4 Utilisation de `python`

Exercice 16 Analyse de trames Ethernet

L'objectif de cet exercice est de décoder des trames Ethernet.

Dans un premier temps, vous décoderez des trames qui ont été enregistrées dans des variables (en binaire).

Exercice noté à rendre sur moodle avant le **26 mars 2023** à 23h59.
- rendez un seul fichier nommé `decodeur_trame.py`
- indiquez éventuellement dans les commentaires les parties ne fonctionnant pas
- **Testez votre programme !!!**

Préliminaires

Utilisation de `struct` Le package `struct` permet de manipuler, et notamment d'appliquer un masque sur un tableau d'octets (*bytes*) pour pouvoir sélectionner les éléments qui nous intéressent.

Exemple : supposons qu'on ait la suite de bits suivante :

```
00000100000000000000000000000000000010100001000001101000011010001
```

et que cette suite doive être interprétée de la manière suivante :

```
00000100 00000000000000000000000000001010 00010000 01101000 01101001
-------- -------------------------------- -------- -------- --------
 entier                entier                entier    char.    char.
```

On peut utiliser le module `struct` pour définir un masque qui permet de séparer la chaîne comme on le souhaite. La commande `unpack` prend en paramètre un *masque* et un tableau d'octet (*bytearray*) et retourne un tuple correspondant à la chaîne "démasquée".

Le masque s'écrit à l'aide d'une chaîne de caractère commençant par `'!'`. Puis on sépare la chaîne en groupe d'octets à l'aide des symboles :
- `B` pour des entiers représentés sur **1** octet
- `H` pour des entiers représentés sur **2** octets
- `L` pour des entiers représentés sur **4** octets
- n`s` pour un tableau de n caractères (sur n octets)

Il existe d'autres symboles pour les masques, voir la doc[3] *pour une documentation complète.*

Dans notre exemple, on souhaite décoder la chaîne en utilisant :

1. un entier sur 1 octet
2. un entier sur 4 octets
3. un entier sur 1 octet
4. 2 caractères

3. Utilisation de struct : **https://docs.python.org/3/library/struct.html**

Le masque que l'on devra appliqué sera donc : `"!BLB2s"`.
Utilisation :

```
import struct

# en python la chaine est représentée en octets, pas en bits
chaine = b'\x04\x00\x00\x00\n\x10hi'

unmask_chaine = struct.unpack("!BLB2s", chaine)
```

À la suite de l'exécution de ce script, `unmask_chaine` contient le tuple : `(4, 10, 16, b'hi')`. On peut accéder à chacun de ses éléments comme pour un tableau (Ex. `unmask_chaine[0]` ...).

16.1. Le tableau d'octets :

`b'\x00\x01\x00\x02trois\x00\x00\x00\x04\x05\x00\x00\x00\x06'`

contient 6 *champs* :

1. 2 entiers codés sur 2 octets
2. 5 caractères (compte pour 1 champ)
3. 1 entier codé sur 4 octets
4. 1 entier codé sur 1 octet
5. 1 entier codé sur 4 octets

Écrivez un script permettant de retrouver la valeurs des différents champs.

Problème des portions d'octets Le module `struct` ne permet de découper un *bytearray* en portion d'octet (Exemple : 4 bits correspondent à un entier, 4 bits correspondent à un autre). Or dans certains cas, les protocoles spécifient des champs sur moins d'un octet.
Exemple : un octet (*de valeur 106*) contient les bits :

`01101010`

- Si on souhaite récupérer seulement la valeur des 3 bits de droite (`010` = 2), on peut utiliser un modulo :
$$106 \ mod \ 2^3 = 2$$
- Si on souhaite récupérer seulement la valeur des 5 bits de gauche (`01101` = 13), on peut utiliser un *décalage de bits* en supprimant ceux qui ne sont pas utiles (ici 3) :
$$106 \ >> \ 3 = 13$$

16.2. Les 4 premiers bits (à gauche) de l'octet `11101101` (237 en décimal) représente un entier. Utilisez une commande `python` pour récupérer cette valeur.

16.3. Récupérez le fichier `decodeur_trame.py` sur moodle.

Décodage de paquet UDP

16.4. Écrivez une fonction `python decode_UDP` prenant en paramètre un tableau d'octets `data` représentant un datagramme UDP et retournant un couple :

- une chaîne de caractère représentant les informations de l'entête UDP :
  ␣␣␣␣␣␣␣␣+++␣Paquet␣UDP␣+++
  ␣␣␣␣␣␣␣␣␣␣␣␣␣Port␣source␣␣␣␣␣␣:␣...
  ␣␣␣␣␣␣␣␣␣␣␣␣␣Port␣Destination␣:␣...
  ␣␣␣␣␣␣␣␣␣␣␣␣␣Longueur␣totale␣␣:␣...
- les données contenues dans le segment UDP (après l'en-tête).

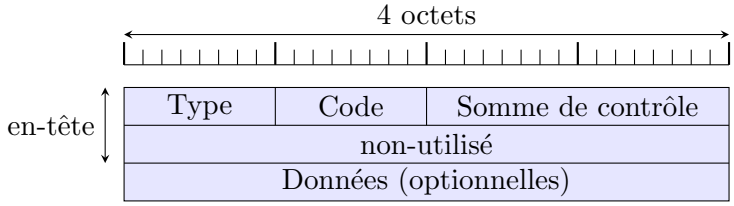
Décodage de paquet TCP

16.5. Écrivez une fonction `python decode_TCP` prenant en paramètre un tableau d'octets `data` représentant un segment TCP et retournant un couple :

- une chaîne de caractère représentant les informations de l'entête TCP :
  ␣␣␣␣␣␣␣␣+++␣Paquet␣TCP␣+++
  ␣␣␣␣␣␣␣␣␣␣␣␣␣Port␣source␣␣␣␣␣␣:␣...
  ␣␣␣␣␣␣␣␣␣␣␣␣␣Port␣Destination␣:␣...
  ␣␣␣␣␣␣␣␣␣␣␣␣␣Longueur␣en-tête␣:␣...
- les données contenues dans le segment TCP (après l'en-tête).

Décodage de paquet ICMP

ICMP est le protocole notamment utilisé par la commande ***`ping`****, il est encapsulé dans des datagramme IP.*



Format d'un paquet ICMP

16.6. Écrivez une fonction `python decode_ICMP` prenant en paramètre un tableau d'octets `data` représentant un paquet ICMP et retournant une chaîne de caractère représentant les informations de l'entête ICMP :

␣␣␣␣␣␣␣␣+++␣Paquet␣ICMP␣+++
␣␣␣␣␣␣␣␣␣␣␣␣␣Type␣:␣...

Décodage de datagramme IP

16.7. Écrivez une fonction `python decode_adresse_IP` prenant un paramètre un entier codé sur 32 bits représentant une adresse IP et retournant une représentation en chaîne de caractère de cette adresse (*`X.Y.Z.T`*).

16.8. Écrivez une fonction `python decode_IP` prenant en paramètre un tableau d'octets `data` représentant un datagramme IP et retournant un triplet composé :

- d'une chaîne de caractère représentant les informations de l'entête IP :
  `␣␣␣␣---␣Paquet␣IP␣---`
  `␣␣␣␣␣␣␣␣Version␣␣␣␣␣␣␣␣␣␣:␣...`
  `␣␣␣␣␣␣␣␣Longueur␣en-tête␣:␣...`
  `␣␣␣␣␣␣␣␣Protocole␣␣␣␣␣␣␣␣:␣...`
  `␣␣␣␣␣␣␣␣Adresse␣source␣␣␣:␣...`
  `␣␣␣␣␣␣␣␣Adresse␣dest.␣␣␣␣:␣...`
- du numéro de protocole encapsulé :
  - ◇ 1 pour ICMP
  - ◇ 6 pour TCP
  - ◇ 17 pour UDP
- des données encapsulées.

Décodage de trames Ethernet

16.9. Écrivez une fonction `python decode_adresse_MAC` prenant en paramètre un tableau de 6 octets représentant une adresse MAC et retournant une représentation en chaîne de caractères de cette adresse (*`A1:A2:A3:A4:A5:A6`*).

Indication : la commande `"%.2x" % n` permet d'obtenir une représentation en hexadécimal de `n`.

16.10. Écrivez une fonction `python decode_Ethernet` prenant en paramètre un tableau d'octets `data` représentant une trame Ethernet et retournant un triplet composé :

- d'une chaîne de caractère représentant les informations de l'entête Ethernet :
  `>>>␣Trame␣Ethernet␣<<<`
  `␣␣␣␣Adresse␣MAC␣Destination␣:␣...`
  `␣␣␣␣Adresse␣MAC␣Source␣␣␣␣␣␣:␣...`
  `␣␣␣␣Protocol␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣...`
- du numéro du protocole encapsulé :
  - ◇ 0x0800 (2048) pour IPv4
  - ◇ 0x86DD (34525) pour IPv6
  - ◇ 0x0806 (2054) pour ARP
  - ◇ ...
- des données encapsulées

Remarques : dans le protocole Ethernet, des données d'en-queue sont rajoutées à la fin de la trame, mais ici on ne les récupérera pas.

Recombinaison

16.11. Écrivez une fonction `decode_trame` prenant en paramètre une trame Ethernet et affichant l'encapsulation des protocoles et les informations (en utilisant les fonctions définies précédemment).

Vous devez être sous linux et disposer des droits administrateurs sur votre machine pour faire la question suivante.

La commande :

```
socket.socket(socket.AF_PACKET,socket.SOCK_RAW,socket.htons(0x0003)
    )
```

permet de définir un socket de niveau 3 écoutant directement sur la carte réseau et récupérant des trames Ethernet.

16.12. Définissez un tel socket et faites afficher les informations passant sur votre carte réseau.

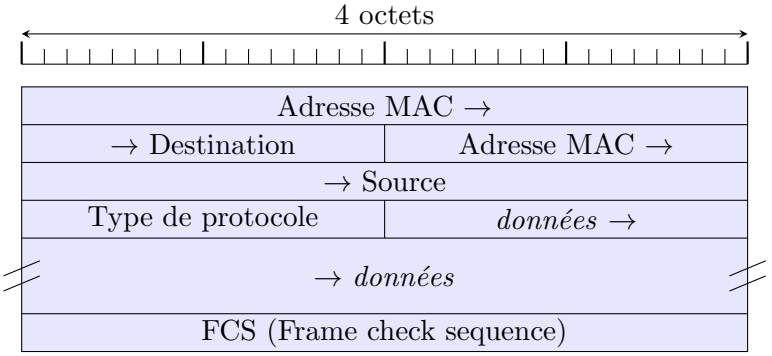
Testez votre programme :

- en effectuant un `ping` sur une machine distante (vérifiez les IP)
- en vous connectant à un site internet (en HTTP)

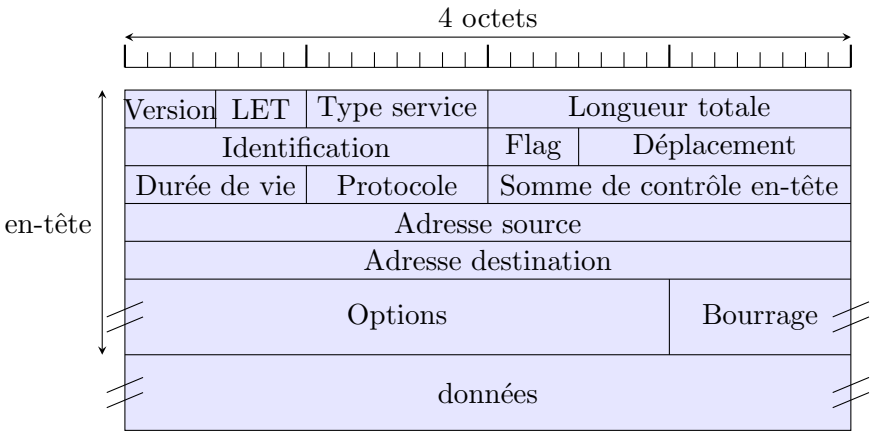
Vérifiez que vous obtenez bien les même informations qu'avec Wireshark.

Rappels sur les format des différents protocoles

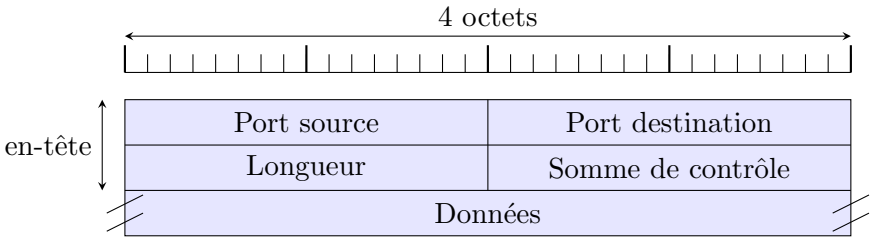
- Ethernet :



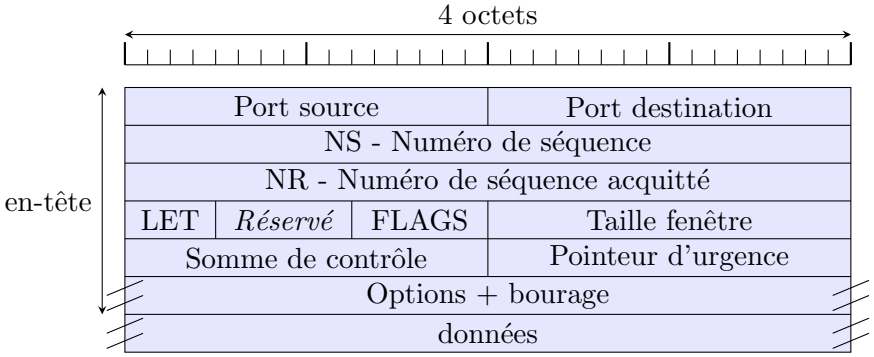
- IP :



- UDP :



- TCP :



Exercice 17 Serveur HTTP

L'objectif de cet exercice est de créer un serveur HTTP qui sera capable de transmettre des données à un navigateur classique (firefox, chrome, ...) le consultant.
Dans un premier temps, les questions seront à faire en local sur votre machine : vous utiliserez 127.0.0.1 comme adresse du serveur.

Exercice noté à rendre sur moodle avant le **02 avril 2023** à 23h59.
- rendez un seul fichier nommé `serveurHTTP.py` (*le fichier* ***`clientHTTP.py`*** *n'est pas à rendre*),
- indiquez éventuellement dans les commentaires les parties ne fonctionnant pas
- **Testez votre programme !!!**

Préliminaires

17.1. En reprenant les exercices du cours, créez deux fichiers de script `python` :
- `serveurHTTP.py` qui :
  - ◇ écoute sur le port 8080 ou un autre port s'il est déjà utilisé (*le port 80 est un port réservé : vous ne pouvez pas l'utiliser sans être* ***`root`*** *sur votre machine*),
  - ◇ affiche dans la console les requêtes reçues.
  - ◇ retourne le message de réception `"Bien reçu !"` aux clients qui lui ont envoyé une requête.
    (*Votre serveur ne doit pas traiter qu'une seule requête : utilisez une boucle infinie qui écoute en continu le port qui lui est attribué.*)
- `clientHTTP.py` qui :
  - ◇ effectue une requête HTTP sur l'adresse 127.0.0.1 (port 8080) pour la page `/index.html`,
  - ◇ affiche la réponse du serveur à l'écran.

  (*Votre client en revanche, n'envoie qu'une seule requête : n'utilisez pas de boucle infinie !*)

Lancez vos scripts (en commençant par le serveur) et vérifiez le bon fonctionnement.

Décodage de la requête HTTP

(Un squelette de code pour ***`serveurHTTP.py`*** *est disponible sur moodle.)*
17.2. Écrivez une fonction `python decode_requete_http` prenant en paramètre une chaîne de caractère représentant une requête HTTP et retournant un couple `page, options` où :
- `page` est la chaîne de caractères représentant la page demandée,
- `options` est un dictionnaire contenant les lignes d'en-tête de la requête.

Par exemple, pour :

```
requete = "GET /page1.html HTTP/1.1\r\nHost: localhost\r\n
    Accept-Language: fr-FR,en;q=0.3\r\n
    User-Agent: Mozilla/5.0 Firefox/98.0\r\n\r\n"
```

(sur une seule ligne dans votre code)

`decode_requete_http(requete)` doit renvoyer : `"/page1.html", dict`, avec :

```
dict : {'Host': 'localhost',
    'Accept-Language': 'fr-FR,en;q=0.3',
    'User-Agent': 'Mozilla/5.0 Firefox/98.0'}
```

Commande utiles :

- `chaine.split(c)` permet, à partir d'une chaîne de caractères `chaine`, de récupérer sous forme de tableau tous les champs entre les caractères `c` (les caractères `c` ne sont pas comptabilisés).
  Exemples :
  `"Le chat est noir".split(" ") -> ['Le', 'chat', 'est', 'noir']`

  `"le riz est de retour".split("r") -> ['le ', 'iz est de ', 'etou', '']`
- une chaîne de caractère est considérée comme un tuple de caractères : on peut utiliser les commandes `python` manipulant des tuples, notamment :
  - ◇ `chaine[3:6]` : permet de récupérer la chaîne composée des caractères 3 (inclus) à 6 (exclus) de `chaine`.
  - ◇ `chaine[:8]` : permet de récupérer le début de `chaine` jusqu'au caractère 8 (exclus).
  - ◇ `chaine[:-2]` : permet de récupérer `chaine` sans les 2 derniers caractères.

  Exemples :
  `"bonjour"[:3] -> 'bon'`
  `"bonjour"[2:6] -> 'njou'`
  `"bonjour"[:-1] -> 'bonjou'`

Construction de la réponse HTTP

Les pages que le serveur peut transmettre sont des fichiers html présents dans des dossiers spécifiques.

17.3. Récupérez le dossier `pages_serveur.zip` et décompressez le dans votre répertoire de travail (dans lequel se trouve `serveurHTTP.py`).

Rappel Une réponse HTTP a le format suivant :

```
HTTP/1.1 200 OK                                    -> Ligne de statut (version, code, sign.)
Date: Mon, 04 Apr 2022 09:06:33 GMT               \
Server: Apache                                    |
Upgrade: h2                                       | -> Lignes d'en-tête :
Connection: Upgrade                               |   infos supplémentaires sur la réponse
...                                               |
Content-Type: text/html                           /
                                                   -> Ligne vide
<html><head><title>Vous Etes Perdu ?...</html>     -> Données
```

Dans cet exercice, on aura deux types de réponse :

- Une réponse `"OK"` quand la page recherchée existe (et est accessible), dont l'entête sera :

```
HTTP/1.0 200 OK
Content-Type:text/html
Content-Length:#à calculer#
```

- et une réponse `"404 NotFound"` quand la page recherchée n'existe pas, dont l'en-tête sera :

```
HTTP/1.0 404 NotFound
Content-Type:text/html
Content-Length:#à calculer#
```

17.4. Écrivez une fonction `python get_reponse` prenant en paramètre l'url d'une page qu'un client cherche à consulter et retournant :

- si cette page existe, la réponse HTTP contenant le contenu de cette page (en une chaîne de caractères),
- sinon, la réponse d'erreur 404 avec le contenu de la page `pages_serveur/page404.html`.

Précisions :

- N'oubliez pas la ligne vide entre l'en-tête et les données.
- N'oubliez pas d'ajouter un passage à la ligne à la fin des données (et de le prendre en compte dans le calcul de la longueur).

Commandes utiles :

- la commande `fichier = open(nom_fichier,"r")` permet d'ouvrir un fichier existant sur la machine. La commande provoque une erreur si le fichier n'existe pas ou ne peut pas être ouvert.
- la commande `fichier.read()` permet de récupérer le contenu du fichier (comme une chaîne de caractère)
- Il est possible de 'récupérer' des erreurs provoquées par des commandes `python` en utilisant `try : ... except Exception : ...` :

```
try :
    commandes_pouvant_provoquer_une_erreur
except Exception :
    commandes_à_exécuter_en_cas_d'erreur
```

Exemples :

◇
```
a = 8
try :
a = a/2
print("pas d'erreur :",a)
except Exception :
a = a/4
print("une erreur !!",a)
```
Provoque l'affichage `"pas d'erreur : 4.0"`.

◇
```
a = 8
try :
a = a/0
print("pas d'erreur :",a)
except Exception :
a = a/4
print("une erreur !!",a)
```
Provoque l'affichage `"une erreur !! 2.0"`.

Serveur complet

17.5. Écrivez une fonction `python traite_requete` prenant en paramètre une requête HTTP et :

- qui décode la requête
- si le champ optionnel `"Accept-Language"` existe et commence par `"fr"` :
  - ◇ cherche la page recherchée dans le dossier `pages_serveur/fr/`
  - ◇ sinon dans le dossier `pages_serveur/en/`

Exemples :

```
traite_requete("GET /page1.html HTTP/1.1\r\n
   Host: localhost\r\n
   Accept-Language: fr-FR,en;q=0.3\r\n
   User-Agent: Mozilla/5.0 Firefox/98.0\r\n\r\n")

   (normalement sur une seule ligne)

-> retourne une réponse HTTP avec le contenu de
   'pages_serveur/fr/page1.html'

traite_requete("GET /pages/index.html HTTP/1.1\r\n
   Host: localhost\r\n
   Accept-Language: en\r\n\r\n")

-> retourne une réponse HTTP avec le contenu de
   'pages_serveur/en/pages/index.html'

traite_requete("GET /autres_pages/toto.html HTTP/1.1\r\n
Host: localhost\r\n\r\n")

-> retourne une réponse HTTP avec le contenu de
   'pages_serveur/en/autres_pages/toto.html'
```

17.6. Intégrez votre fonction `traite_requete` dans le code de votre serveur en le modifiant.

a. Vérifiez que vous obtenez bien les réponses HTTP attendues avec votre client (page en français, page par défaut en anglais, page erreur 404).

b. Connectez vous à votre serveur via votre navigateur (entrez dans la barre d'adresse : *adresse_IP :port_utilisé/url_de_la_page*). Vérifiez que vous obtenez les pages en français.

c. Modifiez les préférences de votre navigateur (dans Firefox : *Paramètres* → *Général* → *Choix de la langue préférée pour l'affichage des pages*), en plaçant 'Anglais [en]' en favori. Vérifiez que les pages envoyées par votre serveur sont en anglais.

17.7. *(non évalué)* En changeant l'adresse ip du serveur par celle de votre machine, vérifier avec d'autres binômes que vous pouvez accéder à des serveurs HTTP distants.

Exercice 18 Mini tchat en UDP

L'objectif de cet exercice est de programmer un mini chat en `python` qui utilisera le protocole UDP.
Dans un premier temps, les questions seront à faire en local sur votre machine : vous utiliserez 127.0.0.1 comme adresse du serveur.

Exercice noté à rendre sur moodle avant le **09 avril 2023** à 23h59.
- rendez deux fichiers nommés `serveur_chat.py` et `client_chat.py`,
- indiquez éventuellement dans les commentaires les parties ne fonctionnant pas
- **Testez vos programmes ! ! !**

Exemple d'utilisation

```
$ python3 serveur_chat.py
recv_data : b'__new_name__:pr'
recv_data : b'__new_name__:as'
recv_data : b'coucou'
recv_data : b'hello'
recv_data : b'__new_name__:hs'
recv_data : b'hey'
recv_data : b'je suis l\xc3\xa0 !'
recv_data : b'__quit__'
```

Serveur

```
$ python3 client_chat.py
Enter your name : pr

***pr*** vient de rentrer sur le chat !


***as*** vient de rentrer sur le chat !

[as] : coucou
hello

***hs*** vient de rentrer sur le chat !

[hs] : hey
[hs] : je suis là !
quit
$
```

Client1

```
$ python3 client_chat.py
Enter your name : as

***as*** vient de rentrer sur le chat !

coucou
[pr] : hello

***hs*** vient de rentrer sur le chat !

[hs] : hey
[hs] : je suis là !

***pr*** a quitté le chat.
```

Client2

```
$ python3 client_chat.py
Enter your name : hs

***hs*** vient de rentrer sur le chat !

hey
je suis là !

***pr*** a quitté le chat.
```

Client3

Principe de fonctionnement :

- niveau serveur :
  - ◇ le serveur garde en mémoire (dans un dictionnaire) une correspondance entre (adresse_ip_client, port_udp_client) et "nom_du_client"
  - ◇ le serveur reçoit trois types de message :
    - une première connexion d'un client, dont le message sera de la forme :
      `__new_name__:#nom#`
      qui entraîne l'envoi à tous les clients connectés (y compris celui-ci) du message :
      `***#nom#***␣vient␣de␣rentrer␣sur␣le␣chat␣!`
    - un message standard de la forme
      `__message__:#message_reçu_par_le_serveur#`
      qui provoque l'envoi à tous les clients connectés, excepté l'expéditeur, du message :
      `[#nom_expéditeur#]␣:␣#message_reçu_par_le_serveur#`
    - un message de départ de la forme :
      `__quit__`
      qui provoque :
      — l'envoi à tous les clients connectés, excepté l'expéditeur, du message : `***#nom#***␣a␣quitté␣le␣chat.`
      — l'envoi à l'expéditeur du message :
      `__quit__`
      — la suppression de l'adresse de l'expéditeur du dictionnaire des adresses.
      Rappel : pour supprimer un élément de clé `cle` d'un dictionnaire `dict`, on utilise :

      ```
      del dict[cle]
      ```

  - ◇ le serveur affiche dans sa console tous les messages qu'il reçoit sous la forme : `recv_data␣:␣#message_reçu_non_décodé#`
- niveau client :
  - ◇ au lancement du script, le programme commence par demander un nom à l'utilisateur avec le message :
    `Enter␣your␣name␣:`
    puis le script envoie un premier message au serveur de la forme :
    `__new_name__:#nom#`
    avec `#nom#` la chaîne de caractères entrée par l'utilisateur.
  - ◇ le client exécute ensuite deux actions en parallèle (on utilisera deux processus) :
    - un *receiver* qui attend de recevoir des données et les affiche lorsqu'elles arrivent.
    - un *sender* qui attend que l'utilisateur entre une chaîne au clavier avant de l'envoyer au serveur.
  - ◇ lorsque l'utilisateur du client souhaite quitter le chat, il tape `quit` (sans espace avant ou après) dans son terminal, ce qui provoque :
    - l'envoi du message :
      `__quit__`
      au serveur.

- l'arrêt du *sender* (par la commande `break` dans la boucle infinie).

◇ lorsque le client reçoit le message :
`__quit__`
il arrête le *receiver* (par la commande `break` dans la boucle infinie), et le programme doit terminer.

Préliminaires

18.1. En reprenant les exercices du cours, créez deux fichiers de script `python` :

- `serveur_chat.py` qui :
  - ◇ écoute sur le port 5005 (pour le protocole UDP) ou un autre port s'il est déjà utilisé,
  - ◇ affiche dans la console les requêtes reçues,
  - ◇ retourne le message de réception `"Bien reçu !"` aux clients qui lui ont envoyé une requête.
  (*Votre serveur ne doit pas traiter qu'une seule requête : utilisez une boucle infinie qui écoute en continu le port qui lui est attribué.*)
- `client_chat.py` qui :
  - ◇ envoie un paquet UDP (contenant ce que vous voulez) sur l'adresse 127.0.0.1 (port 5005),
  - ◇ affiche la réponse du serveur à l'écran.

  (*Votre client en revanche, n'envoie qu'une seule requête : n'utilisez pas de boucle infinie !*)

Client

18.2. Au début de votre script `client_chat.py`, définissez 3 constantes :

```
BALISE_NEW_NAME = "__new_name__:"
BALISE_MESSAGE = "__message__:"
BALISE_QUIT="__quit__"
```

18.3. Écrivez une suite de commandes demandant à l'utilisateur d'entrer son nom et envoyant au serveur le message de connexion avec la spécification donnée en introduction.

Comme expliqué en introduction, le script client aura deux tâches à effectuer en parallèle :

- un *receiver* qui attend de recevoir des données et les affiche lorsqu'elles arrivent.
- un *sender* qui attend que l'utilisateur entre une chaîne au clavier avant de l'envoyer au serveur.

18.4. Dans votre script `client_chat.py`, écrivez deux fonctions `send` et `receive` ne prenant pas de paramètre et programmant ces comportements.

Précisions :

- Utilisez des boucles infinies dans les deux fonctions
- Il n'y a aucune mise en forme des messages reçus à faire : elle est faite au niveau du serveur.
- Dans votre fonction `send`, le message envoyé sera soit précédé de `BALISE_MESSAGE` soit de `BALISE_QUIT` (lorsque le message entré est exactement `"quit"`).

Pour lancer les deux processus à partir de votre programme, vous pouvez utiliser le code suivant :

```
def send() :
    ...
def receive() :
    ...

# Création des processus
send_thread = threading.Thread(target=send)
recv_thread = threading.Thread(target=receive)

# Lancement des processus
send_thread.start()
recv_thread.start()
```

Remarque : lorsque vous lancerez vos scripts depuis un terminal, il faudra utiliser Contrôle+C pour arrêter le serveur.

Serveur

Proposition de résolution Pour implémenter votre serveur, vous pouvez :

a. Définir 3 constantes au début de votre script :

```
BALISE_NEW_NAME = "__new_name__:"
BALISE_MESSAGE = "__message__:"
BALISE_QUIT="__quit__"
```

b. Définir (comme variable globale) un dictionnaire `adresses` qui associera des clés (adresse_ip, port) à des valeurs noms (en chaîne de caractère).

c. Définir une fonction `send_entrance_notification` prenant en paramètre une adresse `addr` (techniquement un couple (adresse_ip, port)) et un nom `name`, et qui :
- vérifie que `addr` n'est pas déjà présente dans `adresses` et ajoute la correspondance `addr -> name`
- envoie à toutes les adresses de `adresses` le message :
  `\n***#nom#*** vient de rentrer sur le chat !\n`

d. Définir une fonction `send_message` prenant en paramètre l'adresse de l'expéditeur `addr` et le message envoyé `message`, et :
- qui envoie à tous les utilisateurs connectés (dans `adresses`) **sauf** l'expéditeur, le message :
  `[#nom_expéditeur#] : message_reçu_par_le_serveur`

e. Définir une fonction `send_quit_notification` prenant en paramètre l'adresse de l'expéditeur `addr`, et :
- qui envoie à tous les utilisateurs connectés (dans `adresses`) **sauf** l'expéditeur, le message :
  `***#nom_expéditeur#*** a quitté le chat.`
- qui envoie le message `__quit__` à l'expéditeur,
- qui supprime l'adresse `addr` du dictionnaire.

f. Définir une fonction `traite_data` prenant en paramètre l'adresse de l'expéditeur `addr` et le message envoyé `data`, et :

- si le message commence par `BALISE_NEW_NAME`, appelle `send_entrance_notification` avec les bons paramètres,
- si le message commence par `BALISE_MESSAGE`, appelle `send_message` avec les bons paramètres,
- si le message **est exactement égal** à `BALISE_QUIT`, appelle `send_quit_notification` avec les bons paramètres.

g. Placer `traite_data` dans votre boucle infinie pour traiter les messages entrant.

18.5. *(non évalué)* En changeant l'adresse ip du serveur par celle de votre machine, vérifier avec d'autres binômes que vous pouvez communiquer avec votre mini-chat sur des machines distantes.