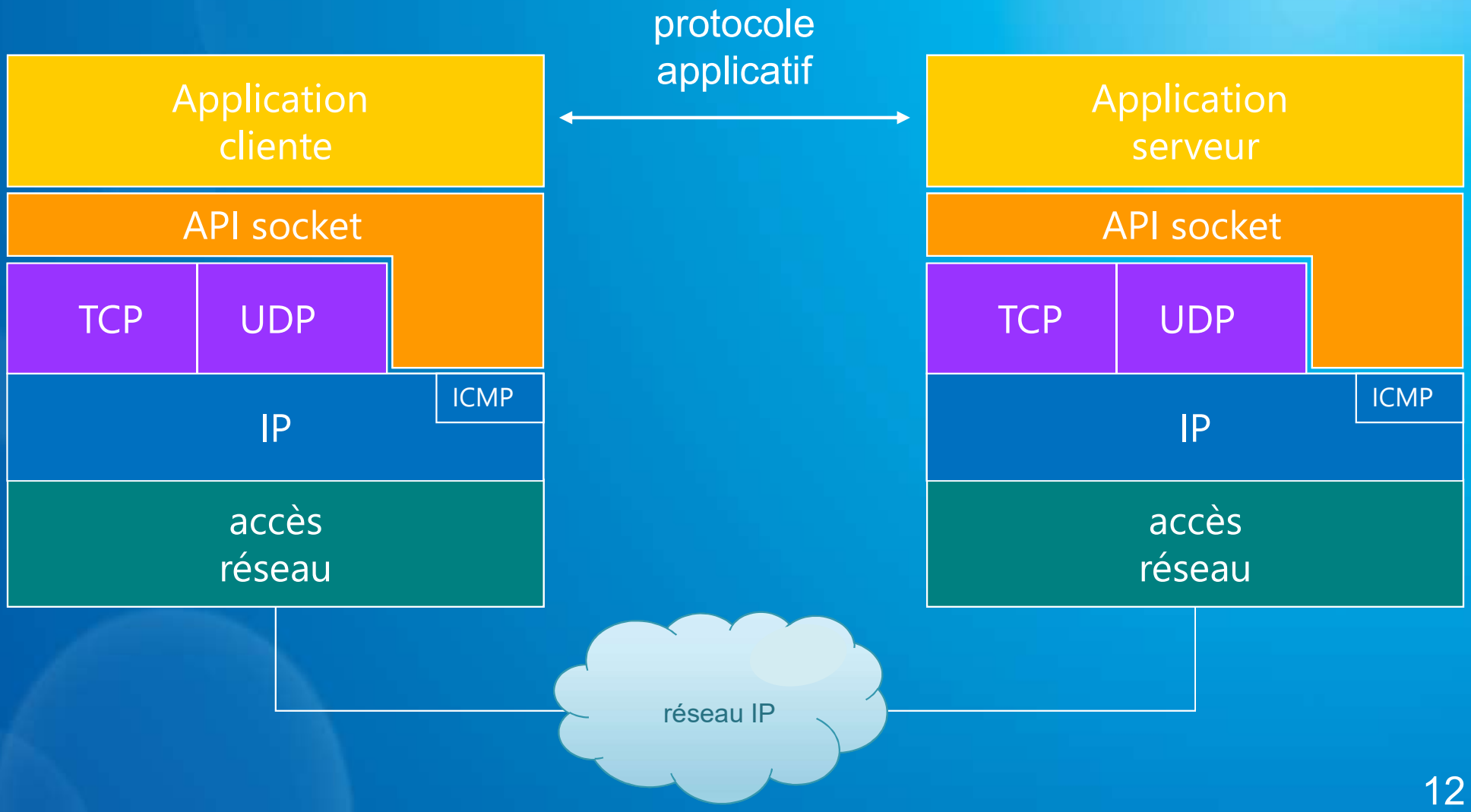
Les protocoles applicatifs

- Le protocole applicatif définit
  - Le format des messages échangés entre émetteur et récepteur (textuel, binaire, …)
  - Les types de messages : requête / réponse / informationnel …
  - L'ordre d'envoi des messages
- Ne pas confondre protocole et application
  - Une application peut supporter plusieurs protocoles (ex : logiciel de messagerie supportant POP, IMAP et SMTP)
  - Navigateur et serveur Web s'échangent des documents HTML en utilisant le protocole HTTP

Les sockets

- API (*Application Program Interface*) socket
  - mécanisme d'interface de programmation des sockets
    - permet aux programmes d'échanger des données
    - les applications client/serveur ne voient les couches de communication qu'à travers l'API socket (abstraction)
    - n'implique pas forcément une communication par le réseau
  - le terme « socket » signifie douille, prise électrique femelle, bref ce sur quoi on branche quelque chose
- L'API socket s'approche de l'API fichier d'Unix
- Développée à l'origine dans Unix BSD (*Berkeley Software Distribution*)

L'API socket


protocole
applicatif
Application
cliente
Application
serveur
API socket
API socket
TCP
UDP
TCP
UDP
ICMP
ICMP
IP
IP
accès
réseau
accès
réseau
réseau IP

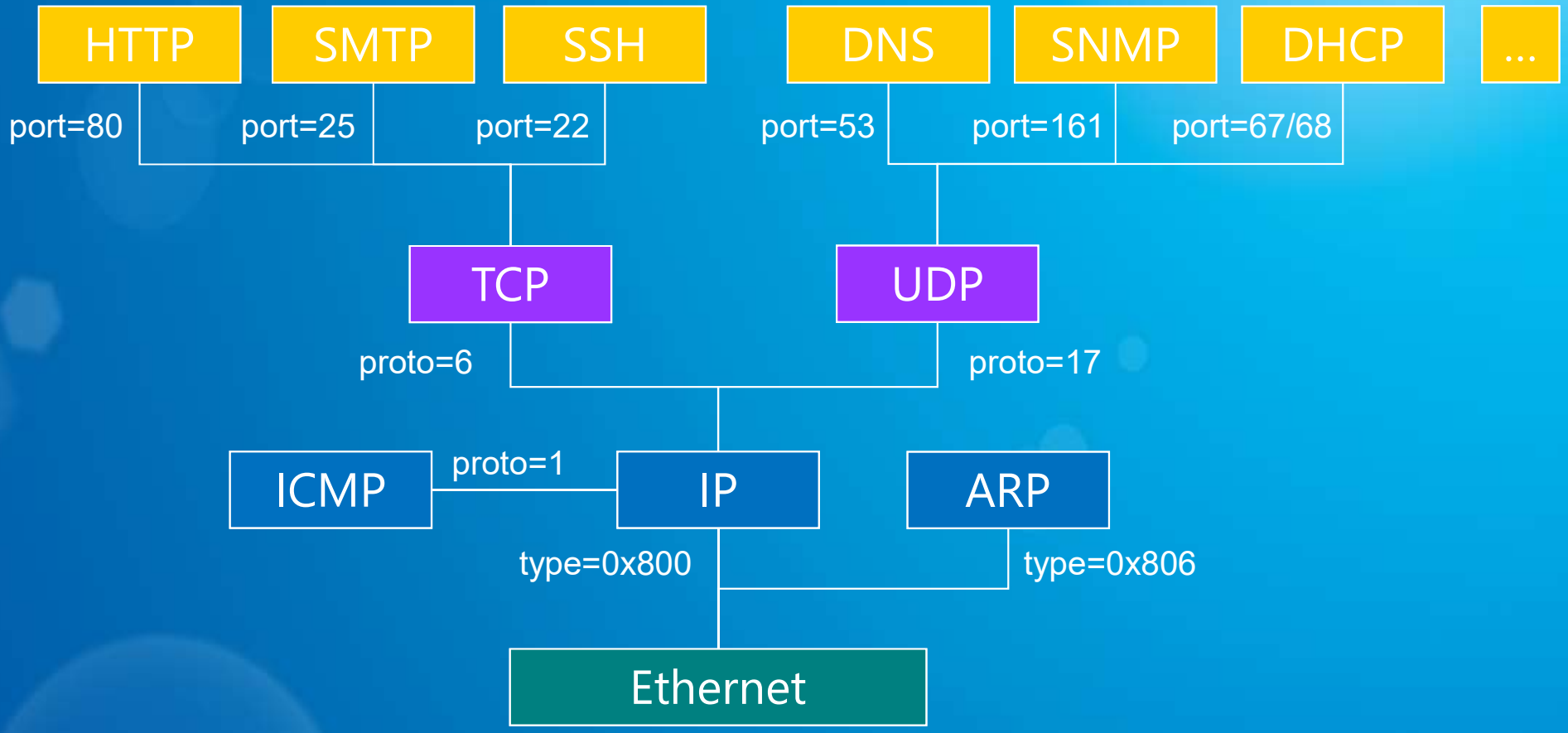
Les sockets

- Avec les protocoles UDP et TCP, une connexion est entièrement définie sur chaque machine par :
  - le type de protocole (UDP ou TCP)
  - l'adresse IP
  - le numéro de port associé au processus
    - serveur : port local sur lequel les connexions sont attendues
    - client : allocation dynamique par le système

Notion de port

- Un service rendu par un programme serveur sur une machine est accessible par un port
- Un port est identifié par un entier (16 bits)
  - de 0 à 1023
    - ports **reconnus** ou **réservés**
    - sont assignés par l'IANA (*Internet Assigned Numbers Authority*)
    - donnent accès aux services standard : courrier (SMTP port 25), serveur web (HTTP port 80) …
  - > 1024
    - ports « **utilisateurs** » disponibles pour placer un service applicatif quelconque
- Un service est souvent connu par un nom (FTP, ...)
  - La correspondance entre nom et numéro de port est donnée par le fichier **`/etc/services`**

Identification des protocoles



Les sockets

Couches Session à Application

n° port : 2345
@ IP : 192.168.20.1

Couche Transport (TCP, UDP)

n° port : 80
@ IP : 192.168.20.2

@ IP : 192.168.20.1

Couche Réseau (IP)

@ IP : 192.168.20.2

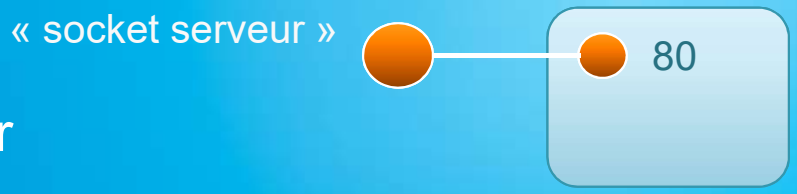
@ Ethernet

Couche Liaison

@ Ethernet

Principes de fonctionnement (mode concurrent)

1. Le serveur crée une « socket serveur » (associée à un port) et se met en attente

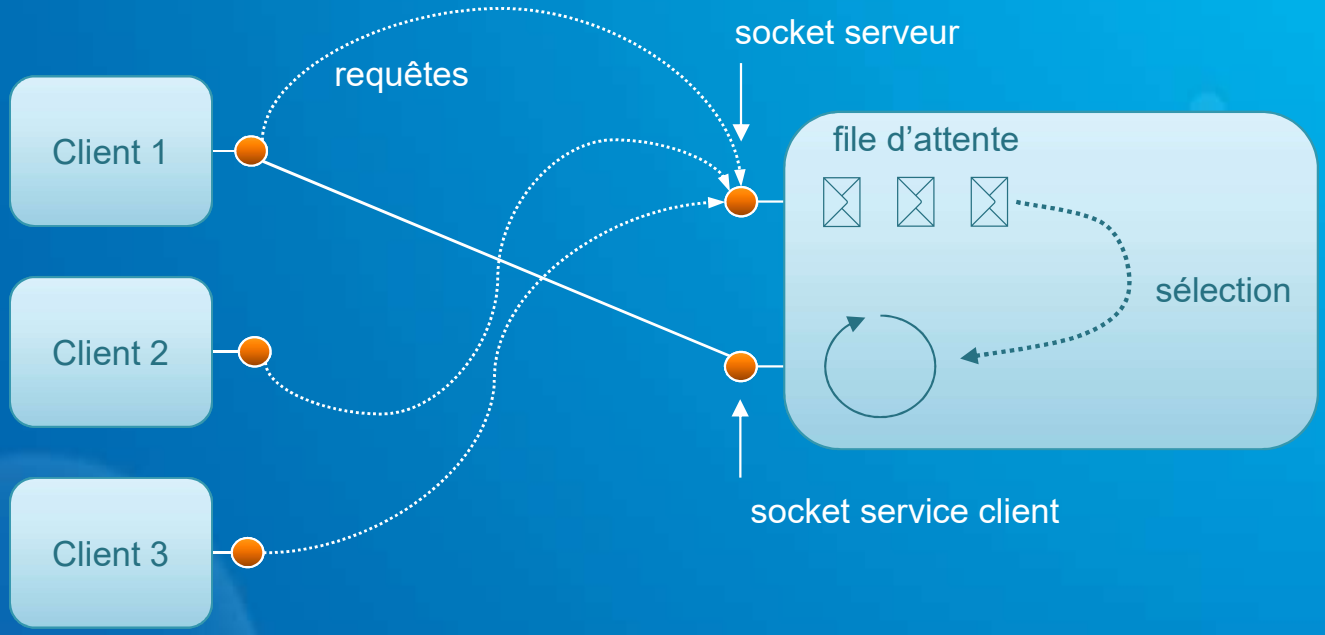


2. Le client se connecte à la socket serveur
   - Deux sockets sont alors créés
     - Une « socket client » côté client
     - Une « socket service client » côté serveur
   - Ces sockets sont connectées entre elles

3. Le client et le serveur communiquent par les sockets
   - L'interface est celle des fichiers (read, write)
   - La socket serveur peut accepter de nouvelles connexions

Gestion du parallèlisme sur le serveur

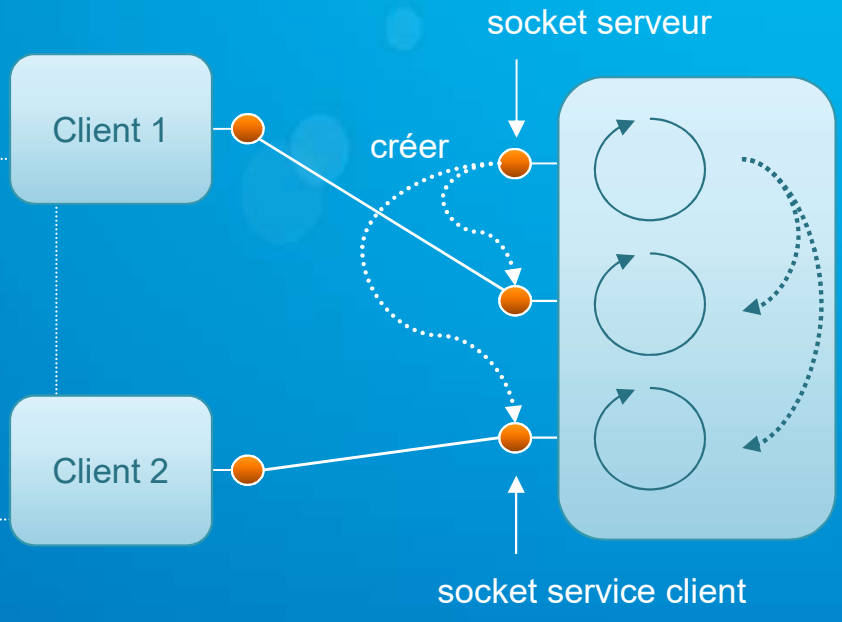
- Utilisation d'**un seul** processus serveur
  - Les clients multiples sont traités en séquence
    - les requêtes sont conservées dans une file d'attente associée à la socket serveur
    - La longueur maximale de cette file d'attente peut être spécifiée lors de la création de la socket serveur (en Java)



Gestion du parallèlisme sur le serveur

- Utilisation du schéma **veilleur-exécutants**
  - Le *thread* veilleur doit créer explicitement un nouveau *thread* exécutant à chaque nouvelle connexion d'un client (donc à l'exécution de accept())
  - Une « socket service client » différente est créée pour chaque client
  - Le *thread* veilleur se remet ensuite en attente

```
while (true) {
  accepter la connexion d'un nouveau client et créer
  une socket service client;
  créer un thread pour interagir avec ce client sur la
  nouvelle socket service client;
}
```



Mode connecté / non connecté

- Deux réalisations possibles
  - Mode connecté (protocole TCP)
  - Mode non connecté (protocole UDP)

- Mode connecté 
  - Ouverture d'une liaison, suite d'échanges, fermeture de la liaison
  - Le serveur préserve son état entre deux requêtes
  - Garanties de TCP : ordre, contrôle de flux, fiabilité
  - Adapté aux échanges ayant une certaine durée (plusieurs messages)

Mode connecté / non connecté

- Mode non connecté 
  - Les requêtes successives sont indépendantes
  - Pas de préservation de l'état du serveur entre les requêtes
  - Le client doit indiquer son adresse à chaque requête (pas de liaison permanente)
  - Pas de garanties particulières (UDP)
    - gestion de toutes les erreurs à la main : il faut réécrire la couche transport !!!
  - Adapté
    - aux échanges brefs (réponse en 1 message)
    - pour faire de la diffusion

Mode connecté / non connecté

- Points communs
  - Le client a l'initiative de la communication
    - le serveur doit être à l'écoute
  - Le client doit connaître la référence du serveur (adresse IP, numéro de port)
    - en la trouvant dans un annuaire, si le serveur l'y a enregistrée au préalable
    - en utilisant les numéros de ports préaffectés par convention
  - Le serveur peut servir plusieurs clients

Utilisation du mode connecté

- Caractéristiques
  - Établissement préalable d'une connexion (circuit virtuel) : le client demande au serveur s'il accepte la connexion
  - Fiabilité assurée par le protocole (TCP)
  - Mode d'échange par **flots d'octets** : le récepteur n'a pas connaissance du découpage des données effectué par l'émetteur
  - Possibilité d'émettre et de recevoir des caractères urgents
  - Après initialisation, le serveur est « passif » : il est activé lors de l'arrivée d'une demande de connexion du client
  - Un serveur peut répondre aux demandes de service de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente
- Contraintes
  - Le client doit avoir accès à l'adresse du serveur (adresse IP, numéro de port)

Utilisation du mode non connecté

- Caractéristiques
  - Pas d'établissement préalable d'une connexion
  - Pas de garantie de fiabilité
  - Adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (1 message)
  - Le récepteur reçoit les données selon le découpage effectué par l'émetteur
- Contraintes
  - Le client doit avoir accès à l'adresse du serveur (adresse IP, numéro de port)
  - Le serveur doit récupérer l'adresse de chaque client pour lui répondre

Généralisation du schéma client / serveur

- Les notions de client et de serveur sont relatives
  - Un serveur peut faire appel à d'autres serveurs dont il est le client
    - Architecture à 3 niveaux (3 tiers)
    - Exemple : un serveur Web faisant appel à un serveur de base de données
  - Clients et serveurs jouent parfois un rôle symétrique
    - Exemple : postes des réseaux Microsoft Windows
  - Systèmes de pair à pair (Peer to Peer, P2P)

L'API Socket en Java

L'API socket en Java

- L'interface Java des sockets (package java.net) offre un accès simple aux sockets sur IP
- Plusieurs classes interviennent lors de la réalisation d'une communication par sockets
  - La classe java.net.InetAddress permet de manipuler des adresses IP
  - Mode connecté (TCP)
    - La classe java.net.SocketServer permet de programmer l'interface côté serveur
    - La classe java.net.Socket permet de programmer l'interface côté client et la communication effective par flot d'octets
  - Mode non connecté (UDP)
    - Les classes java.net.DatagramSocket et java.net.DatagramPacket permettent de programmer la communication en mode datagramme

Classe java.net.InetAddress

- Conversion de **nom** vers **adresse IP**
  - méthodes permettant de créer des objets adresses IP
    - `InetAddress InetAddress.getLocalHost()` pour obtenir une *InetAddress* de l'hôte
    - `InetAddress InetAddress.getByName(String id)` pour obtenir une *InetAddress* d'une machine donnée
    - `InetAddress[ ] InetAddress.getAllByName(String id)` pour obtenir toutes les *InetAddress* d'un site
  - Le paramètre `id` peut être un nom de machine ("*prevert.upmf-grenoble.fr*") ou un numéro IP ("195.221.42.159")
  - Ces 3 fonctions peuvent lever une exception `UnknownHostException`

Classe java.net.InetAddress

- Conversions inverses
  - Des méthodes applicables à un objet de la classe InetAddress permettent d'obtenir dans divers formats des adresses IP ou des noms de site
    - `String getHostName()` obtient le nom complet correspondant à l'adresse IP
    - `String getHostAddress()` obtient l'adresse IP sous forme “%d.%d.%d.%d”
    - `byte[] getAddress()` obtient l'adresse IP sous forme d'un tableau d'octets

Exemples

Obtenir le nom et le numéro IP de la machine sur laquelle on travaille :

```
try{
   InetAddress  monAdresse = InetAddress.getLocalHost();
   System.out.println(monAdresse.getHostName());
   System.out.println(monAdresse.getHostAddress());
} catch(UnknownHostException uhe){
   System.out.println("Inconnu !");
}
```

Obtenir le nom et le numéro IP d'une autre machine :

```
try{
   InetAddress  uneAdresse = InetAddress.getByName(nom);
   System.out.println(uneAdresse.getHostName());
   System.out.println(uneAdresse.getHostAddress());
} catch(UnknownHostException uhe){
   System.out.println("Inconnu !");
}
```

Classe java.net.ServerSocket

- Cette classe implante un objet ayant un comportement de serveur via une interface par socket
- Constructeurs
  - `ServerSocket(int port)`
  - `ServerSocket(int port, int backlog)`
  - `ServerSocket(int port, int backlog, InetAddress bindAddr)`
    - Créent un objet serveur à l'écoute du `port` spécifié
    - La taille de la file d'attente des demandes de connexion peut être explicitement spécifiée via le paramètre `backlog`
    - Si la machine possède plusieurs adresses, on peut aussi restreindre l'adresse sur laquelle on accepte les connexions

Classe java.net.ServerSocket

- Méthodes
  - `Socket accept()` accepte une connexion entrante
    - Méthode bloquante, mais dont l'attente peut être limitée dans le temps par l'appel préalable de la méthode `setSoTimeout` (voir plus loin)
  - `void close()` ferme la socket
  - `InetAddress getInetAddress()` retourne l'adresse du serveur
  - `int getLocalPort()` retourne le port sur lequel le serveur écoute

Classe java.net.ServerSocket

- Une **`ServerSocket`** serveur permet à des clients de se connecter, la connexion étant ensuite gérée par une **`Socket`**

```
// création d'un socket serveur sur le port port
ServerSocket serveurSocket = new ServerSocket(port);
try{
      while(true){
            // Attente de la connexion d'un client
            Socket clientServiceSocket = serveurSocket.accept();
            // et création du socket clientServiceSocket si
            // acceptation de la connexion
          …
}
catch(IOException ioe)          { . . . }
```

Classe java.net.Socket

- Cette classe est utilisée pour la programmation des sockets connectées, côté client et côté serveur
- Constructeurs
  - `Socket(String host, int port)`
    - Construit une nouvelle socket en tentant une connexion à la machine hôte `host` sur le port `port`
    - Levée d'exception :
      - `UnknownHostException` si la machine hôte n'existe pas
      - `IOException` s'il n'y a pas d'application serveur démarrée sur le port `port`

Classe java.net.Socket

- `Socket(InetAddress address, int p)`
  - Construit une nouvelle socket en tentant une connexion à la machine hôte d'adresse `address` sur le port `p`
  - Levée d'exception :
    - `IOException` s'il n'y a pas d'application serveur démarrée sur le port `p`
- Deux autres constructeurs permettent en outre de fixer l'adresse IP et le numéro de port utilisés côté client (plutôt que d'utiliser un port disponible quelconque)
  - `Socket(String host, int port, InetAddress localAddr, int localPort)`
  - `Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)`
- Note : tous ces constructeurs correspondent à l'utilisation des primitives systèmes **`socket()`**, **`bind()`** (éventuellement) et **`connect()`**

`bind()`	définit le port local
`connect()`	définit le port où l'on se connecte

Classe java.net.Socket

- Méthodes
  - `void close()` fermeture de la *socket* (peut lever une exception `IOException`)
  - `void shutDownInput()` fermeture de la *socket* pour la lecture (peut lever une exception `IOException`)
  - `void shudownOutput()` fermeture de la *socket* pour l'écriture (peut lever une exception `IOException`)
  - `InetAddress getLocalAddress()` : adresse de la machine hôte
  - `InetAddress getInetAddress()` : adresse de la machine à laquelle on est connecté
  - `int getLocalPort()` : le port local
  - `int getPort()` : le port sur la machine distante

Classe java.net.Socket : lecture

- La communication effective sur une connexion par socket utilise la notion de flots de données (`java.io.OutputStream` et `java.io.InputStream`)
- Méthode utilisée pour obtenir les flots en entrée (lecture)
  - `InputStream getInputStream()`
  - `InputStream` doit être « habillé » par :
    - `DataInputStream`
    - `InputStreamReader`
- Exemples

```
Socket socket …
DataInputStream entree = new DataInputStream(socket.getInputStream());
String chaine = entree.readUTF();
```

ou

```
BufferedReader entree =
new BufferedReader(new InputStreamReader(socket.getInputStream()) );
String chaine = entree.readLine();
```

Classe java.net.Socket : écriture

- Méthode utilisée pour obtenir les flots en sortie (écriture)
  - `OutputStream getOutputStream()`
  - `OutputStream` doit être « habillé » par :
    - `DataOutputStream`
    - `PrinterWriter`

- Exemples

```
Socket socket …
DataOutputStream sortie =
                        new DataOutputStream(socket.getOutputStream());
sortie.writeUTF(chaine);
```

ou

```
PrinterWriter sortie = new PrinterWriter (socket.getOutputStream());
sortie.println(chaine);
```

- Remarque
  - les opérations sur les sockets et les flots sont bloquantes : l'application est arrêtée tant que l'opération n'est pas terminée (⇒ utilisation des *threads*)

Socket Options

- **TCP_NODELAY** :
  - `void setTcpNoDelay(boolean b)` et `boolean getTcpNoDelay()`
    - vrai : implique que les paquets sont envoyés sans délai
    - faux :
      - les petits paquets sont regroupés avant d'être envoyés
      - le système attend de recevoir le message d'acquittement du paquet précédent, avant d'envoyer le paquet suivant (algorithme de Nagle)
- **SO_LINGER** :
  - `void setSoLinger(boolean b, int s)` et `int getSoLinger ()`
    - ce paramètre indique ce qu'il faut faire des paquets non encore envoyés lorsque la socket est fermée
    - faux : la socket est fermée immédiatement, et les paquets non envoyés perdus
    - vrai : la fermeture de la socket bloque pendant `s` secondes, pendant lesquelles les données peuvent être envoyées, et les acquittements reçus

Socket Options

- `SO_TIMEOUT` :
  - `void setSoTimeout(int timeout)` et `int getSoTimeout()`
    - Prend en paramètre le délai de garde exprimé en millisecondes
    - La valeur par défaut 0 équivaut à l'infini
    - À l'expiration du délai de garde, l'exception `InterruptedIOException` est levée
- `SO_SNDBUF` :
  - `void setSendBufferSize(int s)` et `int getSendBufferSize()`
- `SO_RCVBUF` :
  - `void setReceiveBufferSize(int s)` et `int getreceiveBufferSize()`
- permettent d'ajuster la taille des tampons pour augmenter les performances ; on utilisera des tampons plus grands si :
  - le réseau est rapide
  - de gros blocs de données sont transférés (FTP, HTTP)

Socket Options

- **`SO_KEEPALIVE`** :
  - **`void setKeepAlive(boolean b)`** et **`boolean getKeepAlive()`**
  - quand l'option est mise à vrai : s'il n'y a pas eu de transfert de données sur la socket depuis 2 heures (en général), TCP envoie un paquet au partenaire. 3 possibilités :
    - Le partenaire répond par un message d'acquittement (ACK), tout est OK
    - Le partenaire répond avec un RST ; le partenaire a eu une interruption, suivi d'un redémarrage: la socket est fermée
    - Pas de réponse du partenaire : la socket est fermée

Client en mode connecté

```
BufferedReader entree;

try{
      clientSocket = new Socket(hote, port);
      entree = new BufferedReader(new InputStreamReader (
                              clientSocket.getInputStream () ));
      while (true){
            String message = entree.readLine();
            System.out.println(message);
      }
}catch(UnknownHostException uhe){
      System.out.println("UnknownHostException");
}catch(IOException ioe){
      System.out.println("IOException");
}
```

Client / serveur en mode connecté (très simplifié)

Sur un client

doit être connu du client

```
clientSocket = new Socket("miashs-dc.u-ga.fr", 8254);
```

Sur le serveur (miashs-dc.a-ga.fr)

```
serveurSocket = new ServerSocket(8254);
clientServiceSocket = serveurSocket.accept();
```

Le client et le serveur sont à présent connectés

```
PrintWriter sortie = new PrintWriter (
                clientSocket.getOutputStream(), true);
BufferedReader entree = new BufferedReader (
                new InputStreamReader(
                  clientSocket.getInputStream()));
```

```
PrintWriter sortie = new printWriter (
                clientServiceSocket.getOutputStream(), true);
BufferedReader entree = new BufferedReader (
                new InputStreamReader(
                  clientServiceSocket.getInputStream()));
```

Le client et le serveur peuvent maintenant communiquer

```
BufferedReader clavier = new BufferedReader(
                new InputStreamReader(System.in));

String requete, reponse;

while (true) {
  requete = clavier.readLine(); // utilisateur entre la requête
  sortie.println(requete); // envoyer la requête au serveur
  reponse = entree.readLine(); // attendre la réponse
  System.out.println(reponse); // afficher la réponse
}
```

```
String requete;
String reponse;

while (true) {
  requete = entree.readLine();
  // exécuter le service
  // envoyer la réponse au client
  sortie.println(reponse);
}
```

Un serveur concurrent en mode connecté

Programme des exécutants

```
public class Service extends Thread {
Socket clientServiceSocket;
   String requete, reponse;

   public Service(Socket socket) {
      clientServiceSocket = socket;
   }

   public void run() {
      PrintWriter sortie = new printWriter (
                  clientServiceSocket.getOutputStream(), true);
      BufferedReader entree = new BufferedReader (
                        new InputStreamReader(
                           clientServiceSocket.getInputStream()));

       try {
          requete = entree.readLine();
          // exécuter le service
          // envoyer la réponse au client
          sortie.println(reponse);
        } finally {
            clientServiceSocket.close();
        }
    }
}
```

Programme du veilleur

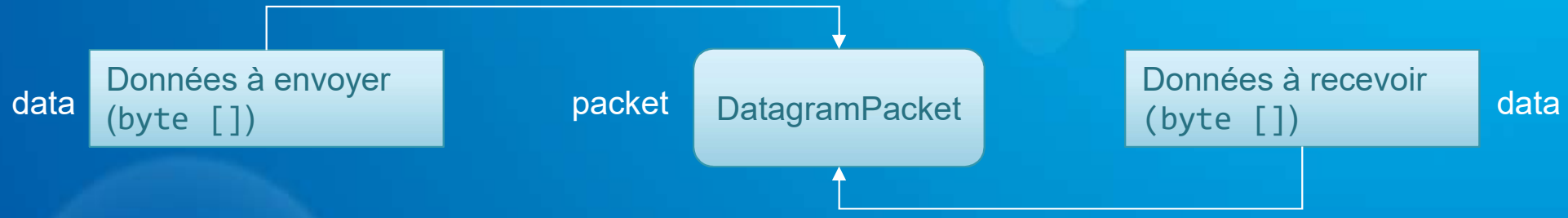
```
serveurSocket = new ServerSocket(8254);

while (true) {
  Socket clientServiceSocket = serveurSocket.accept();

  Service monService = new Service(clientServiceSocket);
  // crée un nouveau thread pour le nouveau client
  monService.start();
  // lance l'exécution du thread
}
```

Le programme du client est inchangé

Sockets en mode non connecté

- Deux classes
  - **`DatagramSocket`** : un seul type de socket
  - **`DatagramPacket`** : format de message pour UDP
    - Conversion entre données et paquets (dans les 2 sens)
    - Les DatagramPacket sont construits différemment pour l'envoi et la réception

```
packet = new DatagramPacket(data, data.length, adresseIP, port)
```




```
packet = new DatagramPacket(data, data.length)
```

Classe java.net.DatagramSocket

- Cette classe permet d'envoyer et de recevoir des paquets (UDP)
- Constructeurs
  - `DatagramSocket()`
  - `DatagramSocket(int port)`
  - Construit une socket datagramme en spécifiant éventuellement un port sur la machine locale (par défaut, un port disponible quelconque est choisi)
  - Levée éventuelle de `SocketException`

Classe java.net.DatagramSocket

- Emission
  - `void send(DatagramPacket p)`
    - Levée éventuelle de `IOException`
- Réception
  - `void receive(DatagramPacket p)`
- Ces opérations permettent d'envoyer et de recevoir un paquet
  - Un paquet est un objet de la classe `java.net.DatagramPacket` qui possède une zone de données et (éventuellement) une adresse IP et un numéro de port (destinataire dans le cas *send*, émetteur dans le cas *receive*)

Classe java.net.DatagramSocket

- `void close()` : fermeture de la socket ; penser à toujours fermer les sockets qui ne sont plus utilisées
- `int getLocalPort()` : retourne le port sur lequel on écoute, ou le port anonyme sur lequel on a envoyé le paquet
- Il est possible de « connecter » une socket datagramme à un destinataire
  - `void connect(InetAddress ia, int p)`
  - Dans ce cas, les paquets émis sur la socket seront toujours pour l'adresse spécifiée
  - La connexion simplifie l'envoi d'une série de paquets (il n'est plus nécessaire de spécifier l'adresse de destination pour chacun d'entre eux) et accélère les contrôles de sécurité (ils ont lieu une fois pour toute à la connexion)
  - La « déconnexion » avec `void disConnect()` défait l'association (la socket redevient disponible comme dans l'état initial)
  - `InetAddress getInetAddress()` : retourne l'adresse à laquelle on est connecté
  - `int getPort()` : retourne le port auquel on est connecté

Classe java.net.DatagramPacket

- Emission
  - `DatagramPacket (byte[] buf, int length, InetAddress ia, int port)`
    - construit un datagramme destiné à être envoyé à l'adresse `ia`, sur le port `port`
    - le tampon `buf` de longueur `length` contient les octets à envoyer
- Réception
  - `DatagramPacket (byte[] buf, int length)`
    - construit un datagramme destiné à recevoir des données
    - le tampon `buf` de longueur `length` contient les octets à recevoir

Classe java.net.DatagramPacket

- `InetAddress getAddress()` : retourne l'adresse de l'hôte d'où vient ou où va le paquet
- `void setAddress(InetAddress ia)` : change l'adresse où envoyer le paquet
- `int getPort()` : le port de l'hôte qui a envoyé, ou vers lequel on envoie
- `void setPort(int port)` : change le port vers lequel on envoie le paquet
- `byte[] getData()` : retourne le tableau contenant les données à envoyer ou reçues
- `void setData(byte[] d)` : change le tableau des données à envoyer
- `int getLength()` : retourne la taille du tableau des données reçues
- `void setLength(int length)` : change la taille du tableau des données à envoyer

Exemple

```
try{
   byte [] tampon;
   tampon = . . . ;
   InetAddress ia = . . .;
   int port = . . .;
   DatagramPacket envoiPaquet = new DatagramPacket (
                  tampon, tampon.length, ia, port);
   DatagramSocket socket = new DatagramSocket();
   socket.send(envoiPaquet);
}catch(SocketException se){
}catch(IOException ioe){
}
```

Pour l'envoi de datagramme

```
try{
   byte [] tampon = new byte[];
   int port = . . .;
   DatagramPacket receptionPaquet = new DatagramPacket (
                                 tampon, tampon.length);
   DatagramSocket socket = new DatagramSocket(port);
   socket.receive(receptionPaquet);
   String s = new String (receptionPaquet.getData());
}catch(SocketException se){
}catch(IOException ioe){
}
```

Pour la réception de datagramme

Client / serveur en mode non connecté

Sur un client

envoi requête

```
DatagramSocket socket = new DatagramSocket();

byte [] envoiTampon = new byte[1024];
byte [] receptionTampon = new byte[1024];
String requete, reponse;

requete = … // dépend de l'application
envoiTampon = requete.getBytes();
DatagramPacket envoiPaquet = new DatagramPacket(
    envoiTampon, envoiTampon.length,
    InetAddress.getbyname("miashs-dc.u-ga.fr"), 8254);

socket.send(envoiPaquet);

DatagramPacket receptionPaquet = new DatagramPacket(
    receptionTampon, receptionTampon.length);
socket.receive(receptionPaquet);
reponse = new String(receptionPaquet.getData());
```

doit être connu du client

réception réponse

Sur le serveur (prevert.upmf-grenoble.fr)

réception requête

```
DatagramSocket socket = new DatagramSocket(8254);

byte [] receptionTampon = new byte[1024];
byte [] envoiTampon = new byte[1024];
String requete, reponse;

while (true) {
  DatagramPacket receptionPaquet = new DatagramPacket(
        receptionTampon, receptionTampon.length);
    socket.receive(receptionPaquet);
    requete = new String(receptionPaquet.getData());

  // déterminer adresse et port du client
  InetAddress clientAdresse = receptionPaquet.getAddress();
  int clientPort = receptionPaquet.getPort();
  // exécuter le service
  …
  // envoyer la réponse au client
  envoiTampon = reponse.getBytes();
  DatagramPacket envoiPaquet = new DatagramPacket(
    envoiTampon, envoiTampon.length, clientAdresse,
    clientPort);
    socket.send(envoiPaquet);
}
```

envoi réponse

et voilà !