Programmation Concurrente, Réactive et Répartie

Cours N°6

Emmanuel Chailloux

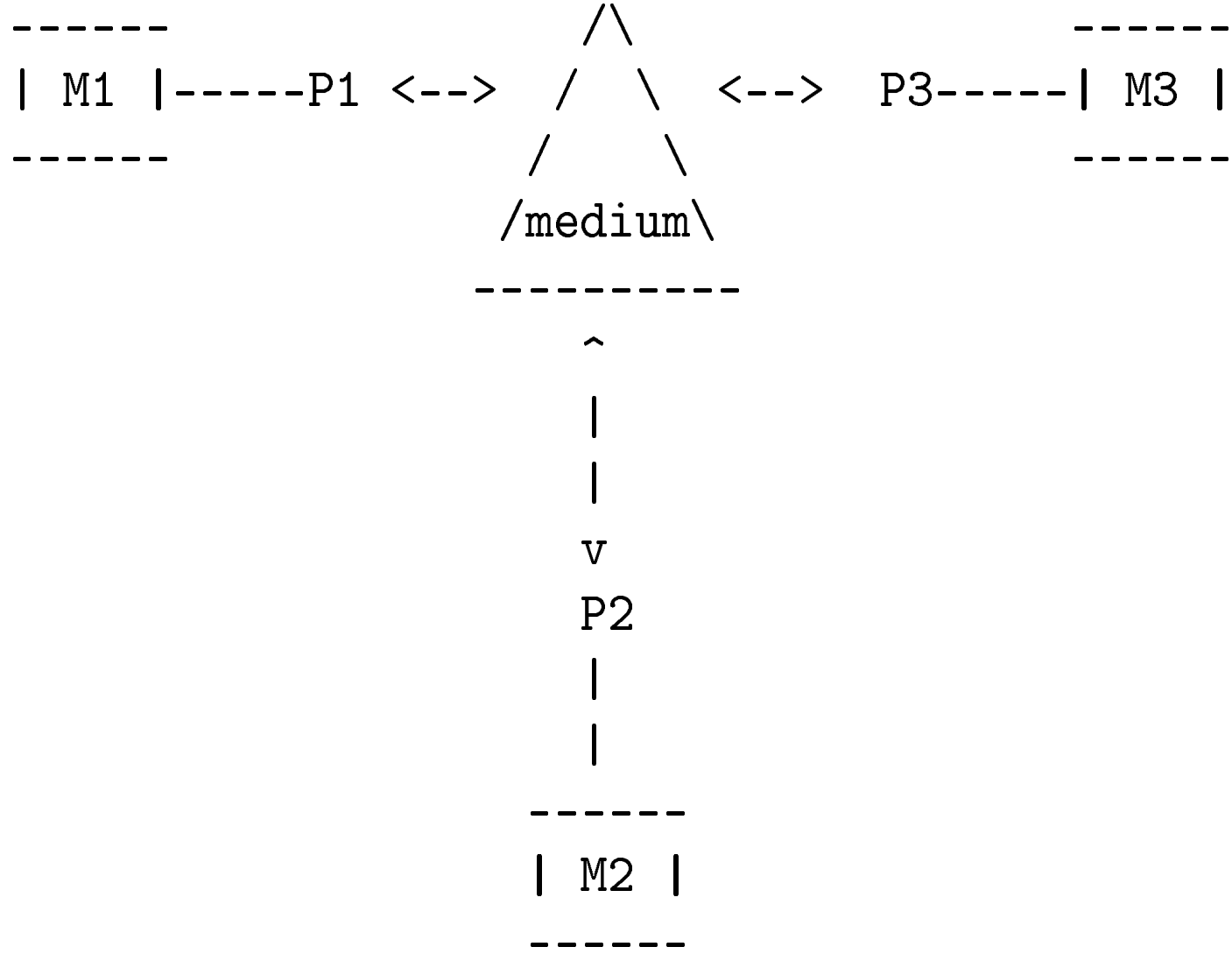
Master d'Informatique
Université Pierre et Marie Curie

année 2011-2012

Cours 6 : Programmation répartie

- modèle à mémoire répartie
- Interneteries
- Client/serveur
- Exemples en O'Caml
- Classes Java
- clients/serveur multi-langages

Modèle à mémoire répartie (1)



La difficulté de ce modèle provient de l'implantation du medium.
Les programmes s'en chargeant s'appellent des *protocoles*.

Protocoles et communication

- **protocole** : organisés en couche. Les protocoles de haut niveau, implantant des services élaborés, utilisent les couches de plus bas niveaux (7 couches : modèle ISO).
- **parallélisme** : modèle valable dans le cas de parallélisme physique (réseau d'ordinateurs) ou logique (processus Unix communiquant par "pipes" ou threads O'Caml communiquant par canaux). Il n'y a pas de valeurs globales connues par tous les processus (comme un temps global). La seule contrainte sur le temps est l'impossibilité de recevoir un message avant son émission.
- **communication et synchronisation** : Dans ce modèle la communication est explicite alors que la synchronisation est implicite (elle est en fait produite par la communication). Ce modèle est le dual du précédent.

Modèles de communication

- **un-à-un** (point-à-point) : communication d'un processus à un autre; les autres processus ignorent cette communication. Les deux primitives sont "envoi d'une valeur sur un canal" et "réception d'une valeur d'un canal".
- **un-à-tous** (diffusion) : communication d'un processus à tous les processus. Les primitives de communication sont : "envoi d'une valeur à tous" et "réception d'une valeur".
- **tous-à-tous** (diffusion) : communication de tous les processus à tous l es processus. La réception tient compte alors des différentes valeurs envoyées.

Types de communication

- **synchrone** : le transfert d'informations n'est possible que lors d'une synchronisation globale des processeurs émetteur et récepteur. L'émission et la réception peuvent être bloquantes.
- **asynchrone** : le medium peut stocker des messages en vue de leur achemineme nt futur. Il faut donc spécifier la capacité de stockage, l'ordre d'achemine ment, les délais de transmissions et la fiabilité de transmission. L'émission est non bloquante.
- **évanescent** : l'émission est non bloquante et le medium ne peut pas sto cker de messages. Le message émis est reçu par les processus prêt à le recevoir et perdu pour les autres.

Internet

- réseau de réseaux
- 2 protocoles :
  - IPv4
  - et IPv6

Machines

- les passerelles

  opèrent le passage d'un réseau à un autre;

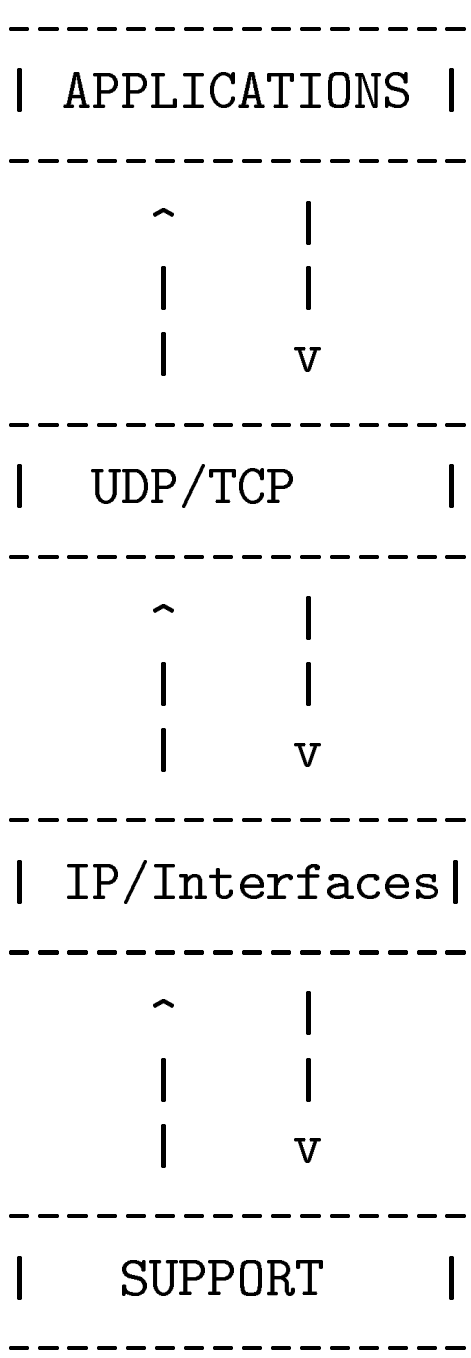
- les routeurs

  connaissent, en partie, la topologie d'Internet et opèrent l'acheminement des données;

- les serveurs de noms

  connaissent la correspondance entre noms de machines et adresses réseau.

Couches



IP

- unité de transfert : datagramme (paquet)
  contient un entête et des données
- entête : contient les adresses du destinataire et du receveur
- non fiable : ni le bon ordre, ni le bon port, ni la non duplication des paquets transmis
- routage correct et signalisation d'erreur (si un paquet n'a pas pu arriver à destination)

UDP

User Datagram Protocol

- sans connexion non fiable

à la manière d'IP pour les interfaces

TCP

Transfert Control Protocol

- mode connecté et fiable
- gestion d'acquitement
- gestion de la retransmission
- gestion de l'ordonnacement des paquets

Optimisation de la transmission par des techniques de fenetrage

Services standard en mode client-serveur

- FTP (File Transfert Protocol)
- TELNET (Terminal Transfert Protocol)
- SMTP (Simple Mail Transfert Protocol)
- HTTP (Hyper Text Transfert Protocol)

D'autres services utilisent ce modèle client-serveur :

- NFS (Network File System)
- X-Window
- les services UNIX : rlogin, rwho ...

Communication par *sockets*

Module Unix (O'Caml) - adressage IP

type abstrait : *inet_addr* pour les adresses IP

```
1 Unix.inet_addr_of_string ;;
2 Unix.string_of_inet_addr ;;
3 Unix.string_of_inet_addr (Unix.inet_addr_of_string "↩
    132.227.89.02") ;;
```

La structure des entrées de la base d'adresse est représentée par :

```
1 type host_entry =
2   { h_name : string;
3     h_aliases : string array;
4     h_addrtype : socket_domain;
5     h_addr_list : inet_addr array } ;;
```

Les deux premiers champs contiennent le nom de la machine et ses alias, le troisième, le type d'adresse et le dernier, la liste des adresses des interfaces de la machine.

nom ou adresse d'une interface

On obtient le nom de sa machine par la fonction :

```
1 Unix.gethostname ;;
2 let my_name = Unix.gethostname() ;;
```

Les fonctions d'interrogation de la base d'adresses nécessitent en entrée soit le nom, soit l'adresse de la machine.

```
1 Unix.gethostbyname ;;
2 Unix.gethostbyaddr ;;
3 let my_entrie_byname = Unix.gethostbyname my_name ;;
4 let my_addr = my_entrie_byname.Unix.h_addr_list.(0) ;;
5 let my_entrie_byaddr = Unix.gethostbyaddr my_addr ;;
6 let my_full_name = my_entrie_byaddr.Unix.h_name ;;
```

Ces fonctions déclenchent l'exception Not_found en cas d'échec de la requête.

base de services

La base de services contient la correspondance entre noms de service et numéros de port. La plupart des services standards d'Internet sont standardisés. La structure des entrées de la base de services est :

```
type service_entry =
  { s_name : string;
    s_aliases : string array;
    s_port : int;
    s_proto : string } ;;
```

Les premiers champs sont le nom du service et ses éventuels alias, le troisième contient le numéro de port du service et le dernier, le nom du protocole utilisé. Un service est en fait caractérisé par son numéro de port et son protocole. Les fonction d'interrogation sont :

```
Unix.getservbyname ;;
Unix.getservbyport ;;
Unix.getservbyport 80 "tcp" ;;
Unix.getservbyname "ftp" "tcp" ;;
```

Prises de communication

La métaphore classique est de comparer les sockets aux postes téléphoniques.

- Pour fonctionner ils doivent être raccordés au réseau (*socket*).
- *Pour recevoir un appel ils doivent posséder un numéro de type* sock_addr (*bind*).
- *Pendant un appel, il est possible de recevoir un autre appel si la configuration le permet (*listen).
- Il n'est pas nécessaire d'avoir soi-même un numéro pour appeler un autre poste (*connect*); *une fois la connexion établie la communication est dans les deux sens.*

Domaines

Domaine

Suivant qu'une prise est destinée à la communication interne ou externe elles appartiennent à un *domaine* différent. La bibliothèque Unix définit deux domaines possibles correspondant aux constructeurs du type :

```
1 type socket_domain = PF_UNIX | PF_INET;;
```

Le premier domaine correspond à la communication locale et le second, à la communication transitant sur le réseau Internet. Ce sont là les principaux domaines d'application des sockets.

Types et protocoles

Quelque soit leur domaine, les sockets définissent certaines propriétés de communications (fiabilité, ordonnancement, etc.) représentées par les constructeurs du type :

```
1 type socket_type = SOCK_STREAM | SOCK_DGRAM
2                  | SOCK_SEQPACKET | SOCK_RAW ;;
```

Suivant le type de socket utilisé, le protocole sous-jacent à la communication obéira aux caractéristiques définies. À chaque type de communication est associé un protocole par défaut.
En fait, nous n'utiliserons ici que le premier type de communication : SOCK_STREAM avec le protocole par défaut TCP. Il garantit fiabilité, ordonnancement et non duplication des messages échangés et fonctionne en mode connecté.

Création

La fonction de création d'une socket est :

```
1 Unix.socket ;;
2 - : Unix.socket_domain -> Unix.socket_type -> int -> Unix.↩
      file_descr = <fun>
```

Le troisième argument (de type int) permet de préciser le protocole associé à la communication. La valeur 0 est interprétée comme « le protocole par défaut » associé au couple (domaine,type), arguments de la création de la socket. La valeur de retour de cette fonction est un descripteur de fichier. Ainsi les échanges pourront se faire en utilisant les fonctions standards, du module Unix, d'entrées-sorties.

Créons une socket TCP/IP :

```
1 let s_descr = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 ↩
      ;;
```

Adresse et connexion

Une socket que l'on vient de créer ne possède pas d'adresse. Pour qu'une connexion puisse s'établir entre deux sockets, l'appelant doit connaître l'adresse du récepteur.

L'adresse d'une socket (TCP/IP) est constituée d'une adresse IP et d'un numéro de port. correspondant au type suivant :

```
1 type sockaddr =
2   ADDR_UNIX of string | ADDR_INET of inet_addr * int ;;
```

L'entier qui fait partie de l'adresse des sockets du domaine Internet correspond au numéro de port.

Attribution d'une adresse

La première chose à faire, pour pouvoir recevoir des appels, après la création d'une socket est de lui attribuer, de la *lier* à, une adresse. C'est le rôle de la fonction :

```
#Unix.bind ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

En effet, nous avons déjà un descripteur de socket, mais l'adresse qui lui est attachée à la création ne peut guère nous être utile comme le montre l'exemple suivant :

```
let (addr_in, p_num) =
  match Unix.getsockname s_descr with
    Unix.ADDR_INET (a,n) -> (a,n)
  | _ -> failwith "pas INET";;
val addr_in : Unix.inet_addr = <abstr>
val p_num : int = 0
Unix.string_of_inet_addr addr_in;;
- : string = "0.0.0.0"
```

Capacité d'écoute/réception (1)

Il faut encore procéder à deux opérations avant que notre socket soit complètement opérationnelle pour recevoir des appels : définir sa capacité d'écoute et se mettre effectivement à l'écoute. C'est le rôle respectif de :

```
1 # Unix.listen;;
2 - : Unix.file_descr -> int -> unit = <fun>
```

Le second argument (de type int) de la fonction listen indique le nombre maximal de connexions en attente toléré.

Capacité d'écoute/réception (2)

et de :

```
1 # Unix.accept;;
2 - : Unix.file_descr -> Unix.file_descr * Unix.sockaddr = <fun>
```

L'appel de la fonction accept attend une demande de connexion. Quand celle-ci survient la fonction accept se termine et retourne l'adresse de la socket ayant appelée ainsi qu'un nouveau descripteur de socket, (dite socket de service). Cette socket de service est automatiquement liée à une adresse. La fonction accept ne s'applique qu'aux sockets ayant exécuté un listen, c'est-à-dire aux sockets ayant paramètrées la file d'attente des demandes de connexion.

demande de connexion

La fonction duale de accept est ;

```
1 # Unix.connect;;
2 - : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

Un appel à `Unix.connect s_descr s_addr` établit un connexion entre la socket locale `s_descr` (qui est automatiquement liée) et la socket d'adresse `s_addr`.

Modèle client-serveur

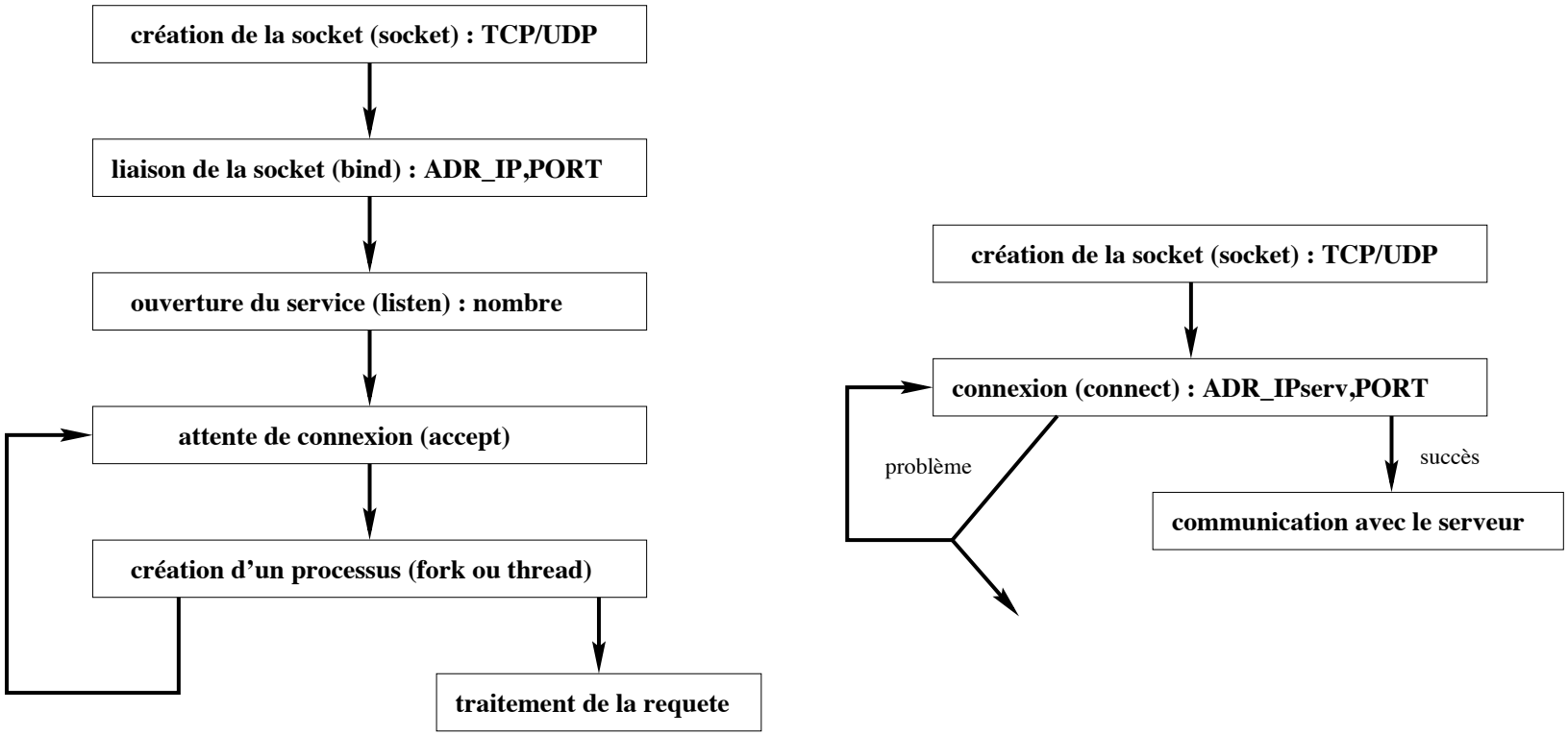
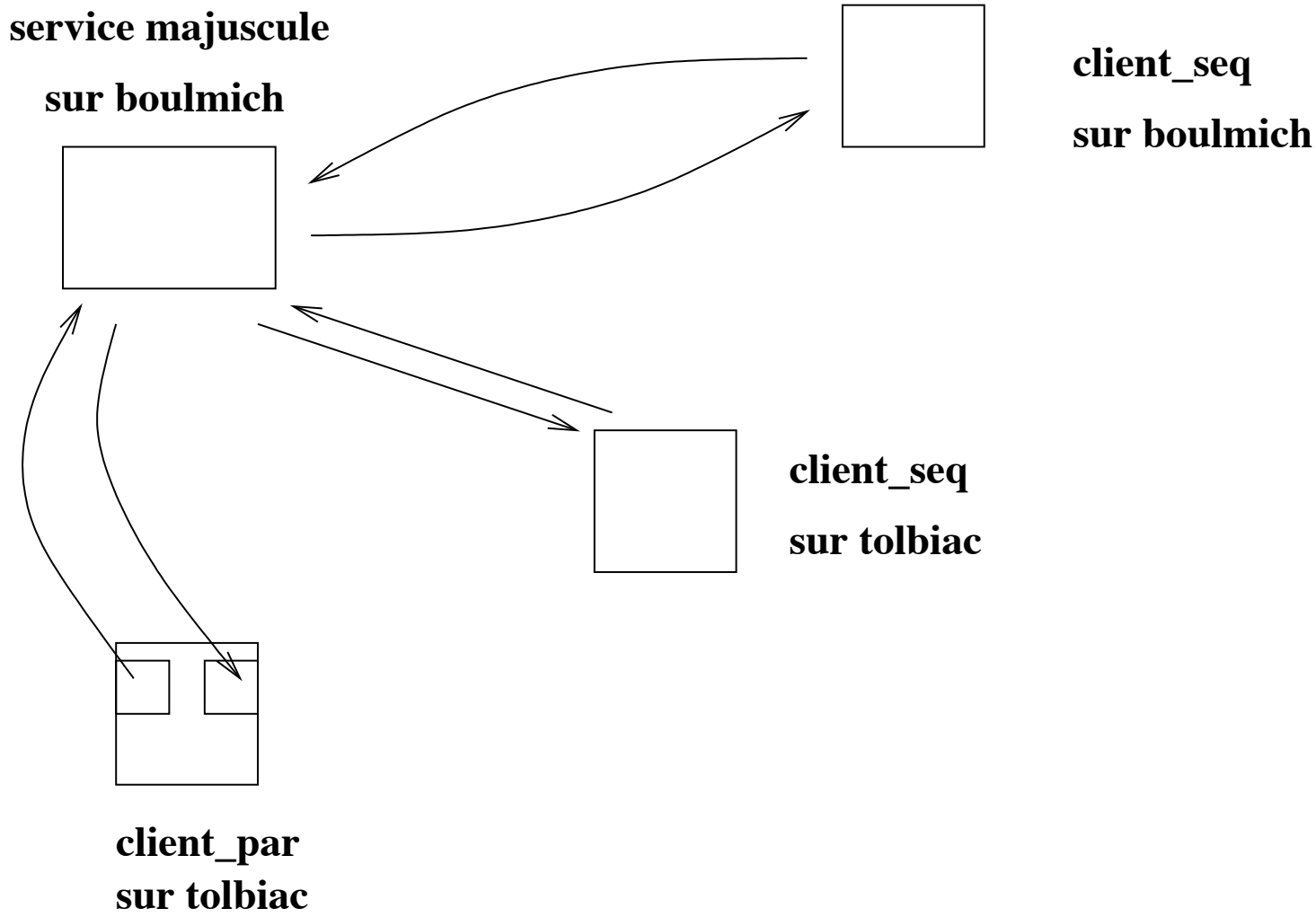


Figure: **Schéma des actions d'un serveur et d'un client**

Programmation d'un client-serveur



Service MAJUSCULE et des clients : certains tournent sur la même machine que le serveur et d'autres se trouvent sur des machines distantes.

server.ml (1)

```
1  (*  ocamlc -o serv.exe -thread -custom unix.cma threads.cma server.ml
2              -cclib -lthreads -cclib -lunix *)
3  class virtual  server  port n =
4  object (s)
5    val port_num = port
6    val nb_pending = n
7    val sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0
8    method start () =
9      let host = Unix.gethostbyname (Unix.gethostname()) in
10     let h_addr = host.Unix.h_addr_list.(0) in
11     let sock_addr = Unix.ADDR_INET(h_addr, port_num) in
12       Unix.bind sock sock_addr ;
13       Unix.listen sock nb_pending ;
14       while true do
15         let (service_sock, client_sock_addr) =
16             ThreadUnix.accept sock
17         in   s#treat service_sock client_sock_addr
18       done
19   method virtual treat : Unix.file_descr -> Unix.sockaddr
       -> unit
20 end ;;
```

server.ml (2)

```
1  let gen_num = let c = ref 0 in (fun () -> incr c; !c) ;;
2  class virtual connexion sd (sa : Unix.sockaddr) b =
3  object (self)
4    val s_descr = sd
5    val s_addr = sa
6    val mutable numero = 0
7    val mutable debug = b
8    method set_debug b = debug <- b
9    initializer
10     numero <- gen_num();
11     if debug then (
12       Printf.printf "TRACE.connexion : objet traitant %d ↩
             cree\n" numero ;
13       print_newline())
14   method start () =  Thread.create (fun x -> self#run x ; ↩
       self#stop x) ()
15   method stop() =
16     if debug then (
17       Printf.printf "TRACE.connexion : fin objet traitant ↩
             %d\n" numero ;
18       print_newline () );
19       Unix.close s_descr
20   method virtual run : unit -> unit
21 end;;
```

server.ml (3)

```
1  exception Fin ;;
2  let my_input_line  fd =
3    let s = " "  and  r = ref "" in
4      while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\n'
5        do r := !r ^s done ;
6      !r ;;
7  class connexion_maj sd sa b =
8  object(self)
9    inherit connexion sd sa b
10   method run () =
11       try
12         while true do
13           let ligne =  my_input_line s_descr
14           in if (ligne = "") or (ligne = "\013") then ↩
                 raise Fin ;
15             let result = (String.uppercase ligne)^"\n"
16             in ignore (ThreadUnix.write s_descr result 0
17                                 (String.length result))
18         done
19       with
20          Fin  -> ()
21     | exn  -> print_string (Printexc.to_string exn) ; ↩
           print_newline()
22 end ;;
```

server.ml (4)

```
1  class server_maj port n =
2  object(s)
3    inherit server port n
4    method treat s sa  =
5     ignore( (new connexion_maj s sa true)#start())
6  end;;
7
8  (*
9  val main : unit -> unit
10 *)
11
12 let main () =
13   if Array.length Sys.argv < 3
14   then Printf.printf "usage :  server port num\n"
15   else
16     let port = int_of_string(Sys.argv.(1))
17     and n = int_of_string(Sys.argv.(2)) in
18       (new server_maj port n )#start();;
19
20 main();;
```

client.ml (1)

```
1  (*     ocamlc -o client.exe -thread -custom unix.cma ↩
       threads.cma client.ml
2            -cclib -lthreads -cclib -lunix *)
3
4  class virtual client serv p =
5  object(s)
6    val sock = ThreadUnix.socket Unix.PF_INET Unix.↩
         SOCK_STREAM 0
7    val port_num = p
8    val server = serv
9
10   method start () =
11     let host = Unix.gethostbyname server in
12     let h_addr =  host.Unix.h_addr_list.(0) in
13     let sock_addr = Unix.ADDR_INET(h_addr,port_num) in
14       Unix.connect sock sock_addr;
15       s#treat sock sock_addr;
16       Unix.close sock
17   method virtual treat : Unix.file_descr -> Unix.sockaddr ↩
       -> unit
18 end;;
```

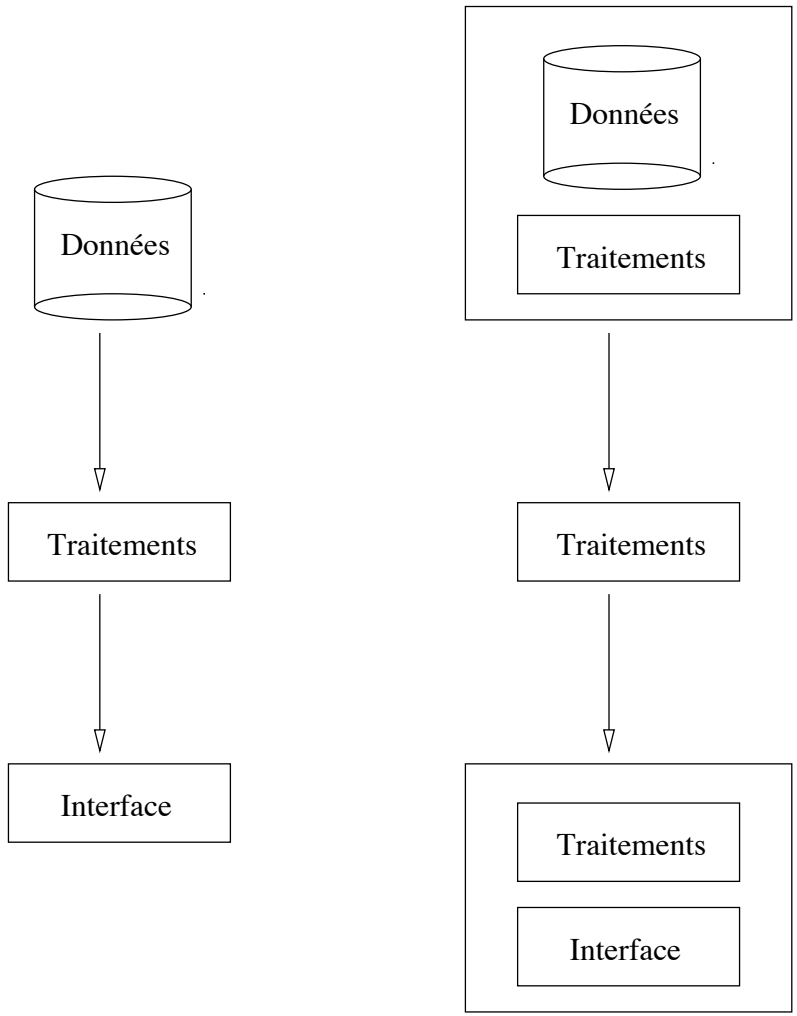
client.ml (2)

```
1
2  class client_maj s p =
3  object
4    inherit client s p
5    method treat s sa =
6      try
7        while true do
8          let si = (my_input_line Unix.stdin) ^ "\n"  in
9           ignore (ThreadUnix.write s si 0 (String.length si↩
               ));
10          let so = (my_input_line s) in
11          if so = "" then raise Fin
12          else (Printf.printf "%s\n" so; flush stdout)
13       done
14     with Fin -> ()
15 end;;
```

client.ml (3)

```
1
2  let main () =
3    if Array.length Sys.argv < 3
4    then Printf.printf "usage :  client server port\n"
5    else
6      let port = int_of_string(Sys.argv.(2))
7      and s = (Sys.argv.(1)) in
8        (new client_maj s  port  )#start();;
9
10 main();;
```

client-serveur à plusieurs niveaux



Différentes architectures de clients-serveurs : Chaque niveau peut être implanté sur des machines différentes. L'interface utilisateur s'exécute sur les machines des utilisateurs de l'application.

communications

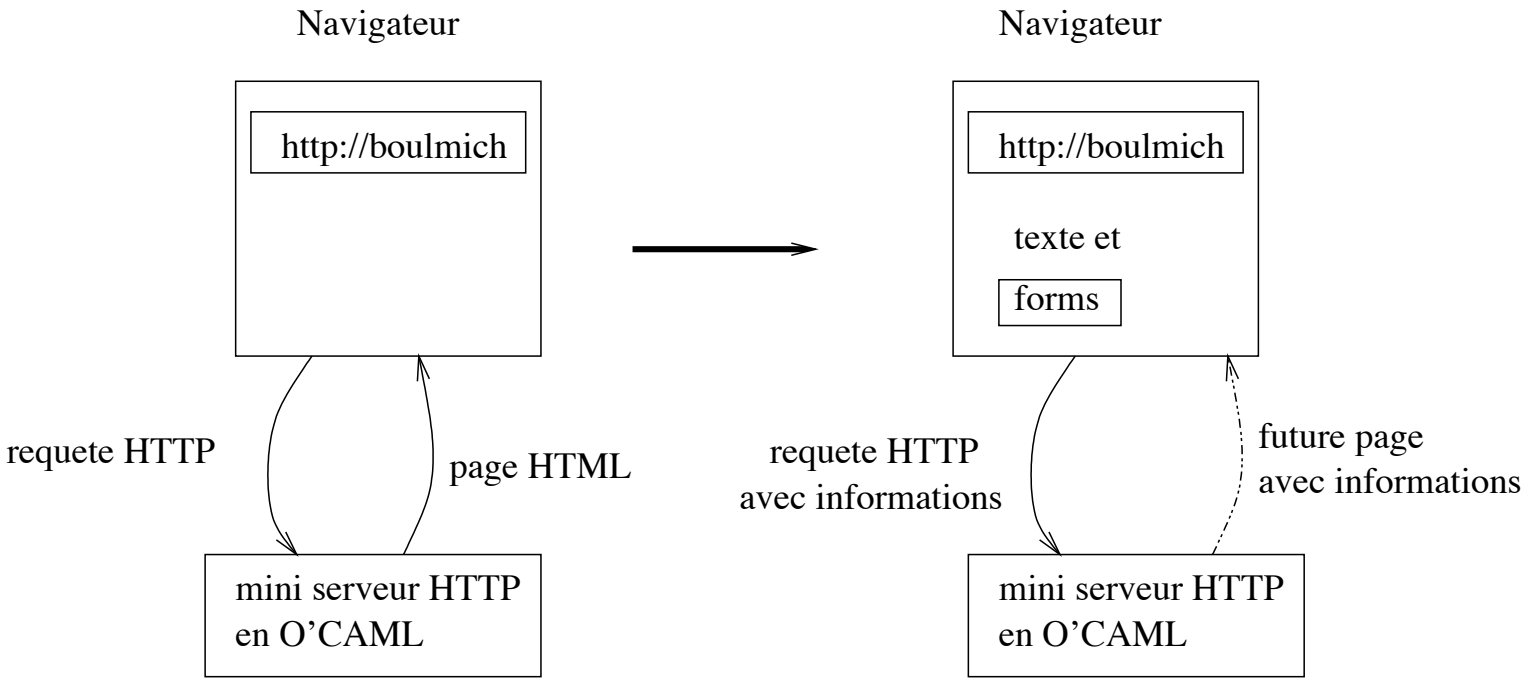


Figure: Communication entre un navigateur et un serveur O'Caml

Sockets

paquetage : `java.net` : fournit un ensemble de classes pour communiquer sur le réseau Internet.

- classe de bas niveau : pour les couches basses (`DatagramPacket`,...)
- et de haut niveau : pour les clients/serveurs

La classe `InetAddress` représente les adresses Internet. Elle est utilisée dans la classe `Socket`.

Classes de bas niveau

- `DatagramPacket` : objets qui contiendront les donnés envoyées ou reçues ainsi que l'adresse de destination ou de provenance du datagram;
- `DatagramSocket` : création de socket UDP

API Java (1)

Constructeurs et méthodes des classes de bas niveau :

- : `DatagramPacket`
  - `public DatagramPacket(byte []buffer,int taille)`
  - `public DatagramPacket(byte []buffer,int taille, InetAddress adresse, int port)`

API Java (2)

Constructeurs et méthodes des classes de bas niveau :

- DatagramSocket
  - `public DatagramSocket() throws SocketException`
  - `public DatagramSocket(int port) throws SocketException`
  - `public void send(DatagramPacket data) throws IOException`
  - `public synchronized void receive(DatagramPacket data) throws IOException`
  - `public synchronized void setSoTimeOut (int timeout) throws SocketException`

Exemple de client/serveur

Conversion de chaînes en MAJUSCULE (reprend le serveur Echo de "Java et Internet").

- SMAJUDP : clase du serveur (ne répond qu'une seule fois)
- CMAJUDP : classe du client

Serveur

```
1  import java.io.*;
2  import java.net.*;
3  class SMAJUDP {
4      public static void main(String []a) {
5      int port = 4321;
6      int taille = 1024;
7      byte []buffer= new byte [1024];
8      try {
9        DatagramSocket socket = new DatagramSocket(port);
10       DatagramPacket data = new DatagramPacket(buffer,↩
             buffer.length);
11       socket.receive(data);
12       String s = (new String(data.getData()));
13       System.out.println("serveur R: " + s);
14       s = s.toUpperCase();
15       System.out.println("serveur E: " + s);
16       data.setData(s.getBytes());
17       socket.send(data);
18     }
19     catch (SocketException se) {}
20     catch (IOException e) {}
21     }
22 }
```

Client

```
1  import java.io.*;
2  import java.net.*;
3  class CMAJUDP {
4    public static void main (String []a) {
5      int port = 4321;
6      InetAddress adr = null;
7      try {
8      adr = InetAddress.getByName(a[0]);
9      DatagramSocket socket = new DatagramSocket();
10     String s = "Le petit chat est mort.";
11     byte [] buffer = s.getBytes();
12     DatagramPacket data = new DatagramPacket(buffer,buffer↩
           .length,adr,port);
13     System.out.println("client E: "+s);
14     socket.send(data);
15     // ...
16     socket.receive(data);
17     String r = new String(data.getData());
18     System.out.println("client R: "+r);
19     socket.close();
20     }
21     catch (UnknownHostException ue) {}
22     catch (SocketException se) {}
23     catch (IOException ie) {}
24   }
```

Classes de haut niveau

- `ServerSocket` : méthode `accept` retourne une socket quand une demande est reçue;
- `Socket` (classe finale) : pour la communication à travers le réseau. Les méthodes `getOutputStream()` et `getInputStream` permettent de récupérer les canaux de communication.
- `URL` : permet de dialoguer via une URL.

Exemple de client/serveur

Conversion de chaînes en MAJUSCULE via un serveur (Java : de l'esprit à la méthode)
Le serveur est construit à partir de 2 classes :

- `Serveur` : classe générique (au numéro du port près)
- `Connexion` : pour les canaux et le traitement

Lors d'une connexion d'un client au serveur (`Serveur.run()`) une nouvelle instance de `Connexion` est créée.

Partie serveur générique

```
1  import java.io.*; import java.net.*;
2  public class Serveur extends Thread {
3    protected static final int PORT =45678;
4    protected ServerSocket ecoute;
5  Serveur () {
6    try  { ecoute = new ServerSocket(PORT);}
7    catch (IOException e) {
8       System.err.println(e.getMessage());
9       System.exit(1); }
10   System.out.println("Serveur en ecoute sur le port : "+↩
         PORT);
11   this.start();
12 }
13 public void run () {
14   try {
15     while (true) {
16       Socket client=ecoute.accept();
17       Connexion c = new Connexion (client);}}
18   catch (IOException e) {
19      System.err.println(e.getMessage()); System.exit(1);}
20 }
21 public static void main (String[] args) {new Serveur();}}
```

Partie serveur spécifique

```
1  import java.io.*;  import java.net.*;
2  class Connexion extends Thread {  protected Socket client;
3  protected DataInputStream in;   protected PrintStream out;
4  public Connexion(Socket client_soc) {
5    client=client_soc;
6    try  { in=new DataInputStream(client.getInputStream());
7      out=new PrintStream(client.getOutputStream());}
8    catch (IOException e) {
9      try {client.close();}
10     catch (IOException e1){}
11     System.err.println(e.getMessage());
12     return; }
13   this.start();
14 }
15 public void run() {
16   try {
17     while (true) { String ligne=in.readLine();
18        if (ligne.toUpperCase().compareTo("FIN")==0) break;
19        System.out.println(ligne.toUpperCase());
20        out.println(ligne.toUpperCase());  out.flush();  }}
21    catch (IOException e)
22     {System.out.println("connexion : "+e.toString());}
23    finally {try {client.close();}  catch (IOException e) ↩
          {}}
24 }
```

Partie client (1)

```
1  public class Client {
2  protected static final int PORT=45678;
3  public static void main(String[] args) { Socket s=null;
4    if (args.length != 1) {
5      System.err.println("Usage: java Client <hote>");
6      System.exit(1); }
7    try { s=new Socket (args[0],PORT);
8      DataInputStream canalLecture = new DataInputStream(s.↩
           getInputStream());
9      DataInputStream console = new DataInputStream (s.↩
           getInputStream());
10     PrintStream canalEcriture = new PrintStream(s.↩
           getOutputStream());
11     System.out.println("Connexion etablie : "+
12                        s.getInetAddress()+" port : "+s.↩
                              getPort());
13     String ligne = new String();
14     char c;
```

Partie client (2)

```
1       while (true) { System.out.print("?"); System.out.flush↩
            ();
2         ligne = "";
3          while ((c=(char)System.in.read()) != '\n') {ligne=↩
               ligne+c;}
4          canalEcriture.println(ligne); canalEcriture.flush()↩
               ;
5          ligne=canalLecture.readLine();
6          if (ligne == null) {System.out.println("Connexion ↩
               terminee"); break;}
7          System.out.println("!"+ligne);
8       }}
9       catch (IOException e) {System.err.println(e);}
10      finally  { try {if (s != null) s.close();} catch (↩
            IOException e2) {}
11    }
12  }
13  }
```

Exécution

```
[chaillou@ippc56]$ java Serveur
Serveur en ecoute sur le port : 45678
ADIEU
MONDE CRUEL
```

coté client

```
[chaillou@ippc56]$ java Client ippc56
Connexion etablie : ippc56/134.157.116.56 port : 45678
?adieu
!ADIEU
?monde cruel
!MONDE CRUEL
?fin
Connexion terminee
```

Communication avancée

Pour établir un nouveau service, il est souvent nécessaire de définir un “protocole” de communication entre le serveur et les clients.

- HTTP
- FTP
- SQL serveur

...