Rapport du projet de EL-3032

Savio BOISSINOT 2e
Martin DEGUEURCE 2e

Plan du rapport:

Partie 1 – Client.c :

Ici notre fichier client.c sert principalement d'IHM, elle permet de demander à l'utilisateur quel menu il désire. Une fois que le client a formulé son souhait, client.c écrit ,sur son pipe nommé attribué, l'entrée du client qui a pour forme *aaaa*|*bbbb*|*cccc* où ici *aaaa* correspond au code serveur, *bbbb* correspond au code faisant référence au restaurant et *cccc* correspond au code de la carte.

```
vboxuser@Debian:~/Documents/Restaurant/bin$ ./main_client.out
Pipe successfully opened
_____________________________
|      Project EL-3032      |
|      Savio BOISSINOT      |
|      Martin DEGUEURCE     |
|___________________________|


Welcome to the restaurant menu management application
Please select an option:
Press 1 to View the menu
Press any other key to leave the application

Your choice:
```

Screenshot de l'IHM

Au lancement du programme client.c récupérera tout d'abord dans le fichier value.txt le nombre maximum de serveur et d'utilisateur. Cela sera utile pour savoir jusqu'à quel pipe le client pourra aller chercher et s'il ne trouve pas de pipe libre jusqu'au maximum de client possible alors ça veut dire qu'il n'y as pas de place disponible pour lui, le programme se fermera alors.

Lorsque le client trouvera un pipe de libre alors il le renommera afin de la marquer comme utilisé.

```
result = initialise_pipe( self: &local_client_pipe, name_1: name_pipe_left, name_2: name_pipe_right);
if (result > 0) {
    char old_name_right[64];
    char old_name_left[64];
    char new_name_right[64];
    char new_name_left[64];
    snprintf( s: old_name_right, maxlen: sizeof(old_name_right), format: "pipe_client_right%d", i);
    snprintf( s: old_name_left, maxlen: sizeof(old_name_left), format: "pipe_client_left%d", i);
    snprintf( s: new_name_right, maxlen: sizeof(new_name_right), format: "used_pipe_client_right%d", i);
    snprintf( s: new_name_left, maxlen: sizeof(new_name_left), format: "used_pipe_client_left%d", i);
    rename( old: old_name_right, new: new_name_right);
    rename( old: old_name_left, new: new_name_left);
    id_pipe = i;
    break;
}
```

Code responsable du flag des pipes utilisés

Une fois les pipes utiles flagués nous entrons dans une boucle while infinie et nous déclarons une structure de type Answer qui servira à contenir la réponse du client elle-même et si celle-ci est de forme valide.

```c
Answer ans;
if (result > 0) {
    interface_start();
    while(1) {
        ans.code = -1;
        interface_choix();
        while (ans.code == -1) {
            ans = interface_menu();
        }
        if (ans.code == 0) {
            continue;
        }
        send_data_to_routing( request: ans.answer,  pipe_pointer: &local_client_pipe);
        show_answer_from_routing();
    }
} else {
    printf( format: "No free pipe found, ending the program\n");
    exit( status: 0);
}
```

Partie fonctionnelle de client.c

Donc une fois nos pipes réservés, nous entrons dans cette partie où nous utilisons plusieurs méthodes. La méthode interface_choix correspond au menu de base où l'utilisateur a le choix de demander un menu ou de quitter l'application.

Interface_menu correspond au menu où l'utilisateur formulera son choix de carte. Si l'utilisateur formule un choix valide alors la valeur de ans sera actualisée et contiendra alors sa réponse. On utilisera alors la méthode send_data_to_routing qui sert à écrire sur le pipe out du client qui envoie la réponse vers le serveur de routing. Le serveur de routing traitera alors sa demande et écrira sa réponse sur le pipe in du client.

```c
int verify_request_shape(char* request) {
    if (strlen( s: request) == 14 && request[4] == '|' && request[9] == '|') {
        return 1;
    }
    else {
        return 0;
    }
}
```

Méthode permettant de vérifier la bonne forme de la demande du client

Vient alors la méthode show_answer_from_routing qui sert comme son nom l'indique à lire et à afficher ce qui a été écrit sur le pipe in du client.

Une fois la réponse affichée, on reboucle et on recommence.

Si par hasard le client décide de quitter l'application, les pipes étant flagués comme utilisés il faut les flaguer de nouveau pour les remettre un statut de libre.
Et pour cela il faut prendre deux cas de figures : celui où le client quitte normalement en utilisant le code et celui où il utilise CTRL+C

Pour le premier cas de figure c'est assez simple : dans interface_choix avant d'utiliser exit(0) on renomme les pipes.

```
switch (choice) {
    case 1:
        printf( format: "\nYou choose to look at the menus\n");
        return 1;
    default:
        char old_name_right[64];
        char old_name_left[64];
        char new_name_right[64];
        char new_name_left[64];
        snprintf( s: old_name_right, maxlen: sizeof(old_name_right), format: "used_pipe_client_right%d", id_pipe);
        snprintf( s: old_name_left, maxlen: sizeof(old_name_left), format: "used_pipe_client_left%d", id_pipe);
        snprintf( s: new_name_right, maxlen: sizeof(new_name_right), format: "pipe_client_right%d", id_pipe);
        snprintf( s: new_name_left, maxlen: sizeof(new_name_left), format: "pipe_client_left%d", id_pipe);
        rename( old: old_name_right, new: new_name_right);
        rename( old: old_name_left, new: new_name_left);
        printf( format: "Goodbye !\n");
        exit( status: 0);
}
```

Code de interface_choix permettant de changer le nom des pipes avant de faire exit(0)

Cependant si le client quitte avec CTRL+C il faut penser à analyser le signal SIGINT avec la ligne de code « signal(SIGINT, ending_process) ; » et agir avec un bout de code correspond à l'action à exécuter au cas où le client appui sur CTRL+C:

```
void ending_process(int signal) {
    if (signal == SIGINT) {
        printf( format: "\nCTRL+C pressed, ending the process.\n");
        char old_name_right[64];
        char old_name_left[64];
        char new_name_right[64];
        char new_name_left[64];
        snprintf( s: old_name_right, maxlen: sizeof(old_name_right), format: "used_pipe_client_right%d", id_pipe);
        snprintf( s: old_name_left, maxlen: sizeof(old_name_left), format: "used_pipe_client_left%d", id_pipe);
        snprintf( s: new_name_right, maxlen: sizeof(new_name_right), format: "pipe_client_right%d", id_pipe);
        snprintf( s: new_name_left, maxlen: sizeof(new_name_left), format: "pipe_client_left%d", id_pipe);
        rename( old: old_name_right, new: new_name_right);
        rename( old: old_name_left, new: new_name_left);
        exit( status: 0);
    }
}
```

Code exécuté lorsque le client appuie sur CTRL+C

Globalement notre code dans client.c marche comme nous le voulons. Si on veut l'améliorer on pourrait peut être complexifier la méthode verify_request_shape afin de prendre plus de cas de figure imaginable, étant donné que notre code est peut être assez simpliste il pourrait certainement avoir des erreurs qui se glisseraient si on entre certaines choses.

Mais étant donné que le code est censé lire des QR code qui sont posés sur les tables des restaurants, on a un contrôle assez strict sur ce qui est injecté dans notre code.

Nous avons volontairement fait un code simpliste ici qui sert juste à recueillir les réponses du client et envoyer/récupérer les réponses sur le serveur de routage car nous avons pas repérer des besoins particulier sur la partie client. Étant donné que les données sont traitées sur la partie routing ou sur la partie data.

Partie 2 – Routing.c :

Ici notre fichier main_routing.c a pour mission de récupérer les requêtes client, de les envoyer aux serveurs de données, de récupérer les résultats des serveurs de données et de renvoyer grâces aux pipes au client qui avait demandé ces données.

main_routing.c génère aussi tous les pipes nécessaires au fonctionnement de l'ensemble, c'est pour cela qu'il faut que ce soit main_routing.c qui soit exécuté en premier.

Lorsqu'on va exécuter main_routing.c celui ci va tout d'abord vérifier si le nombre d'arguments entrés est correct et si les arguments sont corrects

```c
if (argc > 3) {
    printf( format: "Too much args \n");
    exit( status: 1);
}
else if (argc < 3) {
    printf( format: "Not enought args \n");
    exit( status: 1);
}
else if (atoi( nptr: argv[1]) > 5 || atoi( nptr: argv[1]) <= 0) {
    printf( format: "Too much or not enought client, should be between 1 and 5 \n");
    exit( status: 1);
}
else if (atoi( nptr: argv[2]) > 3 || atoi( nptr: argv[2]) <= 0) {
    printf( format: "Too much or not enought server, should be between 1 and 3 \n");
    exit( status: 1);
}
```

Code permettant de vérifier le nombre d'arguments

Une fois que l'utilisateur a entré bon nombre d'arguments. Le code inscrit le nombre de paires de pipes client et le nombre de paires de pipes serveur qui ont été créées dans le fichier value.txt .

```c
FILE *file = fopen( filename: "value.txt",  modes: "w");
if (file == NULL) {
    printf( format: "Failed to open value.txt");
    exit( status: 1);
}
fprintf( stream: file,  format: "%s\n", argv[1]); //maximum of client
fprintf( stream: file,  format: "%s\n", argv[2]); //maximum of server
fclose( stream: file);
```

Code permettant d'inscrire dans value.txt le nombre de paires de pipes serveur et client

Une fois le nombre de paires de pipes inscrit dans value.txt, le code va se charger de générer l'ensemble de ces pipes avec le bon nom et leurs indices qui les accompagnent.

Pour se faire on va utiliser une boucle for accompagné de strings décrivant les noms et de la méthode create_pipe qui est définie dans la librairie pipe_controler.c qu'on a écrit.

```
for (int i = 0; i < atoi( nptr: argv[1]); i++) {
    char name_pipe_client_right[STRING_SIZE];
    char name_pipe_client_left[STRING_SIZE];
    snprintf( s: name_pipe_client_right, maxlen: sizeof(name_pipe_client_right), format: "pipe_client_right%d", i);
    snprintf( s: name_pipe_client_left, maxlen: sizeof(name_pipe_client_left), format: "pipe_client_left%d", i);

    create_pipe( name_1: name_pipe_client_right, name_2: name_pipe_client_left);
}
for (int i = 0; i < atoi( nptr: argv[2]); i++) {
    char name_pipe_server_right[STRING_SIZE];
    char name_pipe_server_left[STRING_SIZE];
    snprintf( s: name_pipe_server_right, maxlen: sizeof(name_pipe_server_right), format: "pipe_server_right%d", i);
    snprintf( s: name_pipe_server_left, maxlen: sizeof(name_pipe_server_left), format: "pipe_server_left%d", i);

    create_pipe( name_1: name_pipe_server_right, name_2: name_pipe_server_left);
}
```

Code permettant de générer les paires de pipes

Désormais une fois que les pipes sont créés nous devons les initialiser et stocker leurs descripteurs dans une structure de type Pipe qui est décrite dans pipe.controler.h

Pour ce faire nous utilisons la méthode initialise_pipe définie elle aussi dans pipe_controler.c

```
if (atoi( nptr: argv[2]) == 3) {
    initialise_pipe( self: &pipe_server_2, name_1: "pipe_server_left2", name_2: "pipe_server_right2");
}
if (atoi( nptr: argv[2]) >= 2) {
    initialise_pipe( self: &pipe_server_1, name_1: "pipe_server_left1", name_2: "pipe_server_right1");
}
if (atoi( nptr: argv[2]) >= 1) {
    initialise_pipe( self: &pipe_server_0, name_1: "pipe_server_left0", name_2: "pipe_server_right0");
}
```

Extrait de code permettant d'initialiser les pipes

Une fois que tous les pipes sont initialisés nous entrons dans une boucles while(1) et dépendant du nombre de client nous vérifions de manière cyclique si les pipes client pour le serveur de routage contiennent quelque chose, et si effectivement ils contiennent quelque chose alors le code procède au traitement de la requête.

```
if (nb_client == 5 && is_pipe_empty( pipe_id: pipe_client_4.id_in) == 0) {
    printf( format: "data received from client 4\n");
    char container[BUFFER_SIZE];
    char address_server[4];
    int reading = read_pipe( self: pipe_client_4.id_in, container);
    if (reading <= 0) {
        printf( format: "Failed to read the pipe\n");
        exit( status: 1);
    }
    strncpy( dest: address_server, src: container, n: 4);
    if (atoi( nptr: address_server) == 1 && nb_server >= 1) {
        write_pipe( self: pipe_server_0.id_out, text: container);
        char container_data[BUFFER_SIZE];
        read_pipe( self: pipe_server_0.id_in, container: container_data);
        write_pipe( self: pipe_client_4.id_out, text: container_data);
    } else if (atoi( nptr: address_server) == 2 && nb_server >= 2) {
        write_pipe( self: pipe_server_1.id_out, text: container);
        char container_data[BUFFER_SIZE];
        read_pipe( self: pipe_server_1.id_in, container: container_data);
        write_pipe( self: pipe_client_4.id_out, text: container_data);
    } else if (atoi( nptr: address_server) == 3 && nb_server >= 3) {
        write_pipe( self: pipe_server_2.id_out, text: container);
        char container_data[BUFFER_SIZE];
        read_pipe( self: pipe_server_2.id_in, container: container_data);
        write_pipe( self: pipe_client_4.id_out, text: container_data);
    } else {
        printf( format: "Server unreachable !\n");
    }
}
```

Extrait de code permettant le routing de la requête vers le serveur de données. Et de récupérer la réponse du serveur.

Si le code rencontre un pipe client non vide alors il va lire ce dernier. Puis écrire la requête du client dans un string nommé « container » de longueur BUFFER_SIZE (2KB).

Il extrait les 4 premiers chiffres de container dans « address_server » qui correspond à l'adresse du serveur.

Ensuite on passe dans une succession de if-else qui transforme address_server en entier avec atoi() et qui vérifie quel serveur le client demande.

Une fois le bon serveur repéré, le serveur de routage écrit la demande du client sur le pipe qui va du serveur de routage à celui de données. Puis le serveur de routage récupère la réponse du serveur de donnée une fois que celui-ci à répondu et renvoie la réponse au pipe du client qui avait demandé la donnée.

Une fois ces étapes effectuées, la boucle while reboucle et on recommence.

Pour améliorer ce code on pourrait essayer de trouver une méthode pour aller jusqu'à n-utilisateur en simultané, car à présent on est limité à 5 utilisateurs en simultané.

Cependant notre code marche globalement bien dans l'état actuel pour ce qu'on lui demande de faire.

Concernant le routage on a tout cloisonné de la sorte pour être sûr que celui qui demande des informations reçoive uniquement les informations qu'il a demandé, et pas celle de quelqu'un d'autre. Ce n'est pas forcément la manière la plus optimisée pour faire mais cela fonctionne plutôt bien et ne pose pas de soucis, le seul souci de cette méthode c'est qu'on est limité dans le nombre d'utilisateurs.

Partie 3 – Data.c :

La partie data représente les serveurs de données. Lors de son lancement, on lui attribue un document où elle ira chercher les différents menus à sa disposition.

Son fonctionnement est assez simple. Le processus attend juste de recevoir des instructions du serveur de routage. Une fois qu'il a reçu les instructions du routage, le processus formate ces dernières afin de créer un path vers le fichier texte contenant le menu à afficher.

```
snprintf( s: old_name_right, maxlen: sizeof(old_name_right), format: "pipe_server_right%d", server_id-1);
snprintf( s: old_name_left, maxlen: sizeof(old_name_left), format: "pipe_server_left%d", server_id-1);
snprintf( s: new_name_right, maxlen: sizeof(new_name_right), format: "used_pipe_server_right%d", server_id-1)
snprintf( s: new_name_left, maxlen: sizeof(new_name_left), format: "used_pipe_server_left%d", server_id-1);
rename( old: old_name_right, new: new_name_right);
rename( old: old_name_left, new: new_name_left);
```

Au lancement du processus, celui-ci va renommer les pipes qu'il utilisera afin de les flaguer comme utilisés.

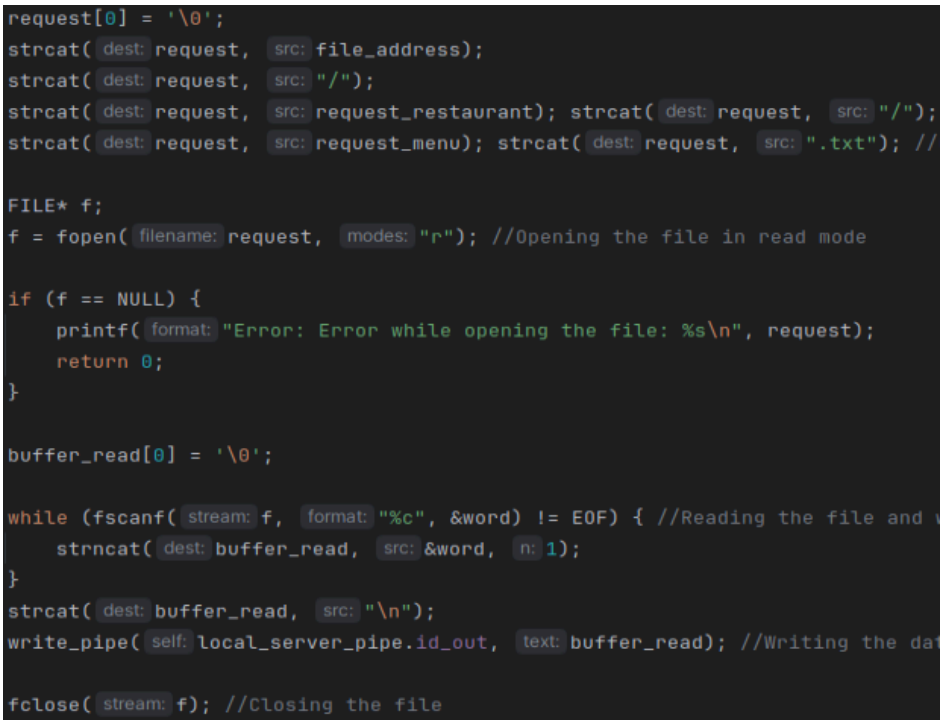
Une fois flagués, le programme entrera dans une boucle infinie où il vérifiera continuellement si quelque chose se trouve dans le pipe d'entrée.

Et si le processus détecte qu'il y a quelque chose alors il lira le pipe puis écrira sa réponse dans le pipe de sortie.

```
while(1) {
    usleep( useconds: 100);
    if(is_pipe_empty( pipe_id: local_server_pipe.id_in) == 0) {
        printf( format: "data received from routing.\n");
        read_txt_doc();
    }
}
```

Code exécutant la boucle infinie afin de vérifier s'il y a une entrée dans le pipe

Une fois que le code a détecté qu'il y a quelque chose dans le pipe entrant alors il lance la méthode read_txt_doc qui va chercher l'instruction dans le pipe et qui va envoyer une réponse dans le pipe sortant.



```
request[0] = '\0';
strcat( dest: request,  src: file_address);
strcat( dest: request,  src: "/");
strcat( dest: request,  src: request_restaurant); strcat( dest: request,  src: "/");
strcat( dest: request,  src: request_menu); strcat( dest: request,  src: ".txt"); //

FILE* f;
f = fopen( filename: request,  modes: "r"); //Opening the file in read mode

if (f == NULL) {
    printf( format: "Error: Error while opening the file: %s\n", request);
    return 0;
}

buffer_read[0] = '\0';

while (fscanf( stream: f,  format: "%c", &word) != EOF) { //Reading the file and
    strncat( dest: buffer_read,  src: &word,  n: 1);
}
strcat( dest: buffer_read,  src: "\n");
write_pipe( self: local_server_pipe.id_out,  text: buffer_read); //Writing the da

fclose( stream: f); //Closing the file
```

Extrait du code qui formate l'instruction de routing et qui va lire et écrire sur le pipe sortant le fichier txt correspondant à la demande du client.

Une fois que le processus à écrit la réponse à la requête sur le pipe sortant alors on reboucle et le code recommence à chercher une demande de la part du serveur de routage.

Pour améliorer notre code, on pourrait imaginer la prise en charge d'autres types de fichier que les fichiers texte pour conserver la donnée.

Par exemple on avait imaginé prendre en charge les .yaml mais malheureusement on a pas eu le temps de maîtriser suffisamment cette technologie afin de l'implémenter de façon solide. On aurait également pu utiliser une base de données SQLite mais on s'est confronté au même souci.

Sinon à part ça le code reste assez simpliste. Ce qui était voulu de notre part puisque nous n'avions pas repéré de besoins particuliers dans le sens que ce code servait juste à accéder à de la donnée et à la renvoyer vers le serveur de routage.

Partie 4 – Pipe_controler.c :

Pipe controler correspond à la librairie que l'on a écris afin d'assurer la gestion des pipes pour la communication inter-processus.

Celle ci permet de:

- Créer des structures Pipe, contenant des paires de pipes.
- Créer des paires de pipes.
- Initialiser des paires de pipes et insérer leurs descripteurs dans la structure Pipe.
- Écrire sur un pipe.
- Lire sur un pipe.
- Vérifier si un pipe est vide ou non.

Notre librairie est assez basique, dans le sens que tous nos pipes créés sont ouverts en lecture et écriture. Il faut donc bien réfléchir dans quel sens on compte lire et écrire dans nos processus si on ne veut pas créer de conflits dans l'ensemble.

```
int initialise_pipe(Pipe* self, char* name_1, char* name_2) {
    int open_in = open( file: name_1,  oflag: O_RDWR);
    int open_out = open( file: name_2,  oflag: O_RDWR);

    self->id_in = open_in;
    self->id_out = open_out;

    if (self->id_in > 0 || self->id_out > 0) {
        printf( format: "Pipe successfully opened \n");
        return 1;
    }
    else {
        printf( format: "Problem encountered while opening the p
        return -1;
    }
}
```

Code permettant l'initialisation en écriture/lecture des pipes.

Pour améliorer notre code on pourrait essayer de limiter les conflits en ouvrant que en lecture ou en écriture. Mais nous avons fait le choix de laisser une liberté plus grande sur les pipes au cas où on aurait un besoin spécifique à un moment donné, mais celui-ci n'est jamais arrivé.

Partie 5 – La répartition du travail :

Le programme à été codé en binôme, nous avons donc décidé de nous partager le travail d'une manière à mettre en avant les capacités de chacun. Savio ayant plus d'expérience dans le développement s'est occupé du développement du code, et Martin ayant une meilleur compétence analytique s'est occupé des phases de test et de recherche des solutions aux problèmes.

Ce qui a été fait par Savio:
- Développement du code du client, du routage et du serveur de données.
- Rédaction du rapport.
- Débug du code

Ce qui a été fait par Martin:
- Travail de recherche sur les solutions possibles
- Développement de la librairie pipe_controler.c
- Phase de test du programme et de recherche des bugs
- Participation à la production du rapport.

Partie 6 – Nos sources :

Les sources utilisées pour ce projet sont nombreuses et perdues pour la plupart d'entre elles car elles se résument soit à des vidéos youtube explicatives soit à des requêtes chatGPT pour comprendre certaines notions. Mais si on devait les résumer on aurait:

- ChatGPT
- Diverses vidéos Youtube
- Stackoverflow