SOMMAIRE

Rapport de programmation réseau

- Dazibao -

"Journal à grandes lettres "

Saibi Mélissa - Fidalgo Alex

14 Mai 2020

Partie 1 - Nos implémentations et extensions

Nos implémentations sur le sujet minimal

Tout d'abord en ce qui concerne le sujet minimal nous avons tout effectué, le côté client conduisant au processus d'innondation, le côté serveur permettant d'analyser les paquets reçus ainsi qu'une interface utilisateur permettant d'avoir accès aux différentes données nécessaires ainsi que de publier une nouvelle donnée.
Nous avons également géré des cas d'erreurs en envoyant des warnings aux adresses concernées.

Extensions implémentées

Ci-dessous les extensions que nous avons implémentées :

→ **Vérification de la cohérence des node state -**

Nous avons vérifier la cohérence d'un node state, consistant à recalculer le hash reçu dans un tlv node state et à ne pas analyser ce tlv en question si notre résultat diffère de celui présent dans le paquet.

→ **Agrégation -**

En ce qui concerne l'agrégation de paquet, nous ne l'avons pas appliquée à l'ensemble de notre code, mais lorsque nous devons envoyer tout nos nodes hash par exemple, nous les agrégeons dans un même paquet sans jamais dépasser la taille de 1024 octets avant de passer au paquet suivant. Dans ce cas nous pouvons agréger jusqu'a 1012 octets dans un même paquet avant de continuer le processus.

→ **Adresses multiples -**

Nous avons tester notre projet sur une machine pouvant communiquer à la fois en IPV6 et en IPV4. Effectivement, nous sommes rendus compte que nous répondions à des adresses v4 par du v4 également, et de façon symétrique également pour du v6. Nous avons donc fini par nous retrouvé dans notre propre table de voisins avec notre adresse IPV4. Notre pair est donc devenu voisin de lui même. En somme ceci n'est pas vraiment un problème nous allons juste communiquer avec nous même et nous renvoyer l'appareil.

→ **Warnings -**

Dans certains cas ; header incorrect, taille incohérente, node hash incorrect pour un noeud ... nous envoyons des warnings à l'émetteur du paquet en question.

Partie 2 - Structure du programme et techniques employées

Nos structures de données

Les threads

Pour commencer, notre programme entier tourne dans trois threads différents. Celui contenant la partie qui gère le serveur, la réception des paquets, parsing des tlv etc.., celui contenant notre côté client étant responsable de l'inondation et le thread principal étant affilié à l'interface utilisateur.
Initialement nous voulions utiliser des appels à fork() afin de créer plusieurs processus, de la même manière que pour les threads mais nous nous sommes vite rendu compte que l'espace mémoire était divisé selon les différents processus et qu'il n'allait donc pas être possible de partager des données selon les différents processus sans faire appel a la méthode mmap permettant de partager la mémoire.
Suite à cela nous nous sommes pencher sur les threads, sachant que les appels à sendto() et à recvfrom() sont thread safe, même lors de l'utilisation d'une même socket.
Nous avons donc créer un thread pour notre serveur, pour notre client et laisser tourner notre interface utilisateur dans le thread principal, en faisant bien sûr attention a utiliser des mutex afin de bloquer/débloquer ce que nous voulions afin de pouvoir y accéder sans collisions.

Listes de données

Afin de construire notre liste de voisins ainsi que notre liste de données nous avons ici opté pour des listes chaînées (doublement chaînées en fait) contenant nos différentes structures.
Nous avons donc une structure propre à nos voisins contenant les informations qui leurs sont relatives et de la même façon une structure propre à nos données, qui seront donc les noeuds de nos listes chainées.
Nous avons donc toutes les méthodes nécessaires afin d'agir sur nos deux structures de listes chainées : supprimer un noeud, ajouter un noeud par id croissant, détruire une liste etc...
Nous avons également ajouté la date comme pour les voisins, dans notre liste de donnée afin d'avoir la date de dernière modification d'un noeud.

Parsing des paquets reçus

Pour parser les paquets que nous recevons nous n'utilisons rien d'autre qu'un grand switch permettant de passer sur chaque type de TLV.

Nous recevons un paquet : exécution en quelques étapes -

→ **Vérification de l'entête du paquet, warning si incohérent.**

→ **Vérification pour ajout des voisins.**

→ **Déclaration d'un indice permettant une vérification de la taille du paquet reçu.**

→ **Analyse des TLV par un grand switch.**

Construction de nos différents TLV

Afin de construire nos différents TLV nous avons une méthode propre à chacun de ces TLV ainsi qu'une méthode nous permettant de construire le header du TLV qui lui, sera toujours le même.
Toutes ces méthodes , sous les noms de *buil_* suivi du noms du TLV renvoient directement un *unsigned char*$*$.
L'envoie de paquet fonctionne toujours de la même façon, $build\_header()$ afin de construire la tête de notre paquet, on build ensuite le TLV que nous souhaitons ajouter à notre paquet via les méthodes décrites ci-dessus, et nous ajoutons ensuite ce TLV au header construit via une méthode nommée $add\_tlv()$ qui prend elle en paramètre notre header ainsi que le TLV que nous souhaitons y ajouter.
Elle concatène donc ce TLV et remet à jour la taille du paquet situé dans les 3 et 4ème octets du header.

Unicité de la socket

Dans l'intégralité de notre programme nous utilisons la même socket, aussi bien du côté client que du côté serveur.
Nous déclarons cette socket dans notre main, qui est donc une variable globale : elle pourra donc ensuite être utilisée dans tout notre programme lorsqu'elle est nécessaire.

Notre interface utilisateur

Notre interface utilisateur tourne dans un $while(1)$ naturellement et récupère les insertions sur la sortie via $fgets$, les places dans un buffer et analyse ensuite ce buffer afin de pouvoir effectuer les différentes commandes possibles.

Ces différentes commandes sont listées ci-dessous :

❶ **man**

→ Permet d'afficher le manuel de l'interface utilisateur.

❷ **ne**

→ Permet d'afficher la liste des voisins.

❸ **da**

→ Permet d'afficher la liste des données.

❹ **new**

→ Permet de publier une nouvelle donnée.

❺ **clear**

→ Permet de nettoyer l'interface utilisateur via un appel système.

❻ **exit**

→ Permet de sortir de l'ensemble du programme.

Sortie du programme

Lorsque nous appelons la commande *exit* à l'aide de l'interface utilisateur, nous sortons du programme.
A ce moment là nous détruisons/fermons toutes structures ayant été nécessaires au bon fonctionnement de notre programme.
Nous devons alors attendre que le thread serveur, et respectivement que le thread client se termine. Le thread client faisant un appel à $sleep()$, nous devons attendre que ce $sleep()$ se termine avant de pouvoir fermer le programme, d'où le fait que la fermeture du programme prenne quelques secondes.

Déroulement de notre main en quelques étapes

Ci-dessous une explication brève décrivant le déroulement de notre main -

→ **Vérification sur le nombre d'arguments du main.**

→ **Initialisation de notre id unique de par notre adresse mac.**

→ **Création de la socket utilisée dans l'intégralité de notre programme avec les options nécessaires.**

→ **Création d'une adresse aléatoire avec** $in6addr\_any$ **(on laisse le système choisir) avec le port voulu : on bind la socket dessus.**

→ **Initialisation de nos lites de voisins et de données.**

→ **Initialisation de nos mutex, threads, lancement de notre serveur et de notre client.**

→ **Interface utilisateur.**

→ **Destruction/fermeture des structures utilisées lors de notre programme.**

Parie 3 - Séparation des tâches

Saibi Mélissa -

→ Création des structures de données, liste des voisins, liste des données, structures des voisins et des données.

→ Toutes les méthodes relatives à nos listes de données.

→ Constructions des différents tlvs, méthodes *build_* et de la fonction d'agrégation de tlvs.

→ Interface utilisateur.

Fidalgo Alex -

→ Partie serveur - parsing des différents tlv et réponses.

→ Partie client - innondation.

→ Création et utilisation des threads.

→ Méthodes sur la récupération d'adresse, récupération de l'adresse mac.

Parie 4 - Les points intéressants/originaux à remarquer

Récupération de l'adresse mac

Au premier abord, nous avons initialiser notre id via une lecture sur le fichier /dev/urandom, qui nous générait donc un id correct mais aléatoire et différent à chaque lancement du programme.
N'étant pas une bonne solution de par une surcharge du nombre de noeuds, à chaque fois différents, nous avons décider d'initialiser notre id de façon unique et non aléatoire en récupérant les 6 octets de notre adresse mac, complétant ensuite les deux octets restant.
Nous étions tout deux sur deux système différents, OS X et Linux, nous avons donc trouvé deux manières de récupérer l'adresse mac sur nos systèmes respectifs. Nous regardons si (SIOCGIFHWADDR) est défini, si tel est le cas, on défini une méthode relative aux systèmes Linux sinon, dans le cas échéant nous définissons une méthode qui fonctionne sur OS X.

Approche de déboggage pour notre programme

Le déboggage étant une partie essentielle de ce projet, nous avons décider de procéder méthodiquement pour débogguer notre programme.
Nous avons donc défini une macro *DEBUG_PRINT* , prenant en argument le niveau de déboggage souhaité, c'est à dire : *L_ERROR* pour les erreurs, *L_DEBUG* pour des traces intéressantes et *L_TRACE* afin de récupérer toutes les traces.
Elle prend également en paramètre un fichier destination où les traces vont être écrite ainsi qu'un dernier argument représentant la trace à écrire.
L'appel de cette macro est utilisée dans l'intégralité de notre programme, et nous permet donc à chaque lancement d'avoir deux fichiers de déboggage, *debug_innondation.txt* et *debug_serveur.txt* , contenant les informations voulues selon le level de déboggage.

Insertion des élements dans notre liste de donnée

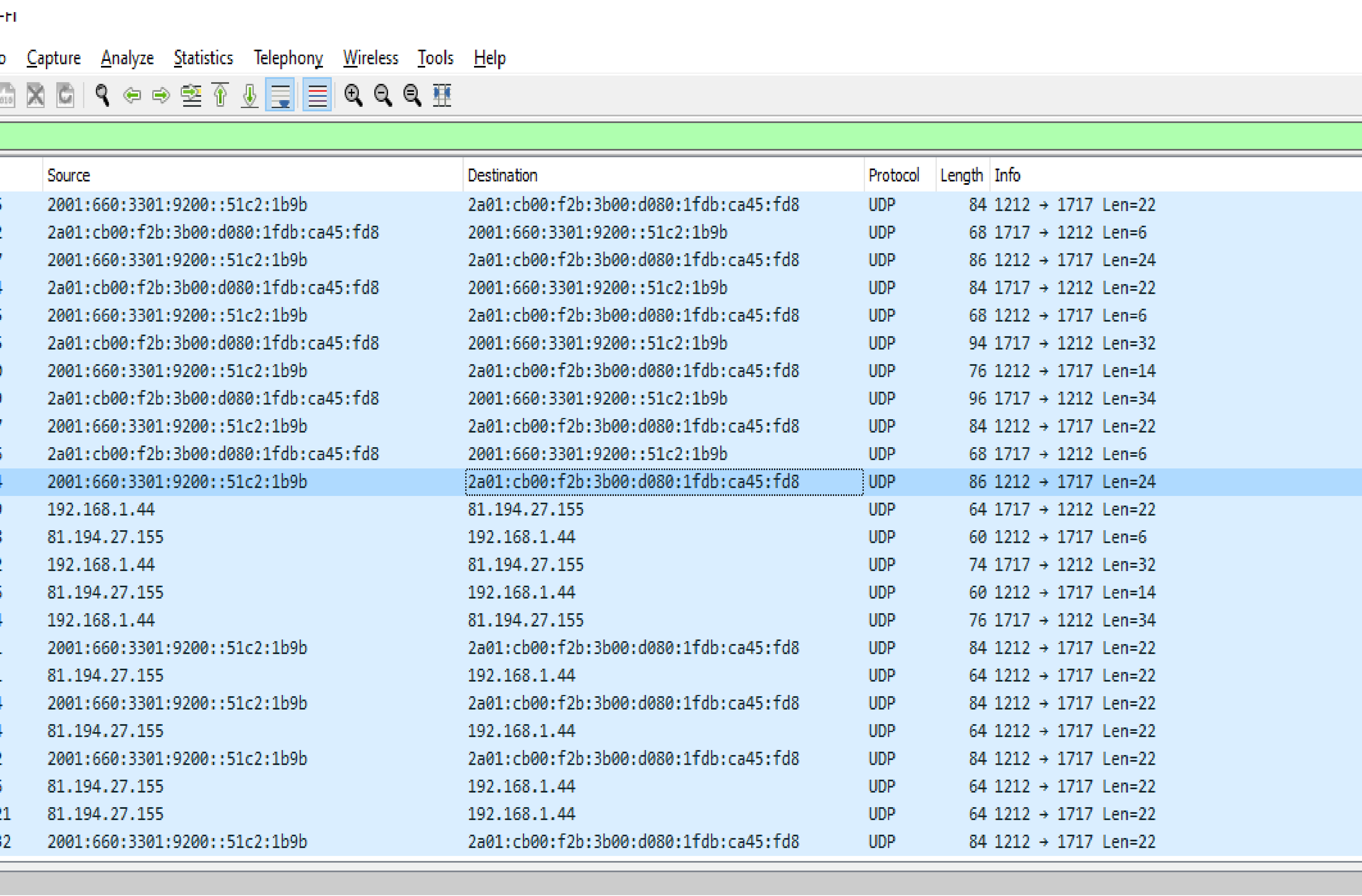
L'insertion des éléments dans notre liste de donnée est relativement importante car le hash doit se calculer par noeud d'id croissant sur les données.
De base, nous avions une méthode permettant de trier notre liste par id et nous l'appelions à chaque fois que nous calculions notre network hash. Cette méthode n'étant que peu optimisée, nous avons décider, à chaque insertion de données, de procéder de façon à ce que les données soient insérées directement triées. De ce fait la liste est triée en permanence et aucunes modifications ne doivent être apportées lorsque nous calculons notre network hash.

Récupération de l'adresse hôte

Pour récupérer l'adresse hôte initiale nous faisons appel à une méthode *get_server_adress*().
Nous passons donc une $struct\ sockaddr\_in6$ en paramètre afin de l'initialiser.
En fait, cette méthode initialise une socket locale et parcours la liste chaînée des adresses grâce à la méthode $getaddrinfo()$, nous essayons d'envoyer un paquet de taille 1 afin de voir si l'appel à $sendto()$ réussi : si c'est le cas on à trouvé notre adresse, le cas échéant, nous continuons de parcourir les adresses jusqu'a réussir l'envoie de ce paquet d'initialisation.

Parie 5 - Quelques exemples d’exécutions avec traces à l’appui

Communication en IPV6 et en IPV4



Source	Destination	Protocol	Length	Info
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	84	1212 → 1717 Len=22
2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	2001:660:3301:9200::51c2:1b9b	UDP	68	1717 → 1212 Len=6
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	86	1212 → 1717 Len=24
2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	2001:660:3301:9200::51c2:1b9b	UDP	84	1717 → 1212 Len=22
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	68	1212 → 1717 Len=6
2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	2001:660:3301:9200::51c2:1b9b	UDP	94	1717 → 1212 Len=32
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	76	1212 → 1717 Len=14
2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	2001:660:3301:9200::51c2:1b9b	UDP	96	1717 → 1212 Len=34
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	84	1212 → 1717 Len=22
2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	2001:660:3301:9200::51c2:1b9b	UDP	68	1717 → 1212 Len=6
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	86	1212 → 1717 Len=24
192.168.1.44	81.194.27.155	UDP	64	1717 → 1212 Len=22
81.194.27.155	192.168.1.44	UDP	60	1212 → 1717 Len=6
192.168.1.44	81.194.27.155	UDP	74	1717 → 1212 Len=32
81.194.27.155	192.168.1.44	UDP	60	1212 → 1717 Len=14
192.168.1.44	81.194.27.155	UDP	76	1717 → 1212 Len=34
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	84	1212 → 1717 Len=22
81.194.27.155	192.168.1.44	UDP	64	1212 → 1717 Len=22
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	84	1212 → 1717 Len=22
81.194.27.155	192.168.1.44	UDP	64	1212 → 1717 Len=22
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	84	1212 → 1717 Len=22
81.194.27.155	192.168.1.44	UDP	64	1212 → 1717 Len=22
81.194.27.155	192.168.1.44	UDP	64	1212 → 1717 Len=22
2001:660:3301:9200::51c2:1b9b	2a01:cb00:f2b:3b00:d080:1fdb:ca45:fd8	UDP	84	1212 → 1717 Len=22

Trace test adresses ipv6 et ipv4

C’est ici une trace Wireshark, elle démontre que nous envoyons des paquets via notre adresse ipv6 ainsi que via notre adresse ipv4, et de la même façon pour la reception.
Dans notre programme nous utilisons le port 1717 afin d’envoyer des paquets et nous écoutons également sur celui-ci lors de la réception. Nous pouvons donc bien voir que nous envoyons des paquets vers le port 1212 via notre v6 ainsi que via notre v4, et idem lorsqu’il s’agit de la reception.

Communication entre nos deux machines

ferences... Address: 2a01:cb00:f2b:3b00:a8e4:953e:e70c:254

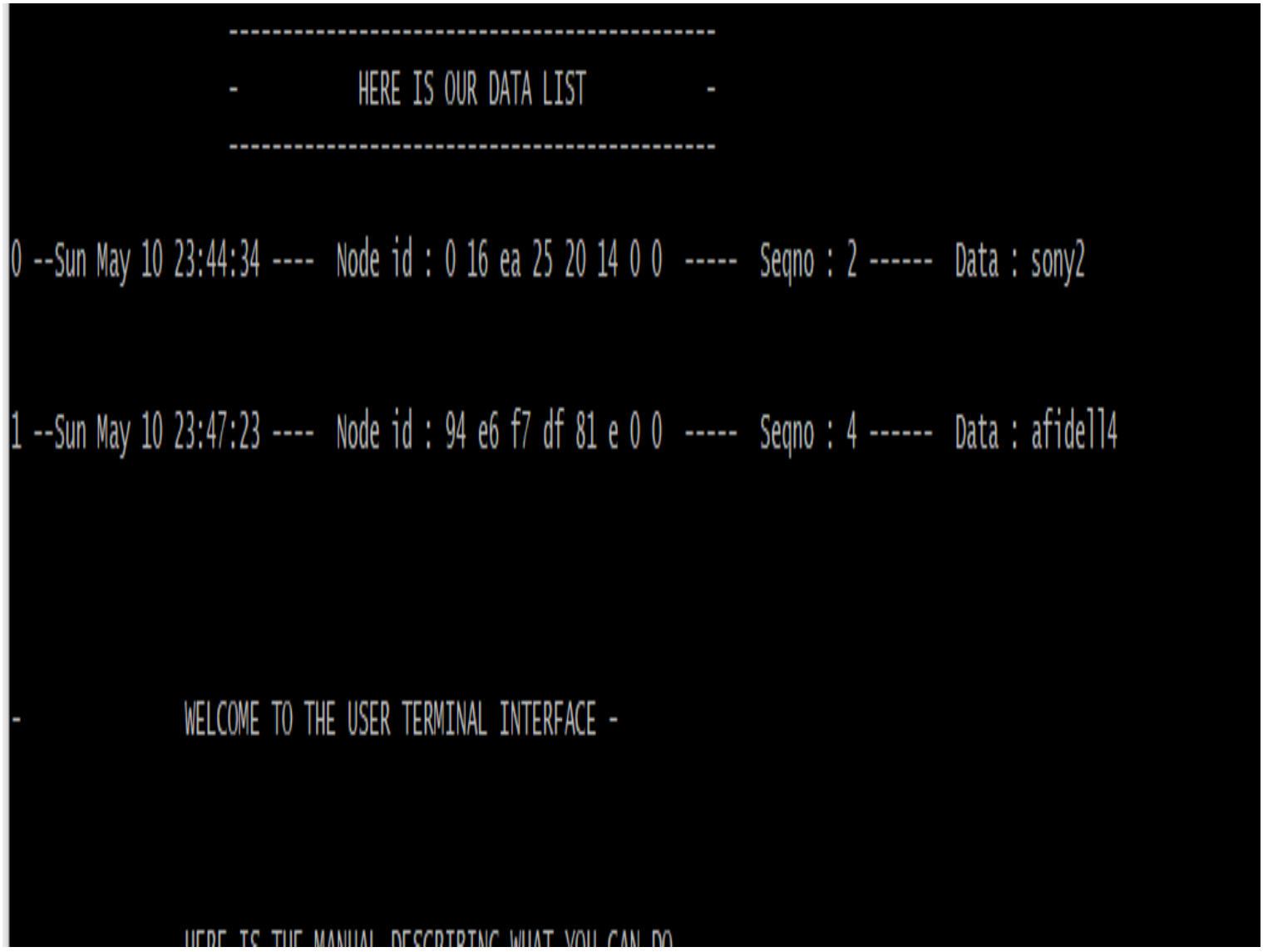
	Time	Source	Destination	Protocol	Length	Info
47	4.910819	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	84	1717 → 1717 Len=22
85	12.447131	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	84	1717 → 1717 Len=22
141	24.911270	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	84	1717 → 1717 Len=22
142	24.911925	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	68	1717 → 1717 Len=6
143	24.914375	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	122	1717 → 1717 Len=60
144	24.915318	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	76	1717 → 1717 Len=14
145	24.919730	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	103	1717 → 1717 Len=41
171	32.447969	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	84	1717 → 1717 Len=22
172	32.453681	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	68	1717 → 1717 Len=6
173	32.453948	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	122	1717 → 1717 Len=60
174	32.457936	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	76	1717 → 1717 Len=14
175	32.458268	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	103	1717 → 1717 Len=41
222	44.914219	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	84	1717 → 1717 Len=22
223	44.914220	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	68	1717 → 1717 Len=6
224	44.915294	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	86	1717 → 1717 Len=24
383	52.449202	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	84	1717 → 1717 Len=22
461	64.913887	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	84	1717 → 1717 Len=22
462	64.913888	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	68	1717 → 1717 Len=6
463	64.914684	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	86	1717 → 1717 Len=24
505	72.450018	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	84	1717 → 1717 Len=22
628	84.914757	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	84	1717 → 1717 Len=22
629	84.915503	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	UDP	68	1717 → 1717 Len=6
630	84.915902	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	86	1717 → 1717 Len=24
685	92.452799	2a01:cb00:f2b:3b00:a8e4:953e:e70c:254	2a01:cb00:f2b:3b00:a11e:bad3:7c63:680f	UDP	84	1717 → 1717 Len=22

= Version: 6

Trace test entre nos deux machines

Une nouvelle fois, une trace Wireshark, cette fois ci nous avons lancer le programme simultanément sur deux machines différentes afin d'observer le comportement de notre programme.
Nous pouvons voir sur cette trace les adresses ipv6 de nos de machines qui sont en train de s'envoyer des paquets et donc de communiquer.
La trace suivante montre bien que nous nous sommes ajoutés en tant que voisins à notre liste.

Nos deux machines sont voisines

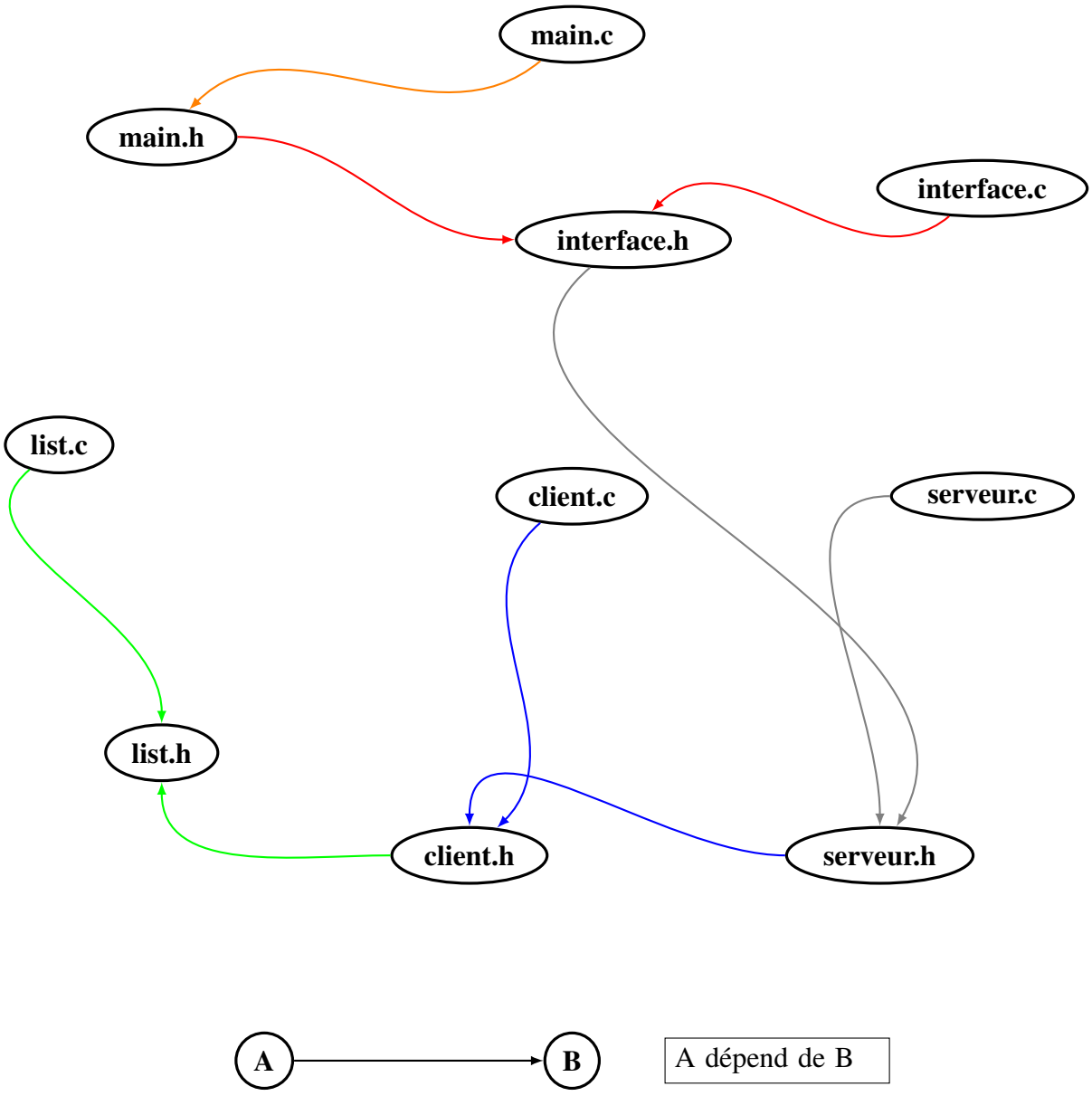


Notre trace dans l'interface utilisateur

Ici, c'est une trace provenant de l'affichage de notre interface utilisateur.
A la suite de notre test ci-dessus, la réception de paquets entre nos deux machines à engender un ajout à notre liste des voisins comme prévu.
Ici, nous voyons bien grâce à la fonctionnalité d'affichage de nos listes dans l'interface utilisateur, que notre deuxième machine est devenue un voisin.

Partie 6 - Schéma de dépendance entre nos fichiers et compilation

Schéma de dépendance et description de nos fichiers



Ci-dessus le schéma de dépendance entre nos différents fichiers. Ils possèdent chacun leurs .h respectif et contiennent les méthodes agissant explicitement sur leurs noms de fichiers. Interface.c/h contenant les méthodes relatives à l'interface etc ..

La compilation de notre projet

Pour compiler notre projet et l'exécuter nous avons un script qui compile chacun des fichiers nécessaires et qui ensuite fait un link entre ces différents fichiers afin de produire notre fichier d'exécution main.
Il suffit donc d'exécuter le script $./Makefile$, et ensuite d'appeler le fichier d'exécution produit de la manière suivante :

$./dazibao$ " $adresse$ " " $destination\_port$ " " $source\_port$ " " $level\_debug$".

Level_debug pouvant prendre les valeurs 0 pour les traces d'erreurs, 1 pour des traces partielles et 2 pour l'ensemble des traces.