FMIN104 : Réseaux - Communication

Hinde Bouziane (bouziane@lirmm.fr)

UM2 - LIRMM

Notes de Cours

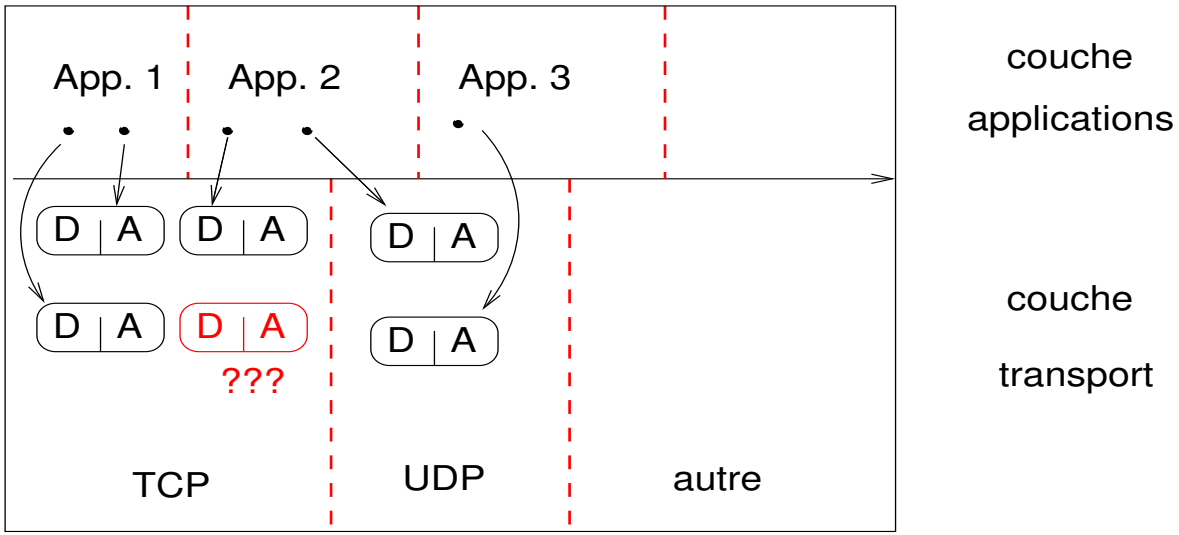
Remerciements à Ehoud Ahronovitz

1. Chapitre 3 – Mise en œuvre d'applications distribuées
   - Généralités
   - Le modèle client - serveur
   - Interface de programmation
   - Communications - mode sans connexion
   - Communications - mode connecté
   - Du blocage des entrées-sorties
   - À titre documentaire

Communications à travers le réseau - vision des applications

- Couche directement au dessus du transport : l'interface de programmation offre un accès au niveau transport
  - voir le modèle OSI (7 couches) et le modèle Internet (4 couches).
- Communication par des boites réseaux (BR) : boîtes où l'acheminement est pris en charge par *des réseaux*
  - analogie : boîtes postales où le transport est effectué par *la poste*
- Une application peut obtenir plusieurs boîtes.
- Chaque boîte a 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 boîtes réseaux



Attention : chaque protocole de transport fait sa propre numérotation des boîtes. Donc un numéro de boîte n'indique pas le protocole utilisé.

? ? ? : pas de BR sans application associée.

Protocoles sous-jacents - protocoles Internet

Protocoles de la couche transport :

- TCP : *Transmission Control Protocol*
- UDP : *User Datagram Protocol*

Services offerts :

TCP	UDP
fiable	non fiable
ordre garanti	ordre non garanti
duplication impossible	duplication possible
mode connecté	**sans connexion**
orienté flot	**orienté message**

Signification

- **Fiabilité** : retour d'un résultat à l'application, éventuellement négatif. Exemple : TCP est capable de vérifier que le destinataire est prêt à recevoir les données et d'informer l'application du cas contraire.
- **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.
  Comment ? TCP est capable de découper des gros paquets de données en paquets plus petits pour que IP les accepte, de numéroter les paquets, et à la réception de vérifier qu'ils sont tous bien arrivés, de redemander les paquets manquants et de les ré-assembler avant de les donner aux applications.
- **Duplication impossible** : s'il y a eu une double réception d'un paquet, le protocole la traite et l'application ne s'en aperçoit pas.

Signification - suite

- **Mode connecté** : les BR des deux extrémités sont liées (connectées) une fois pour toutes. A partir de là, toute les données déposées dans une BR seront transmises à la BR de l'autre extrémité. On parle de *circuit virtuel* entre deux applications.
  Attention : la BR est utilisée pour communiquer de façon exclusive avec une seule autre BR.
  Analogie : le téléphone.
- **Mode non connecté** : a chaque dépôt d'un message dans une BR, l'expéditeur spécifie l'adresse du destinataire. De son coté, le destinataire extrait (depuis une BR) des messages envoyés par un ou des expéditeurs quelconques. A partir de là, l'expéditeur est connu et une réponse peut lui être envoyée.
  Analogie : le courrier postal.

Signification - fin

- **Orienté flot** : le contenu expédié est vu comme un flot continu de caractères. 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, un message n'est pas découpé. 1 lecture $\leftrightarrow$ 1 écriture.

Définitions

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 entités communicantes (protocole d'application).

Fonctionnement

- Un processus "serveur" assure un service : il tourne en permanence en attendant les 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 $\rightarrow$ traitement rapide ou délégation.

Notion de socket

La socket (en français, prise) est une notion qui étend celle de tube. De la même manière, elle permet de définir un canal de communication entre deux processus, sauf qu'elle permet en plus :

- la communication distante (en réseaux) ;
- le choix de différents protocoles de communication.

Dans la suite, deux modes :

- connecté et orienté flot (SOCK_STREAM) ;
- non connecté et orienté message (SOCK_DGRAM).

Remarque : d'autres modes existent (voir documentation).

Interface Socket

Ensemble de primitives permettant de réaliser des applications communiquant sur un réseau. Ces primitives fournissent au programmeur l'accès (l'interface) à la couche transport.

Actions :

- se faire allouer (créer) une ou plusieurs BR locale(s)
- identifier l'application distante
- envoyer des messages
- consulter les messages entrants
- rendre ou fermer la BR si plus nécessaire

Interfaces et implémentations simplifiées sont fournies à l'adresse :

`http://www.lirmm.fr/~bouziane/enseignement/FMIN104/Sock/`

Création d'une BR locale

- Mode non connecté et orienté message

```
Sock breLoc(SOCK_DGRAM, (short)31470, 0);
int descbreLoc;
/* on recupere le descripteur */
 if (breLoc.good())
   descbreLoc = BreLoc.getsDesc();
 else {
   cout<< "problème avec BR locale"<< endl;
   exit(1);
 }
```

- Mode connecté et orienté flot

```
Sock breLoc(SOCK_STREAM, (short)31470, 0);
... // pareil qu'avant
```

Désignation d'une BR distante

```
SockDist saBr("toto.lirmm.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 (domaine, numéro de port, adresse IP). Elle servira par la suite pour le dialogue.

Question : Est-il recommandé de réserver les numéros de ports comme dans les exemples précédents ?

Réservation des ports

Réponse : non.

- Les numéros peuvent être utilisés par d'autres applications,
- pire, ils peuvent être utilisés par des applications courantes, dites *bien connues*,
- 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

- 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 identifiés ; 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.
  - Tous les serveurs `sshd` utilisent strictement le port numéro 22 ;
  - Tous les serveurs `httpd` utilisent *par défaut* le port numéro 80.

Remarques

- Un client peut localiser les serveurs connus, dès lors que ces numéros réservés sont enregistrés dans un fichier local. Le contenu de ce fichier 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`).

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()		

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

Sans connexion - le dialogue

- Expédition

```
int retourSend = sendto(descbreLoc, msg,
                        sizeof(msg), 0,
                        (sockaddr *)adrsaBr, lgsaBr);
```

- Reception

```
int retourRecv = recvfrom(descbreLoc,
                          tamponReception, lgReception,
                          0, NULL, NULL);
```

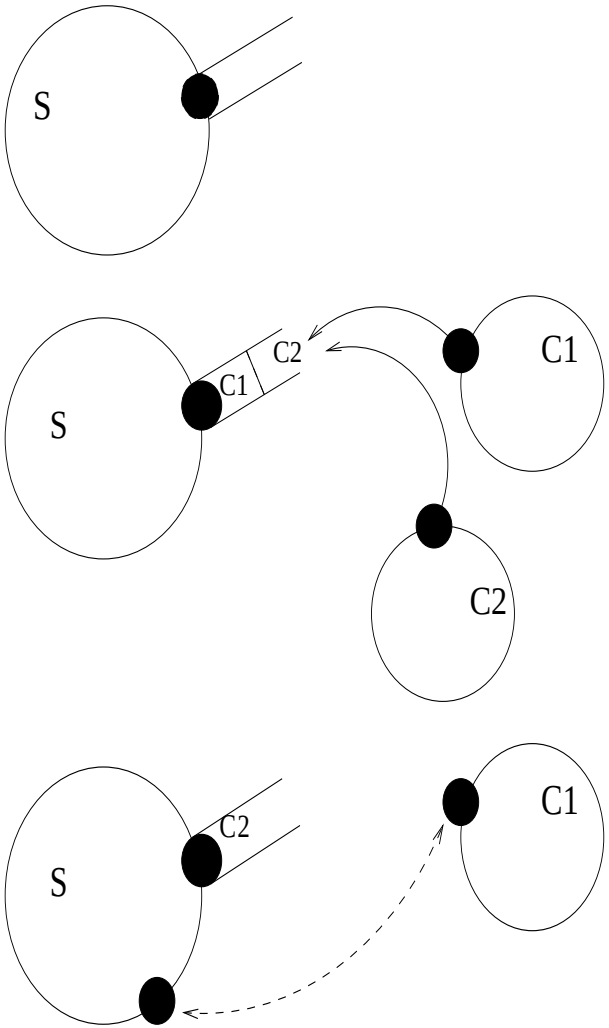
Remarques :

- `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.
- Il n'est pas nécessaire d'être en réception pour recevoir le message.

Mode connecté - introduction

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	savoir accepter l'arrêt
ou	ou
savoir accepter l'arrêt	savoir arrêter

Mode 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

Mode connecté - protocole TCP un retour

- Orienté connexion avec création d'un circuit virtuel
- Bidirectionnel
- Fiable et sans 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 négociation et le contrôle (accusés de réception, flux, . . .) pour 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()		
read() ou recv()		*dialogue*
shutdown()		*fin partielle*
close()		*fin*

Exemple - 1 - côté serveur

Préparation de la BR publique

```
 //suppose que la BR est publiquement connue
Sock brPub(SOCK_STREAM, (short)(21345), 0);
int descBrPub;
if (brPub.good()) descBrPub=brPub.getsDesc();
int res = 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 - 2 - côté client

Préparation de la BR privée et demande de connexion

```
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 de connexion a été déposée dans la BR publique du serveur.

Syntaxe - 3 - dialogue

Côté Serveur

```
char rBuf[256]; char sBuf[]="doremifa solasido";
int res = recv (descBrCv, rBuf, sizeof(rBuf), 0);
...
int ox = send (descBrCv, sBuf, strlen(sBuf), 0);
```

Côté Client : Pareil, en utilisant la boîte réseau privée locale.

Notes :

- il n'y a pas à spécifier le destinataire : `tcp` a bien fait son travail ;
- il est possible 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.

Constat

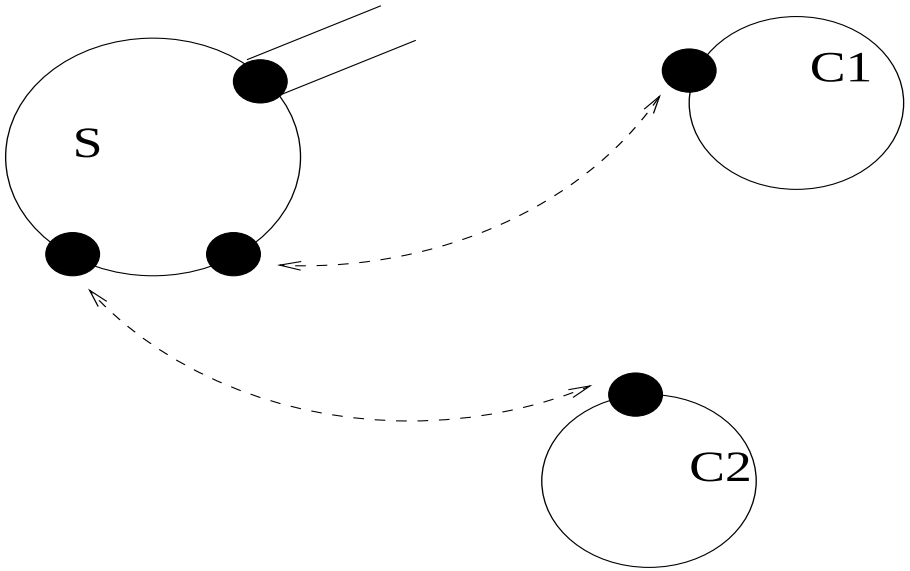
Les entrées-sorties décrites sont majoritairement bloquantes :

- 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.

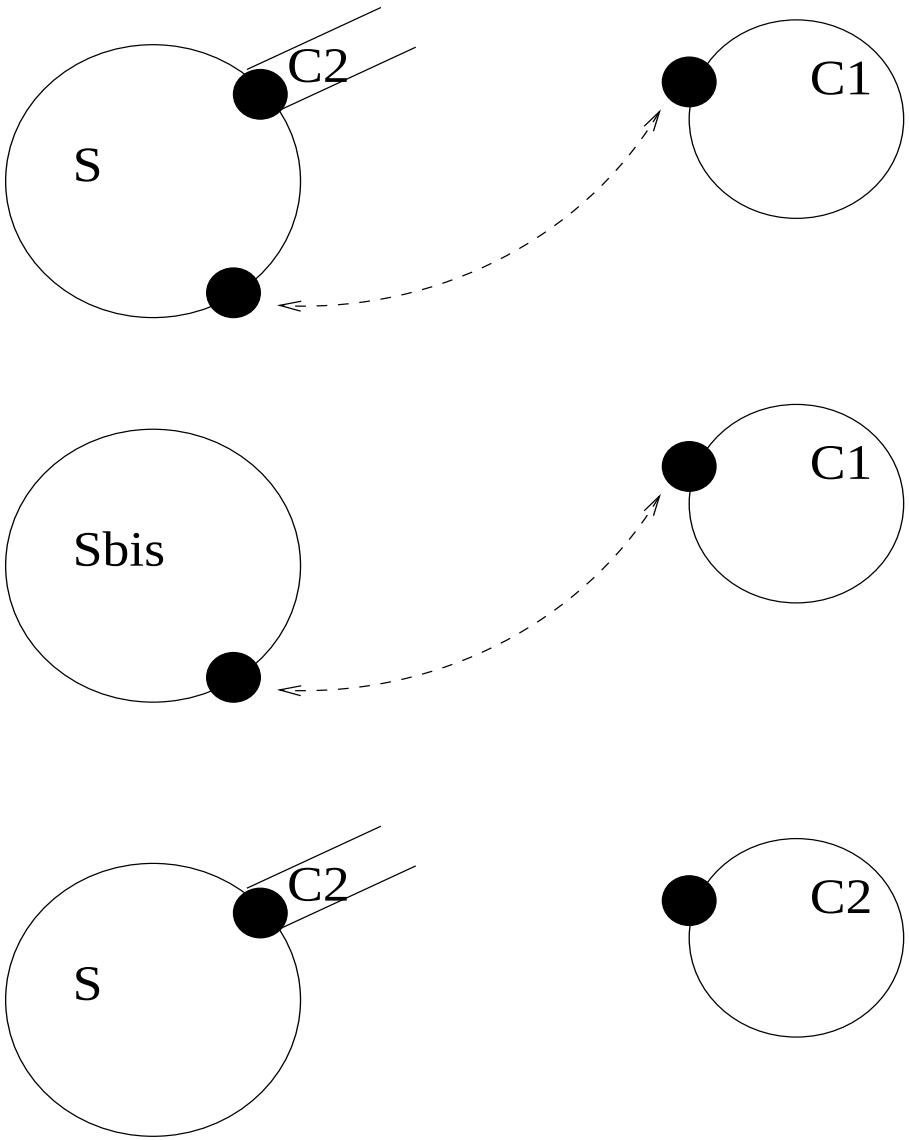


Coté serveur, si on attend une réception sur une des BR et si on n'a pas de réponse, les autres clients patientent lamentablement, quel que soit le point d'attente

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

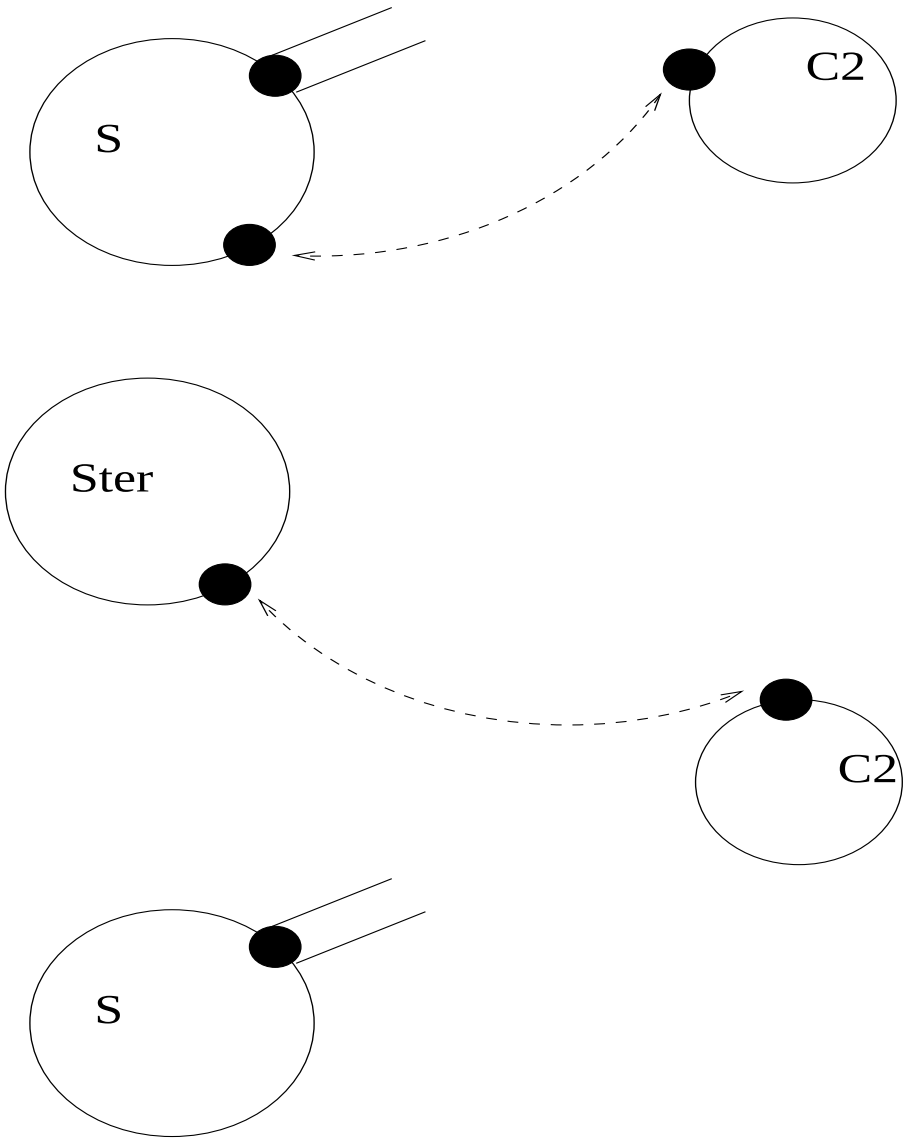


S délègue à Sbis le travail avec C1.

Sbis ferme sa copie de la file d'attente.

S ferme sa copie du circuit virtuel avec C1.

Délégation - suite



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

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 connecté 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.

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.

À titre documentaire 1 - détails des primitives

Obtenir un descripteur pour une BR

```
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 et le plus souvent (voir manuel `protocols` et fichier `/etc/protocols`) ;

retour : descripteur ou -1 en cas d'echec (et errno positionné).

À titre documentaire 2 - détails des 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 (succès) ; -1 (échec avec errno positionné).

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 3 - détails des 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 ;
retour	nombre de caractères envoyé ; -1 (échec avec errno positionné).

À titre documentaire 4 - détails des 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 ;
retour	nombre de caractères effectivement reçus ; 0 en cas de fermeture par l'acolyte ; -1 (échec avec errno positionné).

À titre documentaire 5 - détails des primitives

Mode connecté - création du circuit virtuel (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 avec errno positionné).

Mode connecté - attente de connexions :

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

descripteur celui de la BR publique ;

lgmax longueur maximal de la file de connexions ;

retour 0 (succès) ; -1 (erreur).

À titre documentaire 6 - détails des primitives

Mode connecté - demande de connexion (coté client) :

```
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 (succès) ; -1 (erreur avec errno positionné).

À titre documentaire 7 - détails des 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 avec errno positionné).

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

lg nombre maximum de caractères à recevoir.

retour nombre de caractères effectivement reçus ; -1 (erreur avec errno positionné).