Compte rendu projet FAR

Ayoub MOUJANE - Pierre PERRIN - Julien WIEGANDT

SPRINT 2 :

VERSION 1 :

Protocole de communication :

Comme pour le Sprint 1 nous avons décidé de mettre en place une communication en mode connecté (TCP) pour assurer que les messages échangés ne se perdent pas sur le réseau.
Lorsqu'on lance le serveur, celui-ci attend la connexion de deux clients. Une fois connectés, une conversation démarre entre les clients, mais cette fois-ci chacun peut envoyer autant de messages qu'il souhaite à la suite, sans être bloqué après chaque envoi en attendant une réponse. De plus n'importe quel client peut commencer la discussion, il n'y a pas d'ordre.
Un message envoyé par un client passe d'abord par le serveur qui le transmet ensuite au client destinataire. La déconnexion se fait lorsqu'un des clients envoie « fin ». Lorsque cela arrive les deux sont déconnectés et le serveur attend la connexion de deux nouveaux clients pour pouvoir relancer une conversation.
Voir Annexe (page 5) pour le diagramme de séquence.

Difficultés rencontrées :

Plusieurs difficultés ont été rencontrées. Tout d'abord nous avons eu un problème au niveau de l'envoi des messages qui venait en fait de la fonction fgets(). Celle-ci permettait de stocker un message entré par le client dans le buffer mais elle laissait un « \n » à la fin du message. Cela avait donc pour conséquence que le message n'était pas délimité et par défaut il avait pour taille celle du buffer. On ne pouvait donc par reconnaitre le message « fin » et mettre fin à la discussion. Pour résoudre ce problème on a dû faire une fonction qui parcours le buffer contenant le message et dès que l'on trouve un « \n » on le remplace par un « \0 » pour ainsi permettre de délimiter le message.

Une autre difficulté a été de gérer l'arrêt des threads coté client pour que, une fois que le mot « fin » est envoyé, cela arrête les programmes clients. Le problème était que dans le code nous lancions d'abords les deux threads d'envoi et de réception (avec 2 pthread_create()) puis on attendait que les deux threads se terminent (avec 2 pthread_join()). Dans ce cas les deux clients bloquaient lorsque « fin » était envoyé puisque pour chaque client soit le thread d'envoi se terminait soit le thread de réception (suivant qui avait envoyé le mot fin), mais jamais les deux. Pour résoudre cela, nous avons décidé que lorsqu'un client envoyait le mot fin le serveur le recevait et le renvoyait aux deux clients pour que chacun puisse le recevoir et mettre fin au thread de réception. Nous avions donc plus qu'à attendre que le thread de réception se termine uniquement pour mettre fin au programme client.

Répartition du travail :

	Initialisation Client	Communication Client (2 threads)	Initialisation Serveur	Communication Serveur (2 threads)	Diagramme séquence
Ayoub Moujane					
Pierre Perrin					
Julien Wiegandt					

Compiler et exécuter le code :

Pour compiler :

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

Pour exécuter :

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

VERSION 2 :

Protocole de communication :

Pour les mêmes raisons que la version 1 nous avons décidé de rester sur une communication en mode connecté. Cette fois-ci plusieurs clients (plus de 2) peuvent se connecter au serveur. Nous lançons donc le serveur en premier et celui-ci attend en permanence la connexion des clients. Lorsque le serveur reçoit une demande de connexion, il vérifie d'abord si le nombre maximum de client n'est pas atteint puis il accepte. Si jamais il n'y a plus de place dans le salon, le serveur accepte la demande juste pour pouvoir prévenir le client qu'il n'y a plus de place puis il le déconnecte. Lorsqu'un client se connecte et qu'il y a de la place disponible, le serveur lui demande un pseudonyme et il le stocke dans une variable contenue dans le thread (si le pseudonyme choisit est « fin » le client est déconnecté du serveur). De même un numéro est associé à chaque client. L'envoi et la réception des messages est géré de la même manière que dans la version 1. La déconnexion se fait lorsqu'un client envoie « fin » : seul le client qui émet le message se déconnecte et le serveur libère donc la place.
Voir Annexe (page 6) pour le diagramme de séquence.

Difficultés rencontrées :

Nous avions dû gérer plusieurs problèmes car il nous était demandé de ne pas modifier le programme client utilisé dans la version 1. Par exemple dans la v1 les clients ne choisissaient pas de pseudo, donc si le client envoyait un message c'était forcément pour communiquer avec les autres, du coup si le message « fin » était entré c'était forcément pour se déconnecter donc nous avons choisi de déconnecter le client lorsqu'il envoyait le message « fin ». Ici, un client passe par l'étape choisir son pseudo donc, le problème était qu'un client pouvait envoyer « fin » pour choisir cela comme pseudo du coup il était déconnecté mais le serveur ne le considérait pas comme une déconnexion du coup pour régler ce problème nous avons choisit de considérer cela comme une déconnexion donc on ne peut pas choisir le pseudo « fin ».

Un autre contre-temps fut la gestion du cas où un client rejoignait un serveur étant plein, nous avons choisi du coup de lui envoyer un message d'information et de le déconnecter, pour déconnecter le client il faut que le serveur lui envoi le message « fin ». Deux messages étaient envoyés l'un après l'autre et cela ne marchait pas, le client recevait bien le premier message mais pas le deuxième et n'était pas déconnecté. Cela a été réglé par un sleep() car nous nous doutions que cela était lié au fait que les deux messages étaient envoyés l'un après l'autre quasiment instantanément, or côté client il y a des traitements comme la vérification que le message reçu est « fin » ou non.

Un autre problème assez banal fut le fait qu'on passe l'index de la socket dans le tableau de sockets au thread client côté serveur (c'est le thread qui reçoit les messages d'un client et envoi les messages de ce client aux autres), le passage se faisait par référence donc quand l'index était modifié dans le main, l'index dans le thread ne correspondait plus au client pour lequel le thread était créé. Pour régler ce problème nous avons utilisé un struct qui stockait l'index qui était réaffecté au même struct du thread pour stocker de manière fixe l'index.

Répartition du travail :

	Initialisation Serveur	Communication Serveur (thread)	Diagramme séquence
Ayoub Moujane			
Pierre Perrin			
Julien Wiegandt			

Compiler et exécuter le code :

Pour compiler :

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

Pour exécuter :

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

Annexe :

Diagramme de séquence uml de la version 1 :

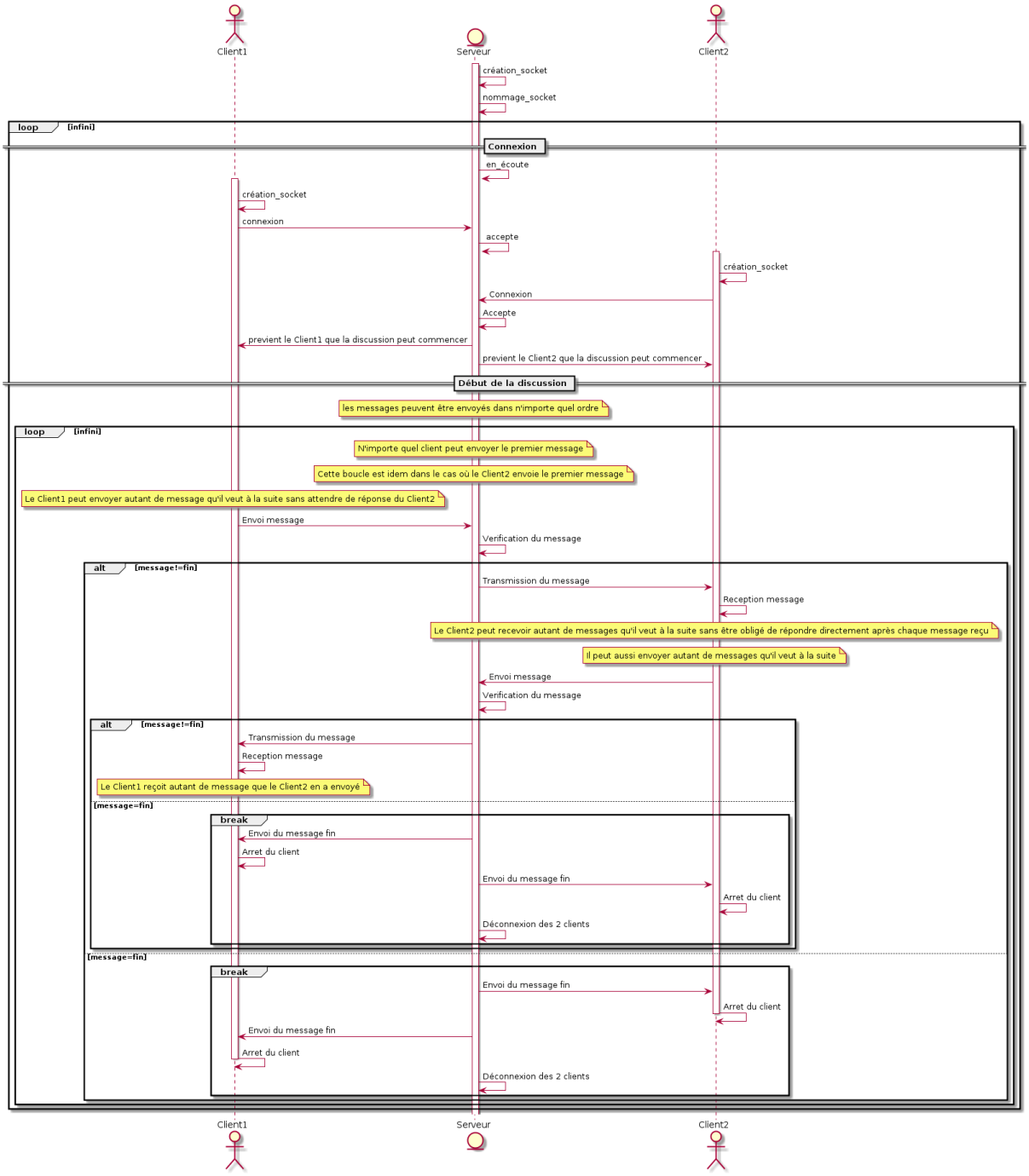

Diagramme de séquence uml de la version 2 :