

Salon de Clavardage

Esteban Launay Sacha Lorient

March 2017

Résumé

Dans le cadre de l'Unité d'Enseignement "Réseaux Informatiques" dirigée par Pierrick Passard, nous avons mis en place une salle de chat. Vous trouverez dans ce document les détails des fonctionnalités implémentées et les différentes difficultés rencontrées au cours du développement.

Table des matières

1	Introduction	2
2	Installer l'application	2
3	Sockets	3
4	Cahier des charges	4
5	Le protocole TCP	4
6	Détails de la réalisation	5
7	Jeux d'essais	6
8	Problèmes réglés depuis la présentation	10
9	Bugs persistants	10
10	Conclusion	11

1 Introduction

L'objectif de notre travail était de produire une petite application usant de "sockets" C. Nous nous sommes alors orienté vers un salon de chat. En effet, classique, ce type d'application est facile à appréhender mais a l'avantage de pouvoir accueillir une grande variété de fonctionnalités. C'est donc hautement évolutif. Nous avons établi un petit cahier des charges visible en section 4. Au fil de la lecture, vous trouverez le détail de notre réalisation illustrée par des captures d'écrans.

2 Installer l'application

L'application est scindée en deux parties. La première, le côté serveur (s'occupant de la réception des messages et de leur retransmission à tous les clients concernés) et la seconde le côté client (permettant l'envoi de requêtes et/ou messages au serveur). Veuillez suivre les instructions suivantes :

- *cd chemin/vers/la/racine/de/l'archive*
- *make*

Deux exécutables ont été générés : `serverApp` et `clientApp`.

Il ne vous reste plus qu'à lancer l'exécutable `serverApp` sur la machine que vous souhaitez transformer en serveur et `clientApp` sur les machines clientes. **Attention**, pour les clients il faut lancer l'exécutable suivi de l'adresse IP du serveur comme ceci : `./clientApp 192.168.1.1` .

Voici le lien de notre dépôt github si vous souhaitez télécharger l'archive : [github/Projet_Réseaux](#)

3 Sockets

Pour ce projet, nous avons eu recours aux "sockets". Mais qu'est-ce ?

Un "socket" peut-être associé à un connecteur réseau / une interface de connexion ou encore un flux de données. C'est un identifiant unique représentant une adresse sur le réseau. Des processus (programmes d'une machine) peuvent s'y connecter pour y envoyer des données ou pour en recevoir. Les processus devront adopter un protocole de communication afin d'assurer un échange de données cohérent. Deux grands protocoles de transport existent : TCP et UDP. L'adresse du "socket" est spécifié par le nom de l'hôte sur lequel on le créer et le numéro de port. Dans notre projet, nous avons choisi le protocole TCP afin de gérer la communication entre le serveur et les clients.

Dans ce contexte, le serveur est le processus qui écoute les nouvelles connexions de clients et effectue la récupération des données. Le client est donc le processus qui va tenter de se connecter au serveur et de lui envoyer des données.

4 Cahier des charges

Abordé rapidement en introduction, voici les objectifs fixés en début de projet. Comme nous l'avons évoqué, nous souhaitions créer une salle de chat. Pour donner un petit exemple, nous voulions nous rapprocher de ce que l'outil [Discord](#) propose. Nous sommes alors parti dans l'optique de :

- Autoriser la connexion sur un salon particulier.
- Envoyer des messages à tous les utilisateurs du salon courant.
- Récupérer la liste des utilisateurs connectés.
- Envoyer un message privé à un utilisateur.
- Changer son pseudo "à la volée".

Ensuite, nous avons quelques idées en tête pour la suite si le temps était de notre côté :

- Une petite interface graphique pour rendre l'utilisation plus plaisante.
- Une gestion d'envoi de fichiers.
- Conversation vocale.

5 Le protocole TCP

Détaillons à présent le protocole utilisé par nos "sockets" afin d'établir la connexion entre les clients et le serveur. TCP est un protocole de transport en mode connecté. En effet, lorsqu'un client voudra communiquer avec le serveur (ou l'inverse), une demande d'ouverture de connexion sera émise, des acquittements seront envoyés tout au long de la connexion et enfin une demande de fermeture de connexion conclura l'échange.

Nous pensions au départ mettre en place un protocole UDP dépourvu d'ouverture / fermeture de liaison et d'acquittements. UDP est conçu pour des émissions rapides. Toutefois, celui-ci ne s'assure pas de la bonne réception des messages. Si l'on devait trouver une métaphore pour UDP, ce serait en quelque sorte l'envoi d'un courrier par la poste. Tandis que TCP serait une communication téléphonique.

Nous avons finalement préféré le protocole TCP assurant la bonne livraison des messages. De plus, l'envoi de messages privés entre deux utilisateurs se trouve facilité par rapport à UDP. Il s'avère que les informations sur les clients sont embarquées dans les "sockets" avec le protocole TCP.

6 Détails de la réalisation

Détaillons à présent comment fonctionne notre salon et les fonctions réellement implémentées. Le serveur gère actuellement un salon. Ceci est modélisé par un tableau contenant les "sockets" (représentant les clients connectés). Le serveur peut accueillir un nombre illimité de clients. En effet, si le nombre de clients dépasse la taille du tableau, une réallocation est enclenchée. En réalité, la limite est fixée à la mémoire que l'on peut allouer pour ce tableau. Un second tableau est utilisé afin de sauvegarder les pseudos des utilisateurs. Illustrons cela d'un exemple.

Sockets[] → Tableau des sockets
Pseudos[] → Tableau des pseudos

Sockets[5] contiendra un "socket" et son identifiant pointera vers un indice de pseudos[]. Donc admettons que l'identifiant de Sockets[5] soit 6 alors pour récupérer le pseudo associé, il faudra consulter pseudos[6].

A l'aide de ceci, nous avons mis en place une commande très simple permettant à un client d'afficher la liste des personnes connectées. Il suffit à l'utilisateur de taper `/l`. Le serveur consultera alors sa liste de "sockets" afin de récupérer les "pseudos" pour ensuite les transmettre à l'émetteur de la requête.

Nous proposons deux autres commandes liées aux "pseudos". Le changement de pseudo (via la commande `/n` suivi du nouveau pseudo) et l'envoi de messages privés (commande `/p` suivi de l'utilisateur cible). Pour la première commande, le serveur ira simplement consulter le tableau des "sockets" à la recherche de l'émetteur puis ira trouver le pseudo correspondant dans le tableau des pseudos pour ensuite le modifier. Pour la seconde commande, c'est l'inverse qui est effectué. Le tableau des pseudos est consulté puis la correspondance avec le tableau des "sockets" est faite pour obtenir le "socket" destinataire.

Une autre commande `/h` permet à l'utilisateur d'obtenir la liste des commandes réalisables. Le serveur est alors consulté et c'est lui qui transmet une version à jour de la liste des commandes disponibles.

Enfin, une dernière commande `/q` autorise un client à fermer la connexion avec le serveur. Ce dernier effectuera alors une réallocation de son tableau de "sockets".

Mis à part ces commandes, une petite interface pour le terminal a été ajoutée afin de rendre les échanges plus fluides. La librairie "ncurses" fut utilisée. Nous proposons toutefois deux versions du client. La première avec l'interface et la seconde sans.

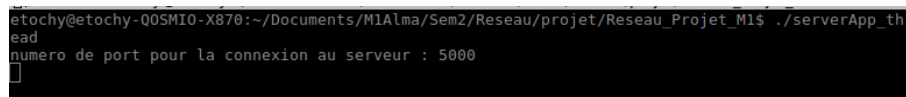
Quant à la taille des messages, nous avons fait le choix de garder les messages des utilisateurs court à l'instar de Twitter (140 caractères). Le dialogue est ainsi concis et percutant. Bien sûr nous avons pris le soin d'empêcher l'utilisateur de pouvoir dépasser cette limite lorsqu'il écrit son message évitant les surprises lors de la validation / l'envoi.

Nous avons utilisé des "Threads" pour la gestion des clients par le serveur. Ainsi, le serveur peut communiquer avec plusieurs clients à la fois. De même pour le client, un "Thread" d'écoute est actif en permanence afin de recevoir les messages. C'est ce qui permet d'envoyer des messages tout en recevant ceux des autres utilisateurs. Pour "ncurses", nous avons eu recours à des forks.

7 Jeux d'essais

Dans cette partie, nous suivrons les possibles étapes d'un envoi de messages.

Dans un premier temps, le serveur doit être lancé. Nous démarrons donc l'exécutable fourni dans l'archive.



```
etochy@etochy-Q0SMIO-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$ ./serverApp_thr
ead
numero de port pour la connexion au serveur : 5000
█
```

FIGURE 1 – *Lancement d'un serveur* .

On constate que le numéro de port par défaut est 5000. On voit également que le serveur est en cours d'exécution.

A présent, tentons de lancer un second serveur afin de voir comment réagit l'application.

```
etochy@etochy-Q0SMIO-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$ ./serverApp_thr
ead
numero de port pour la connexion au serveur : 5000
erreur : impossible de lier la socket a l'adresse de connexion.: Address already in use
etochy@etochy-Q0SMIO-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$
```

FIGURE 2 – *Lancement d'un second serveur .*

On peut voir un message d'erreur nous indiquant qu'un second serveur ne peut être démarré.

Connectons à présent un premier client au serveur. Nous renseignons délibérément une adresse IP erronée.

```
etochy@etochy-Q0SMIO-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$ ./clientApp_text
usage : client <adresse-serveur>: Success
etochy@etochy-Q0SMIO-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$
```

FIGURE 3 – *Lancement d'un client avec une IP erronée .*

Le système nous rappelle à l'ordre avec le bon format de commande pour connecter un client.

Tentons une nouvelle fois avec une adresse valide.



FIGURE 4 – *Lancement d'un client avec une IP correcte .*

C'est un succès, on peut voir l'interface "ncurses" s'afficher.

Voyons à présent ce que nous pouvons effectuer comme commande. Nous entrons donc la commande `"/h"`.



FIGURE 5 – *Demande d'aide* .

Le détail des commandes disponibles apparaît alors. Voyons ce qu'il advient si nous entrons `"/l"`.

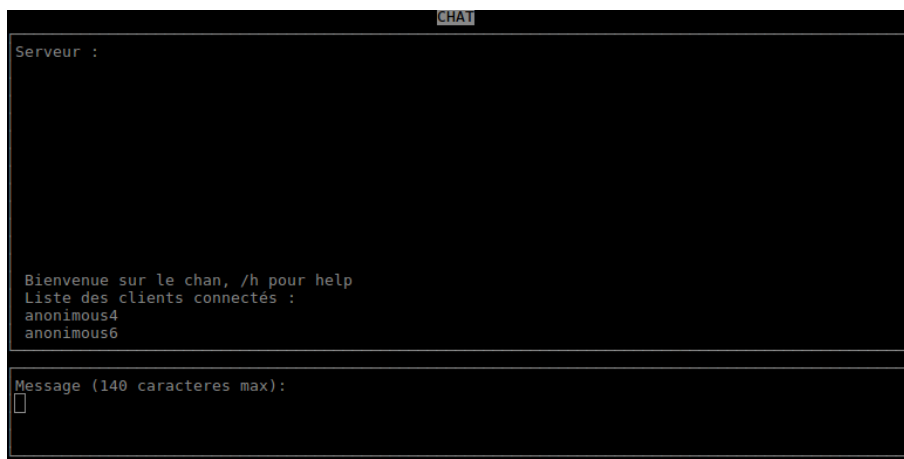


FIGURE 6 – *Utilisateurs connectés* .

On peut voir que seul deux utilisateurs sont connectés. Changeons notre nom à l'aide de la commande `"/n"`.


```
Bienvenue sur le chan, /h pour help
Pseudo modifie avec succes

Message (140 caracteres max):
█
```

FIGURE 7 – *Changement de pseudo* .

Un message nous affirme alors que notre pseudo a bien été modifié.

Tentons d'envoyer un message privé à un utilisateur.

```
Bienvenue sur le chan, /h pour help
Message prive impossible

Message (140 caracteres max):
█
```

FIGURE 8 – *Envoi de message privé avec un utilisateur erroné* .

Mince, il semblerait que le nom de l'utilisateur soit invalide. Réessayons en nous assurant de la validité du pseudo.

```
(prive)anonymou5 : bonjour

Message (140 caracteres max):
█
```

FIGURE 9 – *Envoi de message privé avec utilisateur correct* .

C'est un franc succès ! On peut voir apparaître la mention "(prive)" en provenance de "anonymou5" nous dire "bonjour".

Notons que deux personnes peuvent avoir le même pseudo. Si un message privé est envoyé avec pour argument le pseudo en question alors les deux clients recevront le message.

Notre échange à présent terminé, on ferme la connexion avec la commande `"/q"`, voici ce qui se passe côté client.

```
/q
connexion avec le serveur fermee, fin du programme.
etochy@etochy-Q0SMI0-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$
```

FIGURE 10 – *Fermeture de connexion : Client* .

Un message nous notifie que la connexion avec le serveur est terminée.

Voyons ce qui se passe côté serveur si un client ferme la connexion.

```
etochy@etochy-Q0SMI0-X870:~/Documents/M1Alma/Sem2/Reseau/projet/Reseau_Projet_M1$ ./serverApp_thr
read
numero de port pour la connexion au serveur : 5000
nouveauClient : 4
----- pseudo -----
nouveauClient : 5
list
pseudo : nouveauPseudo
pseudo : anonymous5
quit
nouveauClient : 6
quit
nouveauClient : 5
quit

```

FIGURE 11 – *Fermetures de connexions : Serveur* .

On constate qu'à chaque fermeture de la part d'un client, un message `"quit"` est affiché dans la console du serveur. De même, lorsqu'un client se connecte, un message `"nouveauClient"` avec l'id de celui-ci est affiché.

8 Problèmes réglés depuis la présentation

Depuis la démonstration, nous avons pu régler le problème lié à la demande de liste des clients connectés. Nous avons ajouté un délai côté serveur (`sleep(1)`). Cela a également permis de régler l'affichage des pseudos à la suite sur ncurses.

9 Bugs persistants

Des `"bugs"` subsistent dans la dernière version. On retrouve des soucis quelque peu aléatoire dans l'interface ncurses du client. En effet, lorsqu'un message est envoyé, il se peut que celui-ci soit tronqué ou mélangé avec une autre chaîne de caractères. De même, lorsqu'un message reçu est trop grand

et qu'il nécessite plusieurs lignes pour s'afficher alors ncurses ne sera pas en mesure de l'afficher du moins pas dans sa totalité.

10 Conclusion

Pour conclure, notre application est fonctionnelle et regroupe des fonctionnalités pertinentes. Une interface est disponible mais présente tout de même quelques soucis. Nous avons donc deux versions pour le "client". La première avec l'interface et la seconde sans mais stable. Des améliorations pourraient être faite. Nous pensons notamment à l'ajout de plusieurs salons. Pour le moment un seul est disponible. Il serait aisé d'ajouter des tableaux côté serveur afin de simuler d'autre salons.