

Projet de Middleware Client/Serveur 2016/2017

Equipe : BENCHIHA A., LEFORT R., SALENGRO MA.

| | |
|--|-----------|
| Introduction | 1 |
| Manuel de 'Limite-Limite' | 2 |
| Mise en oeuvre | 4 |
| Utilisation de répertoires. | 4 |
| Format protocolaire de l'application | 4 |
| Etablissement d'une connexion (Client : socket() connect() / Serveur : socket(), bind(), listen(), accept()) | 4 |
| Transfert de données (Client : write() / Serveur : read()) | 4 |
| Fin de connexion(Client/Serveur : Closed) | 5 |
| Constantes | 5 |
| Configurations | 5 |
| Communication | 5 |
| Structure de données | 7 |
| Concernant les sockets | 7 |
| Concernant le jeu | 7 |
| Fonctions primitives | 8 |
| Concernant les sockets | 8 |
| Concernant le jeu | 8 |
| Côté serveur | 8 |
| Côté client | 9 |
| Module de test des fonctions primitives | 9 |
| Lancer l'application | 10 |
| Conclusion | 12 |

Introduction

Le but du projet est de mettre en oeuvre une application en C utilisant les sockets. C'est à dire permettre à des machines sur un réseau local ou un réseau distant de communiquer via des protocoles. Les différents protocoles sont le TCP qui est le mode connecté et UDP le mode non connecté.

Pour notre application, nous allons nous baser sur le jeu "Limite-Limite" permettant de jouer à plusieurs et donc approfondir nos connaissances sur l'utilisation des sockets via un projet. Lors de la phase de test, nous nous aiderons de la commande netstat -an pour surveiller l'activité au niveau du réseau.

Manuel de 'Limite-Limite'

1) **lancer l'application**

Exécuter le programme principal de l'application (sur terminal ?)

2) **rejoindre un salon**

Entrer l'adresse IP du serveur

3) **Début de la partie**

On initialise le jeu. Tous les joueurs reçoivent les cartes blanches.

4) **Désignation du joueur maître pour le tour**

Un des joueurs se retrouve maître pour ce tour. Il doit donc tirer une carte noire. C'est ce joueur qui élira la meilleure carte blanche.

5) **Tirage de la carte noire**

Le joueur maître tire la carte noire. Elle est révélée à tous les joueurs.

6) **Réflexion des joueurs**

Les autres joueurs ont 30 secondes afin de réfléchir à quelle carte blanche. Ils vont envoyées.

7) **Découvertes des cartes blanches**

Tous les joueurs découvrent les cartes blanches que chaque joueur a décidé de révéler pour ce tour.

8) **Vote du joueur maître**

Le joueur maître a alors une minute pour décider quelle carte donne le sens le plus drôle à la phrase présente sur la carte noire. Lorsque le joueur maître a choisi, le joueur qui a proposé cette carte gagne alors un point. Une nouvelle attribution du joueur maître sera effectuée avant que le nouveau tour commence.

9) **Début du tour suivant**

Le tour suivant débute, les joueurs reçoivent donc le nombre de cartes blanches qui leur manque.

10) **Tour suivant**

Le tour suivant s'effectue sur le même principe que le premier avec les mêmes procédures.

11) **Désignation du vainqueur**

Le vainqueur est désigné de la manière suivante: c'est le joueur qui aura obtenu le plus rapidement 10 points.

12) **Fin de partie**

Lors d'une fin de partie, les joueurs voient apparaître une "popup" avec le nom du joueur gagnant. Suite à cet écran, ils se retrouvent de nouveau dans le salon.

Mise en oeuvre

Utilisation de répertoires.

Un répertoire nommé **client** contenant tous les fichiers sources client

Un répertoire **config** contenant les fichiers de configuration du jeu

Un répertoire **script** permettant de nettoyer les fichier .o et exe.

Un répertoire **serveur** contenant les fichiers sources du serveur

Un répertoire **utils** contenant des fichiers pour la communication client/serveur

Cela impose de spécifier les répertoires dans le **makefile** et d'ajouter l'option **-I** au **gcc** pour trouver les fichiers d'en-tête.

A la **racine**, nous aurons le **server** et **joueur** ainsi que le **makefile**.

Format protocolaire de l'application

Chaque message informatiques sont encapsulés dans un format spécifique, appelé trame, avant d'être envoyé sur le réseau. La trame nous fournit l'adresse de destination ainsi que de l'hôte. Pour l'option de remise des messages, nous utilisons en majorité la monodiffusion et la diffusion. Soit on va communiquer à un seul hôte, soit à tous les hôtes.

Concernant les protocoles pour notre application utilisant les sockets, nous avons soit le TCP ou l'UDP. Nous utiliserons pour notre cas, le protocole TCP qui permet une communication fiable avec accusé de réception qui nous servira lors des phases de test.

Pour le protocole TCP, il n'y a pas de restriction sur la localisation des programmes et les 2 programmes peuvent se situer sur la même machine (127.0.0.1) Une connexion TCP est identifiée par 4 données :

Adresse IP sur programme 1

Port du programme 1

Adresse IP sur programme 2

Port du programme 2

Etablissement d'une connexion (Client : `socket()` `connect()` / Serveur : `socket()`, `bind()`, `listen()`, `accept()`)

Le client envoie un segment SYN (numéro de séquence) au serveur.

Le serveur lui répond par un segment SYN/ACK (numéro de séquence du client + serveur)

Le client confirme un segment ACK

Transfert de données (Client : `write()` / Serveur : `read()`)

Les numéros de séquence sont utilisés pour ordonner les segments TCP reçus et de détecter les données perdues. (Sommes de contrôle)

Fin de connexion(Client/Serveur : Closed)

On verra sur la commande netstat un Time_wait (client) et close_wait (serveur) avant la fermeture de la socket (Closed)

TCP côté client

1. socket
2. connect
3. send
4. recv
5. close

TCP côté serveur

1. socket
2. bind
3. listen
4. accept
5. send
6. recv
7. close

Constantes

Configurations

Nous aurons dans le répertoire config des constantes liées à la configuration du jeu et de communication.

Dans config.h, nous aurons un message de communication associé à un code. Nous définissons aussi les constantes liées au nombre max de joueur, à la taille des messages. Dans card.h, nous définissons des fonctions pour avoir le nom des cartes ainsi que les points. Par exemple :

```
char* get_card_white_name(int card);
```

Ce qui va nous retourner une liste de carte définit dans card.c

```
char* get_card_white_name(int card) {  
    return card_white_names[card];  
}
```

card_white_names est un tableau contenant les cartes du jeu

Communication

Dans le répertoire utils, nous aurons des fonctions de communication associées au code des constantes définies dans le répertoire config.

```
void broadcast_light( int msg_code, player* recipients, int rcp_count );
```

fonction définie dans server_utils.h qui sert de communication entre le serveur et le client.

| FONCTION CONCERNEE | CODE DU MESSAGE | | SOCKET | PSEUDO | CARTE DU JOUEUR | BOOL isempty | RCP_COUNT |
|--------------------------|-----------------|-------|--------|--------|-----------------|--------------|-----------|
| fonction broadcast light | 5 | | 4 | czsd | -1 | 0 | 2 |

Côté serveur, nous pouvons voir que l'on a un message. Par exemple dans ce cas là, du serveur :

```

if (jeu_en_cours) {
    if (fin_du_tour && !isempty) {
        fin_du_tour = FALSE;
        broadcast_light(ASK, joueurs, cl_count);
    }
}

```

nous envoyons un message au client avec le code 5 qui correspond à ASK.

Lorsque l'on décode le message ASK côté client,

`void receive_message(int client_socket, char** name)`

pour décoder côté client nous avons ces instructions :

```

} else if(msg_code == ASK) {
    print_cards();
    int choice = -1;
    if (cards_in_hand + cards_in_stash == 1) {
        printf("C'est votre dernière carte à jouer\n");
        send_light_msg(EMPTY, client_socket);
    }
    if (cards_in_hand == 0) {
        refill();
        printf("Vous n'avez plus de carte à jouer\n");
        print_cards();
    }
    int times = 0;
    do {
        if(times > 0) {
            printf("Vous devez jouer une carte\n");
            print_cards();
        }
        printf("Quelle carte voulez vous jouer ?\n");
        if (scanf("%d", &choice) == EOF) {
            //Ctrl+D
            disconnect(FALSE);
        }
        times++;
    } while(choice < 0 || choice > cards_in_hand);
    send_int_msg(PLAY, hand[choice], client_socket);
    int i;
    for (i = choice; i < cards_in_hand-1; i++) {
        hand[i] = hand[i+1];
    }
    cards_in_hand--;
    printf("Voici votre nouvelle main: ");
    print_cards();
    printf("\n");
} else if (msg_code == GIVE){

```

Nous pouvons voir que nous avons aussi des messages client vers le serveur avec des codes comme EMPTY avec la fonction

`void send_light_msg(int msg_code, int socket)`

définie dans `common_utils.h`

Structure de données

Concernant les sockets

```
struct sockaddr_in {
    uint8_t      sin_len;          /* Longueur totale */
    sa_family_t  sin_family;       /* famille : AF_INET */
    in_port_t    sin_port;        /* Le numéro de port */
    struct in_addr sin_addr;       /* L'adresse internet */
    unsigned char sin_zero[8];    /* un champ de 8 zéros */
};

struct sockaddr {
    unsigned char sa_len;          /* Longueur totale */
    sa_family_t  sa_family;       /* famille d'adresse */
    char         sa_data[14];     /* valeur de l'adresse */
};

struct hostent {
    char    *h_name;              /* Nom officiel de l'hôte. */
    char    **h_aliases;          /* Liste d'alias. */
    int     h_addrtype;           /* Type d'adresse de l'hôte. */
    int     h_length;             /* Longueur de l'adresse. */
    char    **h_addr_list;        /* Liste d'adresses. */
}

#define h_addr h_addr_list[0] /* pour compatibilité. */
};
```

Concernant le jeu

La structure que l'on va surtout définir, c'est celle du joueur.

```
typedef struct player {
    int socket;
    char nickname[NAMESIZE];
    int played_card;
    bool isempty;
} player;
```

Un joueur a une socket, un pseudo, joue une carte et un boolean défini config.h :

```
typedef int bool; //type boolean
#define TRUE 1
#define FALSE 0
```

pour vérifier s'il a encore des cartes en jeu.

Fonctions primitives

Concernant les sockets

```
int socket(int domain, int type, int protocol); /* Création d'une socket*/
int close(int fd); /* Fermeture de la socket*/
int send(int s, const void *msg, size_t len, int flags); /* Mode connecté. Envoie
sur la socket s les données pointées par msg pour une taille de len octets : renvoie le
nombre d'octet envoyés*/

int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr
*to, socklen_t tolen); /*Mode non connecté : renvoie le nombre d'octet envoyés*/
int recv(int s, void *buf, int len, unsigned int flags); /*Mode connecté. Elle
reçoit sur la socket s les données qu'elle stockera à l'endroit pointé par buf pour une
taille de len octet. : Renvoie le nombre d'octets reçus.*/

int recvfrom(int s, void *buf, int len, unsigned int flags struct sockaddr *from,
socklen_t *fromlen); /*Mode non connecté : Renvoie le nombre d'octets reçus.*/

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen); /*Lie une socket
avec un struct sockaddr. Lors de la création d'une socket, il n'y a pas d'adresse assignée.
bind affecte l'adresse de addr à la socket. : Renvoie 0 en cas de réussite. */

int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen); /* Connecte
la socket à l'adresse spécifié dans sockaddr : Renvoie 0 en cas de réussite.*/

int listen(int s, int backlog); /*Marque la socket pour accepter les demandes de
connexions entrantes en utilisant accept(). : Renvoie 0 en cas de réussite.*/

int accept(int sock, struct sockaddr *adresse, socklen_t *longueur); /*Accepte la
connexion d'une socket sock. La socket aura été lié avec un port avec la fonction bind().
l'argument adresse sera remplie avec les informations du client qui s'est connecté. Cette
fonction retourne une socket utilisée pour communiquer avec le client*/
```

Concernant le jeu

Côté serveur

Du côté serveur, nous aurons surtout des connexions au serveur au maximum de 4 clients. Nous allons vérifier s'il y au moins deux joueurs, il y aura des fonctions en cas de refus de connexions, lorsque le jeu est en cours, de distribution des cartes et de fin de jeu.

```
void init_server(int*, struct sockaddr_in*); //création et bin du socket
void alarm_handler(int); //alarm et time out
void interrupt_handler(int); //fermer le serveur au signal SIGINT
void shutdown_socket(int); //fermer une socket
void shutdown_server(); //fermer le serveur
void add_client(int, struct sockaddr_in*); //ajouter un client
void add_player(int); //confirmer la connexion et notifier le client
```

```
void remove_player(player*, int, bool); //retirer un joueur
void refuse_connection(int); //refuser un client de se connecter
bool receive_msg(char*, int); //gérer les messages
void clear_lobby(); //informer que le jeu est terminé
void add_nickname(int, char**); //enregistrer le pseudo
void start_game(); //commencer le jeu
void deal_cards(); //melanger le jeu et donner les cartes
void start_round(); //commencer un round
void receive_card(int, char**); //recevoir la carte jouer par le joueur
void end_round(int, char**); //terminer un round
void update_score(int, char**); //mettre à jour le score
void create_nicknames_shared_memory(char* nickname); //création d'un mémoire partagé
void end_game();
int find_index(player*, int);
```

Côté client

Du côté serveur, nous allons surtout avoir des fonctions pour afficher les cartes, interpréter les messages sur serveur et connecter le joueur au serveur.

```
void interrupt_handler(int);
void disconnect(bool information);
void refill();
void clear_cards();
void print_cards();
int calculate_score();
void receive_message(int clientSocket, char** name);
void createNickname(char *name);
void connectToServer(int *clientSocket, char* serverIP, struct hostent *he, struct sockaddr_in *serverAddress);
bool fdp_is_valid(int fdp);
```

Module de test des fonctions primitives

Concernant les test, lorsque l'on lance l'application, nous pouvons à toute instant voir ce qui est envoyé au serveur et ce que le client reçoit et comment il le reçoit. Bien que l'application ne réponde pas aux exigences définies dans le manuel du jeu, nous avons tous les test qui fonctionnent

Lancer l'application

commande **make** pour compiler l'application

```
abenchih@abenchih-X751LN: ~/Documents/MCS/Limite_Limite
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$ make
gcc -Wall -Wextra -c -o serveur/serveur.o serveur/serveur.c
gcc -Wall -Wextra -c -o utils/common_utils.o utils/common_utils.c
gcc -Wall -Wextra -c -o utils/serveur_utils.o utils/serveur_utils.c
gcc -Wall -Wextra -c -o config/cards.o config/cards.c
gcc -Wall -Wextra -c -o serveur/serveur.o utils/common_utils.o utils/serveur_utils.o config/cards.o -o serveur
gcc -Wall -Wextra -c -o client/client.o client/client.c
gcc -Wall -Wextra client/client.o utils/common_utils.o config/cards.o -o joueur
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$
```

Si l'on veut supprimer tous les fichiers .o ainsi que les exécutables, nous avons un script dans le répertoire script. Il suffit de lancer : **./script_clean**

Les exécutables sont à la racine :

```
abenchih@abenchih-X751LN: ~/Documents/MCS/Limite_Limite
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$ ls
client config joueur lib Makefile script serveur serveur_utils
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$
```

Nous lançons d'abord le serveur avec un argument obligatoirement qui correspond au nombre de round que l'on accepte avant de diffuser un score.

```
abenchih@abenchih-X751LN: ~/Documents/MCS/Limite_Limite
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$ ./server
Manque des arguments! Donnez le nombre de round
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$ ./server 1
```

Une fois le serveur lancé, il faut lancer au moins deux clients avec un argument qui correspond à l'adresse IP. (**./joueur <adresseIP>**). S'il n'y a pas deuxième client connecté au bout d'un certains temps, le client est déconnecté du serveur.

```
abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$ ./joueur 127.0.0.1
Entrez un pseudo (20 characters max):

abenchih@abenchih-X751LN:~/Documents/MCS/Limite_Limite$ ./joueur 127.0.0.1
Entrez un pseudo (20 characters max):
```

Nous devons définir deux pseudos pour les joueurs.

Une fois les pseudos définis, nous avons différents messages de test :

```
abenchihagabenchih@X751LN:~/Documents/MCS/Limite_Limite$ ./server 1
Fonction send_int_msg : MESSAGE CODE : 0, PAYLOAD : 10, SOCKET : 4
-----
FONCTION CONCERNEE | MESSAGE | PSEUDO | --- | --- | --- | --- | ---
-----
extract player nickname | joueur1 |
-----
fonction send_int_msg : MESSAGE CODE : 0, PAYLOAD : 10, SOCKET : 5
-----
FONCTION CONCERNEE | MESSAGE | PSEUDO | --- | --- | --- | --- | ---
-----
extract player nickname | joueur2 |
-----
```

Côté serveur nous prenons le nom des joueurs. Après un certains temps, nous avons la distribution des cartes aux clients ainsi que de la carte noire définie dans le serveur. (c'est encore en test)

```
-----
extract player nickname | joueur2 |
-----
Déclenchement d'un signal !
fonction send_msg : MESSAGE CODE : 13, PAYLOAD (BODY): CARTE SERVEUR OK, SOCKET : 4
La carte noire est : CARTE SERVEUR OK
fonction send_msg : MESSAGE CODE : 4, PAYLOAD (BODY): 37 22 32 4 25 44 16 40 15 7 19 51 9 21 38 47 24 6 35 34 36 11 29 48 5 1, SOCKET : 4
Cartes données :
37 22 32 4 25 44 16 40 15 7 19 51 9 21 38 47 24 6 35 34 36 11 29 48 5 1
fonction send_msg : MESSAGE CODE : 13, PAYLOAD (BODY): CARTE SERVEUR OK, SOCKET : 5
La carte noire est : CARTE SERVEUR OK
fonction send_msg : MESSAGE CODE : 4, PAYLOAD (BODY): 30 17 42 33 23 31 0 20 26 45 46 2 41 10 8 50 18 43 27 39 12 13 3 14 49 28, SOCKET : 5
Cartes données :
30 17 42 33 23 31 0 20 26 45 46 2 41 10 8 50 18 43 27 39 12 13 3 14 49 28
-----
FONCTION CONCERNEE | CODE DU MESSAGE | --- | SOCKET | PSEUDO | CARTE DU JOUEUR | BOOL isempty | RCP_COUNT
-----
fonction broadcast light | 5 | --- | 4 | joueur1 | -1 | 0 | 2
-----
```

Nous avons la carte CARTE SERVEUR OK définie et retransmise aux client

```
Adresse IP saisie : 127.0.0.1
Création de la socket réussie
Tour en cours, Attente de la réception des cartes
fonction send_msg : MESSAGE CODE : 2, PAYLOAD (BODY): joueur1,
: 3
Tour en cours, Attente de la réception des cartes
Carte noire :
C'est cette phrase à compléter SERVEUR OK :
Tour en cours, Attente de la réception des cartes
Début du round Voici vos cartes :
Tour en cours, Attente de la réception des cartes
Main courante de carte noire :
```

Les clients reçoivent aléatoirement des cartes blanches (nombre de 7)

Lorsqu'un client joue, on attend que tout le monde joue pour rejouer après et le gagnant aura dans **"Tours gagnés"** sa manche gagnée.

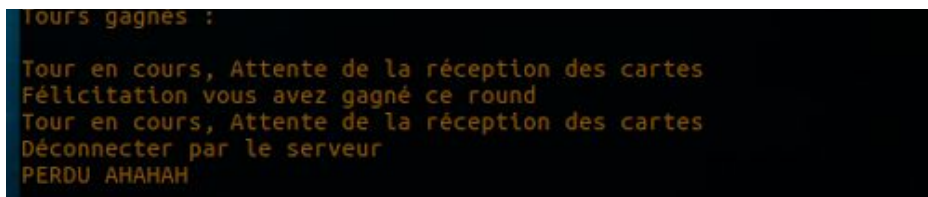
```
3: ♠ Phrase 17 ----- ♠
Tours gagnés : 0: ♦ Phrase 23 ----- ♦ | 1: ♣ Phrase 18 ----- ♣
Vous avez gagné ce tour
Tour en cours, Attente de la réception des cartes
Main courante de carte noire :
0: ♠ Phrase 20 ----- ♠
```

Si un client veut se déconnecter, il doit appuyer sur la touche **Ctrl+D**



```
Tours gagnés : 0: ♦ Phrase 23 _____ ♦ | 1: ♣ Phrase 18 _____ ♣
Quelle carte voulez vous jouer ?
Au revoir l'ami.
```

S'il reste encore un joueur connecté qui veut jouer, il sera automatiquement gagnant et sera déconnecté du serveur.



```
Tours gagnés :
Tour en cours, Attente de la réception des cartes
Félicitation vous avez gagné ce round
Tour en cours, Attente de la réception des cartes
Déconnecter par le serveur
PERDU AHAHAH
```

Conclusion

Ce projet de Middleware Client Serveur nous a permis d'approfondir nos connaissances en réseaux, surtout les interactions client/serveur. Ce projet nous a permis de réaliser un jeu fort sympathique qui pourra être joué à distance. Ce travail de groupe nous a aussi permis de nous enrichir en connaissances sur la gestion de projet.