15 Mai 2020

Projet de Programmation Réseau
Dazibao par inondation non-fiable

Auteurs:

CHEKROUN Hiba-Asmaa
MIRET Blanche

Tables des matières:

1. Vue d'ensemble du programme

Utilisation

Un Makefile permet la compilation par un simple **`make`**. Le programme se lance avec `./dazibao <hostname> <port> [debug]`

Installation

Vous aurez besoin d'installer les bibliothèque *glib*

- Sur Linux

```
sudo apt-get install libglib2.0-dev
```

- Avec HomeBrew sur MacOs :

```
brew install glib
```

et *openssl*

- Sur Linux

```
sudo apt-get install libssl-dev
```

- Avec HomeBrew sur MacOs :

```
brew install openssl
```

Puis le lier manuellement :

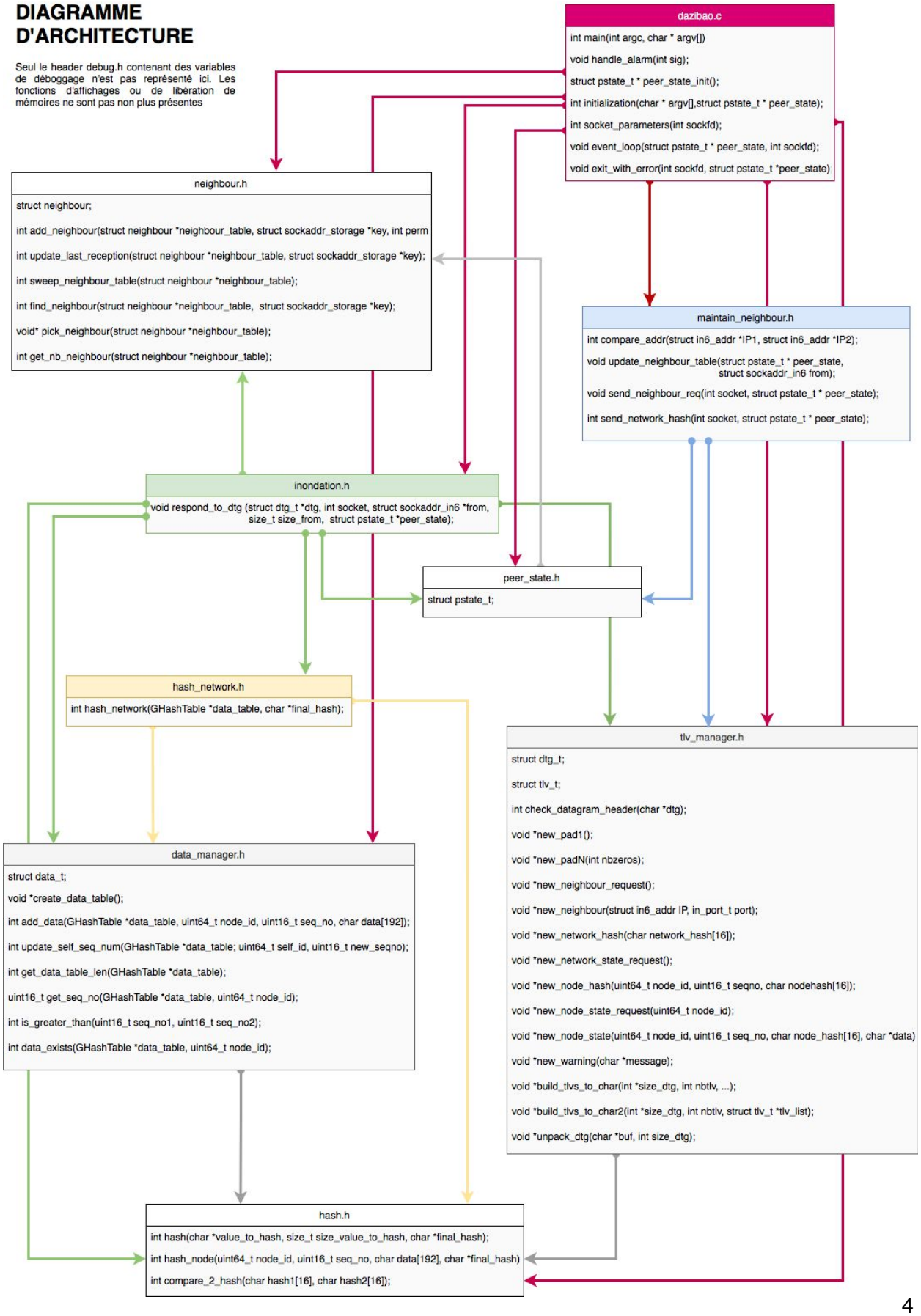
```
ln -s /usr/local/opt/openssl/include/openssl /usr/local/include
ln -s /usr/local/Cellar/openssl/[version]/include/openssl
/usr/bin/openssl
ln -s /usr/local/opt/openssl/lib/libssl.1.0.0.dylib /usr/local/lib/
```

Diagramme

Ci-dessous, le diagramme d'architecture du programme. Les couleurs servent uniquement à le rendre plus lisible. Les fonctions listées pour chaque module sont celles de leur interface uniquement, seul dazibao, qui contient le main, ne possède pas de header.

DIAGRAMME D'ARCHITECTURE

Seul le header debug.h contenant des variables de déboggage n'est pas représenté ici. Les fonctions d'affichages ou de libération de mémoires ne sont pas non plus présentes



Description des modules

- **dazibao.c** : ce module contient le main et appelle les différentes fonctions des autres fichiers. Il sert à initialiser la table de données, des paramètres réseau, ajouter un voisin permanent, envoyer un premier TLV et lancer la boucle à événements principale.

- **debug.h :** contient la variable globale debug utilisée dans plusieurs modules pour avoir un mode debug.

- **hash.c:** contient les différentes fonctions du calcul du hash d'un noeud, et une fonction qui affiche le hash.

- **hash_network.c:** module qui gère le calcul du hash réseau.

- **inondation.c:** ce module gère le protocole d'inondation : la fonction principale utilisée dans dazibao.c est "*respond_to_dtg*", elle prend en argument le datagramme reçu (et autres variables) et répond à chaque TLV contenu dans le datagramme selon le protocole d'inondation.

- **maintain_neighbours.c:** module qui gère la partie maintenance de la table des voisins.

- **neighbour.c:** contient les fonctions qui gèrent la table de voisins tel que l'ajout d'un voisin, sa suppression, l'affichage de la table de voisins, mise à jour de la date de dernière réception...etc

- **peer_state.h:** contient une structure nommée pstate_t qui regroupe la table de données, la table de voisins, l'identifiant, le numéro de séquence, les différents hashes, et la donnée.

- **tlv_manager.c:** contient les différentes fonctions qui créent les TLVs et qui les affichent, ainsi qu'une fonction qui vérifie l'en-tête d'un datagramme.

Regard critique sur le programme

L'intégralité du sujet minimal a été implémenté, de façon assez littérale. Une attention a été portée à l'agrégation des TLV's dans un seul paquet, au moins en réponse à chaque réception : si un datagramme contient plusieurs TLV's demandant des réponses différentes, elles sont envoyées dans le même paquet. Tout semble fonctionner bien que le projet n'a pas été testé sur un circuit privé.

Un point qui pourrait être amélioré est la gestion des erreurs : le suivi des erreurs à travers les fonctions et leur interprétation, une meilleure gestion des erreurs réseau...

---

2. Structures et techniques employées

Table des voisins : tableau de structures

> module *neighbour*

La structure utilisée pour stocker les voisins est un simple tableau. Les éléments sont des structures *struct neighbour* contenant chacune l'adresse de la socket correspondante et l'heure de dernière réception d'un paquet de ce voisin. Deux champs de type *int* traités comme des booléens indiquent d'une part si le voisin est permanent, d'autre part s'il existe. Ce dernier paramètre permet de "supprimer" un voisin ou d'en "créer" un, en indiquant si la structure est active ou non. Les opérations sur la table des voisins s'effectuent ainsi en parcourant les 15 cases du tableau, qu'il s'agisse de compter le nombre de voisins ou de désactiver ceux qui seraient devenus obsolète - en mesurant l'écart entre l'heure courante et l'heure de dernière réception.

Table des données : hashmap

> module *data_manager*

Les données recueillies au fur et à mesure que se déploie le protocole d'inondation sont conservées dans une table de hachage, donnant l'avantage de l'accès en temps constant aux valeurs de la table. Les clés sont les node_id des pairs, les valeurs des *struct data_t* retenant le numéro de séquence de la donnée - stocké en ordre réseau, la donnée elle-même, et le hash du noeud. Celui-ci est calculé à chaque ajout de donnée, recalculé si la donnée est mise à jour.

La table de hachage est implémentée avec le type *GHashTable* de la bibliothèque externe Glib. Un mot sur la gestion de la mémoire : de la mémoire est allouée dynamiquement

aux éléments avant chaque ajout à la table ; mémoire automatiquement libérée aux moments opportuns. Par exemple, la fonction ajoutant une donnée ou la mettant à jour est la même : dans les deux cas de la mémoire est allouée à la clé et à la valeur passées en paramètres, mais s'il s'avère qu'une clé égale est déjà répertoriée dans la table de hachage, la mémoire de la nouvelle clé et de l'ancienne valeur sont libérées.

L'utilisation d'une telle structure implique cependant de trier les données par identifiant à chaque calcul du hash du réseau, c'est à dire dès qu'une donnée est ajoutée ou modifiée. À ce moment, c'est le module *hash_network* récupère tous les identifiants de noeuds, les trie, puis concatène chaque valeur associée à un id avant de hacher le tout.

La donnée publiée par le pair est présente dans la table des données.

Communiquer les connaissances du pair : peer_state

> header *peer_state*

Le pair introduit dans le réseau au lancement du programme conserve notamment tout au long de son temps de vie les deux structures décrites précédemment ainsi que son propre identifiant. Pour faciliter la transmission de ces informations, nécessaires à différents modules et pour différentes fonctions que ce soit en lecture ou en écriture, le header *peer_state.h* définit une structure contenant l'intégralité de ces données. La variable *peer_state* de type *struct pstate_t* est initialisée au démarrage et transmise en paramètre de fonction quand nécessaire.

Interprétation et création des paquets

> module *tlv_manager*

Afin de manipuler plus aisément les TLVs, au moment de la réception et de l'envoi de paquet, sont définies dans le header *tlv_manager.h* la structure *struct tlv_t*, l'union *union tlv_body*, et une série de structures représentant les corps des différents types de TLVs. *Struct dtg_t* contient une liste chaînée de *struct tlv_t* grâce au champ *struct tlv_t *next*.

À la réception d'un paquet et après vérification sommaire de son en-tête, le datagramme est déballé grâce à la fonction *unpack_dtg*. Dans celle-ci plusieurs variables permettent de procéder à l'interprétation : le pointeur *buf* fait office de tête de lecture sur le paquet, *decount* comptabilise le nombre d'octets restant à lire, *tlv* est un pointeur de pointeur et sert de tête d'écriture pour la liste chaînée de TLVs à créer. S'il n'excède pas le nombre d'octets restant à lire, la fonction *unpack_next_tlv* lit le nombre d'octets annoncé pour un TLV donné et, selon son type, le traduit en une structure *struct tlv_t* dont le pointeur est renvoyé. À la sortie de la fonction, le TLV est ajouté à la liste chaînée, *decount* est décrémenté de la taille du TLV, la tête de lecture déplacée en conséquence. L'opération est répétée tant que decount n'atteint pas zéro ou qu'une erreur ne survienne.

Un paquet est ignoré dans son intégralité si son en-tête est incorrect, si la taille annoncée du datagramme est supérieure à la taille réelle, si la taille annoncée d'un TLV dépasse le nombre d'octets restant dans le paquet, ou si le TLV est de type inconnu.

En écho aux différents types de corps de TLV il existe une fonction d'initialisation de structure *struct tlv_t* pour chacun de ces types, prenant en paramètres les données adéquates. Deux fonctions *build_tlvs_to_char* se chargent de créer le datagramme à envoyer en commençant par une en-tête correct, puis à la suite les informations contenues dans les TLVs à envoyer. Deux versions de *build_tlvs_to_char* permettent des utilisations différentes : la première prend un nombre indéfini d'argument, la seconde une liste chaînée de TLVs. Le pointeur renvoyé en fin de fonction pointe vers un paquet prêt à être envoyé sur le réseau. Ces deux fonctions inscrivent également la taille finale du datagramme créé à un pointeur d'entier passé en paramètre.

Boucle à évènements pour l'inondation

Nous avons utilisé dans ce projet une boucle à événements avec gestion de signal afin de gérer la réception et l'envoi des paquets, ainsi que les actions répétées périodiquement.

Pour les actions qui doivent être exécutées chaque 20 secondes, nous avons choisi d'utiliser la fonction alarm() qui programme une temporisation pour envoyer un signal SIGALRM au processus en cours après 20 secondes. Un gestionnaire d'événements a été également ajouté, ce dernier va signaler la capture d'un signal à l'aide d'une variable globale " volatile sig_atomic_t alarm_val " qui indique que l'événement doit être traité dans la boucle principale.

Inondation : répondre aux paquets reçus

> module *inondation*

Il s'agit de répondre correctement aux TLVs reçus, sans gaspiller trop de datagrammes peu remplis et l'interface du module *inondatio*n fournit une unique fonction *respond_to_dtg* responsable de cela. Elle prend en arguments une structure *struct dtg_t*, les paramètres réseau - l'adresse de l'auteur du datagramme reçu, le descripteur de socket - et, via le paramètre *peer_state*, les données connues nécessaire à la constitution des réponses.

La liste chaînée de TLVs reçus, contenue dans la structure *struct dtg_t*, est parcourue en appelant la fonction de réponse correspondant au type de TLV pour chaque élément de la liste. Sauf exception pour certains types requérant un traitement différent - détail plus bas - ces fonctions construisent un TLV de réponse selon le protocole d'inondation et le renvoient jusqu'à *respond_to_dtg* qui les accumule dans une liste chaînée de TLVs, liste à retourner à l'envoyeur du paquet. Si un nouveau TLV de réponse ne peut être ajouté à la liste sans conduire au dépassement de la taille maximale d'un paquet - 1024 octets - alors la liste précédemment

construite est envoyée sur le réseau, sa mémoire libérée, le TLV “de trop” placé en tête d’une nouvelle liste ; et ainsi de suite jusqu’à avoir traité tous les TLVs contenus dans le datagramme reçu.

Certains types de TLVs sont traités différemment, c’est le cas des Network State Request dont la réception entraîne l’envoi d’une longue série de Node Hash. Cet envoi particulier est traité directement dans la fonction de réponse à ce type de TLV, par paquet de 35. La réponse aux Neighbour est également réalisée directement dans la fonction attribuée à ce type, la réponse dans ce cas ne se faisant pas à l’envoyeur du paquet mais à l’adresse de socket contenue à l’intérieur du TLV lui-même. Enfin, la réception d’un Node State demande de procéder à certaines vérifications et éventuellement un ajout ou mise à jour de donnée, sans envoyer de réponse. Les fonctions se différencient par leur nom avec les préfixes “build_res_to” quand la fonction renvoie un TLV de réponse, “respond_to” dans le cas contraire.

Debug

Afin de voir en détail le déroulement du programme, repérer d’éventuelles erreurs et déboguer, nous avons rajouté une variable globale int DEBUG qui est égale à 1 si on lance la commande suivante:
`./dazibao jch.irif.fr 1212 debug.`

---

3. Exemple d’exécution

Voici quelques extraits d’une exécution de notre programme:

Commande utilisé : `./dazibao jch.irif.fr 1212 debug`

Lorsqu’on lance en mode debug on peut voir les différentes étapes du programme en plus de l’affichage des TLVs envoyés et reçus.
Au lancement du programme, on récupère les adresses du premier pair permanent et on les ajoute dans la table de voisins.

```
[DEBUG] Initialisation des données...
IP du premier voisin permanent ---> ::ffff:81.194.27.155
************************
[DEBUG] Voisin permanent ajouté. IP : ::ffff:81.194.27.155
IP du premier voisin permanent ---> 2001:660:3301:9200::51c2:1b9b
************************
[DEBUG] Voisin permanent ajouté. IP : 2001:660:3301:9200::51c2:1b9b
NOMBRE DE VOISINS  2
```

```
[DEBUG] ---------- AFFICHAGE DE LA TABLE DES VOISINS ----------
--------- NEIGHBOUR ---------
Is this neighbour permanent ? : 1
Time of last reception :
2020-05-14 23:05:06.
- Socket informations
sa_family is : iPv6
port is 1212
IP address is : ::ffff:81.194.27.155

--------- NEIGHBOUR ---------
Is this neighbour permanent ? : 1
Time of last reception :
2020-05-14 23:05:06.
- Socket informations
sa_family is : iPv6
port is 1212
IP address is : 2001:660:3301:9200::51c2:1b9b
```

Actions effectuées toutes les 20 secondes:

- On vérifie si un voisin transitoire n'a pas envoyé de paquet depuis 70 secondes
- On envoie un TLV Network Hash à tous les voisins
- On envoie un TLV neighbour Request à un voisin tiré au hasard

---

```
[DEBUG] --- 20 secondes se sont écoulées ! ---
[DEBUG] sweep_neighbour_table, il reste 4 voisins.
[DEBUG] Nombre de voisins supprimés: 0
TLV NETWORK HASH envoyé à tous les voisins!
TLV Neighbour Request Envoyé à l'adresse IP :
2001:0:53aa:64c:2c4d:bb59:a457:3c7f
```

Si on exécute le programme en affichant en détail, on peut voir que dans le cas où on reçoit un TLV Neighbour contenant une adresse, on envoie un TLV Network Hash au pair ayant cette adresse, ensuite lorsqu'on reçoit un paquet correct de ce pair on l'ajoute à la table de voisin.

```
------- Message Reçu ! --------
- Le message provient de l'IP : 2001:660:3301:9200::51c2:1b9b
Magic : 95
Version : 1
```

```
Body_Length : 40
Nb tlv in dtg : 3
     --  TLV NET. STATE REQUEST  --
TLV type : 5
TLV length : 0
     -------  TLV NEIGHBOUR  ------
TLV type : 3
TLV length : 18
IP address is : ::ffff:88.124.234.159
Port is : 49153
     -----  TLV NETWORK HASH  -----
TLV type : 4
TLV length : 16
Network hash is : 6494ac5bc8577249e21c134241a5afd3

--------------------------------------------------------------------------------

**************************************************
Envoi d'un Network hash à l'adresse IP : ::ffff:88.124.234.159
**************************************************

--------------------------------------------------------------------------------
------- Message Reçu ! --------
- Le message provient de l'IP : ::ffff:88.124.234.159
Magic : 95
Version : 1
Body_Length : 2
Nb tlv in dtg : 1
     --  TLV NET. STATE REQUEST  --
TLV type : 5
TLV length : 0
**************************************************
**************************************************
Envoi de 1 Node Hash :

--- DATAGRAMME ENVOYE A
IP : ::ffff:88.124.234.159
Port : 49153
--------------------------
**************************************************
[DEBUG] Voisin transitoire ajouté. IP : ::ffff:88.124.234.159
```

Lorsqu'on reçoit un TLV Network State Request, dans le cas où les deux hashs ne sont pas identiques on envoie plusieurs node hash répartis dans plusieurs paquets car la taille maximale est 1024 octets.

```
------- Message Reçu ! --------
- Le message provient de l'IP :
2a01:e35:2f86:d0d0:d1ed:9f91:a104:e713
Magic : 95
Version : 1
Body_Length : 2
Nb tlv in dtg : 1
Network State Request
**************************************************
**************************************************
Envoi de 35 Node Hash :

--- DATAGRAMME ENVOYE A
IP : 2a01:e35:2f86:d0d0:d1ed:9f91:a104:e713
Port : 1354
--------------------------
**************************************************
**************************************************
Envoi de 35 Node Hash :

--- DATAGRAMME ENVOYE A
IP : 2a01:e35:2f86:d0d0:d1ed:9f91:a104:e713
Port : 1354
--------------------------
**************************************************
**************************************************
Envoi de 35 Node Hash :

--- DATAGRAMME ENVOYE A
IP : 2a01:e35:2f86:d0d0:d1ed:9f91:a104:e713
Port : 1354
```