HAI404I : Licence 2 Informatique

Réseaux : IP, Protocoles et Communications

Anne-Elisabeth Baert – baert@lirmm.fr

2021-2022

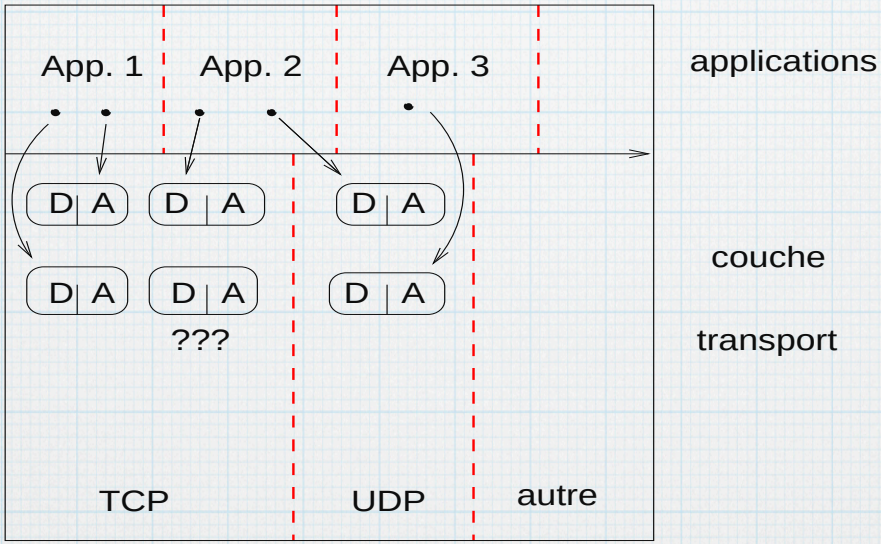
Chapitre – Mise en Œuvre d'Applications

Principes

Vision des Applications

- communication par des boites réseaux (boîtes où l'acheminement est pris en charge par des réseaux, comme des boîtes postales où le transport est effectué par des postes) ;
- directement au dessus du transport : l'interface de programmation offrira un accès au niveau transport ;
- une application peut obtenir plusieurs boîtes ;
- chaque boîte avec deux cases : départ, arrivée ; on dépose ce qu'on veut expédier et on lit ce qui est arrivé ;
- chaque BR a une adresse composée du triplet :
  - adresse hôte (numéro IP)
  - numéro de boîte
  - protocole

Allocation des BR



Attention : chaque protocole de transport fait sa propre numérotation des boîtes. Donc un numéro de boîte n'indique pas de quel protocole il s'agit.

??? : pas de BR sans application associée.

Quel Modèle ?

Questions lancinantes : Ce modèle de programmation s'appuie sur combien de couches ? Par rapport au modèle à 7 couches de l'OSI, peut-on établir une correspondance ?

On ne pourra répondre qu'en connaissant les services offerts et en détectant les services non offerts.

Protocoles sous-jacents

Internet – protocoles transport

Services offerts par les protocoles de transport sous-jacents :

tcp	udp
fiable	non
ordre garanti	non garanti
duplication impossible	possible
mode connecté	sans connexion
orienté flot	orienté message

Signification

Fiable : retourne un résultat à l'application, éventuellement négatif !

Ordre garanti : s'il y a désordre dans l'arrivée des paquets, le protocole prend en charge la remise en ordre et l'application ne s'en aperçoit pas.

Duplication impossible : s'il y a eu une double réception, le protocole la traite et l'application ne s'en aperçoit pas.

Signification - ça se complique

Mode connecté : la boîte réseau est utilisée pour communiquer de façon exclusive avec une seule autre boîte réseau ; on parle alors de circuit virtuel établi entre les deux applications ; analogie : le téléphone (mode connecté) et le courrier postal (mode sans connexion).

Orienté flot : le contenu expédié est vu comme un flot ; il peut être reçu en plusieurs morceaux ; de même, plusieurs expéditions peuvent être délivrées en une seule réception. m lectures $\leftrightarrow$ n écritures, $m \neq n$.

Orienté message : un message est expédié comme un bloc et reçu entièrement (ou non reçu si le protocole n'est pas fiable) ; vu de l'application, il n'est pas découpé. 1 lecture $\leftrightarrow$ 1 écriture.

Client – Serveur

Client : application qui envoie des requêtes à l'application dite serveur, attend une réponse indiquant leurs réalisations et les résultats éventuels.

Serveur : application qui attend des requêtes provenant d'applications clientes, réalise ces requêtes et rend les résultats.

requête : suite d'instructions, commandes, ou simple chaîne de caractères, obéissant à un langage, un accord ou une structure préalables connus des deux acolytes (protocole d'appplication).

Fonctionnement des processus

- Un processus "serveur" assure un service : il tourne en permanence en attendant des requêtes de clients. Il dispose d'une BR publique.
- Les clients arrivent à exprimer des requêtes parce qu'ils connaissent l'existence du service et l'adresse de cette BR publique du serveur.
- Les clients ne savent pas si le serveur est actif (ils l'espèrent actif).
- Les requêtes arrivent dans une file d'attente.
- En général le serveur doit minimiser le temps d'attente dans la file $\longrightarrow$traitement rapide ou délégation.
- **Attention à cette terminologie dans les systèmes de fenêtrage**

Interface de programmation

Interface Socket

Un ensemble de primitives permettant de réaliser des applications communiquant sur un réseau, fournissant au programmeur l'accès (l'interface) à la couche transport.

Principes :

- mode **sans connexion** : on peut communiquer dans les deux sens avec plusieurs BR.
  - se faire allouer une BR locale
  - identifier le distant
  - envoyer des messages / consulter les messages entrants
  - savoir rendre (fermer) la BR si plus nécessaire
- mode **connecté** : principes identiques, mais la communication est dédiée exclusivement à deux BR déterminées (voir circuit virtuel.

Programmation - Sans Connexion

Sans Connexion - Deux Modèles

On peut proposer deux modèles de programmation :

- Un modèle symétrique : chaque application désigne son acolyte, c'est-à-dire la BR qu'elle veut joindre.
- Un modèle asymétrique : une des application, A, désigne l'acolyte B. B prend connaissance de l'adresse de la BR expéditrice lors de la réception du message de A.

Sans Connexion, Symétrique

Appli a	**Appli b**	
demander BR locale BR_a	demander BR locale BR_b	ma BR
désigner distant(BR_b)	désigner distant(BR_a)	BR dest.
sendto()		synchronisation
recvfrom()		par le transport
shutdown()		
close()		

Exemple sans connexion - les BR

```
/* demande de BR locale */
Sock breLoc(SOCK_DGRAM, (short)31470,0) ;
int descbreLoc ;
/* on recupere le descripteur */
if (breLoc.good()) descbreLoc=BreLoc.getsDesc() ;
else {cout<<"pb BR locale"<<endl ;
        exit(1) ;}
/* désignation BR distante */
SockDist saBr("hote.teslunettes.fr",(short)31469) ;
sockaddr_in *adrsaBr= saBr.getAdrDist() ;
```

sockaddr_in est une structure contenant le triplet désignant une adresse de BR dans le monde Internet.

Sans Connexion - le Dialogue

```
/* on expédie */
int retourSend=sendto(descbreLoc, msg, sizeof(msg),0,(sockaddr *)adrsaBr,
lgsaBr) ;
/* et on reçoit*/
int retourRecv=recvfrom(descbreLoc, tamponReception,lgReception, 0,NULL,
NULL) ;
```

Remarque importante : il n'est pas nécessaire d'être en réception afin de recevoir le message. `recvfrom()` est bloquant : s'il n'y a pas de message, l'exécution est bloquée, s'il y en a, la réception est effectuée conformément aux paramètres indiqués.

Sans Connexion - Asymétrique

On attend une première réception, pour détecter qui est l'expéditeur.

Après cette réception, on peut récupérer l'adresse de la BR expéditrice.

On peut alors réfléchir aux fonctions d'un serveur répondant à des clients différents.

```
SockDist expInconnu ;
socklen_t lgInconnu=expInconnu.getsLen() ;
sockaddr_in *adrexpInconnu=expInconnu.getAdrDist() ;
int retourRecv=recvfrom(descbreLoc, tamponReception,lgReception, 0,(sockaddr
*)adrexpInconnu,&lgInconnu) ;
```

Sans les Classes Fournies

On peut ne pas trouver les classes fournies à son goût. Se plonger alors dans les détails :

- Pour l'allocation des BR, voir les appels système
  - `socket()`
  - `bind()`
  - `gethostbyname()` et associés
  - `getservbyname()` et associés.
- Pour le dialogue, les appels sont ceux utilisés ci-avant ;
- Pour le reste, consulter les classes proposées ; attention à la syntaxe, peu encourageante.

De la Réservation des Ports

Question : Est-il logique de réserver les numéros de BR (numéros de ports) comme dans les exemples précédents ?

Réponses :

oui c'est un moyen rapide permettant de construire des exemples,

non pour plusieurs raisons :

- les numéros peuvent être utilisés par d'autres applications,
- pire, ils peuvent être utilisés par des applications courantes, dites bien connues,
- comment connaître les numéros alloués à distance ?
- sans parler des plantages entraînant des délais d'attente lors de la mise au point des programmes (voir les erreurs liées à `bind()`.

Réservation des Ports dans l'Internet

Principes :

- Un client peut demander l'attribution d'une BR sans se soucier du numéro ; une allocation par le système d'un numéro quelconque, libre est une bonne solution ;
- Seuls les serveurs ont nécessairement besoin d'être indentifiés ; ils identifient les clients lors de la première réception ;
- Dans le monde Internet, chaque application connue (donc le serveur correspondant) va se voir attribuer un numéro **publique** connu de tous les hôtes.

Exemples de Réservation

- tous les serveurs `sshd` vont utiliser strictement le port numéro 22 ;
- tous les serveurs `httpd` vont utiliser par défaut le port numéro 80.

Ainsi, tous les clients pourront localiser les serveurs, dès lors que ces numéros réservés sont enregistrés dans un fichier local. Noter que le contenu est universel, du moins pour les applications publiquement connues (voir sous Unix `/etc/services`).

Les numéros jusqu'à 1024 sont officiellement réservés pour ce type d'applications et ne peuvent être demandés par une application d'utilisateur non administrateur (`root`).

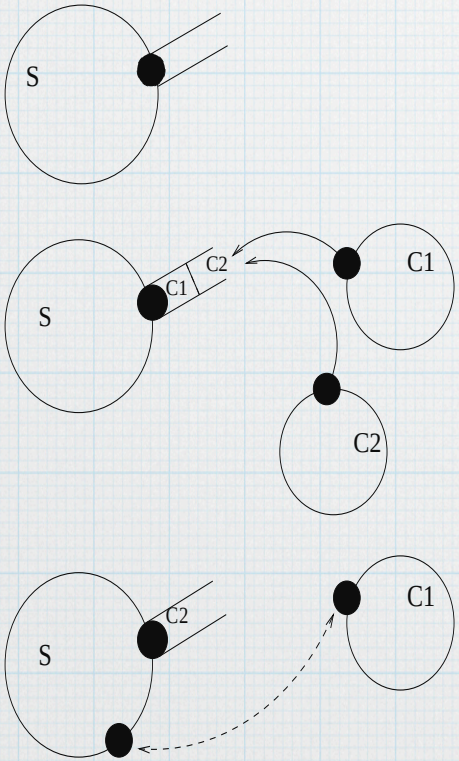
Programmation - Mode Connecté

Mode Connecté - Principes de Fonctionnement

- **avec connexion** :

Serveur	Client
se faire allouer une BR publique	demander une BR privée
	identifier le serveur
attendre des requêtes	
	demander une connexion
répondre (accepter de)	attendre réponse
dialoguer	
savoir arrêter ou savoir accepter l'arrêt	savoir accepter l'arrêt ou savoir arrêter

Connecté – Schéma de principe



Le serveur crée une file d'attente pour recevoir les demandes de connexions sur une BR publique

les clients font une demande de connexion à partir d'une BR privée vers la BR publique du serveur

le serveur accepte une connexion ;
il y a création d'un circuit virtuel avec deux BR privées dédiées à cette communication

Protocole TCP un retour

- protocole orienté connexion avec création d'un circuit virtuel
- bidirectionnel
- fiable : arrivée garantie, pas de duplication
- ordre des paquets garanti
- message vu comme un flot de caractères : pour une écriture on peut avoir besoin de plusieurs lectures et réciproquement.

Conséquences :

- `Tcp` prend en charge la mise en place du circuit virtuel, donc en fait tout la négociation et les contrôles (accusés réception, flux, ...) afin d'assurer le bon fonctionnement du circuit ;
- les processus communiquant doivent se mettre d'accord sur les limites des messages.

Mode Connecté - Mise en Œuvre

Serveur	Client	
créer BR publique	...	adresse connue publiquement
	créer BR privée	adresse indéterminée
listen()	...	longueur file d'attente
	connect()	demande de connexion
accept()		acceptation d'une demande
write() ou send()		dialogue
read() ou recv()		dialogue
shutdown()		fin partielle
close()		fin

Exemple - - Côté Serveur

Préparation de la BR publique

- On suppose que la BR demandée par le serveur est publiquement connue. Voir la discussion sur l'allocation des ports.

```
Sock brPub(SOCK_STREAM, (short)(21345), 0) ;
  int descBrPub ;
  if (brPub.good()) descBrPub=brPub.getsDesc() ;
  int estin=listen(descBrPub,5) ;//longueur file       //se mettre en attente
  struct sockaddr_in brCv ;
  socklen_t lgbrCv = sizeof (struct sockaddr_in) ;
  int descBrCv = accept (descBrPub,(struct sockaddr *)&brCv, &lgbrCv) ;
```

Attention : `desBrCv` est un nouveau descripteur, sur la BR privée allouée pour le circuit virtuel.

Exemple - - Côté Client

Préparation de la BR privée et demande de connexion

- Le client peut se faire attribuer une BR quelconque ; il suffit qu'elle ait un type adapté à une telle communication.

```
Sock brCli(SOCK_STREAM, 0) ;
  int descBrCli ;
  if (brCli.good()) descBrCli=brCli.getsDesc() ;
      //désigner le serveur
  SockDist brPub(argv[1], short(21345)) ;
  struct sockaddr_in * adrBrPub=brPub.getAdrDist() ;
  int lgAdrBrPub=sizeof(struct sockaddr_in) ;
      //demander une connexion
  int erlude = connect(descBrCli,(struct sockaddr *)adrBrPub,lgAdrBrPub) ;
```

Attention : le retour de `connect` indique si la requête a été déposée dans la BR publique du serveur.

Syntaxe - - Dialogue

Côté Serveur

```
char lemagne[256] ;
char lot[]="doremifa solasido" ;
int ensif = recv (descBrCv,lemagne,sizeof(lemagne), 0) ;
...
int ox = send (descBrCv,lot,strlen(lot),0) ;
```

Côté Client : Comme pour le serveur, en utilisant la boîte réseau privée locale.

Noter qu'il n'y a plus besoin de spécifier le destinataire : `tcp` a bien fait son travail.

Noter la possibilité d'utiliser `read()` et `write()` à la place de `recv()` et `send()`. Néanmoins, on perd le dernier argument, qui permet de spécifier plus finement les entrées-sorties.

À Titre Documentaire

À Titre Documentaire - Détails Primitives

**Obtenir un descripteur pour un objet <
>**

`int socket (int famille, int type, int protocole )`

famille : `PF_UNIX, PF_INET, PF_ISO, PF_INET6`

type : `SOCK_DGRAM, SOCK_STREAM, SOCK_RAW`

protocole : 0 par défaut le plus souvent ; voir manuel `protocols` et fichier `/etc/protocols`

retour : descripteur ou -1 (et errno positionné)

À Titre Documentaire - Détails Primitives

Associer un descripteur et une BR déterminée

```
int bind (int descripteur,
      const struct sockaddr *brDem, socklen_t lgDem)
```

descripteur provient de socket() ;

brDem doit être initialisée au triplet de la BR dont on demande l'allocation ;

lgDem longueur du triplet désigné ; par exemple `sizeof(struct sockaddr_in)`

retour 0 (OK) ; -1 (+errno positionnée).

Fermeture :

Comme pour les fichiers `int close (int descripteur)`
ou `int shutdown (int descripteur, int comment )`

comment `SHUT_RD` arrêt réceptions
`SHUT_WR` arrêt émissions
`SHUT_RDWR` les deux

À Titre Documentaire - Détails Primitives

Dialogue sans connexion :

```
int sendto( int descripteur,
        const void *msg, size_t lg, int flags,
        const struct sockaddr *brDest, socklen_t
lgDest)
```

msg	message à expédier ;
lg	longueur du message ;
flags	options ;
brDest	adresse BR destinatrice ;
lgDest	longueur de l'adresse BR dest.

À Titre Documentaire - Détails Primitives

Dialogue sans connexion - encore :

```
int recvfrom (int descripteur,
        const void *tamponrec,size_t lg, int flags,
        const struct sockaddr *brExp, socklen_t
*lgExp)
```

tamponrec	tampon pour le message reçu ;
lg	longueur de ce tampon (max. à recevoir) ;
flags	options ;
brExp	adresse de la BR expéditrice ;
lgExp	longueur de l'adresse BR exp.

À Titre Documentaire - Détails Primitives

Connecté - création du CV :

```
int accept (int descripteur,
            struct sockaddr *brCv, socklen_t *lgbrCv)
```

descripteur : celui de la BR publique ;

brCv : nouvelle BR privée créée ; le CV côté serveur ;

lgbrCv : longueur ; attention : paramètre en entrée et résultat ; à réinitialiser avant chaque accept().

retour : descripteur sur la BR privée ; -1 (erreur).

file d'attente :

```
int listen (int descripteur, int lgmax)
```

retour : 0(OK) ; -1 (erreur).

À Titre Documentaire - Détails Primitives

Connecté - demande de connexion :

```
int connect( int descripteur,
        struct sockaddr *brSrv, socklen_t lgbrSrv)
```

descripteur obtenu par `socket()` ; celui de la BR privée locale ;

brSrv BR publique du serveur ;

lgbrSrv longueur de cette BR serveur ;

retour 0 (OK) ; -1 (erreur).

À Titre Documentaire - Détails Primitives

Dialogue mode connecté :

```
int send (int descripteur, const void *tampon,
          size_t lg, int flags)
```

descripteur côté serveur c'est celui rendu par accept() ;

retour nombre d'octets envoyés ; -1 (erreur).

```
int recv (int descripteur,void *tampon,
          size_t lg, int flags)
```

lg max à recevoir.

retour nombre d'octets reçus ; -1 (erreur).

Du Blocage des Entrées-Sorties

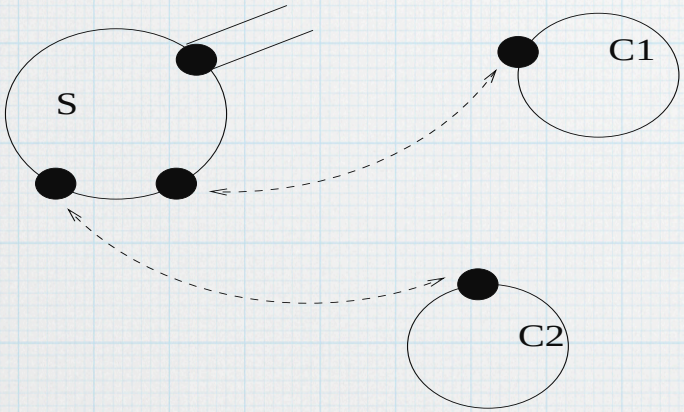
Constat : les entrées-sorties décrites jusque là sont majoritairement bloquantes :

- Les réceptions sont bloquantes de façon visible : si une BR est vide, il y aura attente jusqu'à l'arrivée d'un message ;
- les expéditions le sont aussi, bien que ce soit moins visible ; penser que le tampon d'expédition peut se vider à un rythme lent par rapport au remplissage ;
- les acceptations de connexions le sont de façon évidente ;
- les demandes de connexions le sont aussi, bien que de façon moins visible.

Problème : le schéma établi jusque là en mode connecté n'est valide que lorsqu'il y a un seul client, ou lorsque le traitement d'un client est court, de sorte à ne pas faire patienter la longue file d'attente possible.

Situation Difficile

La situation suivante est pratiquement invivable.



Coté serveur, si on attend une réception sur une des BR et si on n'a pas de réponse, les autres clients patientent lamentablement, quel que soit le point d'attente

Solutions :

- faire des entrées-sorties non bloquantes ; dans la plupart des cas, elles seront d'une inefficacité admirable ;
- déléguer chaque circuit virtuel à un clône ;
- autre ?

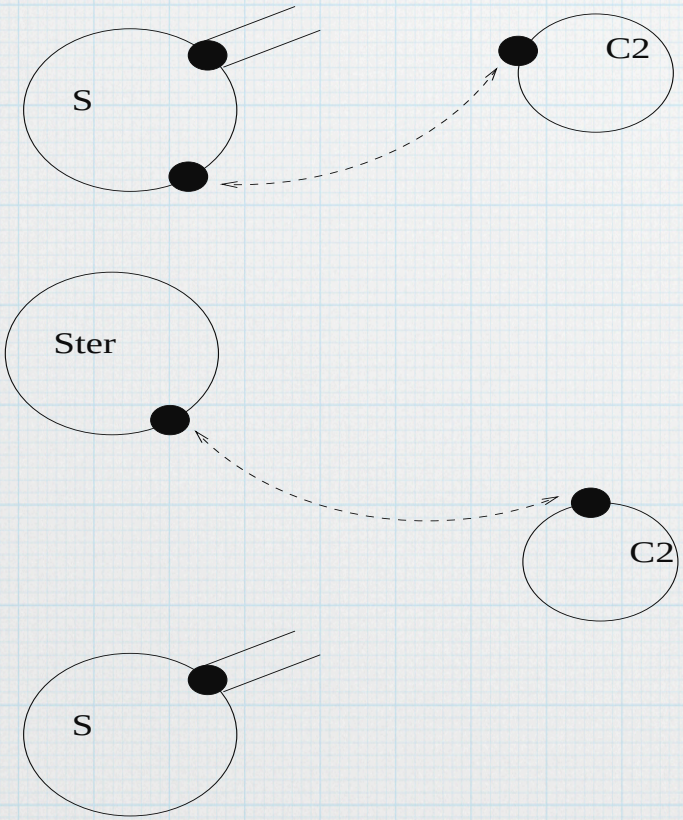
Délégation

S délègue à Sbis le travail avec C1.

Sbis ferme sa copie de la file d'attente.

S ferme sa copie du circuit virtuel avec C1.

Délégation - il en reste



S délègue à Ster le travail avec C2.

Ster ferme sa copie de la file d'attente.

S ferme sa copie du circuit virtuel avec C2.

Types de serveurs

Types de serveurs

Itératif
- ne traite qu'une seule demande de connexion à la fois,
- concevable si les requêtes sont (très) courtes et/ou indépendantes les unes des autres,
- en mode conecté l'établissement de la connexion peut devenir le goulet d'étranglement.

Concurrent
- autant de serveurs que de demandes de connexion,
- multiplication du serveur de base (clônes),
- simple et efficace, mais peut devenir encombrant si les requêtes sont courtes et/ou indépendantes et /ou nombreuses.

Exercice : Citer un exemple positif et négatif dans chaque cas.

Plus difficile : citer un exemple d'application dans lequel aucun des deux cas n'est adapté.

[10pt]article [usenames]color amssymb amsmath [utf8]inputenc

Chapitre – Mise en Œuvre d'Applications

Problèmes, Principes et Solutions

Le Besoin

Problème : La délégation permet de résoudre indirectement une situation de blocage. Mais si on se retrouve avec plusieurs boîtes réseau dans un processus, la question est toujours : comment ne pas rester bloqué, ou comment savoir si un message est disponible dans une BR, sans rentrer dans une situation bloquante.

Deux solutions

1. Faire des entrées-sorties non bloquantes, c'est-à-dire modifier le comportement des entrées-sorties en mode non bloquant. Voir à ce sujet l'appel fcntl() pour les fichiers et même ioctl() pour les périphériques en général.
   Principe : associer le mode non-bloquant à un descripteur. Toute entrée-sortie reçoit alors un résultat.
   Défaut de ce principe : souvent on fait une boucle sur l'entrée-sortie jusqu'à réception d'un résultat positif, boucle appelée attente active.

Solutions - suite

1. Déléguer au système la responsabilité de réveiller le processus lorsqu'un événement est disponible dans un ensemble de descripteurs.
   Principe : donner au système la liste des descripteurs à scruter ; ajouter un délai maximal si nécessaire ; demander au système de réveiller le processus dès qu'un événement est disponible ou que le délai est dépassé.
   Avantage : pas d'attente active, donc pas de mobilisation inutile de ressources, avec un petit inconvénient : rester bloqué (...) jusqu'au réveil...

Déblocage

Déblocage par Réveil

Un processus peut demander au système de se faire réveiller si :

- un événement a eu lieu sur une BR en réception,
- un événement a eu lieu sur une BR en expédition,
- une exception est intervenue sur une BR,
- ou si un délai maximal a été dépassé, même si aucun événement n'a eu lieu. Ce délai est extensible : immédiatement jusqu'à infini.

Exemple :
me réveiller s'il y a quelque chose à consommer (lire) dans la BR_1,
ou si la BR_2 est enfin prête à une expédition,
et au pire dans 5 secondes et 234 μsecondes.

Multiplexage

On parle de multiplexage des entrées-sorties pour un fonctionnement de ce type, permettant d'attendre plusieurs événements simultanément.

Un appel système, select(), permet de réaliser cette attente. Sa syntaxe :

```
int select( int nbsurv,fd_set *desc_en_lect,
        fd_set *desc_en_ecr,fd_set *desc_en_excp,
        struct timeval *taïmeout )
```

Avec la signification suivante :

nbsurv	$\longrightarrow$	limite sup. de descripteurs à surveiller
desc_en_lect	$\longrightarrow$	ensemble à surveiller en lecture
desc_en_ecr	$\longrightarrow$	ensemble à surveiller en écriture
desc_en_excp	$\longrightarrow$	ensemble à surveiller en exception
taïmeout	$\longrightarrow$	intervalle max à attendre

Attention au retour : nombre de descripteurs ou 0 (temps) ou -1 (erreur)

Petite Description des Structures

Qu'est-ce qu'un **`fd_set`** ?
Un ensemble (tableau) de booléens où l'indice est le numéro de descripteur, la valeur (vrai/faux) de chaque élément indiquant s'il faut ou non prendre en compte ce descripteur dans la scrutation.

Exemple :

0	1	2	3	4	5	6	7	8	9	...
f	f	f	v	f	v	f	f	f	f	...

indique qu'on veut scruter les descripteurs 3 et 5 afin d'annoncer au processus demandeur si un événement est disponible, c'est-à-dire s'il y a quelque chose à lire, écrire, ou en exception, selon la demande faite par le processus.

Mise en Œuvre

La mise en œuvre consiste à :

- declarer les ensembles,
- positionner les descripteurs : mettre à vrai les cases correspondantes,
- lancer la scrutation avec select(),
- lors du réveil tester le résultat et agir en conséquence.

Attention : le résultat indique le nombre de descripteurs à traiter ou la cause du réveil (délai, erreur), mais le traitement reste à faire !

Un ensemble de macro-définitions permet de faire des opérations sur des ensembles `fd_set`.

FD_ZERO(fd_set *ens_desc) ; →initialiser à faux un ensemble
FD_SET(int desc, fd_set *ens_desc) ; →positionner à vrai
FD_CLR(int desc, fd_set *ens_desc) ; →repositionner à faux
FD_ISSET(int desc, fd_set *ens_desc) ; →test de l'état

Exemple

```
//obtenir les descripteurs des BR et fichiers concernés
... //déclaration
fd_set detennis ;
...boucle de traitement
//indispensable de réinitialiser
FD_ZERO(&detennis) ;
FD_SET(descBravo, &detennis) ;
//idem pour les autres descripteurs, de fichiers ou BR
nbsurv= .... ;
int erstice=select(nbsurv, &detennis, NULL,NULL,NULL) ;
//Debout, il y a eu déblocage ; il s'est passé quelque
chose
//selon résultat erstice : si 0 délai dépassé ;
//              si positif où (quel descripteur) ?
if FD_ISSET(descBravo, &detennis){
recv(descBravo,.....) ; //traitement réception....}
//et ainsi de suite pour tous les descripteurs
```

Compléments Exemple

Comment déterminer un délai ?

En affectant des valeurs dans la structure :

```
struct timeval {
long tv_sec ; /* secondes */
long tv_usec ; /* microsecondes */} ;
```

Utilisation :

```
struct timeval attenteMax ;
attenteMax.tv_sec = (long) (5) ;
attenteMax.tv_usec = (long) (234) ;
int erstice=select(nbsurv,
&detennis,NULL,NULL,&attenteMax) ;
//si (erstice==0) {traiter dépassement attenteMax} ;
```

Enfin, nbsurv est max(tous les descripteurs à scruter) + 1 . C'est la rançon du système fatigué de compter.

Exercices

1. Quel est l'état général des tableaux de descripteurs (ensembles `fd_set` déclarés) au retour de select ?
2. Que doit-on attendre comme résultat de select() si parmi les descripteurs positionnés en lecture on met celui associé à `stdin` ?
3. **Attention** : il faut distinguer les deux cas
   - délai avec une valeur nulle (nombre de secondes et microsecondes nul),
   - délai infini, donc avec un pointeur valant `NULL`.

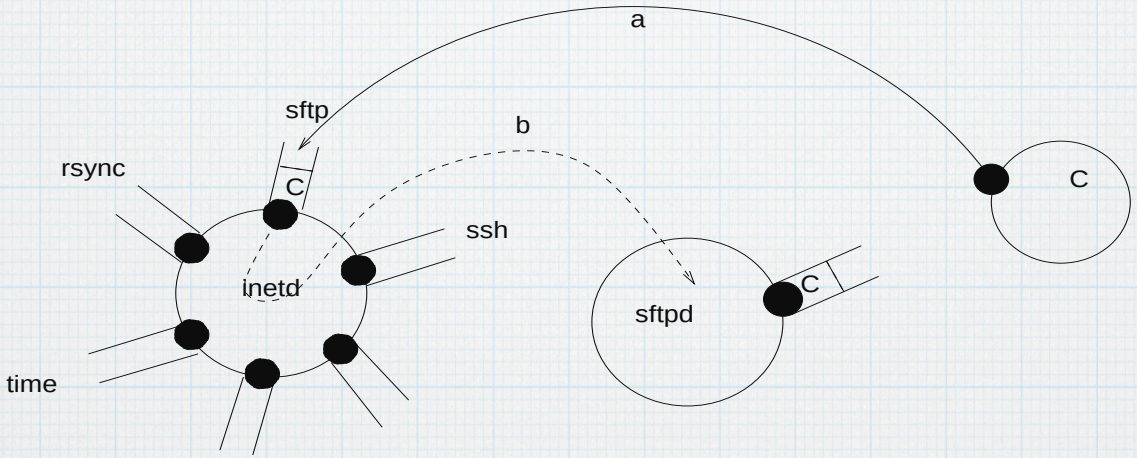
   Analyser le premier cas et donner les résultats possibles de select().

Le Grand Écouteur : Inetd

Rôle de portier pour plusieurs services ; les plus classiques : transferts de fichiers et connexions à distance.
On parle de serveur multiport.

1. crée un point de communication (BR publique) pour le compte de chaque service ;
2. demande le réveil (`select()`) sur l'ensemble des descripteurs créés ;
3. le réveil est provoqué par une demande de connexion d'un client sur l'un des ports <<surveillés>> ;
4. Dès qu'un descripteur est prêt en lecture alors génération du processus assurant le service correspondant : clône et recouvrement (`fork()` et `exec()`).

Schéma Serveur Multiport



- Après la génération du serveur correspondant, on a un fonctionnement classique, sans avoir à regénérer un clône.
- Noter qu'un serveur multiport peut écouter pour des services en tous modes (connecté ou non).
- **Question** : Comment peut-on savoir si un serveur est lié à une génération par inetd ? Plusieurs réponses possibles.
- **Plus difficile** : Citer un exemple où ce fonctionnement est inapproprié et même néfaste.