Réseaux
Projet : tunnel IPv6-IPv4

Manon GIRARD - Enzo CADONI

Table des matières

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.

1 Configuration réseaux

1.1 Topologie et Adressage

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

1.2 Un grand malheur !

Afin de simuler la perte de `VM2-6`, on enlève la machine du réseau et on supprime tout ce qui est superflu sur les autres machines, c'est à dire les interfaces et les routages qui étaietn nécessaires à `VM1-6` et à `VM3-6` pour communiquer avec elle. Il ne reste donc à ces deux machines qu'une interface dotée d'une IPv6.

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

2 L'interface virtuelle TUN

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

2.1 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.

2.2 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::32 dev tun0
```

On affecte le masque /32 à tun0 afin que les adresses IPv6 des différents LAN soient reconnues comme faisant partie du sous-réseau de `tun0`, les paquets passeront donc par le tunnel sans routage supplémentaire sur la machine créant `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

où 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. On peut observer une capture d'un ping sur cette adresse dans le fichier `captures/ping_tun0:10.pcapng`

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`

2.3 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 voir dans le fichier `captures/sortie_test_iftun.log` un exemple de redirection du flux de tun0 sur la sortie standard (filtré par `hexdump`).
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.

Le fichier `captures/sortie_test_iftun.log` correpond à la capture `captures/ping_tun0:10.pcapng` déjà observée précedemment. On peut observer que les paquets (en vue héxadécimale) correspondent à ce que l'on peut voir d'imprimé sur la sortie standard (dans le log).

Enfin, l'option `IFF_NO_PI` permet de se passer des 4 octets précedents les paquets reçus par l'interface étant composés de 2 bits octets de flags et de deux autres renseignant le protocole utilisé.
Si l'on ajoute cette option, on diminuera donc le nombre d'octet lu et écrit dans la fonction de transfert.

3 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.

3.1 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 système `recv` et `send` permettant de lire et d'écrire avec une socket, on observe que ces appels sont respectivement équivalents aux appels système `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 peut tester le déploiement en se plaçant au niveau de `VM2` sur `wireshark` et en faisant un ping de `VM3-6` depuis `VM1-6`. On peut voir dans `captures/sortie_test_ext.log` la sortie standard de `VM3` une fois `ext_out` lancée. En comparant le log à la capture `captures/test_ext_VM2.pcapng` et particulièrement la section données des paquets TCP émis par `172.16.2.131`, on voit que les paquets contiennent bien le datagramme IPv6 encapsulé au niveau de `VM1`.

3.2 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.

On lance `ext_in` sur `VM1`, `ext_out` sur `VM3` puis on capture dans le fichier `captures/test_tun0_ext_out.pcapng` sur l'interface `tun0` les paquets envoyés par `VM1-6` à `VM3-6`. On extrait un paquet en hexadécimal (ASCII dans un premier temps) dans le fichier `captures/paquet_tun0_VM3.hexdump`, on peut aisément voir dans ce paquet les adresses sources et destination, respectivement les adresses de `VM1-6` et de `VM3-6`. Ce paquet est un paquet `ICMPv6` comme on pouvait s'y attendre.

On peut injecter ce paquet manuellement dans tun0 (grâce à `netcat` par exemple), cela aura le même effet que de l'injecter directement à partir du client comme le fait la fonction `ext_out` en l'état actuel. Il faut faire en revanche faire attention à maintenir à jour le CRC du datagramme si on souhaite le modifier, sinon, il y a une très forte probabilité pour que le paquet soit considéré comme corrompu. Le CRC peut être recalculé en faisant le complément à 1 de la somme des compléments à 1 des valeurs des octets du paquet et doit être exprimé sur deux octets.

3.3 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.

On peux vérifier que les communications en réaliant un ping depuis deux terminaux différents sur VM3, on voit que les réponses arrivent alternativement sur les deux terminaux, la communications sont donc bien asynchrones.

3.4 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.

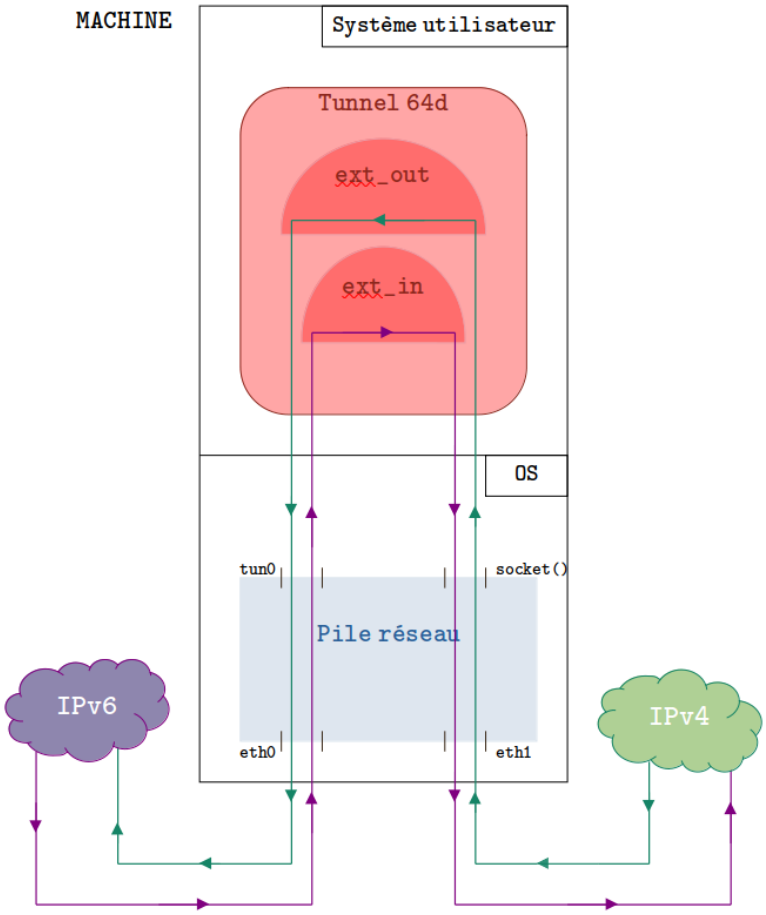

FIGURE 1 – Schéma d'un tunnel IPv6 vers IPv4

3.5 Mise en place du tunnel entre VM1 et VM3 : Système

Nous enfin un exécutable permettant de lancer le tunne bidirectionnel sur une machine, nommé : `tunnel64d`.

Cet exécutable prendra sa configuration dans un fichier `tunnel.conf` qui devra etre situé dans le répertoire `/vagrant` de la machine. Ce fichier devra etre de la forme :

```
...
dev=
...
portin=
...
portout=
...
ipout=
...
```

— `dev` doit etre égal à l'interface TUN que le tunnel va devoir utiliser (après l'avoir créé), exemple : `tun0`.
— `portin` doit etre égal au port par lequel on souhaite écrire les données transitant par l'interface.
— `portout` doit etre égal au port sur lequel on souhaite écouter les données émises par l'autre extrémité du tunnel.
— `ipout` doit etre égal à l'IPv4 de l'autre extrémité du tunnel

— ... peut représenter tout texte superflu, dont d'éventuels commentaires

On peut voir en lancant le tunnel sur les deux extrémités que les machines peuvent se voir sur le réseau, on peut lancer un ping d'une machine vers l'autre et observer que le ping reçoit une réponse, le tunnel est donc bien bidirectionnel.

4 Validation Fonctionnelle

4.1 Configuration

Le tunnel est en place sur les machines `VM1` et `VM3`, en respectant l'architecture réseau vue précédemment. Les paramètres réseaux significatives sur les machines lorsque le tunnel est actif sur le réseau sont :

- `VM1`
  - Adresses IP :
    - `eth1 : 172.16.2.131/28`
    - `eth2 : fc00:1234:3::1/64`
    - `tun0 : fc00:1234:ffff::1/32`
  - Routes :
    - `172.16.2.160/28 via 172.16.2.132 dev eth1`
- `VM2`
  - Adresses IP :
    - `eth1 : 172.16.2.132/28`
    - `eth2 : 172.16.2.162/28`
- `VM3`
  - Adresses IP :
    - `eth1 : 172.16.2.163/28`
    - `eth2 : fc00:1234:4::3/64`
    - `tun0 : fc00:1234:ffff::1/32`
  - Routes :
    - `172.16.2.128/28 via 172.16.2.162 dev eth1`
- `VM1-6`
  - Adresses IP :
    - `eth1 : fc00:1234:3::16/64`
  - Routes :
    - `default via fc00:1234:3::1`
- `VM3-6`
  - Adresses IP :
    - `eth1 : fc00:1234:4::36/64`
  - Routes :
    - `default via fc00:1234:4::3`

4.2 Couche 3

Dans le but de vérifier que le tunnel est fonctionnel, on peut réaliser un ping de `VM1-6` depuis `VM3-6`. On peut observer que le ping reçoit une réponse, le tunnel est donc fonctionnel.

La capture `captures/test_ping_tunnel_VM2.pcapng` est ce que l'on peut observer au moment du ping sur `VM2` et la capture `captures/test_ping_tunnel_VM3.pcapng` est ce que l'on peut observer au moment du ping sur `VM3-6`, la machine cible.

On peut voir sur la capture de `VM3-6` que des paquets `ICMPv6` arrivent bien sur la machine avec comme adresse source celle de `VM1-6` et que la machine lui répond avec d'autres paquets `ICMPv6`. Aussi depuis VM2, on peut observer les requêtes et les réponses que se font `VM1` et `VM3`.

On pourrait aussi essayer de réaliser un ping depuis `VM1` vers `VM3` ou bien `VM3-6`. On remarque dans un premier temps que cela ne fonctionne pas. En effet, en regardant sur wireshark, on peut voir que la source du paquet `ICMPv6` est l'adresse IP de `tun0`, la réponse du ping arrive donc sur cette meme adresse IP et se trouve donc ignorée.

Il est néanmoins possible de réaliser ce ping en précisant l'interface duquel on souhaite initialement lancer le ping, il suffit de rajouter l'option `-I` suivie de l'adresse IP de l'interface depuis laquelle on souhaite lancer le ping

(exemple : la commande `ping6 -I fc00:1234:3::1 fc00:1234:4::3` permet de ping la machine `VM3` depuis `VM1` par le tunnel).

4.3 Couche 4

Afin de tester si il est possible d'utiliser le service `echo` de `VM3-6` depuis une machine du `LAN3`, nous allons créer un fichier que nous allons envoyer grâce à `netcat`.

On met dans un fichier `test.txt` le message `"test abcd 1234"`. Nous faisons une capture sur `VM3-6` au moment où l'on exécute la commande `telnet fc00:1234:4::36 echo < test.txt`. On peut voir dans la capture `captures/test_service_echo.pcapng` que le message est bien reçu par `VM3-6` et que cette dernière répond.

4.4 Couche 4 : bande passante

L'utilitaire `iperf3` permet de réaliser divers tests sur les performances de connexion, nous allons l'utiliser pour tester la bande passante du tunnel.
On teste le débit avec différentes tailles de tampon

taille du tampon	débit
10o	2.98Kbits/sec
2Ko	610Kbits/sec
128Ko	2.99Mbits/sec
1Mo	3.39Mbits/sec

FIGURE 2 – Comparaisons de la bande passante du tunnel en fonction de la taille du tampon utilisée

On remarque que la bande passante augmente quand la taille du tampon augmente.

5 Améliorations

5.1 Prise en compte de la taille des paquets

Afin d'alléger la lecture de l'extrémité finale du tunnel, on peut prendre en compte la taille des paquets y transitant.

Tout d'abord, rappelons que l'appel système `read` renvoie le nombre d'octet qu'il a lu à partir du descrpteur de fichier qu'on lui à donné, on peut se servir de ce retour pour déterminer la taille d'un paquet dans `ext_in`. Le nombre d'ocets à lire passé en paramètre de `read` dans le transfert de `ext_in` sera égal au `MTU` de `tun0`.
Sachant que la taille d'un paquet peut être supérieur à 256, on va écrire la taille du paquet sur deux octets. On va ensuite envoyer ces deux ocets avant d'écrire le paquet dans le tunnel. À la réception de cette entête, l'autre extrémité du tunnel (la fonction `ext_out`) va pouvoir lire exactement le nombre d'octets que contient le paquet, puis l'écrire dans le `tun0` local.

En terme d'implémentation, on passera un paramètre à la fonction existante `transfert` de la bibliothèque `iftun`, selon si on est dans la fonction `ext_out`, ou bien dans la fonction `ext_in`. On peut voir l'impléntation finale de la fonction `transfert` est disponible dans le fichier `src/iftun.c`.

5.2 Configuration ansible

On pourrait se demander si une configuration via `ansible` peut permettre de lancer le tunnel, la difficulté de cette tâche est de faire lancer un processus asynchone par `ansible`.

Tout d'abord, `ansible` est doté d'un module `make` qui permet d'exécuter la commande make dans le dossier choisis en spécifiant un champ `chdir`, on ajoute donc les lignes suivantes aux configurations qui en auront besoin.

```
- name: Compilation de l'utilitaire de lancement du tunnel
  make:
    chdir: /mnt/partage/tunnel64d
```

Il est aussi possible de lancer une tache asynchrone dans `ansible` à l'aide des champs `async` et `poll`, exemple :

```
- name: ...
  shell: ...
  async: 50
  poll: 10
```

`async` représente le nombre de secondes que laisse `ansible` à la tâche pour s'exécuter. `poll` représente l'intervalle de temps que va mettre `ansible` à revenir vérifier le statut de la tâche.

La solution serait pour notre cas, avec une valeur 2628000 pour `async`, l'équivalent d'un mois en secondes :

```
- name: Lancement du tunnel
  shell: /mnt/partage/tunnel64d/bin/tunnel64d
  async: 2628000
  poll: 0
```

L'inconvénient de cette option est qu'elle est limitée dans le temps, bien que nous puissions mettre l'équivalent d'un mois en secondes, nous avons cherché une autre façon de procéder.

La commande `UNIX` `nohup` permet de lancer un processus et le rendre persistent même si l'on ferme le terminal où la commande est appellée et même si l'utilisateur qui à exécuté le commande se déconnecte.

On peut utiliser la commande comme suit dans la configuration `ansible` :

```
- name: Lancement du tunnel
  shell: nohup /mnt/partage/tunnel64d/bin/tunnel64d
```

Nous avons enfin remarqué quelque chose d'étrange, en effet, le programme lancé ne se ferme pas si la sortie standard et la sortie d'erreur du programme lancé ne sont pas redirigées, comme ceci :

```
- name: Lancement du tunnel
  shell: nohup /mnt/partage/tunnel64d/bin/tunnel64d >/dev/null 2>&1 &
```

Une autre particularité est le rajout d'un utltime `&` à la fin de la commande, normalement inutile puisque `&` sert à lancer un processus en fond, dépendant en revanche du terminal contrairement à `nohup`. Nous observons qu'`ansible` bloque au lancement du programme et ne veux pas laisser la main au système sans.

Quand aux redirections des sorties, nous pensons qu'ansible ne permet pas à `nohup` de lèguer la main au système tant que la sortie du programme continue d'imprimer, et le programme contient effectivement des appels à la fonction `fprintf` vers la sortie d'erreur.