Julien Duprat

Rendu de projet
Réseau

https://github.com/Draal8/servDNS

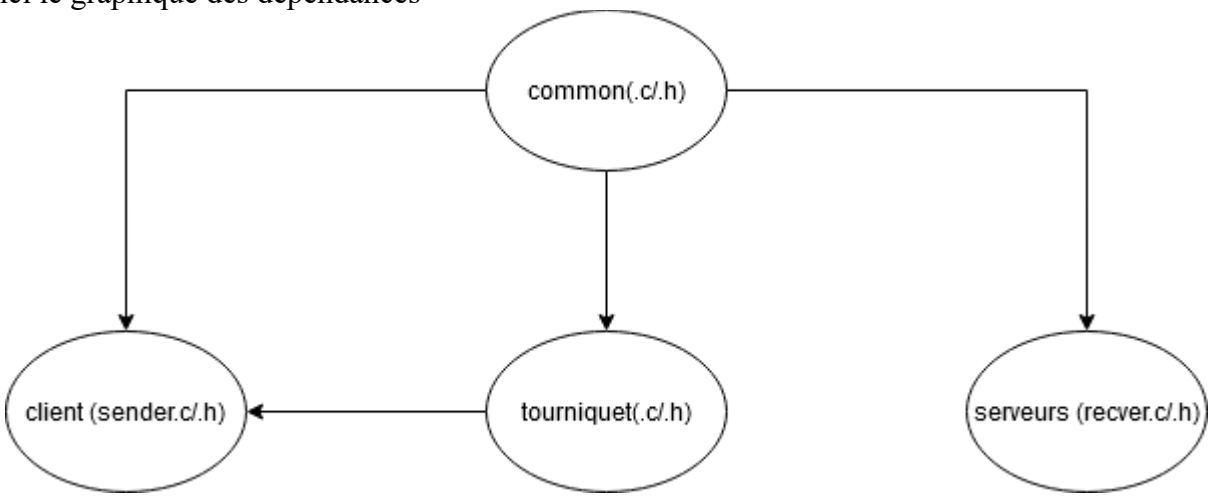
Table des matières :

I – Présentation globale

Tout d'abord n'oubliez pas de jeter un coup d'œil au README.md qui vous éclairera sur certains points.

Pour l'implémentation j'ai tenté de suivre le plus possible le sujet.
Le nombre de serveurs ainsi que le nombre de niveau maximums pour un nom de domaine n'étant pas précisés j'ai choisi de considérer qu'ils pouvaient être infinis (j'ai une limite fixe de taille des strings qui est de 4096 octets donc c'est ma limite en vérité).
Il ma semblé évident qu'il me faudrait un programme serveur général capable de faire la tache d'un serveur de n'importe quel niveau, sinon on aurait encore a se soucier du nombre de serveurs a faire ou a poser une limite sur le niveau des noms de domaine.
Quand je parle de niveau chaque '.' indique un niveau supplémentaire ex : "unistra.fr" est de niveau 2 mais "moodle.unistra.fr" est de niveau 3.

Voici le graphique des dépendances



Pour lancer le tout il faut :

- lancer un client avec :
  - argv[1] le chemin vers un fichier texte qui contient adresse et port de serveur(s) racine.
  - argv[2] le chemin vers un fichier texte qui contient nom de domaine a resoudre par ligne.
  - (argv[3]) un timeout optionnel (par défaut 18ms)
- lancer un ou plusieurs serveurs :
  - argv[1] un numéro de port libre et cohérent
  - argv[2] le chemin vers un fichier texte qui contient nom de domaine, adresse et port de serveur(s) de niveau supérieur.
  - (argv[3]) un délai optionnel (aléatoire entre 1 et 20ms par défaut)

Dans l'ordre serveurs puis client.
Se référer au README pour plus d'informations.

II – Implémentation serveur

Le serveur supporte IPV4 comme IPV6.
J'ai implémenté cette fonctionnalité en traduisant l'IPV4 en IPV6 pour la socket de manière automatique grâce a

```
CHECK(setsockopt(sockfd, IPPROTO_IPV6, IPV6_V6ONLY, &(int){0}, sizeof(int)));   //On désactive ipv6 only
```

Cette façon de faire présente plusieurs avantages. Le code est plus lisible, plus concis, on a aucune condition sur la nature de l'IP a gérer (aussi bien pour des fonctions réseaux que pour la gestion des strings dans le serveur).

Cette manière m'a aussi permis de n'utiliser qu'une seule socket. Comme le sujet ne parle d'utiliser qu'un seul client et de ne pas faire de parallélisme je peux me le permettre. N'utiliser qu'une seule socket améliore aussi la concision.

La boucle while(1) est une solution au problème qui est de savoir quand arrêter le serveur. Essentiellement le serveur attends un signal pour tuer le processus sans que j'utilise l'API posix de gestion des signaux.

III – Implémentation client

Le client supporte IPV4 comme IPV6.
Le client sur ce point-la fonctionne de la même manière que le serveur.

Toute la partie tourniquet se trouve sur le fichier tourniquet.c et le reste sur sender.c.
Le tourniquet comprends les tests de ping des serveurs, la gestion des timeouts, il fait tourner les serveurs en fonction de leur vitesse et de si on a déjà tente de les contacter.
Sender.c s'occupe de créer/envoyer/recevoir/déchiffrer les messages et de feed ces données au tourniquet pour choisir un nouveau serveur. Il lis aussi les noms de domaines a résoudre un par un.

Sender.c contient 2 fonctions principales : sender() et suite().
La première assure le rôle du client sur le serveur racine, et la deuxième est récursive et assure le comportement général hors serveur racine.

La partie timeout est gérée ainsi dans la fonction sender() :

```
CHECK(setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &time, sizeof(struct timeval)));
```

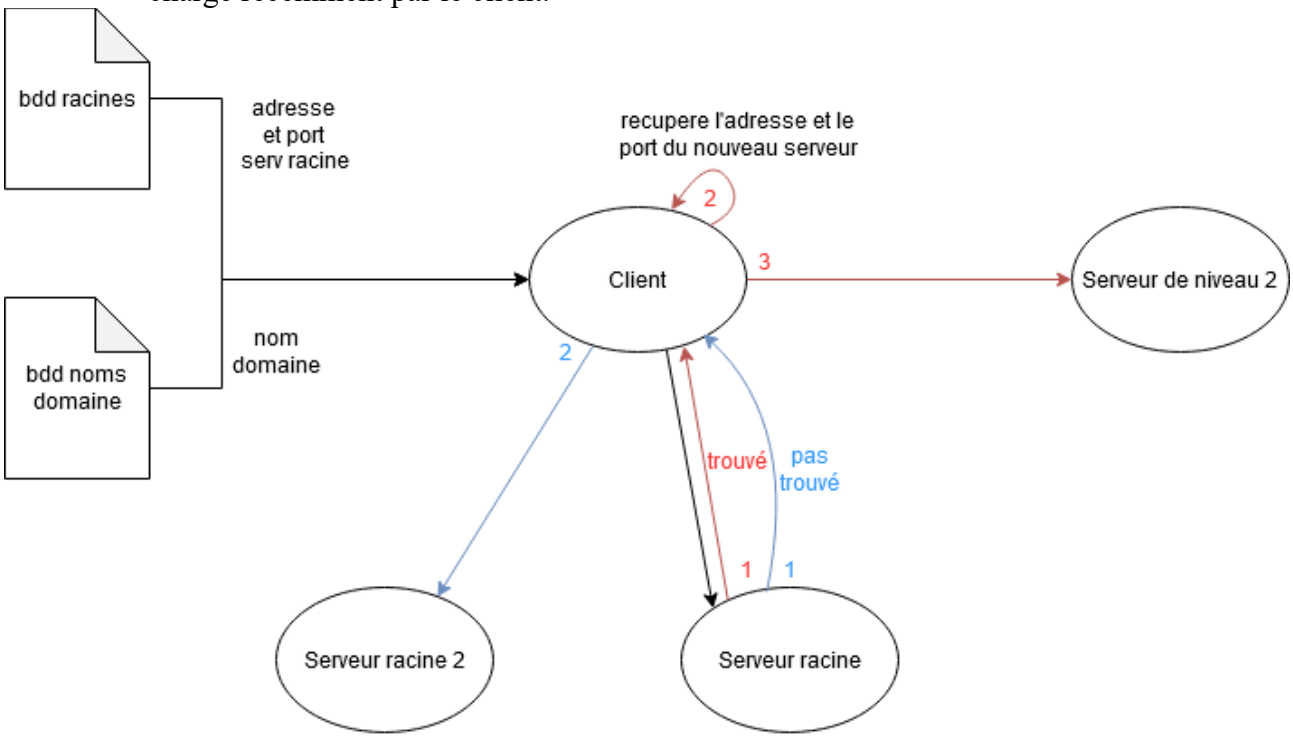
```
do {
CHECK(sendto(sockfd, message, strlen(message)+1, 0, (struct sockaddr *) &dest, addrlen));

recvfrom(sockfd, buff, STR_SIZE, 0, (struct sockaddr *) &retour, &addrlen);
i++;
} while ((errno == EAGAIN || errno == EWOULDBLOCK) && i < 3);

if (errno == EAGAIN || errno == EWOULDBLOCK) {
    errno = 0;
    printf("timeout\n");
    register_time(buff, adrs_port, 1);
    free(message);
    return -1;
}
```

Et de manière très similaire dans suite().
J'utilise EAGAIN et EWOULDBLOCK pour une question de portabilité car certains systèmes renvoient l'un ou l'autre après un timeout sur recvfrom().

IV – Implémentation tourniquet

Les adresses introuvables sont détectées si le serveur renvoi "msg_origine |-1|".
Le client appelle alors le tourniquet qui selon le cas :

- trouve le prochain serveur candidat dont on ne connaît pas encore la latence
- trouve le prochain serveur candidat qui a eut (relativement a sa latence) le moins de charge récemment par le client.

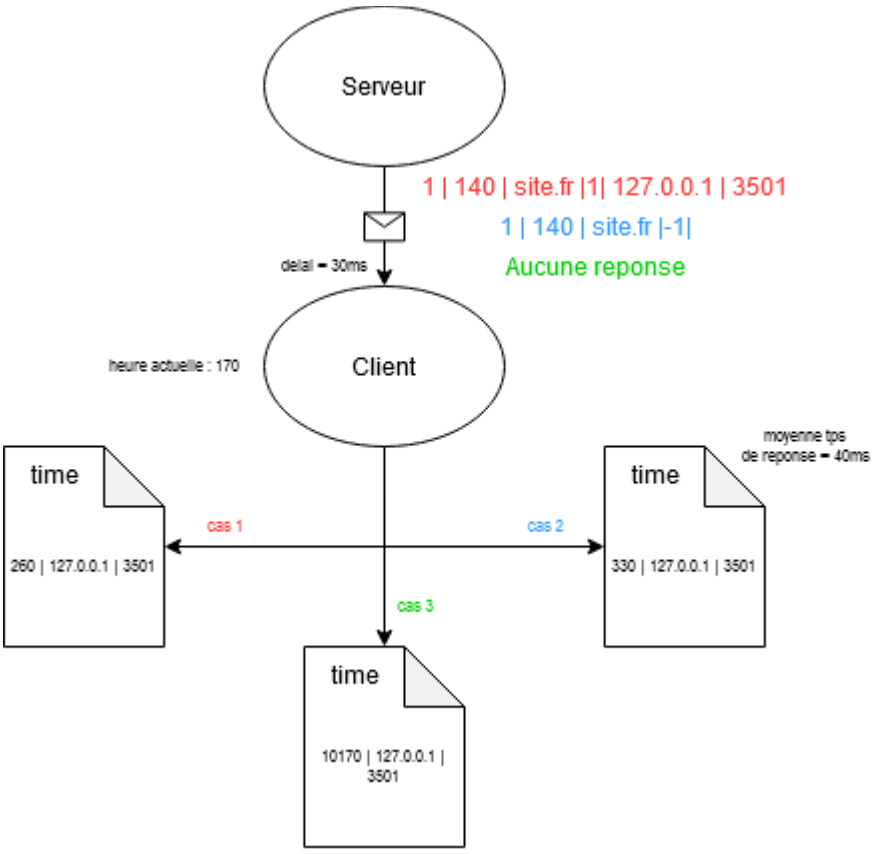


Le tourniquet utilise un système de ELO.
Si le serveur n'a jamais été appelé alors il ne possède pas d'entrée dans le fichier time que le logiciel créer. Sinon il possède une entrée et on peut donc comparer son ELO a celui des autres serveurs candidats (cad ceux qui peuvent avoir le nom de domaine que le client chercher selon le serveur de niveau inférieur ou la liste des serveurs racines fournie).
Le ELO fonctionne ainsi :

- si le serveur trouve le nom de domaine alors le client renseigne dans time l'adresse, le port et le délai de réponse au carré ajouté au temps du jour en ms
- si le serveur ne trouve pas le nom de domaine et renvoi |-1| alors le client renseigne dans time l'adresse, le port et un délai moyen de réponse au carré ajouté au temps du jour en ms
- si le serveur ne répond pas alors le client renseigne dans time l'adresse, le port et un délai 10 fois supérieur au délai moyen, le tout au carré et ajouté au temps du jour en ms

Le tourniquet choisi ensuite le premier serveur qui a le ELO le plus bas possible. Ainsi tous les serveurs tournent et le délai de réponse au carré permet de le prendre plus en compte dans le choix du serveur.
L'avantage par rapport a un stockage du ping simple c'est que avec un algorithme très simple et rapide on arrive a déterminer avant le prochain passage dans time qui devra passer en premier. On a aussi +/- l'information, si le serveur a timeout ou si il n'a pas trouvé l'adresse, de gardé en mémoire.
Le deuxième avantage et pas des moindres, est qu'on ne peux pas surcharger les serveurs ainsi car leur valeur change a chaque passage. Donc même si ils sont les plus rapides ils seront obliges de laisser la main de temps en temps. Ce qui demande a stocker plus de données si on n'écrit que le ping. Le seul désavantage pas encore patché est si le client fait des appels durant le passage au jour

suivant. Alors le système sera totalement biaisé en faveur des serveurs qui ont déjà été appelé après minuit.



Le client sur certains points mériterais d'être retravaillé pour être refactoré, ce que je n'ai pas eut le temps de faire malheureusement. Notamment avoir une fonction suite() et sender() qui font globalement la même chose ainsi que - par conséquent - une fonction tourniquet() et tourniquet_suite() qui font aussi globalement la même chose.

V – Implémentation des tests

Les tests ont été rédigés en bash pour plusieurs raisons :

- Je savais que je pouvais faire de l'intégration continue avec sur github
- Facilite a lancer en commande ou avec un makefile
- J'avais envie de m'améliorer en bash

Je remercie Mr. Montavont pour ses tests bash en programmation système qui m'ont bien aidé.

Batterie.sh est le fichier qui lance tous les tests a la suite et affiche si ils ont fail ou si ils sont passés. Les autres tests se nomment testX.sh (ou X est un nombre).
On peut soit appeler batterie.sh avec les bons arguments ou j'ai mis a disposition la commande 'make test' ou 'make tests' qui fonctionne si l'utilisateur l'appelle depuis le répertoire parent du projet (celui ou se trouve le Makefile).

Chaque test incrémenté un peu plus la difficulté qu'a le codeur a le passer. Ils testent les fonctionnalités de manière séparée une par une. Ces tests dans l'ordre font :

1. test des arguments
2. teste si le programme s'arrête bien pour une adresse introuvable

3. teste si le programme s’arrête bien pour trop de timeouts
4. test serveur unique avec '.fr' en entrée
5. test adresse normale de taille 3 avec un serveur unique pour chaque niveau
6. test IPV6
7. test multiples serveur qui se partagent les adresses de maniere exclusive IPV4
8. test multiples serveur (dont des serveurs racines) qui se partagent les adresses de maniere exclusive IPV4/IPV6 et elles s'entrelacent
9. test nom de domaine tres long

Le tourniquet au final est teste dans plusieurs situations (2 il doit bien s’arrêter, 3 il ne doit pas boucler sur le même serveur trop longtemps, 5 fonctionnement classique presque sans tourniquet, 7 il ne doit pas s’emmêler les pinceaux et accéder aux bons serveurs, 8 il doit utiliser l’algorithme a base d'ELO)