Rapport de projet
Groupe 5 - M1-APP-RS

ARTN71 - Programmation multitâches et temps réel

7 Octobre 2021

Antoine Joly
Victor Garnier
Nicolas Magnani

Tuteur : M. BOUGUEROUA Lamine

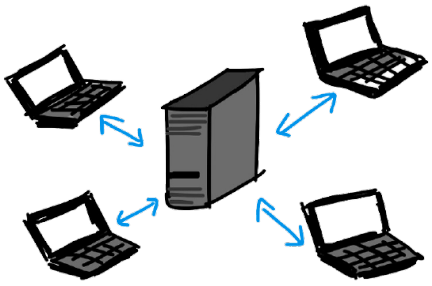


Table des matières

1 Introduction

Dans le cadre de notre cours de **Programmation multitâche et temps réel**, il nous a été demandé de développer une application multitâche en **Java** ou en **C**.

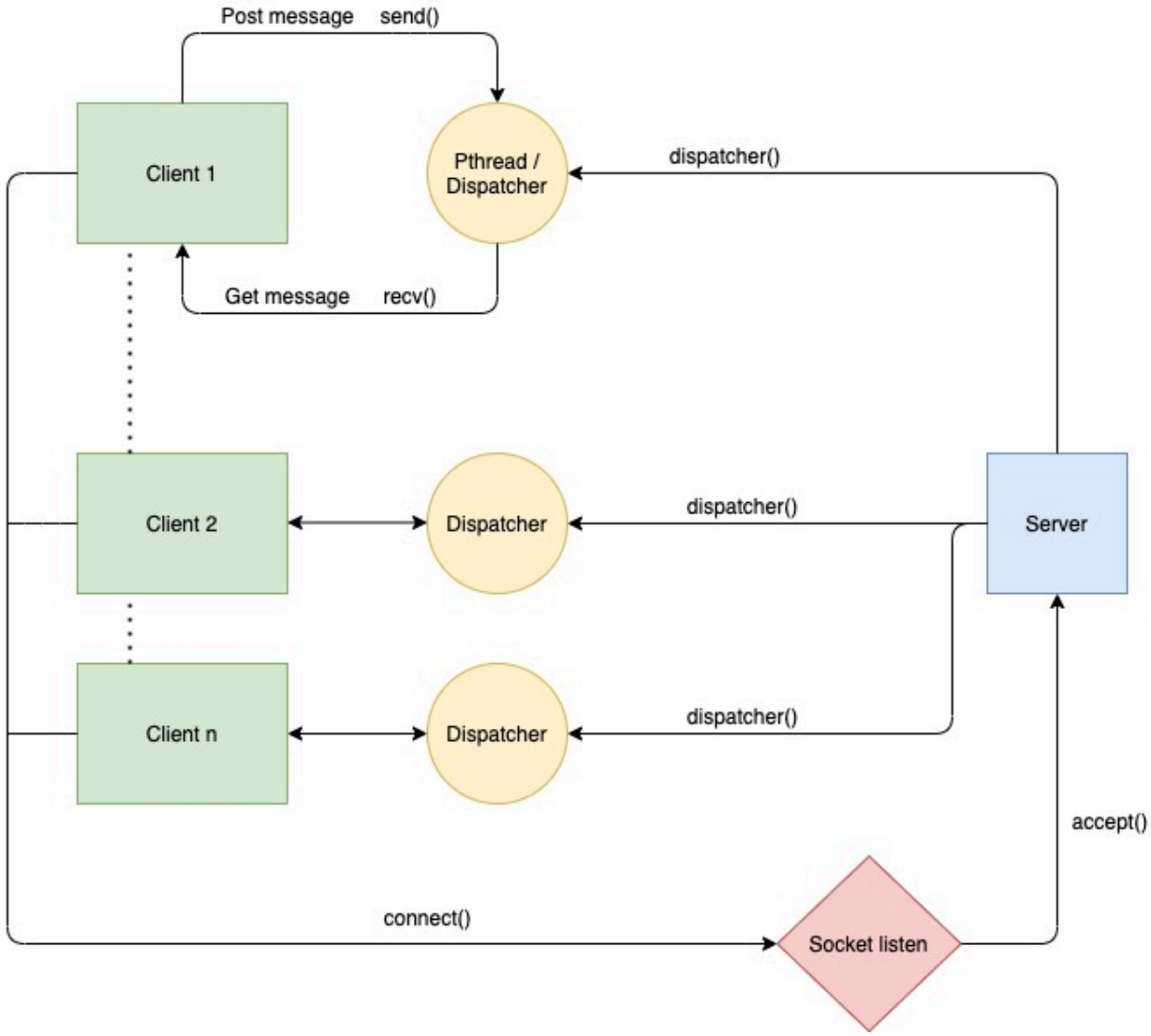
Cette application réalisée par Antoine, Nicolas et Victor est capable de mettre en valeur les **notions** que nous avons pu apprendre durant les différents cours avec M. BOUGUEROUA Lamine tels que la **synchronisation client/server**, l'**exclusion mutuelle** ou encore l'**interblocage de processus**.

Afin de réaliser ce travail, nous avons décidé d'utiliser le **langage de programmation C** couplé à sa **librairie GTK** pour l'interface graphique qui permettra à l'utilisateur d'avoir le meilleur confort d'utilisation possible.

2 Projet

Notre projet se compose de **trois** fichiers principaux :

- **Makefile** : fichier qui contient l'ensemble des paramètres de compilation avec gcc
- **server.c** : le code de configuration de la partie serveur (back-end)
- **client.c** : le code de configuration de la partie client (front-end)



Comme exposé dans l'introduction, notre objectif est de construire un système **multitâche** entre des **clients** et un **serveur**. L'ensemble des tâches réalisées sont exposées dans la suite de ce document.

2.1 Socket

Pour que deux ordinateurs puissent communiquer entre eux, nous utilisons les **sockets**. Simplement, il s'agit d'un **descripteur de fichier** (entier associé à un fichier ouvert (index)) qui permet la **communication** entre les programmes. Vu que tout est un fichier sur UNIX, notre file descriptor peut prendre beaucoup de formes comme un terminal, un pipe, etc.

Afin de créer cette communication dans notre client/server, nous utilisons le systemcall **socket()**. Il nous retourne un socket descriptor que nous pouvons utiliser avec différentes fonctions comme **send()** et **recv()** pour envoyer ou recevoir un message.

Il existe plusieurs types de socket. Dans notre projet, nous utilisons les **stream sockets** avec les protocoles **TCP/IP**. Les streams sockets permettent d'assurer que tout ce qui a été tapé dans la console arrive dans le même ordre dans le server. **TCP/IP** fait en sorte que les données arrivent les unes apres les autres et sans erreurs.

Voici comment nous avons implémenté nos sockets pour le projet :

```
1   sockfd = socket(AF_INET, SOCK_STREAM, 0); //call socket()
2   server_addr.sin_family = AF_INET; //IPv4
3   server_addr.sin_addr.s_addr = inet_addr(ip); //IP address
4   server_addr.sin_port = htons(port); // Port
```

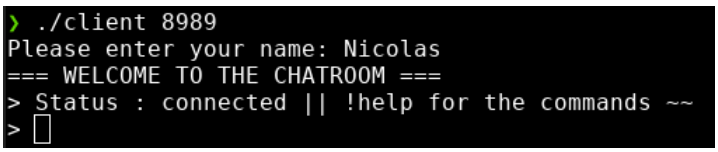
2.2 Client

Le client est la partie **front-end** de notre projet : il est l'initiateur de la connexion avec le serveur et permet l'envoi/reception des messages.

Schématiquement, il propose les fonctions suivantes :

```
1    socket() // client socket
2    connect() // connecting to the server
3    recv() // receiving message
4    send() // sending message
```

Le client appelle **socket()** et se lance dans une tentative de connexion au serveur via **connect()**. S'il est accepté, il utilise les syscalls socket **recv()** et **send()** pour les messages via **stdin/stdout**.



2.3 Serveur

Le server est la partie **back-end** de notre projet : il permet la gestion de l'ensemble des clients et de dispatcher les messages reçus.

Schématiquement, il propose les fonctions suivantes :

```
setsockopt() // check if the port is available
bind() // associate the socket with a port on the local machine
listen() // Listen to remote connections
accept() // accepting connections
recv() // receiving message
send() // sending message
```

Le serveur va dans un premier temps vérifier que le **port** demandé n'est pas utilisé pour un autre service. Il va ensuite associé son **addresse IP** avec le port via **bind()**. Il entre en mode écoute via **listen()** pour gérer l'ensemble des clients qui souhaitent se connecter. **Accept()** permet l'authentification (acceptation) des différentes connexions. De la même manière que le client, le serveur utilise les syscall **recv()** et **send()** via socket pour la gestion des données envoyées/reçues.

Pour la gestion des clients, nous avons implémenté une **struct client** qui se compose des données suivantes :

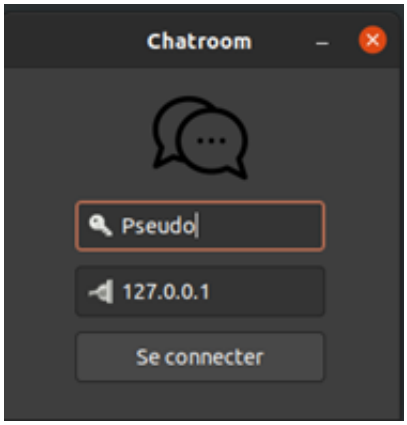
```
//Information needed from the client side
struct client
{
   int socket; //current file descriptor
   struct sockaddr_in address; //current address
   char nickname[16]; //current name
   struct client *next; //linked list of the clients
   int id; //id client
 };
```

- **socket** : le file descriptor du client (entier)
- **sockaddr_in** : les informations contenues dans le socket (address family, port number, IP, etc.)
- **nickname** : le pseudo utilisé par l'utilisateur
- **client *next** : la liste chainée de l'ensemble des clients
- **id** : l'id du client

Toutes ces informations permettent au serveur d'**identifier** correctement le client.

2.4 Interface

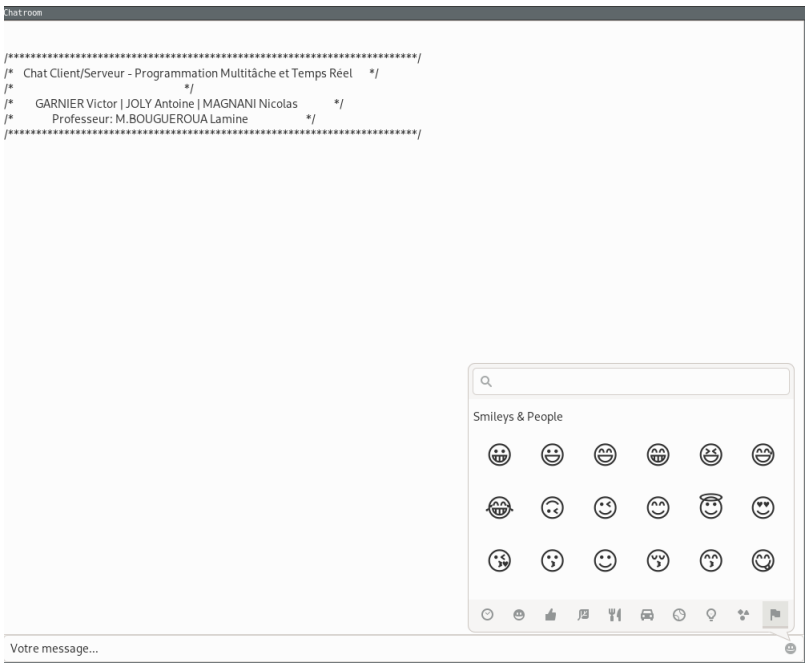
Afin de rendre notre projet plus accessible, nous avons utilisé la librairie **GTK3.0+**.



L'interface propose les fonctionnalitées suivantes :

- **Pseudo** : choix du pseudo de l'utilisateur
- **IP** : l'addresse IP du serveur
- **Se connecter** : demande de connexion au serveur

L'interface graphique du chat a été codé avec la librairie **GTK** du langage C et est intégrée directement dans le code du **client.c**. Avec cette interface, nous avons voulu reproduire au plus proche l'expérience utilisateur d'une plateforme tel que **Messenger** qui permettrai à l'utilisateur d'entrer son message de façon simple et efficace. Un onglet **emoji** a également été ajouté.



2.5 Multitâche

L'un des buts du projet fut de mettre en place un moyen de **gestion** des clients connectés au serveur. Il a été convenu de résoudre le problème de collisions entre les connexion des clients, l'envoi et la reception des messages et surtout les mises à jour régulières de la liste des membres.

Pour ce faire, nous avons utilisé les **librairies** suivantes :

- **pthread.h** : permet de créer une version de notre programme avec les paramètres propres du client
- **semaphore.h** : mise en place de la gestion de la concurrence entre les threads

Prenons un exemple de notre code, **server.c** :

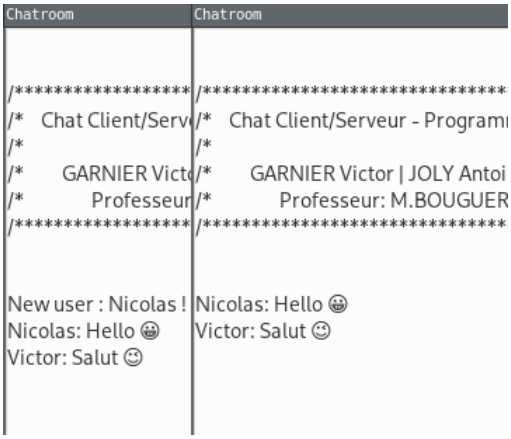
```
if (pthread_create(&tid, NULL, &dispatcher, (void *)c1) == -1)
       {
         perror("Error creating thread\n");
         exit(5);
       }
```

Dans un premier temps, le client connecté va créer un **thread** avec l'ensemble de ses paramètres dans la fonction **dispatcher()**. La thread va tourner en permanence en concurrence avec les autres threads (clients) en cours jusqu'à la fermeture du client (**fin du thread**).

```

void client_add(struct client *c1)
    {
      sem_wait(&semData);

      count++;

     if (header == NULL)
      {
        header = c1;
        (*c1).next = NULL;
      }
      else
      {
        (*c1).next = header;
        header = c1;
      }

      sem_post(&semData);

      return;
    }
```

Lorsque le thread va modifier des données partagées entre tous les threads, nous avons besoin de créer une **zone spéciale** (critical zone) avec le sémaphore pour protéger la modification entre tous les threads. Dans ce cas, nous utilisons les fonctions suivantes :

- **sem_wait** : lorsque la fonction est appelée, sa valeur est diminuée de 1 et devient blocante lorsqu'elle vaut 0.
- **sem_post** : lorsque la fonction est appelée, sa valeur est augmentée de 1 et devient ouvrante si sa valeur est différente de 0.




```
❯ ./server
IP : 127.0.0.1 || Port : 8989
Initialisation du chat
..
Listening : OK
Number of clients connected : 1
--------> IN : Nicolas
Number of clients connected : 2
--------> IN : Nicolas
Nicolas: Bonjour 😀
Nicolas: Hello ! :)
<------ OUT : Nicolas
Number of clients connected : 1
Number of clients connected : 2
--------> IN : Victor
Victor: Salut 😊
<------ OUT : Victor
Number of clients connected : 1
Nicolas: ddd
<------ OUT : Nicolas
Number of clients connected : 0
Number of clients connected : 1
--------> IN : Victor
Number of clients connected : 2
--------> IN : Nicolas
Nicolas: Hello 😀
Victor: Salut 😊
```

3 Conclusion

Ce projet nous a permis de mieux comprendre le fonctionnement d'un **système multitâche** et de son utilité avec notamment l'utilisation de **thread**. Parallèlement, nous avons pu apprendre à créer une interface graphique avec **GTK**.

Nous aurions aimé aller un peu plus loin et rendre nos tests plus concrêts en implémentant plus de fonctionnalités à notre client. En effet, nous voulions afficher plus de **détails** dans notre interface, comme l'heure d'envoi. De la même manière, il nous a paru judicieux d'y insérer un menu contextuel avec différentes commandes pour offrir au client une **meilleure expérience utilisateur**.

3.1 Répartition du travail

Tâches	Nom	Complète
Interface	Antoine	X
client.c	Victor	X
server.c	Nicolas	X
Rapport	N,V,A	X

4 Bibliographie

- *ARTN71 - Programmation multitâche et temps réel* par M. BOUGUEROUA Lamine, 2021.

- *Beej's Guide to Network Programming* par Brian "Beej Jorgensen" Hall, 2020.

- *Unix Network Programming : The Sockets Networking Api* par W. Richard Stevens, 1990.