PROJET DE RÉSEAU : Dazibao par inondation non-fiable

KHALFAT Célina MIMOUNI Ahmed

Bonjour, bienvenue dans notre projet !

Table des matières

I – Sujet de base

a) Les structures de données

Premièrement, la **table des voisins** : c'est une structure composée uniquement d'un tableau de neighbours et du nombre de voisins contenus dans la table. Un *neighbour* lui est constitué d'un pointeur vers une *sockaddr_in6,* d'un entier (1 pour permanent, 0 pour intermittent), et d'un *time_t,* la date du dernier paquet reçu émis par ce voisin. On n'utilise que des *sockaddr_in6*, les adresses IPv4 étant mappées en IPv6.

La méthode *getneighbour(...)* renvoie l'indice dans la table du voisin de port et IP égaux à ceux de la *sockaddr_in6* donnée en argument, et *addneighbour(...)* créée un nouveau *neighbour* et l'ajoute à notre table (si possible) à partir de la *sockaddr_in6* donnée en argument.

Comme indiqué dans l'énoncé, on envoie toutes les 20 secondes une *neighbour_request* à un voisin choisi aléatoirement, on envoie ensuite un *network hash* à notre potentiel nouveau voisin, qu'on ajoute à notre table s'il nous renvoie un paquet correct.

Durant nos exécutions on a très rarement dépassé les 5 voisins, mais nous avons préféré nous cantonner à ce que le protocole indiquait en matière d'envoi de *neighbour request* et d'ajout à la table des voisins.

La seconde table importante est celle des **données publiées** : c'est une structure, qui a pour champs sa longueur effective, c'est à dire le nombre de data effectivement présents dans la table, sa capacité (en effet, la table est dynamique, on pourra augmenter sa capacité si nécessaire : on doublera sa taille si une insertion n'est plus possible), le *network hash*, et un tableau d'objets *data*.

Un **objet data** a cinq champs : le *node id* sur 8 octets, le numéro de séquence, un pointeur vers la donnée, un champs *len* indiquant la taille de la donnée, et son *hash*, sur 16 octets.

Le tableau est trié : on y insère et cherche une donnée par dichotomie grâce à la méthode *getData(...)*, qui renvoie l'indice de la donnée si elle est présente, ou l'indice à laquelle elle devrait être insérée. On appelle alors *insertData(...)*, qui s'occupe d'agrandir la table si nécessaire, de faire les bons décalages mémoire, et d'entretenir la variable globale *ourdata_index* qui conserve la position dans la table de notre donnée. Cela permet de modifier notre donnée/numéro de séquence rapidement et efficacement.

On utilise une **socket polymorphe non bloquante**, initialisée par la méthode *creation_socket(...),* qui est stockée dans la variable globale *int s* . Cela permettait principalement une économie d'argument dans les méthodes : on utilise tout au long du programme cette seule et même socket.

La dernière structure utilisée est celle des tlv. Un **objet tlv** contient tous les champs nécessaires à la construction d'un tlv : *type*, *length*, *pad*, *seqno*, *node_id*, *hash*, *ip, port*, *data_len*, *data*. Il permet la standardisation de l'envoi d'un tlv, et donc d'un paquet.

b) Structure du programme pour l'envoi

Les paquets sont construits grâce à 2 fonctions :

- *package_builder(...)* : cette fonction prend comme paramètre un tableau d*'objets tlv*, et le nombre d'éléments de ce tableau. Elle récupère de ce tlv son type, qui indique quel élément du tableau de pointeurs de fonctions tlv_builder doit être appelé. C'est un tableau de fonctions qui prennent un tlv et un buffer *package*, et qui remplissent le buffer du tlv décrit par l'objet en paramètre. Ainsi, pas de long switch pas très lisible, on voit facilement comment est construit chacun des tlv.
- *package_sender(...)*, qui à un sockaddr_in6 envoie le paquet pointé par *package.* La longueur du paquet est retrouvée dans le paquet.

c) Structure du programme pour la réception

On a choisi d'utiliser l'appel système select pour recevoir nos paquets sans bloquer notre programme. On écoute deux descripteurs de fichiers : notre socket *s*, mais aussi l'*entrée standard.*

Pour ce qui est de l'entrée standard, l'utilisateur peut entrer un message à tout moment, qui deviendra sa nouvelle donnée. S'il entre simplement *q\n,* le programme s'arrête et affiche l'état actuel de la table des données, le nombre de données, notre numéro de séquence, notre dernier message publié, et d'autres valeurs.

Pour ce qui est de la socket, on utilise *recvfrom(...)* pour récupérer le message et l'envoyeur. On vérifie ensuite grâce à la méthode *paquet_ok(...)* que le paquet est conforme au protocole. S'il ne l'est pas, on envoie un warning à l'envoyeur et on ignore le paquet. Sinon, on suit le protocole comme indiqué dans l'énoncé. On en vient à re-parcourir le paquet, mais cette fois en lisant les TLV : on utilise un autre tableau de pointeurs de fonctions : *tlv_processor(...).* Le principe sera de créer **une table de tlv**, qui contiendra tous les *tlv réponses* aux *tlv reçus*. Une fonction de type *tlv_processor(...)* prend comme argument un pointeur vers le début du *tlv*, la table de *tlv*, et le nombre d'élément de la table. Elle renvoie le nombre de *tlv* qu'elle a ajouté à la table, et procède à tout ce qui doit être fait en fonction du type de *tlv* reçu. On avance ensuite de la longueur du *tlv* + 2 jusqu'au début du prochain *tlv*. La prochaine étape est de construire et d'envoyer un *tlv* à partir du tableau de *tlv* qui vient d'être construit, grâce aux méthodes précédemment évoquées : *package_builder(...)* et *package_sender(...).*

d) Compilation et exécution

Pour exécuter notre projet, rien de plus simple ! Dans le dossier de notre projet, lancez ***make*** pour compiler, puis ***./dazibao*** pour lancer le projet.

II – Extensions implémentées

a) Les warning

La méthode *paquet_ok(...)* nous fournit une chaîne de caractère indiquant quel est le soucis avec le paquet reçu : "Magic incorrect", "Version incorrecte", "Taille de paquet incorrecte", ou "Taille de TLV incorrecte". À la réception d'un paquet déclaré incorrect par *paquet_ok(...)*, nous répondons à l'envoyeur un warning contenant la valeur de retour de *paquet_ok(...)*.

b) Vérification de la cohérence des Node State

À la réception d'*un node state*, nous avons tout le nécessaire pour recalculer le *hash du noeud* concerné. Nous le comparons ensuite avec le *hash* indiqué par l'envoyeur : s'il est différent, nous ne prenons pas en compte le *tlv* et notifions d'un warning son auteur.

c) Agrégation des paquets

Après avoir construit notre *table d'objets tlv* prêts à l'envoi, nous devons décider par combien nous les enverrons. Nous avions initialement majoré la taille d'un paquet pour minorer le nombre de paquets entrant dans un paquet dans le pire des cas, mais n'avoir que 4 *node hash* par paquet n'était pas une bonne solution au problème. Les faire entrer dans l'ordre de rencontre jusqu'à ne plus pouvoir en faire entrer ne nous suffisait pas non plus, et reparcourir la table à chaque fois à la recherche de paquet assez petit non plus. Nous avons décidé de trier notre table par ordre croissant de poids : dès qu'un *tlv* ne peut pas entrer, pas la peine de continuer, les suivants non plus. Ce n'est pas la meilleure façon de résoudre le problème, mais une suffisamment bonne pour nous.

d) Calcul des hashes

Après avoir convergé, nous pouvions conserver nos *hash* un petit moment avoir de devoir les changer, parfois le temps de plusieurs comparaisons. Les recalculer à chaque comparaison ou à chaque modification de la table ne nous semblait pas être une bonne méthode. Nous avons donc utilisé deux variables, *updated_our_data_hash(...)*, et *updated_our_network_hash(...)*, qui indiquent si nos *network hash* et *our data hash* sont à jours avec les données que nous avons. Nous ne recalculons les *hash* donc que si nécessaire : si ces variables valent 1 et que nous devons comparer les *hash*, rien ne sera recalculé. Chaque changement de donnée les remet à 0, et chaque calcul de *hash* les remet à 1.

e) Contrôle de flots

Nous avons décidé d'une vitesse **plafond de 128Ko/sec**, arbitrairement, et faisons s'arrêter les envois le temps nécessaire à ce que la vitesse retombe à cette valeur dès que plus de 128 000 octets ont été envoyés en moins d'une seconde. En pratique, on attend une seconde à chaque fois, les 128Ko partant en moins d'une microseconde.

III - Nos soucis

a) Fuites de mémoire

Pendant longtemps, nous codions dans l'ignorance, avec l'impression de faire du bon boulot. Et puis vint *splint*. Puis *valgrind*. Deux bonne journées ont été allouées à passer de quelques millions d'octets de perdus à moins de mille, en passant par de multiples erreurs de segmentation dues à des *free* hasardeux. Aujourd'hui nous ne rencontrons plus d'erreur de segmentation et n'avons plus que très peu de fuite de mémoire.

b) Le network hash

Notre unique dernier gros problème est celui du *network hash*. Nous ne semblons pas arriver à le calculer correctement : même après avoir récupéré toutes les données depuis nos pairs, notre *network hash* n’est que très rarement le même que ceux que nous recevons. Nous n’avons reçu aucun warning d’inconsistant hash : nos *node hash* sont corrects. Nous avons aussi vérifié, notre table des data est bien triée dans l’ordre croissant des *node id*, par comparaison d’entiers en small endian. Nous avons aussi enlevé les éléments relatifs au point II-d) Calcul des hashes pour essayer de déboguer le *network hash.* Nous espérons au moins d’ici la soutenance trouver la cause de notre ultime soucis.