Dao THAUVIN

Thomas BIGNON

Projet de Programmation Réseau

Dazibao par inondation non-fiable

Sommaire

Introduction

Ici nous vous présentons rapidement ce qui a été fait dans notre projet.

Sujet de Base

Nous avons complètement implémenté le sujet de base du projet et tout à l'air de fonctionner parfaitement.

Extensions du sujet

- Nous avons réalisé 4 extensions et 2 extensions de façon partielle : *Vérification de la cohérence des Node State, Calcul des hashes, Agrégation, Adresses multiples (partiel), Adresses locales au lien* et *Traversée de Firewalls, de NAT (partiel)*.

Autres Extensions

Nous avons implémenté 7 autres extensions au projet qui n'était pas indiqué dans le sujet, la plus part des ajouts sont simples, certains sont très intéressants.
Nous vous laissons la surprise pour plus tard sur ce que sont ces extensions, pour indication nous nous sommes notamment intéressés à la corruption des données reçus et la réduction du trafic des paquets.

Structure du projet

- `main.c`

  Le main qui gère les paramètres, créer un nœud et lance la communication de celui-ci.

- `inondation.c`

  L'inondation dans son ensemble :

  - Une boucle d'évènement dans `startCommunication()` dans laquelle on répond aux différents TLV dans un paquet reçu.
  - Une fonction de réponse pour chaque TLV avec le nom `receiveTLVNameTLV()`.
  - Des petites fonctions pour gérer UDP comme `sendPackage()` qui envoie un paquet.

- `constants.h`

  Un ensemble de constantes dont par exemple le port et l'id de nœud par défauts ou les serveurs STUN avec lesquels on communique.

- `node.c`

  Gère la structure des nœuds et ses voisins, notamment la suppression des voisins inactifs avec `removeVoisinsInactifs()`.

- `tlv.c`

  Gère les TLVs : une fonction par type de TLV pour leurs créations, des getters et aussi des fonctions `verify()` et `verifyIP()` (voir *Adresses locales au lien*) qui permettent respectivement de vérifier un TLV et une IP.

- `package.c`

  Nous utilisons ici le paterne Builder, on ajoute au package des TLVs avec `addTLV()` puis on applique `build()` pour avoir le paquet à envoyer.

  Dans notre structure `package`, les TLVs sont triés par ordre décroissant des types de TLV (voir *Request et Respond*).

  Une fonction `paquet()` permet de créer un paquet à partir d'un datagramme.

- `database.c`

  Gère les données et la base de données. Les données sont stockées dans un tableau dynamique et trié. Les fonctions de hash se trouvent notamment ici. L'accès à ces données peut notamment se faire grâce à un itérateur.

- `history.c`

  On peut stocker des TLV dans un historique (voir *Request et Respond*) qui est géré ici.

  Chaque TLV peut être soit rangé dans l'historique avec la notion `SENT` (envoyé) ou `RECEIVE` (reçu), on stocke aussi le voisin avec qui on a communiqué pour ce TLV.

  La plupart des actions (suppression, recherche) sur les TLVs dans l'historique sont faisables avec des filtres pour :

  - Le type du TLV
  - Le voisin avec lequel on a communiqué.
  - L'id contenu dans le TLV.
  - Si le TLV est marqué `SENT` ou `RECEIVE`.

  Comme pour les données les messages sont accessibles grâce à des itérateurs.

  L'historique est assez complet et ses fonctionnalités ne sont presque pas utilisées dans le reste du code.

- `crypt.c`

  Gère les clés RSA pour les changements de message de notre nœud (voir *TLV 42 Change Message*).

  `generate_keys()` génère des clés publique et privé `public.pem` et `private.pem` dans un répertoire `.key/`.

  `encrypt_message()` et `decrypt_buffer()` permettent respectivement de chiffrer et déchiffrer un message en utilisant les clés locales dans le répertoire `.key/`.

- `change_msg.c`

  Envoie un TLV 42 chiffré permettant de changer de message de nœud (voir *TLV 42 Change Message*).

- `change_zoo.c`

  Petit programme qui modifie toutes les minutes le message du nœud avec un message aléatoire parmi une trentaine, donnant l'heure et un petit message rigolo avec un animal.

- `stun_gestion.c`

Ce fichier permet à partir d'un serveur STUN de récupérer son adresse public IPv4 (voir *Traversée de Firewalls, de NAT*). Ce fichier est inspiré du projet suivant https://github.com/0xFireWolf/STUNExternalIP.

- `print_data.c`

  `Envoie un TLV particulier demandant au nœud de stocker dans un fichier les données que le nœud voit, le programme récupéra et affichera ces données.`

Les Ajouts

Les ajouts sont aussi récapitulés dans le fichier `main.c`.

Extensions du sujet

- *Vérification de la cohérence des Node State*

  La fonction `verify()` vérifie notamment la cohérence du hash et nous envoyons un warning si celui-ci n'est pas correct. Il est calculé dans tous les cas.

- *Calcul des hashes*

  Les hashes sont stockés avec les données pour éviter de les recalculer et le network hash est stocké dans la database, ainsi on ne recalcule les hashes que lors de la modification d'une donnée.

  Une option de compilation du `Makefile` permet d'afficher les hash calculés : `TEST_CALCUL_HASH`.

  L'extension *Vérification de la cohérence des Node State* a aussi été implémenté, on calcul donc aussi un hash à chaque réception de Node State.

  De plus, on ne compare pas les hashes si le numéro de séquence n'est pas le même dans le TLV Node Hash : si le numéro de séquence dans notre base de données est plus faible on envoie un TLV Node State Request, si le notre est plus haut on envoie directement un Node State.

- *Agrégation*

  Un paquet que nous envoyons ne dépasse pas 1024 octets, la limite que tout le monde devrait accepter en pratique.

  A chaque paquet reçu, nous mettons toutes les réponses dans un même paquet tant qu'on ne dépasse pas 1024 octets.

  Ainsi la taille des paquets reçus est optimale, on se limite à une taille raisonnable en essayant de placer le plus de données dans un même paquet.

- *Adresses multiples (partiel)*

  Chacune de nos structures de voisin stockent une adresse qui correspond à l'adresse avec laquelle on a communiqué avec le voisin, qui sera donc réutilisé pour les autres communications avec ce voisin.

  Nous utilisons les messages ancillaires vu dans le cours 10 pour cela.

  Deux problèmes se posent ici qui empêchent d'avoir une seule fois un nœud en voisin :

    - Si un pair à plusieurs adresses IP et n'a pas cette extension, il est impossible de relier ses différentes adresses, ainsi il est possible d'obtenir deux fois le même nœud en voisin.
    - Même si tout le monde à cette extension, il est possible que nous communiquions à deux nœuds différents avec deux adresses différentes, si un TLV Neighbour contenant une de nos IP transite entre les deux nœuds, il est impossible que le nœud ayant reçu le TLV sache qu'il a déjà ce voisin, ainsi celui-ci aura deux fois notre nœud en voisin. Le même raisonnement est valable sur notre nœud.

  Le fait d'ignorer nos propres adresses n'est pas un problème dans le fonctionnement du protocole mais celui-ci augmente le trafic, ce qui n'est pas vraiment voulu. Ainsi nous ignorons nos propres adresses IPv6 mais aussi IPv4 obtenu grâce à un serveur STUN.

- *Adresses locales au lien*

  La fonction `verifyIP()` vérifie si on est en face d'une adresse local au lien.

  Pour cela nous utilisons `IN6_IS_ADDR_LINKLOCAL` et `IN6_IS_ADDR_MC_LINKLOCAL`.

  Si c'est le cas pour l'envoyeur, on ignore le paquet et si on reçoit une adresse locale au lien dans un TLV, on envoie un warning.

  Les adresses Loop Back ne sont pas traitées simultanément, en effet il doit être possible de s'envoyer soit même un TLV permettant de changer de message. Ainsi les adresses Loop Back sont considérées comme notre propre adresse (si le port est respecté) et seul les TLV permettant de changer de message seront traités. Dans le cas de la réception d'une telle adresse dans un TLV, celle-ci est traité comme une adresse locale au lien.

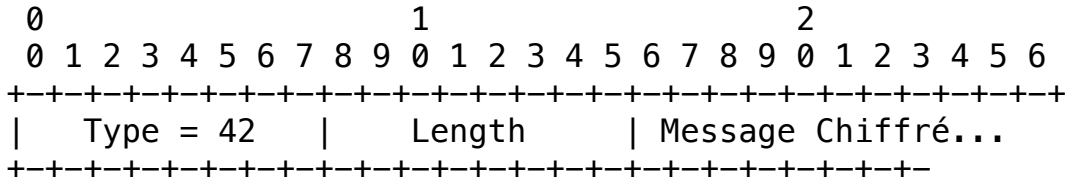
- *Traversée de Firewalls, de NAT (partiel)*

  Nous utilisons des serveurs STUN pour récupérer notre adresse IPv4 public même derrière un NAT, ainsi il nous est possible d'ignorer notre adresse IPv4 si nous nous envoyons un message à nous-même.

Autres Ajouts

- *TLV 42 Change Message*

  Nous avons ajouté un TLV permettant de modifier notre donnée.



  Le TLV contient un message chiffré par une clé RSA qui sera déchiffré par notre nœud, si le chiffrement n'était pas correct on l'ignore, sinon on change le message et les clés. Les fonctions de chiffrement et déchiffrement se font dans le fichier `crypt.c`. Les clés sont conservées en local dans un dossier `.key/`.

  Il reste possible de donner la clé à un tiers pour lui permettre de modifier la valeur du nœud, mais cette clé ne sera utilisable qu'une seule fois.

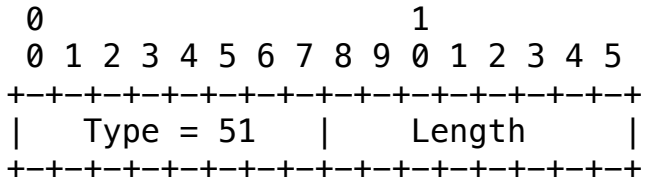
  La bibliothèque OpenSSL a été utilisée pour cela.

  Ainsi `change_zoo` et `change_msg` permettent d'envoyer de tels messages à un nœud.

  Un autre problème s'est posé, comment éviter l'ajout en tant que voisin lors de l'envoie de ce TLV ?

  En réalité ce n'est pas vraiment un problème, le voisin disparaitra après 70 secondes mais nous avons tout de même choisis de mettre un port par default sur les messages des deux scripts (`CHANGE_PORT`), si le message vient de notre propre machine et que le port est respecté alors il ne sera pas ajouté en voisin.

- *TLV 51 Stock*



  Un TLV demandant à notre nœud de stocker les données qu'il voit (sous un format hexadécimal pour éviter les messages avec des caractères spéciaux) et répond par un TLV Pad1 pour expliquer que le message a bien été reçu. Pour éviter de bloquer le code le temps de l'écriture des données, un fork est

réalisé. Mais il faut aussi éviter que plusieurs écritures se fassent en même temps, pour cela nous prenons 2 précautions :

- Le TLV n'est traité que s'il s'agit de nous-même.
- Un verrou est placé sur le fichier pour éviter d'écrire plusieurs fois le fichier simultanément (ce qui rend le TLV idempotent).

Ainsi pour lire le fichier, il suffit de:

- Envoyer un TLV 51
- Attendre la réponse
- Après réception, attendre que le verrou soit disponible.
- Lire le fichier

Dans notre programme, le fichier est alors supprimé.

- *Request et Respond*

  On renvoie un TLV Request sans réponse après 20 secondes, pour cela nous les stockons dans un historique.

  Si un TLV est reçu en réponse à un TLV Request alors celui-ci est retiré de l'historique, si le TLV Request correspondant n'est pas dans l'historique, on ignore la réponse.

  Pour éviter de traiter une réponse d'un TLV Request qui a été envoyé en réponse à ce même paquet, on traite les TLV d'un paquet en ordre décroissant des types de TLV.

  Un tel comportement n'est pas vraiment voulu en réalité car cela rend le protocole plus rigide mais cela permet aussi d'éviter à un certain point la corruption des données (si quelqu'un se contente d'envoyé des TLV Node States fantaisistes, alors ils seront ignorés).

  Certains TLV Request ne sont pas renvoyé toutes les 20 secondes, en effet le TLV Network State Request n'est jamais enlevé de l'historique car plusieurs réponses sont attendues (on limite le nombre de TLV Network State Request à 1 par voisin) et le TLV Neighbour Request sera renvoyé si nous n'avons toujours pas 5 voisins, il n'est donc pas ajouté à l'historique.

  Les TLV sont automatiquement enlevés de l'historique si ont perd le voisin associé.

- *Des warnings en plus*

  Nous avons ajouté des warnings pour :

  - Les tailles de paquet trop grandes par rapport à la taille du datagramme reçu.

    - Les tailles de TLV qui ne correspondent pas à leur taille donnée dans le sujet.
    - Les TLV dont la taille indique une taille dépassant la place restante dans le paquet.

  Deux autres warnings sont indiqués dans *Adresses locales au lien* et *Vérification de la cohérence des Node State.*

  Il est possible de tester la plupart des warnings en dé-commentant les commentaires (il faudra changer de port de bind sinon les TLV warnings ne seront pas envoyés) dans le code change_msg et de lancer un dazibao_msg après compilation.

- *Réception des Warnings*

  Chaque warning reçu est stocké dans un fichier `log_file` sous la forme :

  ```
  IP Warning
  ```

  Où `IP` correspond à l'IP qui a envoyé le warning.

- *Suppression de débit*

  Notre structure stockant les paquets reçus n'ajoute pas les TLV qui sont déjà présent dans celui-ci (sauf dans le cas où un ID de nœud est précisé dans le TLV, il faudra alors que l'ID soit le même). Ainsi les doublons ne sont pas pris en compte dans notre code ce qui peut permettre de réduire le nombre de paquet à envoyer et à traiter.

  On évite aussi l'envoie de TLV network Hash lors de la réception d'un TLV Neighbour si celui-ci est déjà notre voisin ou qu'il s'agit de nous-même.

- *Voisins permanents multiples*

  Il est possible de donner plusieurs sources de voisins permanents au début du programme, nous ajoutons en voisins permanents toutes les adresses disponibles même si cela n'est pas en accord avec l'extension *Adresses multiples* (cela permet vous permettra de tester plus facilement la communication IPv4)

Exemple

Ici nous donnerons un exemple d'utilisation avec deux executions parallèles de notre programme qui communiquerons ensemble.
Nous appellerons dans la suite les deux instances du programme : serveur et client, où serveur est la première instance lancée. On compile avec les options de compilations :

```
-D TEST_SEND -D TEST_VOISINS -D TEST_DATA
```

On lance le serveur avec le seqno, l'id de noeud et le port par défaut :

```
./dazibao_node -n
```

On stocke les messages dans un fichier `serveur` pour pouvoir les lire plus facilement.

```
--- VOISINS ---
--- ------- ---
--- DataBase ---
Network Hash : 1f7d971b0b9d621721bde1f01683b115
- Size : 1
- Data :
     ID : 9a02000000000000
     Seqno : 0

     Node Hash : 1a9e1d48517e216f34985ae7e236469a
--- -------- ---
```

On observe qu'il n'y a aucun voisin et que notre donnée correspond à un espace vide.
Il est temps de lancer le client !
On utilise les options de compilation suivantes :

```
-D TEST_SEND -D TEST_VOISINS -D TEST_DATA
```

On le lance avec l'id de nœud 5 et le port 41248 et on stock le résultat dans un fichier `client` :

```
./dazibao_node -d -i 5 -p 41248 2a01:e35:2fba:21b0:ed7c:55ff:
140f:b274 42069
```

L'IP doit correspondre à l'IP d'où est lancé le premier nœud, `42069` correspond au port par defaut (qui est donc le port du serveur).
Remarque: ici on peut être notre propre ami car nous sommes sur des ports différents.

```
VOISINS : one more neighbour
VOISINS : number of neighbour : 1
VOISINS : number of permanent neighbours : 1
Seqno : 0
Node ID 5 (0500000000000000)
Port : 41248
--- VOISINS ---
- 2a01:e35:2fba:21b0:ed7c:55ff:140f:b274 : 42069
--- ------- ---
--- DataBase ---
Network Hash : baf94661082cf8fec47a28c81047c968
- Size : 1
- Data :
     ID : 0500000000000000
     Seqno : 0

     Node Hash : a09c0eb5f6573e91903df6f638f627c7
--- -------- ---
```

Après cette affichage la valeur de notre nœud va être mis à jour avec `I see 0001 data in this database` grace à l'option d'execution `-d.`

On observe bien que l'on a un voisin sur le port 42069 qui correspond à notre autre noeud, pour l'instant nous n'avons pas sa donnée.
Le premier réflexe du client sera d'envoyé un TLV network et neighbour request :

```
SEND : TO 2a01:e35:2fba:21b0:ed7c:55ff:140f:b274
--- SEND ---
-> Magic : 95
-> Version : 1
-> Size : 18
-> TLV :  type 4 of size 18
--- ---- ---
VOISINS : number of neighbour : 1
SEND : TO 2a01:e35:2fba:21b0:ed7c:55ff:140f:b274
--- SEND ---
-> Magic : 95
-> Version : 1
-> Size : 2
-> TLV :  type 2 of size 2
--- ---- ---
```

Le serveur va bien sur répondre par un TLV de type 5 (State Request) et un TLV de type 3 (Neighbour).

```
--- SEND ---
-> Magic : 95
-> Version : 1
-> Size : 2
-> TLV :  type 5 of size 2
--- ---- ---
SEND : TO 2a01:e35:2fba:21b0:ed7c:55ff:140f:b274
--- SEND ---
-> Magic : 95
-> Version : 1
-> Size : 20
-> TLV :  type 3 of size 20
--- ---- ---
```

Remarque : dans notre exemple les TLV Neighbour sont tous ignorés car on ne répond pas à ceci s'il s'agit de nous même ou qu'il s'agit d'un de nos voisins.

Les deux programmes vont alors communiquer jusqu'a ce que leurs données convergent.

```
--- DataBase ---
Network Hash : a859c0ff9a8a0df6381045d316630b6b
- Size : 2
- Data :
     ID : 0500000000000000
     Seqno : 1
     I see 0001 data in this database
     Node Hash : 10c2d88b243badf5a27aa6b7ff9337ca
- Data :
     ID : 9a02000000000000
     Seqno : 0

     Node Hash : 1a9e1d48517e216f34985ae7e236469a
--- -------- ---
```

Puis le client va changer de message car il a maintenant deux données dans sa base de données, son message sera alors : `I see 0002 data in this database`.

Ce changement va mener au serveur à envoyer un TLV Network State Request lors de la réception d'un TLV network hash (ici on simplifie un peu l'exemple car il est très probable que le client le demande aussi).

On observe bien une réponse du client avec 2 TLV de type 6 (Node Hash) :

```
SEND : TO 2a01:e35:2fba:21b0:ed7c:55ff:140f:b274
--- SEND ---
-> Magic : 95
-> Version : 1
-> Size : 10
-> TLV :  type 7 of size 10
--- ---- ---
```

Vu que la donnée du serveur n'a pas changé, il n'enverra qu'un seul TLV Node State Request.

```
SEND : TO 2a01:e35:2fba:21b0:ed7c:55ff:140f:b274
--- SEND ---
-> Magic : 95
-> Version : 1
-> Size : 56
-> TLV :  type 6 of size 28
-> TLV :  type 6 of size 28
--- ---- ---
```

Le client va alors recevoir cette demande, envoyer un TLV Node State et les données des deux programmes ont convergé.

J'envoie maintenant une nouvelle donnée au serveur avec dazibao_msg :

```
./dazibao_msg 42069 "C'est une question de timing"
```

On observe bien que la donnée change pour le serveur, et, client et serveur convergent de nouveau.

```
--- DataBase ---
Network Hash : fa01853d83bad49a22f72bf8e7578992
- Size : 2
- Data :
     ID : 0500000000000000
     Seqno : 2
     I see 0002 data in this database
     Node Hash : 743d2852097e661dd800b1c743d1d3d5
- Data :
     ID : 9a02000000000000
     Seqno : 1
     C'est une question de timing
     Node Hash : 5f9fd41402d3ee86c46d3bbcd5afca28
--- -------- ---
```

On observe aussi, si on laisse tourner le programme, qu'il n'y a plus de TLV de type plus grand que 4 qui sont envoyés.
Voilà l'exemple s'arrête ici, le but de cet exemple était de montrer que notre projet fonctionne, de vous familiariser avec l'affichage et les options de compilation et d'exécution de notre code.