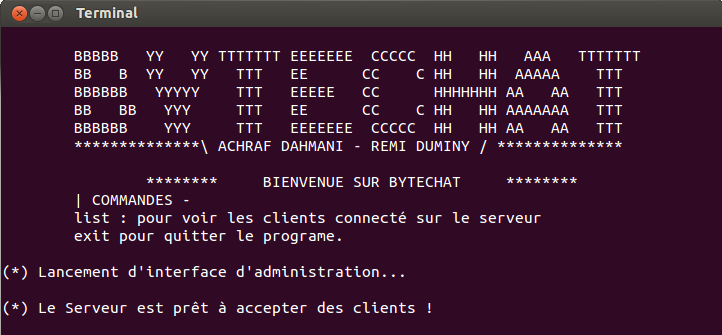
|  |
| --- |
| Middleware et Client/Serveur |
| Achraf DAHMANI, Rémi DUMINY-SAGOT |
|  |
|  |



Sommaire

[Introduction 2](#_Toc410514589)

[Présentation du projet 3](#_Toc410514590)

[1. D’une idée à un projet 3](#_Toc410514591)

[2. Réponse à la problématique du thème 3](#_Toc410514592)

[3. Langage de programmation 3](#_Toc410514593)

[4. Choix du protocole de transmission de données 3](#_Toc410514594)

[Fonctionnement du projet 5](#_Toc410514595)

[1. Diagramme d’utilisation (USE CASE) 5](#_Toc410514596)

[2. Diagramme de flux (FLOW CASE) 6](#_Toc410514597)

[Conclusion 9](#_Toc410514598)

# Introduction

Lors de notre troisième année au sein de l’IG2I, nous nous somme vu proposer la création et l’implémentation d’un projet en la matière « Middleware et Client/Serveur ».

Qu’est-ce qu’un Middleware ?

Un Middleware est dans le cadre de l’informatique, un programme utilisant toutes les couches logicielles et qui permettent à des applications de communiquer à distance. On s’est d’abord intéresser à la problématique que pose un middleware afin d’élabore la meilleure conception possible. Le premier problème qui survient est de pouvoir permettre à un programme de s’exécuter sur plusieurs machines pour autant qu’elles soient reliées à un réseau (local ou internet). Pour se faire, il faut faire interagir plusieurs domaines de l’informatique à savoir :

* Le système d’exploitation
* Langage de programmation
* Choix du protocole de transmission de données

Que veut dire « orienté Client/Serveur » ?

Ceci est un principe de communication fondamental, il se veut synchrone contrairement à son confrère asynchrone qui désigne l’interaction par messagerie («Message-Oriented Middleware»). On trouve le synchronisme dans la communication de type requête/réponse, en effet, pour une requête donnée, le serveur donne une réponse, ce type de requête est donc un appel procédural, le client n’étant pas sur la même machine que le serveur.

Le projet que nous avons donc choisi fut l’élaboration d’un programme de communication entre clients via un serveur que nous avons dénommé « Projet MSN ». Dans la suite de ce rapport, il y sera traité du but du projet, des méthodes mises en place et de comment celui-ci fonctionne. Enfin il y traitera des problèmes rencontrés et de leurs solutions, des améliorations possibles et des pistes de recherches.

# Présentation du projet

### D’une idée à un projet

Lorsque l’on nous a présenté la matière et le projet que nous devrions mener, il nous est apparu évident qu’une des manières les plus basiques de montrer les différents échanges possibles entre un client et un serveur était de créer un système de messagerie instantanée. Seulement, une messagerie instantanée de bas niveau n’est pas une chose des plus facile à réaliser, il a donc fallu conceptualiser cette idée afin de pouvoir monter le projet : Le projet MSN était né.

### Réponse à la problématique du thème

Comme précisé en introduction, la thématique du cours pose plusieurs restrictions, et nous avons dû bien sûr les prendre en compte afin de correspondre aux critères. La première étant le système d’exploitation, elle fut la plus facile. En effet, il n’existe pas énormément de système d’exploitation capable de supporter toutes les librairies que nous comptions utiliser et qui s’adapterait à notre logiciel. De plus, le programme étant au départ un programme de type console, il est vite devenu improbable de le commencer à partir de Window, ainsi on s’est donc fixé comme premier objectif de travailler sur un système libre, soit Linux (et plus précisément, Ubuntu).

### Langage de programmation

Le langage de programmation fut un choix très important. Il permet à la fois au professeur de voir et à l’étudiant de montrer qu’il a compris comment fonctionnait le système d’échange de socket et par la même occasion de pouvoir les manipuler à son gré. Etant donné que les premiers travaux pratiques furent porté sur le langage C et que le professeur a su nous montrer les différents moyens de communications offert par ce langage (et du système d’exploitation). Il est devenu évident que le langage C nous permettrait d’exploiter ce type de transmission qu’est l’échange par socket contrairement à Java par exemple où tout le système de communication est masqué.

### Choix du protocole de transmission de données

Pour que le client puisse communiquer à la fois avec d’autres clients (amis) mais aussi avec le serveur, il faut pouvoir établir un protocole de communication avec celui-ci. On a choisi la méthode de communication ci-dessous.

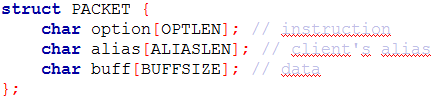
Lorsque le client souhaite envoyer une requête au serveur, il doit la faire de la manière suivante :

**/Requete [Destinataire] [Data]**

Quand le client appuie sur entrée pour envoyer son message, il est découpé en plusieurs parties. La première partie est la requête, on pourra remarquer que la requête est précédée d’un slash évitant tout envoi inopiné en cas d’un appui malheureux sur la touche d’envoi alors qu’aucune commande n’a été faite. La requête permet au serveur de distinguer s’il doit envoyer un message en broadcast aux amis ou à une personne en particulier par exemple. Selon le type de requête, il faut aussi remplir les différentes options qui sont Destinataire et Data. Il existe plusieurs types de requêtes qui sont :

* /send data : Envoi un message à tous les amis connectés du client.
* /whisp dest data : Envoi un message à un ami en particulier du client (ici nommé dest)
* /getami : Envoi une requête serveur afin d’avoir ses amis et de voir qui sont connectés (le client ne peut pas envoyer cette requête, c’est une requête interne).
* /addFriend nom : Demande au serveur d’ajouter une personne dans la liste d’ami du client (il s’agit ici de nom).
* /delFriend nom : Demande au serveur de supprimer une personne de sa liste d’ami (ici nom).
* /quit : permet d’envoyer une requête serveur pour se déconnecter.

Pour pouvoir envoyer les requêtes clairement au serveur et sans ambigüité, nous avons décidé de créer la structure :

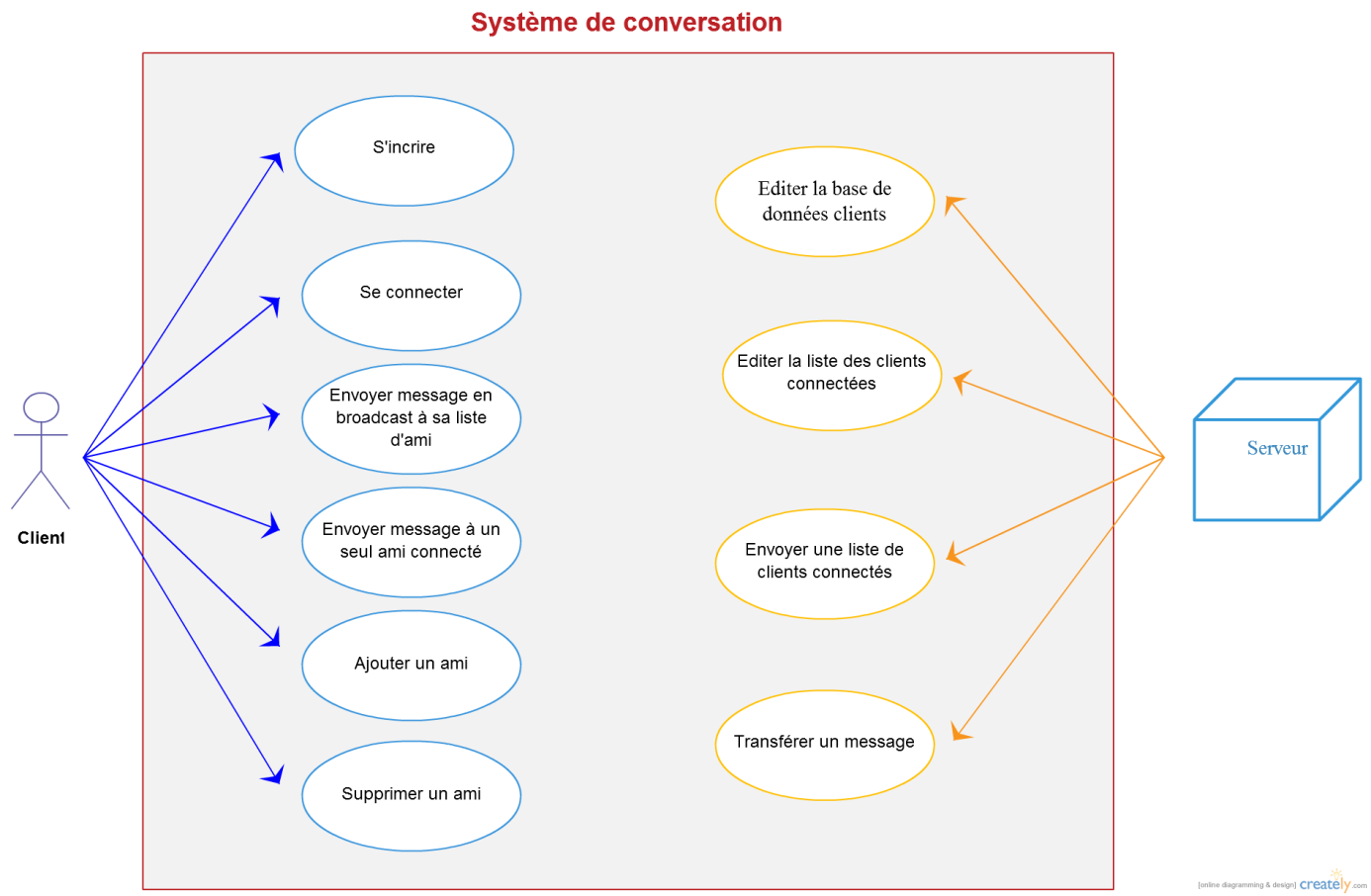


Dans ce sens, la partie « /requete » est retiré de la chaine pour être inséré dans « option ». « alias » est un tableau de caractère contenant le nom de la personne envoyant le paquet. « buff » est un cas particulier. Il peut être vide, ceci est le cas lors d’une requête système, il peut contenir une chaine de caractère représentant le message du client lors d’un envoi en broadcast à sa liste d’ami, il peut aussi contenir à la fois le destinataire et le message, étant chacun séparé par un espace, le séparateur choisi a été l’espace car il n’est pas nécessaire d’en mettre un plus significatif. En effet, un client ne peut avoir son nom contenant des caractères tels que l’espace. Donc le premier mot de « buff » est le client à qui le message doit être adressé. Cependant le client doit être dans la liste d’ami. Enfin il peut simplement contenir le nom d’un client en cas d’ajout ou de suppression d’un ami.

# Fonctionnement du projet

Afin de réaliser un projet sans anicroche, il est préférable de faire une bonne conception auparavant. Nous avons donc pris des dispositions en synthétisant notre logiciel sous différents graphes UML que nous allons vous expliquer.

### Diagramme d’utilisation (USE CASE)



Notre système comprend n client et un unique serveur qui centralise toutes les demandes. Un client peut demander à gérer à un compte au serveur, l’administration sera la plus simple possible. Il s’agira de pouvoir créer un compte, le supprimer et bien entendu, afin d’avoir accès au service, de se connecter. Ensuite le client aura plusieurs possibilités. Dans un premier temps, il pourra envoyer un message à tous ses amis avec une requête de type **send**. Il pourra également chuchoter, c’est-à-dire envoyer un message à un ami spécifiquement à l’aide de la requête de type **whisp** (pour whisper). Enfin il pourra gérer sa liste d’ami avec une suite de commande : **addFriend** afin d’ajouter un ami à sa liste d’ami, le nom de la personne devra être existant dans la base afin de pouvoir l’ajouter. Il pourra également supprimer un ami avec **delFriend**, la personne devra bien évidemment être dans sa liste pour pouvoir le supprimer. Finalement, la commande **getami** qui renverra un tableau d’ami avec une mention de connexion afin que le client sache qui est connecté n’est pas apparente, cette fonction est appelée à chaque connexion, ajout ou suppression d’un ami.

Concernant le serveur, il peut modifier directement la base de données cliente grâce à des requêtes internes (ajout, suppression, modification, affichage). Il peut éditer la liste des clients connectés, c’est-à-dire accepter et/ou supprimer des personnes qui sont connectées sur le serveur (les clients). Enfin il possède deux fonctions dépendant directement d’un client, il s’agit de « l’envoi d’une liste d’ami » et de « transférer un message ». Dans ces deux cas, ces fonctions ne sont pas faites manuellement, mais sous l’action d’une requête cliente, on peut parler d’évènement.

### C:\Users\Duminy Remi\Desktop\IG2I ET CAV\IG2I\Cours\Middleware\Information Systems Help Desk.pngDiagramme de flux (FLOW CASE)

* L’image est attachée avec le compte rendu pour une meilleur résolution -

Ce diagramme de flux expose de façon directe et concrète la manière dont la programmation a été faite. On y voit clairement les différentes procédures et la manière dont le client et le serveur se lie. Dans un premier temps nous parlerons du client et de son fonctionnement, puis du serveur.

#### Le client

Lorsque l’on démarre l’application client, on voit un menu avec les commandes possibles actuellement sur le client. Il s’agit de /launch et /exit. La fenêtre ne s’affiche pas pour l’instant, la toute première commande à effectuer est donc « /launch » afin d’afficher l’interface graphique. Une connexion est établie grâce à un socket avec la commande /connect. On parle ici de socket cliente car elle est créé chez le client, et ce n’est pas lui qui définit l’ip ni le port de connexion. Lorsque celle-ci est effectuée, le client crée deux threads : un thread qui servira à écrire dans le socket pour l’envoyer au serveur et un thread d’écoute pour récupérer ce que le serveur envoie comme données et il analyse les données afin d’afficher les informations à la bonne place et de la bonne façon, par exemple pour différencier un chuchotement d’un message d’acquittement.

L’affichage des données quant à lui s’effectue grâce à la librairie « ncurses », ce qui a posé un problème par la suite et limiter nos améliorations. En effet, nous n’avons appris, que lorsqu’il a finalement été trop tard, que la librairie ncurses ne supportait pas l’implémentation en thread ou en fork pour l’affichage. A cause de cela, nous n’avons donc pas pu développer notre arbre de thread permettant de créer un thread par conversation avec un ami mais nous avons dû opter pour une conversation avec tous les amis connectés avec possibilité d’envoyer un message en particulier à quelqu’un.

Pour écrire des données, il faut suivre le protocole expliqué dans la présentation du projet, en fonction de ce qui est envoyé un type de réponse est fourni. Si c’est un message, soit de broadcast, soit personnel (soit système), il est affiché dans la zone de discussion. Lorsque l’on reçoit un tableau d’ami, il met à jour la zone contenant ledit tableau et enfin, si c’est un acquittement dû à une requête de log out, alors le programme ferme son socket et se termine.

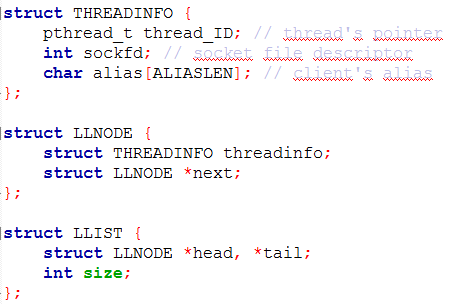
#### Le serveur

Le serveur se lance manuellement. Une fois lancée il lance une boucle infinie servant à la communication et un thread servant à l’administration manuelle.

* Administration manuelle : Ce thread autorise un tiers que nous appellerons « propriétaire » de manipuler le serveur alors qu’il est en fonctionnement. Il peut ainsi gérer chaque client de sa base de données en fonction de la commande entrée. Le plus grand avantage à avoir créé ce système d’administration est que, comparé à certains projets, ce système permet d’arrêter « naturellement » le serveur, sans avoir de problème au niveau de la fermeture des sockets et des problèmes que cela pourrait engendrer (impossibilité de relancer le serveur à cause d’un socket mal fermé par exemple). Cela est implémenté de la manière suivante : on ferme toutes les sockets d’écoute, on arrête tous les threads et enfin on ferme le socket serveur. Le fait de mettre une administration manuelle est aussi un moteur en termes d’améliorations possibles.
* La communication : Un socket, dit serveur car il bind le socket à une adresse ip et à un port de connexion afin d’être l’hôte (le principe du serveur en somme) est créé afin d’écouter et surtout, d’autoriser, les connexions. Il s’agit de la boucle infinie de notre programme, en effet c’est par cette boucle que le principal de l’application se déroule.

Lorsqu’un client souhaite se connecter, la partie écoute du programme se charge d’accepter la connexion. Il crée ainsi presque instantanément un thread qui lui-même créera un socket d’écoute pour pouvoir laisser le socket serveur accepter d’autres clients. Ce socket d’écoute fonctionnera de façon synchrone. En effet, il attend de recevoir un PACKET (structure de contrôle des commandes) du client, le traite et répond à la commande. Cette réponse est sous plusieurs formes. La première est un message d’acquittement lorsqu’il s’agit d’une commande système, un ajout d’ami par exemple. La deuxième est un message d’erreur en cas de mauvaise manipulation du client. Il effectue les demandes du client tant que celui-ci n’envoie pas la commande « /quit » ou que le serveur ne soit arrêté manuellement. Si la commande est « /quit », alors ceci entraîne la fermeture du socket et du thread.

La partie mémoire n’est pas apparente dans le diagramme de flux. Lorsqu’un client se connecte, on alimente une liste chaînée LLIST. Cette liste chaînée est constituée de maillons (ici LLNODE qui contient une structure d’information très intéressante et le maillon suivant). La structure THREADINFO est intéressante car elle contient la référence sur le thread en cours, le numéro du socket d’écoute mais aussi le nom du client. Ainsi on peut intervenir sur les trois plans, à savoir : Le thread qui fait tourner le socket, le socket qui contient les « coordonnées » du client à travers le réseau et qui s’avère fort pratique pour retransmettre des messages ou une réponse et enfin le nom du client qui permet un traitement en base de données plus sécurisée qu’un échange client-serveur.



# Conclusion

Lors de l’élaboration du projet, nous avons pu apprendre comment travaillé avec les sockets. Ce fut aussi l’occasion de mettre en commun toutes les connaissances acquises au cours de ces trois dernières années à l’ig2i dans le cadre de la programmation sur linux et en langage C. En effet, nous avons utilisé du traitement de fichier pour nous en servir en tant que base de données, nous avons utilisé des structures plus ou moins complexes afin de pouvoir faire des connexions et des recherches plus aisément, on a également utilisé des tableaux ou des listes chaînées pour pouvoir soit travailler de façon statique, soit travailler en allocation dynamique. Ensuite, nous avons apporté les dernières connaissances acquises cette année à savoir les threads et les sockets. Enfin, l’utilisation de la librairie ncurses et la conception du projet ont permis à la fois d’élargir notre répertoire mais aussi d’ajouter un plus à notre expérience. Tout cela représente les points positifs à avoir fait ce projet.

Si l’on doit faire une critique rétroactive sur ce qui a été fait, on ne déplorera que le fait d’avoir tardivement rencontré l’erreur dans l’utilisation des threads avec ncurses qui s’est avéré à la fois difficile dans sa mise en place et impossible dans l’implémentation d’une discussion à plusieurs fenêtres. Cependant, ce projet a été terminé dans les délais et cette amélioration n’était qu’un objectif subsidiaire, tous les points critiques ont été traités et aucun bug n’a été relevé.

En conclusion, le projet terminé est bien un produit final et nous espérons que vous l’apprécierez en tant que tel.