

Rapport personnel réseau

VERSION THEORIQUE (maquette)

I. LES STRUCTURES

Pour commencer, nous avons défini les différentes structures qui nous semblaient indispensables à ce moment du projet : « addr », « nGroup » et « payload ».

J'ai ainsi fait 3 fichiers .h afin de les définir avec les headers des premières fonctions :

addr.h :

```
typedef char[BUFFER_SIZE] addr;
```

nGroup.h :

```
typedef struct{
    addr id;
    addr parentId;
    addr childId[NB_CHILDREN];
} nGroup;
```

payload.h :

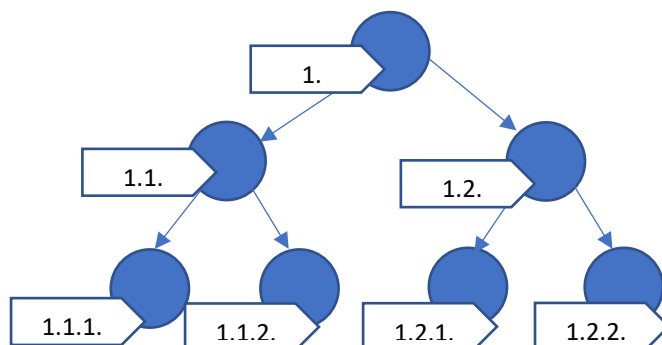
```
typedef struct{
    nGroup grpDest;
    char msg[LENGTH_MESSAGE];
    nGroup grpSrc;
} payload;
```

```
void emit(payload pl);
void receive(payload *pl);
```

Explications :

addr.h :

Au début du projet, nous pensions redéfinir notre propre système d'adresse afin d'accéder aux différents groupes, on l'avait ainsi imaginé de la manière suivante :



nGroup.h :

Pour définir un groupe, on a imaginé que c'était simplement une adresse, l'adresse de son parent (que l'on imaginait unique à ce stade) et les adresses de ses enfants.

payload.h :

Le « payload » correspond à un message qu'un groupe envoie à un autre. On s'est ainsi limité à un groupe source, un groupe destination et un message

II. FONCTION DEPTH

Cette fonction devait simplement retourner la profondeur d'un groupe par rapport au groupe principal « root ».

Tout simplement, en utilisant notre structure d'adresse, on compte le nombre de « . » dans la chaîne de caractères :

```
int depth(nGroup grp){
    int i, stage;
    for (i = 0; i < strlen(grp.id); i++)
    {
        if (grp.id[i] == '.')
            stage++;
    }
    return stage;
}
```

III. FONCTION RECURWAY (réalisée avec Clément Charpentier)

La structure à ce stade du projet correspondait à un arbre assez classique. Ainsi, afin de « se balader » sur l'arbre en cherchant le premier chemin qui nous emmène à notre destination.

```
int recurWay(nGroup* src, nGroup* dst, nGroup** road, int* n){
    int i;
    for (i = 0; i < src->nb_children; i++){
        if (src->id == dst->id){
            road[*n] = src;
            *n = *n + 1;
            return 1;
        }
        else if ((src->nb_children != 0)){
            road[*n] = src;
            *n = *n + 1;
            if (recurWay(src->children[i], dst, road, n) == 1)
                return 1;
        }
        *n = *n - 1;
    }
    return 0;
}
```

```
void getWay(nGroup* src, nGroup* dst, nGroup** road, int* n){
    nGroup* currentState = src;
    while (strcmp(currentState->id,"0") != 0){
        road[*n] = currentState;
        *n = *n + 1;
        currentState = currentState->parents[0];
    }
    recurWay(currentState,dst,road,n);
}
```

Explications

La fonction getWay() remonte jusqu'au groupe « ROOT » (ici 0), le groupe le plus haut placé et stocke son trajet dans road[].

La fonction recurWay() est simplement une fonction récursive de parcours d'arbre, inspiré du parcours en profondeur classique. Elle retrace le trajet de « ROOT » vers la destination et stocke son trajet dans road[].

VERSION PRATIQUE

I. Fonctionnement en réseau local

A ce moment-là, notre architecture fonctionnait uniquement sur un ordinateur, en utilisant les ports TCP. Afin de la faire fonctionner en réseau local, il fallait à la fois prendre en compte les ports et les adresses IP.

```
typedef struct {
    int descriptor;
    int port;
    char addip[BUFSIZE];
} group;
```

Ensuite, sachant qu'une adresse peut faire une taille maximale de 15 caractères, on a défini la taille à 16 (« XXX.XXX.XXX.XXX\0 »).

Il a ainsi fallu modifier chaque fonction existante en prenant en compte l'adresse IP et remplacer l'utilisation locale (127.0.0.1) par l'adresse IP du groupe.

De plus, afin de récupérer une adresse IP, il fallait connaître l'interface réseau sur laquelle communiquer. Ainsi, j'ai fait une fonction qui prend en paramètre le nom de l'interface (que l'on va mettre en argument du programme) et qui va renvoyer l'adresse IP de l'interface sur laquelle on communique :

```
void getIP(char* iface, struct in_addr *addr){
    struct ifaddrs *ifaddr, *ifa;
    int s, n;
    char host[BUFSIZE];
    if (getifaddrs(&ifaddr) == -1) {
        perror("getifaddrs");
    }
    for (ifa = ifaddr, n = 0; ifa != NULL; ifa = ifa->ifa_next, n++) {
        if (ifa->ifa_addr == NULL || ifa->ifa_addr->sa_family ==
AF_INET6)
            continue;
        else if (strcmp(ifa->ifa_name,iface) ==0){
            if((s = getnameinfo(ifa->ifa_addr,sizeof(struct
sockaddr_in),host,BUFSIZE,NULL,0,NI_NUMERICHOST)) == 0){
                inet_aton(host,addr);
            }
        }
    }
    freeifaddrs(ifaddr);
}
```

II. Debug envoi de messages (réalisé avec Clément Charpentier)

A ce moment-là, notre programme ne pouvait qu'envoyer des « attach » et « detach » et pourtant, nous avons des problèmes d'envoi/réception de paquets TCP. En effet, lorsqu'on envoyait une grande chaîne de caractères, TCP découpait le paquet et l'envoyait en plusieurs morceaux (tel qu'il est prévu) mais à la réception, c'était illisible par notre programme.

Or pour l'avenir où nous voulons envoyer des messages suffisamment longs, nous ne pouvons pas nous limiter à moins de 2000 octets. Ainsi, nous avons cherché à partir de quelle taille le programme *faisait n'importe quoi*. Puis, on a utilisé Wireshark pour analyser les paquets transmis, et c'est là qu'on a compris d'où venait le problème : notre programme n'attendait pas de recevoir tout le payload pour traiter les paquets. Ainsi, le payload reçu n'était pas conforme à notre structure.

La problème était une petite erreur à l'élaboration du programme, nous avons utilisé « read() » qui va prendre en compte uniquement le premier paquet qui arrive. Alors qu'il fallait qu'on utilise le flag « MSG_WAITALL » disponible uniquement avec la fonction « recv() ».

Ainsi, à la suite de ce debug, on a décidé de tester la robustesse de notre architecture en transmettant le plus gros paquet possible : nous avons envoyé un paquet de 4Go qui a été reçu avec succès (après 15-20 minutes d'attente).

III. Chiffrement des messages

J'ai travaillé sur le chiffrement des messages entre les groupes afin qu'ils ne soient pas lisibles en cas de sniff du réseau.

Pour le chiffrement, j'ai utilisé la librairie « openssl » qui permet de générer des clés RSA afin de chiffrer les messages avec la clé publique et de les déchiffrer avec la clé privée.

a. Fonctions

J'ai créé dans un fichier spécifique (rsa_tools.c) les fonctions principales à utiliser pour le chiffrement des messages :

```
void generate_rsa_keys();
unsigned char* encrypt_message(char* filename, unsigned char* from);
unsigned char* decrypt_message(char* filename, unsigned char* from);
RSA* createRSAPublicWithFilename(char* filename);
RSA* createRSAPrivateWithFilename(char* filename);
unsigned char* getPublicKeyFromFile(char* filename);
void savePublicKeyFile(group* g, unsigned char* content);
```

Explications :

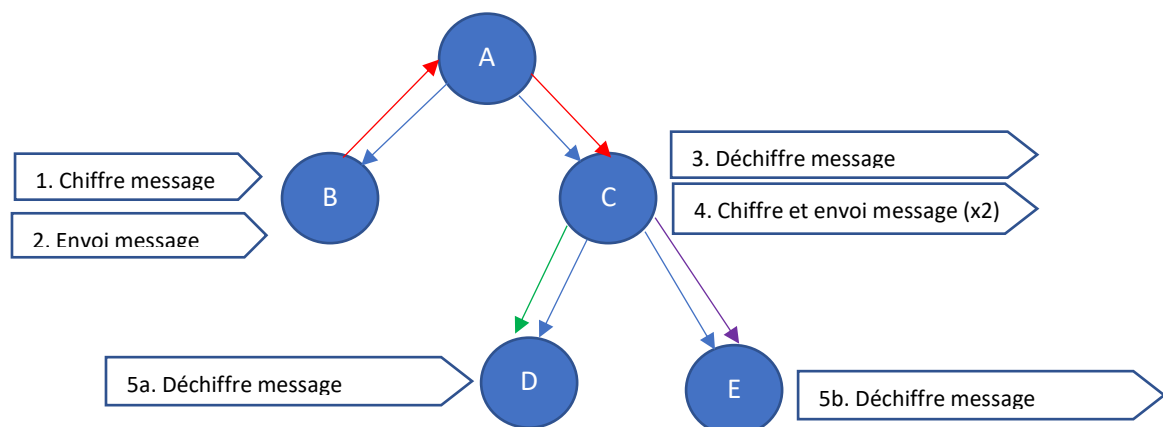
- La fonction « generate_rsa_keys » génère des clés de taille 2048 et les enregistre dans le dossier courant sous les noms « public.pem » et « private.pem »

- La fonction « `encrypt_message` » ouvre le fichier de la clé publique à utiliser pour chiffrer le message *from*.
- La fonction « `decrypt_message` » ouvre le fichier de la clé priver à utiliser pour déchiffrer le message *from*.
- Les fonctions « `createRSAxxxxxWithFilename` » ouvrent le fichier clé nommé « `filename` » donné en paramètre et renvoient la clé en type « `RSA*` ».
- La fonction « `getPublicKeyFromFile` » va récupérer la clé publique pour l'envoyer à ses enfants/parents lors de l'attach.
- La fonction « `savePublicKeyFile` » va enregistrer la clé publique reçue par un enfant/parent lors de l'attach.

b. Déroulement

1. Au démarrage du programme, on appelle `generate_keys()` pour créer les 2 clés de façon aléatoire.
2. Lorsqu'un groupe FILS s'attache à un groupe PERE, le FILS envoie sa clé publique en contenu du payload. Le PERE enregistre cette clé sous le nom « **adressesIP:port.pem** » et renvoi sa clé publique au FILS, qui va l'enregistrer sous un nom équivalent.
3. Lors de l'envoi de message, le FILS va envoyer un message à un PERE. Ainsi, le FILS chiffre le message avec la clé publique du PERE en question. Maintenant, le PERE veut envoyer le message à tous ses FILS, ainsi il déchiffre le message avec sa clé privée puis va rechiffrer le message mais avec la clé publique de chacun des FILS.

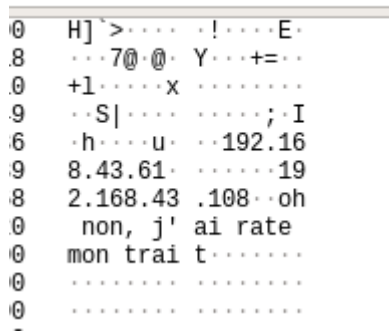
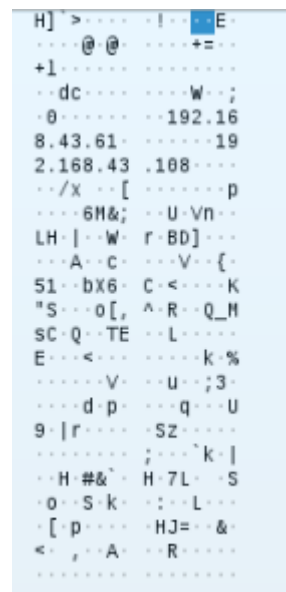
B envoie un message à C pour ses fils



c. Résultat

Lorsque que quelqu'un est connecté sur le même réseau que nous, il peut voir les messages envoyés/reçus en sniffant le réseau. Ainsi, afin de constater l'effet de la fonctionnalité, on a fait le test sur un réseau local et on a sniffé les paquets.

Dans les deux cas, le message envoyé est « oh non, j'ai raté mon trait » :

Non chiffré	Chiffré
 <pre> 0 H]`>... !...E. 8 ...7@...@ Y...+=... 0 +l...x... 9 ..S;I 6 .h...u... 192.16 9 8.43.61... 19 8 2.168.43 .108 oh 0 non, j' ai rate 0 mon trai t 0 0 - </pre>	 <pre> H]`>... !...E. ...@...@ +=... +l... ...dc... ..W...; ..0... ..192.16 8.43.61... 19 2.168.43 .108... .../x...[.....p ...6M&... ..U·Vn... LH· ·W· r·BD]... ...A·C·...V·{· S1·bX6· C·<...K "S·o[, ^·R·Q_M SC·Q·TE·L· E·<...k·% ...V·...u·;3· ...d·p·...q·U 9· r·...Sz· ...;...`k· ·H·#&· H·7L·S ·o·S·k·:·L· ·[·p·...HJ=·& <·,·A·R· </pre>
(Capture Wireshark)	(Capture Wireshark)

d. Problèmes rencontrés

- Taille : Lorsqu'on crée des clés de taille N, on ne peut pas envoyer des messages plus grands que N/8 caractères. Or, on voulait que notre application puisse envoyer des messages très long. Mais, si on essayait de générer des clés trop grandes, le programme pouvait mettre plusieurs minutes avant de démarrer. Ainsi, j'ai fait des clés de 2048 octets (pas de chargement visible) et les messages sont donc limités à 256 caractères
- Chiffrement/Déchiffrement : Pour des raisons plutôt floues, j'ai eu des problèmes lors du déchiffrement des messages. Pour déchiffrer un message avec une clé de 2048 octets, il faut absolument un message chiffré de taille 256. Or, parfois (7 fois sur 10 environ), après le chiffrement d'un message, la taille retournée est inférieure. Ainsi, lors du déchiffrement, la fonction renvoie une erreur. *[HYPOTHESE : je pense que la fonction calcule la taille du message avec strlen() ou équivalent et comme les messages cryptés sont écrits en base 64, il pourrait y avoir des '\0' dans le message, retournant ainsi une valeur inférieure à 256].* J'ai croisé ce problème sur les forums mais aucune solution viable. Ainsi, pour y remédier, je recommence l'opération de chiffrement des messages jusqu'à ce que la taille de sortie soit 256 (maximum 10 essais).
- Signature : Le chiffrement se faisant à l'aide d'une clé publique, il est difficile de s'assurer de la provenance du message. Ainsi, la seule façon de vérifier d'où provient le message, on considère que le couple adresse IP/port est unique et infalsifiable. J'ai essayé de créer un mot de passe afin de signer les messages mais je n'ai pas trouvé de moyen de vérifier l'origine des messages avec ce mot de passe.

e. Améliorations

- La taille de messages à chiffrer étant limitée à 256, il serait possible de récupérer un message plus grand puis de le découper en tranches de 256 pour les chiffrer une par une. Puis après le déchiffrement, on peut toutes les concaténer pour retrouver le message.
- L'application en mode chiffrée pourrait gérer l'envoi et la réception de fichiers
- Avec le fonctionnement que j'ai utilisé pour gérer les clés publiques, un groupe ne peut pas envoyer un message à un grand-père mais uniquement à son père, qui va lui-même l'envoyer à ses fils.

IMPORTANT :

De par sa complexité de fonctionnement, et le risque d'empiètement sur la robustesse de la version non cryptée de l'architecture, le chef de projet a approuvé le fait de ne pas l'intégrer directement dans l'application.

Il y a sur le GIT une branche nommée « RSA » qui contient une « ancienne » version de l'application (avec les fonctionnalités de base : attach/msg/detach) et qui implémente le chiffrement des messages.

Cette version est peu documentée mais fonctionne parfaitement en réseau (ne pas lancer l'application 2 fois depuis le même dossier, sinon les clés de la dernière lancée vont écraser les premières).

IV. Envoi de fichiers

Alors qu'on faisait des tests de notre architecture, j'avais besoin de récupérer des fichiers auprès de Pierre. On a trouvé dommage le fait d'être connecté, de pouvoir s'envoyer du contenu mais de ne pas pouvoir s'envoyer de fichier.

Ainsi, j'ai pris en charge cette fonctionnalité en me basant sur l'envoi des messages. Le système est complètement identique à part qu'on envoi dans le payload : le nom du fichier + son contenu (séparés par un '&' pour parser le contenu à la réception). Cette différence m'a obligé à définir de nouvelles fonctions et un nouveau type de payload PTYPE_FILE (et PTYPE_FILEROOT).

A la base, le système devait nous permettre d'envoyer tout type de fichier en lisant et écrivant les fichiers en mode « byte ». Pour des raisons que je ne comprends pas, ça n'a pas fonctionné pour d'autres contenu que du texte.

Amélioration possible

- Permettre d'envoyer tout type de fichier (texte brute, données compilées, données compressées, etc)
- Pouvoir envoyer des fichiers sans limite de taille. Cette amélioration nécessite de définir un payload de taille dynamique et ayant deux espaces de texte (nom et contenu du fichier).

A PROPOS

- J'ai développé depuis 3 systèmes différents où les noms « git » étaient différents. Ainsi, mes commits portent l'un des 3 noms suivant :
 - CFOISSARD
 - FOISSARD
 - TheRaccoon00
- Le sujet de base était assez difficile à comprendre et nous a demandé de beaucoup travailler ensemble avec des feutres et un tableau avant de nous y mettre. Ce projet m'a permis de rendre concret les cours de réseau du premier semestre. De plus, j'ai eu l'opportunité d'utiliser de nouvelles bibliothèques (comme openssl) et ai pu compléter mes connaissances en matière de programmation réseau.