université de BORDEAUX

TP3

1 Introduction aux sockets avec le protocole Daytime

1.1 Utilisation du service Daytime avec Telnet

Telnet est un protocole simple de connexion à distance : il permet de transmettre des caractères entre une machine locale (écran + clavier) et une machine distante [1].

Par défaut, un client telnet se connecte au service TCP *telnet* (sur le port 23), mais il est possible de préciser le service TCP de la façon suivante : `telnet M S`. Dans ce cas, le client *telnet* se connecte au service S de la machine M.

- Quel est l'effet de la commande suivante ? `telnet time.nist.gov daytime`
- Décrire le protocole *daytime* d'après le résultat de cette commande. Vous trouverez sur le site web de l'IETF (*Internet Engineering Task Force*) la spécification officielle de ce protocole, décrite dans la RFC 867.
- Il est également possible d'indiquer explicitement le numéro de port du service plutôt que le nom du service. Consultez le fichier `/etc/services` et cherchez avec `grep` le numéro de port du service *daytime*. Essayez maintenant de vous connecter avec le numéro de port trouvé.
- Avec la commande de terminal *grep*, retrouvez efficacement les numéros de port par défaut associés aux services (ou protocoles) suivant : http, ftp, smtp, telnet, ssh, echo.
- Au CREMI, la machine *tesla* accueille le service *daytime* en version TCP et UDP sur IPv4 et IPv6. A l'aide de la commande `nc tesla 13`, essayez de vous connecter aux 4 différentes versions de ce service : on utilisera les options -u, -4, -6. Consultez le *manuel en ligne* de cette commande pour comprendre le rôle des options : `man nc` (faire 'q' pour quitter le manuel). Pourquoi ne peut-on pas utiliser la commande *telnet* pour faire ces expériences ? Rappelez les principales différences entre TCP et UDP. Expliquez rapidement la différence de fonctionnement du service *daytime* entre TCP et UDP ?

Pour info pour plus tard : Pour forcer la terminaison d'une session Telnet, il faut faire `Ctrl+Alt Gr+]`, puis il faut taper `quit` lorsque l'invite `telnet>` s'affiche.

1.2 Programmation d'un client Daytime avec les Socket

Considérons le programme en Python 3 ci-dessous, qui permet de se connecter au service *daytime* à l'aide de la bibliothèque *socket*, disponible sur moodle : daytime.py

1. À essayer depuis chez vous : `telnet towel.blinkenlights.nl`.

```python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = "time-c.nist.gov"
port = 13
s.connect((host, port))
data = s.recv(1024)
print(data)
s.close()
```

- Ouvrez votre éditeur de texte (ou environnement de programmation) préféré et recopiez le code de ce programme dans un fichier *daytime.py*.
- Lancez ce programme à l'aide de la commande : `python3 daytime.py`
- A l'aide de la documentation (https://docs.python.org/library/socket.html), comprenez les étapes principales de ce programme et repondez aux questions suivantes. En particulier, expiquez le sens des constantes `AF_INET` et `SOCK_STREAM`. Quel est le rôle de la variable *s* ? Quelles lignes déclenchent la connexion & la déconnexion TCP/IP ? Quelle fonction sert à recevoir des données dans ce code ? Cherchez dans la documentation la fonction qui sert à envoyer des données ?
- Pour transformer ce programme en script exécutable, il faut commencer par ajouter sur la première ligne : `#!/usr/bin/python3`, puis il faut donner les droits d'exécution (+x) au fichier *daytime.py* en tapant la commande en ligne suivante : `chmod +x daytime.py`. Vous pouvez maintenant lancez ce programme directement : `./daytime.py`

2 Socket Python : requête HTTP à la main

2.1 Echauffons-nous d'abord avec Telnet

Nous avons déjà étudié avec *Wireshark* ce qui se passe quand un navigateur consulte une page web, comme par exemple http://www.perdu.com. Revenons rapidement sur la trace capturée dans le fichier http.pcap. A l'aide de Wireshark, observez en détail dans l'en-tete HTTP la requête GET du client (trame 10).

A l'aide de la commande *telnet*, reproduisez la requête "HTTP GET" vers le serveur www.perdu.com en se limitant aux options *Host* et *Connection* :

```
$ telnet www.perdu.com 80
GET / HTTP/1.1
Host: perdu.com
Connection: close
```

Attention, il ne faut pas oublier le double saut de ligne à la fin de la requête ! Aussi, le serveur n'est pas très patient et produit rapidement une erreur `Timeout`, il vaut mieux préparer le texte prêt à coller depuis un éditeur de texte.

2.2 Programmons notre client web !

Écrivons maintenant dans le fichier *httpget.py* un programme Python 3 qui récupère la page d'accueil d'un site web (passé en argument), dont l'utilisation sera :

```
$ ./httpget.py www.perdu.com
```

Nous allons nous inspirer de *daytime.py* écrit précédemment. Puisque l'on veut ouvrir une connexion TCP sur le port 80, il faut adapter de la façon suivante :

- Importer le module *sys* avec `import sys`, afin de récupérer l'argument `sys.argv[1]` dans la variable `host`.
- Mettre à jour la variable *port*.
- Ajoutez une variable *request* qui contient la requête HTTP (disponible sur moodle : http-request.py)

```
request = "GET / HTTP/1.1\r\n"  \
    "Host: " + host + "\r\n"    \
    "Connection: close\r\n\r\n"
```

- Utiliser la méthode `sendall()` de l'objet socket pour envoyer la requête. Attention, il faut convertir la chaîne de caractères *request* en tableau d'octets avant de l'envoyer : `request.encode("utf-8")` qui retourne un tableau d'octets. Pensez bien à ajouter deux retours chariot `\r\n\r\n` à la fin de la requête.
- Utiliser la méthode `recv()` de l'object socket pour récupérer le résultat (on peut lui passer la taille 1024). Il suffit ensuite d'utiliser un `print()` pour l'afficher.
- Testez votre programme avec le site web www.w3.org. Pourquoi ce dernier est-il tronqué ?
- Que votre requête soit correcte ou pas, le serveur va toujours retourner une page web du moment que la connexion est faite. Comment savoir si le serveur a bien compris votre requête ? Corrigez éventuellement votre code.

2.3 Raffinements

Pour récupérer toute la page, il faut mettre une boucle autour des appels `recv()` et `print()`. Lorsque le serveur referme la connexion TCP parce que la page est finie, `recv()` retourne le tableau d'octets vide `b""`, il suffira alors de tester cela pour savoir quand s'arrêter.

Par ailleurs, il faut noter que `recv()` retourne un tableau d'octets qu'il faut convertir en chaîne de caractères (*str*) Python avec la fonction `decode()` avant de l'afficher avec `print()`. Comme la chaîne de caractères contient déjà des retours lignes `'\n'`, il n'est pas nécessaire que la fonction `print()` en rajoute (option `end=''`) :

```
answer = s.recv(1024)
# affichage en byte array
print(answer)
# decodage et affichage en string utf-8
print(answer.decode("utf-8"), end="")
```

En cas d'erreur lors des étapes de connexion, il vaut mieux afficher un gentil message à l'utilisateur plutôt que laisser l'exception terminer le programme ! On peut récupérer une exception avec la construction syntaxique suivante :

```
try:
    # ...
```

```
except Exception as e:
    print("socket error!")
    sys.exit(1)
```

Traitez différemment les cas d'erreur pour apporter à l'utilisateur une aide spécifique à chaque cas : essayez par exemple avec `localhost` (sur lequel aucun serveur web ne tourne) et avec `trucbidule` (qui n'existe pas), cela lève une exception différente.

3 Serveur Echo en Python

Nous allons maintenant étudier la programmation d'un serveur TCP en Python, en prenant l'exemple du service *echo*.

Le service `echo` est un service des plus simples : il répète ce qu'on lui dit. Il existe à la fois en version UDP et en version TCP (et même en AppleTalk). Quel est son numéro de port ?

On ne pourra pas lancer notre serveur sur ce port-là car les ports de numéro inférieurs à 1024 sont réservés à l'utilisateur `root`. On utilisera donc le port 7777.

3.1 Petit rappel

Il sera utile de se référer aux documentations suivantes :

— socket : https://docs.python.org/3/library/socket.html

— string : https://docs.python.org/3/library/stdtypes.html#str

Par ailleurs, il faut rappeler que les fonctions `send() & recv()` de la bibliothèque *socket* manipulent uniquement des tableaux d'octets. Par conséquent, on aura souvent besoin de faire des conversions entre des tableaux d'octets (`b"h\xc3\xa9ho"`) et des chaînes de caractères (`"hého"`). La différence entre les deux est simplement la notion d'encodage des caractères, on va prendre comme convention que les tableaux d'octets encodent les chaînes de caractères en UTF-8 :

Pour convertir une chaîne de caractères en un tableau d'octets avant de pouvoir envoyer vers le réseau :

```
s = "hého"
c = s.encode("utf-8")
```

Et inversement, quand on a lu un tableau d'octets depuis le réseau et que l'on veut l'afficher :

```
c = b"h\xc3\xa9ho"
s = c.decode("utf-8")
```

3.2 Version TCP

Un serveur TCP fonctionne presque de la même manière qu'un client TCP, la différence essentielle est que l'on doit gérer à la fois une socket d'écoute des connexions et les sockets pour les clients. On se contente ici de gérer un client à la fois.

— Créer une *socket* à l'aide de la fonction `socket.socket()`, de famille `socket.AF_INET6` (qui gère à la fois en v4 et en v6), de type `socket.SOCK_STREAM`, et laisser le protocole 0 pour que le système choisisse automatiquement TCP.

— Utiliser la méthode `bind` de la socket pour greffer notre *socket* au port 7777. Pour l'adresse, il suffit de spécifier le paramètre `("", 7777)` pour être à l'écoute sur toutes les cartes réseaux de la machine. Attention, il faut donc mettre 2 parenthèses, pour que `("", 7777)` soit un seul argument.
— Appeler la méthode `listen` pour indiquer au système qu'on va accepter des connexions. Un backlog de 1 suffira.
— Dans une boucle infinie,
  • Appeler la méthode `accept` pour accepter une connexion entrante. Notez bien que cette fonction vous retourne un tuple contenant en premier une *nouvelle* socket, représentant la connexion acceptée, dont on va appeler les méthodes `recv` et `send`, et en deuxième son adresse qui ne nous sera pas utile. Il faut bien sûr conserver la socket initiale pour le prochain `accept`, pour l'instant on s'occupe seulement de cette connexion.
  • Dans une boucle infinie,
    — Utiliser la méthode `recv` pour réceptionner des données, en lui passant comme taille 1500. Si la longueur des données reçues est 0, c'est que le client s'est déconnecté et l'on peut utiliser `break` pour sortir de la boucle.
    — utiliser la méthode `send` pour ré-expédier les données à l'envoyeur. Il suffit de passer le message obtenu !
  • Fermer la *socket* de la connexion à l'aide de la méthode `close`.

Cette version passe ainsi son temps à effectuer `accept`, une boucle de `send/recv`, et `close`. Pour tester, utilisez `nc localhost 7777`. Utilisez `netstat -tuap` (ou `ss -tuap`) pour remarquer la présence de votre serveur. Relancez `nc` plusieurs fois, pour constater que le numéro de port côté client change effectivement à chaque fois.

Il se peut que `bind` échoue avec l'erreur `Address already in use`. Si vous regardez dans netstat ou ss, vous verrez une ligne du genre

```
tcp6       0      0 ::1:7777                ::1:49346               FIN_WAIT2   -
```

Cela signifie donc que le système préfère éviter que vous relanciez un serveur sur ce port alors qu'il reste encore des connexions qui ne se sont pas terminées, et il faut alors attendre quelques minutes. Pour éviter que le système soit si précautionneux, avant l'appel à `bind` utilisez appeler la méthode `setsockopt` pour mettre l'option `socket.SO_REUSEADDR` (dans le *level* `socket.SOL_SOCKET`) à 1. Notez que si vous avez eu ce problème, cette option ne va pas fonctionner immédiatement, car il aurait fallu que cette option soit positionnée par l'instance du serveur que vous aviez lancée. Patientez quelques minutes, et les instances suivantes, lancées avec l'option, permettront d'en lancer d'autres.

3.3 Version UDP (bonus)

La version UDP est relativement simple à réaliser.
— Créer une *socket* de famille `socket.AF_INET6` et de type `socket.SOCK_DGRAM`.
— Utiliser ensuite la méthode `bind`.
— Dans une boucle infinie,
  — Appeler la méthode `recvfrom` de la socket.
  — Appeler la méthode `sendto` de la socket pour ré-expédier le message à l'envoyeur. Il suffit de passer le message et l'adresse obtenus !

Pour tester, vous pouvez utiliser `nc -u localhost 7777` . Utilisez `strace` pour bien

observer les appels effectués par votre serveur. Utilisez `netstat -tuap` (ou `ss -tuap`) pour remarquer la présence de votre serveur. Relancez `nc` plusieurs fois, pour constater que le numéro de port côté client change effectivement à chaque fois.