**Rapport de Projet de Réseau**

**Sommaire :**

**1. Introduction**

**2. Le Makefile**

**3. Outils utilisé pour le développement**

**4. Conditions d’utilisation**

**5. Brève description du déroulement du programme**

**6. La partie sur les TLV**

**7. Partie communication**

**8. La découverte des voisins**

**9. L’inondation**

**10. Extensions ajoutés**

**11. Ce qui reste à faire**

**1. Introduction :**

Le programme est composé de deux grandes parties relativement indépendantes, la partie implémentation du protocole à savoir tout ce qui concerne les TLV (le dossier protocole) qui contient les fonctions qui traitent les chaines de caractères,

Afin de mettre à  disposition tous les outils nécessaires pour pouvoir transformer un message en TLV et vice-versa, l’autre partie va contenir tout ce qui est utile à  la communication, donc la partie réseau du projet.

Le fichier **main.c** contient l’entrée du programme et la boucle principale.

Au même niveau que le répertoire **src** on trouve le répertoire **test** qui contient les tests unitaires.

**2. Le Makefile :**

Le Makefile principal est dans le même répertoire (**src**), que le fichier **main.c** donc pour lancer le programme il suffit de taper la commande **make all** ensuite **. /out ;**

Mais on peut constater qu’on a également des sous-Makefiles dans chaque sous-répertoire dans le but de créer un Makefile récursif (qui visite chaque sous répertoire et exécute les commandes correspondantes à chaque dossier), on peut également constater qu’on a créé des librairies qui seront incluses à la compilation et ce à des fins d’extensibilité et de maintenabilité du programme qu’on a choisis d’adopter de choix d’architecture.

Pour lancer les tests unitaires il faut se rendre dans le répertoire **tests** etexécuter la commande **make all** ensuite **. /out.**

**3. Outils utilisés pour le développement :**

Os : Ubuntu

IDE : Visual Studio Code

Versioning System : GitLab

Units Tests : la librairie Check

Visualisation du trafic réseau : Wireshark

**4. Conditions d’utilisation :**

* Avoir un système Linux
* Le port 5000 doit être libre
* Etre capable de se connecter à l’interface jch.irif.fr :8082

**5. Brève description du déroulement du programme :**

Quand on exécute le programme il faut attendre un instant et un message **Connected** sera affiché si la connexion est bien établie, dans le cas contraire il faut quitter le programme en tapant **exit** et le relancer.

Pour envoyer des messages et il suffit de le taper et d’appuyer sur **entrer,** en effet on a utilisé le descripteur de fichier de l’entrée standard pour récupérer les messages tapés aux claviers, une description plus détaillée sera donner un peu plus bas.

Les messages reçus seront affichés au terminal ainsi que les messages d’erreurs s’il y en a.

**6. La partie sur les TLV :**

Comme cité dans l’introduction les fonctions qui traitent des chaines de caractères sont totalement isolées de la partie réseau du projet ceci afin d’avoir une bonne cohésion et limiter au maximum les interdépendances entre les modules.

Voici une description de chacun des modules et de leurs utilités :

* **Protocole/header/header.h :**

C’est le module au dessus de tous les modules, il contient tous les **include**, et l’ensemble des primitives qui sont globales.

* **Protocole/tlvs/tlvs.h :**

Contient les fonctions qui manipulent les structures :

* **TLV1, TLV2,..., TLV7**: qui représente chacune un type de message
* **TLV** : qui est en quelque sort un type abstrait et générique à toutes les autres structures de type TLVn / n = {1,..,7}, en d’autres termes par exemple une TLV1 est aussi une TLV.

L’intérêt résulte dans le fait que quand on reçoit un message, celui-ci est d’abord stocké dans une structure TLV générale (ou une liste chainée de TLV), avant d’être converti, ne sachant pas au départ quel est le type de la TLV reçu ce mécanisme nous permet de simuler du polymorphisme, et de la même manière quand on peut construire une liste de TLVn/ n = {1,..,7}, et de la transformer en une liste de TLV, dans le module on retrouve donc les fonction qui permettent de faire ces transformations dans les deux sens.

* **Protocole/tlvs/create\_message.h :**

Contiens les fonctions qui transforment une liste de TLV en un message (un tableau de caractères), en rajoutant l’entête, et de faire l’inverse, à partir d’un message construire une liste de TLV, en vérifiant bien que le message respecte bien les critères du protocole, autrement le message sera ignoré.

**7. La partie réseau :**

Voici une description de chaque module et de son utilité :

**pair/data\_pair/data\_peer.h :**

C’est le module principal de cette partie du réseau, il contient les structures et des fonctions qui permettent de faire de la découverte de voisin ainsi que l’inondation, nous allons décrire en détaille chaque structure :

La structure **contact** :

Stocke l’IP et le port d’un contacte, on retrouve un ensemble de fonctions qui lui y sont dédiés comme l’ajout, la suppression, d’un contacte

La structure **neighbour** :

Stocke les informations relatives un voisin, comme l’heure à la quelle il a été contacté la dernière fois, s’il s’agit d’un voisin symétrique ou s’il est récent … et sans oublier un pointeur vers **contact**, on trouve aussi des fonctions qui manipulent cette structure.

La structure **data** :

Va contenir le message à inonder, ou à afficher, ainsi que des informations relatives comme une liste de **neighbour**, le nonce, …etc., un ensemble de fonctions sont dédiés à la manipulation de cette structure.

La structure **data\_neighbour** :

Cette structure contient un pointeur vers une liste de **neighbour** et un **compteur** qui va contenir le nombre de fois qu’une data lui a été envoyé, en effet l’existence de cette structure n’a pour but que de palier à un problème de complexité, on ne peut pas mettre l’attribut **compteur** dans la structure **neighbour**, étant donné que chaque **data** contient une liste de **neighbour** à qui la data vers être inondée, si on met l’attribut **compteur** dans la structure **neighbour** cela nous obligerait à réallouer l’espace mémoire de toute la liste de **neighbour** pour chaque **data**, car évidemment pour chaque **neighbour** et pour chaque **data** on retrouve le nombre de fois que la **data** lui a été envoyé, et comme on risque d’avoir une liste de **data** assez longue , l’espace mémoire alloué risque de devenir rapidement problématique, donc une solution simple mais un peu duplicative, est de créer une autre structure **data\_neighbour**, ce qui nous dispense de réallouer de la mémoire en séparant l’attribut **compteur** de **neighbour**. Un ensemble de fonctions existent pour manipuler cette structure.

* **pair/emet/emeteur.h :**

Contient les fonctions de récupération des IP à partir d’un nom d’hôte et d’envois de message.

* **pair/recept/reception.h :**

Il n’est pas utilisé dans le projet, mais on l’a rajouté pour des raisons d’évolutivité.

**8. La découverte des voisins :**

On tient dans le module **data\_peer.h** vu précédemment, une liste de voisins (**neuighbour**) ainsi qu’une liste de voisins potentiels (**contact**), Quand on reçoit une **TLV3,** on extraits le contact et on l’ajout à la liste des voisins potentiels, si en plus la liste des voisins est inférieure à 8, alors on lui envoie un hello long, si on reçoit un hello court ou long on l’insère dans la liste des voisins.

Pour maintenir la liste des voisins, chaque 30 secondes on envoie un hello long à tous nos voisins :

// copier la liste des voisins dans une autre variable

voisin = \_voisins;

while (voisin != NULL)

{

// récupérer l’heure courant

time\_t current\_time = time(0);

if ((current\_time - voisin->contacted\_at) > 30)

{

t = createTLV2(\_getSource\_id(), voisin->id);

mess = init\_message(t);

addr.sin6\_port = voisin->neighbour->port;

str = smalloc(40);

            //convertir l’ip en chaine de caractères

ipv6\_to\_str\_unexpanded(str, voisin->neighbour->ip);

inet\_pton(AF\_INET6, str, &addr.sin6\_addr);

send\_msg(sock, &addr, mess);

//on met à jour l’heure à laquelle on l’a contacté

voisin->contacted\_at = current\_time;

}

voisin = voisin->next;

}  

**9. L’inondation :**

On parcourt la liste des **data** stockée dans une structure **\_contents** dans le module **data\_peer.h**, on extraits la liste de voisins symétriques de chaque data, si la data existe depuis plus de 5 min ou si la liste des voisins est vide alors elle sera supprimée et on passe à la data suivante, et on parcourt pour chaque data sa liste de voisins, pour chaque voisin, si le nombre de fois qu’on lui a envoyé cette data est supérieur à 5 alors on lui envoie un message d’avertissement avant de le supprimer de la liste de cette data, et on passe au voisin suivant, sinon si le temps écoulé depuis qu’on lui a envoyé la **data** est compris entre 2 n−1 et 2 n secondes, où n est le compteur, alors on lui envoie la **data**.

//inondation

contents = \_contents;

while (contents != NULL)

{

// on récupère la liste des voisins d'une data

data\_voisin = contents->voisins\_sym;

temps\_actu = time(0);

// supprimer la data au bout de 300 sec/5 min ou si la liste est vide

if (((temps\_actu - contents->received\_at) > 300) || (data\_voisin == NULL))

{

delete\_data(&\_contents, contents->sender\_id, contents->nonce);

goto cont;

}

t = createTLV4(contents->sender\_id, contents->nonce, 0,

contents->content);

mess = init\_message(t);

// parcourir la liste des voisins. Rq : il s'agit de struct data\_neighbour \*, qui est composée de voisin (neighbour \*) et le compt

while (data\_voisin != NULL)

{

// récuperer le voisin

voisin = data\_voisin->voisin;

// puts("how many time we were here ?");

// supression du voisins de la liste data\_neighbours, à tester

if (data\_voisin->compt > 5)

{

// puts("are we here ?");

// printf("nonce : %d \n", contents->nonce);

t = createTLV6(2, "je n'ai pas été acquitté depuis longtemps");

mess = init\_message(t);

addr.sin6\_port = voisin->neighbour->port;

str = smalloc(40);

ipv6\_to\_str\_unexpanded(str, voisin->neighbour->ip);

inet\_pton(AF\_INET6, str, &addr.sin6\_addr);

send\_msg(sock, &addr, mess);

delete\_data\_neighbour(&contents->voisins\_sym,

\*voisin->neighbour);

goto next\_data\_neighbour;

//TODO: ajouter au voisins potentiels

}

// si le voisin est sym

if ((data\_voisin->voisin->sym == TRUE) &&

(contents->received\_at +

((int)pow(2, data\_voisin->compt - 1)) <= temps\_actu) &&

(contents->received\_at +

((int)pow(2, data\_voisin->compt)) >= temps\_actu))

{

// puts("we never been here ?");

addr.sin6\_port = voisin->neighbour->port;

str = smalloc(40);

ipv6\_to\_str\_unexpanded(str, voisin->neighbour->ip);

inet\_pton(AF\_INET6, str, &addr.sin6\_addr);

send\_msg(sock, &addr, mess);

data\_voisin->compt++;

}

next\_data\_neighbour:

data\_voisin = data\_voisin->next;

}

cont:

contents = contents->next;

}

Si on reçoit un acquittement de la part d’un voisin on le supprime de la liste des voisins à inonder pour cette data :

case 4:

t4 = getTLV4(t);

/\* si c'est un ami sym, et que la data est déjà dans la liste, on considère ça

comme un acquittement et on le supprime de la lise des voisins à inonder pour cette data

\*/

if (exist\_data(\_contents, t4->sender\_id, t4->nonce))

{

d = \_contents;

while (d != NULL)

{

if (d->sender\_id == t4->sender\_id && d->nonce == t4->nonce)

{

delete\_data\_neighbour(&d->voisins\_sym, \*c);

}

d = d->next;

}

}

else

{//Le reste du programme

**10. Extensions :**

* Le programme marche aussi bien en ipv4 qu’en ipv6
* Le programme fait la segmentation des messages de taille supérieurs à 255
* Le programme envoie un message de taille supérieure à 255
* Le programme tient en compte de la gestion des erreurs
* Le programme ne bloque pas sur un pair quand il fait l’inondation
* Le programme tient compte de Sécurité de l’implémentation

**11. Ce qui reste à faire :**

* Données non textuelles
* Contrôle de la congestion
* Transfert de fichiers
* Traversée de Firewalls, de NAT
* L’interface graphique