TP6: Communiquer et jouer en réseau

Module ArcSys

L'objectif de ce petit projet est de vous faire écrire un petit programme C communiquant grâce à des sockets. Ce sera l'occasion de réfléchir en pratique à la notion de protocole applicatif. De plus, si vous avez le temps, vous allez pouvoir appliquer ce que vous avez appris sur le jeu des sept couleurs que vous connaissez bien.

1 Nos premières chaussettes

En guise d'échauffement, nous allons créer une application client-serveur très simple : le client envoie une chaîne de caractères au serveur qui lui renvoie la même chaîne de caractères. C'est une application *echo*. Pour cela, on va utiliser des sockets en mode connecté (protocole TCP, socket de type `SOCK_STREAM`). On va construire le code pas à pas en commençant par le serveur. Le code serveur et le code client seront dans des fichiers séparés. Les premières lignes de ces deux fichiers (après mention des auteurs et de la licence) seront :

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

1.1 La chausette serveur

La configuration d'une socket serveur comprend 4 étapes à effectuer dans l'ordre suivant :

▷ **Question 1:** Faire un `man 2 socket` pour voir comment utiliser la fonction `socket(2)` [1], que l'on retrouve très souv pour créer une socket. On utilisera IPv4 et pour le protocole, on mettra `IPPROTO_TCP`. Bien penser à récupérer et traiter l'erreur si la création ne se passe pas bien.

Ensuite, il faut utiliser la fonction `bind(2)` pour connecter la socket à l'adresse locale. Mais d'abord, on va créer une structure de type `sockaddr_in` pour configurer la connexion.

▷ **Question 2:** Où se trouve le fichier qui déclare cette structure ?

On va donc créer une structure de type `sockaddr_in` avec `AF_INET` comme `sin_family`. Pour plus d'infos sur ce paramètre, consulter `http://www.gnu.org/software/libc/manual/html_node/Address-Formats.html`.

Il n'est pas nécessaire de définir tous les champs de la structure. Les champs dont on a besoin sont `sin_addr.s_addr` dont on veut qu'il accepte toutes les adresses et qu'on va donc définir en utilisant `INADDR_ANY` ; et le champ `sin_port` qui correspondra au port passé en paramètre au programme (`argv`).

▷ **Question 3:** Définir la structure décrite ci-dessus. On pourra s'aider des fonctions `htonl(3)` et `htons(3)`.

▷ **Question 4:** Écrire l'appel à la fonction `bind(2)` en utilisant la structure déclarée à la question précédente. On n'oubliera pas de tester si la fonction s'est correctement exécutée.

Maintenant que la socket est connectée comme il faut, elle doit écouter.

▷ **Question 5:** Écrire l'appel à la fonction qui permet de dire à la socket d'écouter. Ne pas oublier que `man` peut renseigner sur les paramètres à passer à une fonction.

Enfin, la dernière étape côté serveur consiste à indiquer à la socket d'accepter les connexions des clients. Il faut pour cela utiliser la fonction `accept(2)` dans une boucle pour que le serveur ne s'arrête pas après une requête.

▷ **Question 6:** Écrire l'appel à la fonction `accept(2)`. On pourra ensuite utiliser la fonction `inet_ntoa(3)` pour afficher sur la sortie standard l'adresse du client.

Maintenant qu'on est connecté à la socket client, on peut recevoir et envoyer des messages. Comme on est en train de coder un serveur *echo*, on doit recevoir un message et le réémettre.

▷ **Question 7:** Écrire l'appel à la fonction `recv`. On utilisera 0 comme flag.

▷ **Question 8:** Écrire l'appel à la fonction `send` qui va permettre de renvoyer la chaîne de caractères reçus. On utilisera 0 comme flag.

On en a maintenant fini avec le serveur.

1. Le 2 entre parenthèse indique qu'il faut faire `man 2 socket` pour voir la documentation de cette fonction.

1.2 La chaussette client

On va maintenant s'attaquer au client. La configuration d'une socket client est plus simple que la configuration d'une socket serveur puisqu'elle ne comprend que 2 étapes : la création et la connexion.

▷ **Question 9:** Écrire l'appel à la fonction `socket` pour créer la socket client.

▷ **Question 10:** Écrire l'appel à la fonction `connect` qui permet de connecter la socket créée au serveur. On utilisera une structure de type `sockaddr_in` pour l'adresse du serveur. On pourra s'aider de la fonction `inet_addr` pour convertir l'adresse du serveur qui sera passée en paramètre en ligne de commande (`argv`).

Ensuite, comme pour le serveur, il faut envoyer et recevoir des messages. Comme il s'agit d'une application *echo*, le client doit envoyer une chaîne de caractères et réceptionner une chaîne de caractères renvoyée par le serveur (avec un peu de chance, la même).

▷ **Question 11:** Écrire l'appel à la fonction `send` qui envoie au serveur une chaîne de caractères passée en paramètre de la ligne de commandes (`argv`).

▷ **Question 12:** Écrire l'appel à la fonction `recv` qui reçoit la réponse du serveur et l'affiche sur la sortie standard.

Et voilà, c'est fini pour cette partie !

1.3 La communication entre chaussettes

▷ **Question 13:** Tester le code écrit précédemment en utilsant l'adresse *localhost* : `127.0.0.1`.

▷ **Question 14:** Tester le code écrit précédemment (client et serveur) en essayant de communiquer avec le code d'un(e) camarade.

▷ **Question 15:** Essayer de déterminer la taille du buffer utilisée par le code serveur de vos camarades.

Pour ceux qui ne sont pas encore au point, un très bon tutoriel se trouve ici :
`http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html`.

2 Jouons à plusieurs (optionnel)

Il est maintenant temps de recommencer à jouer aux sept couleurs. Nous allons juste changer ce qui doit l'être pour pouvoir jouer à plusieurs sur des machines différentes. La première chose à faire est de reprendre le code que vous aviez rendu pour le précédent TP noté.

2.1 Retransmission des matchs en direct

Dans cette première version, nous aurons toujours deux joueurs locaux (soit deux humains, soit deux AI, soit un de chaque). La seule différence est qu'au démarrage de la partie, le programme attendra qu'un observateur se connecte. Ensuite, la partie se déroulera comme avant, mais tous les coups joués seront envoyés à l'observateur pour lui permettre de retransmettre la partie en direct sur une autre machine.

Deux solutions s'offrent à vous pour le protocole applicatif : soit on envoie l'état complet du plateau à chaque mise à jour, soit on n'envoie l'état complet que lors du premier envoi, tandis que les envois suivants ne contiennent que le coup joué. Cette seconde version est préférable car plus économe en bande passante, même si elle demande à l'observateur de savoir recalculer l'effet d'un coup donné.

▷ **Question 16:** Au besoin, refactorisez votre code afin que deux programmes puissent utiliser les mêmes modules d'affichage du plateau de jeu et de calcul de l'effet d'un coup donné.

▷ **Question 17:** Modifier votre programme principal afin qu'il attende une connexion sur le port 7777 par défaut, puis qu'il diffuse le match en direct sur la chaussette ouverte lors de la partie.

▷ **Question 18:** Écrivez un programme observateur pouvant se connecter sur votre serveur et afficher le résultat du match.

▷ **Question 19: (Bonus)** Autorisez les observateurs à se connecter après le début de la partie. Comme l'appel `accept(2)` est bloquant, il est indispensable d'utiliser `select(2)` afin d'assurer que quelqu'un est connecté à la chaussette principale avant d'invoquer `accept()`.

▷ **Question 20: (Bonus)** Autorisez plus d'un observateur à la fois.

▷ **Question 21:** Testez votre programme, commentez et nettoyez votre code.

2.2 Poste mono-client

Nous allons maintenant modifier notre programme pour permettre à l'un des joueurs de jouer depuis une autre fenêtre, possiblement localisée sur une autre machine. Cette possibilité ne sera offerte qu'à un seul joueur,

tandis que l'autre devra encore jouer avec le processus du serveur. Cela demande naturellement d'augmenter le protocole applicatif, qu'il vous faut donc spécifier.

▷ **Question 22:** Proposez plusieurs solutions possibles pour cette extension du protocole, et discutez leurs avantages respectifs.

▷ **Question 23:** Implémenter la solution qui vous semble préférable.

Après cela, vous aurez trois programmes (ou trois modes d'exécution du même programme, activables par des options en ligne de commande). Le **serveur** est le premier processus lancé. Après l'arrivée des autres processus, il lance la partie quand tout le monde est connecté (dans la version de base, un seul processus distant est attendu avant de commencer). L'**observateur** se connecte au serveur et ne fait qu'afficher la rediffusion du match, sans permettre d'y prendre part. Le **client distant** se connecte au serveur et mène une partie normale avec lui.

▷ **Question 24: (Bonus)** Autorisez à avoir à la fois un client distant et un (ou plusieurs) observateurs d'une même partie.

▷ **Question 25:** Améliorez la robustesse de votre programme afin que chaque programme réagisse de façon sensée aux déconnections et aux erreurs de protocole.

▷ **Question 26:** Testez votre programme, commentez et nettoyez votre code.

2.3 Compétition équitable

Nous avons maintenant envie d'utiliser le mode de jeu en réseau pour comparer des AIs (si, si). Pour corser un peu le jeu, on veut tenir compte du temps pris par chaque AI, car il est probablement moins difficile de trouver le coup parfait si l'on n'est pas limité en temps. Imposer cette limitation dans le programme de 7 colors est cependant techniquement assez difficile, et seule la première étape est obligatoire.

▷ **Question 27:** Dans un premier temps, le serveur devra chronométrer le temps pris par les joueurs (distants et locaux) pour communiquer leur coup après l'envoi du coup joué par leur adversaire. En fin de partie, un décompte du total du temps pris par chacun sera affiché.

▷ **Question 28: (Bonus)** Si l'un des joueurs ne répond pas dans le temps imparti (que vous déterminerez), le serveur jouera une couleur aléatoire pour lui.

Cette modification interdit naturellement aux joueurs de modifier leur plateau avant d'avoir reçu la confirmation du serveur, car la couleur jouée peut ne pas être celle qu'ils avaient calculée. Il faut également trouver une solution (probablement à base de `select(2)`, de threads ou de `alarm(2)`) pour que le calcul du prochain coup soit interrompu quand le temps est dépassé. Cela pose enfin des problèmes théoriques intéressants (mais faciles à résoudre en pratique) pour décider quels coups annuler ou non après un timeout.

2.4 Multi-⋆ (multi-star — extra bonus)

Cette section liste des idées intéressantes, mais que rien ne vous oblige à implémenter. Certaines d'entre elles semblent assez naturelles, mais d'autres ressemblent à des pots de miels destinés à piéger les plus procrastinateurs d'entre vous. Soyez prudent.

- Permettre aux deux joueurs de se connecter à distance.
- Permettre à quatre joueurs de jouer ensemble à chaque coin du terrain.
- Permettre à un nombre arbitraire de joueurs de jouer ensemble, en leur donnant des positions équidistantes sur un monde torique.
- Proposez une AI qui tire le meilleur parti du temps imparti. Il s'agit de donner le meilleur coup calculé jusqu'à présent juste avant la date fatidique.
- Permettre au serveur d'héberger plusieurs parties en même temps. Cela ne devrait pas nécessiter de modification du protocole applicatif existant (même s'il peut être nécessaire d'ajouter quelques messages).
- Permettre de jouer avec les programmes d'un autre binôme. Cela vous demandera de faire converger vos protocoles applicatifs.
- Permettre de jouer avec notre programme, dont le binaire est fourni à l'adresse suivante : `http://people.irisa.fr/Martin.Quinson/Teaching/ArcSys/7netcolors-static`. Cela vous demandera d'utiliser wireshark (ou équivalent) pour retrouver le protocole que nous avons choisi d'implémenter.