Cours de Réseau et communication Unix n°5

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/rezo/

Lien court : http://j.mp/rezocom

Plan du cours n°5

1. La communication par datagrammes
2. La communication en mode connecté

1 - La communication par datagrammes

Sockets de type `SOCK_DGRAM`, protocole UDP (*User Datagram Protocol*).

Datagrammes

Deux processus qui veulent dialoguer avec des sockets de type `SOCK_DGRAM` s'échangent des datagrammes : des messages de taille limitée.

La taille limite d'un message est le MTU : Maximum Transfert Unit

Si un message est trop grand, son envoie échoue
(ou selon système : le message peut être tronqué).

Pour les socket internet, le MTU est en général 1500 octets.

Pour les socket Unix, MTU beaucoup plus grand :

163 808	linux 3.2.0 i386
124 896	linux 2.6.32 x86_64
2 048	macos 10.6.8 i386

Sockets non connectés

Chaque demande d'envoi d'un datagramme s'accompagne de l'adresse du destinataire.

Chaque réception d'un datagramme s'accompagne de l'adresse de l'expéditeur.

Avantage : permet un nombre de correspondants illimité

Inconvénient : pour chaque message reçu il faut retrouver le contexte à partir de l'adresse de l'expéditeur.

On peut simplifier envoi et réception en réalisant une pseudo-connexion.

Protocole UDP

L'émetteur ne sait pas si le message est arrivé à destination.

Dans le domaine `AF_UNIX`,

- il sait si l'adresse de destination est valide et si une socket lui est associée ;
- le message n'est pas dupliqué.

Si plusieurs messages sont envoyés à la suite, l'émetteur ne sait pas si ils seront délivrés dans le bon ordre.

Si le récepteur reçoit un datagramme, son intégrité est garantie.

Envoi de datagrammes

Le processus qui envoie le datagramme (l'émetteur) doit :

- créer une socket locale ;
- l'attacher à une adresse locale ;
- construire l'adresse du destinataire ;
- envoyer le message en spécifiant les 2 adresses.

Ceci exclut a priori `write`.

sendto

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto (
   int sockfd,                              // socket
   const void *buf,                         // message
   size_t len,                              // longueur message
   int flags,                               // 0
   const struct sockaddr *dest_addr,        // caster
   socklen_t addrlen );                     // du struct
```

Renvoie : > 0 le nombre de caractères envoyés
-1 erreur

`sendto` est non bloquant en UDP.

Erreurs de `sendto`

Si `len` $>$ MTU, `sendto` échoue avec `errno = EMSGSIZE`

Si la queue d'envoi de l'interface réseau est pleine,

- sous Linux, le paquet est silencieusement perdu (sans erreur) ;
- sous MacOs, `sendto` renvoie -1 avec `errno = ENOBUFS`.

Réception de datagrammes

Le processus qui veut recevoir un datagramme (le récepteur) doit :

- créer une socket locale ;
- l'attacher à une adresse locale ;
- se mettre en attente bloquante d'un message.

Il recevra en même temps le message et l'adresse de l'émetteur.

Ceci exclut a priori `read`.

recvfrom

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom (
   int sockfd,                          // socket
   void *buf,                           // buffer
   size_t len,                          // taille buffer
   int flags,                           // 0
   struct sockaddr *src_addr,           // caster
   socklen_t *addrlen );                // initialiser
```

Renvoie : > 0 le nombre de caractères reçus
-1 erreur

Avant l'appel, il faut initialiser `*addrlen` à la taille du struct de l'adresse ; après l'appel, il contient la taille véritable.

`recvfrom` est bloquant en UDP.

Pseudo-connection

Il est possible de mémoriser dans une socket une adresse distante permanente :

- on peut alors utiliser `read` et `write` ;
- on peut continuer à utiliser `sendto` vers d'autres adresses ;
- mais on ne reçoit plus que de cette adresse (filtre).

Faire la pseudo-connexion :

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, const struct sockaddr *addr,
             socklen_t addrlen);
```

Défaire : nouvel appel avec `addr = NULL`
ou avec `addr->sun_family = AF_UNSPEC`

Récupérer l'adresse distante : `getpeername`

2 - La communication en mode connecté

Sockets de type `SOCK_STREAM`, protocole TCP (*Transmission Control Protocol*).

socket connectées

Permet à deux processus d'échanger un flux de caractères (comme dans un tube).

Les étapes :

1. établissement de la connexion ;
2. dialogue avec `read` et `write` ;
3. déconnexion.

Étapes

1. Établissement :
   - un processus se met en attente (le serveur) ;
   - un autre processus demande la connexion (le client) ;
   - le serveur accepte ou refuse.
2. Dialogue :
   - chacun peut lire ou écrire ;
   - le protocole TCP est fiable.
3. Déconnexion :
   - l'un des processus se déconnecte (ou se termine) ;
   - l'autre processus détecte la fin.

Établissement de la connexion : côté serveur

1. Création socket : `socket`
2. Attachement à une adresse connue : `bind`
3. Ouverture du service : `listen`
   La socket devient une "socket d'écoute"
4. Attente bloquante d'une demande de connexion,
   puis obtention d'une "socket de service" : `accept`

Puis : dialogue sur la socket de service : `read` et `write`

Établissement de la connexion : côté client

1. Création socket client : `socket`
2. Attachement à une adresse quelconque : `bind`
3. Construction adresse serveur
4. Demande bloquante de connexion au serveur : `connect`

Puis dialogue sur la socket avec le serveur : `read` et `write`

Les primitives : `listen`

```
#include <sys/types.h>
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

Permet à un processus de déclarer un service ouvert auprès du S.E.

La socket ouverte `sockfd` devient une socket d'écoute.

`backlog` est la taille maximale de la file d'attente des connexions $<$ `SOMAXCONN` (128 pour linux)

Renvoie 0 succès, -1 erreur.

Les primitives : accept

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *addr,
            socklen_t *addrlen);
```

Appel bloquant ; revient lorsqu'il y a au moins une connexion pendante.

`addr` reçoit l'adresse du client en attente de connexion ; ce client est retiré de la liste des connexions pendantes.

`*addrlen` doit être initialisé à la taille du struct passé en paramètres ; puis il contient la longueur réelle.

Renvoie : $\geqslant 0$ socket de service connectée au client
-1 erreur

Les primitives : connect

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, const struct sockaddr *addr,
             socklen_t addrlen);
```

Appel bloquant, revient lorsque

- le serveur a fait accept,
- ou au bout d'un certain délai $\rightarrow$ ETIMEDOUT

Renvoie 0 succès, -1 erreur

connect ne doit être appelée qu'une fois pour une socket de type SOCK_STREAM.

Déconnexion

Comme pour les tubes :

- l'un des processus fait `close` :
  - le client sur sa socket
  - le serveur sur la socket de service
- l'autre processus détectera la fermeture
  - lors d'un `read` $\rightarrow$ retour 0
  - lors d'un `write` $\rightarrow$ `SIGPIPE` et `EPIPE`

Refus de connexion

Le serveur peut refuser une connexion :

1. il doit faire `accept` pour connaitre l'adresse du client ;
2. il décide un refus (mauvais client, ...) ;
3. il ferme la socket avec `close`.

Côté client : `connect` réussit, puis

- `read` $\rightarrow 0$
- `write` $\rightarrow$ `SIGPIPE` et `EPIPE`

Scrutation

Côté serveur, la socket d'écoute et les sockets de service peuvent être scrutées :

- si la socket d'écoute est éligible, alors on peut faire un `accept` immédiat ;
- si une socket de service est éligible, alors on peut faire un `read` ou `write` immédiat.

Serveurs UDP ou TCP

Pour dialoguer avec n clients :

- En UDP : simple boucle `recvfrom` / `sendto`
- En TCP :
  - scruter socket d'écoute + n sockets de services
  - père `accept` socket d'écoute puis délègue la socket de service à un nouveau fils $\rightarrow n$ fils
  - solution mixte : père scrute socket d'écoute et jusqu'à k sockets de services ; si la liste est pleine, il délègue ces k sockets à un nouveau fils $\rightarrow \lfloor n/k \rfloor$ fils.

Surveiller l'état des sockets avec `netstat`

`netstat -a` affiche toutes les sockets et sockets d'écoute

`netstat -t` affiche les sockets TCP/IP

```
<> netstat -at
Connexions Internet actives (serveurs et établies)
Proto Recv-Q Send-Q Adresse locale    Adresse distante         Etat
tcp    0    0 *:ssh                  *:*                      LISTEN
tcp    0    0 localhost:ipp          *:*                      LISTEN
tcp    0    0 localhost:smtp         *:*                      LISTEN
tcp    0    0 capri.lidil.univ-:35638 we-in-f16.1e100.n:imaps CLOSE_WAIT
tcp    0    0 capri.lidil.univ-:33619 wi-in-f16.1e100.n:imaps CLOSE_WAIT
tcp    0    0 capri.lidil.univ-:44491 61.128.17.93.rev.:imap2 CLOSE_WAIT
tcp    0    0 localhost:ssh          localhost:36570          ESTABLISHED
tcp    0    0 localhost:53623        localhost:ssh            ESTABLISHED
tcp    0    0 capri.lidil.univ-:52134 bureau-imapp2.uni:imaps ESTABLISHED
tcp    0    0 capri.lidil.univ-:58713 129.128.17.93.rev:imap2 CLOSE_WAIT
tcp    0    0 localhost:36570        localhost:ssh            ESTABLISHED
tcp    0    0 capri.lidil.univ-:52028 bureau-imapp2.uni:imaps ESTABLISHED
tcp    0    0 localhost:ssh          localhost:53623          ESTABLISHED
tcp6   0    0 [::]:www               [::]:*                   LISTEN
tcp6   0    0 [::]:ssh               [::]:*                   LISTEN
tcp6   0    0 localhost:ipp          [::]:*                   LISTEN
```

Sous Windows

Mêmes fonctions avec les winsocks ; les erreurs sont différentes.

```
int listen (
  _In_  SOCKET s,
  _In_  int backlog
);

SOCKET accept (
  _In_     SOCKET s,
  _Out_    struct sockaddr *addr,
  _Inout_  int *addrlen
);

int connect (
  _In_  SOCKET s,
  _In_  const struct sockaddr *name,
  _In_  int namelen
);
```

Petites différences sous Windows

```
int closesocket (    // au lieu de close
  _In_  SOCKET s
);

int send (           // idem Unix ; au lieu de write
  _In_  SOCKET s,
  _In_  const char *buf,
  _In_  int len,
  _In_  int flags
);

int recv (           // idem Unix ; au lieu de read
  _In_   SOCKET s,
  _Out_  char *buf,
  _In_   int len,
  _In_   int flags
);
```