L'API socket en résumé

le minimum pour communiquer par réseau (sans toutefois y comprendre grand chose)

Fabrice HARROUET

É*cole* **N***ationale d'***I***ngénieurs de* **B***rest*

Propos

Programmation des mécanismes de communication par réseau

- Échanges de messages individuels en *UDP*
- Flots bidirectionnels en *TCP*

Mise en œuvre dans différents langages

- Les principes sous jacents sont les mêmes $\forall$ outil/langage/système
- Seuls *C*, *Python* et *Java* sont traités ici

Avertissements

Ce document n'explique rien !

- Il ne donne que des *"recettes"* à appliquer telles quelles
  - Essentiellement à base d'exemples
- Pour en savoir plus, voir les documents (cf. page perso.) :
  - "API *socket* et programmation réseau"
  - "Les entrées/sorties"

Les anomalies ont été volontairement ignorées !

- Nous ne cherchons à réaliser que quelques exemples illustratifs
- Des indications sont toutefois données à la fin du document

Table des matières

Principe d'UDP

Échange de messages distincts (délivrance non garantie)

- Métaphore de la *Poste* : un message = une lettre + une destination

Envoi d'un message

- Création d'une *socket internet* de type *datagram*
- Envoi à une *adresse IP* et un *numéro de port* choisis

Réception d'un message

- Création d'une *socket internet* de type *datagram*
  - Attachement à un *numéro de port* choisi explicitement
- Réception du prochain message envoyé à cette *socket*

Envoi d'un message UDP en C

```
//---- extract destination IP address ----
struct hostent *host=gethostbyname("destination.there.org");
struct in_addr ipAddress=*((const struct in_addr *)(host->h_addr));


//---- create UDP socket ----
int udpSocket=socket(PF_INET,SOCK_DGRAM,0);


//---- send message to the specified destination/port ----
char msg[]="Hello";
struct sockaddr_in toAddr;              toAddr.sin_family=AF_INET;
toAddr.sin_port=htons(portNumber);      toAddr.sin_addr=ipAddress;
sendto(udpSocket,msg,strlen(msg),0,
       (const struct sockaddr *)&toAddr,sizeof(toAddr));


//---- close UDP socket ----
close(udpSocket);
```

Envoi d'un message UDP en Python

```
#---- extract destination IP address ----
ipAddress=socket.gethostbyname("destination.there.org")


#---- create UDP socket ----
udpSocket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0)


#---- send message to the specified destination/port ----
msg='Hello'
toAddr=(ipAddress,portNumber)
udpSocket.sendto(msg,toAddr)


#---- close UDP socket ----
udpSocket.close()
```

Envoi d'un message UDP en Java

```
//---- extract destination IP address ----
InetAddress ipAddress=InetAddress.getByName("destination.there.org");


//---- create UDP socket ----
DatagramSocket udpSocket=new DatagramSocket(null);


//---- send message to the specified destination/port ----
String msg="Hello";
byte[] data=msg.getBytes();
InetSocketAddress toAddr=new InetSocketAddress(ipAddress,portNumber);
DatagramPacket pkt=new DatagramPacket(data,data.length,toAddr);
udpSocket.send(pkt);


//---- close UDP socket ----
udpSocket.close();
```

Réception d'un message UDP en C

```
//---- create UDP socket ----
int udpSocket=socket(PF_INET,SOCK_DGRAM,0);
// ... bound to any local address on the specified port
struct sockaddr_in myAddr;              myAddr.sin_family=AF_INET;
myAddr.sin_port=htons(portNumber);      myAddr.sin_addr.s_addr=htonl(INADDR_ANY);
bind(udpSocket,(const struct sockaddr *)&myAddr,sizeof(myAddr));


//---- receive message ----
char msg[0x100]; struct sockaddr_in fromAddr; socklen_t len=sizeof(fromAddr);
int nb=recvfrom(udpSocket,msg,0xFF,0,(struct sockaddr *)&fromAddr,&len);

//---- display message and source address/port ----
msg[nb]='\0';
printf("from %s:%d : %s\n",
       inet_ntoa(fromAddr.sin_addr),ntohs(fromAddr.sin_port),msg);


//---- close UDP socket ----
close(udpSocket);
```

Réception d'un message UDP en Python

```
#---- create UDP socket ----
udpSocket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0)
# ... bound to any local address on the specified port
myAddr=('',portNumber)
udpSocket.bind(myAddr)


#---- receive message ----
(msg,fromAddr)=udpSocket.recvfrom(0x100)

#---- display message and source address/port ----
print 'from %s:%d : %s'%(fromAddr[0],fromAddr[1],msg)


#---- close UDP socket ----
udpSocket.close()
```

Réception d'un message UDP en Java

```
//---- create UDP socket ----
DatagramSocket udpSocket=new DatagramSocket(null);
// ... bound to any local address on the specified port
InetSocketAddress myAddr=new InetSocketAddress((InetAddress)null,portNumber);
udpSocket.bind(myAddr);


//---- receive message ----
byte[] data=new byte[0x100];
DatagramPacket pkt=new DatagramPacket(data,data.length);
udpSocket.receive(pkt);
InetSocketAddress fromAddr=(InetSocketAddress)pkt.getSocketAddress();

//---- display message and source address/port ----
String msg=new String(pkt.getData(),0,pkt.getLength());
System.out.println("from "+fromAddr.getAddress().getHostAddress()+
                   ":"+fromAddr.getPort()+" --> "+msg);


//---- close UDP socket ----
udpSocket.close();
```

Diffusion de messages UDP

Un même message reçu par plusieurs destinataires

- Le message ne circule qu'une fois (très efficace)
  - Pas besoin de le répeter pour chaque destinataire
- Émetteur et destinataires dans le même sous-réseau

Programmation de la *socket*

- Destination : *adresse de diffusion* du sous-réseau
  - Il faut la connaître ou savoir la déterminer
    (exemple : `192.168.1.255` pour le sous-réseau `192.168.1.0/24`)
  - Ou alors adresse de diffusion générique `255.255.255.255`
- Activer une option sur la *socket* pour autoriser la diffusion

Diffusion de messages UDP

En *C*

```
struct hostent *host=gethostbyname("255.255.255.255");
struct in_addr broadcastAddress=*((const struct in_addr *)(host->h_addr));

int on=1;
setsockopt(udpSocket,SOL_SOCKET,SO_BROADCAST,(const char *)&on,sizeof(on));
```

En *Python*

```
broadcastAddress=socket.gethostbyname('255.255.255.255')

udpSocket.setsockopt(socket.SOL_SOCKET,socket.SO_BROADCAST,1)
```

En *Java*

```
InetAddress broadcastAddress=InetAddress.getByName("255.255.255.255");

udpSocket.setBroadcast(true);
```

Table des matières

Principe de TCP

Établissement de flots de données (délivrance garantie)

- Métaphore du téléphone : connexion puis dialogue
  - Deux interlocuteurs identifiés pour toute la durée de communication
- Rôles dissymétriques : architecture *client/serveur*

Rôle client

- Création d'une *socket internet* de type *stream*
  - *Connexion* à une *adresse IP* et un *numéro de port* choisis
- *Dialogue* bidirectionnel avec le *serveur* contacté

Rôle serveur

- Création d'une *socket internet* de type *stream*
  - *Écoute* sur un *numéro de port* choisi
- *Acceptation* des connexions des clients
  - *Dialogue* bidirectionnel avec chacune des connexions acceptées

Client TCP en C

```
//---- extract destination IP address ----
struct hostent *host=gethostbyname("destination.there.org");
struct in_addr ipAddress=*((const struct in_addr *)(host->h_addr));


//---- create client socket ----
int clientSocket=socket(PF_INET,SOCK_STREAM,0);
// ... connected to the specified destination/port
struct sockaddr_in toAddr;              toAddr.sin_family=AF_INET;
toAddr.sin_port=htons(portNumber);      toAddr.sin_addr=ipAddress;
connect(clientSocket,(const struct sockaddr *)&toAddr,sizeof(toAddr));


//---- dialog with server ----
...


//---- close client socket ----
close(clientSocket);
```

Client TCP en Python

```
#---- extract destination IP address ----
ipAddress=socket.gethostbyname("destination.there.org")


#---- create client socket ----
clientSocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM,0)
# ... connected to the specified destination/port
toAddr=(ipAddress,portNumber)
clientSocket.connect(toAddr)


#---- dialog with server ----
...


#---- close client socket ----
clientSocket.close()
```

Client TCP en Java

```
//---- extract destination IP address ----
InetAddress ipAddress=InetAddress.getByName("destination.there.org");


//---- create client socket ----
Socket clientSocket=new Socket();
// ... connected to the specified destination/port
InetSocketAddress toAddr=new InetSocketAddress(ipAddress,portNumber);
clientSocket.connect(toAddr);


//---- dialog with server ----
...


//---- close client socket ----
clientSocket.close();
```

Serveur TCP en C

```
//---- create listen socket ----
int listenSocket=socket(PF_INET,SOCK_STREAM,0);
// ... bound to any local address on the specified port
struct sockaddr_in myAddr;            myAddr.sin_family=AF_INET;
myAddr.sin_port=htons(portNumber);    myAddr.sin_addr.s_addr=htonl(INADDR_ANY);
bind(listenSocket,(const struct sockaddr *)&myAddr,sizeof(myAddr));
// ... listening connections
listen(listenSocket,10);

...
    //---- accept new connection ----
    struct sockaddr_in fromAddr; socklen_t len=sizeof(fromAddr);
    int dialogSocket=accept(listenSocket,(struct sockaddr *)&fromAddr,&len);
    printf("new connection from %s:%d\n",
           inet_ntoa(fromAddr.sin_addr),ntohs(fromAddr.sin_port));

    //---- dialog with client ----
    ...

    //---- close dialog socket ----
    close(dialogSocket);

//---- close listen socket ----
close(listenSocket);
```

Serveur TCP en Python

```
#---- create listen socket ----
listenSocket=socket.socket(socket.AF_INET,socket.SOCK_STREAM,0)
# ... bound to any local address on the specified port
myAddr=('',portNumber)
listenSocket.bind(myAddr)
# ... listening connections
listenSocket.listen(10)


...
    #---- accept new connection ----
    (dialogSocket,fromAddr)=listenSocket.accept()
    print 'new connection from %s:%d'%(fromAddr[0],fromAddr[1])

    #---- dialog with client ----
    ...

    #---- close dialog socket ----
    dialogSocket.close()

#---- close listen socket ----
listenSocket.close()
```

Serveur TCP en Java

```
//---- create listen socket ----
ServerSocket listenSocket=new ServerSocket();
// ... bound to any local address on the specified port, and listening
InetSocketAddress myAddr=new InetSocketAddress((InetAddress)null,portNumber);
listenSocket.bind(myAddr,10);

...
    //---- accept new connection ----
    Socket dialogSocket=listenSocket.accept();
    InetSocketAddress fromAddr=(InetSocketAddress)
                               dialogSocket.getRemoteSocketAddress();
    System.out.println("new connection from "+
                       fromAddr.getAddress().getHostAddress()+":"+
                       fromAddr.getPort());

    //---- dialog with client ----
    ...

    //---- close dialog socket ----
    dialogSocket.close();

//---- close listen socket ----
listenSocket.close();
```

Écriture dans un flot TCP

En *C*

```
char msg[]="Hello";
send(dialogSocket,msg,strlen(msg),0);
```

En *Python*

```
msg='Hello'
dialogSocket.send(msg)
```

En *Java*

```
String msg="Hello";
byte[] data=msg.getBytes();
OutputStream output=clientSocket.getOutputStream();
output.write(data);
```

Lecture depuis un flot TCP

En *C*

```
char msg[0x100];
int nb=recv(dialogSocket,msg,0xFF,0);
if(nb>0)
  { msg[nb]='\0'; printf("msg=%s\n",msg); }
```

En *Python*

```
msg=dialogSocket.recv(0x100)
if msg:
  print 'msg=%s'%msg
```

En *Java*

```
byte[] data=new byte[0x100];
InputStream input=dialogSocket.getInputStream();
int nb=input.read(data);
if(nb>0)
  { String msg=new String(data,0,nb); System.out.println("msg="+msg); }
```

Problème de relance d'un serveur TCP

En cas d'arrêt brutal d'un serveur (plantage)

- Le port d'écoute reste indisponible quelques temps (état `TIME_WAIT` pour les paquets retardataires)
- Le serveur ne pourra pas être relancé immédiatement !!!
  - Nécessité d'attendre quelques secondes/minutes (selon le système)
  - Ou alors choisir un autre port d'écoute

Possibilité d'autoriser la relance immédiate

- Option lors de la création de la socket d'écoute (avant `bind()`)
- En *C* :

```
int on=1;
setsockopt(listenSocket,SOL_SOCKET,SO_REUSEADDR,
           (const char *)&on,sizeof(on));
```

- En *Python* :

```
listenSocket.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
```

- En *Java* :

```
listenSocket.setReuseAddress(true);
```

Table des matières

Servir plusieurs clients TCP simultanément

La solution simpliste précédente est trop restrictive

- Les clients ne peuvent être servis que l'un après l'autre
  - Très pénalisant en cas de long dialogue (transfert volumineux ...)
- Très dépendante du bon comportement des clients
  - Si un client se connecte mais ne dialogue pas !?!?
  - Si un client ne se déconnecte pas !?!?

Contraintes sur l'architecture du serveur

- Pouvoir accepter à tout moment de nouvelles connexions
  - Même si des dialogues sont déjà en cours avec d'autres clients
- Dialoguer avec chaque client au rythme qui lui convient
- Précautions nécessaires autour des opérations bloquantes (`accept()`, `recv()`, ...)
  - Ne pas rester bloqué dans une alors que d'autres peuvent avoir lieu

Serveur TCP à scrutation passive

Scrutation passive avec l'appel système `select()`

- Fabriquer un ensemble de ressources à surveiller (*sockets* ou autres)
- Demander au système de surveiller l'ensemble
  - Appel bloquant, le processus est en sommeil pendant ce temps
  - Évite un gaspillage de calcul et d'énergie
- Au réveil, le système nous indique quels ressources sont prêtes

Exploitation des ressources prêtes

- L'opération souhaitée sur chacune d'elles ne sera pas bloquante
  - Parce que les circonstances attendues sont présentes (succès immédiat)
  - Ou parce qu'une erreur a été détectée (échec immédiat) !
- Seule la **prochaine** opération ne sera pas bloquante
  - Une seconde tentative le serait !
  - Certains recommandent de configurer les *sockets* en *non-bloquantes*
    - Essentiellement en cas de maladresse de programmation

Serveur TCP à scrutation passive en C

```
int clients[0x100]; int nbClients=0;
for(;;)
  {
  fd_set rdSet; FD_ZERO(&rdSet); // empty file-descriptor set
  FD_SET(listenSocket,&rdSet);   // watch listen socket
  int i, maxFd=listenSocket;
  for(i=0;i<nbClients;++i) // watch every dialog socket
    { FD_SET(clients[i],&rdSet); if(clients[i]>maxFd) maxFd=clients[i]; }
  //---- wait for new connections or incoming messages ----
  select(maxFd+1,&rdSet,NULL,NULL,NULL);

  if(FD_ISSET(listenSocket,&rdSet)) //---- accept and store new connection ----
    { int dialogSocket=accept(listenSocket,NULL,NULL);
      clients[nbClients++]=dialogSocket; }
  for(i=nbClients;i--;)
    {
    if(FD_ISSET(clients[i],&rdSet))
      { ... //---- read client message and send reply ----
        if(recv_result<=0) // manage disconnection
          { close(clients[i]); clients[i]=clients[--nbClients]; }
      }
    }
  }
```

Serveur TCP à scrutation passive en Python

```
clients=[]
while True:
  rdSet=[listenSocket]+clients; # watch listen socket and every dialog socket
  #---- wait for new connections or incoming messages ----
  (readyR,readyW,readyE)=select.select(rdSet,[],[])


  for s in readyR:
    if s is listenSocket: #---- accept and store new connection ----
      (dialogSocket,fromAddr)=listenSocket.accept()
      clients.append(dialogSocket)
    else:
      ... #---- read client message and send reply ----
      if not recv_result: # manage disconnection
        s.close()
        clients.remove(s)
```

Serveur TCP à scrutation passive en Java

```
// in Java, a Selector object does not work directly on sockets or devices
// but on Channel objects (ServerSocketChannel and SocketChannel here)
ServerSocketChannel listenChannel=ServerSocketChannel.open();
ServerSocket listenSocket=listenChannel.socket();
// configure listenSocket as usual


Selector selector=Selector.open();
listenChannel.configureBlocking(false); // mandatory in Java !
listenChannel.register(selector,SelectionKey.OP_ACCEPT); // watch listen socket
for(;;)
  {
  //---- wait for new connections or incoming messages ----
  selector.select();
```

Serveur TCP à scrutation passive en Java

```
Iterator it=selector.selectedKeys().iterator();
while(it.hasNext())
  {
  SelectionKey selKey=(SelectionKey)it.next();
  it.remove();
  if(selKey.channel()==listenChannel)
    {                                //---- accept and store new connection ----
    SocketChannel dialogChannel=listenChannel.accept();
    dialogChannel.configureBlocking(false);
    dialogChannel.register(selector,
                           SelectionKey.OP_READ); // watch new dialog socket
    }
  else
    {
    SocketChannel dialogChannel=(SocketChannel)selKey.channel();
    //---- read client message and send reply ----
    //   needs a ByteBuffer object with clear() flip() get() put() ...
    //   to read() write() the dialog channel.  Good luck !
    if(read_result==-1) // manage disconnection
      { dialogChannel.close(); }
    }
  }
}
```

Serveur TCP multi-threads

Chaque traitement bloquant dans un *thread* distinct

- Le programme principal ne fait qu'accepter les nouvelles connexions
- Chaque dialogue a lieu dans un nouveau *thread*
  - Autant de traitements en parallèles que de clients
  - Chacun peut rester bloquer sans bloquer l'ensemble
- Facilité d'écriture des dialogues avec les clients

Limitations

- Très peu performant en cas de forte charge (nombreux clients)
  - Multiples piles d'exécution et entrées dans l'ordonnanceur
- Mal adapté aux dialogues courts
  - Plus de temps pour démarrer/terminer le *thread* que pour le traitement
- Synchronisation de l'accès aux données communes !
  - Espace d'adressage commun à tous les *threads* d'un même processus
  - Les données propres à chaque *thread* doivent être dupliquées

Serveur TCP multi-threads en C

```
void *
dialogThread(void *arg)
{
pthread_detach(pthread_self()); // no need for join() at termination
int dialogSocket=*(int *)arg; // this pointer was an int * at thread creation
free(arg);
//---- dialog with client ----
...
close(dialogSocket);
return NULL;
}


for(;;)
  {
  //---- accept new connection ----
  int *dialogSocket=(int *)malloc(sizeof(int));
  *dialogSocket=accept(listenSocket,NULL,NULL);
  //---- start a new dialog thread ----
  pthread_t th;
  pthread_create(&th,NULL,dialogThread,dialogSocket);
  }
```

Serveur TCP multi-threads en Python

```
def dialogThread(dialogSocket):
  #---- dialog with client ----
  ...
  dialogSocket.close()


...
while True:
  #---- accept new connection ----
  (dialogSocket,fromAddr)=listenSocket.accept()
  #---- start a new dialog thread ----
  th=threading.Thread(target=dialogThread,args=(dialogSocket,))
  th.setDaemon(True) # process will not wait for thread termination
  th.start()
```

Serveur TCP multi-threads en Java

```
public class TcpThreadServer implements Runnable
{
Socket dialogSocket;
TcpThreadServer(Socket dialogSocket) { this.dialogSocket=dialogSocket; }
public void run()
  {
  try {
      //---- dialog with client ----
      ...
      dialogSocket.close();
      } catch(Throwable t) { t.printStackTrace(); }
  }


...
for(;;)
  {
  //---- accept new connection ----
  Socket dialogSocket=listenSocket.accept();
  //---- start a new dialog thread ----
  Thread th=new Thread(new TcpThreadServer(dialogSocket));
  th.setDaemon(true); // process will not wait for thread termination
  th.start();
  }
```

Serveur TCP multi-processus

Chaque traitement bloquant dans un *processus* distinct

- Même idée générale que pour la solution multi-threads
  - Solution historique, avant les *threads*
- Duplication de l'espace d'adressage
  - Plus grande consommation de mémoire
  - Bon cloisonnement des traitements
    (pas de communication implicite, plantages isolés)
  - Bien adapté au recouvrement de processus (`execve()`)
  - Bien adapté aux redirections d'entrées/sorties (`dup2()`)

Création d'un processus enfant avec l'appel système `fork()`

- Processus enfant $\simeq$ copie conforme du parent
  - Fermer la *socket de dialogue* dans le parent
  - Fermer la *socket d'écoute* dans l'enfant
- Aucun équivalent en *Java*

Serveur TCP multi-processus en C

```
void waitChildren(int signum) // automatic wait() at child termination
{ int r; do { r=waitpid(-1,NULL,WNOHANG); } while(r>0); }

...
struct sigaction act;           memset(&act,0,sizeof(struct sigaction));
act.sa_handler=waitChildren;    sigaction(SIGCHLD,&act,NULL);
for(;;)
  {
  //---- accept new connection ----
  int dialogSocket=accept(listenSocket,NULL,NULL);
  //---- start a new dialog process ----
  if(!fork()) // in child process
    {
    close(listenSocket) // useless for child process
    //---- dialog with client ----
    ...
    close(dialogSocket);
    exit(0) // child process termination
    }
  else // in parent process
    { close(dialogSocket); } // useless for parent process
  }
```

Serveur TCP multi-processus en Python

```
def waitChildren(signum,frame): # automatic wait() at child termination
  try:
    while True:
      (pid,status)=os.waitpid(-1,os.WNOHANG);
      if pid<=0: break
  except:
    pass

...
signal.signal(signal.SIGCHLD,waitChildren)
while True:
  #---- accept new connection ----
  (dialogSocket,fromAddr)=listenSocket.accept()
  #---- start a new dialog process ----
  if not os.fork(): # in child process
    listenSocket.close() # useless for child process
    #---- dialog with client ----
    ...
    dialogSocket.close()
    sys.exit(0) # child process termination
  else: # in parent process
    dialogSocket.close() # useless for parent process
```

Table des matières

Du texte ou des données ?

Tout n'est que séquence d'octets !

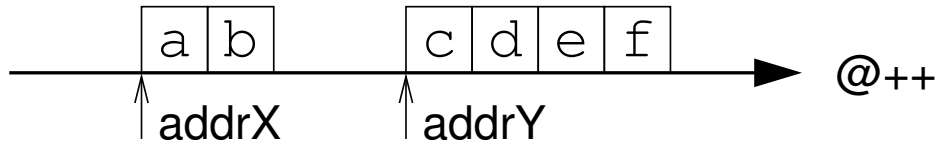
- Les *sockets* ignorent la signification de ces octets
- Ce sont les applications qui en interprètent le contenu
  - Doivent se mettre d'accord sur la signification des échanges
- Chaque caractère d'un texte n'est qu'un octet comme un autre

Exemple : la séquence de 4 octets 65 66 67 68 représente

- la chaîne de caractères `ABCD`
- l'entier de 32 bits `1094861636` (ou `1145258561`)
- les deux entiers de 16 bits `16706` et `17220` (ou `16961` et `17475`)
- le réel `12.141422272` (ou `781.035217285`)
- ...

Ordre des octets

L'interprétation dépend du processeur



- Sur une machine *big-endian*
  - l'entier de 16 bits en `addrX` vaut `(256*a)+b`
  - l'entier de 32 bits en `addrY` vaut `(16777216*c)+(65536*d)+(256*e)+f`
- Sur une machine *little-endian*
  - l'entier de 16 bits en `addrX` vaut `(256*b)+a`
  - l'entier de 32 bits en `addrY` vaut `(16777216*f)+(65536*e)+(256*d)+c`

Données au format *hôte* ou au format *réseau*

- On travaille selon l'ordre *hôte* habituel (spécifique à la machine)
- On communique selon l'ordre *réseau* standard : *big-endian*
  - Les machines *little-endian* doivent donc s'adapter
    - Inverser l'ordre des octets des valeurs de plus de 8 bits
  - Traiter les champs des structures un par un (attention à l'alignement)

Ordres hôte et réseau en C

```
int32_t value=1234;                       // compute a useful 32-bit integer value
value=htonl(value);                       // host-to-network long-int conversion
send(dialogSocket,(const char *)&value,4,0); // send these 4 bytes


int32_t value;                            // prepare a 32-bit integer storage
recv(dialogSocket,(char *)&value,4,0); // receive 4 bytes
value=ntohl(value);                       // network-to-host long-int conversion
...                                       // work with this 32-bit integer value


int16_t values[]={10,20,30,40,50};    // compute 5 useful 16-bit integer values
for(int i=0;i<5;++i)
  values[i]=htons(values[i]);         // host-to-network short-int conversion
send(dialogSocket,(const char *)values,5*2,0); // send these 10 bytes


int16_t values[5];                        // prepare 5 16-bit integer storages
recv(dialogSocket,(char *)values,5*2,0); // receive 10 bytes
for(int i=0;i<5;++i)
  values[i]=ntohs(values[i]);         // network-to-host short-int conversion
...                                       // work with these 5 16-bit integer values
```

Ordres hôte et réseau en Python

```
value=1234                      # compute a useful integer value
data=struct.pack('!l',value); # pack as a network-order 32-bit integer
dialogSocket.send(data);      # send these 4 bytes


data=dialogSocket.recv(4)          # receive 4 bytes
(value,)=struct.unpack('!l',data) # unpack as a network-order 32-bit integer
...                                # work with this integer value


values=(10,20,30,40,50)          # compute 5 useful integer values
data=struct.pack('!5h',*values) # pack as 5 network-order 16-bit integers
dialogSocket.send(data);         # send these 10 bytes


data=dialogSocket.recv(5*2);     # receive 10 bytes
values=struct.unpack('!5h',data) # unpack as 5 network-order 16-bit integers
...                               # work with these 5 integer values
```

Ordres hôte et réseau en Java

```
int value=1234;                          // compute a useful 32-bit integer value
ByteArrayOutputStream baos=new ByteArrayOutputStream();
DataOutputStream dos=new DataOutputStream(baos);
dos.writeInt(value);                  // write as a network-order 32-bit integer
byte[] data=baos.toByteArray();
OutputStream output=dialogSocket.getOutputStream();
output.write(data);                                        // send these 4 bytes


byte[] data=new byte[4];                        // prepare a 32-bit integer storage
InputStream input=dialogSocket.getInputStream();
input.read(data);                                            // receive 4 bytes
ByteArrayInputStream bais=new ByteArrayInputStream(data);
DataInputStream dis=new DataInputStream(bais);
int value=dis.readInt();                // read as a network-order 32-bit integer
...                                        // work with this 32-bit integer value
```

Ordres hôte et réseau en Java

```
short[] values={10,20,30,40,50};     // compute 5 useful 16-bit integer values
ByteArrayOutputStream baos=new ByteArrayOutputStream();
DataOutputStream dos=new DataOutputStream(baos);
for(int i=0;i<5;++i)
  dos.writeShort(values[i]);  // write them as 5 network-order 16-bit integers
byte[] data=baos.toByteArray();
OutputStream output=dialogSocket.getOutputStream();
output.write(data);                                    // send these 10 bytes
```

```
byte[] data=new byte[10];                  // prepare 5 16-bit integer storages
InputStream input=dialogSocket.getInputStream();
input.read(data);                                        // receive 10 bytes
ByteArrayInputStream bais=new ByteArrayInputStream(data);
DataInputStream dis=new DataInputStream(bais);
short[] values=new short[5];
for(int i=0;i<5;++i)
  values[i]=dis.readShort();         // read as 5 network-order 16-bit integers
...                                  // work with these 5 16-bit integer values
```

Table des matières

Résultats des appels systèmes en C

Le résultat principal indique les anomalies

- Généralement un entier qui vaut `-1` en cas d'erreur
  - Le diagnostic est donné par la variable global `errno`
  - Message descriptif par `perror()` ou `strerror()`
- Il **faut** contrôler ce résultat afin de traiter l'anomalie éventuelle
  - Au minimum un message de diagnostic et une terminaison éventuelle
  - Sinon grande complication de la mise au point du programme (il se poursuit quand même, malgré l'échec !)
  - Une *"vraie"* application devrait réagir de manière appropriée (quitter, abandonner, retenter avec des variantes éventuelles . . .)

Ce résultat peut également donner d'autres informations utiles

- Signification très variable selon le rôle de l'appel système
- Descripteur de fichier, quantité de données transmises . . .
  - Les quantités transmises **doivent** être contrôlées attentivement

Résultats des appels systèmes en C

gethostbyname() : pointeur ou NULL

```
// gethostbyname() is not a system call but a function involving
// many subsequent system calls, thus the result is not just an int
// and the diagnostic is not given by errno
struct hostent *host=gethostbyname("destination.there.org");
if(!host) { fprintf(stderr,"unknown host\n",); exit(1); }
struct in_addr ipAddress=*((const struct in_addr *)(host->h_addr));
```

socket() : descripteur de fichier ou -1

```
int udpSocket=socket(PF_INET,SOCK_DGRAM,0);
if(udpSocket==-1) { perror("socket"); exit(1); }

int tcpSocket=socket(PF_INET,SOCK_STREAM,0);
if(tcpSocket==-1) { perror("socket"); exit(1); }
```

Résultats des appels systèmes en C

connect()/setsockopt()/bind()/listen() : 0 en cas de succès ou -1

```
if(connect(clientSocket,(const struct sockaddr *)&toAddr,sizeof(toAddr))==-1)
  { perror("connect"); exit(1); }
if(setsockopt(listenSocket,SOL_SOCKET,SO_REUSEADDR,
              (const char *)&on,sizeof(on))==-1)
  { perror("setsockopt"); exit(1); }
if(bind(listenSocket,(const struct sockaddr *)&myAddr,sizeof(myAddr))==-1)
  { perror("bind"); exit(1); }
if(listen(listenSocket,10)==-1)
  { perror("listen"); exit(1); }
```

accept() : descripteur de fichier ou -1

```
int dialogSocket=accept(listenSocket,(struct sockaddr *)&fromAddr,&len);
if(dialogSocket==-1) { perror("accept"); exit(1); }
```

select() : nombre de descripteurs prêts ou -1

```
if(select(maxFd+1,&rdSet,NULL,NULL,NULL)==-1)
  { perror("select"); exit(1); }
```

Résultats des appels systèmes en C

sendto()/send() : quantité émise ou -1

```
if(sendto(udpSocket,(const char *)data,nb,0,
          (const struct sockaddr *)&toAddr,sizeof(toAddr))==-1)
  { perror("sendto"); exit(1); }

void
sendAll(int dialogSocket,       // this function ensures that the total
        const void *dataToSend, // amount of data has been sent through
        int nbBytesToSend)      // the dialog socket
{
const char *ptr=(const char *)dataToSend;
int remaining=nbBytesToSend;
while(remaining)
  {
  int r=send(dialogSocket,ptr,remaining,0);
  if(r==-1) { perror("send"); exit(1); }
  ptr+=r; remaining-=r;
  }
}
```

Résultats des appels systèmes en C

recvfrom()/recv() : quantité reçue ou 0 (*fin de fichier*) ou -1

```
char buffer[0x100];
int r=recvfrom(udpSocket,buffer,0x100,0,(struct sockaddr *)&fromAddr,&len);
if(r==-1) { perror("recvfrom"); exit(1); }
printf("%d bytes in buffer\n",r);

int
recvAll(int dialogSocket, // this function ensures that the total
        void *buffer,     // amount of data has been received from
        int bufferSize)   // the dialog socket
{
char *ptr=(char *)buffer;
int remaining=bufferSize;
while(remaining)
  {
  int r=recv(dialogSocket,ptr,remaining,0);
  if(r==-1) { perror("recv"); exit(1); }
  if(r) { ptr+=r; remaining-=r; } else break; // end of file
  }
return bufferSize-remaining;
}
```

Résultats et exceptions en Python

Repose fondamentalement sur des appels systèmes en C

- Les résultats sont contrôlés par *Python* et des exceptions sont levées
  - Le programme sera brutalement terminé avec un message d'erreur
    - Le problème ne passera pas inaperçu
  - **!!! Une** *"vraie"* **application devrait les capturer et les traiter !!!** (ajouter de nombreux blocs `try/except`)
- Signification des résultats $\simeq$ celle des appels en *C* sans les erreurs (-1)

Résultats et exceptions en Python

socket.sendto()/socket.send() : quantité émise

```
nbBytesSent=udpSocket.sendto(data,toAddr)

def sendAll(dialogSocket,   # this function ensures that the total
            dataToSend,     # amount of data has been sent through
            nbBytesToSend): # the dialog socket
  nbSent=0
  while nbSent<nbBytesToSend:
    nbSent+=dialogSocket.send(dataToSend[nbSent:])

# nb: the standard socket method dialogSocket.sendall(data)
#     is similar to sendAll(dialogSocket,data,len(data))
```

Résultats et exceptions en Python

socket.recvfrom()/socket.recv() : chaîne reçue ou '' (*fin de fichier*)

```
(buffer,fromAddr)=udpSocket.recvfrom(0x100)
print len(buffer)," bytes in buffer"

def recvAll(dialogSocket, # this function ensures that the total
            bufferSize):  # amount of data has been received from
  result=''               # the dialog socket
  while len(result)<bufferSize:
    r=dialogSocket.recv(bufferSize-len(result));
    if r:
      result+=r
    else:
      break # end of file
  return result
```

Résultats et exceptions en Java

Repose fondamentalement sur des appels systèmes en C

- Les résultats sont contrôlés par *Java* et des exceptions sont levées
  - Le programme sera brutalement terminé avec un message d'erreur
    - Le problème ne passera pas inaperçu
  - **!!! Une** *"vraie"* **application devrait les capturer et les traiter !!!** (ajouter de nombreux blocs `try/catch`)
- Signification des résultats $\neq$ celle des appels en *C*
  - Bien plus que l'accès aux appels systèmes
  - Énormément d'objets à manipuler

Résultats et exceptions en Java

`DatagramSocket.send()` : pas de résultat

```
DatagramPacket pkt=new DatagramPacket(data,data.length,toAddr);
udpSocket.send(pkt);
```

`Socket.getOutputStream().write()` : pas de résultat

```
byte[] data=...
OutputStream output=clientSocket.getOutputStream();
output.write(data); // Java iterates until data.length bytes are sent
```

Résultats et exceptions en Java

DatagramSocket.receive() : mise à jour d'un DatagramPacket

```
byte[] buffer=new byte[0x100];
DatagramPacket pkt=new DatagramPacket(buffer,buffer.length);
udpSocket.receive(pkt);
System.out.println(pkt.getLength()+" bytes in buffer");
```

Socket.getInputStream().read() : quantité reçue ou -1 (*fin de fichier*)

```
int
recvAll(Socket dialogSocket,            // this function ensures that the total
        byte[] buffer) throws Throwable // amount of data has been received from
{                                       // the dialog socket
InputStream input=dialogSocket.getInputStream();
int nbBytesRcvd=0;
while(nbBytesRcvd<buffer.length)
  {
  int r=input.read(buffer,nbBytesRcvd,buffer.length-nbBytesRcvd);
  if(r!=-1) { nbBytesRcvd+=r; } else break; // end of file
  }
return nbBytesRcvd;
}
```

Consultez la documentation !

En *C* : les pages de manuel

- ex : `man 2 recv` , `man 3 gethostbyname` , `man 7 tcp` ...
- C'est la documentation de référence
  - Plus que le langage, le fonctionnement du système est décrit
  - Tous les autres langages/outils reposent sur ces fonctionnalités

En *Python* : la documentation de la bibliothèque standard

- `http://docs.python.org/library/socket.html`
- Respecte bien l'usage des appels systèmes
  - De nombreuses références aux pages de manuel sont faites

En *Java* : la documentation de l'API

- `http://java.sun.com/javase/6/docs/api/`
- Un très grand nombre de classes et de paquets à consulter !
  - Beaucoup d'encapsulation autour des appels systèmes