

Deuxième partie

Communication par flots
Interface socket

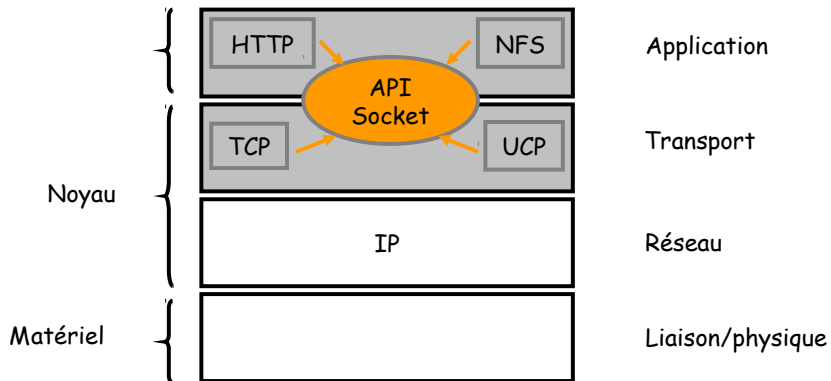


Plan

- 1 Présentation générale
 - Objectifs
 - Éléments de base
- 2 Structure client/serveur
- 3 Programmation (API C)
 - Exemples
 - API principale
 - Divers
- 4 Programmation (API Java)
 - Mode connecté
 - Mode non connecté



L'API socket dans la pile IP



Objectifs de l'API socket

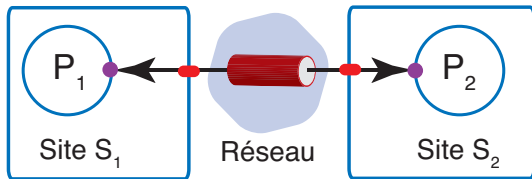
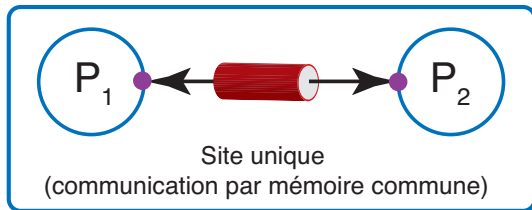
- Fournir une interface homogène aux applications
 - service TCP
 - service UDP
- Conforme au système de gestion de fichiers
 - flot d'octets
 - lecture/écriture
- Modèle client/serveur



La notion de flot centralisé → réparti

Idée

- Pipe « réparti »
- Désignation ?
- Protocole ?
- Interface ?
- Fiabilité ?
- Hétérogénéité ?

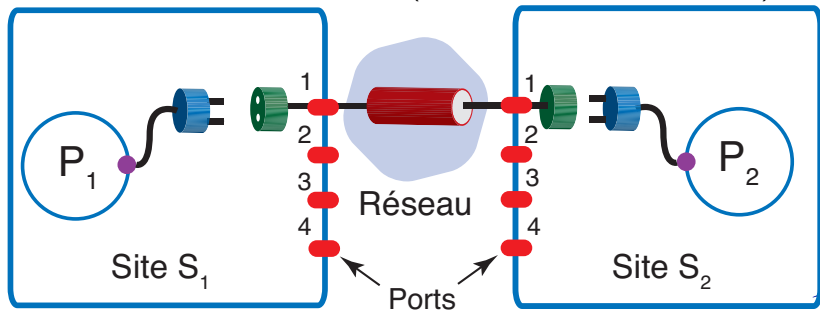


Désignation globale via la notion de port

Comment désigner un processus à distance ?

- Un processus a un nom local : numéro d'ordre par exemple ;
- Un site a un nom global : adresse IP par exemple ;
- Un port est un « point d'accès » à un site ;

Nom **global** d'un processus : (adresse site, numéro de port)

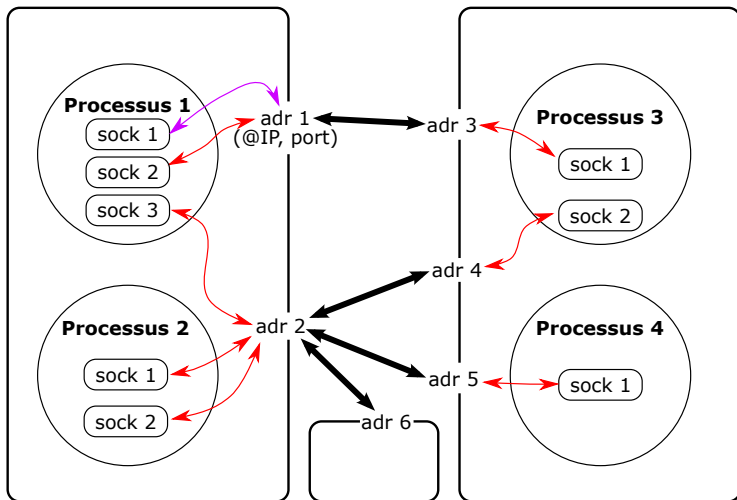


Éléments de base

- Socket → descripteur de fichier
- Protocole / domaine d'adresses → famille d'adresses
- Adresse → (adresse IP, numéro de port)
- Liaison → attribution d'une adresse à un socket
- Association → domaine + couple d'adresses (client/serveur ou émetteur/récepteur)



Schéma général



Numéro de ports standards (notorious)

/etc/services

ftp	21/tcp	
telnet	23/tcp	
smtp	25/tcp	mail
http	80/tcp	www
#		
# UNIX	specific services	
#		
exec	512/tcp	
login	513/tcp	
printer	515/tcp	spooler
who	513/udp	whod
talk	517/udp	



Services offerts

Orienté connexion (TCP)

- établissement/acceptation de connexion
- flot d'octets fiable et ordonné
- terminaison de connexion

Orienté datagramme (UDP)

- pas de connexion
- un message applicatif = une opération
- à la fois flot et messages



Plan

- 1 Présentation générale
 - Objectifs
 - Éléments de base
- 2 Structure client/serveur
- 3 Programmation (API C)
 - Exemples
 - API principale
 - Divers
- 4 Programmation (API Java)
 - Mode connecté
 - Mode non connecté



Structure générale

Client

- initie la communication
- doit connaître le serveur

Serveur

- informe le système de sa disponibilité
- répond aux différents clients
- clients pas connus a priori



Client/serveur non connecté

Client

```
créer un socket
répéter
    émettre une requête
        vers une adresse
    attendre la réponse
jusqu'à réponse positive
    ou abandon
```

Serveur

```
créer un socket
attribuer une adresse
répéter
    attendre une requête
    traiter la requête
    émettre la réponse
jusqu'à fin du service
```



Client/serveur connecté

Client

créer un socket
se connecter au serveur
dialoguer avec le serveur
par le socket connecté
terminer la connexion

Serveur

créer un socket
attribuer une adresse
informer le système
répéter
attendre une demande
de connexion
dialoguer avec le client
par le socket ainsi créé
jusqu'à fin du service

- un socket d'écoute pour accepter les connexions,
- un socket connecté pour chaque connexion



Plan

- 1 Présentation générale
 - Objectifs
 - Éléments de base
- 2 Structure client/serveur
- 3 **Programmation (API C)**
 - Exemples
 - API principale
 - Divers
- 4 Programmation (API Java)
 - Mode connecté
 - Mode non connecté

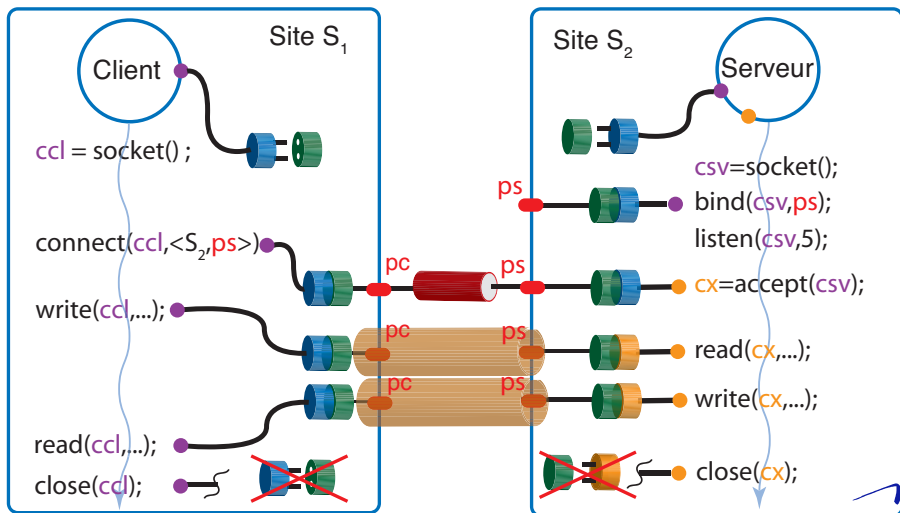


Primitives principales

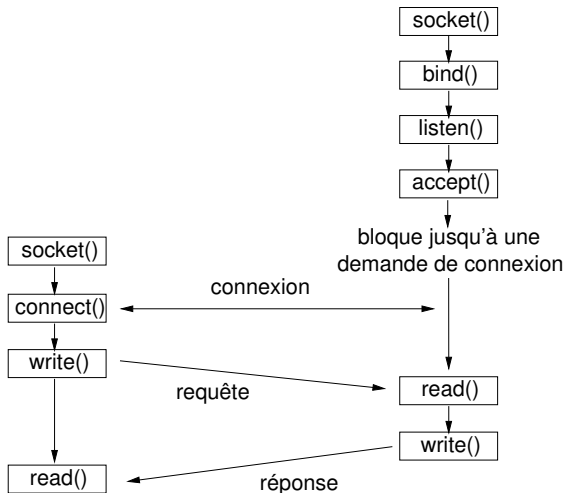
socket	création
bind	nommage : liaison d'un socket à une adresse
connect	connexion : établissement d'une association
listen	prêt à attendre des connexions
accept	attente de connexion : acceptation d'association
close	fermeture
shutdown	fermeture (obsolète)
read/write	
recv/send	
recvfrom/sendto	



Exemple – mode connecté



Exemple – mode connecté



Exemple – mode connecté

Le client

```
struct sockaddr_in adrserv; // adresse de socket dans le monde inet (IPv4)
char reponse[6];

int scl = socket (AF_INET, SOCK_STREAM, 0); // création d'un socket flux
dans le monde IP
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
inet_aton("147.127.133.111", &adrserv.sin_addr); // aton=AsciiToNetwork
adrserv.sin_port = htons (4522);
connect (scl, (struct sockaddr *)&adrserv, sizeof(adrserv)); //
contrôles d'erreurs indispensables ici, (-1 renvoyé en cas d'erreur)
write (scl, "hello", 6);
read (scl, reponse, 6);
close (scl);
```

ATTENTION : il manque le contrôle d'erreur, INDISPENSABLE.

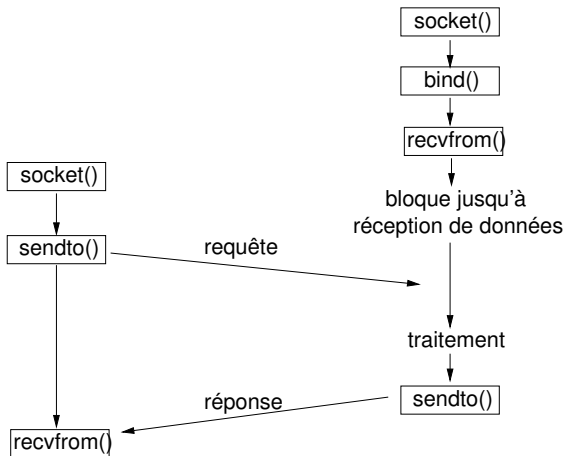
Exemple – mode connecté

Le serveur

```
struct sockaddr_in adrserv;  
int sserv = socket (AF_INET, SOCK_STREAM, 0);  
bzero (&adrserv, sizeof(adrserv));  
adrserv.sin_family = AF_INET;  
adrserv.sin_addr.s_addr = htonl (INADDR_ANY);  
adrserv.sin_port = htons (4522);  
bind (sserv, (struct sockaddr *)&adrserv, sizeof(adrserv)); // permet  
de lier un socket à une adresse (couple IP + port)  
listen (sserv,5); // socket d'écoute  
while (1) {  
    char requete[6];  
    int scl = accept (sserv, NULL, NULL); //accept est bloq, jusqu'à ce qu'il cli se conct  
    read(scl, requete, 6);  
    if (strcmp(requete, "hello") == 0) write(scl, "world", 6); else  
    write(scl, "bouh", 5);  
    close(scl);  
}
```




Exemple – mode non connecté



Exemple – mode non connecté

Le client

```
struct sockaddr_in adrserv;  
struct hostent *sp;  
char *requete = "hello";  
char reponse[10];  
  
int scli = socket (AF_INET, SOCK_DGRAM, 0); // socket packet, mode datagramme  
bzero (&adrserv, sizeof(adrserv));  
adrserv.sin_family = AF_INET;  
sp = gethostbyname("turing.enseeiht.fr"); // assume OK  
memcpy (&sins.sin_addr, sp->h_addr_list[0], sp->h_length);  
adrserv.sin_port = htons (4522);  
  
sendto (scli, requete, strlen(requete), 0,  
        (struct sockaddr *)&adrserv, sizeof(adrserv));  
recvfrom (scli, reponse, sizeof (reponse), 0, NULL, NULL);  
close (scli);
```



Exemple – mode non connecté

Le serveur

```
struct sockaddr_in adrserv, adrcli; char requete[10];

int sserv = socket (AF_INET, SOCK_DGRAM, 0);
bzero (&adrserv, sizeof(adrserv));
adrserv.sin_family = AF_INET;
adrserv.sin_addr.s_addr = htonl (INADDR_ANY);
adrserv.sin_port = htons (4522); // host to network
bind (sserv, (struct sockaddr*)&adrserv, sizeof(adrserv));
while (1) {
    bzero (&adrcli, sizeof (adrcli));
    int adrclilen = sizeof (adrcli);
    recvfrom (sserv, requete, sizeof(requete), 0,
              (struct sockaddr *)&adrcli, &adrclilen);
    sendto (sserv, "world", 6, 0,
            (struct sockaddr *)&adrcli, adrclilen);
}
```



Création d'un socket

socket crée un socket en spécifiant le famille de protocole utilisée.

```
int socket(int domain, int type, int protocol)
```

où

- domain = AF_INET, AF_INET6, AF_UNIX, AF_X25...
- type = SOCK_STREAM, SOCK_DGRAM
- protocol = 0

Retour : un « descripteur » de fichier ou -1



Adresse

```
struct sockaddr_in {  
    short sin_family;  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8];  
};
```

- `sin_family` doit être `AF_INET`
- `sin_port` numéro de port sur 16 bits, dans une représentation standard : "network byte ordered" (big endian)
- `sin_addr.s_addr` = adresse IP sur 32 bits, correctement ordonnée (big endian)



Représentation des entiers

La pile IP est « big endian »

Conversion de données (numéro de port, adresse IP)

htonl	<i>host-to-network, long int</i>
htons	<i>host-to-network, short int</i>
ntohl	<i>network-to-host, long int</i>
ntohl	<i>network-to-host, short int</i>

Conversion ascii ↔ in_addr

```
int inet_aton(const char *cp, struct in_addr *inp);  
char *inet_ntoa (struct in_addr in);
```



Service de nommage (DNS)

```
struct hostent {  
    char    *h_name;          /* nom canonique */  
    char    **h_aliases;     /* liste d'alias */  
    int     h_addrtype;      /* type des adresses */  
    int     h_length;        /* longueur d'une adresse */  
    char    **h_addr_list;   /* liste d'adresses */  
}; /*  
struct hostent *gethostbyname(char *name);  
struct hostent *gethostbyaddr(void *addr, int len, int type);
```

- `gethostbyname` avec un nom de machine
`turing.enseeiht.fr` permet d'obtenir ses autres noms et son (ses) adresse(s). Actuellement `h_addrtype == AF_INET` ou `AF_INET6`, et utiliser `h_addr_list[0]`.
- Pour une adresse format `sin_addr` de type `== AF_INET`, `gethostbyaddr` permet d'obtenir le(s) nom(s) en clair.

Liaison socket/adresse

bind nomme localement le socket (machine, port).

Obligatoire pour accepter des connexions ou recevoir des messages.

```
int bind(int sd, struct sockaddr *addr, int addrlen);
```

où

- sd : descripteur du socket
- addr : adresse attribuée à ce socket
- addrlen : taille de l'adresse (`sizeof(struct sockaddr_in)`)

Retour : 0 si ok, -1 si échec avec `errno` :

- EACCESS = permission refusée (adresse réservée)
- EADDRINUSE = adresse déjà utilisée pour un nommage
- ...



Connexion (côté client)

connect identifie l'extrémité distante d'une association.

```
int connect(int sd, struct sockaddr *addr, int addrlen);
```

où

- sd : descripteur du socket (du client)
- addr : adresse du socket du serveur
- addrlen : taille de l'adresse

Retour : 0 si ok, -1 si échec.

- EISCONN = socket déjà connecté
- ECONNREFUSED = connexion refusée (pas d'écouteur)
- ENETUNREACH = réseau inaccessible
- ETIMEDOUT = délai de garde expiré avant l'établissement de la connexion
- ...



Déclaration du serveur

`listen` établit une file d'attente pour les demandes de connexions.

```
int listen(int sd, int backlog);
```

où

- `sd` : descripteur du socket (du client)
- `backlog` : nombre max de clients en attente

Retour : 0 si ok, -1 si échec.

- `EADDRINUSE` = un autre socket déjà à l'écoute sur le même port.



Acceptation d'une connexion

`accept` prend la première demande de connexion et crée un nouveau socket ayant les mêmes caractéristiques que `sd` mais connecté à l'appelant \Rightarrow établissement d'une association.

```
int accept(int sd, struct sockaddr *peer, int *addrlen);
```

où

- `sd` : socket existant de type `STREAM`
- `peer` : adresse du socket du client (valeur en retour)
- `addrlen` = taille de l'adresse fournie et taille de l'adresse retournée (utiliser `sizeof(struct sockaddr_in)`)

Retour : un nouveau descripteur de socket si ok, ou `-1` en cas d'erreur

- `EOPNOTSUPP` = `sd` n'est pas de type `STREAM`



Communication de données

- Appels système classiques : read/write.

```
int write(int sd, const void *buf, int len);  
int read(int sd, void *buf, int len);
```

- Flot d'octets : les frontières entre les messages ne sont pas préservées
- \Rightarrow protocole applicatif nécessaire



Communication, mode non connecté

```
int sendto(int sd, void *buf, int len, int flags,  
           struct sockaddr *dest, int addrlen);  
int recvfrom(int sd, void *buf, int len, int flags,  
             struct sockaddr *src, int *addrlen);
```

où

- sd : socket
- buf, len : message à envoyer
- dest : adresse du socket destinataire (sendto, entrée)
- src : adresse du socket émetteur (recvfrom, sortie)
- addrlen : longueur de l'adresse (entrée et sortie pour recvfrom)

Retour : ≥ 0 si nombre d'octets émis/reçus, -1 si erreur



Liaisons implicites/explicites

bind implicite

- Lors d'un connect ou d'un sendto, le socket doit avoir une adresse locale \Rightarrow attribution d'un numéro de port non utilisé si nécessaire.
- Il est possible de nommer le socket (bind) avant connect ou sendto, mais guère d'utilité.

connect explicite

Il est possible de « connecter » un socket en mode datagramme (utilisation de connect sur un socket SOCK_DGRAM) :

- plus nécessaire de spécifier le destinataire de *chaque* message
- sans connect, le même socket peut être utilisé vers différents destinataires

Fermeture

- `close` termine l'association et libère le socket après avoir délivré les données en attente d'envoi.

```
int close(int sock);
```

Note : comportement inattendu s'il reste des octets à lire (écritures en attente perdues).

- `shutdown` permet une fermeture unilatérale :

```
int shutdown(int sock, int how);
```

où `how` = `SHUT_RD` (fin de réception), `SHUT_WR` (fin d'émission), `SHUT_RDWR` (fin de réception et d'émission)

Rq : il faudra quand même appeler `close` pour libérer les ressources du système.



Configuration (en tant que fichier)

```
int fcntl(int fd, int cmd, ...)
```

Par exemple : `fcntl(sd, F_SETFL, O_NONBLOCK)`
pour mettre en non bloquant.



Configuration (en tant que socket)

```
int setsockopt(int sockfd, int level, int optname,  
               const void *optval, socklen_t optlen);
```

Par exemple :

```
int ra = 1;  
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &ra, sizeof(ra));
```

level	option	description
SOL_SOCKET	SO_REUSEADDR	réutilisation immédiate d'adresse locale
	SO_KEEPALIVE	maintient en vie une connexion
	SO_BROADCAST	autorise les diffusions
IPPROTO_TCP	TCP_MAXSEG	taille max d'un segment TCP
	TCP_NODELAY	disable Nagle's algorithm
IPPROTO_IP	IP_OPTIONS	options des entêtes IP

Adresse d'un socket

- obtenir l'adresse de l'extrémité locale d'une association

```
int getsockname(int sd,  
                struct sockaddr *addr, int *addrlen);
```

- obtenir l'adresse de l'extrémité distante d'une association
(mode connecté)

```
int getpeername(int sd,  
                struct sockaddr *peer, int *addrlen);
```



Plan

- 1 Présentation générale
 - Objectifs
 - Éléments de base
- 2 Structure client/serveur
- 3 Programmation (API C)
 - Exemples
 - API principale
 - Divers
- 4 Programmation (API Java)
 - Mode connecté
 - Mode non connecté



Les classes

- `java.net.InetAddress` pour manipuler des adresses IP
- Mode connecté : `java.net.Socket` et `java.net.SocketServer`
- Mode datagramme : `java.net.DatagramSocket` et `java.net.DatagramPacket`

Note : les interfaces présentées sont incomplètes (exceptions supprimées).



La classe `java.net.InetAddress`

- Deux sous-classes `Inet4Address`, `Inet6Address`
- Obtention :
 - `static InetAddress getLocalHost\(\)`;
renvoie l'adresse IP du site local d'appel.
 - `static InetAddress getByName(String host)`;
Résolution de nom (sous forme symbolique
`turing.enseeiht.fr` ou numérique `147.127.18.03`)
 - `static InetAddress[] getAllByName(String host)`;
Résolution de nom → toutes les adresses IP d'un site
- Accesseurs
 - `String getHostName()`
le nom complet correspondant à l'adresse IP
 - `String getHostAddress()`
l'adresse IP sous forme `d.d.d.d` ou `x:x:x:x:x:x:x`
 - `byte[] getAddress()`
l'adresse IP sous forme d'un tableau d'octets.



Classe `java.net.ServerSocket`

Représente un socket d'écoute.

- Constructeurs :

- `ServerSocket(int port);`

- `ServerSocket(int port, int backlog, InetAddress bindAddr);`

Réalise socket – bind – listen.

- Méthodes :

- Socket `accept();`

- Renvoie un socket connecté. Bloquant. (= accept en C).

- InetAddress `getInetAddress();`

- Renvoie l'adresse IP locale.

- int `getLocalPort();`



La classe `java.net.Socket`

Représente un socket connecté, côté client comme côté serveur.

- Constructeurs (côté client) :

```
Socket(String host, int port);  
Socket(InetAddress address, int port);  
Socket(String host, int port,  
        InetAddress localAddr, int localPort);  
Socket(InetAddress addr, int port,  
        InetAddress localAddr, int localPort);
```

= socket – bind (éventuellement) – connect.

Constructeur bloquant !

- Accès aux flots de données :

```
InputStream getInputStream();  
OutputStream getOutputStream();
```



La classe `java.net.Socket` (suite)

- Accesseurs :
 - `InetAddress getLocalAddress()`;
Renvoie l'adresse IP locale. (\approx `getsockname` en C)
 - `int getLocalPort()`;
Renvoie le port local.
 - `InetAddress getInetAddress()`;
Renvoie l'adresse IP distante. (\approx `getpeername` en C)
 - `int getPort()`;
Renvoie le port distant.



Socket en Java : exemple client

```
public class Client {  
    public static void main(String[] args) throws Exception {  
        Socket socket = new Socket("bach.enseeiht.fr", 8080);  
        // Un BufferedReader permet de lire par ligne.  
        BufferedReader plec = new BufferedReader(  
            new InputStreamReader(socket.getInputStream()));  
        // Un PrintWriter possède toutes les opérations print classiques.  
        // En mode auto-flush, le tampon est vidé (flush) lors de println.  
        PrintWriter pred = new PrintWriter(  
            new BufferedWriter(  
                new OutputStreamWriter(socket.getOutputStream()),  
                true);  
        String str = "bonjour";  
        for (int i = 0; i < 10; i++) {  
            pred.println(str + i);           // envoi d'un message  
            str = plec.readLine();           // lecture de l'écho  
        }  
        pred.println("END") ;  
        plec.close();  
        pred.close();  
        socket.close();  
    }  
}
```

Attention aux
exceptions !

Socket en Java : exemple serveur

```
public class Serveur {  
    public static void main(String[] args) throws Exception {  
        ServerSocket s = new ServerSocket(8080);  
        while (true) {  
            Socket soc = s.accept();  
            BufferedReader plec = new BufferedReader(  
                new InputStreamReader(soc.getInputStream()));  
            PrintWriter pred = new PrintWriter(  
                new BufferedWriter(  
                    new OutputStreamWriter(soc.getOutputStream()),  
                    true);  
            while (true) {  
                String str = plec.readLine();           // lecture du message  
                if (str.equals("END")) break;  
                pred.println(str);                       // renvoi d'un écho  
            }  
            plec.close();  
            pred.close();  
            soc.close();  
        }  
    }  
}
```

Attention aux
exceptions !

Socket en mode datagramme `java.net.DatagramSocket`

- Constructeurs :

```
DatagramSocket(); // port quelconque disponible  
DatagramSocket(int port);
```

- Méthodes :

```
void send(DatagramPacket p);  
void receive(DatagramPacket p);
```

- Classe `java.net.DatagramPacket` :

```
DatagramPacket(byte[] buf, int length);  
DatagramPacket(byte[] buf, int length,  
                InetAddress addr, int port); // destination
```

+ getters et setters



Conclusion

Principes de base

- Extension d'une notion issue du monde « centralisé »
- Connexion point à point entre processus
- Bases du modèle (protocole) client-serveur
- Communication en mode datagramme ou connecté
- Pas de transparence de la communication

Pour aller plus loin

- Trouver une abstraction du contrôle plus simple
⇒ réutiliser la notion de procédure
- Principe de conception : idée de **transparence**

