Développement d'une application Client/Serveur sur TCP/IP

L'objectif de ce TP est d'acquérir les bases de la programmation de la communication en mode connecté répartie entre un rôle de processus client et un rôle de processus serveur.
Ainsi, le choix est fait d'utiliser les services de transport offerts par les protocoles TCP/IP pour supporter les échanges fiables des messages applicatifs, à savoir les requêtes et les réponses, entre les processus clients et serveurs.

On s'appuie sur un exemple applicatif relevant d'un jeu simple. Les deux programmes à développer concernent respectivement le rôle de **client** et celui de **serveur**. Leur programmation sera réalisée en Langage C et utilisera **l'API des sockets POSIX**.

1. Comprendre l'application exemple

Le jeu ciblé est une version extrêmement simplifiée de la bataille navale. Le joueur joue contre l'ordinateur et doit découvrir au plus vite les coordonnées d'un trésor caché par l'ordinateur au sein d'un plateau de NxN cases. Lors d'une partie, le joueur doit trouver le trésor en essayant de minimiser le nombre total de points associé à chaque essai.

La figure ci-dessous illustre comment est réalisé le calcul des points pour un essai portant sur une case caractérisée par ses coordonnées. On suppose que la case colorée en vert contient le trésor. Chaque case du tableau indique le nombre de points attribués au joueur lorsqu'il jouera cette case. Plus il cible une case proche du trésor, moins il ramasse de points, et meilleur sera le résultat final de sa partie.

3	3	3	3	3	3	3
3	3	3	2	3	3	3
3	3	2	1	2	3	3
3	2	1	**0**	1	2	3
3	3	2	1	2	3	3
3	3	3	2	3	3	3
3	3	3	3	3	3	3

La fonction C qui réalise ce calcul de points vous est fournie.

Pour mieux visualiser le fonctionnement de l'application, vous pouvez lancer l'exécution du programme `test-jeu-centralise.c` qui est une implémentation non distribuée de ce jeu (ou `test-jeu-centralise-nc.c` qui est une version utilisant la bibliothèque `ncurses` pour l'affichage). Exemple :

```
$ make test-jeu-centralise
$ ./test-jeu-centralise
```

Remarque : par défaut le trésor est positionné dans la case de coordonnées (4,5) d'un plateau de 10x10. Un appel du programme avec le paramètre « `rand` » positionne le trésor dans un emplacement aléatoire.

2. Distribuer l'application

La version distribuée qu'il vous est demandé de développer doit s'appuyer sur une conception impliquant deux entités logicielles différentes :

- un programme à destination du joueur endossant un rôle de **processus applicatif client** ;
- un programme endossant un rôle de **processus applicatif serveur**, assurant la gestion des parties des joueurs.

Ainsi les processus client et serveur pourront être déployés et exécutés sur des machines différentes.

Les messages applicatifs échangés entre les deux types de processus seront transportés par TCP/IP et relèvent de l'une des deux catégories suivantes :

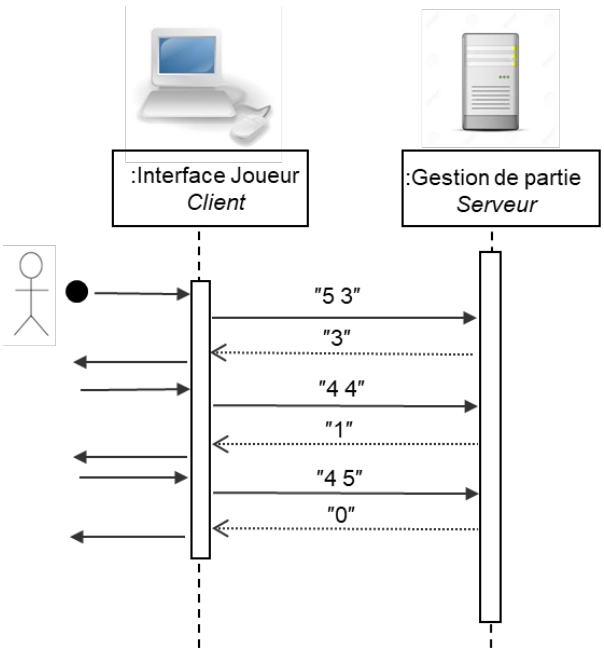
- un message équivalent à une **requête** est construit par le processus client, puis envoyé à destination du processus serveur ;
- un message équivalent à une **réponse** est construit par le processus serveur (en réponse à une précédente requête) puis envoyé à destination du processus client.

Habituellement, le format des messages Requêtes/Réponses est soumis à la spécification d'un **protocole applicatif**. Un tel protocole spécifie la structure de chaque message et la sémantique de chaque champ défini.

Dans notre cas, pour simplifier, nous adopterons une forme protocolaire minimaliste :

- un message requête correspond à **deux entiers** représentant les coordonnées de la case ciblée par le joueur lors d'un essai ;
- un message réponse correspond à **un entier** représentant le nombre de points obtenus à l'issue de l'évaluation de l'essai précédent.

La figure ci-dessous représente un scénario d'échange de messages applicatifs entre un processus client et un processus serveur s'exécutant sur deux machines distantes. Le scénario illustré est le cas d'une partie où le joueur trouve en 3 essais.



3. Concevoir et programmer la communication répartie

Le choix a été fait de développer cette application Client/Serveur au-dessus du protocole de transport **TCP**. (*Vous pouvez trouver un exemple de programmation au-dessus du protocole UDP dans le code fourni lors du TP précédent, code qui met en œuvre les échanges de paquets entre les routeurs*).
En conséquence, c'est le mode connecté qui doit structurer le schéma de communication entre un processus client et un processus serveur.
De façon générale, le schéma de communication en **mode connecté** est opéré selon 3 phases successives :

- Établissement de la connexion ;
- Transfert de données ;
- Terminaison de la connexion.

L'API des *sockets* en mode connecté (**`SOCK_STREAM`**) offre les primitives nécessaires pour programmer chacune de ces phases.

Une *socket* est l'abstraction d'une extrémité d'un canal de communication. Dans le cas de communications distantes, elle relève d'un protocole (TCP ou UDP) et est naturellement associée à un numéro de port ET à une adresse IP.

La primitive **`socket`** permet à un processus de créer sa propre extrémité sur un canal de communication. Le programmeur, grâce à la primitive **`bind`** peut, si nécessaire, associer de façon statique le numéro de port et l'adresse IP à une *socket*. En principe, dans le cas d'une *socket* associée à un processus serveur, au moins le numéro de port doit être assigné. Dans le cas où la primitive **`bind`** n'est pas programmée, lors de l'exécution, le système attribuera dynamiquement un numéro de port au processus.

a/ Mise en œuvre de la phase d'établissement d'une connexion TCP

Côté processus initiateur : l'ouverture est dite active. La primitive **`connect`** permet de programmer l'établissement d'une connexion TCP entre la *socket* locale et la *socket* distante dont les caractéristiques TCP/IP (numéro de port, adresse IP) sont fournies en argument.

Côté processus répondant : l'ouverture est dite passive. La primitive **`listen`** permet de programmer l'écoute de demandes de connexions entrantes portant sur la *socket* locale générale dédiée à cela, associée au numéro de port préalablement fixé par la primitive **`bind`**.
Ensuite, le programmeur peut invoquer la primitive **`accept`** sur cette *socket* locale. Cette primitive est bloquante. Lors de l'arrivée d'une demande d'établissement de connexion émanant d'un processus distant, une *socket* spécifique à cette connexion est créée localement et est retournée par la primitive **`accept`**. Cette *socket* ainsi créée sera considérée comme l'extrémité locale de la connexion TCP, connexion établie avec le processus initiateur distant, pour programmer la phase de transfert de données qui suivra.

Dans le modèle d'interaction Client/Serveur, c'est par principe le rôle de client qui est à l'initiative de l'interaction. Le client doit opérer une ouverture active de connexion alors que le serveur doit être programmé pour une ouverture passive.

b/ Mise en œuvre de la phase de transfert de données sur une connexion TCP établie

Quel que soit le rôle du processus, celui qui veut émettre ou recevoir des données le fera à partir de sa *socket* représentant l'extrémité locale de la connexion TCP/IP précédemment établie. Pour rappel, une connexion TCP est bi-directionnelle, dans le sens où elle supporte des échanges simultanés de deux flux d'octets (un flux du processus client vers le processus serveur, et un autre allant en sens inverse).

Pour une émission, un processus invoque la primitive **`send`**.
Pour une réception, c'est la primitive bloquante **`recv`** qui est invoquée.
(À noter que, respectivement, les primitives **`write`** et **`read`** peuvent être aussi utilisées)

Au-delà du transport effectif sur le réseau des données, une question importante concerne leur encodage. Dans le cas présent, les données échangées sont des entiers. Il existe deux possibilités pour encoder un entier :

- Sous forme d'une chaîne de caractères ;
- Sous forme d'une suite d'octets encodant directement la valeur de l'entier. Se pose alors la question du nombre d'octets utilisé pour représenter un entier et de leur ordre (problème connu sous le terme de « boutisme » (*endianness*) avec les conventions *Big-Endian* et *Little-Endian*).

Exemple d'encodage de l'entier 3 :

- Chaîne de caractères de deux octets (code ASCII de '3' et caractère null) : 0011 0011 0000 0000
- Entier de quatre octets en *big-endian* : 0000 0000 0000 0000 0000 0000 0000 0011

Pour ce TP nous choisirons d'encoder les coordonnées comme une chaîne de caractères. Ainsi, l'envoi des coordonnées 5 et 3 se traduira par l'envoi de la chaîne de caractères « `5 3` », c'est-à-dire, les 4 octets suivant : `'5'`, `' '`, `'3'`, `'\0'`.

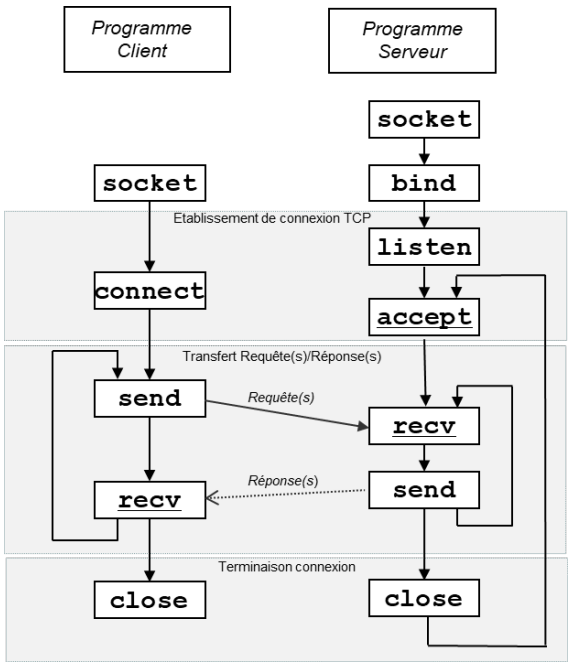
Remarque : pour opérer les opérations de changement de représentation d'entiers en chaîne de caractères et inversement, vous pourrez utiliser les fonctions **`sprintf`** et **`sscanf`** respectivement.

c/ Mise en œuvre de la phase de terminaison de connexion

La primitive **`close`** provoque la fermeture de l'extrémité de connexion associée à la *socket* sur laquelle elle est invoquée (notez que la primitive **`shutdown`** permet de gérer plus finement la terminaison de la connexion).

d/ Synthèse

La figure ci-dessous résume les invocations successives des primitives au sein de chacun des deux rôles Client et Serveur. Le fait que l'identifiant de la primitive soit souligné indique qu'il s'agit d'une primitive bloquante.



4. Programmer le rôle de Client

Le programme « Client » est à destination du joueur. À l'exécution, le processus client permet à l'utilisateur de jouer une partie. Tant que le joueur n'a pas trouvé le trésor, il soumet des essais au serveur sous forme d'une chaîne de caractères contenant les deux entiers représentant les coordonnées de la case du plateau qu'il cible (avec un espace entre les deux entiers). À l'issue de chaque coup, le serveur lui renvoie le nombre de points obtenus. Le processus doit se terminer lorsque la valeur 0 est retournée par le serveur puisque l'essai précédent est gagnant.

Ecrire le programme `client.c` qui implémente ce rôle. Pour la saisie des coordonnées et l'affichage du jeu, vous pouvez reprendre ce qui a été fait dans la version centralisée (`test-jeu-centralise.c`).

Pour tester votre programme vous pourrez utiliser un processus serveur déployé sur la machine `146.59.237.169` et écoutant les demandes de connexions entrantes sur le port TCP `5555`.

5. Programmer le rôle de Serveur en mode itératif

Le programme « Serveur » gère chaque partie d'un joueur particulier.
Pour chaque joueur (sous-entendu pour chaque processus client successif pour lequel il a accepté une connexion TCP), il initialise une partie, puis, pour chaque coup : reçoit les coordonnées, calcule les points obtenus en fonction de la case jouée et retourne le résultat au client. La gestion d'une partie se termine lorsque le joueur a trouvé le trésor. Le serveur va alors gérer la partie du joueur suivant.
Dans cette version dite itérative, au sein d'une boucle infinie, le processus serveur ne traite qu'un seul client à la fois.

Écrire le programme `serveur1.c` qui implémente ce rôle. Pour le calcul du résultat du coup du joueur, vous utiliserez la fonction `recherche_tresor(…)` du fichier `tresor.h`.

Vous testerez votre programme avec le client développé à l'étape précédente.

6. Programmer le rôle de Serveur en mode concurrent

La version précédente permet au processus serveur de ne traiter qu'un seul processus client (joueur) en même temps. La programmation d'une version dite concurrente permettrait de pouvoir traiter plusieurs clients en même temps. Une telle approche, reposant sur le potentiel des APIs de pseudo-parallélisme offerts par les systèmes d'exploitation et les langages de programmation, permet d'obtenir des processus serveurs capables de répondre à de nombreuses sollicitations simultanément.

Pour des questions de temps accordé à ce TP, nous adopterons une stratégie qui consiste en la création dynamique d'un processus fils dédié à chaque client pour lequel une connexion a été demandée et acceptée. Le principe d'héritage du contexte du processus père permettra à un processus fils créé dynamiquement, de pouvoir travailler sur la *socke*t créée par le processus père en retour du **`accept`**. Il pourra ainsi recevoir et émettre sur la connexion TCP établie spécifiquement avec le processus client distant.

Écrire le programme `serveur2.c` qui adopte une modalité d'exécution concurrente.

Vous testerez votre programme avec plusieurs clients lancés en parallèle.

7. Evaluation

Ce travail est à réaliser individuellement. **Vous disposez d'une séance de TP encadrée et d'une séance libre**. Vous devrez déposer sur moodle une archive de votre code source commenté.

A titre indicatif, le barème (sur 10) sera le suivant :

- Client : 4 pts (section 4) : 4 pts
- Serveur en mode itératif (section 5) : 4 pts
- Serveur en mode concurrent (section 6) : 2 pts

La qualité du code (lisibilité, commentaires, etc.) sera considérée sur 0,5 pt dans chacun des trois programmes attendus.