大字报

> Un grand bond en avant pour le pair à pair du XXIème siècle

Étudiants préparant le projet de réseau, 2020 *By 人民画报 - 《人民畫報》1967年, Public Domain, Link*

Abstract

Dazi bao est l'implémentation d'un protocole de commérage, qui permet de publier un message, ce dernier est envoyé de proche en proche à chaque pair du réseau.

L'utilisation se fait depuis un terminal, en ligne de commande.

Structure et déroulement

Le programme est structuré autour du fichier `node.c`, dans lequel se trouve la

fonction `main()`. Cette fonction va se charger d'appeler les fonctions d'initialisation, et de lancer le protocole. La construction des TLV, la production d'un hash et les fonctions de gestion et d'affichage d'erreurs sont dans leurs propres fichiers, car ce sont des points qui peuvent être amenés à changer lors de l'implémentation d'extensions.

Le noeud n'a pas de mémoire, et va donc être initialisé à partir de 0 à chaque lancement. Entre le lancement, et le moment où l'on connaît l'ensemble des messages publiés (avec le site web comme référence, bien que ça ne soit pas forcément valide), le temps de convergance est de l'ordre de quelques secondes.

Structure

Le programme se découpe en 2 parties ; l'initialisation, puis la récupération et publication des messages.

Initialisation

Lorsque l'on exécute le programme, on donne une URL ainsi qu'un numéro de port. Cette URL va être décodée par `getaddrinfo`, qui va nous donner >=1 adresses IP. Pour chaque adresse IP, nous créons un pair, que l'on va marquer comme permanent. Si plusieurs adresses IP sont données, mais qu'elles correspondent à un seul pair, nous n'en utilisons qu'une seule, pour évider les congestions et autres erreurs. On y associe le numéro de port donné lors du lancement. Nous créons aussi le socket que l'on utilisera ensuite. Nous le fixons sur le port 1212.

Par soucis de simplicité, et dans l'espoir que IPv4 disparaisse rapidement des internet, nous traitons tout les pairs comme étant en IPv6. Les adresses IPv4 sont *IPv6 mapped*. Une fois le programme lancé, on ne peux plus ajouter de pair permanent, ni les supprimer. On pourrait étendre le programme en permettant d'ajouter les pairs de façon dynamique, en les ajoutant ou supprimant. On pourrait aussi choisir de "bloquer" un pair, mais cela reste impossible en l'état, voir la note 1.

Nous initialisations aussi la liste des messages, en créant un premier message, vide. Il est vide, pour plusieurs raisons : d'une part, lorsque l'on lance le

programme, on peux ne pas avoir envie de publier quoique ce soit, et de juste recevoir les messages. On peux aussi avoir envie d'effacer les messages publiés précédemment. Lors de la publication de notre message, il remplacera les messages précédents connu des pairs par une phrase vide, ce qui effacera le message. À noter que, vu qu'il n'existe pas, dans le protocole de base, d'authentification, ça peux aussi être un moyen d'effacer tout les messages du réseau.[1]

La liste de pairs et la liste de messages sont des listes chainées simples. Elles sont suffisantes, car permettent l'ajout et la suppression. Lors de l'envoi de messages, nous devons parcourir toute la liste de pairs, et lors de la réception ou l'inondation de messages, nous devons aussi passer en revue tous les messages.

Une extension possible serait de stocker le hash de chaque message, afin d'avoir une hash map. On pourrait donc vérifier uniquement le hash, sans avoir besoin de le recalculer. Cela réduirait aussi fortement le temps de mise à jour et d'accès aux messages.

Déroulement

Le noeud va ensuite initialiser une structure `pollfd`, une variable de délai, ainsi que deux tableaux de char. Ces derniers font 1024 bytes, ce qui est recommandé dans le projet. C'est suffisant, car un paquet ne peux pas faire plus de 1024 bytes, et nous traitons les paquets les uns à la suite des autres. La structure `pollfd` va servir lors de l'appel à la fonction `poll`, qui se charge de surveiller deux descripteurs de fichiers : `stdin`, et le socket définit lors de l'initialisation. Quand des données sont écrites dans ces fichiers, nous effectuons une lecture.

La lecture de stdin permet de récupérer à la fois des commandes pour la gestion du noeud, ainsi que le message à publier. La lecture du socket nous permet de récupérer les paquets entrants.

La variable de délai est utilisée pour mettre à jour la liste de pairs, demander plus de pairs le cas échéant, et demander l'état du réseau aux autres pairs. Nous l'initialisons à une valeur inférieure à 20 secondes, car lors du lancement, nous ne connaissons rien du réseau. Pas la peine d'attendre. Une fois la

première exécution passée, nous mettons un délais de vingt à trente secondes. C'est suffisant, car la publication de nouveaux messages sur le réseau est assez lente. Nous pourrions faire évoluer ce délais en fonction du rythme de publication de nouveaux messages.

Ajouts de nouveaux messages

L'ajout de nouveaux messages est fait uniquement si on écrit un message non-vide. Dans le cas contraire, on affiche les messages connus.

On va ensuite appeler la fonction `add_message(char * message, int message_len)` . Elle se charge de vérifier qu'on n'ajoute pas un message vide, puis va chercher le message publié par notre propre identifiant. On met à jour le numéro de séquence, et l'on copie le nouveau message dans la liste. Afin d'éviter des erreurs, on libère l'ancienne allocation, et on alloue une nouvelle mémoire de la taille du nouveau message.

Ce message sera envoyé aux pairs en faisant la demande lors de l'inondation.

Réception des paquets et traitement

Lors de la réception d'un message, on effectue plusieurs vérifications. La première consiste à vérifier l'entête du paquet. En particulier, il faudra vérifier que le champ body_length du paquet est bien conforme aux spécifications données et correspond bien au nombre d'octets reçus par recvmsg. Une fois cette entête validée, on ajoute le pair à la liste des pairs connus.

On passe ensuite à la validation de chaque TLV. On commence par créer un paquet vide, qui va nous servir lors du renvoi éventuel de paquets vers les pairs. Ensuite, l'un à la suite des autres, nous validons les TLVs. Il faudra vérifier que leur longueur n'est pas différente de ce que l'on attend selon leur type, mais aussi il faudra vérifier que la longeur annoncée ne dépasse pas les bornes du paquet à la position courrante.

La validation du TLV se fait selon plusieurs critères : est-ce que la taille annoncée correspond bien à la taille que l'on constate, est-ce que le type de TLV existe bien, ou encore est-ce que le TLV doit être traité (dans le cas du padding par exemple).

Une fois qu'un TLV est valide, nous le traitons. En fonctions des cas, nous allons recalculer les hash, ou mettre à jour des données. Si nous avons besoin de renvoyer un paquet, nous allons soit envoyer un seul TLV dans un paquet dans certains cas spécifiques, soit agréger des TLVs dans un paquet. Celui-ci servira de buffer pour la fonction d'envoi, qui sera appelée lorsque le paquet ne peux plus accueillir de nouveaux TLV, suite à quoi le paquet sera réinitialisé afin de pouvoir le réutiliser pour des futurs envois.

Après avoir traité tous les TLVs reçus, nous enverrons le paquet courant, qui peut encore contenir des TLVs non envoyés.

Gestion et génération des hashes.

Nous utilisons la librairie OpenSSL pour générer notre hash SHA256. Dans le cas où nous ne générons que le hash d'un seul noeud, on concatenne les données id, seqno et data (en faisant attention à convertir id et seqno en big endian) dans un même buffer et on le passe à la fonction de hashage.

Dans le cas d'un network hash, on concatènera dans un grand buffer tous les hashes de chaque donnée connue (qui sont ajoutées peu à peu dans une liste chainée de façon à ce qu'elle soit en ordre croissant par rapport à l'id) puis nous passerons ce buffer à la fonction de hashage.

On n'utilisera que les 16 premiers bytes de ces hashes.

Choix d'implémentation

TLV

Les TLVs sont représentés par un type `union TLV`, qui contient des pointeurs vers le `struct` TLV en lui-même, différent pour chaque type de TLV. Afin de garder une facilité d'accès aux TLVs, ils sont tous définis avec comme premier champs, sont type, qui est de même taille pour tout TLV.

poll()

L'utilisation de `poll()` nous permet de réagir à des évènements, et donc de ne pas avoir à attendre inutilement quand rien ne se passe. `poll()` va attendre

10ms sur chaque descripteur de fichier, en bloquant le reste du programme. Si nous recevons un message alors que nous n'observons pas le socket, rien de grave ne se passe ; on attendra un autre message. De plus, il y a déjà un buffer interne au système.

Envoie des paquets avec sendmsg

L'utilisation de `sendmsg` ainsi que d'un buffer vectorisé n'a pas de réel avantage par rapport à un buffer classique. Mais, il pourrait permettre de stocker des messages à envoyer en l'attente d'une connexion. On pourrait ainsi envoyer plusieurs paquets d'un coup, au lieu de les envoyer les uns à la suite des autres.

Affichage et gestion des erreurs

Notre programme contient une variable `DEBUG_LEVEL` définie dans `debug.h`, qui permet de compiler avec plusieurs niveaux de verbosité. Lorsqu'elle est à 0, seul des messages d'information et d'erreur sont affichés, tel que l'ajout d'un message, la réception d'un message, ou l'affichage des messages connus.

Puis, entre 1 et 9, des message de débug avec plus ou moins de détails sont affichés, ce qui permet d'avoir une idée de la "vie" du pair en temps réel. À partir de 9, l'éxécution se fait étape par étape, en appuyant sur entrée à chaque étape.

Une attention particulière est porté sur la gestion des erreurs. En effet, notre pair va essayer au maximum de continuer à vivre malgré les bugs possible. Vu qu'il est possible de recevoir tout type de message, nous préférons afficher un message d'erreur ou de débug plutôt que d'arrêter l'exécution. À noter que le fait de ne pas avoir de pair au début n'est pas une erreur fatale, car il se peut qu'un autre pair communique avec nous, sans que nous ne le connaissions au préalable.

Extensions

Nous avons implémenter l'agrégation de TLV dans un paquet.

Nous vérifions la cohérence des *Node State* ; si on reçoit un node state, on vérifie que son hash est cohérent.

Si notre pair as plusieurs adresses, nous communiquons sur toutes les adresses qu'il possède.

Tests

Nous avons réalisé de nombreux test, y compris avec d'autres pairs, écrit par des camarades. Cela nous as permis de renforcer la robustesse de notre programme.

Nous nous ajoutions mutuellement comme pair par défault, ou alors nous formions une "chaîne", où l'on ajoute son voisin, et il ajoute un autre voisin.

Nous avons aussi essayé de faire du renvoi de paquet depuis WireShark, afin de tester la réaction de notre pair à la réception de plusieurs paquets identiques, ou de messages tronqués, corrompu, ou malveillants...

Toutefois, la licence AGPL s'applique toujours ; aucune garantie n'est possible...

[1] Pour cela, il suffit de publier un message vide, tout en se donnant comme numéro d'identification ceux que l'on connait, venant d'autres pairs.