# Réseaux
# Projet : tunnel IPv6-IPv4

Manon GIRARD - Enzo CADONI

# Table des matières

# 1 Introduction

On se propose de réaliser un tunnel permettant à deux LAN IPv6 de communiquer de manière bidirectionnelle par le biais d'un ou plusieurs LAN IPv4. Le tunnel aura donc pour but de se positionner sur une machine pouvant communiquer avec les deux types de LAN et ainsi permettre la communication. Afin de pouvoir transmettre le traffic IPv6 par des LAN IPv4, le tunnel encapsulera le traffic IPv6 dans des paquets TCP/IPv4.

Le langage choisi pour programmer ce tunnel est le langage C. En effet, ce langage permet de pouvoir observer en détail le fonctionnement des objets que l'on utilise, notamment les appels système. Aussi, C permet de programmer de manière modulaire plus clairement qu'avec Python.

# 2 Configuration réseaux

Nous disposons d'un réseau de 5 machines virtuelles (VM), VM1, VM2 et VM3 peuvent communiquer par IPv4 entre elles, tout comme les couples VM1/VM1-6 et VM3/VM3-6 le peuvent par IPv6.

Les VMs disposent toutes d'un fichier de configuration ansible situé dans leur répertoire. On pourra le retrouver dans le système de fichier interne des VMs dans le répertoire `/vagrant/`.

Il faut désormais construire un tunnel afin que VM1-6 et VM3-6 puisse communiquer.

# 3 L'interface virtuelle TUN

Afin de manipuler des interfaces virtuelles TUN, on crée la bibliothèque `iftun`.

## 3.1 Création de l'interface

Afin de pouvoir créer une interface TUN, une fonction `tunalloc` est rajoutée à la bibliothèque `iftun`.
On remarque que l'interface (ici tun0) créé en utilisant `tunalloc` n'est pas persistente quand le programme englobant l'appel de `tunalloc` termine, il faut donc pouvoir la configurer avant afin de l'utiliser.
`tunalloc` aura aussi pour but de renvoyer le descripteur de fichier correspondant à l'interface.

## 3.2 Configuration de l'interface

Dans le but de configurer tun0 après sa création et avant que le programme ne commence à l'utliser, nous allons créer un script bash permettant à `tun0` de s'activer et de posséder une adresse IPv6, en l'occurence `fc00:1234:ffff::1`.

Ce script est donc pour l'instant composé des deux commandes suivantes

```
# Activation de tun0 dans le cas ou elle serait inactive
ip link set tun0 up

# Ajout d'une IPv6 à tun0
ip -6 a add fc00:1234:ffff::1/64 dev tun0
```

#### DONNER LA CAPTURE DES PINGS SUR TUN0

Lorsqu'on réalise un ping sur l'adresse IP de l'interface `tun0` à savoir `fc00:1234:ffff::1`, on peut observer qu'aucun paquet n'est intercepté par `wireshark` sur tun0, mais que le ping obtient une réponse depuis la machine on le réalise.
En revanche, lorsqu'on réalise un ping sur une adresse telle que `fc00:1234:ffff::10`, wireshark intercepte les paquets en destination de l'adresse sur `tun0`, le ping n'obtient pas de réponse, l'adresse ne menant à aucune machine.
Une explication pourrait etre la suivante, l'interface `tun0` ne traite pas les paquets qui ont pour destination l'interface elle meme, elle peut néanmoins renvoyer une réponse si le paquet en exige une, par exemple dans le cas d'un paquet ICMP envoyé durant un ping. Lorsqu'on réalise un ping sur une adresse présente dans le sous-réseau de `tun0`, à savoir `fc00:1234:ffff::/64`, `tun0` transmet les paquets, et donc les traite, ils sont donc visible sur `tun0` depuis `wireshark`

### 3.3 Récupération des paquets

L'interface `tun0` est créé, on dispose maintenant d'un descripteur de fichier lui correspondant permettant de lire des informations. On crée une fonction `transfert` dans le bibliothèque `iftun`, ayant pour but d'écrire d'un fichier `src` dans un fichier `dest`.

Nous allons nous servir de la fonction transfert afin de rediriger des flux vers tun0 et réciproquement.

On peut imprimer le flux de `tun0` sur la sortie standard en utilisant la fonction de transfert (la source est le descripteur de fichier de `tun0`, la destination est le descripteur de la sortie standard, soit 1). On filtrera la sortie grace à `hexdump` afin de ne pas essayer d'imprimer des caractères non imprimables.

On peut observer que rien n'est imprimé lorsque l'on réalise un ping `fc00:1234:ffff::1` et que le traffic IPv6 transféré par `tun0` est imprimé lorsqu'on réalise un ping sur `fc00:1234:ffff::10`. L'hypothèse permettant d'expliquer ce résultat est la meme que celle permettant d'expliquer pourquoi `wireshark` intercepte ou non les paquets envoyés durant ces pings.

**Ici une comparaison d'un paquet reçu sur la sortie standard et d'un paquet wirshark**

L'option `IFF_NO_PI` permet de se passer des 4 octets précedents les paquets reçus par l'interface ayant pour but de préciser la version du protocole IP utilisé par le paquet.

Si l'on ajoute cette option, on diminuera donc le nombre d'octet lu et écrit dans la fonction de transfert.

## 4 Un tunnel simple pour IPv6

Nous allons ici construire pas à pas le tunnel bidirectionnel qui permettra aux deux LAN (3 et 4) séparées de pouvoir à nouveau communiquer.

### 4.1 Redirection du trafic entrant

Dans le but de pouvoir gérer le traffic entre les extrémités du tunnel, nous allons créer la bibliothèque `extremite`, elle contiendra deux fonctions, `ext_out` et `ext_in`

`ext_out` aura pour but, dans un premier temps, de créer un serveur via une socket. Ce serveur aura pour but d'écouter sur le port 123 et de rediriger le traffic sur la sortie standard.

`ext_in` devra se connecter via une connexion TCP au serveur qu'à ouvert l'autre extrémité du tunnel sur le meme port. La fonction devra ensuite transmettre le traffic lu sur `tun0` sur la socket.

En lisant le manuel des appels systèmes `recv` et `send` permettant de lire et d'écrire avec une socket, on observe que ces appels sont respectivement équivalents aux appels systèmes `read` et `write` si l'option `flags` de ces appels est égal à 0, c'est à dire que nous n'avons pas besoin d'options particulière. Nous n'avons besoin d'aucune option, nous pouvons donc utiliser la fonction `transfert` de la bibliothèque `iftun` transferer un flux vers la socket ou réciproquement.

On déploie un programme exécutant `ext_in` sur VM1 et un autre exécutant `ext_out` sur VM3, on ping ensuite `fc00:1234:ffff:10` sur VM1 afin d'injecter du traffic dans l'interface `tun0` de VM1.

On observe que le traffic est bien injecté dans VM1 et imprimé sur VM3, on peut d'ailleurs le voir passer par VM2 en capturant les paquets sur cette dernière à l'aide de `wireshark`

**CAPTURE WIRESHARK VM2**

### 4.2 Redirection du trafic sortant

Afin de pouvoir transmettre le traffic sortant par `ext_out` sur l'interface `tun0` locale, au lieu de rediriger le traffic sur l'entrée standard, on le redirige directement dans `tun0` dans `ext_out`.

Cette modification permettra au datagramme IPv6 conduit par le tunnel d'etre transmis par `tun0` vers sa destination. En effet, lorsque l'on écrit dans le fichier correspondant à `tun0`, l'interface interprète les paquets comme si ils étaient envoyé par une machine (par exemple par un ping), `tun0` se charge donc de les retransmettre.

**TEST D'UN PAQUET TRANSMIS PAR VM2**

## 4.3 Intégration Finale du Tunnel

Nous avons désormais la possibilité de réaliser un tunnel unidirectionnel en lançant `ext_in` sur une extrémité de tunnel et `ext_out` sur l'autre. Pour assurer la bidirectionnalité du tunnel, il faudra lancer les deux fonctions en meme temps, nous allons pour cela utiliser des threads POSIX (pthreads).

les deux fonctions (`ext_in`, `ext_out`) seront chacune lancées dans un thread, il y aura donc deux threads en plus du thread principal.

le programme ainsi lancé sur les deux extrémités, ces dernières pourront communiquer de manière bidirectionnelle.

## 4.4 Mise en place du tunnel entre VM1 et VM3 : Schémas

Ici un schéma détaillé du traffic passant par le tunnel sur une extrémité de ce dernier.