Projet Réseau:
Gestion de comptes bancaires

Qianhui JIN - Liyun YANG
2020 - 2021

Sommaire

Introduction **3**

TCP **3**

Client 4

Initialisation 4

Communication avec le serveur 5

Serveur 5

Structure des données 5

Initialisation 6

Connection 6

Fonctions 7

UDP **10**

Client 11

Serveur 13

Fonction de communication du client et du serveur 15

Test du protocole UDP

Optimisation **20**

Conclusion **21**

I. Introduction

Le Réseau est un module d'enseignement de notre deuxième année du cycle ingénieurs en EISE (électronique, informatique et système embarqué). A la fin de ce module, nous avons un projet final que nous devons implémenter une application permettant de gérer des comptes bancaires en utilisant une architecture de type client-serveur (TCP et UDP) et codée en C.

Dans le système bancaire, il y a plusieurs clients. Pour chaque client, il peut avoir un ou plusieurs comptes. Le client peut verser ou retirer les argents sur son compte. Et il peut aussi regarder son solde et la date de dernière opération. En plus, il peut aussi contrôler ses 10 dernières opérations effectuées qui contiennent le type d'opération (retirer ou verser), la date d'opération et le montant qu'il a versé ou retiré.

II. TCP

TCP est un protocole de transfert fiable pour les données, il charge à contrôler de perte, de flux et de congestion en mode connecté. Mais son temps d' exécution est lent. Alors pour bien implémenter l'architecture TCP, on doit suivre le processus qui montre dans le figure 1.

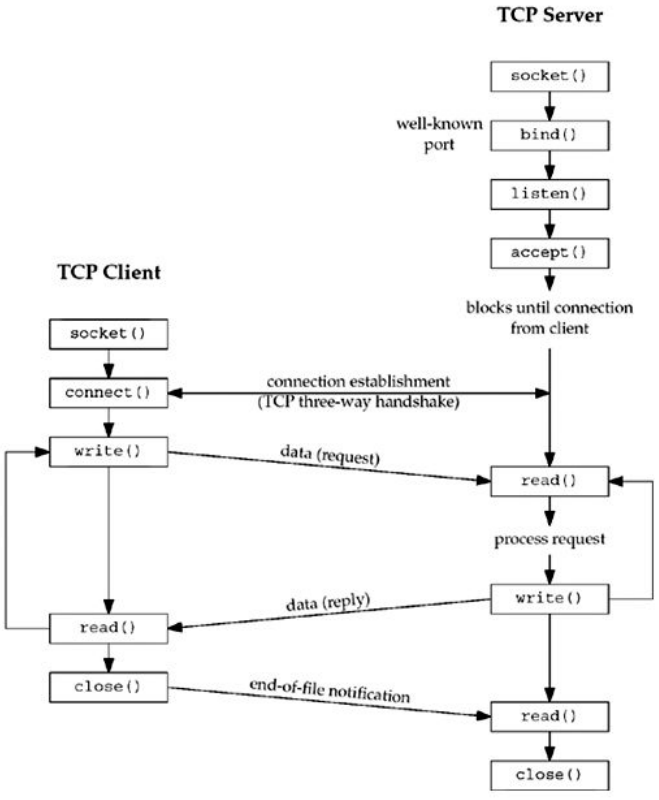

Figure 1: Le processus de la connexion entre le client et le serveur TCP

1. Client

La partie client pour le protocole TCP est écrite dans le fichier **TCPclient.c**.

a. Initialisation

Afin d'établir la connexion entre le client et le serveur, on a besoin d'un socket. Donc, pour l'initialisation, on va d'abord créer un socket et vérifier si on a bien réussi à créer ce socket. Puis on va initier la structure sockaddr_in pour le client (son port et adresse IP).

Voici les composants dans la structure sockaddr_in et notre code dans le figure 2 et 3.

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Figure 2: Les composants de la structure sockaddr_in

```
int sockfd;
struct sockaddr_in cliaddr;

// creation de socket et vérification
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1)
{
    printf("Echec de la création du socket\n");
    exit(0);
}
else
{
    printf("La création du Socket a réussi.\n");
}
bzero(&cliaddr, sizeof(cliaddr)); //initialisation

// distribuer IP, PORT
cliaddr.sin_family = AF_INET;
cliaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
cliaddr.sin_port = htons(PORT);
```

Figure 3: L'initialisation du socket

b. Communication avec le serveur

Le client doit pouvoir envoyer des messages au serveur et être capable de recevoir les réponses du serveur. Pour réaliser ces actions, le client doit connecter le serveur par la fonction *connect* dans la librairie socket. Une fois que le port de client prêt à écouter, on rentre dans dans la fonction *commu(int sockfd).* Il décrit une boucle infinie qui sert à envoyer et recevoir des messages par la fonction *read* et *write.* Après chaque envoie ou reçu, on doit réinitialiser les tableaux de message pour le futur. Et on utilise *strncmp* pour savoir si le client veut quitter, si oui, on sort de la boucle infini. Et à la fin on ferme le socket par *close.*

```
if ((strncmp(reponse, "Exit", 4)) == 0)
{
    printf("Client Exit...\n");
    break;
}
```

Figure 4: Vérification de l'intention de quittance du client

2. Serveur

La partie serveur pour le protocole TCP est codée dans le fichier **TCPserver.c**.

a. Structure des données

Dans la partie serveur, on définit une structure **CLIENT** qui contient un identifiant du client (**id_client)** et ses comptes décrits par un tableau **COMPTE.** Dans notre cas, on définit un client peut avoir au plus 10 comptes. Dans la structure de **COMPTE**, il y a un identifiant du compte (**id_compte)**, son mot de passe (**password)**, son **solde** et un tableau de **OPERATION** qui stock les 10 dernière opération effectuée par le client sous forme: **type, montant** et **date.** Pour obtenir la **date**, on utilise la structure *time_t* et la fonction *ctime.*

```
typedef struct
{
    int id_client;
    COMPTE compte[nbCompt];
}CLIENT;
```

```
typedef struct
{
    int id_compte;
    char *password;
    int solde;
    OPERATION operation[10];
}COMPTE;
```

```
typedef struct
{
    char *type;
    int montant;
    char *date;
}OPERATION;
```

Figure 5: Structure des données

b. Initialisation

Comme l'initialisation pour le client, on a fait la même chose pour le serveur qui peut se connecter avec tous les clients qui ont des différentes adresses IP mais le même port. A part cela, on a aussi fait l'initialisation du tableau de compte qui définit les 10 comptes et donne un mot de passe en commun "jin" et les soldes.

```
for (int i=0; i<nbCompt; i++)
{
    p[i].id_compte=i;
    p[i].password="jin";
    p[i].solde=i*10;
}
```

Figure 6: L'initialisation de CLIENT

c. Connection

Afin d'établir la connexion avec le client, on doit utiliser les fonctions *bind, listen* et *accept* de la librairie socket. Une fois la connexion établie, le serveur peut lire le message du client et le répondre dans la fonction *commu(int sockfd)* qui a des différences avec ce qui écrit dans la partie du client. Dans le boucle infini, après l'initialisation des tableaux des messages, on demande au client d'entrer son identifiant du client, son identifiant du compte, son mot de passe de pas à pas et le stocke dans le serveur en utilisant les fonctions *strcpy (*copie le message du serveur dans le tableau *reponse), write, read, sscanf (*capter le nombre en int dans le tableau de chaine de caractère: *demande)* et un boucle while qui ne capter que les lettres ou les entiers dans le tableau *demande* pour stocker le mot de passe. Comme ce que vous avez vu, dans notre cas, le mot de passe ne peut comporter que les entiers ou les lettres, mais pas les caractères spéciaux.

```
//capter le Id du client
strcpy(reponse,"Entré Id du client : ");
write(sockfd,reponse,sizeof(reponse));
bzero(reponse, MAX);  //initialiser le buff en
read(sockfd, demande, sizeof(demande)); // lir
sscanf(demande,"%d",&id_client);
printf("Client connecté %d\n",id_client);
bzero(demande, MAX);
```

```
int n=0;
while(isalpha(demande[n]) || isdigit(demande[n]))
{
    password[n]=demande[n];
    n++;
}
password[n]='\0';
bzero(demande, MAX);
```

Figure 7: Le processus pour lire et stocker les informations différentes

Après on va demander au client de saisir son opération qu'il veut effectuer (*Ajout, Retrait, Solde* ou *Operations)*, et on utilise *strncmp* pour capter sa demande. Si l'opération demandée est *Ajout* ou *Retrait,* on va demander ensuite le montant avec le processus dans le figure 7 et on effectue la fonction correspondante et obtenir un nombre *Res* qui sert à vérifier si l'opération s'est bien dérouler. Si *Res=-1*, la commande est en échec, on envoie au client un message "KO" en utilisant *strcpy, read* et write. Si *Res!=-1,* on envoie "OK". Si l'opération demandée est *Solde* ou *Operations*, on va effectuer directement la fonction correspondante. Si les opérations demandées ne correspondent pas aux 4

opérations qu'on a définies, on va dire au client "*Pas de opération correcte*". Avant la ferme du socket, on va demander si le client veut quitter en utilisant la même manipulation dans la partie du client montré dans le figure 4.

d. Fonctions

Dans la gestion des comptes, on peut effectuer 4 opérations qui correspondent aux fonctions: *ajout, retrait, solde et operations.*

Pour la fonction *ajout,* on va retourner un nombre entier *res.* Au début, on va définir *res=-1.* Puis on va vérifier si l'identifiant du client existe, si oui, on va vérifier si son identifiant du compte et le mot de passe sont corrects, si oui on va ajouter le montant qu'il verse dans son solde et mettre *res=0.* Ensuite, on fait d'abord un boucle while pour examiner la position de l'opération la plus récente. Si la position égale 10, c'est à dire le tableau d'opération est plein, donc on va décaler l'indice afin de supprimer le plus ancien opération et ajouter l'opération qu'on vient d'effectuer avec le type *"Ajout",* la date et le *montant=somme* dans le tableau de l'indice égale 9.

```
if (cli[k].id_client==client) //vérifier si ce client existe
{
    for (int i = 0; i<nbCompt; i++)
    {
        //vérifier s'il a un compte correspond et le mot de passe est correcte ou pas
        if(cli[k].compte[i].id_compte==compte && (strcmp(cli[k].compte[i].password,password)==0))
        {
            cli[k].compte[i].solde +=somme; //versement
            res = 0;
```

Figure 8: Vérification des informations du client

```
//Boucle pour chercher la position du dernier opérations
int j=0;
while (j<10 && cli[k].compte[i].operation[j].type !=NULL)
{
    j++;
}

if(j==10) //le tableau est plein
{
    //on supprime le plus ancien et ajouter le plus récente opération
    for (int x=0; x<9; x++)
    {
        cli[k].compte[i].operation[x]=cli[k].compte[i].operation[x+1];
    }
    cli[k].compte[i].operation[9].type = "Ajout";
    cli[k].compte[i].operation[9].montant = somme;
    cli[k].compte[i].operation[9].date = ctime(&rawtime);
}
else //le tableau d'opération n'est pas plein, on ajoute l'opération act
{
    cli[k].compte[i].operation[j].type = "Ajout";
    cli[k].compte[i].operation[j].montant = somme;
    cli[k].compte[i].operation[j].date = ctime(&rawtime);
}
```

Figure 9: L'enregistrement des 10 dernières opérations

Pour la fonction *retrait*, on a fait la même manipulation que la fonction *ajout,* la seule différence est qu'on diminue le montant qu'il retire dans son solde et mettre le type en *"Retrait".*

Voici les testes pour les deux fonctions en cas de réussi et en échec:

Figure 10: Cas réussi pour la commande Ajout

Figure 11: Cas d'échec pour la commande Ajout

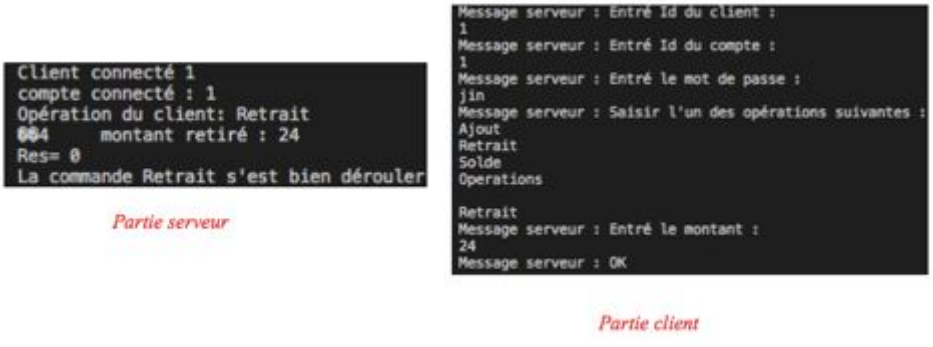

Figure 12: Cas réussi pour la commande Retrait

Figure 13: Cas d'échec pour la commande Retrait

Figure 14: Cas de opération fausse

Dans la fonction *solde,* on va d'abord vérifier si les informations du client sont correctes comme le figure 8. Puis on va utiliser le même moyen pour trouver la position de la dernière opération. Ensuite on va stocker le solde actuel dans le tableau de la chaîne de caractère *str*. Puis on stocke aussi les informations de la dernière opération dans *str.* A la fin, on les envoie au client par la fonction *write.*
Voici le test pour la fonction *solde*: le solde du client égale à 20€ au début, on fait un retrait de 2€, on demande l'affichage du solde actuel, il égale à 18€.

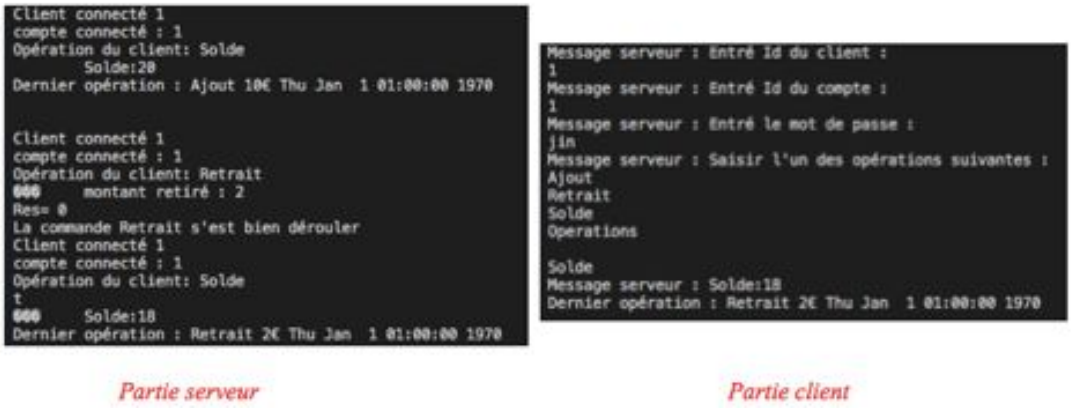

Figure 15: Le test pour la fonction solde

Pour la fonction *operations*, le début est comme le figure 8*,* on vérifie si les informations du client est correcte, si oui, on va faire un boucle *for* pour parcourir le tableau d'opération et un *if* pour vérifier si dans chaque position l'opération est vide. Sinon, on le stocke dans le tableau de la chaîne de caractères *str2,* et avant de passer à la position suivante, on va

utiliser la fonction *strcat* pour rassembler les informations du *str2* dans le *str*, un tableau de chaine de caractère qui a un taille assez grande. A la fin, on envoie ces informations des opérations qui de l'ordre le plus récent jusqu'à le plus ancien au client.
Voici les tests pour la fonction *operations:* on effectue d'abord 10 opérations et on fait l'appel de la fonction *operations,* on obtient bien les informations des 10 dernières opérations. Puis on fait une nouvelle opération, on rappelle la fonction *operations,* on obtient l'information du nouveau opération et le plus ancien est éliminé.

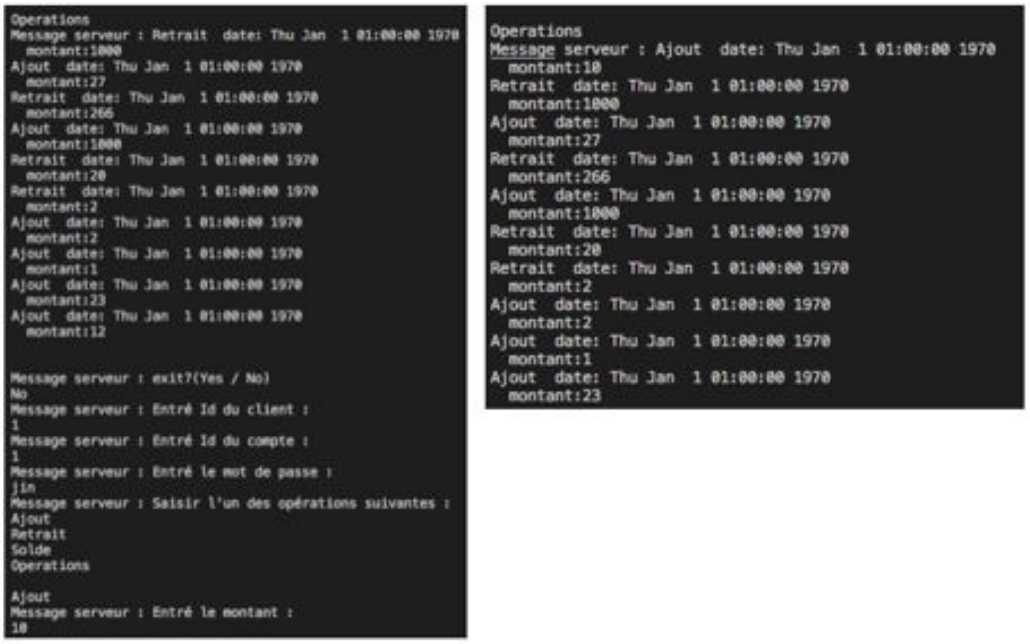

Figure 16: Le test pour la fonction operations

Or la communication dans la partie du serveur commence par *write,* afin de bien dérouler la communication entre le client et le serveur, on doit suivre la boucle de ordre *write(serveur)-read(client)-write(client)-read(serveur)* comme le figure 1 a montré. Donc après chaque *write,* on doit ajouter un fonction *read.*

III. UDP

Contrairement au protocole TCP, le protocole UDP est plus rapide, car il n'a pas besoin d' établir une connexion entre le client et le serveur. Donc il n'est pas fiable. En plus il ne vérifie que des erreurs. Alors pour bien implémenter l'architecture UDP, on doit suivre le processus qui montre dans le figure 17.

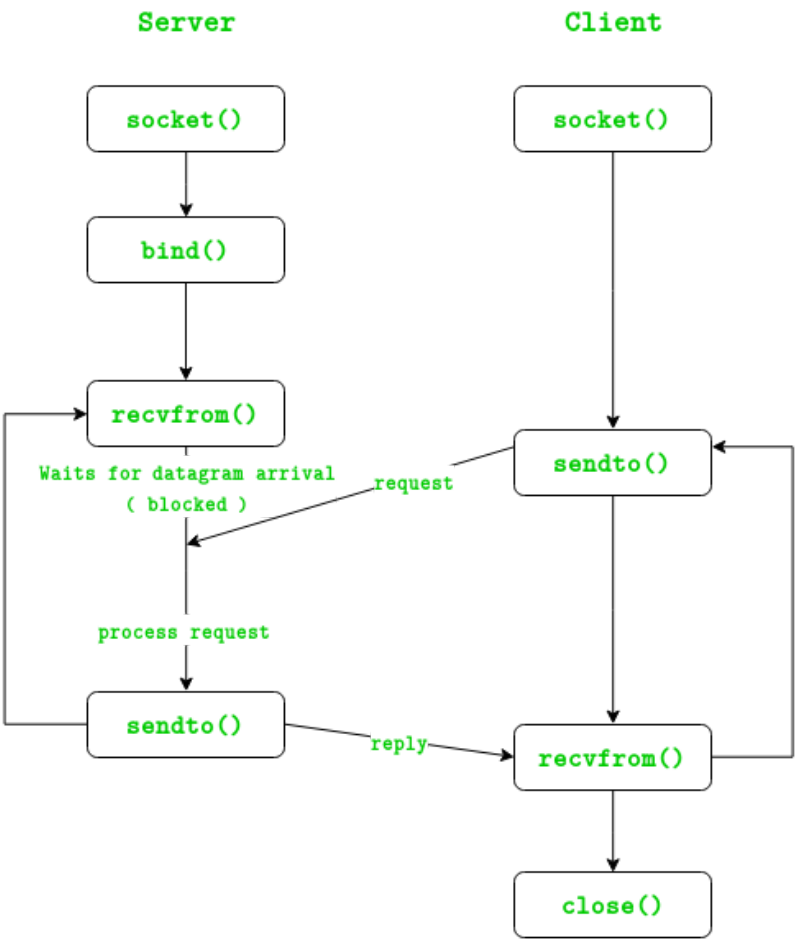

Figure 17: Le processus de la connexion entre le client et le serveur UDP

1. Client

a. Processus principale du côté Client

Or au protocole UDP, Client n'a pas besoin de demander la connexion au Serveur. Dans ce cas là, il faut tout simple de suivre le processus ci-dessous:

1. Créer le socket UDP
2. Envoyer un message au Serveur
3. Attendre la réponse de Serveur s'il a bien reçu le message de Client
4. Répondre au serveur, si nécessaire, retour à l'étape 2 pour envoyer encore de message
5. Fermer la description de socket et sortir ce processus.

b. Fonctions nécessaires pour Client

Pour établir le processus en dessus, il faut avoir des fonctions nécessaires:

1. La fonction de socket, le prototype de cette fonction est:

```
int socket(int domain, int type, int protocol)
```

domain - la communication spécifiée, soit AF_INET for IPv4, soit AF_INET6 for IPv6
Nous avons mis AF_INET pour cette variable.
type - le type de socket à créer, soit SOCK_STREAM pour le protocole TCP soit SOCK_DGRAM for UDP, évidemment on met SOCK_DGRAM pour cette variable.
protocol - le protocole à utiliser pour ce socket, on met '0' pour cette variable, c'est-à-dire que l'on utilise le protocole par défaut pour l'adresse familiale.

2. La fonction d'envoyer un message depuis un socket

```
ssize_t sendto(int sockfd,const void *buf,size_t len, int flags,
            const struct sockaddr *dest_addr, socklen_t addrlen)
```

sockfd - le socket que l'on avait créé par la fonction socket()
***Buf** - l'application Buffer à envoyer, qui contient les datagrammes
len - la taille de l'application Buffer
flags - à modifier l'action de socket on utilise "MSG_CONFIRM" pour ce paramètre
dest_addr - l'adresse de la destination
addrlen - la taille de dest_addr

3. La fonction de la réception d'un message depuis un socket

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen)
```

sockfd - le socket que l'on avait créé par la fonction socket()
***Buf** - l'application Buffer à reçevoir, qui contient les datagrammes de client
len - la taille de l'application Buffer
flags - à modifier l'action de socket on utilise "MSG_WAITALL" pour ce paramètre
src_addr - l'adresse de la source
addrlen - la taille de src_addr

4. La fonction de fermer le socket

```
int close(int fd)
```

fd - le file de description, ici, on prend sockfd: `close(sockfd)` pour fermer le socket

5. Une fonction de la communication que l'on a créé

```
void commu(int sockfd)
```

sockfd - le socket

2. Serveur

a. Processus principal du côté Serveur

Le processus de Serveur est aussi facile:

1. Créer le socket UDP
2. Aveugler le socket créé à l'adresse de Serveur
3. Attendre le trame de datagramme de client qui arrive
4. Régler le trame et envoyer une réponse au Client
5. Retourne à l'étape 3

b. Fonctions nécessaires pour Serveur

Dans le côté de Serveur, il contient aussi les fonctions du côté de Client. De plus, il contient des fonctions pour traiter la demande du client.

b.1) Les structures nécessaires

D'abord, on définit la structure de l' opérations, de compte et de client et de l'ID de client dans le serveur, comme on avait fait dans TCP.

```
typedef struct
{
    char *type;
    int montant;
    char *date;
}OPERATION;
```

```
typedef struct
{
    int id_compte;
    char *password;
    int solde;
    OPERATION operation[10];
}COMPTE;
```

```
typedef struct
{
    int id_client;
    COMPTE compte[10];
}CLIENT;
```

Figure 18: Les structures de client

Ces structures permettent de récupérer facilement les informations du client.

Puis, on initialise 3 clients (nbCli = 3) leurs ID est 1, 2 et 3. Chaque client contient 10 comptes, l'ID de comptes est de 100 à 109. Le code de chaque compte est "eise", le montant de chaque compte est de 10 à 90 par défaut.

```
// Initialisation des paramètres de client
COMPTE p[10];
CLIENT c[nbCli];
time_t rawtime;
```

```
/*
    Initialisation ID de client, ID de compte, password et solde

*/
for (int i=0; i<10; i++)
{
    p[i].id_compte=100+i;  // IDs de compte sont: 100, 101, 102...
    p[i].password="eise";   // password par defaut : "eise"
    p[i].solde=i*10;    // solde par defaut : 10,20,30...
}
for (int j=0; j<nbCli; j++)
{
    c[j].id_client = j;
    for(int k = 0; k<10; k++)
    {
        c[j].compte[k]=p[k]; // les infos de ce client prend les infos de compte p
            //que on vient d'initialiser en dessus
    }
}
```

Figure 19: L'initialisation du compte de client

b.2) Les fonctions traitantes de la demande du client

Comme dans le protocole TCP, on utilise les mêmes fonction de traitements:

```
int ajout(int client, int compte, char *password, int somme)
int retrait(int client, int compte, char *password, int somme)
void solde(int client, int compte, char *password, int sockfd)
void operations(int client, int compte, char *password,int sockfd)
```

Par contre, on modifie la fonction de réponse de TCP par la fonction de réponse de UDP qui sera utilisé dans la fonction de `solde()` et de `operations()`:

```
void solde(int client, int compte, char *password, int sockfd)
{
    char str[MAXLINE];
    for(int k = 0; k <nbCli; k++)
    {
        if(c[k].id_client==client) //vérifier si ce client existe
        {
        for (int i = 0; i<10; i++)
        {
            //vérifier s'il a un compte correspond et le mot de passe est correcte ou pas
            if(c[k].compte[i].id_compte==compte && (strcmp(c[k].compte[i].password,password)==0))
            {
                int cpt=0;
                //chercher le plus récent opération et ne pas dépasser la taille du tableau
                while (c[k].compte[i].operation[cpt].type !=NULL && cpt<10)
                {
                    cpt++;
                }
                //stoker le solde actuel dans le chaine str
                sprintf(str, "RES_SOLDE:%d\n", c[k].compte[i].solde);

                //si le tableau n'est pas vide, stoker la dernière opération dans le chaine str2
                if (cpt!=0){
                    char str2[MAXLINE];
                    sprintf(str2,"Dernier opération : %s %d€ %s\n",c[k].compte[i].operation[cpt-1].type,
                            c[k].compte[i].operation[cpt-1].montant,
                            c[k].compte[i].operation[cpt-1].date);
                    strcat(str, str2); //concaténer les 2 chaines
                }

                 //afficher le solde actuel et la dernière opération
                sendto(sockfd, str, sizeof(str), MSG_CONFIRM,
                 (const struct sockaddr *) &cliaddr,
                 sizeof(servaddr));
            //printf("\n********SOLDE******\n%s",str);
            }
        }
        }
    }
}
```

Figure 20: La fonction de l'opération `solde()` avec la fonction de la réponse de UDP

```
void operations(int client, int compte, char *password,int sockfd)
{
    char str[MAXLINE];
    for (int k = 0; k< nbCli; k++)
    {
        if(c[k].id_client==client) //vérifier si ce client existe
        {
        for (int i = 0; i<10; i++)
        {
            //vérifier s'il a un compte correspond et le mot de passe est correcte ou pas
            if(c[k].compte[i].id_compte==compte && (strcmp(c[k].compte[i].password,password)==0))
            {
                // afficher la date, le montant et le type des opérations
                // en ordre le plus récent jusqu'à le plus ancien
                for (int j=9; j>=0; j--)
                {
                    if (c[k].compte[i].operation[j].type != NULL)
                    {
                        char str2[256];
                        sprintf(str2,"%s - date: %s - montant:%d\n",c[k].compte[i].operation[j].type,
                                                                    c[k].compte[i].operation[j].date,
                                                                    c[k].compte[i].operation[j].montant);
                        strcat(str, str2);
                    }
                }
                sendto(sockfd, str, sizeof(str),
                MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
                sizeof(servaddr));
                //printf("\n*****OPERATION LISTE*****\n%s",str);
            }
        }
        }
    }
}
```

Figure 21: La fonction de l'opération `operations()` avec la fonction de la réponse de UDP

3. Fonction de communication du Client et du Serveur

a) La fonction commu() du client

```
void commu(int sockfd)
{
    int n;
    char buffer[MAXLINE];
    char demande_client[MAXLINE];
    int res = 0;
    for (;;) {
        n = 0;
        // mise en zero la demande de client et la réponse de serveur
        bzero(demande_client,MAXLINE);
    bzero(buffer,MAXLINE);
    printf("\n\t*****Envoyer un datagram******\n");
    while ((demande_client[n++]= getchar()) != '\n');
    demande_client[n]='\0';
    sendto(sockfd, demande_client, sizeof(demande_client), MSG_CONFIRM,
        (const struct sockaddr *) &servaddr, sizeof(servaddr));
    //printf("\n*****TEST*****Message sent from Client : %s\n", demande_client);
    //printf("****Test*********");
    recvfrom(sockfd, buffer, MAXLINE, MSG_WAITALL,
            (struct sockaddr *) &servaddr, NULL);
    printf("\nServer : %s\n", buffer);
    if(strncmp("EXIT",buffer,4) == 0){
        printf("___AU REVOIR___");
        break;
    }
    }
}
```

Figure 22: La fonction de l'opération `operations()` avec la fonction de la réponse de UDP

Cette fonction permet au client d'envoyer la demande et de recevoir la réponse du serveur.

Dans un premier temps, on crée deux variables `char buffer[MAXLINE]` et `char demande_client[MAXLINE]` qui correspondent la réponse du serveur et la demande du client.

Dans une boucle infinie, on met en zéro ces des variables par la fonction `bzero()`. on récupère la demande du client dans `demande_client` par `while((demande_client[n++]= getchar()) != '\n')`, le client doit saisir leur demande dans le terminal.

Puis on envoie la demande du client par la fonction `sendto()`, on récupère la réponse du serveur par la fonction `recvfrom()` et on l'affiche dans le côté du client :

```
sendto(sockfd, demande_client, sizeof(demande_client),MSG_CONFIRM,
        (const struct sockaddr *) &servaddr, sizeof(servaddr));
recvfrom(sockfd, buffer, MAXLINE, MSG_WAITALL,
        (struct sockaddr *) &servaddr, NULL);
printf("\nServer : %s\n", buffer);
```

b) La fonction commu() du serveur

La fonction `commu()` de serveur contient deux partie:

- l'une partie est comme la fonction commu() du client: créer les variables de buffer (la demande du client) et la réponse, mettre en zéro ces deux variables, recevoir la demande du client par la fonction recvfrom() et l'affiche dans le terminal:
- l'autre partie est de traiter la demande du client, et donne la réponse correcte.

b.1)Traitement de l'opération du client

Dans un premier temps, on créer les variables suivantes:

- **int id_client**: récupère la partie de l'ID du client dans la demande
- **int id_compte**: récupère la partie de l'ID de compte du client dans la demande
- **char password[10]**: récupère le code saisie du client
- **int somme**: récupère le montant d'ajout ou le montant de retrait
- **char oper[10]**: récupère l'opération saisie par client (soit Ajout, Retrait, Solde, Operation, Exit, Yes ou No)

On considère qu'il y a 3 types de demande (datagramme de client), ces 3 types sont différenciés par les différents types d'opérations : AJOUT, RETRAIT, SOLDE, OPERATION, EXIT, Yes et No. AJOUT et RETRAIT ont la même forme de datagramme, SOLDE et OPERATION ont la même forme, EXIT Yes et No ont une autre forme de datagramme. Alors ces 3 types de datagramme seraient comme les formes au-dessous:

- DEMANDE <id_client id_compte password somme>
  (DEMANDE = AJOUT/RETRAIT)
- DEMANDE <id_client id_compte password >
  (DEMANDE = SOLDE/OPERATION)
- EXIT/Yes/No

On initialise ces variables à 0 dans la boucle infinie `for(;;)`. On met à jour ces variables avec les données correspondantes par la fonction `sscanf()`.

```
//on récupérer les infos de la demande
int total_demande;
total_demande = sscanf(buffer,"%s%d%d%s%d",oper,&id_client,&id_compte,password,&somme);
printf("\nTotal Info: %d ",total_demande);
printf("\nOpération demandée de client: %s ",oper);
printf("\nID_client: %d ",id_client);
printf("\nID_compte: %d ",id_compte);
printf("\nPasseword: %s ",password);
printf("\nMontant : %d ",somme);
```

Figure 23: Récupérer de la demande du client dans les variables

`total_demande` est la longueur de datagramme de la demande, à partir de cette variable, on peut traiter différemment la demande du client.

Si `total_demande` est égale à 5, la demande est donc soit AJOUT soit RETRAIT, puis on compare `oper` avec `Ajout` et `Retrait`. par la fonction `strncmp()`. Si `oper` est l'un des ces opérations, on utilise ensuite la fonction correspondante:

- `ajout(id_client, id_compte,password,somme)`
- `retrait(id_client, id_compte,password,somme)`

On utilise une variables `int res` pour récupérer le résultat de ces deux fonctions, si ces fonctions marchent bien, `res` vaut 1, on met la variable `reponse` à `OK`, si `res = -1`, on met la `reponse` à `KO`, puis serveur envoie cette réponse par `sendto()` sous forme de:

```
sendto(sockfd, reponse, sizeof(reponse), MSG_CONFIRM,
      (const struct sockaddr *) &cliaddr, sizeof(servaddr));
```

Si la longueur de datagramme est 5, mais client n'a pas saisi correctement l'opération, c'est-à-dire, client a saisie `Ajou, Ajous, Retai, Rtrait.` On met directement la variable `reponse` à `KO Votre operation n'est pas correcte`, et le serveur envoie cette réponse au client par la fonction `sendto()`.

On fait le même processus pour la longueur de datagramme à 4 (`oper` devait être `Solde` ou `Operation`).

Particulièrement pour `oper` est `Exit, YES` ou `No`. Si oper est `Exit`, serveur demande la vérification de client par l'envoie d'une réponse de `Vous voulez quitter la session? [Yes / No]`, serveur à attendre la vérification du client.

Si client dit `Yes`, on affiche "`Client Exit ...`" dans le terminal du serveur, et donne la réponse `EXIT` au client, pour dire que le serveur sera fermé, puis on utilise `break` à fermer la session de serveur.

Si client dit `No`, on donne directement la réponse "`CONTINUE`" au client, et attend la nouvelle demande du client.

Si client n'a pas saisie correctement, on donne la réponse "`KO Votre demande n'est pas correcte`" au client.

Si `total_demande` est ni 1, ni 4, ni 5, on donne la réponse "`KO IL MANQUE/IL Y A TROPE DES INFORMATIONS`"

Cette partie du traitement la demande du Client est dans la figure 24 et 25 et 26.

```c
//on récupérer les infos de la demande
int total_demande;
total_demande = sscanf(buffer,"%s%d%d%s%d",oper,&id_client,&id_compte,password,&somme);
printf("\nTotal Info: %d ",total_demande);
printf("\nOpération demandée de client: %s ",oper);
printf("\nID_client: %d ",id_client);
printf("\nID_compte: %d ",id_compte);
printf("\nPasseword: %s ",password);
printf("\nMontant : %d ",somme);
if (total_demande == 5){
    if(strncmp("Ajout",oper,5) == 0){
        int res;
        res = ajout(id_client, id_compte,password,somme);
        if(res == -1){
            strcpy(reponse, "KO");
        }else{
            strcpy(reponse, "OK");
        }
        //printf("\n\t*****TEST*****Message sent of AJOUT.\n");
        // Envoyer la réponse de serveur
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
    }
    else if(strncmp("Retrait",oper,7) == 0){
        int res;
        res = retrait(id_client, id_compte,password,somme);
        if(res == -1){
            strcpy(reponse, "KO");
        }else{strcpy(reponse, "OK");}
        //printf("\n\t*****TEST*****Message sent of RETRAIT.\n");
        // Envoyer la réponse de serveur
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
    } // si operation saisie n'est pas correcte
    else{
        strcpy(reponse, "KO Votre operation n'est pas correcte");
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
    }
```

Figure 23: Traitement la demande du client (Partie 1)

```c
}else if (total_demande == 4){
    if(strncmp("Solde",oper,5) == 0){
        solde(id_client,id_compte,password,sockfd);
        //printf("\n\t*****TEST*****Message sent of SOLDE.\n");
    }
    else if(strncmp("Operation",oper,10) == 0){
        operations(id_client,id_compte,password,sockfd);
        //printf("\n\t*****TEST*****Message sent of OPERATION.\n");
    }
    else{
        strcpy(reponse, "KO Votre operation n'est pas correcte");
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
    }
```

Figure 24: Traitement la demande du client (Partie 2)

```
}else if(total_demande == 1){
    if(strncmp("Yes",oper,3) == 0){
        printf("Client Exit...\n");
        strcpy(reponse, "EXIT");
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
        break;
    }else if(strncmp("No",oper,2) == 0){
        strcpy(reponse, "CONTINUE");
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
    }else if(strncmp("Exit",oper,4)==0){
        strcpy(reponse, "Vous voulez quitter la session? [Yes / No]");
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
    }else{
        strcpy(reponse, "KO Votre demande n'est pas correcte");
        sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
}else{
    strcpy(reponse, "KO IL MANQUE/IL Y A TROPE DES INFOMATIONS");
    sendto(sockfd, reponse, sizeof(reponse),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
            sizeof(servaddr));
}
```

Figure 24: Traitement la demande du client (Partie 3)

4. Test du protocole UDP

Voici les tests de la communication par le protocole UDP:
Une communication normale:

```
jk@jk-virtual-machine:~/Bureau/Reseau/Projet_R-seau_UDP-main$ ./client

       *****Envoyer un datagram******
Ajout 1 100 eise 1000

Server : OK

       *****Envoyer un datagram******
Retrait 1 100 eise 500

Server : OK

       *****Envoyer un datagram******
Solde 1 100 eise

Server : RES_SOLDE:500
Dernier opération : Retrait 500€ Thu Jan  1 01:00:00 1970



       *****Envoyer un datagram******
Operation 1 100 eise

Server : Dernier opération : Retrait 500€ Thu Jan  1 01:00:00 1970

Retrait - date: Thu Jan  1 01:00:00 1970
 - montant:500
Ajout - date: Thu Jan  1 01:00:00 1970
 - montant:1000


        *****Envoyer un datagram******
Exit

Server : Vous voulez quitter la session? [Yes / No]

        *****Envoyer un datagram******
Yes

Server : EXIT
jk@jk-virtual-machine:~/Bureau/Reseau/Projet_R-seau_UDP-main$
```

```
jk@jk-virtual-machine:~/Bureau/Reseau/P

Client : Ajout 100 100 eise 100


Total Info: 5
Opération demandée de client: Ajout
ID_client: 100
ID_compte: 100
Passeword: eise
Montant : 100
Client : Ajout 1 100 eise 1000


Total Info: 5
Opération demandée de client: Ajout
ID_client: 1
ID_compte: 100
Passeword: eise
Montant : 1000
Client : Retrait 1 100 eise 500


Total Info: 5
Opération demandée de client: Retrait
ID_client: 1
ID_compte: 100
Passeword: eise
Montant : 500
Client : Solde 1 100 eise


Total Info: 4
Opération demandée de client: Solde
ID_client: 1
ID_compte: 100
Passeword: eise
Montant : 0
```

```
Total Info: 4
Opération demandée de client: Operation
ID_client: 1
ID_compte: 100
Passeword: eise
Montant : 0
Client : Exit


Total Info: 1
Opération demandée de client: Exit
ID_client: 0
ID_compte: 0
Passeword:
Montant : 0
Client : Yes


Total Info: 1
Opération demandée de client: Yes
ID_client: 0
ID_compte: 0
Passeword:
Montant : 0 Client Exit...
jk@jk-virtual-machine:~/Bureau/Reseau/Pr
```

Figure 25: Communication normale

Communication anormale:

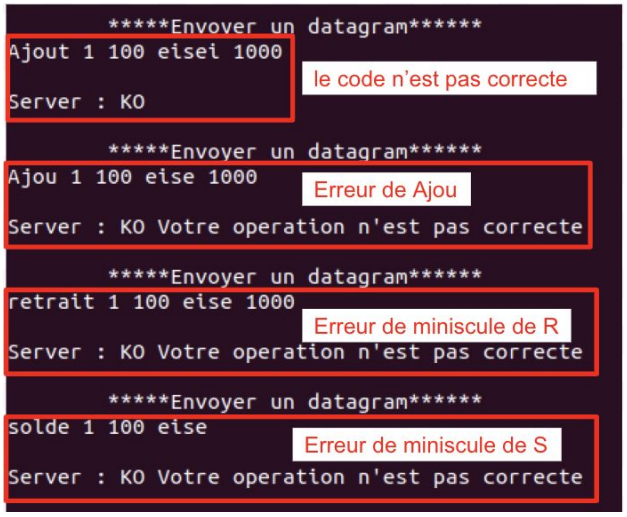

Figure 25: Communication normale à cause de l'opération

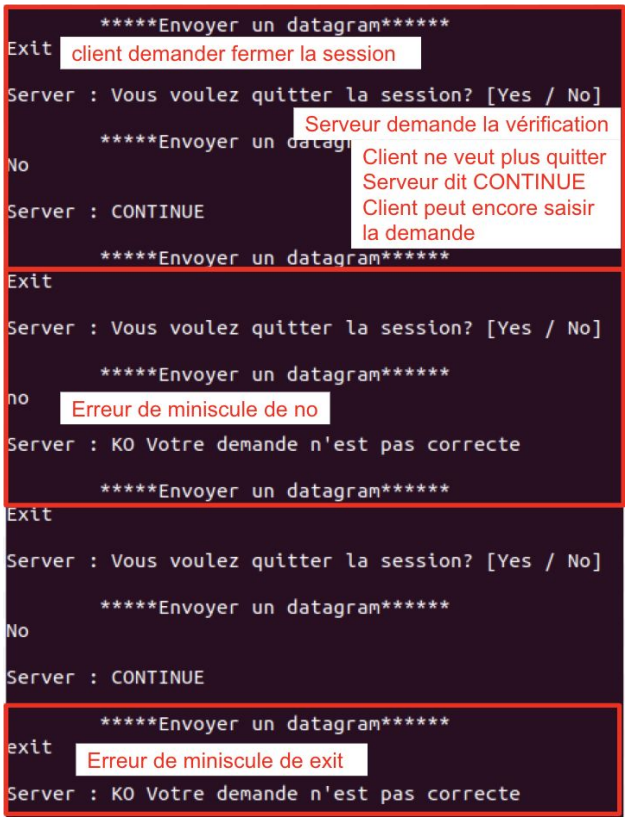

Figure 26 : Vérification de EXIT

IV. Optimisation

Pour gérer plusieurs clients qui ont un ou plusieurs comptes, on ajoute un tableau de **CLIENT** de taille *nbCli*, puis on le initie par un boucle *for*.

```
for(int j=0; j<nbCli; j++)
{
    cli[j].id_client=j;
    for(int k=0; k<nbCompt; k++)
    {
        cli[j].compte[k]=p[k];
    }
}
```

Figure 18: L'initialisation du tableau de CLIENT

Et on ajoute un boucle *for* avant la vérification de l'identifiant du client pour chaque fonction.

```
for(int k=0; k<nbCli; k++)
{
    if (cli[k].id_client==client) //vérifier si ce client existe
    {
```

Figure 19: Parcours du tableau CLIENT

En effet, dans notre cas, comme le figure 16 a montré, le temps des opérations est reste la même, donc on doit revoir notre code pour enregistrer l'horaire d'opération. En plus, on pourra améliorer notre code en entrant une seule fois l'identifiant et le mot de passe pour toutes les opérations d'un même client.

V. Conclusion

Dans ce projet, on a appris comment créer le socket et établir la connexion entre le serveur et le client à partir de ce socket. En plus, on a bien compris la différence entre l'architecture du protocole TCP et UDP. On trouve que le protocole UDP est plus facile et plus rapide à utiliser, mais il faut renvoyer la trame si le datagramme contient des erreurs. Par contre, TCP peut garantir l'envoie de datagramme par la connexion de serveur-client.