Projet de Systèmes distribués / Systèmes d'exploitation
Compte rendu

Lasserre Jean-Baptiste, Loustaunau Benjamin

24 avril 2015

Sommaire

1 Fonctionnement global du système

L'application est composé de 5 programmes :

— Une application client pour les utilisateurs du chat.
— Une application affichant les messages reçus par le client.
— Un serveur de chat écrit en C.
— Un serveur RMI écrit en Java pour la gestion des comptes
— Un client RMI / serveur TCP écrit en Java assurant la communication entre le serveur de chat et celui de gestion des comptes.

1.1 Lancement de l'application

Les deux applications clients sont lancées via un script shell, chacune dans son propre terminal. Elles communiquent entre-elles à l'aide d'un pipe nommé. C'est le client de chat qui interagi avec l'utilisateur et communique avec le serveur. Le programme d'affichage ne fait que lire et afficher sur la sortie standard les messages que le client écrit dans le tube de communication.

Le serveur RMI est lancé directement depuis un terminal et la partie centralisée à l'aide d'un script shell lançant le serveur de chat et le client RMI.

1.2 Communication entre les clients et le serveur de chat

La communication entre les clients et le serveur de chat se fait en deux temps. Tout d'abord, le client doit découvrir le serveur (IP et port) en émettant une requête sur le groupe multicast utilisé par l'application. Le serveur écoute sur sa socket UDP multicast les demandes de découverte et répond - là aussi en multicast - en envoyant les informations requises pour une connexion TCP. Si le serveur ne répond pas à la requête, le client indique l'erreur et tente une nouvelle découverte quand cela est nécessaire. Dès que le serveur est localisé, le client initie une connexion TCP avec lui. Toutes les communications entre le client et le serveur se font via TCP. Le serveur est donc en écoute sur un certain port TCP et à chaque nouvelle connexion, il crée un thread dédié à la communication avec le client et lui fournit la socket de service.

1.3 Communication entre le serveur de chat et le serveur de gestion de compte

Le serveur de gestion de comptes est séparé en deux partie : une effectuant les traitements et une faisant l'intermédiaire avec le serveur de chat.

1.3.1 Serveur de chat ↔ client RMI

La communication entre le serveur de chat et l'intermédiaire (client RMI) se fait via TCP. Les deux programmes étant supposés sur la même machine, il n'y a pas de phase de découverte. On considère qu'ils se connaissent déjà. Au lancement, le client RMI se met en écoute sur sa socket TCP et attend une connexion TCP. Quand la connexion est initié, le client RMI se met alors en attente de requête. Si à un moment donné, la connexion est rompue, le client RMI se remet simplement en attente d'une nouvelle connexion TCP. Quand le serveur de chat est lancé, il initie une connexion TCP avec le client RMI. Si cela échoue le serveur ne fait rien de particulier, il retentera plus tard. Le serveur utilisera cette connexion pour communiquer avec le client RMI à chaque fois qu'il en aura besoin. Si à un moment donné la connexion est perdue, toute requête se soldera par une erreur. Le serveur de chat retentera une connexion à chaque fois qu'il aura une requête pour le serveur de gestion de compte.

1.3.2 Client RMI ↔ serveur RMI

Le client RMI et le serveur RMI communiquent par RMI. Le serveur exporte un objet à son lancement l'ajoute au registry. Il écoute ensuite toutes les connexions sur une socket UDP multicast. À chaque demande de découverte reçue, le serveur envoie en multicast les informations nécessaires au fonctionnement de RMI. Le client RMI doit donc initialement découvrir le serveur avant de débuter la connexion RMI. Si la découverte échoue ou si la connexion est rompue à un moment donné, une nouvelle tentative est effectuée quand nécessaire.

1.4 Résistance aux pannes

Si une des parties Java (ou les deux) ne répond pas, le serveur de chat continue tout de même à fonctionner. Les utilisateurs connectés peuvent continuer à s'échanger des messages. Toutes les demandes concernant les comptes utilisateurs (connexion/ajout/suppression/...) sont indisponibles, on prévient le client de ce problème, la déconnexion est toujours possible. Une nouvelle tentative de connexion aux éléments manquant est effectuée à chaque requête.

1.5 Fonctionnement du client

Le client de chat est composé de deux applications distinctes travaillant ensembles. L'une d'elle est le programme principal du client avec l'interface qui interagi avec l'utilisateur. Elle affiche les menus et s'occupe des saisies de texte. Elle possède également tous les mécanismes de communication avec le serveur. L'autre application est un simple afficheur des messages. La partie principale écrit dans le pipe de communication les messages que l'afficheur doit écrire à l'écran. Cela peut être les messages des autres utilisateurs ou des messages d'information (connexion/déconnexion d'utilisateurs, etc.). Le programme d'affichage ne fait que lire le pipe et afficher le contenu demandé.

1.6 Fonctionnement du serveur de chat

Le serveur de chat est le cœur de l'application. Il écoute en permanence les requêtes de découvertes qui lui parviennent via sa socket UDP multicast. Il est également à l'écoute de toute nouvelle connexion TCP afin de lancer un thread pour la gérer. Le rôle principal du serveur est d'attendre les requêtes des clients afin de les traiter. Il retransmet à tous les clients les messages reçus et au serveur de gestion de compte les requêtes concernant les utilisateurs. En interne, le serveur est décomposé en trois parties coopérant pour assurer le bon fonctionnement du chat. Une partie est dédiée à l'écoute des connexions et requêtes des clients, une autre à l'envoie de messages et enfin une dernière à la gestion des requêtes.

1.7 Fonctionnement du serveur de gestion de compte

Le serveur de gestion de compte permet de créer ou supprimer des utilisateurs et de savoir si un utilisateur et un mot de passe donnés existent. Il permet aussi de bannir/gracier un utilisateur

et de changer le statut d'un utilisateur (admin/normal). Chaque modification entraîne la sauvegarde de la liste des utilisateurs dans un fichier. Une copie de la dernière sauvegarde est effectuée avec la nouvelle sauvegarde, afin de pouvoir rétablir (manuellement) un état cohérent en cas de problème lors de la sauvegarde. Le serveur retourne un code représentant le succès ou non des requêtes qui lui sont demandées. Une valeur différente a été associée à chaque type d'erreur afin de les différencier. Ainsi, le serveur de chat connaît l'erreur ayant empêché la requête et peut tenter de la corriger ou tout simplement informer le client. Le client RMI ne fait que transmettre les informations entre le serveur de chat et le serveur de gestion de compte.

2 Gestion des utilisateurs

L'ajout de compte connecte l'utilisateur et la suppression le déconnecte. Quand un utilisateur se connecte ou se déconnecte, un message est envoyé à tout le monde pour informer les autres utilisateurs. Le serveur ne sauvegarde pas les messages envoyés par les clients, un utilisateur ne peut voir que les messages envoyés depuis sa connexion.

2.1 Différents types de comptes et sanctions

Afin de contrôler les échanges sur le chat, on différencie deux types de comptes utilisateurs. Il y a les simples utilisateurs du chat et les modérateurs/administrateurs. Par défaut, le serveur possède un unique utilisateur ayant les droits d'administrateur. Cet utilisateur est créé à la compilation, son nom et son mot de passe peuvent être modifiés dans un fichier d'en-tête. Cet utilisateur ne peut pas être supprimé et ni perdre ses privilèges. Tous les autres utilisateurs peuvent obtenir ou perdre les privilèges d'administrateur. Les administrateur ont accès à une fonctionnalité supplémentaire (dans le client du chat), ils peuvent bannir temporairement ou définitivement un utilisateur. Cela se fait via une fonction en renseignant le pseudo de l'utilisateur à sanctionner et la durée de la sanction. Si la durée de la peine est nulle, l'utilisateur est simplement averti et déconnecté. Si la durée saisie est négative, il s'agit d'un ban. Sinon, toute durée positive indique le nombre de minutes du ban. Un utilisateur banni est automatiquement déconnecté et ne peut plus se connecter. Les administrateurs peuvent eux aussi être bannis, ils perdent alors leurs privilèges.

Pour matérialiser les privilèges et les peines, trois informations sont ajoutées à chaque utilisateur :

— un booléen indiquant le statut : 0 pour utilisateur et 1 pour administrateur ;
— un booléen indiquant une peine : 1 pour un utilisateur banni ;
— une date java de fin d'une peine, pertinent uniquement lors d'un ban. Une valeur nulle lors d'un ban indique une peine permanente.

2.2 Connexion

Quand un utilisateur tente de se connecter, le serveur de gestion de compte vérifie s'il est banni. Si c'est le cas, il compare la date actuelle à celle de fin de ban pour éventuellement mettre un terme à la peine de l'utilisateur. La connexion n'est acceptée que si l'utilisateur n'est pas banni.

Une fois connecté, un utilisateur est identifié par sa socket. On retient donc l'association utilisateur $\leftrightarrow$ socket. Cela facilitera les échanges par la suite, l'utilisateur n'ayant pas à décliner son identité à chaque message.

3 Serveur de chat

3.1 Description

Le serveur de chat est composé de deux éléments :

— le serveur de chat écrit en C/C++ ;

— le client RMI permettant de gérer les comptes.

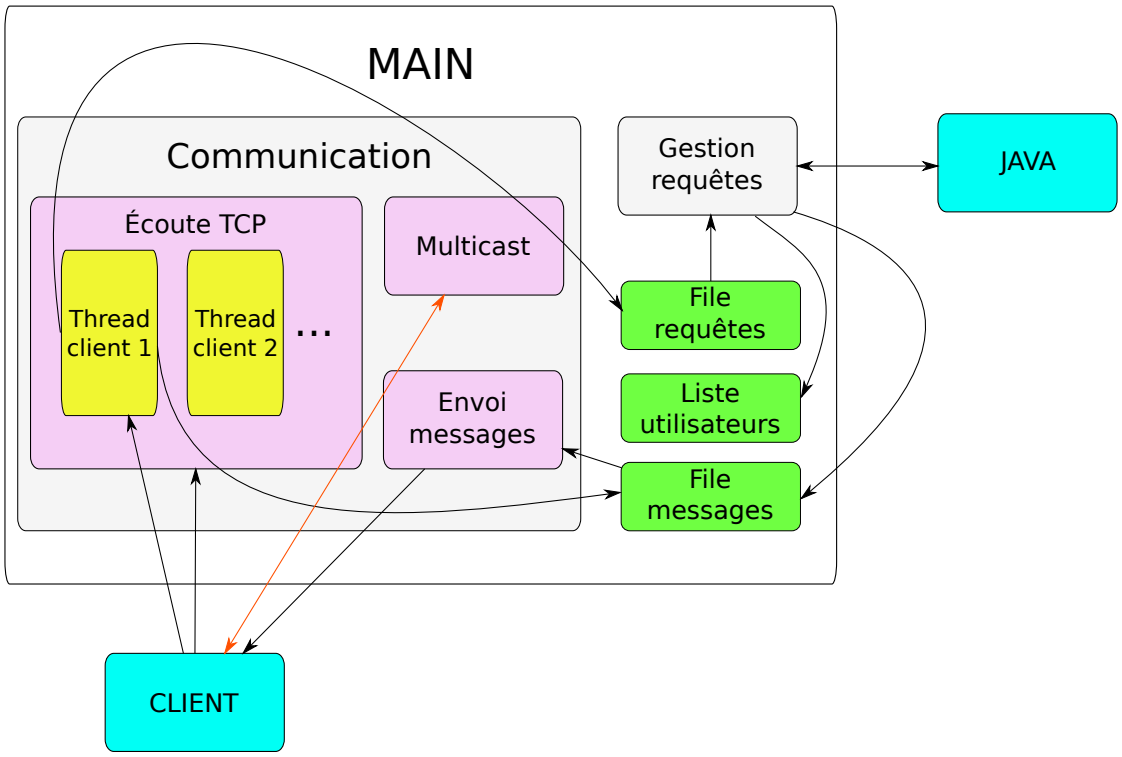
Il fait intervenir plusieurs threads :

— un thread de communication écoutant les données reçues sur la socket UDP multicast et y écrivant lorsque nécessaire ;
— un thread de communication écoutant les connexions sur la socket d'écoute TCP ;
— autant de threads de communication que de clients, chacun utilisant une socket de service TCP pour écouter et décoder les messages envoyés par les clients.
— un thread de gestion des requêtes relatives aux comptes utilisateurs ;
— un thread de communication destiné à envoyer des messages à un ou tous les clients via TCP.

Pour assurer la communication entre ces threads, on utilise les structures de données suivantes (partagées) :

— une file de messages de requêtes : communication unidirectionnelle entre le thread d'écoute et celui de gestion ;
— une file de messages à envoyer : communication unidirectionnelle entre le thread de gestion ou celui d'écoute et le thread d'envoi des messages ;
— une liste des utilisateurs connectés : modifiée par le thread de gestion des requêtes et consultée par les autres.

3.2 Architecture du serveur



Légende :

— en bleu : les éléments externes au serveur
— en vert : les structures de données partagées
— en gris : les threads lancés par le MAIN
— en rose : les threads lancés par Communication
— en jaune : les threads lancés pour chaque client

3.3 Gestion des requêtes

Le thread de gestion des requêtes est en attente de requêtes à traiter (données dans la file de messages de requêtes). Quand une requête arrive, plusieurs cas sont possibles :

— S'il s'agit d'une demande de connexion :
  — Émission d'une requête de connexion au serveur de gestion de compte (via le client RMI).
  — Si la connexion est acceptée, on enregistre l'utilisateur dans la liste des connectés. La socket permet d'identifier le client s'étant connecté à ce compte.
  — Sinon, la connexion est refusée.
— S'il s'agit d'une demande de déconnexion :
  — On retire l'utilisateur de la liste des connectés et on ferme la connexion avec le client.

Une fois la requête traitée, ce thread se met en attente d'une nouvelle requête.

3.4 Gestion des communications clients/serveur sur le serveur

Pour gérer les communications, on utilise un tableau de threads. Quand une communication est terminée (volontairement ou sur erreur), on ne termine pas le thread associé. On le conserve et on le met en attente jusqu'à qu'il y ait une nouvelle connexion TCP dont on peut le charger. Ainsi, lors d'une nouvelle connexion TCP, on cherche en priorité un thread libre avant d'en créer un autre. Pour que la recherche soit rapide on utilise les structures de données suivantes :

— un tableau de threads
— un tableau de sockets
— un tableau de threads libres

La position d'un thread dans le tableau des threads est fixe, elle lui est fournie en paramètre lors de sa création. Le thread numéro i de ce tableau utilise la socket numéro i du tableau de sockets. Le tableau de threads libres contient en réalité les index des threads libre dans le tableau des threads. Ainsi, lors d'une nouvelle connexion, si le tableau de threads libres est vide, on crée un nouveau thread, sinon on prend le premier de ce tableau. L'attente d'un thread prend fin deux cas : une nouvelle socket est disponible ou la variable de fin de programme est à vrai.

Lorsque la connexion client/serveur est perdue suite à une erreur, le thread ferme la socket et attend d'avoir une autre connexion à traiter.

3.5 Arrêt du serveur

Le serveur s'arrête en cas d'erreur irrécupérable ou à la demande de l'utilisateur. Avant l'arrêt du serveur, on envoie à chaque client un message pour les prévenir de l'arrêt, puis on attend 30s pour que les utilisateurs ne soient pas pris de court. Lors de l'arrêt du serveur, on envoie également une requête aux parties Java pour qu'elles s'arrêtent elles aussi. Le programme principal force l'arrêt des threads en fermant leur socket et en leur demandant de se terminer et attend que chacun d'eux se termine proprement. La terminaison des threads est prévue comme suit :

— Mise à vrai de la condition d'arrêt.
— Fermeture forcée de toutes les sockets.
— Chaque thread teste alors la condition d'arrêt et la voyant à vrai, se termine.

4 Client

4.1 Gestion des communications clients/serveur sur le client

Lorsque la connexion client/serveur est perdue suite à une erreur, on affiche un message d'erreur à l'utilisateur et on l'invite à se reconnecter. Si la connexion UDP/TCP au serveur est impossible, on préviens l'utilisateur de l'indisponibilité du serveur.

5 Communications

5.1 Paquets multicast UDP : découverte des serveurs

Pour découvrir le serveur, les clients envoient une requête multicast. Afin d'assurer un maximum de performances, le paquet envoyé ne contient qu'un entier (valeur 0) pour identifier le type de requête. Ainsi, si le serveur de chat est en écoute sur le groupe multicast, il comprend qu'il doit se présenter aux autres et les clients en écoute ignorent simplement le paquet.

La réponse du serveur est composé de deux parties, toutes deux de taille fixe : l'en-tête et les informations de connexion. L'en-tête est là aussi un entier (valeur 1) pour identifier le type de requête. Ainsi, le serveur recevant le paquet l'ignore et les clients en écoute utilisent les informations du paquet pour obtenir les informations nécessaires à l'établissement d'une connexion TCP avec le serveur. Les informations de connexion occupent 8 octets. Les quatre premiers représentent l'adresse IP du serveur et les 4 suivants le numéro de port (un entier) du serveur.

Ce fonctionnement est utilisé pour la découverte du serveur de chat par les client et pour la découverte du serveur RMI par le client RMI.

Paquet UDP pour la découverte :

— 4 octets : type de message (0)

Paquet UDP pour la présentation :

— 4 octets : type de message (0)
— 4 octets : adresse IP
— 4 octets : port

5.2 Paquet TCP

5.2.1 Paquets client → serveur

Les paquets TCP envoyés par les clients au serveur de chat sont composés de deux parties : l'en-tête de taille fixe et le corps de message de taille variable.

L'en-tête contient le code de la requête (un entier). Les requêtes possibles pour les client sont les suivantes :

— connexion
— déconnexion
— création compte
— suppression compte
— envoi d'un message aux autres utilisateurs
— liste des utilisateurs connectés
— kick/ban : uniquement pour les admins
— modifier les privilèges : uniquement pour les admins
— arrêt du serveur : uniquement pour les admins

Le corps du message est de taille variable, son contenu dépend du type de requête.

— Connexion :
  — En-tête : 8 octets :
    — taille nom utilisateur
    — taille mot de passe
  — Corps du message :
    — nom d'utilisateur
    — mot de passe

— Déconnexion : pas de corps

— Création de compte :

    - En-tête : 8 octets :
        - taille nom utilisateur
        - taille mot de passe
    - Corps du message :
        - nom d'utilisateur
        - mot de passe

- Suppression de compte :
    - En-tête : 8 octets :
        - taille nom utilisateur
        - taille mot de passe
    - Corps du message :
        - nom d'utilisateur
        - mot de passe

- Envoi d'un message aux autres utilisateurs :
    - En-tête : 4 octets : taille message
    - Corps du message : message

- kick/ban :
    - En-tête : 4 octets :
        - durée du ban (un entier long)
        - taille nom utilisateur à kick/ban
    - Corps du message :
        - nom d'utilisateur

- modifications droits :
    - En-tête : 4 octets :
        - booléen (vrai si admin)
        - taille nom utilisateur
    - Corps du message :
        - nom d'utilisateur

- liste des utilisateurs connectés : pas de corps

- arrêt du serveur : pas de corps

5.2.2 Paquets serveur → client

La réponse du serveur TCP est elle aussi variable. Tous les envois sont gérés par le thread d'envoi de messages. Quand un thread gérant la communication avec un client reçoit des données sur sa socket TCP, il les décode. S'il s'agit d'une requête, on envoie le code de retour. S'il s'agit d'un message, il doit être envoyés aux autres utilisateurs. Pour cela, le thread crée un message contenant :

- le code du message (pour distinguer plusieurs types de messages)
- la date du message (timestamp UNIX)
- la taille du nom d'utilisateur de l'expéditeur
- la taille du message envoyé
- le nom d'utilisateur de l'expéditeur
- le message envoyé par l'utilisateur

Ce message est ensuite stocké dans la file des messages à envoyer et le thread d'envoi réveillé. Ce dernier est constamment en attente de données à envoyer. Dès que la file des messages à envoyer contient un message, il se charge de l'envoyer au(x) destinataire(x), sans modifier le message. Ce thread est donc capable d'envoyer tout type de messages.

L'envoi d'un message se fait simplement en parcourant la liste des sockets et en envoyant le message sur chaque. On assure donc que tous les messages sont reçus dans l'ordre et avec un horodatage cohérent, l'heure étant donnée par le serveur. S'il y a un destinataire précis pour le message, le message n'est envoyé qu'à ce dernier. Ensuite, le thread se remet en attente d'un autre message à envoyer.

S'il y a un problème lors de l'envoi (perte de connexion TCP), on passe simplement à la socket suivante. Le thread de communication avec le client est lui aussi en erreur dans cette situation et se

gère de fermer proprement la socket.

5.2.3 Paquets serveur → RMI

Les paquets sont constitués d'une en-tête contenant le code du message et la taille des données et des données elles-mêmes.

Les données sont :
- Connexion :
  - taille nom utilisateur
  - taille mot de passe
  - nom d'utilisateur
  - mot de passe
- Création de compte :
  - taille nom utilisateur
  - taille mot de passe
  - nom d'utilisateur
  - mot de passe
- Suppression de compte :
  - taille nom utilisateur
  - nom d'utilisateur
- kick/ban :
  - durée du ban (un entier long)
  - taille nom utilisateur à kick/ban
  - nom d'utilisateur
- modifications droits :
  - booléen (vrai si admin)
  - taille nom utilisateur
  - nom d'utilisateur
- arrêt du serveur : pas de corps

5.2.4 Paquets RMI → serveur

Le serveur de gestion de compte envoie un simple entier (code de retour) au serveur de chat.

6 Le serveur de gestion de compte

La gestion de compte est écrite en Java, elle est composé d'un serveur RMI et d'un client RMI. Le serveur RMI gère les comptes utilisateurs : création, suppression, sauvegarde de la liste des utilisateurs dans un fichier et permet également de vérifier si le nom d'utilisateur et le mot de passe donnés correspondent bien à un utilisateur inscrit (pour la connexion). Le serveur RMI charge la liste des utilisateurs au chargement (lecture du fichier) et la sauvegarde à chaque modification (réécriture du fichier). Toutes les fonctionnalités offertes par le serveur sont appelées par le client RMI. Ce dernier sert uniquement à faire le lien entre le serveur de chat écrit en C et le serveur de gestion de compte écrit en Java. Le client RMI est donc également un serveur TCP. Il est en attente d'une connexion TCP du serveur de chat, il ne gère qu'une connexion à la fois, nous n'utilisons pas de threads pour ce serveur. Une fois une connexion TCP établie, le client RMI se met en écoute sur la socket de service associée à la connexion, en attente d'une requête de la part du serveur de chat. Dès que des données sont présentes, elles sont décodées afin d'appeler la fonction adéquate sur le serveur. On propose les fonctionnalités suivantes :

- connexion d'un utilisateur
- suppression d'un utilisateur

— inscription d'un utilisateur
— sanction d'un utilisateur (kick/ban)
— modification des privilèges
— arrêt du serveur

Dans les trois premiers cas, le client RMI appelle une méthode distante sur le serveur RMI puis renvoie via TCP un code indiquant le succès ou l'échec de la requête. Si le serveur RMI est injoignable, le client RMI retourne immédiatement une erreur au serveur de chat et en profite pour tenter de se reconnecter au serveur. Si la connexion réussie, les prochaines requêtes pourront alors être traitées. Si la connexion entre le serveur de chat et le client RMI est rompue, le client RMI se remet en attente d'une connexion TCP. Le serveur de chat et le client RMI étant sur la même machine, on suppose qu'ils se connaissant à l'avance (ip et numéro de port). Le client RMI découvre le serveur RMI à l'aide d'une requête multicast, un groupe multicast dédié à la partie gestion compte est utlisé pour ne pas interférer avec le groupe multicast utilisé par le serveur de chat et ses clients.

Sur le serveur de chat, si la connexion avec le client RMI est rompue, toutes les requêtes des utilisateurs faisant appel au serveur de gestion de compte renvoient immédiatement une erreur au client. Il est prévenu de l'indisponibilité du serveur de gestion de compte. À chaque nouvelle requête faisant appel au serveur de gestion de compte, une nouvelle tentative de connexion est effectuée. Si elle réussie, les prochaines requêtes pourront alors être traitées.

7 Problèmes rencontrés

Nous avons voulu faire une conception globale plutôt que minimaliste, complexifiant le problème et rallongeant la durée du projet. Ceci dans le but de gérer l'entièreté des cas ou à minima la majeure partie d'entre-eux et ce, dès le début. Après l'élaboration de nos structures, nous avons commencé en parallèle la partie RMI et l'architecture de base du serveur de chat. Nous avons poursuivi avec la gestion des requêtes sur le serveur de chat et l'affichage sur le client. Cette mauvaise organisation nous a fait rencontrer un problème majeur : la communication entre les différentes parties n'était pas fonctionnelle et les jeux de test inefficaces. Afin de régler ce problème, nous avons décrit l'ensemble des messages, ainsi que leur structures, pour relier nos différents modules. Une deuxième erreur est alors apparue : tous nos buffers étaient de type (void *). Nous n'avons toujours pas compris pourquoi mais cela causait des pertes de données dans nos sockets. À ce stade là, nous avions des modules fonctionnels indépendamment mais aucune transmission cohérente et correcte des données entre-eux. Il a alors fallu débugger nos modules, nos transmission, nos données ainsi que nos requêtes en même temps. Ces deux gros problème en ont causé un dernier : le temps. Pris de cours sur le débuggage, nous avons donc fait le choix de n'assurer qu'une seule requête parcourant correctement l'ensemble de nos modules. Nous pouvons, à l'heure actuelle, nous connecter avec le client en tant qu'utilisateur, pour peu que le nom et le mot de passe soient valides. Les autres requêtes sont normalement valides mais nous n'avons pas pu traiter les différents retours : le serveur plante lors de l'envoi de la réponse au client.