TP/TD machine
INFO3 "Conception de protocoles de communication"
Programmation Internet via API-socket en C

vincent.ricordel@polytech.univ-nantes.fr

1 Pour commencer

Le but est de vous faire développer des "outils" communiquant au travers le réseau. Ce seront des briques de base que vous pourrez reprendre pour les adapter à vos besoins (et en particulier ceux du Mini-Projet "Conception Protocoles"). Vous programmerez en utilisant l'API-socket en C. Sous Linux, voici quelques commandes utiles que vous connaissez déjà :

- la configuration de vos interfaces réseau : `ifconfig`
- la connectivité d'une cible : `ping`, et la route empruntée par le paquet pour y aller : `traceroute`
- l'état des sockets : `netstat -utnp` ou `netstat -lnp`

Ayez le réflexe d'utiliser le manuel en ligne (cf. commande `man`).
Pour compiler le fichier source `fich.c` et obtenir l'exécutable `cible`, vous utiliserez la commande :
`gcc -Wall -o cible fich.c`
Attention à ne laisser aucun "Warning". Vous aurez aussi le temps d'écrire un `Makefile`, pour cela voir l'exemple en annexe A.
Pour construire votre "paquet", regardez en annexe A.

2 Socket en mode non connecté (UDP)

Tous les programmes que nous ferons ensemble reposent une architecture de type client/serveur, le client qui fait les requêtes, le serveur qui y répond. Ici le dialogue repose sur l'utilisation du protocole UDP. Ce service réseau permet le minimum : le client envoie un ou des paquets (on parle de "datagrammes"). Ceux-ci voyagent de façon indépendantes dans le réseau. Il n'y a pas de contrôle effectué (erreurs, séquencement, ...). Conséquence : on ne

pourra pas voir grand chose (juste vérifier que le serveur écoute bien sur le port `ps` avec : `netstat -uln`). Le principe de l'algorithme est montré à la figure 1, pour vous aider à coder n'oubliez pas la documentation fournie et le manuel en ligne. Et la figure 2 illustre le dialogue client/serveur.

Remarque : Le "broadcast" s'appuie sur UDP pour la diffusion d'un datagramme sur le réseau local. Au niveau du code, il faut alors positionner une option sur la prise à l'aide de la fonction `setsockopt()`

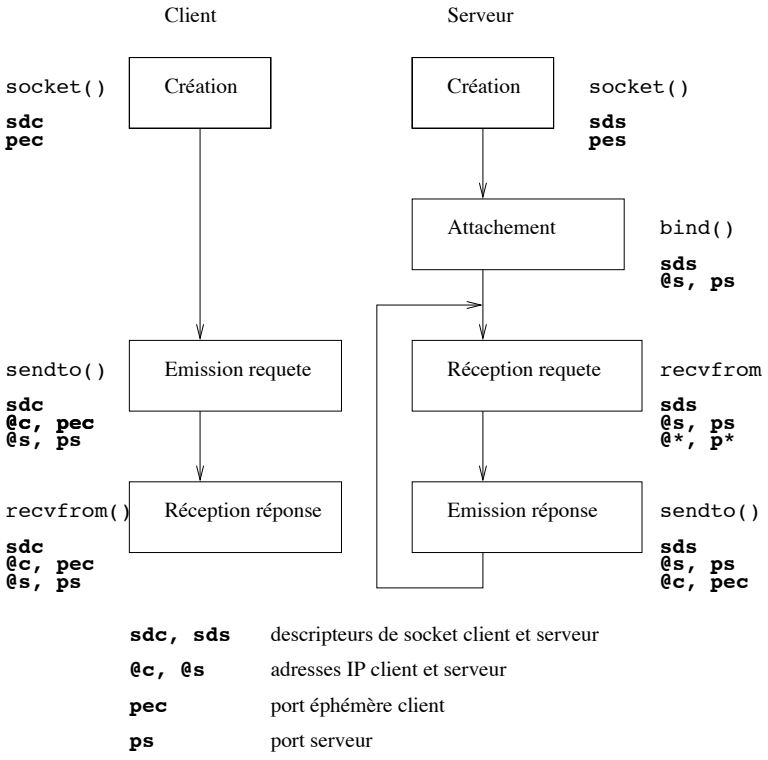

Figure 1: Socket UDP.

3 Socket en mode connecté (TCP)

3.1 Serveur séquentiel

Il s'agit cette fois de s'appuyer sur le protocole TCP offrant un service en mode connecté. Il y aura notamment 3 phases lors de la communication entre le client et le serveur : la connexion, l'échange d'information et enfin la libération. Il sera donc possible cette fois de vérifier (à l'aide de `netstat -tn`) l'état de connexion des sockets. Le principe de l'algorithme

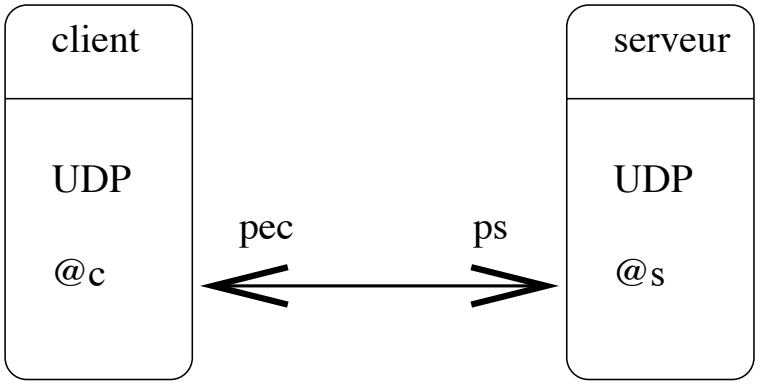

Figure 2: Dialogue client/serveur.

est illustré à la figure 3. Coté serveur, il existe une file pour la mise en attente des demandes de connexion (cf. `listen()`), et le serveur est en mode séquentiel : il traite les requêtes les unes après les autres. Si le service est long, les clients s'impatientent. Si la file est pleine, les requêtes qui parviennent alors sont rejetées. La figure 4 illustre la filtre d'attente créée, remarquez le rôle de `sds` qui écoute (en permanence), et celui de `newsds` qui rend le service courant et est fermé à son issue (avant passage à l'éventuel client suivant).

3.2 Serveur parallèle

L'algorithme ressemble à celui précédent, mais cette fois le serveur doit fonctionner en mode parallèle. Le principe de l'algorithme est illustré à la figure 5 et la figure 6 montre le système de file d'attente créé : un premier processus (dit "père") écoute et reçoit dans sa file d'attente les requêtes clientes, les traitements des clients sont confiés à d'autres processus (les "fils") qui tournent en parallèle. La seule limite (lenteur) est donc liée à la capacité de traitement du PC. Coté mise en oeuvre on utilisera la fonction système `fork()`.

A Makefile

La commande `make` permet de lancer toute une série de commandes à interpréter par le système (`make` fait aussi tout un ensemble de tests, par exemple une vérification des dates de création des sources, ...). C'est très utile notamment pour lancer une compilation. Dans un fichier `Makefile` sont placés des items du type :

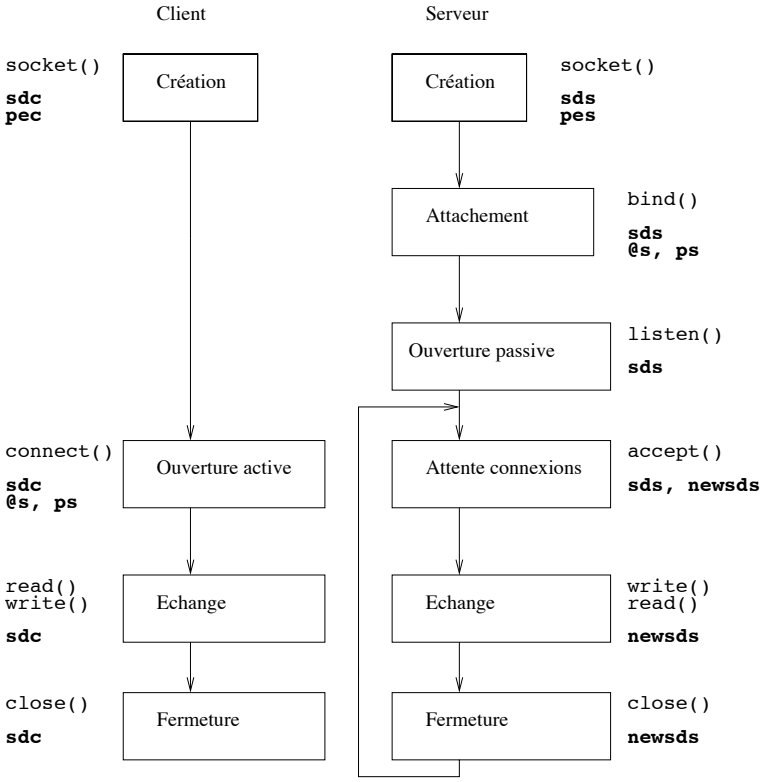

Figure 3: Socket TCP, serveur séquentiel.

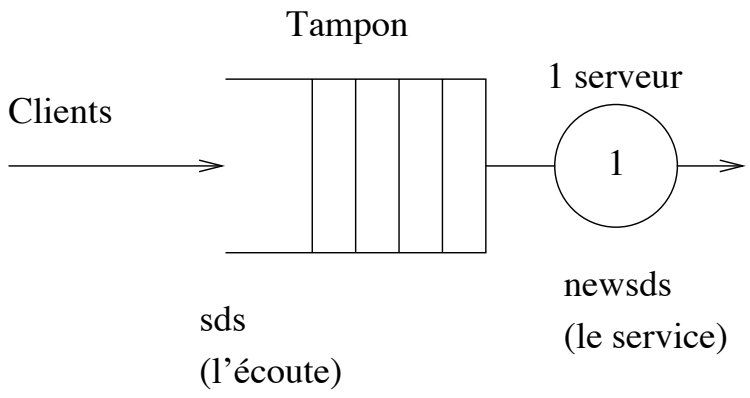

Figure 4: File d'attente du serveur séquentiel.

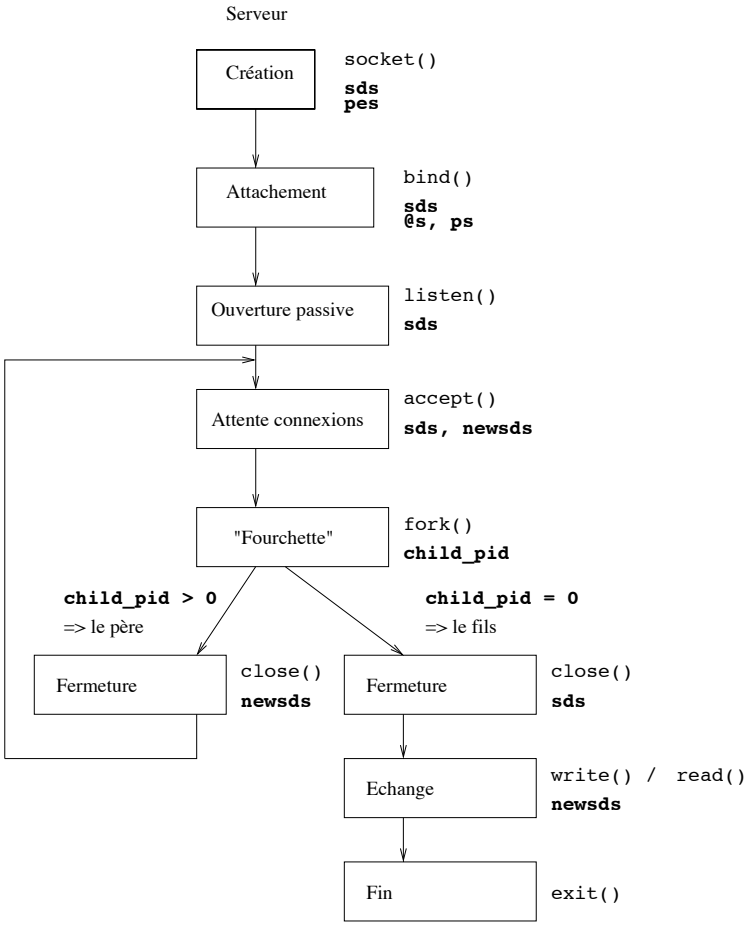

Figure 5: Socket TCP, serveur parallèle.

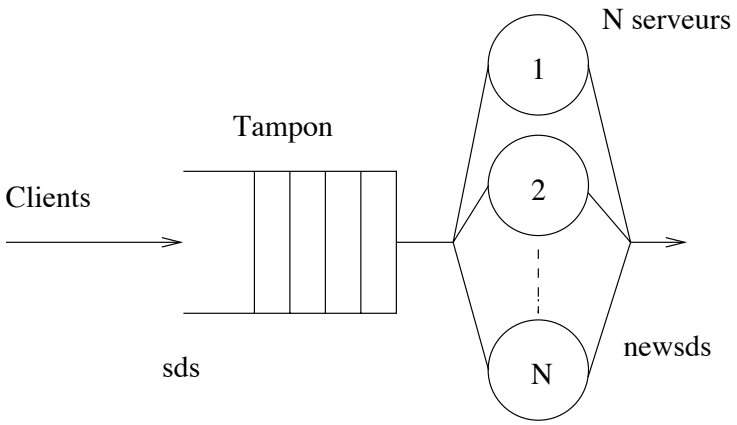

Figure 6: File d'attente du serveur parallèle.

```
cible: dépendances
    commandes (pour réaliser la cible à partir des dépendances)
```

Si l'éxécutable dépend des 2 fichiers source `fich1.c` et `fich2.c`, et fait appel aux fonctions mathématiques de la librairie standard. Le `Makefile` peut être :

```
all: executable clean

executable: fich1.o fich2.o
	gcc -Wall -o executable fich1.o fich2.o -lm

fich1.o: fich1.c
	gcc -Wall -c fich1.c

fich2.o: fich2.c
	gcc -Wall -c fich2.c

clean:
	rm *.o *.*~
```

Et ça ce complique car la commande `make` use de règles implicites et de symboles (`$@, $^ , ...`), et impose une syntaxe rigoureuse. Pour en savoir plus regardez :
> `info make`

Par exemple notre `Makefile` précédent devient :

```
CC=gcc

all: executable clean

executable: fich1.o fich2.o
	$(CC) -Wall -o $@ $^ -lm

clean:
	rm *.o *.*~
```

Pour lancer la commande :

- comportement par défaut avec recherche du fichier `Makefile` dans le répertoire courant :
  > `make`
- recherche du fichier cible `fichier_makefile_cible` au bout du `/chemin` :
  > `make -f /chemin/fichier_makefile_cible`

A Structure du paquet

Pour construire votre "paquet", vous pouvez définir un type structuré, par exemple :

```
struct  Paquet {
        int champ1;
        int champ2;
        char information[10];
        };
```

La définition d'une variable de type structuré `Paquet` se fait ainsi :
`struct Paquet p;`

Vous accéderez à ses champs, ici le `champ1`, ainsi :
`p.champ1 = 18;`

Et vous calculerez sa taille, ainsi :
`taille_paquet = 2 * sizeof(int) + 10 * sizeof(char);`