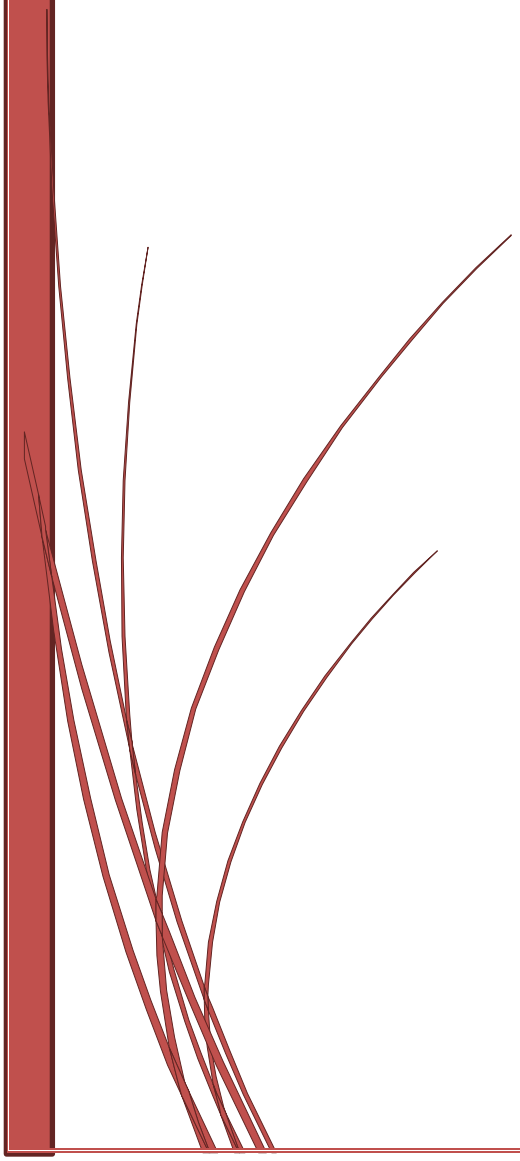
13/04/2019

Projet FAS

Itération 1

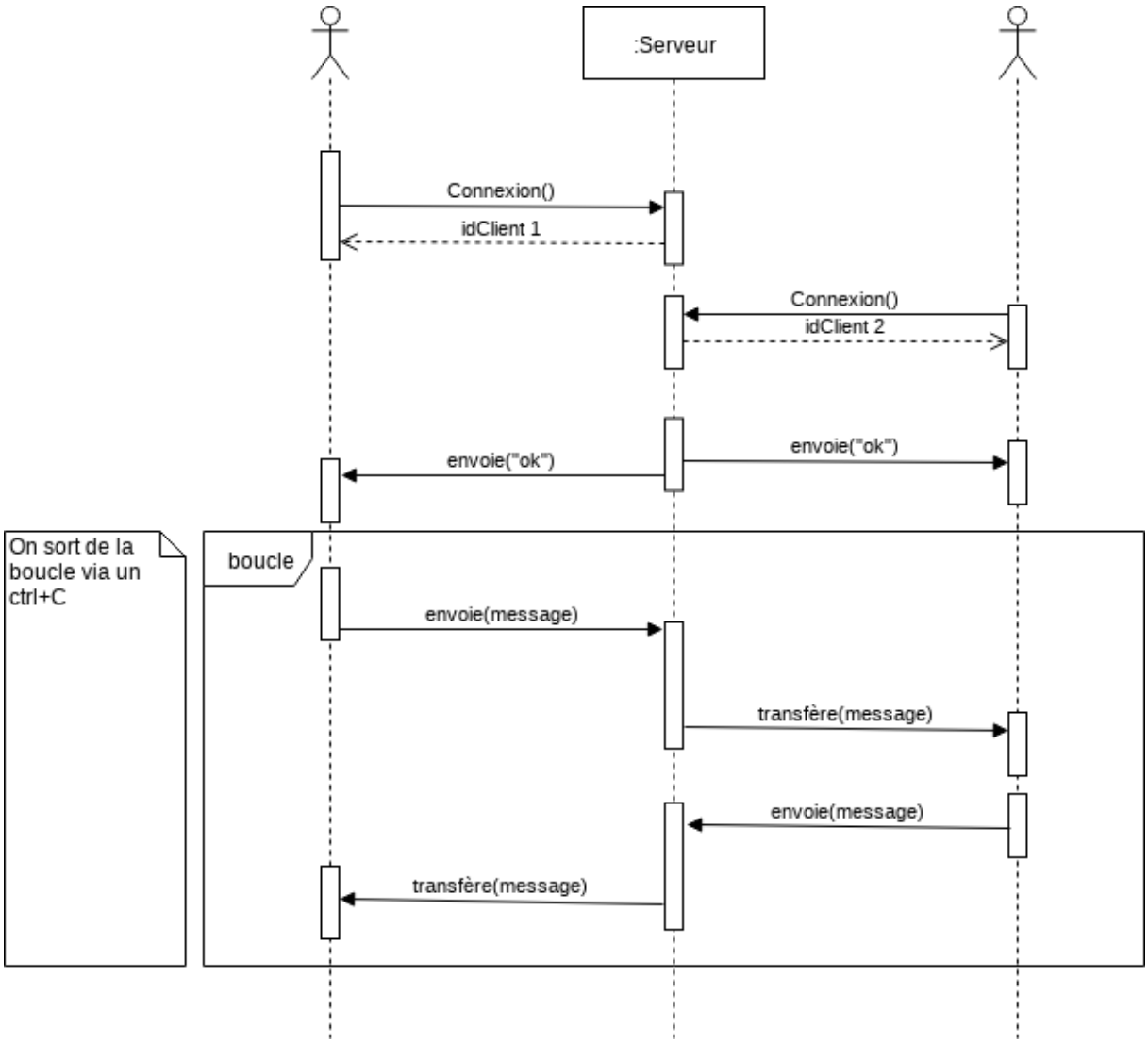


Lauren UNQUERA
Etienne SAIMOND

SOMMAIRE

I – Diagramme de séquence UML

Voici le diagramme de séquence décrivant le protocole de communication entre le(s) client(s) et le serveur.



II – Compilation et exécution du code

But : Créer une messagerie instantanée codée en C et basée sur le protocole Socket

Itération 1 :

Un serveur relaie des messages textuels entre deux clients (livrable 1)

Il doit y avoir 1 programme serveur et 1 programme client. Ce dernier devant être lancé deux fois (deux processus distincts).

1 seul processus/thread serveur doit gérer les 2 clients,

qui envoient leurs messages à tour de rôle (client 1 : write puis read, et client 2 : read puis write)

L'échange de messages s'arrête lorsque l'un des clients envoie le message « fin ».

Ceci n'arrête pas le serveur, qui peut attendre la connexion d'autres clients.

Délais : Semaines du 1er et du 8 avril

Pour compiler les fichier :

A l'aide du bash, tapez la commande suivante :

make

(Nous avons réalisé un makefile afin de pouvoir compiler rapidement, efficacement et simplement).

Pour lancer le serveur :

A l'aide du bash, tapez la commande suivante :

bash launch_server.sh

Pour lancer le client :

A l'aide du bash, tapez la commande suivante :

bash launch_client.sh

Déroulement :

1. Le Serveur attend une connexion.
2. Le Client 1 se connecte.
3. Le serveur envoie au Client 1 son numéro.
4. Le Client 1 attend de recevoir son role du serveur.
5. Le Serveur attend une autre connexion.
6. Le Client 2 se connecte.
7. Le serveur envoie au Client 2 son numéro.
8. Le Client 2 attend de recevoir son role du serveur.
9. Le Serveur envoie au Client 1 "emi" (Mode émission).
10. Le Serveur envoie au Client 2 "rec" (Mode réception).

*** Debut de boucle**

11. Le client 1 envoie son message au serveur et se met en mode reception.
12. Le Serveur le recoit et le retransmet au Client 2.
13. Le client 2 recoit le message du client 1.
14. Le client 2 envoie son message au serveur et se met en mode reception.
15. Le Serveur le recoit et le retransmet au Client 1.
16. Le client 1 recoit le message du client 2.

*** Fin de Boucle**

Arrêt du programme client :

1. Si un client envoie "fin" au serveur, il ferme ses port et le programme se termine.
2. Le serveur va lui fermer le socket du client concerné, et attendre la connexion d'un autre client.
3. Un nouveau client se connecte.
4. Le serveur envoie au nouveau client le numéro du client précedemment deconnecté.
5. Le serveur envoie au nouveau client "emi" (Mode reception).

6. Le serveur envoie aussi au nouveau client le dernier message envoyé au client precedemment deconnecté.

7. Le client va alors entrer dans son fonctionnement normal.

Fermeture du serveur :

1. Si le serveur est fermé via CTRL+C, il envoie aux deux client "exit" et ferme ses socket.

2. Les deux client à la reception du message "exit" vont fermer leurs socket

Sources :

- [Guide pour la programmation réseaux de Beej's](http://vidalc.chez.com/lf/socket.html)

- [Les sockets en C de developpez](https://broux.developpez.com/articles/c/sockets/#L3-2-1-c)

III – Difficultés du projet

1) Difficultés rencontrées

Une des difficultés qu'on a rencontrées a été de gérer la déconnexion d'un des clients tout en maintenant le bon fonctionnement du serveur et de l'autre client. Afin de surmonter cette difficulté, nous avons implémenté la notion de rôle pour la partie client. Il existe ainsi 2 rôles (émetteur, noté « emi », et récepteur, noté « rec »). L'émetteur est le client qui est en phase d'envoyer un message tandis que le récepteur désigne le client lorsqu'il est en phase de recevoir un message. De plus, nous avons fait en sorte que lorsqu'un client se reconnecte, il reçoit le dernier message qui lui a été envoyé afin d'éviter toute perte.

2) Evolution

Lorsque le client est en phase de réception, il ne peut pas se déconnecter car il n'est pas en état d'envoyer le message « fin » au serveur. Ceci n'est pas un problème en soit, mais une possible amélioration. Cela est dû au fait que nous n'utilisons pas plusieurs threads et ce sera traité lors d'une prochaine itération.

IV – Répartition du travail et organisation

Concernant la charge de travail, nous voulons la répartir équitablement et de manière qu'on soit le plus efficace possible. Pour ce faire, nous avons décidé que Lauren se chargerait du code pour la partie client et qu'Etienne se chargerait du code pour la partie serveur pour une première base.

Néanmoins, afin d'être plus performant et d'améliorer au mieux notre travail, nous mettons en commun nos différentes production et nous apportons des modifications, lorsqu'elles sont discutées et validées, sur le travail de l'autre. En effet, n'ayant pas les mêmes capacités ou la même vision des choses, nous jugeons bon de relire le travail de l'autre afin de perfectionner le code.

De plus, avec la mise en place d'une communication régulière et d'une bonne entente, nous pouvons nous entraider assez aisément, et cela a été très utile, que ce soit d'un côté ou de l'autre.