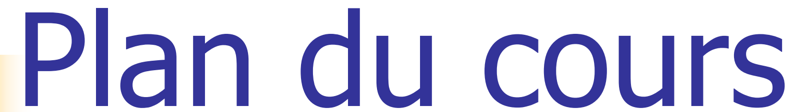
NSY107 - Intégration des systèmes client-serveur

Cours du 20/05/2006 + début 27/05/2006 (4 à 6 heures)

© Emmanuel DESVIGNE
<emmanuel@desvigne.org>

Document sous licence libre (FDL)



Plan du cours

- Rappels
- Concepts réseau & système du modèle client-serveur
- Les APIs réseau UNIX
- TP1 et TP2
- Quelques outils système UNIX
- TP3

Rappels [1/3]

- Dans la suite de ce cours, nous limiterons la notion de client/serveur à un « programme client » <-> « programme serveur », quelle que soit la machine sur laquelle tournent ces programme (pour nos TPs, il pourra s'agir de la même machine)
- La majorité de nos exemples (tous ?) se feront sur un réseau « IP ». Mais il serait possible d'appliquer les mêmes concepts à d'autres protocoles réseau (SNA, IPX/SPX, DDP –AppleTalk–, etc.)

Rappels [2/3]

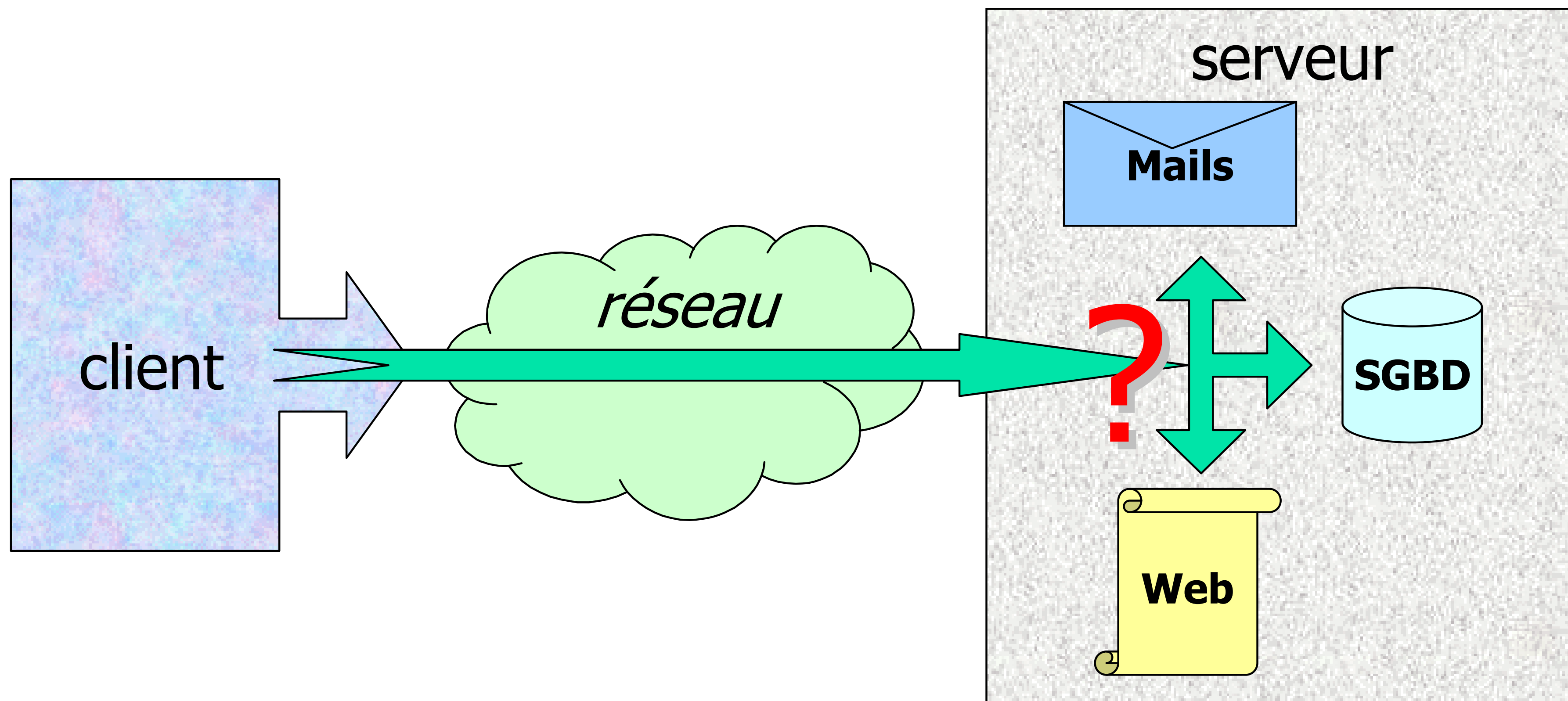
- Il existe 2 grands modes de fonctionnement client/serveur :
  - Le mode « datagramme », ou « échanges de paquets » : le client envoie sa requête dans un paquet, et le serveur lui répond dans un ou plusieurs paquets (ex: TFTP)

Rappels [3/3]

- Le mode connecté : un « canal de communication » se crée entre le client et le serveur (à la demande du client), et les échanges (ordres, accusés de réception, données...) transitent par ce canal, ou alors via d'autres canaux ouverts pour l'occasion (ex : HTTP, IMAP, ...) ;

Concepts réseau & système du modèle client-serveur [1/9]

- Quand un paquet contenant une requête arrive sur un serveur, comment l'OS sait à quel service il doit donner la requête ?



Concepts réseau & système du modèle client-serveur [2/9]

- En fait, chaque paquet réseau contient :
  - L'adresse IP de la machine d'origine (le client dans le cas d'une requête),
  - L'adresse IP de la machine de destination (le serveur dans notre cas),
  - Et... une information qui permet de savoir à quel « service » est destiné le paquet. On parle alors de « **numéro de service** » ou « **numéro de port** »

Concepts réseau & système du modèle client-serveur [3/9]

- L'expéditeur possède aussi un numéro de port, alloué dynamiquement par le système, utilisé pour la réponse
- Il existe une liste de « *well known ports* », i.e. les services standards (dans **`/etc/services`** sous Unix, dans **`%SystemRoot%\System32\drivers\etc\services`** pour Windows). Exemples :
  - **`telnet      23/tcp`**
  - **`smtp        25/tcp`**
  - **`tftp        69/udp`**

Concepts réseau & système du modèle client-serveur [4/9]

- Réservation des n° de port, <u>par convention</u> :
  - **Port n° 0** : non utilisable pour une application. C'est un « jocker » qui indique au système que c'est à lui de compléter le numéro (Cf. N°49152 à 65535)
  - **Ports 1 à 1023** : ports réservés au superutilisateur (root/Administrateur). On y trouve les serveurs « classiques » (DNS, FTP, SMTP, Telnet, SSH…). Il existait anciennement un découpage 1-255, 256-511, et 512-1023 qui n'est plus utilisé.
  - **Ports 1024 à 49151** : services enregistrés par l'IANA (Internet Assigned Numbers Authority), accessibles aux utilisateurs ordinaires
  - **Ports 49152 à 65535** : zone d'attribution automatique des ports, pour la partie cliente

Concepts réseau & système du modèle client-serveur [5/9]

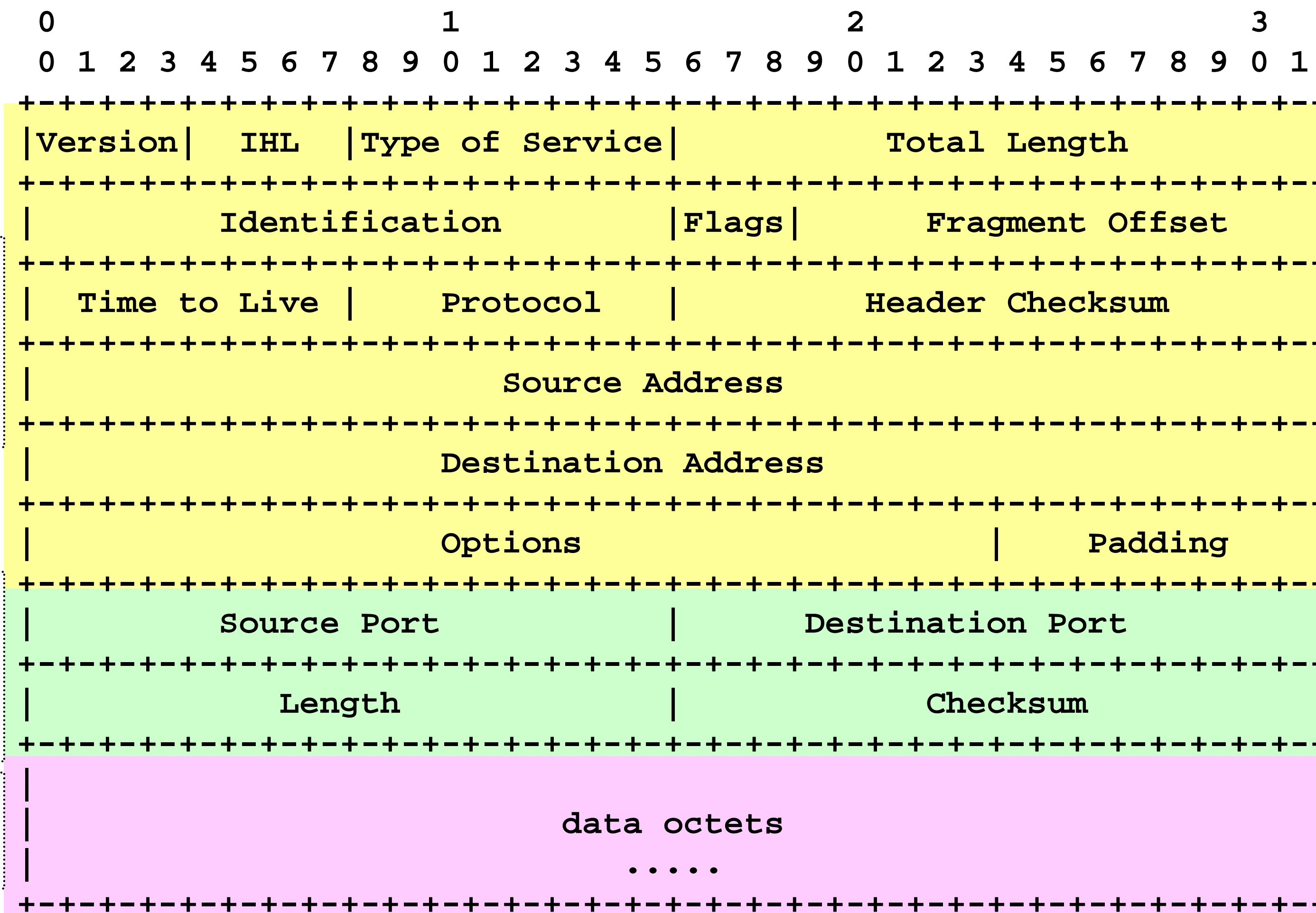
- Sur Internet, 2 protocoles traduisent les 2 fonctionnements déjà vus :
  - UDP : mode non connecté (échange de paquets), défini dans le RFC 768
  - TCP : mode connecté (établissement d'un canal de communication). En pratique, les données transmises dans ce canal sont découpées elles-aussi en paquets (RFC 793)
- Ces deux protocoles s'appuient sur le protocole IP (RFC 791)

Concepts réseau & système du modèle client-serveur [6/9]

- RFC = Request For Comments :
  - Publiés depuis 1969
  - Documents faisant office de proposition
  - Beaucoup sont tombés dans l'oubli, certains sont devenu des « standards de fait »
  - Publiés par l'IETF (Internet Engineering Task Force), organisme de proposition de technologies pour Internet
  - Exemples : UDP=768, IP=791, TCP=793, FTP=959, Telnet=854, PPP=1661, HTTP=2068, SMTP=2821
  - URL pour trouver un RFC : http://www.faqs.org/rfcs

Concepts réseau & système du modèle client-serveur [7/9]

- Exemple : paquet IP/UDP (RFC 791, 768)

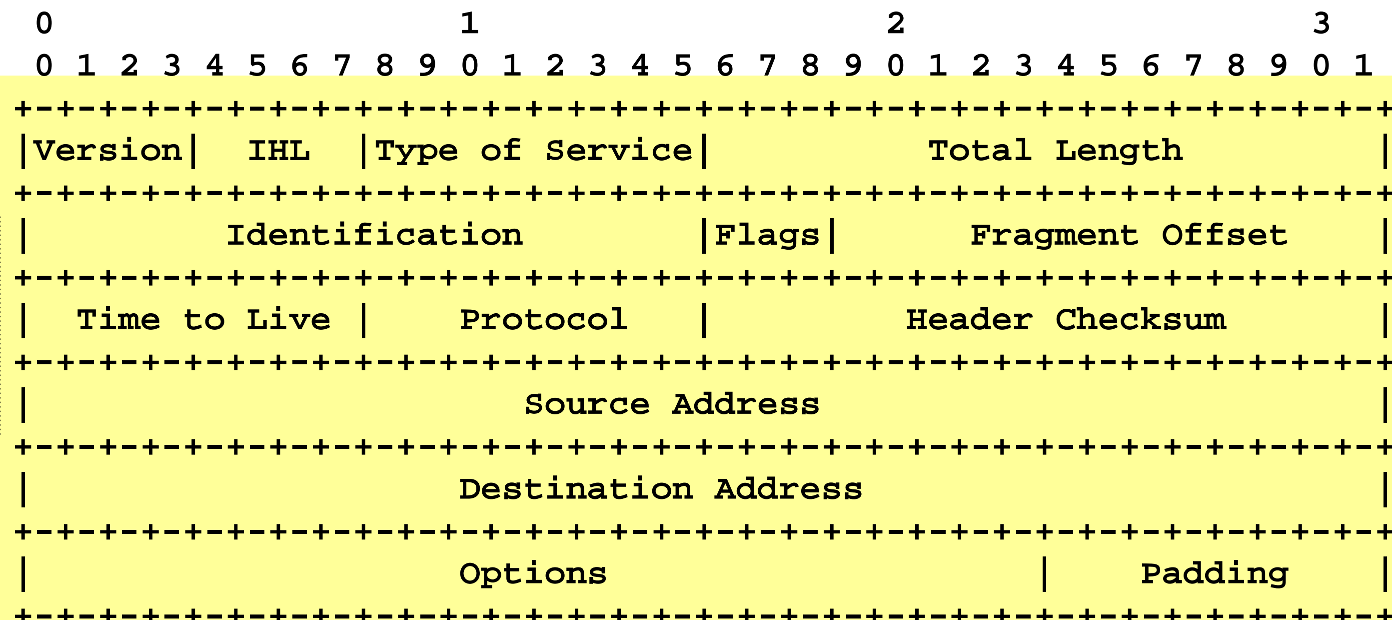


Concepts réseau & système du modèle client-serveur [8/9]

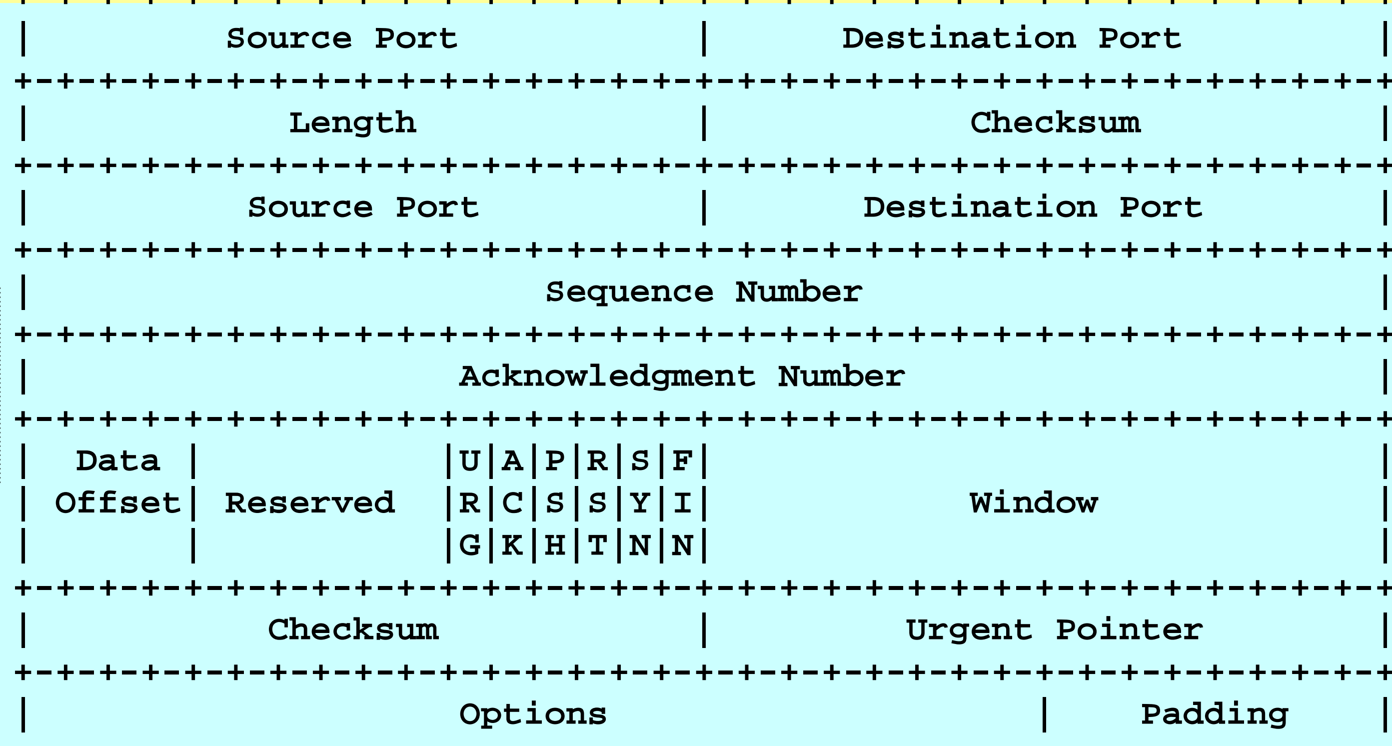
- Exemple : paquet IP/TCP (RFC 791, 793)

(Cf. diapositive suivante)

entête IP



entête TCP



données

Les APIs réseau UNIX [1/27]

- Pour écrire un programme capable de dialoguer sur un réseau, quel que soit le langage et le système d'exploitation, il faut utiliser des APIs
- API = Application Programming Interface => interface de programmation :
  - Ensemble de définitions, de règles, de structures de données, d'objet, de fonctions, d'appels systèmes... qui permettent d'accéder et d'utiliser une ressource

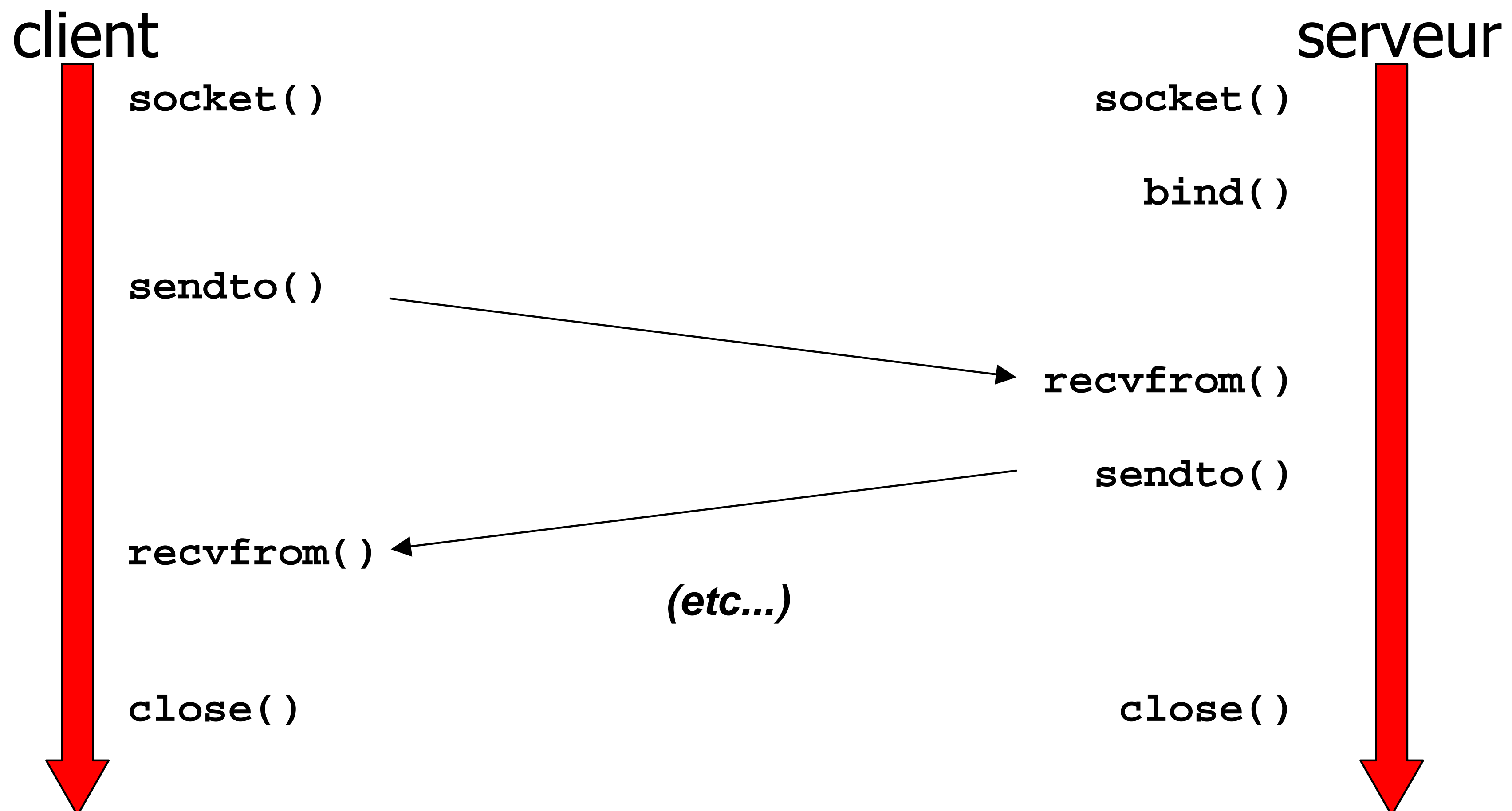
Les APIs réseau UNIX [2/27]

- L'API réseau sous Unix s'appuie sur les « sockets », développés par Berkeley
- Le socket peut être vu comme une « fiche »

Modèle sockets	Modèle OSI
Applications utilisant les sockets	Application
	Présentation
	Session
UDP/TCP	Transport
IP, ARP, ICMP...	Réseau
Ethernet, 802.11, IEEE1394, etc.	Liaison
	Physique

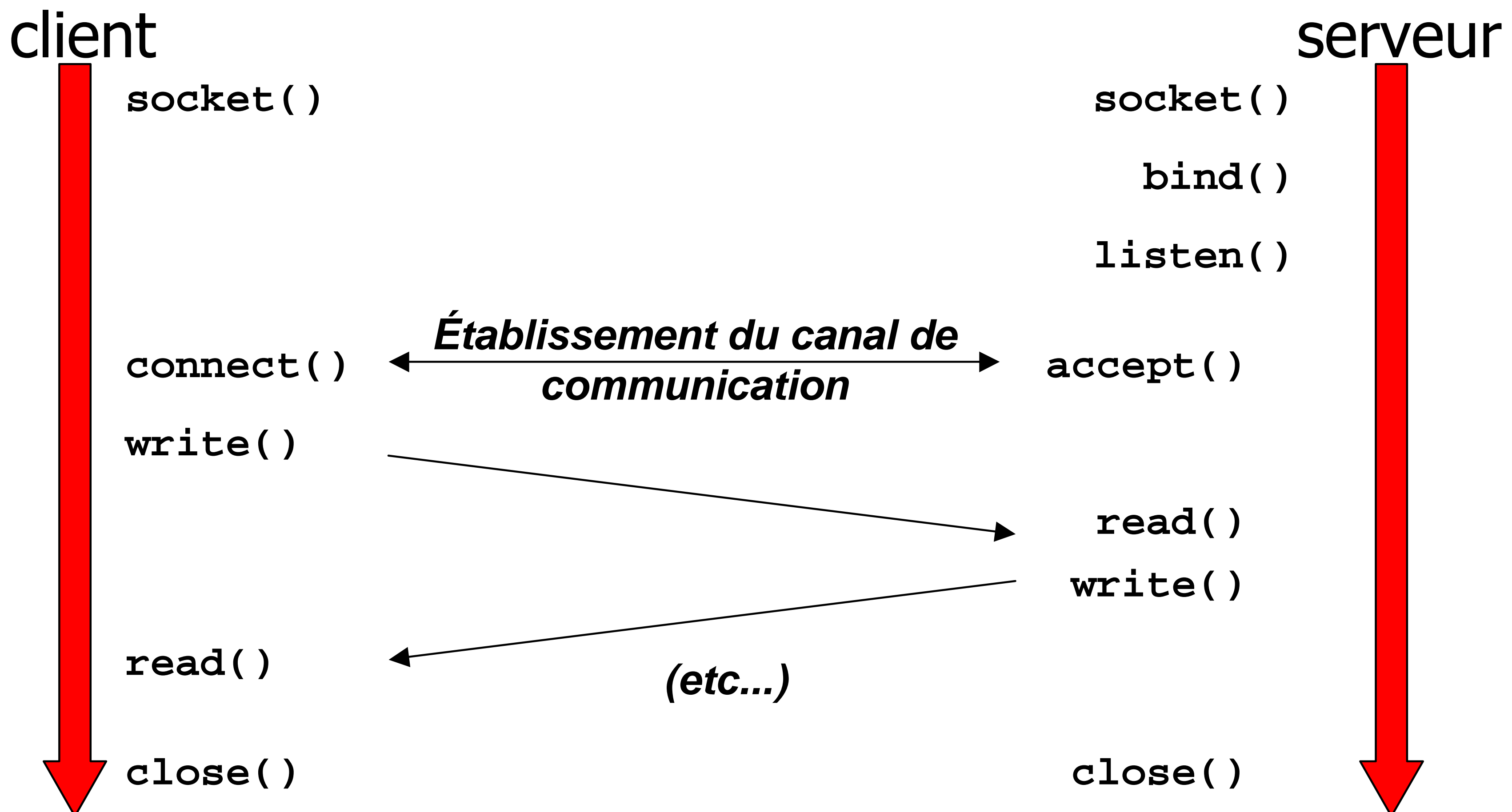
Les APIs réseau UNIX [3/27]

- Mode non connecté, UDP (datagrammes)



Les APIs réseau UNIX [4/27]

- Mode connecté, TCP (canal/flux)



Les APIs réseau UNIX [5/27]

- Autres fonctions utiles :
  - **`gethostbyname()`** : effectue la résolution DNS pour connaître l'IP en fonction du nom (rq : il existe aussi **`gethostbyaddr()`**).
  - **`htons()`**, **`ntohs()`**, **`htonl()`**, **`ntohl()`** : fonctions permettant de traduire des entiers courts (16 bits) ou longs (32 bits) du format du microprocesseur local au format standard qui est utilisé sur le réseau Internet (modèle Big Endian), et réciproquement.

Les APIs réseau UNIX [6/27]

- Endianness :
  - Selon les systèmes, il existe 2 façons de coder en mémoire les mots de plusieurs octets. Ex : mot de 4 octets 0xA0B7105E
    - **Big Endian** (gros-boutiste) : case mémoire N = A0 (octet poids fort) , case mémoire N+1 = B7, case mémoire N+2 = 10, case mémoire N+3 = 5E (octet poids faible). Ex : Motorola 68000, SPARC (Sun)
    - **Little Endian** (petit-boutiste) : case mémoire N = 5E (octet poids faible) , case mémoire N+1 = 10, case mémoire N+2 = B7, case mémoire N+3 = A0 (octet poids fort). Ex : Intel/AMD 80x86
    - Certains processeurs peuvent fonctionner suivant les 2 modes (PowerPC –IBM –, ARM, DEC Alpha, MIPS, PA-RISC –HP–, IA-64 –Intel–) : **bytesexual**, **bi-endian**

Les APIs réseau UNIX [7/27]

- Les « *includes* » utiles en C :

```
/* Directives include pour les fonctions standards */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

/* Pour les fonctions et structures liées au réseau */
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

- Les doc. (en français) de toutes les fonctions utiles aux TP (appels système) sont ici :
  http://jp.barralis.com/linux-man/

Les APIs réseau UNIX [8/27]

- Les structures de données en C :
  - ```
    struct sockaddr_in
    {
      short   sin_family;       /* Fam. protocole; AF_INET */
      u_shor  sin_port;         /* numero de port */
      struct in_addr sin_addr;  /* @ IP machine */
      char    sin_zero[8];      /* jam pour faire 16 oct. */
    };
    ```
  - ```
    struct in_addr
    {
      u_mong  s_addr;
    };
    ```
  - ```
    struct hostent {
      char *h_name;         /* nom de la machine */
      char **h_aliases;     /* liste d'alias, terminée par NULL */
      int  h_addrtype;      /* type de l'adresse; AF_INET */
      int  h_length ;       /* taille en octets de l'adresse */
      char **h_addr_list;   /* liste d'@, terminée par NULL */
    };
    ```
  - ```
    #define  h_addr   h_addr_list[0]
    ```

Les APIs réseau UNIX [9/27]

- Structure d'un serveur UDP en C :

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <string.h>
#include  <errno.h>

#include  <netdb.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>

int       main(int argc, char* argv[])
{
  int                     desc_sockserv;
  struct sockaddr_in      sockadrs_serv;
  struct sockaddr_in      sockadrs_cli;
  size_t                  lg_sockadrs_cli;
  char                    buffer_in[256];
  ssize_t                 taille_msg_recu;
  char                    buffer_out[256];
  ssize_t                 taille_reponse;

                    .../...
```

Les APIs réseau UNIX [10/27]

```
[…]
  desc_sockserv=socket(AF_INET, SOCK_DGRAM, 0);
  if (desc_sockserv < 0)
  { Traitement de l'erreur et on quitte}

  bzero(&sockadrs_serv, sizeof(struct sockaddr_in));
  bzero(&sockadrs_cli, sizeof(struct sockaddr_in));
  sockadrs_serv.sin_family=PF_INET;
  sockadrs_serv.sin_addr.s_addr=htonl(INADDR_ANY);
  sockadrs_serv.sin_port=htons(2424);
  if (bind(desc_sockserv, (struct sockaddr *) &sockadrs_serv, \
                              sizeof(struct sockaddr_in)) < 0)
  { Traitement de l'erreur et on quitte}
  while (1)
  {
    taille_msg_recu=recvfrom(desc_sockserv, buffer_in, \
      sizeof(buffer_in),0,(struct sockaddr *)&sockadrs_cli,&lg_sockadrs_cli);
    if (taille_msg_recu < 0)
    { Traitement de l'erreur et on quitte}
    /* […] Prépare la réponse dans buffer_out & dans taille_reponse */
    if (sendto(desc_sockserv, buffer_out, taille_reponse, 0, \
      (struct sockaddr *)&sockadrs_cli , lg_sockadrs_cli ) <  0)
    { Traitement de l'erreur et on quitte}
  }
}
/* fin serveur UDP */
```

Les APIs réseau UNIX [11/27]

- Structure d'un client UDP en C :

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <string.h>

#include  <errno.h>
#include  <netdb.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>

int       main(int argc, char* argv[])
{
  int                        desc_sockcli;
  struct hostent             *serv_hostinfo;
  struct sockaddr_in         sockadrs_serv;
  char                       buffer_out[256];
  ssize_t                    taille_question;
  char                       buffer_in[256];
  ssize_t                    taille_reponse;


                    .../...
```

Les APIs réseau UNIX [12/27]

```
[…]

  desc_sockcli=socket(AF_INET, SOCK_DGRAM, 0);
  if (desc_sockcli < 0)
  { traitement de l'erreur et on quitte }
  bzero(&sockadrs_serv, sizeof(struct sockaddr_in));
  serv_hostinfo=gethostbyname("nom.serveur.com");
  if (serv_hostinfo == NULL)
  { traitement de l'erreur et on quitte }
  sockadrs_serv.sin_family=serv_hostinfo->h_addrtype;
  memcpy((char *)&sockadrs_serv.sin_addr.s_addr, serv_hostinfo->h_addr_list[0],\
                                                      serv_hostinfo->h_length);
  sockadrs_serv.sin_port=htons(2424);
  /* On prépare la question à poser dans buffer_out et taille_question */
  if (sendto(desc_sockcli, buffer_out, taille_question, 0, \
          (struct sockaddr *)&sockadrs_serv, sizeof(struct sockaddr_in)) <  0)
  { traitement de l'erreur et on quitte }
  taille_reponse =recvfrom(desc_sockcli, buffer_in, sizeof(buffer_in),0,NULL,0);
  if (taille_msg_recu<0)
  { traitement de l'erreur et on quitte }

  /* On traite la réponse... */
  close(desc_sockcli);
  exit(0);
}
```

Les APIs réseau UNIX [13/27]

- Structure d'un serveur TCP en C :

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <string.h>
#include  <errno.h>

#include  <netdb.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>

int       main(int argc, char* argv[])
{
  int                     desc_sockserv;
  int                     desc_connexion;
  struct sockaddr_in      sockadrs_serv;
  char                    question[128];
  int                     taille_question;
  char                    reponse[128];
  int                     taille_reponse;

  desc_sockserv=socket(AF_INET, SOCK_STREAM, 0);
  if (desc_sockserv < 0)
  { Traitement de l'erreur et on quitte }
                                                    .../...
```

Les APIs réseau UNIX [14/27]

```
[…]
bzero(&sockadrs_serv, sizeof(struct sockaddr_in));
sockadrs_serv.sin_family=PF_INET;
sockadrs_serv.sin_addr.s_addr=htonl(INADDR_ANY);
sockadrs_serv.sin_port=htons(8424);
if (bind(desc_sockserv, (struct sockaddr *) &sockadrs_serv, \
      sizeof(struct sockaddr_in)) < 0)
{ Traitement de l'erreur et on quitte }

if (listen(desc_sockserv,5) < 0)
{ Traitement de l'erreur et on quitte }

while (1)
{
  desc_connexion=accept(desc_sockserv, (struct sockaddr *)NULL, NULL);
  if (desc_connexion < 0)
  { Traitement de l'erreur et on quitte }
  taille_question=read(desc_connexion, question, sizeof(question));
  if (taille_question  > 0)
  {
    /* [...] on calcule la réponse dans reponse & taille_reponse */
    if (write(desc_connexion, reponse, taille_reponse) < 0)
    { Traitement de l'erreur et on quitte }

  }
  close(desc_connexion);
}
}
```

Les APIs réseau UNIX [15/27]

- Structure d'un <u>client TCP</u> en C :

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <string.h>
#include  <errno.h>

#include  <netdb.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>

int       main(int argc, char* argv[])
{
  int                      desc_sockcli;
  struct hostent           *serv_hostinfo;
  struct sockaddr_in       sockadrs_serv;
  char                     question[128;
  int                      taille_question;
  char                     reponse[128];
  int                      taille_reponse;

  desc_sockcli=socket(AF_INET, SOCK_STREAM, 0);
  { traitement de l'erreur et on quitte }

                                                  .../...
```

Les APIs réseau UNIX [16/27]

```
  [...]
  bzero(&sockadrs_serv, sizeof(struct sockaddr_in));
  serv_hostinfo=gethostbyname("nom.serveur.com");
  if (serv_hostinfo == NULL)
  { On traite l'erreur et on quitte }

  sockadrs_serv.sin_family=serv_hostinfo->h_addrtype;
  memcpy((char *)&sockadrs_serv.sin_addr.s_addr, serv_hostinfo->h_addr_list[0],\
                                                     serv_hostinfo->h_length);
  sockadrs_serv.sin_port=htons(8424);
  if (connect(desc_sockcli, (struct sockaddr *)&sockadrs_serv,\
                                              sizeof(sockadrs_serv)) < 0)
  { On traite l'erreur et on quitte }

  /* [...] On prépare la question à poser dans question et taille_question */
  if (write(desc_sockcli, question, taille_question) < 0)
  { On traite l'erreur et on quitte }

  taille_reponse=read(desc_sockcli, reponse, TAILLE_MAX);
  if (taille_reponse < 0)
  { On traite l'erreur et on quitte }

  /* [...] on traite la réponse */
  close(desc_sockcli);
  exit(0);
}
```

Les APIs réseau UNIX [17/27]

- Rappel : compilation en C
  - Programme en C « test.c » :

```
#include <stdio.h>
#include ...

int main(int argc, char*argv[])
{
    code blablabla...;
}
```

  - Compilation :

```
gcc test.c -o test
```

  - Exécution :

```
./test
```

Les APIs réseau UNIX [18/27]

- Les sockets en JAVA :
  http://pages.univ-nc.nc/~tourai/Java/node44.html
  - Deux classes JAVA pour la communication par UDP (mode datagramme) :
    - la class **DatagramPacket** en charge de l'encapsulation des données en paquets :

```
 public final class DatagramPacket
{
 public DatagramPacket(byte ibuf[], int ilength)
 public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)
 public synchronized InetAddress getAddress()
 public synchronized int getPort()
 public synchronized byte[] getData()
 public synchronized int getLength()
 public synchronized void setAddress(InetAddress iaddr)
 public synchronized void setPort(int iport)
 public synchronized void setData(byte ibuf[])
 public synchronized void setLength(int ilength)
}
```

Les APIs réseau UNIX [19/27]

- la class **DatagramSocket** qui envoie et reçoit les paquets fabriqués avec DatagramPacket :

```
Class java.net.DatagramSocket
{
 public DatagramSocket() throws SocketException
 public DatagramSocket(int port) throws SocketException
 public DatagramSocket(int port, InetAddress laddr) throws SocketException
 public void send(DatagramPacket p) throws IOException
 public synchronized void receive(DatagramPacket p) throws IOException
 public InetAddress getLocalAddress()
 public int getLocalPort()
 public synchronized void setSoTimeout(int timeout) throws SocketException
 public synchronized int getSoTimeout() throws SocketException
 public void close()
}
```

Les APIs réseau UNIX [20/27]

- Structure d'un serveur UDP en JAVA :

```
import java.io.*;
import java.net.*;
public class UdpTestEcho {
 public static void main(String[] args) throws Exception {
  String jeReçois;
  DatagramSocket socket = new DatagramSocket(2424);
  while (true) {
   byte[] buf = new byte[56];
   DatagramPacket paquet  = new DatagramPacket(buf, buf.length);
   socket.receive(paquet);
   jeReçois = new String(paquet.getData(), 0, paquet.getLength());
   System.out.println("j'ai recu : " + jeReçois + " " + jeReçois.length());
   InetAddress address = paquet.getAddress();
   int port = paquet.getPort();
   paquet = new DatagramPacket(buf, buf.length, address, port);
   socket.send(paquet);
  }
 }
}
```

Les APIs réseau UNIX [21/27]

- Structure d’un client UDP en JAVA :

```
import java.io.*;
import java.net.*;
public class Client {
 public static void main(String[] args) throws Exception, NumberFormatException {
  Socket serveur = new Socket("nom.machine.com", 2424);
  PrintWriter Sout = new PrintWriter(serveur.getOutputStream());
  BufferedReader Sin = new BufferedReader(new \
                              InputStreamReader(serveur.getInputStream()));
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  String jEnvoie, jeReçois;
  do
  {
   jEnvoie = in.readLine();
   Sout.println(jEnvoie);
   Sout.flush();
   jeReçois = Sin.readLine();
   System.out.println(jeReçois);
  } while (! jEnvoie.equals("fin"));
  Sin.close();
  Sout.close();
  serveur.close();
 }
}
```

Les APIs réseau UNIX [22/27]

- Deux classes JAVA pour la communication par TCP (mode connecté) :
  - la classe **Socket** qui permet d'initialiser la connexion avec un serveur

```
public class Socket {
 protected Socket()
 protected Socket(SocketImpl impl) throws SocketException
 public Socket(String host, int port) throws UnknownHostException, IOException
 public Socket(InetAddress address, int port) throws IOException
 public Socket(String host, int port, InetAddress localAddr, int localPort) \
                                                             throws IOException
 public Socket(InetAddress address, int port, InetAddress localAddr, int localPort)\
                                                             throws IOException
 public Socket(String host, int port, boolean stream) throws IOException
 public Socket(InetAddress host, int port, boolean stream) throws IOException
 public InetAddress getInetAddress()
 public InetAddress getLocalAddress()
 public int getPort()
 public int getLocalPort()
                                              .../...
```

Les APIs réseau UNIX [23/27]

```
[…]
    public InputStream getInputStream() throws IOException
    public OutputStream getOutputStream() throws IOException
    public void setTcpNoDelay(boolean on) throws SocketException
    public boolean getTcpNoDelay() throws SocketException
    public void setSoLinger(boolean on, int val) throws SocketException
    public int getSoLinger() throws SocketException
    public void setSoTimeout(int timeout) throws SocketException
    public int getSoTimeout() throws SocketException
    public void close() throws IOException
    public String toString()
    public static void setSocketImplFactory(SocketImplFactory fac) \
                                                              throws IOException
   }
```

Les APIs réseau UNIX [24/27]

- la classe **ServerSocket** qui permet de créer un serveur à l'écoute des connexions à venir :

```
public class ServerSocket {
 public ServerSocket(int port) throws IOException
 public ServerSocket(int port, int backlog) throws IOException
 public ServerSocket(int port, int backlog, InetAddress bindAddr) \
                                                throws IOException
 public InetAddress getInetAddress()
 public int getLocalPort()
 public Socket accept() throws IOException
 protected final void implAccept(Socket s) throws IOException
 public void close() throws IOException
 public void setSoTimeout(int timeout) throws SocketException
 public int getSoTimeout() throws IOException
 public String toString()
 public static void setSocketFactory(SocketImplFactory fac) throws IOException
}
```

Les APIs réseau UNIX [25/27]

- Structure d'un serveur TCP en JAVA :

```
import java.io.*;
import java.net.*;

public class TestEcho {
 public static void main(String[] args) throws Exception {
  ServerSocket serveur = new ServerSocket(8424);
  while (true) {
   Socket client = serveur.accept();
   PrintWriter Cout = new PrintWriter(client.getOutputStream());
   BufferedReader Cin = new BufferedReader(new \
                    InputStreamReader(client.getInputStream()));
   String jeReçois;
   do  {
    jeReçois = Cin.readLine();
    Cout.println("Vous avez dit " + jeReçois);
    Cout.flush();
   } while (! jeReçois.equals("fin"));
   Cin.close();
   Cout.close();
   client.close();
  }
 }
}
```

Les APIs réseau UNIX [26/27]

- Structure d'un client TCP en JAVA :

```
import java.io.*;
import java.net.*;
public class Client {
 public static void main(String[] args) throws Exception, NumberFormatException {
  Socket serveur = new Socket("nom.machine.com", 8424);
  PrintWriter Sout = new PrintWriter(serveur.getOutputStream());
  BufferedReader Sin = new BufferedReader(new \
I                  nputStreamReader(serveur.getInputStream()));
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  String jEnvoie, jeReçois;
  do  {
   jEnvoie = in.readLine();
   Sout.println(jEnvoie);
   Sout.flush();
   jeReçois = Sin.readLine();
   System.out.println(jeReçois);
  } while (! jEnvoie.equals("fin"));
  Sin.close();
  Sout.close();
  serveur.close();
 }
}
```

Les APIs réseau UNIX [27/27]

- Rappel : compilation en JAVA
  - Programme en JAVA « test.java » :

```
import java.applet.Applet;

public class test extends Applet
{
  public static void main (String [ ] args)
  {
   code blablabla...;
  }
}
```

  - Compilation :

```
javac test.java
```

  - Exécution :

```
java -cp . test
```

Première séance TP [1/3]

- TP 1 – programmer (en C ou en JAVA) :
  - un serveur en mode datagramme, qui attend des requêtes en UDP sur le port 2424. Pour chaque entier reçu, retourne en réponse un entier valant le double de celui reçu ;
  - un client, qui envoie au serveur décrit précédemment un entier, récupère la réponse, et affiche le résultat.

Première séance TP [2/3]

- TP 2 – programmer (en C ou en JAVA) :
  - un serveur TCP, à l’écoute sur le port 8424. Attend en entrée une (et une seule) chaîne de caractères, et retourne la chaîne en ordre inverse ;
  - un client TCP, qui envoie une requête au serveur précédent, récupère la réponse, et affiche le résultat

Première séance TP [3/3]

- Réflexions sur ces TPs :
  - Quels sont les problèmes rencontrés (hors pb liés aux langages de programmation) ? Pour vous aider à les découvrire :
    - Quels sont les questions que vous vous êtes posés à la lecture du sujet ?
    - Interconnectez un client développé par un groupe avec un serveur développé par d'autres. Cela fonctionne-t-il ?
    - Simulez (à l'aide de la fonction **`sleep()`**) un traitement long sur le serveur, et lancez simultanément 1, puis 2, ... puis 6 clients simultanément. Que se passe-t-il ?
    - Lancez un client avant le serveur. Que se passe-t-il ?
    - Lancez serveurs sur la même machine. Que se passe-t-il ?
    - Simulez l'arrêt du serveur avant qu'il ne réponde (fonction **`exit()`**). Comment réagit le client ?

Quelques outils système UNIX [1/6]

- Pour éviter qu'un client ou un serveur soit bloqué en attente d'un message qu'il pourrait ne jamais recevoir, il est possible de programmer un « timeout en rendant la lecture du socket non bloquante par un :
  - ```
    #include <fcntl.h>
    [...] fcntl(sock, F_SETFL, O_NONBLOCK | \
                              fcntl(sock, F_GETFL, 0));
    ```
- Ou encore par un :
  - ```
    #include <sys/ioctl.h>
    [...] int on=1;
    [...] ioctl(sock, FIONBIO, &on);
    ```

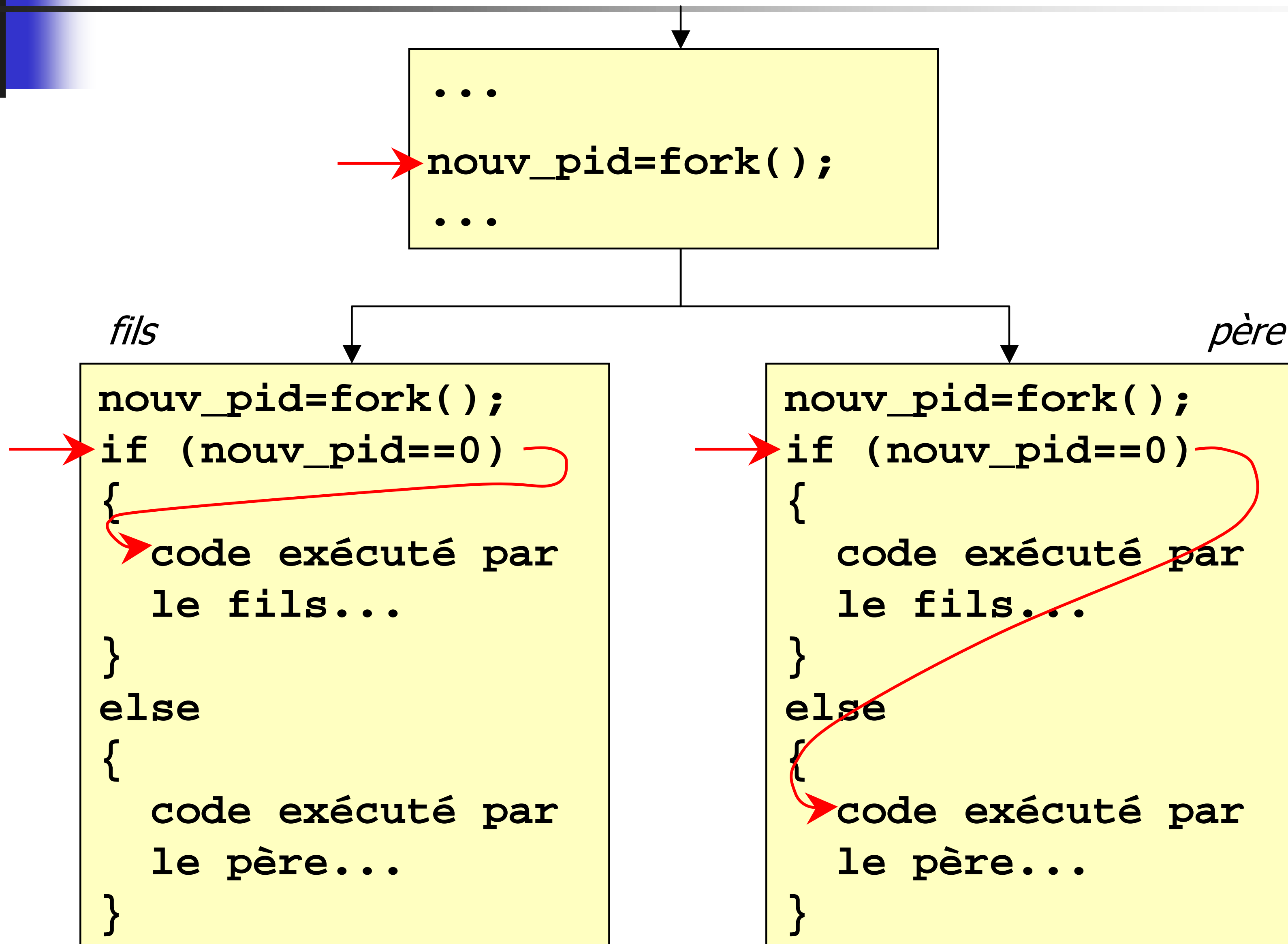
Quelques outils système UNIX [2/6]

- Problème : si le socket est non bloquant, il vous faut alors l'interroger régulièrement pour savoir si vous avez reçu des choses (technique de polling). Cette technique engendre une perte de temps CPU non négligeable ➔
  Polling = technique à proscrire
- Solution élégante : utiliser la fonction :
  - `select()` (Cf. « man select »)

Quelques outils système UNIX [3/6]

- Pour éviter que le serveur soit bloqué par la requête d'un client, en général, le serveur délègue le travail à réaliser à un autre processus (i.e. à un autre programme qui tournera en parallèle), afin de retourner le plus vite possible à l'écoute d'une autre requête provenant d'un autre client. La fonction pour créer un autre processus est :
  - `fork()` (Cf. « man fork »)

Quelques outils système UNIX [4/6]



Quelques outils système UNIX [5/6]

- Après le `fork()`, le fils traitera la requête, alors que le père fermera le socket ouvert par `accept()` et retournera à l'écoute d'une autre requête.
- Problème : lorsque le fils a fini son traitement, le système ne libère pas les ressources (mémoire, etc.) qu'il possède. On dit alors que le processus reste à l'état de « zombie ».

Quelques outils système UNIX [6/6]

- Solution : utiliser **`signal()`** et **`waitpid()`**

```
#include <sys/wait.h>
#include <signal.h>

[...] void signal_enfant(int non_utilise)
{
    /* Nettoyage des processus fils */
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

[...] int main(int argc, char *argv[])
{
    ...
    signal(SIGCHLD, signal_enfant);
    ...
    if (fork() == 0)
    ...
}
```

Deuxième séance TP [1/1]

- TP 3 – corriger le code du TP n°2 en fonction des derniers éléments de programmation système qui viennent d’être fournis, afin de palier aux problèmes qui ont été identifiés