Année : 2024 – 2025
Projet : RE2xx Réseaux et applications réparties

Documentation de Messag-eirb

Mathieu Koltok - Nicolas Dubuisson

19 octobre 2024

Sommaire

1 Introduction **3**

2 Organisation des fichiers **3**

3 Démarrer l'application **3**

4 Connexion **4**

5 Sécurité et gestion d'erreur **4**

6 Protocole de communication de la couche applicative **5**

6.1 Requêtes/Réponses 5

6.2 Sémantique et syntax des messages 7

6.3 Objets transférés 8

6.4 Régles de traitement 8

7 Axe d'amélioration **10**

8 Conslusion **11**

1 Introduction

Pour ce projet, nous avons choisi le sujet 1. Ce sujet nous a particulièrement intéressé, car il nous semblait plus pertinent dans son implémentation et d'être en lien direct avec les fonctions que nous avons vu en cours précédement. Il a permis de mettre en avant notre motivation à travers ces séances. Nous avons donc développé une messagerie instantanée "sécurisée" où il est possible de communiquer avec plusieurs utilisateurs. Nous pouvons communiquer soit directement avec un autre utilisateur, soit avec plusieurs utilisateurs en même temps grâce à la création de salons. Mais il est aussi possible d'envoyer des fichiers au destinataire. On peut retrouver toutes ses fonctionnalités grâce à une interface intéractive en indiquant nos choix d'actions.

Note : Maintenant notre application est entièrement fonctionnelle mais nous regrettons beaucoup lors de la démonstration de ne pas avoir fait fonctionner la partie sur l'envoi des fichiers. En fait, dans notre implémentation de code, nous avons involontairement omis le contrôle d'envoi et de bonne réception du nombre d'octets envoyés à chaque message à travers des boucles "while()". Cela a pu créer des erreurs lors de notre implémentation et nous remercions M. Bruneau-Queyreix de nous avoir éclairé sur ce point. Tout de suite après notre démonstration, nous avons corrigé cet aspect mal fonctionnel de notre code pour apaiser notre frustration. Maintenant l'entièreté de nos fonctions marchent.

2 Organisation des fichiers

Nos avons organisé notre projet en 5 fichiers. Un fichier client (**client.c**), un fichier server (**server.c**), deux fichiers auxilières (**aux.c** et **aux.h**) qui regroupent les fonctions communes du client et du server. Puis nous avons un fichier **Makefile** pour générer les éxecutables de ces fichiers. Enfin, nous avons un fichier **README.md** avec la liste des exécutables. A la suite du "make", une documentation est aussi générée grâce à l'outil doxygen, celle-ci peut-être consulté sur un navigateur grâce à son fichier index.html.

3 Démarrer l'application

Merci de suivre ces étapes clés pour le bon fonctionnement de l'application :

- Commencez par exécuter le Makefile : make
- Vous avez la possibilité de nettoyer le répertoire : make clean
- Tapez **./server** dans un terminal
- Utilisez **./client** pour créer un client
- Une fois le client lancé, vous devez créer un compte obligatoirement puis vous y connecter.

4 Connexion

Connectez-vous avec le compte que vous vous êtes créé.

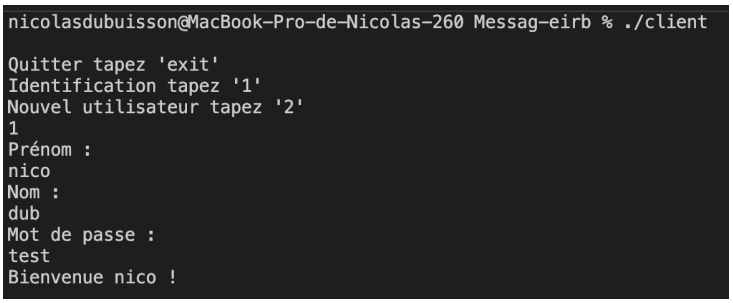

FIGURE 1 – Identification

Vous avez plusieurs fonctionnalités qui ont été implémentées, vous pouvez :

- Quitter proprement la messagerie.
- Obtenir la liste des utilisateurs connectés
- Obtenir la liste de salons ouverts et les utilisateurs qui sont actuellement connectés dessus.
- Discuter avec un utilisateur.
- Envoyer des fichiers à un utilisateur.
- Créer un salon.
- Discuter dans un salon avec d'autres utilisateurs.

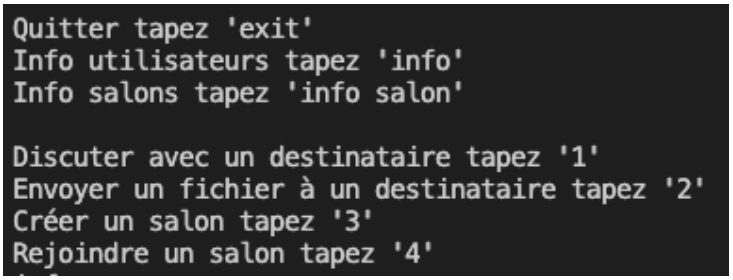

FIGURE 2 – Identification

Lorsque vous êtes dans un salon ou simplement en train de communiquer avec un autre utilisateur, vous avez la possibilité de rejoindre le menu en utilisant **menu**.

5 Sécurité et gestion d'erreur

La sécurité s'éffectue surtout lors de la connection d'un utilisateur. Lorsqu'un utilisateur créé un compte, nous créons une socket sur le server avec les informations necessaires, puis le server écrit sur un fichier texte les informations de l'utilisateur avec la syntaxe suivante : **prenom ;nom ;motdepasse**.
Lorsqu'un utilisateur (déjà présent dans la base de données) se connecte nous vérifions si les 3 champs sont identiques à ceux de la base de données. Il existe 3 cas possible :

- L'utilisateur existe, il arrive dans un menu.
- L'utilisateur s'est trompé de mot de passe ou d'identifiant (nom, prénom), alors il doit recommencer.
- L'utilisateur essaie de se connecter avec un compte déjà connecté, alors il doit recommencer. La figure 3 montre une tentative de connexion alors que cet utilisateur est déjà connecté avec le même compte.

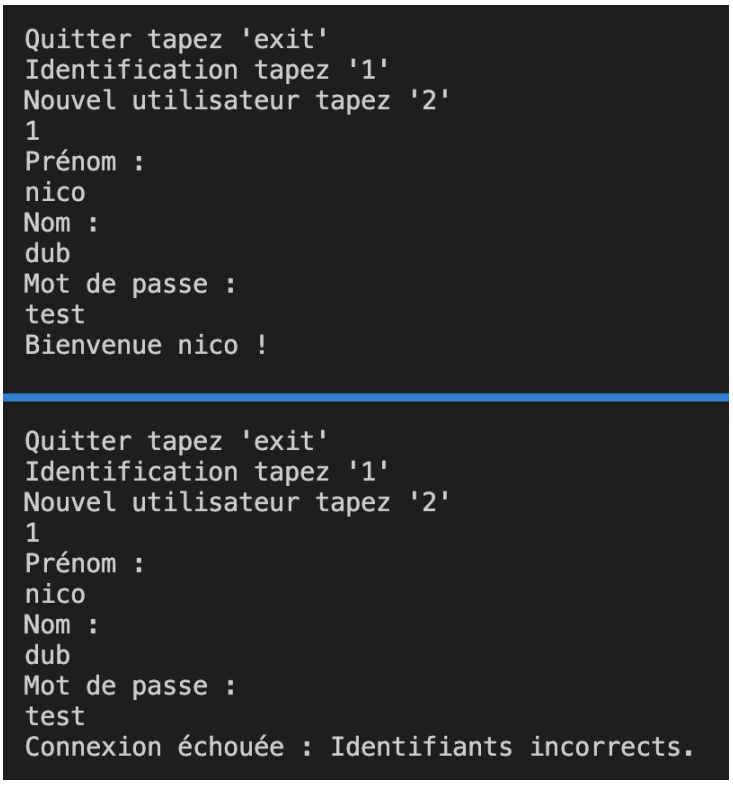

FIGURE 3 – Tentative de connexion

6 Protocole de communication de la couche applicative

Dans notre application développée en C utilisant le protocole de couche transport "TCP" grâce aux fonctions read() et write() qui permettent de communiquer sur des sockets. Nous ne suivons pas une norme standardisée. Par conséquent notre protocole de couche applicative à ses propres spécifications que nous allons détailler ci-dessous.

6.1 Requêtes/Réponses

Toutes nos requêtes se traduisent sous une même forme, c'est-à-dire l'envoie des différents champs d'une structure bien définie répondant à nos besoins.

```c
typedef struct {
    char* prenom;
    char* nom;
    char* mdp;
    int socket_fd;
    char* type;
    char* message;
    char* dest;
} Utilisateur;
```

FIGURE 4 – Structure de données

Pour toutes les requêtes envoyées par un serveur ou un client, on retrouve un "type" qui est le champ le

plus important et qui détermine quelle interprétation le serveur doit faire avec cette requête. Une requête va utiliser aussi "socket_fd" qui permet de différencier les clients grâce à leur socket. Comme ces deux informations sont présentes dans la même structure, il est très simple de les associer. Par exemple un client peut réaliser ces différentes demandes :

- connect : Requête vérifiant les informations d'authenfication de l'utilisateur grâce aux champs "prenom" "nom" "mdp".
- new : Requête permettant la création d'un nouvel utilisateur grâce aux champs "prenom" "nom" "mdp".
- info : Requête permettant d'afficher l'ensemble des utilisateurs actuellement connectés sur le serveur.
- message : Requête permettant l'envoie d'un message à un destinataire choisi.
- file : Requête permettant l'envoie d'un fichier à un destinataire choisi.
- deconnexion : Requête permettant d'indiquer la déconnexion d'un utilisateur du serveur.
- info salon : Requête permettant l'affichage des différents salons créés sur le serveur.
- new salon : Requête permettant la création d'un nouveau salon.
- join salon : Requête permettant de rejoindre un salon existant.
- message salon : Reqûete permettant l'envoie d'un message dans un salon à l'ensemble des utilisateurs présents dans ce salon.
- dec salon : Requête permettant d'indiquer la déconnexion d'un utilisateur d'un salon.

Tous ces types sont complémentaires avec les autres champs nécessaires de la structure selon les besoins. Par exemple lors de l'authentification d'un utilisateur, le champ "message" n'a pas besoin d'être complété, il sera donc laissé à vide.

Toutes les requêtes passent par le serveur. Le serveur répond en fonction de la requête au client source ou à un/des client(s) distant(s). Le server répond par l'envoi d'un simple message avec un "write()". Le server n'envoi pas la structure au client. Le schéma ci-dessous montre le fonctionnement du système que nous avons mis en place dans le cas d'une connexion.

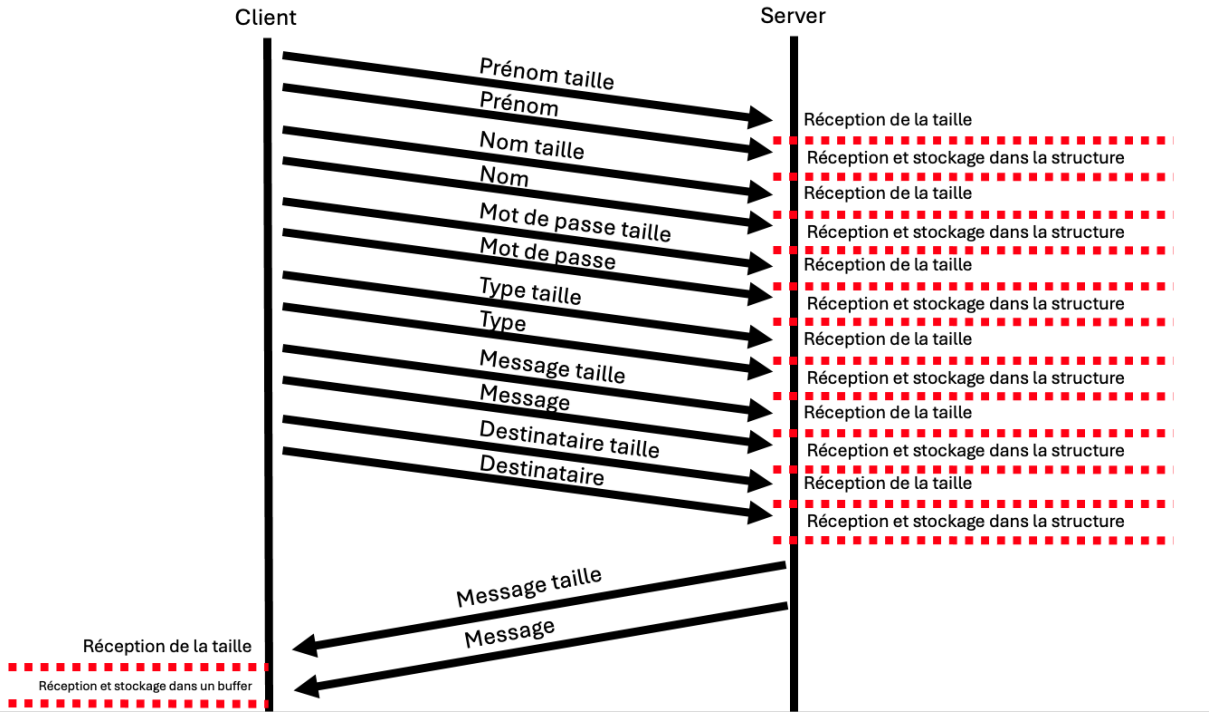

FIGURE 5 – Structure de données

6.2 Sémantique et syntax des messages

Lors d'une requête un formatage spécifique est opéré en fonction du type d'action que l'utilisateur effectue. La structure "user" est remplie avec les informations que donne l'utilisateur. Puis des "write()" envoient chaque champ de la structure au serveur.

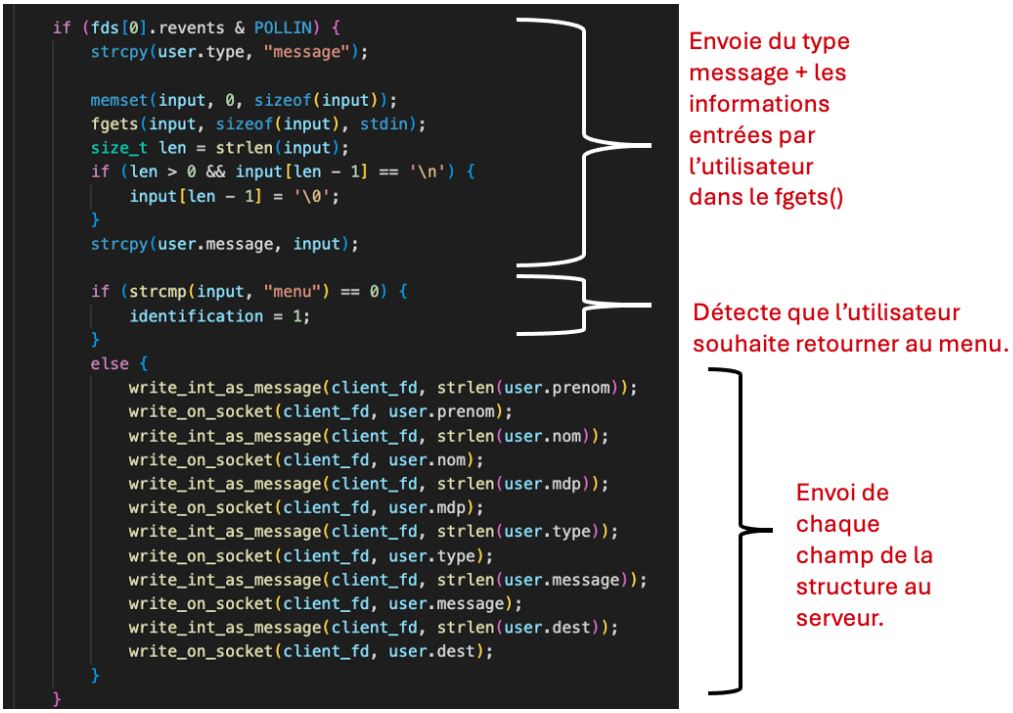

FIGURE 6 – Envoie de la structure côté client

Du côté du serveur, on utilise la méthode "poll". On va parcourir chaque socket (les sockets des clients) et regarder si un client à envoyé des données. Si un client envoi des données, il a envoyé les différents champs de la structure décrite précédement. Pour cette raison sur sa socket nous allons lire les différentes données (les unes après les autres) et remplir chaque champ de notre structure "user" avec les informations récupérées par chaque "read()". Par la suite, les informations seront utilisées et traités en fonction du champ "user.type" qui définit le type d'action et le champ "user.dest" qui détermine le destinataire qui reçoit les données.

```
Utilisateur user;

int message_size = read_int_from_socket(client_fd);
printf("message_size %d\n", message_size);
if (message_size <= 0 || message_size >= BUFFER_SIZE) {
    printf("Closing connection for user.prenom\n");
    close(client_fd);
    fds[i].fd = -1;
    fds[i].revents = 0;
    fds[i].events = POLLIN;
    continue;
}
user.prenom = (char*)malloc(message_size + 1);
read_message_from_socket(client_fd, user.prenom, message_size);
user.prenom[message_size] = '\0';
```

FIGURE 7 – Réception d'un champ de la structure côté serveur

Lorsque le serveur traîte la demande et effectue les redirections nécessaires, il écrit soit au client source

ou destinataire un message. La capture ci-dessous montre la réception d'un message provenant du serveur.

```
    else if (fds[1].revents & POLLIN) {

        int message_size = read_int_from_socket(client_fd);
        if (message_size <= 0 || message_size >= BUFFER_SIZE) {
            printf("Invalid message size: %d\n", message_size);
            break;
        }

        char* buffer = (char*)malloc(message_size + 1);
        int read_status = read_message_from_socket(client_fd, buffer, message_size);
        if (read_status <= 0) {
            printf("Server disconnected or error occurred.\n");
            free(buffer);
            break;
        }
        buffer[message_size] = '\0';
        printf("%s\n", buffer);

        free(buffer);

    }
}
```

FIGURE 8 – Réception d'un message provenant du serveur

6.3 Objets transférés

Nous avons déjà plus ou moins parlé des objets transférés lors d'une requête, mais selon l'action choisi par l'utilisateur le choix des objets diffère selon l'importance de l'action. Voici la liste des objets envoyés pour chaque type d'action :

- connect : prenom,nom,mdp,type
- new : prenom,nom,mdp,type
- info : prenom,type
- message : prenom,type,message,dest
- file : prenom,type,message,dest
- deconnexion :prenom,type
- info salon : prenom,type
- new salon : prenom,type,dest
- join salon : prenom,type,dest
- message salon : prenom,type,message,dest
- dec salon : prenom,type

6.4 Régles de traitement

Authentification des utilisateurs :

Lorsqu'un client se connecte, il envoie ses informations d'authentification (prénom, nom, mot de passe) au serveur.

Traitement : Le serveur vérifie si l'utilisateur existe et si le mot de passe correspond en comparant avec les données stockées (probablement dans un fichier).

Si l'authentification réussit, le client peut accéder aux fonctionnalités comme les salons et l'envoi de messages.

Envoi de messages :

Lorsqu'un utilisateur envoie un message, il doit spécifier la destination (un autre utilisateur ou un salon).

Traitement : Le serveur vérifie si la destination (un utilisateur ou un salon) existe, puis transfère le message à la bonne destination (un client spécifique ou tous les clients dans le salon).

Gestion des salons de discussion :

Les utilisateurs peuvent créer, rejoindre ou quitter des salons.

Requête de création de salon (new salon) : Le client envoie une commande new salon au serveur avec le nom du salon à créer. Si le salon n'existe pas déjà, le serveur l'ajoute.

Requête de rejoindre un salon (join salon) : Un utilisateur peut rejoindre un salon existant en envoyant une requête join salon, le serveur ajoute cet utilisateur à la liste des utilisateurs du salon.

Traitement : Le serveur garde une liste des salons actifs et des utilisateurs dans chaque salon. Cela permet de savoir à qui envoyer les messages lorsqu'un utilisateur envoie un message dans un salon.

Notifications et déconnexions :

Lorsqu'un utilisateur se déconnecte, une notification est envoyée au serveur, et celui-ci doit enlever l'utilisateur des salons ou des conversations actifs.

Traitement : Le serveur doit gérer les connexions et déconnexions, en s'assurant que les sockets soit bien fermées et les ressources (structure/tables) soient libérées.

Envoi de fichiers :

Le client peut envoyer un fichier.

Traitement : Le serveur reçoit les données du fichier (nom du fichier, contenu). Il doit transmettre le fichier au bon destinataire, à travers la structure de données.

7 Axe d'amélioration

Durant notre projet, nous avons omis, par manque de compréhension (cela nous a causé beaucoup de problèmes lors de l'implémentation du transfert de fichier) de suivre la structure de read() et write() recommandée à travers le cours.

Voulant implémenter le maximum de fonctionnalités pour notre messagerie, nous sommes passés à côté de la bonne gestion de l'envoi et de la réception des données. En fait, notre application ne s'assure pas de la bonne réception des données et dans un ordre cohérent. Si jamais des octets ne s'envoient pas correctement ou dans le mauvais ordre, alors notre application n'est pas fonctionnelle.

Les "write()" décrits plus tôt dans le rapport n'ont aucune vérification que l'ensemble des octets d'un champ de la structure ont été correctement envoyés avant de passer à l'envoi du champ suivant de la structure.

Les "read()" décrits plus tôt dans le rapport n'ont aucune vérification que l'ensemble des données reçues sont lues dans et dans le bon ordre et sont toutes présentes, avant de passer au "read()" du champ suivant de la structure.

Avec ces deux conditions réunies il est très probable d'arriver à des erreurs. Avant l'implémentation de la fonctionnalité d'envoi de fichiers, les erreurs n'existaient pas car nous faisons tourner notre machine en local et la quantité de données envoyée est très très faible. Donc, il est peu probable de rencontrer des erreurs.

Avec l'envoi des fichiers des problèmes ont commencé à apparaître. Cependant, étant donné que notre code commençait à être très consistant, nous n'avons pas eu le temps de faire face à ce problème avant la démonstration. Mais comme dit dans l'introduction, nous avons su rebondir suite à la démonstration pour corriger notre code, et le rendre fonctionnel.

Ce que nous avons modifié :

C'est très simple, nous utilisons toujours la même structure et nous envoyons/lisons toujours chaque champ de la structure avec des read() et des write(). Mais en plus, nous avons implémenté des fonctions permettant de vérifier le bon fonctionnement de l'envoi ou la réception des octets avant de passer à l'envoi ou la réception des octets d'un champ suivant de la structure.

Les fonctions vérifiant cela sont communes au client et au serveur, on les a donc implémenté dans le fichier "aux.c" :

```c
// Continue de lire jusqu'à ce que tous les
// octets nécessaires à l'entier soient reçus.
// Retourne l'entier lu.
int read_int_from_socket(int fd) {
  int val;
  int total_bytes_read = 0;
  int bytes_read;
  int size = sizeof(val);

  while (total_bytes_read < size) {
    bytes_read = read(fd, ((char*)&val) + total_bytes_read, size - total_bytes_read);
    if (bytes_read == 0) return 0;  // Connexion fermée.
    print_error(bytes_read, "read_int");
    total_bytes_read += bytes_read;
  }

  return val;
}

// Lit un message de 'size' octets depuis le socket 'fd' et le stocke dans 'buffer'.
// La fonction continue de lire jusqu'à ce que la totalité des octets soient reçus.
// Retourne le nombre total d'octets lus ou 0 si la connexion est fermée.
int read_message_from_socket(int fd, char* buffer, int size) {
  int total_bytes_read = 0;
  int bytes_read;

  while (total_bytes_read < size) {
    bytes_read = read(fd, buffer + total_bytes_read, size - total_bytes_read);
    if (bytes_read == 0) return 0;  // Connexion fermée.
    print_error(bytes_read, "read_message");
    total_bytes_read += bytes_read;
  }
```

FIGURE 9 – Fonctions vérifiants la bonne réception des messages

```c
// Utilise une boucle pour s'assurer que
//tous les octets de l'entier sont envoyés.
void write_int_as_message(int fd, int val) {
  int size = sizeof(val);
  int send = 0;

  while (send < size) {
    int temp_send = write(fd, ((char*)&val) + send, size - send);
    print_error(temp_send, "write_int");
    send += temp_send;
  }
}


// Utilise une boucle pour s'assurer que
//tous les octets de la chaîne sont envoyés.
void write_on_socket(int fd, char* s) {
  int size = strlen(s);
  int send = 0;

  while (send < size) {
    int temp_send = write(fd, s + send, size - send);
    print_error(temp_send, "write");
    send += temp_send;
  }
}
```

FIGURE 10 – Fonctions vérifiants le bon envoi des messages

8 Conslusion

Nous avons beaucoup apprécié implémenter cette application à travers ce projet. Nous avons choisi ce projet car c'était la première fois qu'on utilisait l'implémentation de sockets pour faire fonctionner différents scripts entre eux à travers un réseau. Nous avons rencontré des difficultés à certains moments, mais nous avons aussi su les surpasser en apportant toujours des corrections et modifications même après la démonstration. Des améliorations pourraient toujours être apportées, mais le projet a été aussi une bonne expérience en termes de gestion de projet, organisation des idées, ordre d'implémentation...