Compte rendu projet FAR

Ayoub MOUJANE - Pierre PERRIN - Julien WIEGANDT

SPRINT 3 :

VERSION FACILE :

VERSION 1 :

Protocole de communication :

Nous avons décidé de rester sur une communication en mode connecté comme les sprints précédent étant donné que c'est le protocole de communication le plus fiable surtout lorsqu'il s'agit d'assurer les transferts de fichiers. Lors de ce sprint le fait d'éviter toutes pertes de données lors des échanges était très important puisque cela pouvait endommager le fichier et celui-ci ne pouvait donc pas être lu par le client récepteur.
Nous avons repris les mêmes bases que pour la deuxième version du sprint précédant. La gestion de l'envoi de messages quelconque et la déconnexion d'un client est donc la même. En plus de cela, un client a la possibilité de taper 2 commandes : « /filename » et « /file ». Si le client envoie « /filename », cela va lui afficher tous les fichiers pouvant être transférés (qui se trouvent dans un dossier prévu à cet effet : filetosend). Le client peut décider d'envoyer un de ces fichiers en envoyant ensuite « /file ». Cela permet donc de créer un thread pour gérer l'envoi du fichier : il doit ensuite rentrer le nom du fichier qu'il veut envoyer. Suivant la taille du fichier, l'envoi se fait par paquets, tout comme la réception. Le fichier sera d'abord envoyé au serveur qui le stockera temporairement dans un dossier prévu à cet effet (filetosubmit). Une fois que le serveur a reçu le fichier dans sa totalité, il le transmet aux autres clients de la même manière. Les clients reçoivent donc le fichier un par un, grâce à un thread dédié à la réception, et le stocke dans un dossier (filereceived). Chaque client possède donc deux répertoires : un contenant les fichiers à envoyer, et l'autre contenant les fichiers reçus. Une fois que le serveur a transféré le fichier correctement à tous les autres clients, il le supprime. Dans le cas où un client n'a pas reçu le fichier correctement (s'il n'a pas reçu le nombre d'octets auquel il s'attendait), celui-ci envoie un message au serveur pour qu'il puisse le renvoyer avant de le supprimer.
Voir Annexe (page 8) pour le diagramme de séquence.

Difficultés rencontrées :

La première difficulté rencontrée a été le fait de pouvoir afficher les fichiers pouvant être transférer dans un terminal à part. La commande *system("gnome-terminal")* donnée par le professeur ne marchait pas sur mac car gnome-terminal n'existe pas sur mac. Nous avons donc choisi simplement d'afficher les fichiers sur le même terminal pour éviter les erreurs liées aux différences d'environnement.

Ensuite nous avons rencontrés d'autres difficultés concernant l'envoi du fichier. Le fichier n'était pas envoyé dans sa totalité ce qui impliqué que du côté du thread de réception le programme bloquait dans la boucle *while((recv_size < size_file)) car* recv_size (variable contenant le nombre d'octet reçu) n'était jamais égal à size_file (variable contenant le nombre d'octets attendu) et le recv() dans la boucle attendait à l'infini. Pour remédier à cela il a fallu utiliser une structure *timespec* nous permettant d'utiliser la fonction nanosleep() qui suspend l'exécution du thread appelant jusqu'à ce que soit le temps indiqué ait expiré. Cela nous a donc permis d'avoir une fréquence définit pour l'envoi des paquets ce qui à réglé le problème.

Un autre problème à été de supprimer le fichier du côté serveur. En effet pour éviter de prendre trop de place dans notre application il ne fallait pas garder le fichier une fois qu'il ne servait plus. Le problème n'était pas la suppression en elle-même mais plutôt le moment où on devait supprimer le fichier. Celui-ci devait être supprimer après que tous les clients aient reçu le fichier et il fallait aussi prévoir le cas d'un mauvais envoi et le renvoyer directement au client concerné avant qu'il soit supprimer. Il fallait donc que le serveur soit au courant de la bonne ou mauvaise réception du fichier par le client.
Dans le cas d'une bonne réception, au début nous avons fait en sorte de mettre en place un envoi de message « ok » au serveur, à la fin du thread de réception du client pour dire que la réception était ok. Or cela causé des problèmes dont nous ne sommes pas parvenus à comprendre la cause. Nous avons donc tenté une autre approche et nous avons décidé de mettre le code pour la suppression du fichier juste après la boucle while qui permettait d'envoyé le fichier à tous les clients. Nous avons placé juste avant ce code de suppression une autre boucle while bloquante pour gérer le cas d'un renvoi de fichier et éviter la suppression avant le renvoi. Ainsi dans le cas d'un renvoi de fichier une variable est mise à false pour bloquer la suppression du fichier et on attendait donc que le fichier soit renvoyé en entier pour mettre la variable à true. Le fichier pouvait donc être supprimé.
Dans le cas d'une mauvaise réception comme nous l'avons énoncé dans un problème précédant, le programme bloquait dans la boucle *while((recv_size < size_file)* du thread de reception. Nous avons trouvé une solution à cela en cherchant sur internet : nous avons utilisé une structure timeval pour mettre en place un timer ce qui nous permettait d'exécuté le protocole de renvoi du message en envoyant « FILE_ERROR » une fois le temps (que nous définissions) expiré.
Lien de la page ayant aidé :
(https://stackoverflow.com/questions/15445207/sending-image-jpeg-through-socket-in-c-linux)

VERSION PLUS COMPLIQUEE :

Le serveur sert d'annuaire les transferts de fichier sont réalisés par peer-to-peer

Protocole de communication :

Comme pour le Sprint 1 et 2 nous avons décidé de mettre en place une communication en mode connecté (TCP) pour assurer que les données échangées ne se perdent pas sur le réseau. Le fonctionnement des messages n'a pas changé par rapport à la v2, la nouveauté dans cette version est qu'un client peut désormais transférer des fichiers aux autres clients présents sur le chat.

Chaque client possède un dossier **/filetosend** qui contient les fichiers qu'il peut transférer, les fichiers qu'il reçoit d'autre clients sont enregistrés dans le dossier **/filereceived**.

Pour envoyer un fichier un client envoi « file » dans le chat, à son écran s'affiche les fichiers présents dans son dossier **/filetosend**, il peut ensuite entrer le nom d'un fichier, tant qu'il n'a pas entré le nom d'un fichier présent dans le dossier on lui redemande d'entrer le nom d'un fichier.

Une fois un nom valide entré :

- Le client « passe en mode serveur », il attend la connexion d'autre clients pour transférer le fichier choisi,
    - **V1 : une fois un client connecté il commence à lui envoyer le fichier.**
    - **V2 : une fois un client connecté il créer un thread d'envoi du fichier au client qui vient de se connecter.**
- Le serveur reçoit l'information : le client « envoyeur » est prêt à partager des fichiers. Le serveur transmet aux autres clients l'adresse ip et le port du client « envoyeur ».

Les clients recevant les données d'un client « envoyeur » se connectent automatiquement à ce dernier et sont prêt à recevoir le fichier.
Voir Annexe (page 9) pour le diagramme de séquence.

Difficultés rencontrées :

Considérations sur les formats des données :
Nous nous sommes rendu compte que l'ajout de fonctionnalités pouvait être difficile si elle n'était pas prévue à l'avance. Pour le transfert des données du client « envoyeur » par exemple il fallait envoyer l'ip et le port de ce client à tous les autres clients. Une question s'est posée directement, comment différencier ce message d'un message normal ? Car avec ce message d'information on veut faire quelque chose différent de simplement l'afficher. Nous avons trouvé une solution sommaire qui consiste à voir si le message commençait par « 1 », c'est de cette manière que l'on reconnaît si le message reçu est un message ou bien de l'information. Cette solution posait problème car si un client avait un pseudo commençant par un 1 et qu'il envoyait bonjour le client recevait « 1pseudo : message » et donc on lançait le transfert de fichier. Pour solutionner cela, nous avons mis un espace au début de chaque message pour que la confusion soit impossible.
Nous savons que cette solution n'est pas la plus propre qu'il soit nous avons pensé à une meilleure solution qui serait étiqueter les informations circulantes entre client et serveur c'est-à-dire formater les informations pour les reconnaître facilement (Type#...) ainsi pour les messages le format serait #message#pseudo#texte et pour le transfert d'informations de client #peer#ip#port. Cette manière est plus propre ainsi on reconnaît de manière fiable les informations reçues mais il aurait fallu reprendre tout notre système.

L'utilisation de données formatées :
Une autre problématique a été l'exploitation des données reçues, comment bien les récolter. Imaginons qu'un client reçoit une adresse ip et un port (exemple : 127.0.0.1#30482#), il faut récupérer ces données. Pour cela nous avons créé une fonction qui récupère les différents champs mais une autre question a été de comment stocker le résultat et nous avons opté pour la création d'un struct qui stock l'adresse ip et le port ainsi, on passe en argument de la fonction le struct et le texte reçu et on obtient le struct avec les bonnes informations à l'intérieur.

La problématique de la création du serveur peer :
Ce fut une des difficultés les plus dur à solutionner de par le côté technique de la chose, en effet un client qui envoie un fichier passe en mode serveur et il fallait du coup que le serveur transmette le port choisit par le client serveur et qu'il le transmette aux autres clients. Il fallait que ce port soit défini ou déduit par le serveur car ne nous pouvons pas nous permettre que le client devienne serveur puis qu'ensuite il envoi au serveur le port affecté et qu'ensuite le serveur transmette les infos, cela est beaucoup trop laborieux. Ainsi pour solutionner cela, nous avons choisi une solution simple qui consiste à dire qu'un client sur un port N est en mesure de devenir serveur sur le port N+1000, de cette manière si un client envoi file au serveur le serveur n'a qu'à simplement envoyer aux autres clients le port de ce client (qu'il connaît déjà grâce au tableau de socket) + 1000.
Il fallait aussi du coup que le client sache à quel port il était affecté, nous ne savions pas du tout comment faire avec l'aide de M. TIBERMACINE qui nous fournit la page suivante : https://stackoverflow.com/questions/2360304/get-the-client-port-in-c-after-a-call-to-connect, nous étions en mesure de récupérer le port et d'ainsi créer le serveur à ce port + 1000.

Problématique liée de réception des informations dans le thread lié au faut que les variables utilisées soient des pointeurs :

Au départ ce problème était très bloquant car nous n'y pensions pas directement mais au fur et à mesure problème était facilement repérable, il s'agit du problème lié au fait qu'on ait une boucle qui accepte des clients et après chaque accept() on lance un thread avec comme argument l'adresse d'une structure donnant des informations concernant le client venant d'être accepté cependant juste après le lancement de ce thread on acceptait un nouveau client et du coup les données dans la structure concernaient le client suivant et du coup dans le thread créer on récupérait les mauvaises données, il a donc fallu ajouter un timesleep pour que la thread ait le temps de récupérer les données du bon client et ensuite on accepte un autre client.

Compiler et exécuter le code :
Attention il est important de placer les exécutables dans les bons dossiers.

Pour compiler :

Serveur : « *gcc -o serveur serveur.c* »
Client : « *gcc -o client client.c* »

Pour placer au bon endroit :

« cp serveur.c serv »
« cp client.c client1 »
« cp client.c client2 »
« cp client.c client3 »
« cp client.c client4 »
« cp client.c client5 »

Créer les ressources nécessaires :

Chaque dossier client doit posséder un dossier filetosend et un dossier filereceived donc dans chaque dossier client taper les commandes suivantes :
« Mkdir filetosend »
« Mkdir filereceived »
(Les dossiers sont déjà placés pour éviter les erreurs de nom de dossier)

Pour exécuter :

Il faut d'abord lancer le serveur, ensuite les clients.
Serveur : *« ./serveur »*

Pour chaque client se déplacer dans le dossier clientN
Client : *« ./client »*

Répartition du travail :

Version facile :

	Gestion d'erreur dans la réception du fichier	Réception fichier	Envoi fichier	Diagramme séquence
Ayoub Moujane				
Pierre Perrin				
Julien Wiegandt				

Version compliquée :

	Création serveur peer	Transmission des informations peer	Réception fichier	Envoi fichier	Diagramme séquence
Ayoub Moujane					
Pierre Perrin					
Julien Wiegandt					

Annexe :

Diagramme de séquence uml de la version facile :

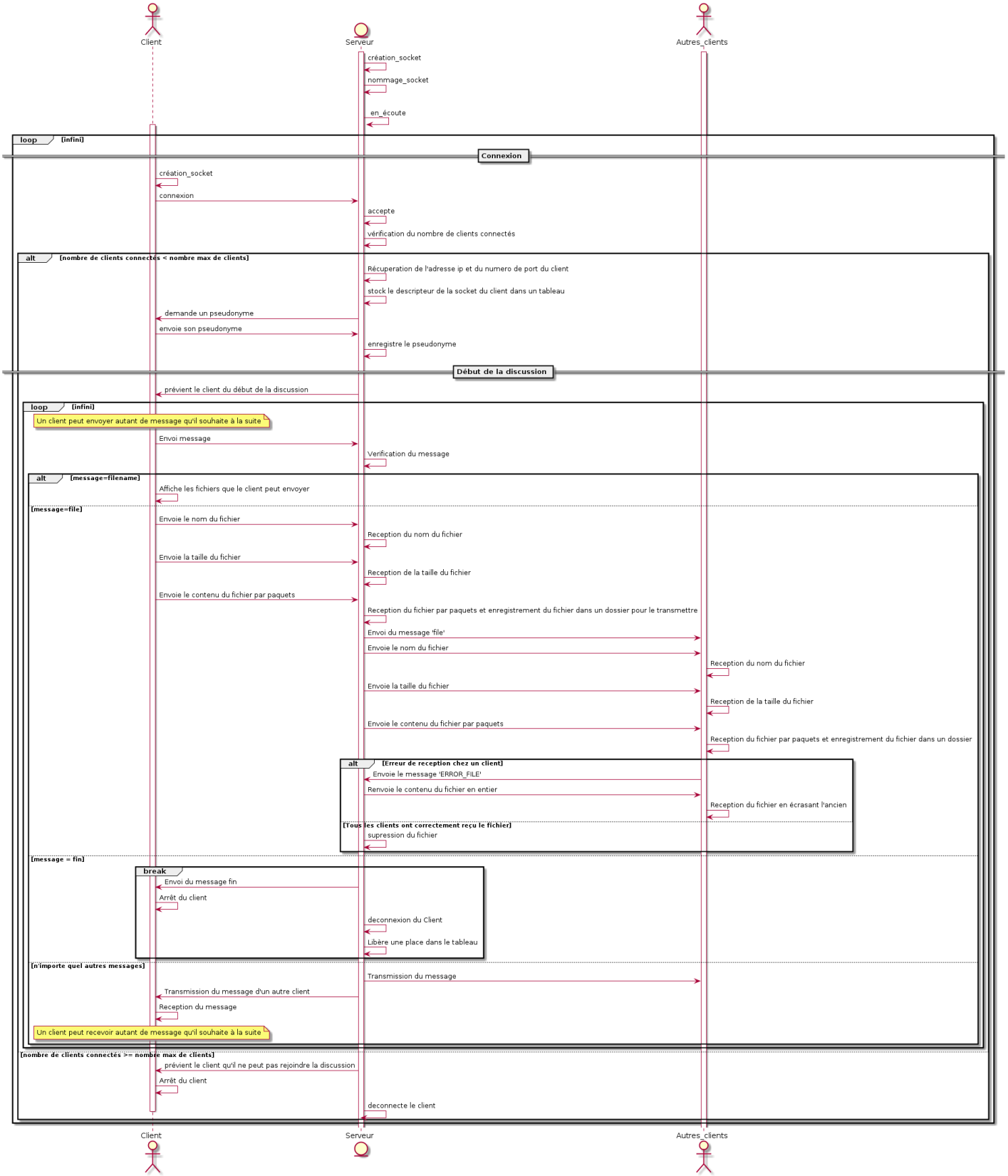

Diagramme de séquence uml de la version plus compliquée :