Projet Réseau

INSA CVL - Groupe 5 - Théo Guidoux

Vue d'ensemble du projet

Dans l'ensemble, j'ai beaucoup appris pendant ce projet. Notamment sur le fonctionnement et l'utilisation des méthodes de réseau en C. La gestion d'équipe s'est faite naturellement par tous les membres de l'équipe, répartition des tâches etc.

Grandes étapes

1. Réflexion sur le sujet

Avant de se lancer dans le code, nous avons réfléchi à quatre sur le sujet qui, pour nous, n'était pas très facile à s'approprier du fait que nous ne comprenions pas la finalité. Nous avons donc fait le choix de d'abord faire fonctionner en local (en utilisant aucun aspect du réseau) les aspects primordiaux du programme : attach, detach etc. Nous avons compris certaines choses notamment la manière d'identifier les noeuds pour représenter leur parenté ou encore les structures à utiliser.

Après ceci, j'ai fait une base saine qu'on utilisera par la suite (avec, cette fois-ci du réseau). J'ai ensuite expliqué cette base à mon équipe pour qu'ils puissent avancer dans leurs travaux respectifs.

2. Structure de base

La base que j'ai fait est suffisamment permissive et ouverte pour que nous n'ayons plus qu'à y ajouter des contraintes pour respecter le cahier des charges. Ceci permet d'éviter de revoir la totalité du code pour rajouter une fonctionnalité. Nous avons donc, dans le programme principal, un accès aux listes des parents et enfants, une initialisation des composants principaux (notamment pour récupérer le port et créer l'extrémité réseau), puis la mise sur écoute des événements.

Je n'avais pas encore compris qu'on pouvait mettre le descripteur d'entrée standard sur écoute au même titre que les descripteurs réseaux. J'avais donc fait, au début, un passage d'arguments aux processus qui permettait de lancer une connexion attach() par exemple.

La structure de base contient au final : le programme principal, les listes chaînées pour pouvoir avoir connaissance de la structure de l'arbre, les structures de groupe, contrôler les nouveaux clients et les différencier des clients déjà connectés, simplification et intégration de la structure utilisée avec les FD_SET(), et la fonction attach qui permet d'attacher un processus à un autre en local (en utilisant uniquement les ports, sans IP).

3. Structure des *groups*

Le but de la structure des groupes était de créer un genre de carte d'identité de chaque processus et se partager cette carte d'identité avec les parents et les fils. Les listes des parents et des enfants de chaque processus sont une liste de carte d'identité. Cette carte d'identité contient deux informations : le port et le descripteur qui permet au processus courant de contacter le processus distant. On a donc chaque carte d'identité locale qui dépend du processus distant et du processus local. Les adresses IPs ont ensuite été rajoutées sur la carte d'identité. Le processus courant possède aussi une carte d'identité qui est initialisée au lancement et c'est celle-ci qui est utilisée pour identifier les émetteurs de paquets. Cette structure de carte d'identité est complétée par les *payloads*.

4. Structure des *payloads*

Nous avons choisi de structurer de manière identique tous ce qui envoyé et reçu par notre programme, c'est-à-dire que c'est une structure de taille fixe qui est envoyée. Ce *'payload'* qui est envoyé possède suffisamment d'information pour savoir quel type de message est transporté, l'émetteur, le récepteur et le message à passer. Ainsi, pour envoyer un nouveau type de message, nous n'avons plus qu'à créer un nouveau type et non pas un nouveau paquet complet.

J'ai d'abord voulu envoyer des pointeurs vers les *groups* dans les structures mais je me suis vite rendu compte que c'était quelque chose d'incompatible puisque les pointeurs dépendent de la machine. La structure payload contient alors le port/ip d'envoi, le port/ip de réception, le type du payload qui est un entier statique et le message à envoyer.

5. Fonction attach()

La première version de attach() permet de lier un processus à un autre pour qu'il puisse échanger des messages. Il s'agit en fait simplement de créer un contact entre les processus de manière à ce qu'ils échangent leur identifiant (leur port car suffisant en local). Pour le père, le port du fils dans la liste des fils et pour le fils, le port du père dans la liste des parents.

La première implémentation de attach() faisait des acquittements de paquets pour représenter au mieux un échange : {émetteur 1 : envoie du msg à client 1}, {client 1 : j'ai bien reçu le msg à émetteur 1}. Après discussion avec l'équipe, nous avons supprimé les acquittements car ceux-ci sont implémentés de base dans TCP que nous utilisons.

Pour l'intégration des IPs faite par Clément Foissard, nous avons simplement rajouté les IPs dans les identifications des processus lors de l'échange de données.

6. Makefile

Le makefile utilisé est classique, il permet de compiler le programme, d'abord les fichiers objets puis ensuite le binaire, supprimer tous les fichiers objets et le binaire (clean) et d'installer le binaire sur l'ordinateur, c'est-à-dire le copier dans /usr/bin.

7. Fonctions utiles/Parser

J'ai développé toutes les fonctions qui permettent de réduire notre code, notamment des fonctions qui permettent de chercher un *group* parmi une liste de *groups* (liste des parents, listes de fils…), cela nous permet de déterminer si un client est nouveau ou s'il a déjà été connecté. Parmis ces fonctions, des fonctions qui cherchent un *group* en fonction de son port/ip/descripteur, une fonction qui détermine si un *group* est une racine ou non etc.

Une fonction qui a été particulièrement compliquée à faire : le parseur de commande. Nous envoyons des commandes de ce type : /msg 12345 0.0.0.0 "msg à envoyer", nous voulions une fonction qui nous retourne quelque chose semblable à argc et argv. La difficulté première était de déterminer où couper les arguments. La première idée fut de couper aux espaces, mais j'ai dû rajouter une règle pour permettre de lire correctement les paramètres passés entre guillemets pour pouvoir avoir des arguments contenant des espaces. Ainsi, pour la commande /msg 12345 0.0.0.0 "msg à envoyer", nous avons quatres arguments qui sont "/msg", "12345", "0.0.0.0" et "msg à envoyer".

8. Fermeture propre des clients

Pendant un long moment, nous fermions les clients en envoyant un signal SIGSTOP au client et les parents/enfants recevaient un paquet TCP disant que le client s'est arrêté. Nous devions donc déterminer, parmi toutes les connexions d'un processus, lequel vient de se déconnecter. Une manière plus propre de déconnecter un client est d'envoyer une commande detach() à toutes les connexions (parents et enfants) pour ensuite terminer le programme proprement. C'est donc ce que j'ai fait, en créant une fonction d'exit custom qui remplace la fonction exit() de base du programme. J'ai également rajouté des fonctionnalités pour que l'arrêt soit plus naturel lors des manipulations.

9. Fonction Exit

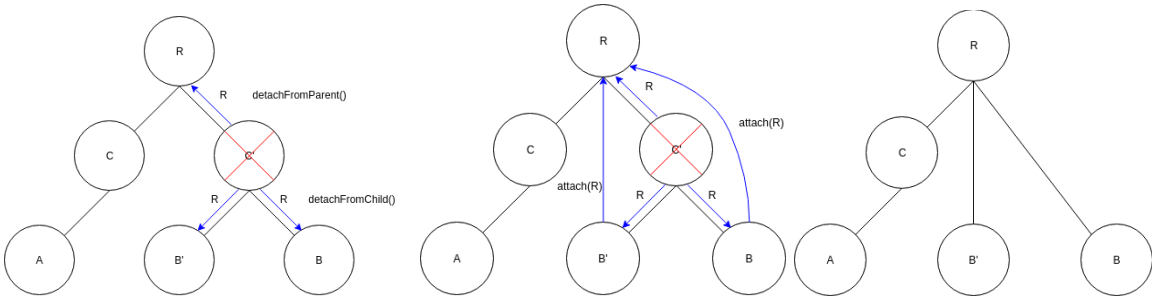
Pour naturaliser le protocole d'arrêt d'un processus j'ai installé un gestionnaire de signal sur SIGINT qui nous permet de quitter correctement l'arbre des processus. J'ai également créé une commande qui nous permet de quitter : /exit.

Dans un premier temps, les fils du *group* sont détachés de l'arbre et le fils reste détachés tandis que la liste des fils des parents est simplement mise à jour avec le *group* détaché en moins. Mais nous avons ensuite mis au point une fonctionnalité d'auto-reattach qui permet à tous les fils du *group* de se rattacher aux parents du *group* détaché.

La fonction /exit est bien différente de la fonction /detach car /detach permet uniquement de se déconnecter d'un seul *group* (parent ou fils) et /exit se détache de tous les fils et parents avant de terminer le processus (et rendre la main).

10. Fonction auto-reattach

La fonction auto-reattach permet, lorsqu'un *group* sort de l'arbre, à tous ses fils de rester dans l'arbre et se reconnectant à tous ses parents. J'ai dû créer deux nouvelles fonctions qui font la différence entre deux types de déconnexion : le détachement d'un parent detachFromParent() et le détachement d'un fils detachFromChild(). Cela est nécessaire du fait que c'est la même fonction qui gère le détachement des *groups*. On avait besoin de différencier d'où le détachement provient. Dans les deux cas, le processus qui se détache envoi la liste de ses parents aux processus. Si le processus qui reçoit le detach est un fils du processus quittant alors il se connecte à tous les parents envoyés.

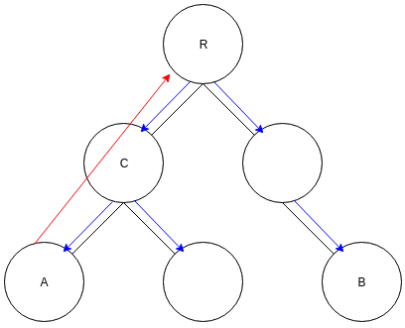


11. Messages

Les messages sont une part très importante de projet. Ils permettent tout simplement l'utilisation de la structure en arbre. Nous avons eu beaucoup de mal à comprendre le fonctionnement des messages voulus. Nous avons donc fait 3 versions.

Version 1:

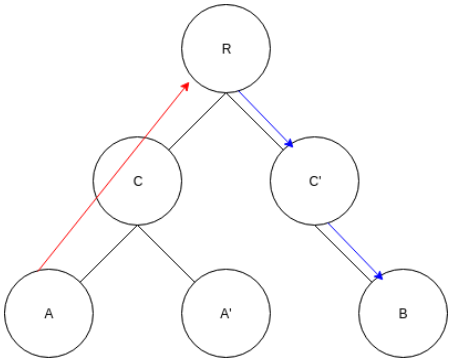
La version 1 permettait tout simplement d'envoyer des messages à un *group*. On avait alors le message qui remonte jusqu'à la racine de l'arbre et qui redescend dans l'arbre, à la manière d'un broadcast, jusqu'à ce que le group voulu soit atteint.



Sur ce schéma, A envoie un message à B, le message remonte alors jusqu'à R en passant par C pour qu'il soit ensuite dispatché à tous les fils etc. jusqu'à atteindre B.

Version 2:

La version 2 est une amélioration de la version 1 que nous avons fait après l'implémentation de la liste de routing (expliqué plus bas). Cette amélioration consiste à envoyer le message à un fils uniquement si le fils est le receveur ou final ou s'il possède le receveur final parmis ses fils (en utilisant la liste des fils présents dans le sous-arbre).

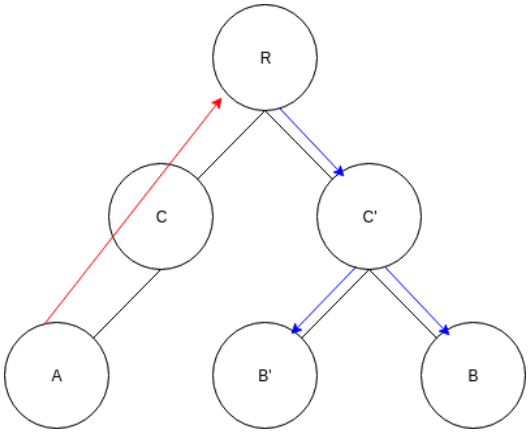


Sur ce schéma, A envoie un message à B. Le message remonte alors jusqu'à R en passant par C. Il est ensuite envoyé uniquement à C' car parmi C et C', il n'y a uniquement C' qui contient B parmi son sous-arbre.

Nous nous sommes malheureusement rendu compte que ce fonctionnement n'était pas celui escompté et que nous devions rajouter une fonctionnalité en plus sur la version 2 pour avoir le fonctionnement des messages escompté.

Version 3:

La dernière fonctionnalité que nous avons rajoutée est la suivante : une fois que le *group* a été trouvé (avec la version 2), on envoie un genre de broadcast à tous ses fils. De cette manière, la connexion point à point que nous avions aux versions précédentes devient un envoi de messages à un groupes de processus. D'un point de vue fonctionnalité, en prenant A : un élève, C : la promo MRI 2021 et C' : la promo STI 2021, on a désormais un élève de MRI qui peut envoyer un message à l'ensemble de la promo STI 2021 (comme le fonctionnement des listes de mails par exemple).



12. Noms

Pour des raisons de facilités d'utilisation, nous avons décidé de nommer les groupes. C'est-à-dire qu'au lieu d'envoyer un message à 192.168.0.100:5000, on enverra un message à 'INSA' par exemple. Pour cela nous avons dû rajouter un argument au lancement du programme permettant de spécifier le nom du processus sur le réseau. J'ai eu pas mal de difficultés à implémenter cette fonctionnalité puisque nous ne l'avions pas prévu au début de notre développement. L'idée derrière cette fonctionnalité est de créer un clone de chaque commande qui fonctionne et de remplacer le classique ip:port par le nom. Chaque nouvelle commande retrouve l'ip et le port et appelle la commande de base qui correspond.

Dans la version actuelle, nous ne pouvons envoyer des messages en utilisant les noms qu'aux parents du processus courant. En effet, pour pouvoir retrouver l'adresse ip et le port du groupe correspondant au nom, le fait d'envoyer un message à un groupe supérieur au parent implique le fait d'avoir conscience des groupes supérieurs. Cependant, avec la fonction message classique (en donnant l'ip et le port), le problème n'existe pas.

On a le même problème avec la fonction /attach. En effet, un processus qui vient d'être lancé ne connaît pas du tout son environnement (son arbre) puisqu'il n'est connecté à personne. Nous devons spécifier obligatoirement l'adresse ip et le port du parent pour la commande /attach.

13. Threads

Dans une première version de notre programme nous avions pensé aux threads pour gérer de manière plus fluide les requêtes. L'idée était de gérer en arrière-plan les requêtes qui arrivent et de gérer au premier plan les requêtes que nous souhaitons envoyer. Ainsi, nous aurions envoyé /attach sur l'entrée standard et le group qui reçoit gère cette requête de manière transparente en arrière-plan. Cela aurait également permis de gérer l'envoi/réception de plus gros paquets sans bloquer l'envoi sur le client qui reçoit.

J'ai implémenté une grosse partie des threads dans le programme mais après discussion avec les encadrants, nous avons opté pour une solution plus simple à mettre en oeuvre : mettre la lecture de l'entrée standard sur le même thread que celui des clients.

L'idée reste néanmoins en tête pour une version future (très éloignée !).

14. Cas panne processus

La panne d'un processus peut être assimilée à deux cas :

- La machine qui s'arrête brusquement
- Un arrêt soudain de l'application sans que l'utilisateur ait choisi de quitter

Dans les deux cas, le programme envoie un dernier paquet de fin de vie à toutes les connexions. C'est une fonctionnalité de TCP que nous n'avons pas implémentée mais que nous avons découvert avec wireshark. Nous avons donc décidé de mettre sur écoute ce paquet. J'ai implémenté cette fonctionnalité sur le programme en ajoutant un cas dans les paquets qui arrivent. On détermine alors quel descripteur est coupé et avec le descripteur trouvé, on peut déterminer quel *group* s'est éteint subitement et de ce fait l'enlever proprement de la liste des parents ou enfants et couper l'écoute sur ce descripteur.

15. Liste de routage

La liste de routage est une liste que possèdent tous les processus. Elle contient tous les processus de son sous-arbre, c'est-à-dire les fils du processus, les fils des fils du processus, les fils des fils de fils du processus etc. C'est une fonctionnalité très intéressante pour déterminer si un processus fait partie d'un groupe. Elle nous sert notamment à éviter les boucles dans le réseau et à réorienter les paquets de manière plus intelligente lorsque la racine envoie le message.

L'implémentation de ce genre de liste est très intéressante à faire car il faut, en plus de chaque attach() d'un processus fils au processus du courant, mettre au courant tous les parents du processus courant du nouveau fils. Pour régler ce problème j'ai mis en place un système de 'notification' d'un nouvelle attach(). Ainsi, à chaque attach() d'un processus B à un processus A, le processus A prévient tous ses parents (en remontant récursivement jusqu'à la racine de l'arbre) que le processus B s'est attaché et peut donc mettre à jour sa liste de routage. La liste de routage de la racine contient donc tous les processus. Le même principe est appliqué lorsqu'un groupe se détache en envoyant une notification de detach().

16. Contraintes

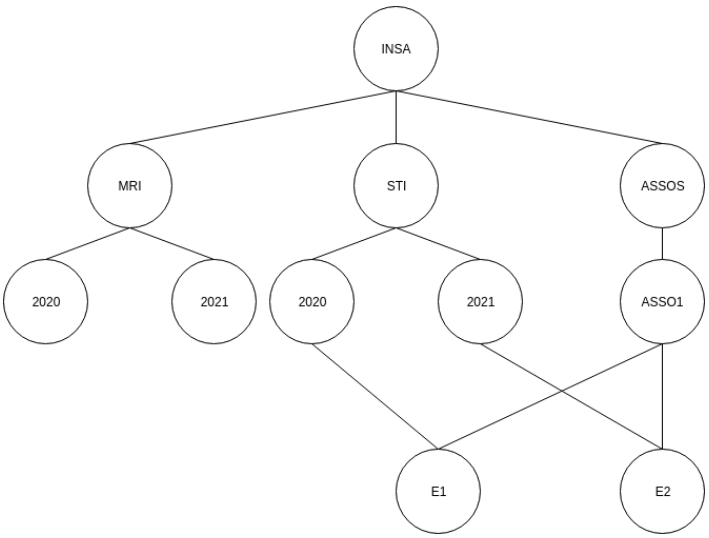
Nous avions plusieurs contraintes :

- Un processus ne doit pas pouvoir s'attacher à lui-même
- Un processus ne doit pas pouvoir s'attacher à un groupe dont il fait déjà partie

La partie connexion à soi-même a été facilement gérée avec une simple condition lors du attach().

La deuxième partie était plus intéressante à implémenter. Nous avons utilisé la liste de routage des processus. Si on processus appartient à la liste de routage de ce processus alors l'attach() est annulé car il fait déjà partie intrinsèquement à ce groupe. En utilisant cette technique un groupe ne peut pas s'attacher aux groupes supérieurs dont il fait déjà partie intrinsèquement et peut uniquement s'attacher aux groupes qui sont au même niveau ou à un niveau inférieur tout en gardant la possibilité de s'attacher à un autre arbre.

Mise en situation :



Avec cet exemple et l'implémentation précédente, un processus peut appartenir à plusieurs groupes : E1 appartient à INSA-STI-2020 et INSA-ASSOS-ASSO1. E1 peut s'inscrire au groupe MRI mais pas au groupe ASSOS car il en fait déjà partie par l'intermédiaire de ASSO1. De même, E1 ne peut pas s'attacher à INSA et STI car il en fait déjà par l'intermédiaire de INSA-STI-2020. Cependant E1 peut s'attacher à INSA-STI-2021 car E1 n'appartient pas au groupe INSA-STI-2021. De la même manière, E1 peut également s'attacher à E2.

17. Gestion Git

Avec le groupe, nous avons élu un gestionnaire de git. Cette personne devra gérer tous les merge, vérifier que le code après le merge fonctionne comme il le devrait, gérer les versions master/release/develop etc. J'ai été élu à ce poste du fait que j'avais le plus de connaissance dans l'utilisation de git parmi l'équipe. Cependant, à la fin du projet, tout le monde était en capacité de gérer le git de manière correcte.

La gestion du git s'est globalement bien passée. Cependant j'ai été confronté à quelques problèmes, notamment lors de merge conflict mal géré. J'ai appris de mes erreurs et cette erreur n'est arrivée qu'une fois et a pu être résolue grâce aux sauvegardes que je faisais régulièrement. J'ai appris également à utiliser des outils de merge conflict.

Nous avons choisi de ne mettre que les versions stables de notre projet sur master, les versions qui fonctionnent sur release et develop servait de base pour toutes les nouvelles fonctionnalités. Nous n'avons fait qu'un seul checkpoint sur la branche release. Le temps nous a empêché de reprendre le code de release et de corriger les derniers bugs c'est pourquoi il n'y a rien sur master. La version finale de notre projet sera évidemment sur master avec un maximum de correction de bugs.

18. Documentation

Pour la documentation, nous avons choisi d'utiliser doxygen pour C. En créant un Doxyfile avec doxywizard, j'ai mis la readme.md du git comme page d'accueil.

En ce qui concerne la documentation des fonctions, j'ai mis, dès le début, un maximum de commentaire et de documentation dans la base que j'ai faite.

Le fait d'avoir commencé les commentaires dans la base, l'équipe a été motivée et a continué à commenter toutes les fonctions au fur et à mesure de l'avancement du projet.

La documentation avec doxygen a été générée uniquement à la fin du projet.