Rapport de projet IMA4 2018/2019

Projet 26 : Discussion pair à pair

Étudiants : Fabien DI NATALE
Ibrahim BEN DHIAB

Encadrants : Xavier REDON
Thomas VANTROYS

POLYTECH
LILLE

Sommaire

Introduction

Sur Internet, lorsqu'on souhaite communiquer avec une personne, on passe généralement par un service de discussion instantanée qui transmet le message via leurs serveurs. La discussion n’est donc pas privée car le service voit passer toutes les communications. Un moyen pour privatiser ces discussions est de réaliser une communication en pair à pair sans faire passer les messages à un tier. Cependant, le principe de pair à pair sur Internet a été mis à mal par l’utilisation massive d’adresses IPv4 privées. Les utilisateurs sont majoritairement derrière une “box” fournie par leur fournisseur internet, et cette “box” effectue les translations d’adresses qui fournissent les IPv4 privées. Si l’utilisateur souhaite jouer le rôle de serveur, il est nécessaire qu’il modifie les paramètres de la “box”, qui ne lui sont pas toujours accessibles. Pour répondre à ce problème, nous avons choisi la technique du “Hole Punching” dont on détaillera la mise en place tout au long de ce rapport.

I. Définition du projet

A. Le sujet

Il nous est demandé de concevoir un dispositif permettant de retrouver une communication en pair à pair sans communication des messages à un tier. En l'occurrence mettre en place une application sous PC et Android. Et également ajouté une couche de chiffrage sur les communications.

Plusieurs techniques nous sont proposées : communication en IPv6 pour les utilisateurs dont les opérateurs offrent ce service ; communication en Ipv4/UDP avec envoi simultané de messages pour ouvrir les ports sur les "boxes" (synchronisation par SMS ou par tier) ; communication en IPv6/TCP avec contournement de la mascarade des "boxes" par communication des numéros de séquence par tier. Nous nous sommes orientés sur le “Hole Punching” pour IPv4/UDP & TCP, et en IPv6, une simple communication devrait suffire. Il est également demandé de mettre en place une maquette composé de routeurs configurés sur le principe de la mascarade pour tester nos prototypes, et pour pouvoir ensuite les tester en grandeur nature.

B. Le Hole Punching

1. NAT

Pour comprendre l'intérêt du “Hole Punching”, il est nécessaire de comprendre ce qu’est un NAT. Il y a plusieurs années, chaque appareil connecté chez l'utilisateur avait sa propre adresse IPv4 publique, hors IPv4 peut fournir 2^32 (4,294,967,296) adresses et avec l'évolution massive d'Internet ces adresses auraient étaient rapidement déplétées, en effet, on compte 23 milliards appareils connectés en 2018. *Network Address Translation* (NAT) a permis de réduire l'utilisation d'adresse IPv4 considérablement, en attribuant à chaque foyer une seule adresse IPv4 publique et en distribuant des IP privées à chaque appareil du foyer. C’est également pour cette raison que IPv6 a été inventé, il peut fournir 2^128 adresses, ce qui rendrait l’utilisation de NAT inutile.

2. UDP

L’UDP Hole Punching permet à 2 utilisateurs de mettre en place une communication pair à pair directe à l’aide d’un serveur de rendez-vous, même s’ils sont tous les deux derrière un NAT. Le Hole Punching considère que les deux utilisateurs, **A** et **B**, ont déjà des sessions UDP avec le serveur rendez-vous **S**. Quand un utilisateur s’enregistre avec **S**, le serveur mémorise son *endpoint* (adresse IP et port). Maintenant supposons que **A** veut communiquer en UDP avec **B**, **A** ne sait initialement pas comment accéder à **B**, donc il demande à **S** de l’aider à établir une session UDP avec **B**. **S** répond à **A** avec un message contenant **B** endpoint et en même temps il envoie à **B** l’endpoint de **A**. Une fois ces messages reçus par chaque utilisateur, **A** et **B** connaissent le endpoint de leur pair. Puis **A** envoie un paquet UDP à l’endpoint de **B**, de même pour **B** vers **A**. Si le paquet de **A** vers **B** atteint le NAT de **B** avant que le premier paquet de **B** vers **A** ait atteint son propre NAT, alors le NAT de **B** refusera sûrement le paquet de **A** et s’en débarrassera. Cependant, le

paquet de **B** pourra traverser le NAT de **A** comme **A** a ouvert le port de son NAT en le traversant. La figure ci-dessous schématise le fonctionnement du Hole Punching UDP :

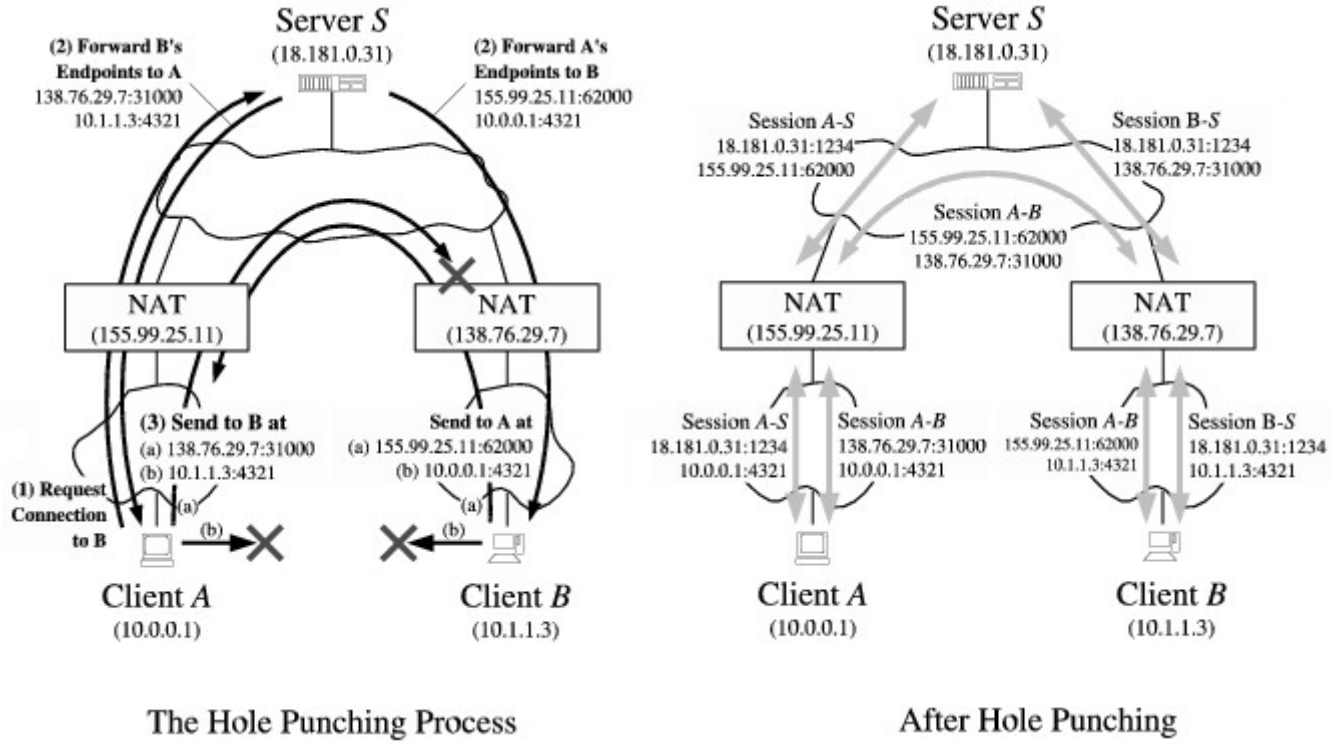

Figure 1 UDP Hole Punching, pairs derrière différents NATs[1]

3. TCP

Etablir une connexion pair à pair TCP entre deux utilisateurs derrière des NATs est plus complexe que pour l'UDP et la majorité des NATs ne le supportent pas à moins d'être configuré au préalable, ce qui n'est pas le but de ce projet.

En TCP, une socket peut initier une connexion sortante via connect(), ou bien écouter les connexions entrantes via listen() et accept(), mais jamais les deux en même temps. Egalement, les sockets TCP sont en correspondances bijectives avec des ports TCP sur le localhost. Après qu'un programme lie une socket à un port TCP local spécifique, il n'est plus possible de lier une autre socket à ce port.

Pour que le TCP Hole Punching fonctionne, il est nécessaire d'utiliser un seul port TCP pour écouter les connexions TCP entrantes et en même temps initier plusieurs connexions sortantes TCP. Pour cela, il existe une option pour les sockets qui est SO_REUSEADDR, qui permet au programme de lier plusieurs sockets au même port local tant que cette option est paramétrée pour chaque socket créée. Il est également nécessaire de paramétrer SO_REUSEPORT pour chaque socket.

Reprenons l'exemple précédent et supposons que l'utilisateur **A** souhaite établir une connexion TCP avec **B**. On suppose également que les utilisateurs ont une connexion TCP déjà réalisée avec le serveur **S**. Le même protocole qu'en UDP est utilisé, la différence est que l'application utilise le même port TCP que **A** et **B** ont utilisé pour s'enregistrer auprès de **S** pour effectuer les demandes de connexions entre **A** et **B** et en même temps pour accepter les connexions entrantes. C'est-à-dire que chaque utilisateur a besoin d'une socket pour leur connexion à **S**, une socket sur laquelle écouter et accepter les connexions entrantes du pair,

[1] Image obtenue sur http://bford.info/pub/net/p2pnat/index.html

et enfin une socket pour initier les connexions sortantes vers le endpoint du pair. Cette structure est schématisée ci-dessous :

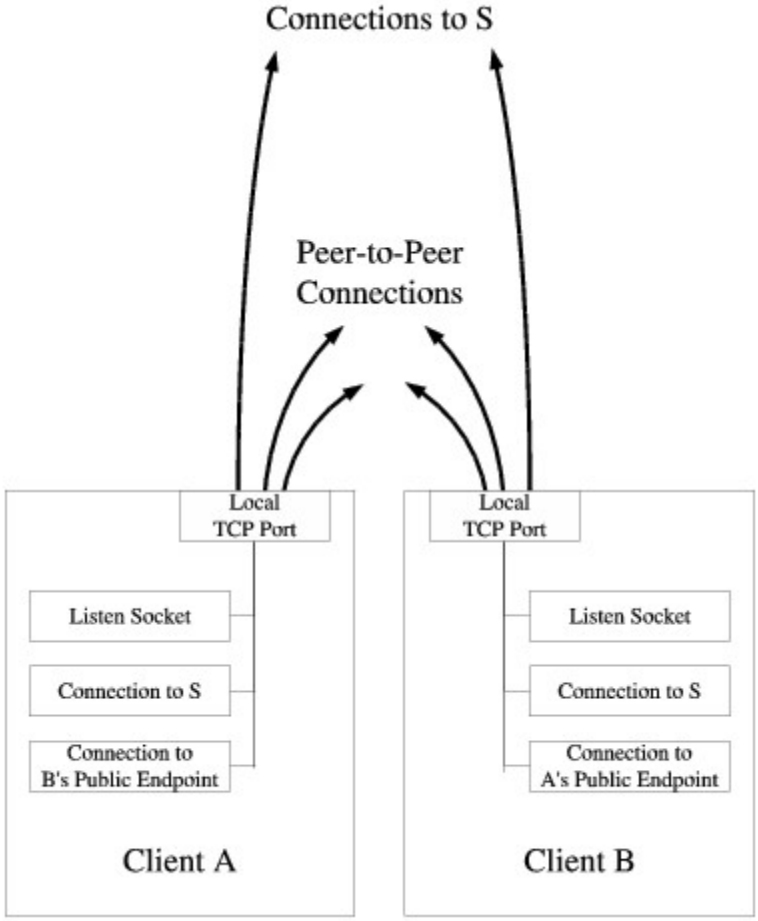

Figure 2 Sockets et ports pour le TCP Hole Punching[2]

Les tentatives de connexions de chaque utilisateur vers le endpoint de leur pair permet à leur NAT respectif d'ouvrir des « trous » permettant la communication TCP directe entre **A** et **B**. Si par chance les NATs se comportent bien, la connexion TCP s'effectue automatiquement entre eux. Cependant, si le paquet SYN de **A** vers **B** atteint le NAT de **B** avant que le paquet SYN de **B** vers **A** ait atteint le NAT de **B**, alors le NAT de **B** abandonnera le paquet SYN de **A**. Mais le premier paquet SYN de **B** vers **A** devrait par la suite arriver à traverser, parce que le NAT de **A** voit ce SYN comme appartenant à la connexion sortante vers **B** que le premier paquet SYN de **A** avait initialisé.

[2] Image obtenue sur http://bford.info/pub/net/p2pnat/index.html

II. Architecture du code

A. Client UDP

Nous avons divisé ce projet en deux parties que l'on a ensuite partitionné en plusieurs sous parties. Pour commencer, nous allons nous concentrer sur la partie Client UDP qui a été codée en premier. Afin de répondre aux attentes du sujet, nous avons codé un client UDP permettant d'établir une session Peer to Peer avec un second client du même type à l'aide d'un serveur rendez-vous évitant toute manipulation d'adresse IP par l'utilisateur et à la place il peut utiliser un pseudo.
Pour commencer, nous avons le fichier principale *Connexion_client_udp.c* qui est composé des fonctions suivantes :

- *ClientUDP*
- *connexion_serveur_udp*
- *hole_punching_udp*
- *connexion_hole_punching*
- *keep_alive*
- *communication_peer_to_peer*
- *gestion_message_peerB_envoie*
- *gestion_message_peerB_recu*

Nous avons tout d'abord la fonction de lancement *ClientUDP*, cette fonction est appelée par le "main" une fois que celui-ci a traité les options de lancement permettant à l'utilisateur de choisir le numéro de port et l'adresse IP du serveur. Par défaut, l'adresse IP est l'adresse du serveur rendez-vous et le port est 2020. *ClientUDP* est la fonction principale faisant appels aux autres fonctions permettant ainsi la communication avec le serveur rendez-vous et l'exécution de toutes les tâches nécessaires pour le Hole Punching.

La fonction *connexion_serveur_udp* est là pour initialiser la socket du client en UDP permettant par la suite la communication avec le serveur.

La fonction *hole_punching_udp* a pour but de traiter les informations envoyées par le serveurrendez-vousafin de créer la structure *sockaddr* correspondante. Cette fonction nous envoie ensuite sur la fonction *connexion_hole_punching* qui permettra d'entamer la session avec le pair. Une fois que cette session est établie, notre fonction *hole_punching_udp* lancera le thread *keep_alive* et la fonction *communication_peer_to_peer*.

La fonction *keep_alive* lancée en thread est présente pour maintenir la session ouverte entre les deux pairs en laissant percé les NATs. Elle envoie des paquets bien distincts vers le pair afin de pouvoir les reconnaître et ainsi ne pas les afficher lors de la conversation entre les deux pairs.

La fonction *communication_peer_to_peer* est, quant à elle, faite pour réceptionner les messages en provenance du pair et les afficher sur la console tout en envoyant les messages rentrés sur la console au pair. Cependant cette fonction fait appel aux fonctions *gestion_message_peerB_envoie* et *gestion_message_peerB_recu* qui permettent de ne pas afficher les messages keep-alive et de se déconnecter en cas de réception ou d'envoi du message "exit\n".

Le second fichier destiné à la communication UDP est *Communication_udp.c*et est composé des fonctions suivantes :

- *envoie_message_udp*
- *reception_message_udp*
- *Identification_serveur_udp*

Les deux premières fonctions permettent l'envoi ou la réception de message UDP à destination ou en provenance du serveur rendez-vous. La troisième fonction est destinée à l'identification du client auprès du serveur rendez-vous en demandant le pseudo de l'utilisateur et en envoyant la trame d'identification au serveur pour qu'il puisse traiter la requête de session et enregistrer les données de l'utilisateur.

Enfin, le fichier *Gestion_requete.c,* qui est commun aux client UDP et TCP, est composé de trois fonctions :

- *gestion_requete_serveur*
- *gestion_requete_client*
- *affichage_menu*

La fonction *gestion_requete_serveur* identifie les demandes du serveur rendez-vous, pour la session UDP nous avons juste implémenter la requête *hole_punching* permettant à l'utilisateur de démarrer le Hole Punching avec le pair.

La fonction *gestion_requete_client* identifie les demandes de l'utilisateur et les transmet proprement vers le serveurrendez-vous ou les refuse et affiche les demandes possibles sur le terminal du client. Cette fonction gère seulement deux requêtes mais peut être extensible à n'importe quelle requête. Les deux requêtes gérées sont l'arrêt de la session ainsi que la demande de session*hole_punching* avec un pair.

affichage_menu permet d'afficher toutes les requêtes que l'utilisateur peut demander au serveur.

B. Client TCP

Pour le client TCP, nous nous sommes inspirés de la même structure de fichier que pour le client UDP, ainsi, le fichier *gestion_requete.c* est commun aux deux. De plus, les fichiers *Connexion_client_tcp.c* et *Communication_tcp.c* ont pour différence la partie Hole Punching, qui en TCP est bien plus complexe. Nous avons donc créé un fichier supplémentaire *Hole_punching.c* composé des fonctions suivantes :

- *hole_punching*
- *connexion_hole_punching_TCP*
- *Initialisation_serveur_P2P*
- *gestion_requete_peer*
- *Discussion_P2P*
- *Serveur_P2P*

La fonction *hole_punching* décrypte le message contenant les informations du pair puis lance la fonction *connexion_hole_punching_TCP* qui gère tout le reste de la partie Hole Punching en TCP. Elle initialise tout d'abord le serveur auquel le pair se connectera à l'aide de la fonction *Initialisation_serveur_P2P*. Puis elle lancera le thread *Serveur_P2P* à condition que ce soit le pair qui est à l'origine de la connexion TCP pair à pair. Sinon elle fera des tentatives de connexions en boucle sur le serveur du pair ayant demandé la connexion.
Une fois la connexion établie, la fonction *Discussion_P2P* se lance et les pairs peuvent alors communiquer jusqu'à la demande de déconnexion gérée par la fonction *gestion_requete_peer*.

C. Serveur UDP

Cette partie est dédiée au serveur rendez-vous en commençant par la partie UDP. Pour cette partie Serveur UDP nous avons divisé notre code en trois fichiers, le principal est *Lancement_Serveur_UDP.c* contenant les fonctions :

- *initialisationServeur_UDP*
- *boucleServeur_UDP*
- *traitement_message_UDP*

La fonction *initialisationServeur_UDP* est la fonction basique permettant de créer la socket d'écoute du serveur avec l'option SO_REUSEADDR. Le *main* lance ensuite la fonction *boucleServeur_UDP* qui est la fonction principale de notre serveur UDP. Elle a pour but de réceptionner les messages UDP et les envoyer vers des fonctions de traitements du message. La fonction *traitement_message_UDP* sera appelée à chaque message reçu par le serveur, elle envoie le message vers une fonction de gestion des requêtes *gestion_requete* et en affiche le résultat. Cette fonction est contenue dans le fichier *Gestion_Client.c* et a pour but la détection des trames particulières tout en faisant appel à une fonction de protection serveur permettant de vérifier que le message reçu ne vient pas d'un client non réglementé.
Ce fichier contient les fonctions suivantes :

- *initialisation_liste_client*
- *gestionClient*
- *gestion_requete*
- *login*
- *Connexion*
- *Initiation_hole_punching*
- *force_deconnexion_client_TCP*
- *force_deconnexion_client_UDP*
- *protection_serveur*

La fonction *gestionClient* est la fonction de traitement pour les clients du serveur TCP, nous l'aborderons dans la partie suivante.
La fonction *initialisation_liste_client* permet d'initialiser le tableau de données que le serveur utilisera pour stocker les différents clients connectés ainsi que leurs informations nous permettant ainsi de mettre en relation les différents clients connectés au serveur.
Les fonctions *login* et *connexion* sont les deux fonctions de traitement des requêtes, si le serveur reçoit une demande d'identification alors la fonction *login* sera lancée et enregistrera

les informations du client dans notre base de données (notre tableau) à l'aide du troisième fichier *memorisation_client.c* que l'on étudiera plus bas. Cependant, si la fonction de *Connexion* est appelée, cela signifie qu'un client déjà enregistré souhaite se connecter à un autre client présent dans notre base de données, cette fonction est simplement là pour récupérer les deux pseudos et lancer la fonction *Initiation_hole_punching* avec ces informations. La fonction *Initiation_hole_punching* fabrique les messages à envoyer aux deux clients à partir des pseudos qui lui ont été transmis par la base de données.
La fonction *force_deconnexion_client_UDP* a pour rôle d'envoyer un message à tous les clients connectés au serveur afin qu'ils s'arrêtent proprement lors de l'arrêt du serveur, la *fonction_deconnexion_client_TCP* fait exactement la même chose mais pour le serveur TCP.
Lors de l'analyse des fonctions de ce fichier, nous avons pu voir plusieurs fonctions faisant appel à notre base de données, celle-ci sont gérées dans le fichier *memorisation_client.c* à l'aide des fonctions suivantes :

- *Connexion_client*
- *hachage*
- *Deconnexion_client*
- *nombre_de_client_connecte*

La fonction *nombre_de_client_connecte* permet de connaître le nombre de clients connectés au serveur et ainsi garder la variable *client_connecte* en *static* et éviter de la mettre en *global*. La fonction hachage est une fonction de hachage basique par pseudo (chaîne de caractères) renvoyant la position où le client devrait être positionné ou où il est positionné dans le tableau des clients mémorisés.
Les fonctions *Connexion_client* et *Deconnexion_client* permettent la mémorisation des informations des clients lors de leur « connexion » et la suppression de leurs informations lors de leur « déconnexion » (UDP = non connecté mais pour faciliter la compréhension on utilise la terminologie « connexion »).

D. Serveur TCP

L'étude de la partie serveur UDP nous a permis d'aborder la plupart des fonctions ou fichiers qui sont aussi utilisés pour notre serveur TCP tels que le fichier *Memorisation_client.c* et aussi *Gestion_client.c*. Cependant, ce dernier comporte une fonction spécialement dédiée à notre serveur TCP, la fonction de traitement des clients TCP. Elle a pour but de lire les commandes reçues par le serveur ainsi qu'envoyer les commandes que l'utilisateur peut utiliser. Elle reste très proche de la fonction *Traitement_message_UDP* mais adapté au protocole TCP et lancé en thread afin d'avoir une multitude de clients connectés en même temps.
Le fichier *Lancement_Serveur_TCP.c* est dédié au serveur TCP contenant deux fonctions :

- *initialisationServeur_TCP*
- *boucleServeur_TCP*

La fonction *initialisationServeur_TCP* permet l'initialisation, le bind et la mise en place des options de socket pour lancer le serveur proprement et pouvoir l'utiliser par la suite dans la fonction *boucleServeur_TCP* qui a elle pour but, la récupération des informations du pair venant de se connecter et le lancement de la fonction de traitement que l'on a vu plus haut.

Ainsi, nous avons un code assez polyvalent mais aussi complexe suite aux différents problèmes que nous avons rencontré.

E. Partie Ipv6

Pour l'IPv6, nous avons seulement réalisé la partie UDP parce que nous ne sommes pas parvenus à faire fonctionner la partie Ipv4/TCP totalement. Pour cela, nous avons simplement récupéré les fichiers en rapport avec l'Ipv4/UDP, et avons ajusté les fonctions spécifiques et variables à l'Ipv4 (provenant des librairies socket.h …) vers les fonctions et variables spécifiques à l'Ipv6, notamment celle ci-dessous :

- Inet_ntoa() -> inet_ntop() qui est utilisé pour la conversion des adresses pour pouvoir les affichées ;
- Inet_aton() -> inet_pton() qui converti les adresses IPv6 de texte en forme binaire ;
- Struct sockaddr_in -> sockaddr_in6 ;
- AF_INET -> AF_INET6 ;
- Le premier paramètre de socket() devient PF_INET6.

Egalement, nous avons désactivé les parties où le Hole Punching est réalisé ainsi que le thread *keep_alive* qui ne sont pas nécessaires en IPv6.

III. Les difficultés rencontrées

A. Les problèmes rencontrés

1) Le premier problème rencontré a été l'envoi des commandes au serveur, on ne souhaitait pas les réglementer afin que le serveur les gère toutes afin de garder le code client le plus simplifié possible mais nous avons très vite remarqué que cela donnait trop de travail au serveur et le rendait bien trop vulnérable aux messages que l'utilisateur pouvait envoyer.

2) Cela amène un second problème, nous n'étions plus capables de faire des tests à l'aide de la commande netcat et selon ce que l'on envoyait le serveur l'interprétait mal ou plantait complètement. En plus de cela, la sécurité des informations était compromise car trop facilement modifiable.

3) Une fois les problèmes généraux liés aux messages résolus, nous nous sommes aperçus que la détection de déconnexion des clients en mode UDP était impossible à détecter.

4) De plus, nous avons aussi très vite remarqué qu'il nous était impossible de savoir qui envoyait des requêtes au serveur en mode UDP. Il était donc impossible de savoir si la personne était dans notre base de données ou si elle devait s'identifier. Egalement, le serveur ne pouvait pas récupérer les informations pour le Hole Punching aussi facilement et aussi rapidement qu'on le souhaitait.

5) Pour le mode TCP, il y avait un problème lors de la connexion Hole Punching. La connexion était effectuée mais on ne voyait aucun message passer malgré des appels aux fonctions *pthread_cancel* et *pthread_join*.

B. Les choix réalisés

1) Après réflexion et études de nos possibilités, nous nous sommes rendu compte que le serveur ne pouvait pas tout gérer et que les utilisateurs ne pourront pas appliquer les protocoles sans erreurs. Nous avons donc créé des trames pour les différentes fonctions que le serveur pourrait recevoir et pour les différentes fonctions que le client pourrait recevoir.

2) Afin de résoudre les problèmes de crash serveur et éviter les utilisateurs d'utiliser un autre programme que notre programme client (ce qui pourrait nuir à notre programme), nous avons ajouté une fonction *protection_serveur* permettant d'éviter tout problème et renvoyant un message à l'utilisateur indésirable pour qu'il utilise notre client.

3) Afin de garder notre base de données la plus propre possible, nous nous sommes intéressés à la déconnexion des différents clients et nous en sommes venu à la

conclusion que le client UDP ne pouvait et ne devait pas se déconnecter sans en informer le serveur. Nous avons donc ajouté une fonction de « déconnexion » et empêché l'interruption par le signal SIGINT.

4) Afin de régler ce problème d'identification pour le mode déconnecté, nous avons ajouté dans le client une variable pseudo lui permettant d'envoyer les messages au serveur en y ajoutant le pseudo de l'utilisateur. En parallèle, nous avons ajouté une fonction dans le serveur afin de décrypter ce message, vérifier que le client est bien enregistré dans notre base de données, cette fonction modifie ensuite le message en enlevant le pseudo afin que nos autres fonctions puissent continuer de fonctionner sans problème. Nous avons été obligés d'utiliser cette technique afin de garder un programme efficace, la seconde solution était de faire un hachage par adresse IP pour la mémorisation des clients mais cela impliquait de parcourir le tableau de données pour chaque demande de Hole Punching car celle-ci se fait par pseudo.

5) Nous avons pu voir plus haut que la connexion Hole Punching TCP nécessite que les deux clients possèdent un serveur chacun et essaye en boucle de se connecter à son pair. Cependant avec les tests, ceci était impossible lorsque les deux connexions arrivaient à s'effectuer en même temps. On ne pouvait pas voir les messages car le programme principal prenait le pas sur le thread et on ne pouvait voir afficher sur le terminal que les messages écrits sur le terminal mais pas les messages reçus car ils étaient cachés par le client.
   Plus clairement, en prenant ClientA pour le client TCP créé par le premier peer et ServeurA pour le serveur créé par le premier peer et de même pour le second peer avec ClientB et ServeurB, nous avons le cas suivant :
   ClientA connecté sur le ServeurB
   ClientB connecté sur le ServeurA
   Avec le clientA nous pouvons écrire des messages qui partent à destination du ServeurB et nous recevons les messages du ServeurB et pour le clientB nous pouvons écrire des messages qui partent à destination du ServeurA et nous recevons les messages du ServeurA.
   Ce qui pose le problème suivant, le ServeurA et le ServeurB sont en tâche arrière et n'affiche rien sur le terminal ou ne lisent pas les entrées du terminal. Nous avons donc fait le choix que le client demandant la connexion ouvre un serveur et que celui recevant la demande de connexion fasse seulement des tentatives de connexions en boucle. Sans oublier que le serveur tout comme le client doit vérifier que la connexion a bien été établie avec le pair souhaité évitant tout problème de connexion non souhaitée.

IV. Banc de test et tests à échelle réelle

A. Le banc de test

Le sujet nous demandait de mettre en place un banc d'essais avec lequel nous pouvions tester notre programme localement avant de s'orienter vers les tests à échelle réelle. Le banc d'essais doit donc être similaire à un cas réel, nous nous sommes donc inspiré des réseaux que l'on peut retrouver chez le particulier, c'est à dire un modem avec habituellement l'adresse IP 192.168.0.1/24, qui délivre à tous les appareils connectés du domicile une adresse IP à l'aide d'un NAT dynamique PAT par masquerade. Cependant, dans notre cas une seule machine est connectée au routeur, il n'est pas nécessaire d'avoir un NAT dynamique, un NAT statique est suffisant. Pour cela, nous avons mis en place le schéma suivant :

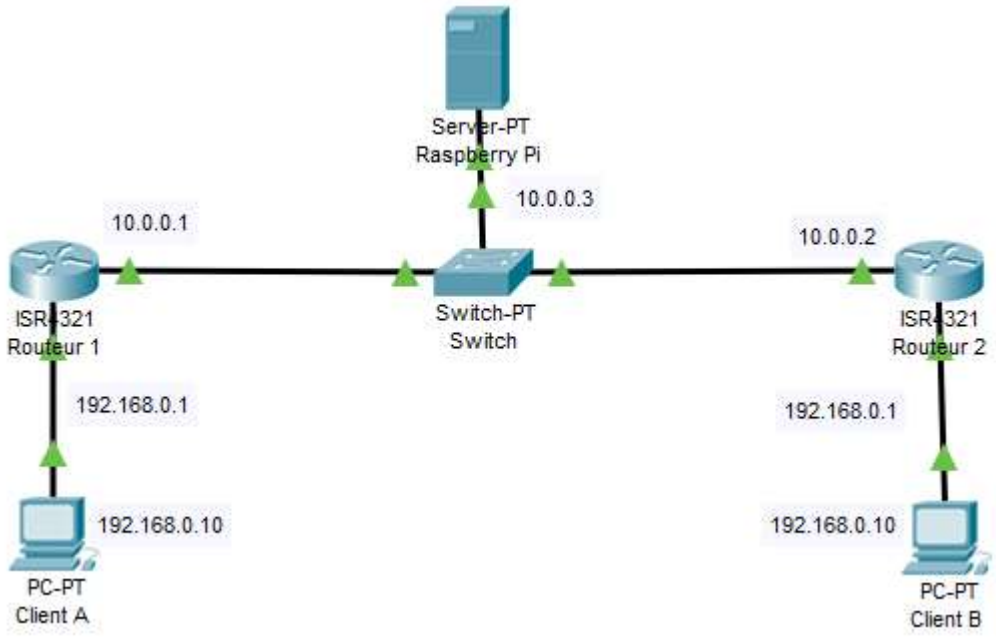

Figure 3 Schéma de notre banc de test

Les routeurs à notre disposition étaient des CISCO ISR4221, le serveur était hébergé sur un Raspberry Pi et un switch est placé entre les 3 pour être interconnectés, le matériel a été installé dans un rack en E304 à l'aide de M. Redon. Le Pi était équipé d'un dongle WiFi pour récupérer aisément l'exécutable du serveur rendez-vous en SSH. Il a été configuré au préalable en connexion série.

Comme le schéma l'indique, les deux machines ont la même IP privée, mais une IP publique différente. Si on utilise netcat pour envoyer à paquet une machine à l'autre, le NAT le bloquera et l'abandonnera car il ne sait pas quoi en faire. Et donc pour tester si notre système est opérationnel pour le Hole Punching, on le simule avec netcat en écoutant sur le port 2020 avec le client A et le client B émet des paquets UDP vers l'adresse 10.0.0.1 (adresse routeur 1) sur le port 2020. Si le système fonctionne, le client A doit pouvoir recevoir les messages du client B, même en présence du NAT. En effet, comme le client A indique au routeur qu'il écoute sur le port 2020, si le routeur reçoit des paquets sur ce port, il les redirigera vers le client A, ce qui équivaut à un perçage du routeur 1.

On a par la suite testé notre programme en UDP et TCP IPv4 et il s'avérait fonctionnel. Pour l'IPv6, nous avons réussi à acquérir un serveur externe équipé d'une IPv6, on a ainsi décidé de réaliser les tests IPv6 directement avec ce serveur au lieu d'essayer de configurer les routeurs en IPv6.

B. Tests à échelle réelle

Ces tests ont été réalisé à l'aide du serveur externe cité précédemment. Tout d'abord, nous avons testé IPv4/UDP, en nous connectant chacun chez nous et donc derrière des NATs différents, et le serveur rendez-vous hébergé sur le serveur externe. Ces tests ont été réussis, nous avons également essayé d'initier des communications entre plusieurs clients en même temps et tout fonctionné, également, lorsqu'un client essaie de communiquer avec un autre qui est déjà en communication avec quelqu'un, rien ne se passe, la communication entre les deux n'est pas interrompue. Maintenant, pour IPv4/TCP, les tests ont été réalisés dans le même environnement, mais ils ont malheureusement échoué sûrement à cause de la robustesse des NATs.

Conclusion

Ce projet nous a permis d'utiliser l'entièreté de nos connaissances en C et en réseau acquises au cours de notre formation à Polytech Lille. Nous avons également acquis de nouvelles connaissances, notamment le fonctionnement du NAT et l'utilisation des IPv6. Nous sommes parvenus à répondre en partie aux objectifs du sujet, en réalisant avec succès un programme pour communiquer en pair à pair en IPv4&6/UDP, et partiellement en IPv4/TCP. Nous aurions souhaité rendre fonctionnel IPv4&6/TCP, mais le Hole Punching en TCP ne s'avère pas optimale, s'orienter vers d'autres techniques pair à pair plus rigoureuses en TCP serait idéale. Nous nous sommes bien aperçus que la communication était bien plus simple à réaliser en IPv6, ce qui met en avant l'avantage considérable de ce protocole, en sachant qu'il sera de plus en plus diffusé les années à venir avec la saturation des IPv4.
Notre programme est capable d'être bien amélioré, il serait intéressant de mettre en place un chiffrage de messages pour empêcher tout individu de s'interposer entre les pairs et d'espionner la conversation, également, l'ajout d'options telles que pouvoir refuser une demande de communication d'un pair ou encore la possibilité de voir qui est en ligne et savoir s'ils sont en discussions ou non, ou enfin pouvoir communiquer en groupe et pas seulement 1 à 1.