Systèmes d'Exploitation - Examen

12X009 - TP05

Noah Munz (19-815-489)

Département d'Informatique
Université de Genève

Mardi 31 Janvier 2023

Code: Lien GitHub du code
Plan: Lien GitHub du plan de l'implémentation

Table of Contents

Table of Contents

Rappel : But du TP

- Implémenter un serveur établissant une connection avec plusieurs connections connections clients afin de jouer au “guessing game” avec chacun simultanément

Les principaux défis de ce TP sont :

- L'implémentation de l'architecture client/serveur TCP correspondante
- Manipuler les appels liés aux sockets (création, écoute, attente, connection...)
- Gérer simultanément un proccessus enfant par client plus un parent qui attend une connection d'un client puis crée un enfant pour jouer avec lui.

Table of Contents

TADs & leurs relations: Décomposition modulaire

3 modules (+ `server.c`, `client.c`) ont été créés pour la réalisation du TP :

1. Le module `optprsr.c` (repris des TPs 2 à 4)
   contient 5 fonctions, servant à l'extraction & copie. . .des arguments passés paramètres.

2. Le module `util.c` (repris des TPs 3 à 4)
   Fonctions servant divers usages allant de la gestion d'erreurs à la gestion de chaînes de caractères, en passant par des wrappers qui incluent lesdites fonctions de gestions d'erreurs.

TADs & leurs relations: Décomposition modulaire

3. Le module `functions` qui gères les travaux communs que doivent faire le client et le serveur pour pouvoir communiquer ensemble (e.g. création d'une struct `sockaddr_in`)

4. `client.c` Contient le "driver code" pour se connecter au server et jouer avec lui. Se décompose en une partie *connection*, (i.e. création de socket, connection au socket)
puis en une partie *jeu* (i.e. lecture/écriture sur la socket)

5. `server.c` Contient le "driver code" pour ouvrir des sockets attendre des connections clients et créé des enfants pour jouer avec les clients qui arrivent. Se décompose en une partie *connection*, (i.e. création de socket, attente & acceptations de nouveau clients) géré par le parent
puis en une partie *jeu* (i.e. lecture/écriture sur la socket) géré par les enfants.

TADs & leurs relations: Structures de données utilisées

L'implémentation de structure n'a pas été nécessaire.

TADs & leurs relations: Décomposition fonctionnelle

Client :

Pour faire tout ce dont il a besoin, le client doit :

1. Créer une socket
2. Se connecter au server en initiant la connection sur la socket avec l'adresse et le port donné
3. Lire/écrire de/sur la socket
4. Fermer la socket

TADs & leurs relations: Décomposition fonctionnelle

Pour ceci, nous avons implémenté les différentes fonctions suivantes :

1. module `functions` : `int new_socket()` crée un nouveau socket de famille `AF_INET`, type `SOCK_STREAM` et de protocole 0 (TCP/IP)
2. même module :
   `sockaddr_in new_sockaddr(int port, const char* addr_reppr)`
   Créé une socket address pour un client ou un server où `port` est le numéro de port à partir duquel on va "bind" la socket et `addr_repr` est un string (qui peut être nul quand on l'appel depuis `server`) qui est simplement la représentation de l'addresse IP (par défaut vaut `INADDR_ANY`)
   retourne une `struct sockaddr_in`, il ne reste plus qu'à appeler `socket()` pour initier la connection (client) et finir l'étape binding-listen-accept (server)

TADs & leurs relations: Décomposition fonctionnelle

Pour la lecture/écriture on a simplement utilisé les appels systèmes `read()` et `write()`. En effet, les sockets sont encapsulés par des descripteur de fichiers i.e. tous les appels qui opèrent sur des socket prennent un descripteur de fichier en paramètre comme si c'était un fichier normal et font les opérations nécéssaire avec.

De la même manière, pour fermer la socket on a juste à utiliser `close()`.

TADs & leurs relations: Décomposition fonctionnelle

Serveur :

Pour faire tout ce dont il a besoin, le serveur doit :

1. Créer une socket
2. *"bind"* la socket serveur à une adresse
3. Ecouter/attendre le début d'une connection client
4. Accepter la connection, obtenir la socket client et créer un enfant pour s'en occuper pour pouvoir continuer à attendre d'autre connections en même temps
5. Lire/écrire de/sur la socket
6. Répéter l'étape 4 dès que nécessaire
7. fermer la socket client
8. fermet la socket serveur

TADs & leurs relations: Décomposition fonctionnelle

Pour ceci, nous avons implémenté les différentes fonctions suivantes : Les points 1) 5) 7) et 8) ont déjà été abordés

2. On a simplement appelé la fonction `bind()` avec une `struct sockaddr_in` et un file descriptor lié à la socket, crée comme vu plus haut.
3. Pour la partie écouter et gérer les clients en même temps on a une boucle infine du type `for (;;) {...}` où au début de la boucle notre processus attend avec un appel à `accept()` puis dès qu'il a fini d'attendre (i.e. on est juste à la ligne d'après) il va `fork()` et laisser l'enfant s'occuper et continue sa boucle.

   En réalité pour ce tp on fait un double fork pour faire en sorte que tous les procesus qui discutent directement avec le client soit des petits-enfants du processus qui attend les connections et que leurs parent direct meurent. (afin qu'ils deviennent tous orphelins et qu'ils soit récupéré par `init` et terminés correctment à leurs mort, pour éviter les zombies) En effet à cet moment nous n'avions pas encore vu la gestions de signaux, avec `SIGCHLD` ...

Table of Contents

Tests realisés pour valider le fonctionnement du TP

Les tests réalisés ont simplement été de "jouer au jeu", avec un ou plusieurs clients, la procédure attendu étant assez guidé l'étendu de panel de test à réaliser en est moindre.
Cependant elle en demeure non trivial du à l'échange d'informations via un serveur. Pour vérifier si les sockets sont bien fermées on peut utiliser la commande `ss -tn`
`man ss` affiche :
" *`ss` is used to dump socket statistics. It allows showing information similar to netstat. It can display more TCP and state information than other tools.*
`-t` : Display TCP sockets.
`-n` : Do not try to resolve service names. Show exact bandwidth values, instead of human-readable."

Voir le screenshot ci-après pour plus de détail sur les tests.

Tests realisés pour valider le fonctionnement du TP

```
WSL at ~ > BA3 > 12X009-OS-TPs > TP05
tp05-new≡ 11 -1 [22:04:36]
$ ./server 65002
Waiting for clients at port 65002.
PID: 17700
PID: 0
Waiting for clients at port 65002.
Client 4 connected with IP: 127.0.0.1.
Selected value for client 4: 10.
Client 4 proposes: 5
Client 4 proposes: 7
Client 4 proposes: 8
Client 4 proposes: 9
Client 4 proposes: 10
Waiting for clients at port 65002.
PID: 17850
PID: 0
Waiting for clients at port 65002.
Client 5 connected with IP: 127.0.0.1.
Selected value for client 5: 18.
Client 5 proposes: 5
Client 5 proposes: 6
Client 5 proposes: 7
Client 5 proposes: 8
Client 5 proposes: 9
Waiting for clients at port 65002.
PID: 18113
PID: 0
Waiting for clients at port 65002.
Client 4 connected with IP: 127.0.0.1.
Selected value for client 4: 52.
Client 4 proposes: 16
Client 4 proposes: 32
Client 4 proposes: 48
Client 4 proposes: 52
Waiting for clients at port 65002.
```

```
WSL at ~ > BA3 > 12X009-OS-TPs > TP05
$ ./client 127.0.0.1 65002
IP address: 127.0.0.1
Port: 65002
Minimum: 0
Maximum: 64
Guess the number: 16
Proposition sent: 16

Number is higher
Guess the number: 32
Proposition sent: 32

Number is higher
Guess the number: 48
Proposition sent: 48

Number is higher
Guess the number: 52
Proposition sent: 52

Number guessed correctly, you win!
```

Table of Contents

Réponses aux questions (générales)

Code : Lien GitHub du code

Plan : Lien GitHub du plan de l'implémentation