

# Projet: Analyseur de Protocoles Réseau 'Offline'

Nous avons créés 4 fichiers pour ce projet: **interface.py**, **outils.py**, **analyse.py**, et **diction.py**. Le fichier **diction.py** contient tous les dictionnaires correspondants aux type, protocole, classe des couche de la trame. En entrée, nous avons **interface.py** qui permet de prendre un fichier contenant les octets codés en 2 chiffres hexadécimaux des trames capturés sur le réseau, et retourner une liste d'octet de ces trames. Puis nous utilisons **analyse.py** pour analyser chaque couche et retourner les informations qui devraient être affiché, à l'aide des fichiers **diction.py** et **outils.py** (contenant des fonctions utiles pour aider à analyser la trame). Et à la fin nous utilisons encore **interface.py** pour ouvrir un interface graphique qui permet d'afficher le résultat de l'analyseur dans une fichier texte.

---

## L'entrée

### – **interface.py**

Tout d'abord, on lit le fichier dans **interface.py**. On crée une fenêtre. Et on place un panedwindow sur la fenêtre. Et on place 2 frame sur le panedwindow.

On crée un barre de menu, en cliquant le bouton 'open', on peut sélectionner et ouvrir un fichier.

Avec la fonction **openfile()**, il peut avoir le path de la répertoire où le programme se place. Puis on efface tout sur les 2 frame. Et on lire le fichier on a choisir et on appelle la fonction **clean\_trame** qui analyse le trame. Les erreur va être affiché sur frame2.

Dans le fonction **clean\_trame**, il y a plusieurs parties. La première est **splitTrame(fichier)**, on lit le fichier et divise en plusieurs trames par '0000' (le premier offset).

La deuxième partie est **checkOffset(a\_trame, n)**, cette fonction est pour trouver des erreurs dans l'offset. On divise tout en lignes par '\n'. Et on prend 2 lignes chaque fois, trouve leur offset, vérifie si l'offset de deuxième est supérieur à la première. Et on vérifie si l'offset est du longeur 4. Et puis on vérifie si tous les octets sont en hexadécimal. Et on calcule les différence de nombre d'octet entre 2 ligne pour vérifie si ce nombre est bon.

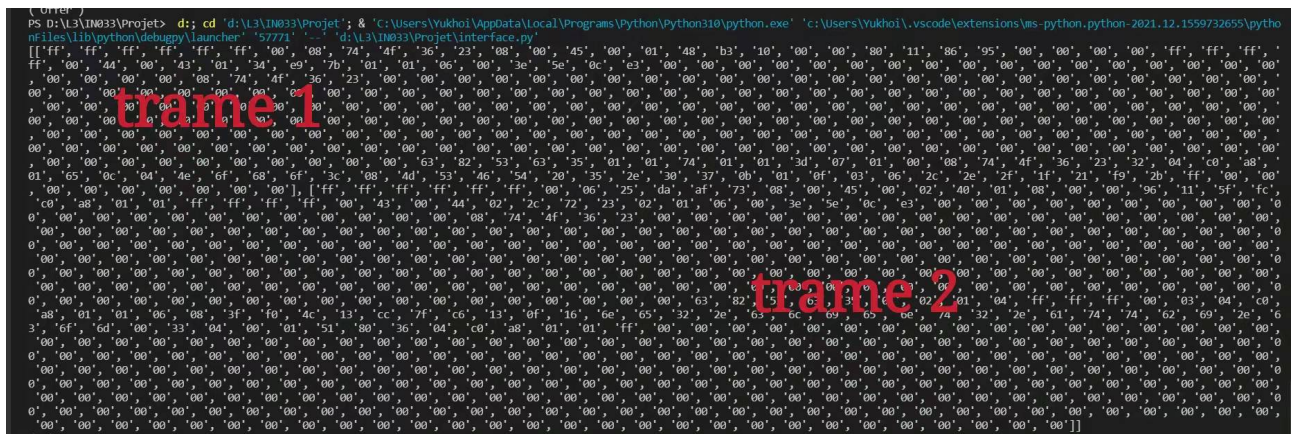
La troisième partie est **cut\_into\_bytes(trame)**, on enlève les offsets et on divise la trame en octet.

La quatrième partie est **rebuild\_trame(trames)**, on vérifie la liste n'a pas de cas vide, si oui, on l'enlève.

Il nous donne un liste de trame, et chaque trame est une sous-liste de octet.

Exemple :

```
(C:\Users\...> cd 'd:\L3\IM33\Projet'; & 'c:\Users\Yukhoi\AppData\Local\Programs\Python\Python310\python.exe' 'c:\Users\Yukhoi\.vscode\extensions\ms-python.python-2021.12.1559732655\pythonFiles\lib\python\debugpy\launcher' '57771' 'd:\L3\IM33\Projet\interface.py'
```



## L'analyse

Dans cette partie, on a trois fichiers **analyse.py**, **diction.py** et **utils.py** pour analyser les différentes couche.

### 1. Outils.py

Dans le fichier **utils.py**, nous avons tous les fonctions utiles pour aider à analyser la trame.

#### - **is\_IPv4(s)**

Nous écrivons d'abord une fonction **is\_IPv4** de paramètre s (chaîne de caractère), nous allons vérifier si s est égal à "0800", si oui, cette fonction retourne True, sinon Faux. Cette fonction est donc créer pour vérifier la type de cette trame, et elle retourne True si la trame est IPv4, False sinon.

#### - **affichage(ind, s1, s2, s3)**

Puis nous avons une fonction affichage qui permette de retourner une chaîne de caractère d'une concaténation des 3 chaînes de caractère s1, s2, s3, avec un paramètre ind qui indique l'indentation de cette chaîne.

#### - **to\_bin(i, octet)/to\_hex(i, octet)**

Et on a deux fonctions similaires **to\_bin** et **to\_hex**, comme leur nom indique, **to\_bin** est pour transformer un int i en binaire et **to\_hex** pour le transformer en hexadécimal avec le nombre d'octet que l'on souhaite. Nous utilisons d'abord 3 fonctions **int**, **bin** et **str** pour que la valeur i (qui devrait être en décimal) transforme en binaire, et donc nous pouvons retenir une chaîne de caractère b de valeur en binaire qui commence par 0b, et on utilise `binaire=b[2:]` pour l'enlever. Parce que la trame est composé par des octets codés par 2 chiffres hexadécimaux, donc 8 chiffres binaires, alors nous vérifions si la longueur de la variable binaire est un multiple de 8 fois le nombre d'octet, si oui, retourne le, sinon, nous ajoutons 8\*octet-1 fois "0" devant la variable binaire et le retourne. Même démarche pour **to\_hex** sauf 1 octet est 2 chiffre hexadécimaux donc 2\*octet au lieu de 8\*octet.

#### - **time(i)/index(liste, debut, fin, s)**

Ensuite, nous avons une fonction **time** qui permet de transformer un int en temps. Et une fonction **index** peut vérifier la validation de index de l'analyse: les index peuvent pas être plus ou égal à la taille de la liste.

### 2. diction.py

Dans ce fichier, nous avons les dictionnaires correspondants de la trame.

### 3. **analyse.py**

Nous allons commencer analyser la trame par la couche 2: Ethernet.

#### - **Ethernet(liste, position)**

Dans la fonctions **Ethernet**, nous créons d'abord une chaîne de caractère vide `res`, nous ajoutons le nom Ethernet dans `res`.

- Adresse destination: nous allons distinguer la position du premier et dernier octet de l'adresse destination: les variables `debut` et `fin`, on sait que l'adresse source est codé par 6 octets, donc `fin=debut+6`, nous vérifions que `debut` et `fin` sont bien des index dans la liste et nous lisons ces 6 octets puis les ajoutons dans le `res`.
- Adresse source: maintenant, le premier octet de l'adresse source est celui après le dernier octet de adresse destination, donc la variable `debut=fin` et `fin=debut+6` car adresse source est aussi codé par 6 octets. Puis on lise ces 6 octets et l'ajoute dans `res`.
- Type: Avec la même démarche, sauf que type est codé sur 2 octets, et nous utilisons la fonction **if\_IPv4** que nous avons déjà représenté pour vérifier si le type est IPv4.

À la fin de cette fonction, on retourne la variable `res`.

Nous travaillons ensuite sur la couche 3 IP.

#### - **IP(liste, position)**

Dans la fonction **IP**, nous avons aussi une chaîne de caractère `res`, nous ajoutons le nom IP dans `res`.

Ensuite pour toutes les informations de IP, nous devons localiser les octets correspondants, créer une chaîne de caractère par lecture de ces octets, si besoin, nous allons aussi transformer cette chaîne de caractère en décimal, et ajouter l'affichage de ces informations dans `res`. Mais il y a des parties spéciales dont la démarche précédente n'est pas suffisante:

- Fragmentation: Pour cette partie, nous devons transformer la chaîne de caractère en binaire à l'aide de la fonction `to_bin`, on obtient donc 16 chiffres binaires, les trois premiers bits sont R, DF, MF, et les 13 restes sont pour offset, pour cela, nous retransformons les 13 bits en chiffres hexadécimaux puis ajoutons l'affichage de ces informations dans `res`.
- Protocol: Pour le protocole, il y a une condition supplémentaire, il faut que ce protocole existe dans la dictionnaire `dict_protocol` de **diction.py**, sinon c'est un protocole inconnu.
- Option: Si header length n'est pas 20, c'est-à-dire qu'il existe des options à traiter, les parties type, length, pointer suivent le même traitement. Dans la partie type, on a encore copy, class, et number qui suivent le même logique.
- Padding: si length de l'option +20 est inférieure à header length de IP, on ajoute header length-20-length de l'option d'octets de padding dans `res`.

À la fin, nous retournons la variable `res` et `fin`.

Nous trainons maintenant la couche 4 UDP:

#### - **UDP(liste, position)**

Dans cette partie, nous avons port source qui indique le type de la couche 7 que nous allons avoir, port destination, length, et checksum que nous devons aussi localiser les octets correspondants, créer une chaîne de caractère par lecture de ces octets, si besoin, nous allons aussi transformer cette chaîne de caractère en décimal, et ajouter l'affichage de ces informations dans `res`. Enfin nous retournons le `res` et la variable `fin`.

Ensuite, nous allons analyser la couche DNS à l'aide des fonctions `name`, `queries` et `Answer`.

#### - **name(liste, debut, fin, name\_tab)**

Tout d'abord, nous allons voir la fonction **name**. C'est une fonction qui peut décoder les octets en lettre en parcourant une liste d'octet, en même temps, cette fonction compte le label count aussi. Nous déclarons une chaîne de caractère vide `n`, si le début de la liste est commencé par 00, alors `n=<Root>`. Sinon, tant que la variable l'octet du début dans la liste n'est pas 00, l'octet de l'index début est une taille d'après la transformation en valeur décimale, et lorsque la variable taille (déclaration en 0) est inférieure à la taille, on utilise `chr` pour décoder l'octet suivant (`fin`) en une lettre puis l'ajouter dans le nom `n`, d'après l'incrément de `fin` et `taille`, on continue la boucle jusqu'à la variable est égal à la taille, et on change `debut` en `fin`, `fin` en `debut+1`, on ajoute un point . après `n`, et la variable `taille` revient à 0, `label` devient `label+1` et ainsi de suite. Si `liste[debut]` n'est pas 00 mais est `c0`, alors nous utilisons l'octet suivant de `c0` pour calculer la différence entre cet octet et 0c puis localiser la position des octets de nom que l'on doit récupérer, et on retourne ce nom.

#### - **queries(liste, debut, fin, name\_tab, nb)**

Pour la fonction **queries**, nous avons `type`, `classe`, `name length` et `label count` à afficher, pour `type` et `classe`, on utilise la même façon, et pour `name length` on utilise `len`.

#### - **Answer(liste, debut, fin, name\_tab, nb)**

La fonction **Answer** est suffisante pour les parties **Answer**, **Authoritative** et **Additional records**, car ils ont tous le même format: **Name**, **Type**, **Class**, **TTL**, **RDLenght** et **RData**. Pour les parties `type`, `class`, `TTL` et `RDLenght` on utilise toujours la même méthode, pour `name`, on fait l'appel de la fonction **name**. Et pour **RData**, il y a des différents formats pour les différents types: soit il faut afficher encore un nom, on appelle alors encore une fois la fonction **name**; soit c'est de type **SOA**, nous avons alors `mname` et `rname` qui doivent faire encore un appel de fonction **name**, et pour `serial`, `refresh`, `retry`, `expire`, et `minimum TTL` nous utilisons encore la façon précédente; etc.

#### - **DNS(liste, position)**

Dans la fonction **DNS**, les parties **ID**, **Question**, **Answer RRs**, **Authority RRs**, et **Additional RRs** suivent tous la méthode que l'on a déjà expliqué, et dans la partie **Flags**, nous devons transformer la chaîne de caractère que nous avons obtenu en binaire à l'aide de la fonction `to_bin`, et chaque sous-partie prennent un ou plusieurs bits de cette valeur binaire. Enfin pour les parties **Queries**, **Answers**, **Authoritative**, et **Additional records**, nous utilisons la fonction précédentes: **queries** et **Answer**.

À la fin, nous retournons le `res`.

Enfin, nous pouvons travailler sur la couche 7 DHCP.

#### -**DHCP(liste, position)**

D'abord on prend 2 paramètres : le trame et la position. Et on calcule le source port de UDP pour vérifier si DHCP existe dans ce trame. Et on crée des dictionnaires pour les options DHCP, `type DHCP`, et `parameter request` de l'option 55 d'DHCP.

On vérifie si la source porte d'UDP est bonne, si oui, on commence à analyser.

On trouve si ce trame est le type **Boot Request** ou **Boot Reply**. Et puis, on trouve le **Hardware Type**, le `hlen`, le `hops`, le **Transaction ID** et le **Seconds elapsed**. Puis on trouve le **flag** et détermine si'il est en **broadcast** ou **unicast**. Et on détermine **Client IP adress**, **Your IP adress**, **Next server IP adress**, **Relay agent IP adress**, **Client MAC adress** et son `pading`. En plus c'est `sname` et `file` et `magic cookie`.

Et on travaille sur les options. On calcule la position de début d'option. Et on crée une variable pour calculer la longueur des options pour calculer le `pading`. On crée une boucle `for`. On vérifie si le premier octet d'option existe dans le dictionnaire, si c'est le cas, on l'analyse, sinon, on définit

son type comme 'Unknown'. A la fin de la boucle, on ajoute le longueur d'option pour avoir le premier octet de l'option prochaine.

Enfin on travaille sur le padding, et retourne les résultats d'analyse.

---

## En sortie

### – interface.py

Tout d'abord, on crée une fenêtre. Et on place un panedwindow sur la fenêtre. Et on place 2 frame sur le panedwindow.

On crée un barre de menu, en cliquant le bouton 'open', on peut sélectionner et ouvrir un fichier.

Avec la fonction **openfile()**, il peut avoir le path de la répertoire où le programme se place. Puis on efface tout sur les 2 frame. Et on lire le fichier on a choisir et on appelle la fonction **clean\_trame** qui analyse le trame. Les erreur va être affiché sur frame2.

Après, on appelle la fonction **creat\_button(liste)**, pour créer les bouton pour tout les trame qui reste (les trames qui a des problème de offset et format a été ignoré). Puis on place les boutons sur frame1. On définit les boutons de trame est de la couleur bleu.

En cliquant le bouton, on appelle la fonction **click()**, on commence à analyser les trame en appelant les fonctions **Ethernet, IP, Couche\_UDP, DNS et DHCP**. Et on écrit les résultat d'analyse dans un fichier de format txt. Et on appelle la fonction **creat\_small\_button(t,name)**, pour crée un petit bouton jaune pour chaque trame.

En cliquant le petit bouton jaune (appelle **click\_small\_button(t)**), on ouvre le fichier des résultats d'analyse.

---

## Conclusion :

On pourra encore améliorer notre interface graphique, et traiter plus d'options. D'après ce projet, on a mieux compris les couche réseau et savoir mieux utiliser le langage Python.