1 PROGRAMMATION RESEAU

1.1. Rappels théoriques

1.1.1 Adresse physique

L'adresse physique, également appelée adresse MAC ou adresse Ethernet, est un nombre de 48 bits qui identifie de manière unique une carte réseau. Ce numéro est inscrit sur la carte réseau par le fabricant. L'adresse MAC est utilisée pour la transmission de données au niveau de l'accès au réseau dans le modèle TCP / IP. Une adresse MAC est représentée en regroupant les 48 bits en 6 octets et en écrivant chaque octet dans la base 16 (ex: **00:A0:C9:14:C8:29** ou **00A0-C914-C829**).

1.1.2 Adresse IP

Une adresse IP identifie une station dans un réseau à la couche internet du modèle TCP / IP.

Fondamentalement, l'IP d'une station est un nombre, 32 bits dans le cas du protocole IPv4 ou 128 bits dans le cas du protocole IPv6. L'adresse IPv4 est écrite sous la forme de 4 nombres en base 10, de **0** et **255**, séparés par des points (ex: **192.168.0.14**), et dans le cas d'IPv6, l'adresse IP est écrite sous la forme de 8 groupes de nombres en base 16, avec des valeurs comprises entre **0000** et **ffff**, séparées par deux points.
(ex: **2001:0db8:85a3:0000:0000:8a2e:0370:7334**).

1.1.3 Port

Le port est un identifiant utilisé pour la transmission de données au niveau de la couche transport. Pour garantir que deux applications communiquent dans un réseau, nous devons spécifier l'adresse IP des stations communicantes, ainsi que les ports utilisés. Alors que l'adresse IP garantit que les paquets arrivent à destination, le port garantit que le paquet reçu est destiné à l'application appropriée.

Par exemple, deux ordinateurs peuvent communiquer à la fois pour échanger des e-mails, mais aussi pour réaliser une visioconférence. Ainsi, les deux ordinateurs échangent des messages pour les deux applications (messagerie électronique et visioconférence). Alors que les adresses IP sont utilisées pour garantir que tous les paquets arrivent entre les deux systèmes, les applications de messagerie électronique et de visioconférence auront un port différent.

Dans les applications que nous développerons, nous devrons nous assurer de choisir des numéros de ports libres et inutilisés sur le système. Nous utiliserons les valeurs **8000** ou **8080**, qui ne sont pas utilisées par les services connus.

1.2. Socket

Afin de pouvoir transmettre des messages sur le réseau, nous utiliserons des structures de socket. Un socket permet la transmission de messages entre applications sur la même machine ou sur différentes machines physiques, d'une manière similaire à celle utilisant des descripteurs de fichiers.

En Python, nous utiliserons le module **socket**[1] pour les opérations de communication en réseau. Etudions les opérations que nous pouvons effectuer en utilisant le module **socket** figurent.

[1] voir https://docs.python.org/fr/3/library/socket.html

1.2.1 Obtenir des informations sur la station actuelle

La fonction gethostname renvoie l'adresse IPv4 de la station locale.

```
socket.gethostname()
```

1.2.2 Obtenir des informations sur d'autres éléments du réseau

La fonction getaddrinfo renvoie une liste de 5 éléments qui contiennent des informations sur une adresse et un port spécifiés. La liste renvoyée contient les informations suivantes: `(family, type, proto, canonname, sockaddr)`, et `sockaddr` est une autre liste qui contient `(address, port)`.

```
socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0)
```

La fonctiongethostbyname renvoie l'adresse IPv4 d'un `hostname`.

```
socket.gethostbyname(hostname)
```

1.2.3 Création de connexions TCP

En utilisant des sockets, nous pouvons réaliser deux types de transmission de données: UDP ou TCP.

Pour les deux types de transmissions, nous devons créer un nouvel objet **socket** à l'aide de la fonction **`socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)`**. Selon les paramètres transmis au constructeur de l'objet, le socket est configuré pour un certain type de transfert de données.

- **`family`** - représente le type d'adressage utilisé (ex: pour IPv4, le type utilisé est **`AF_INET`**, pour IPv6 le type est **`AF_INET6`**); dans notre cas, nous utiliserons **`AF_INET`** (**`socket.AF_INET`**, en Python);
- **`type`** - représente le type de transfert de données utilisé; en général, nous choisirons entre **`SOCK_STREAM`** (pour le transfert TCP) et **`SOCK_DGRAM`** (pour le transfert UDP);
- **`proto`** - représente le protocole utilisé, dans notre cas nous laisserons la valeur par défaut;
- **`fileno`** - représente un **`descripteur de fichier`**, et si ce paramètre est passé, toutes les 3 autres valeurs sont définies automatiquement en fonction de celui-ci.

```
import socket
s = socket.socket()
```

Pour le transfert **TCP**, on utilise le paradigme **client-serveur**, dans lequel un processus attend les connexions (**serveur**), tandis que d'autres processus peuvent s'y connecter (**client**). En utilisant **TCP**, les données transmises atteindront sûrement la destination ou une erreur sera reçue.

Initialisation du serveur :

Pour démarrer un serveur en mode socket, nous devons effectuer les opérations suivantes:

- Attribuer une adresse et un port au socket (bind)
- Préciser que nous attendons des connexions et combien (listen)
- Accepter les connexions entrantes (accept); la fonction renvoie un tuple avec deux valeurs: la connexion et une adresse; la connexion est une variable socket qui sera utilisée pour transmettre des messages; l'adresse est l'adresse du programme qui s'est connecté.

Si l'adresse IP transmise à la fonction `bind` est `0.0.0.0`, le socket prend en charge les connexions sur toutes les adresses IP du système.

Exemple :

```
import socket
s = socket.socket()
s.bind(('0.0.0.0', 8000))
s.listen(0)
conn, addr = serv.accept()
```

Initialisation du client :

Pour établir une connexion à partir du serveur, en utilisant le mode socket, nous devons appeler la fonction connect.
Pour mettre fin à la connexion avec le serveur, nous utiliserons la fonction close.

Exemple :

```
import socket
s = socket.socket()
s.connect(('localhost', 8000))
# send/receive data
s.close()
```

Pour se connecter à un service exécuté sur la machine locale, on peut utiliser l'adresse `localhost` ou `127.0.0.1`.

1.2.4 Échange de données en réseau

Envoi de données :

Pour envoyer des données à l'aide du module socket, nous pouvons utiliser les fonctions send et sendto.

```
socket.send(bytes[, flags])
socket.sendto(bytes, address)
socket.sendto(bytes, flags, address)
```

- La fonction **send** envoie des données uniquement si le socket est connecté à un autre socket, il ne peut donc être utilisé que pour une transmission TCP.
- La fonction **sendto** envoie des données à une adresse passée en paramètre, sans nécessiter de connexion entre les deux objets socket, afin qu'elle puisse être utilisée pour la transmission UDP.

Le paramètre **address** est un tuple de type **(adresse, port)**, par exemple:**("0.0.0.0", 8000)**.

Réception de données :

Pour recevoir des données en utilisant le module **socket** nous pouvons utiliser les fonctions recv et recvfrom.

```
socket.recv(bufsize[, flags])
socket.recvfrom(bufsize[, flags])
```

- La fonction **recv** lit les données à leur taille maximale, spécifiée comme paramètre (**bufzise**), et les retourne comme un objet **bytes**. Cette fonctionnalité convient à la transmission TCP, où nous avons une connexion constante à une autre socket.

- La fonction **recvfrom** lit des données similaires à la fonction **recv** , mais renvoie deux objets: le message et l'adresse source. Ainsi, nous pouvons utiliser l'adresse retournée pour renvoyer des messages. Cette fonctionnalité convient à la transmission UDP, où nous n'avons aucune connexion à une autre prise.

L'objet **address** renvoyé est un tuple de type **(adresse, port)**, par exemple **("0.0.0.0", 8000)**.

1.2.5 Exercice 1 – Création d'une application client/serveur UDP

Créez deux scripts python avec le code ci-dessous.

serveur_udp.py

```
# coding: utf8
import socket
import ipaddress

buffersize = 2048
# create socket
s = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
# bind socket to address and port
s.bind (("0.0.0.0", 8000))

# listen for data forever and reply with "[+] Server : Send message received"
while True:
    data, (ip, port) = s.recvfrom(buffersize)
    print (f"Data from {ipaddress.IPv4Address(ip)} is {data.decode()}")
    msg = str.encode ("[+] Server : Send message received")
    s.sendto (msg, (ip, port))
```

client_udp.py :

```
# coding: utf8
import socket
import ipaddress

buffersize = 2048

# create socket
s = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
# create bytes object
msg = str.encode("Hello")
# send message
s.sendto (msg, ("localhost", 8000))
# read message
data, (ip, port) = s.recvfrom (buffersize)
print (f"[+] Client : Data from {ipaddress.IPv4Address(ip)} is {data.decode()}")
```

1. Pour tester l'exemple, exécutez d'abord le serveur puis le client sur un autre terminal sur une même machine.

2. Copier le script serveur sur votre VM Kali Linux, modifiez le code de votre script client et tester la communication inter-machines. (utilisez python3 pour interpréter votre script)
3. Identifiez les adresses IP et numéros de port affichés par les deux scripts.

1.2.6 Exercice 2 - Création d'une application client/serveur TCP

Créez deux scripts python avec le code ci-dessous.

serveur_tcp.py

```
# coding: utf8
import socket
import ipaddress

buffersize = 2048
# create socket
s = socket.socket (family=socket.AF_INET, type=socket.SOCK_STREAM)
# bind socket to address and port
s.bind (("0.0.0.0", 8000))
# wait for connections
s.listen (0)
# accept connections forever
while True:
    conn, (ip, port) = s.accept ()
    print (f"Connected to {ipaddress.IPv4Address(ip)}:{port}")

    # listen for data and reply with "Message received"
    while True:
        data = conn.recv(buffersize)
        # client finished sending message
        if not data:
            break
        print (f"[+] Serveur : Received {data.decode()}")
        msg = str.encode ("[+] Serveur : Send message received")
        conn.send (msg)
    # close connection
    conn.close ()
```

client_tcp.py :

```
# coding: utf8
import socket

buffersize = 2048

# create socket
s = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
# connect to server
s.connect (("localhost", 8000))
# create bytes object
msg = str.encode("Hello")
# send message
s.send (msg)
# read message
data = s.recv (buffersize)
# close connection
s.close ()
```

```
print (f"[+] Client : Data is {data.decode()}")
```

1. Pour tester l'exemple, exécutez d'abord le serveur puis le client sur un autre terminal sur une même machine.
2. Copier le script serveur sur votre VM Kali Linux, modifiez le code de votre script client et tester la communication inter-machines (utilisez python3 pour interpréter votre script)
3. Identifiez les adresses IP et numéros de port affichés par les deux scripts.

1.2.7 Exercice 3 - Informations sur le réseau

1. Créez un script python qui affiche l'adresse IP d'un Full Qualified Name entré au clavier.

1.2.8 Exercice 4 - Netcat

Dans cet exercice, nous allons utiliser **`netcat`** pour échanger des messages avec des programmes python dans les exemples ci-dessus.

1. Exécutez l'application serveur à partir de l'exercice 2 de transmission TCP, puis exécutez la commande:

   `netcat -z -v 192.168.254.5 8000`

   Notez la réponse reçue.
2. Exécutez l'application serveur à partir de l'exercice 1 de transmission UDP, puis exécutez la commande:

   `echo test | netcat -u 192.168.254.5 8000`

   Notez le comportement résultant (l'option -u spécifie le mode de transmission UDP).
3. Envoyer un message au programme du serveur TCP en utilisant **`netcat`**.
4. Sur la VM Kali, exécutez la commande **`netcat -lu -p 8080`**, qui démarre un serveur UDP sur le port **`8080`**. Modifiez l'exemple de laboratoire pour envoyer un message au serveur démarré.
5. Exécutez la commande **`netcat -l -p 8080`**, qui démarre un serveur TCP sur le port **`8080`**. Modifiez l'exemple de laboratoire pour envoyer un message au serveur en cours d'exécution.

1.2.9 Exercice 5 - UDP

Créez deux programmes qui communiquent en mode UDP via socket afin que l'un des programmes reçoive trois nombres séparés par `;` et réponde avec leur moyenne arithmétique.

1.2.10Exercice 6 - Serveur TCP

Simulez un serveur TCP qui reçoit des commandes bash et répond avec leur résultat. Créez un client qui envoie des commandes et affiche le résultat reçu du serveur.