TP Programmation Socket

Réseaux : Configuration & Programmation

Dino López – dino.lopez@unice.fr

Comme nous l'avons vu en cours, la programmation de Sockets en Python est une procédure relativement simple, si on comprend bien l'algorithme du client ou du serveur à écrire.

Ce TP a donc pour objectif de réaffirmer vos connaissances sur la programmation de sockets TCP. Cependant, nous allons commencer tout d'abord par connaître netcat, qui est considéré comme le « couteau suisse » des réseaux, afin de mieux comprendre la différence entre le protocole UDP et TCP.

1 Premier contacte avec les sockets : UDP vs TCP

Netcat (exécuté sous la commande nc ou netcat) est un outil qui vous permet de mettre en place des services réseaux très rapidement en quelques commandes (par exemple, faire du *port forwarding* pour atteindre un serveur se trouvant derrière un firewall).

Pour exécuter netcat en mode serveur (càd, en attente des requêtes), vous devez utiliser le paramètre « -l » (comme *listen*). Voici un résumé de l'utilisation de netcat, valable principalement pour les systèmes Linux. Pour indiquer le port d'écoute du serveur, utilisez le paramètre « -p ». Par défaut, netcat utilise le protocole TCP. Pour utiliser le protocole UDP, il faut ajouter le paramètre « -u ». Donc, un serveur UDP qui écoute sur le port 1234 peut être créé avec la commande « nc –u –l –p 1234 ».

Un client qui se connecte vers un serveur est créé uniquement en fournissant comme arguments l'adresse IP ou nom de la machine, plus le numéro de port d'écoute de la machine distante. La remarque faite précédemment en ce qui concerne le protocole UDP et TCP applique ici. Exemple d'un client TCP qui se connecte vers le port HTTP de www.unice.fr « nc www.unice.fr 80 ».

1) Tous les tests à faire dans cette section devront être exécutés sur le réseau *single* de Mininet. Déployez le réseau.

2) Puisque netcat lit depuis stdin et écrit sur stdout sa sortie, comment devriez-vous faire pour envoyer un fichier texte disponible dans le home directory du host h1 vers le dossier /tmp du host h2 en utilisant le protocole TCP, plus la commande cat ?

   H2) nc –l 1234 > /tmp/fichier.txt
   H1) cat ~/fichier.txt | nc 10.0.0.1 1234

3) Avec le protocole TCP, le serveur doit être démarré après ou avant le client ? Argumentez.

   Avant le client. Sinon, l'essaie connexion depuis le client échoue. En effet, TCP est un protocole qui travaille en mode connecté.

4) Avec le protocole UDP, que se passe-t-il si le serveur est démarré après que le client ait commencé d'envoyer des données ? Argumentez.

   Le protocole UDP n'ayant pas besoin d'une connexion, le client peut envoyer des messages au serveur, même si ce dernier n'est pas présent. Les données seront perdues dans le réseau, mais aucune erreur n'est remarquée côté client.

5) Déployez netcat en mode serveur UDP sur h1. Essayez de vous connecter au serveur depuis h2 à l'aide de netcat en mode client TCP. Que se passe-t-il ? Argumentez.

   Le client TCP ne peut pas se connecter au serveur UDP. La connexion échoue, et le serveur UDP ne voit rien passer. En effet, le multiplexage au niveau de la couche réseau, qui envoie les informations à la pile protocolaire TCP, empêche que l'application « serveur UDP » voit quoi que ce soit.

2 Les sockets UDP – Programmation d'une application de diffusion TV

Malgré le titre de cette section, vous allez travailler sur la programmation d'un programme Python assez simple et rudimentaire, mais qui utilise les bases de la transmission des images pour la télévision numérique terrestre (TNT) ou sur Internet (IPTV). Cela vous permettra par la même occasion de voir l'impacte que les caractéristiques de l'application à déployer ont sur la programmation des sockets de communication.

Premièrement, sachez que la vidéo à transmettre ici (video1.mpg à télécharger depuis la page web du cours) est déjà disponible sous le bon standard vidéo : MPEG-TS. Mais ce quoi MPEG ? et, ce quoi TS ? De manière très brève, MPEG est un standard qui fournit les règles pour créer des conteneurs sur lesquels la vidéo (et audio si disponible) seront envoyés. Le standard MPEG-2 spécifie 2 format de conteneurs : un format PS (Program Stream pour ses sicles en anglais), qui est utilisé lorsque la vidéo passe par des canaux de communication très fiables (e.g. bus d'un lecteur DVD), et un format TS (Transport Stream), conçu pour la transmission des fichiers dans des liens à faible fiabilité (ou dit autrement, des liens à très forte probabilité de perte d'information), tel que le réseau Internet ou les systèmes de broadcast de télévision. Le standard MPEG2-TS est donc utilisé dans la TNT (mais attention, c'est le standard MPEG-4 qui est utilisé pour la TNT-HD, d'où la incompatibilité de certaines TV actuelles avec le système TNT-HD qui est en train de se mettre en place cette année en France).

Par rapport au MPEG-PS, le standard MPEG-TS prévoit l'utilisation des systèmes de corrections d'erreurs et de synchronisation, ce qui permet de maintenir une bonne qualité d'image en présence d'un taux de perte d'information importante et de montrer les images au plus vite lors qu'on se connecte au flux vidéo.

Voici maintenant les exercices proposés. Notez que les questions concernent aussi les caractéristiques de l'application car cela impacte directement la solution réseau que vous proposerez :

6) Sachant que le fichier video1.mpg a été encodé à 638kbps en mode CBR (Constant Bit Rate), calculez à partir de la taille du fichier la longueur en temps du flux vidéo à transmettre.

= (4785540*8/638000) seconds. 4785540 a été obtenu par la commande « ls –l »

7) A votre avis, pourquoi dans les systèmes TNT la vidéo est codée en mode CBR et non pas en mode VBR (Variable Bit Rate) ?

Un encodage en mode CBR simplifie le circuit électronique de la TV, ce qui rend l'appareil moins cher.

8) Expliquez pourquoi le système MPEG-TS est mieux adapté que le système MPEG-PS pour la transmission de TV terrestre ou satellitaires.

Le lien sans-fil, entre l'émetteur du signal TV et l'antenne de la TV, étant très peu fiable dû aux plusieurs facteurs d'interférence, le système de correction d'erreurs très robuste du MPEG2-TS le rend le candidat idéale pour la transmission de la TNT.

9) Déployez avec Mininet un réseau de test avec la commande « mn --topo single,3 ». Ce réseau Mininet sera utilisé dans tous les prochains exercices de la partie sockets UDP, sauf si autre chose vous est indiqué.

   a. Quelles sont les caractéristiques du réseau déployé ?

3 hosts, un switch

   b. Sur h1 exécutez vlc de la manière suivante « sudo –u your_real_login vlc udp://@ :5000 ». La commande précédente indique que vous exécutez vlc avec les droits de votre utilisateur standard, et vlc lit le flux multimédia depuis une socket UDP, par la première interface réseau disponible et par le port 5000. Si le host possédait plusieurs interfaces (e.g. h1-eth0 avec 10.0.0.1 et h1-eth1 avec 11.0.0.1), il aurait fallu indiquer spécifiquement sur quelle interface vlc devrait écouter (e.g. udp://@11.0.0.1 :5000)

   c. Sur h2 exécutez la bonne combinaison de commandes qui permettrait d'envoyer le contenu de video1.mpg au host h1 (port 5000 bien sûr) avec la commande netcat. Donnez la commande.

Par extension de l'exercice 2), cat video1.mpg | nc 10.0.0.1 5000

   d. Commentez le résultat de votre expérience

VLC joue la vidéo, mais une bonne partie de la vidéo est perdue. Cela s'explique par le fait que la totalité de la vidéo est envoyé à la vitesse maximale des liens du réseau. Or, VLC ne lit qu'à une vitesse de 638kbps, comme indiqué dans les métadonnées du fichier MPEG.

10) En reprenant le code des exemples fournis en cours pour l'écriture d'un programme UDP, et en s'inspirant du code disponible à l'adresse https://docs.python.org/2/library/socket.html, créez un programme UDP robuste qui prévoit le management d'erreurs pouvant se produire lors de la manipulation et communication par sockets.

a. Votre programme ne fera pour l'instant qu'envoyer le mot « Hello » et finira tout de suite son exécution.

b. L'adresse et port du récepteur devront être donné en paramètre lors de l'exécution du programme. E.g. « python difftv.py 10.0.0.1 1234 ». Notez que pour obtenir le premier argument (dans l'exemple, 10.0.0.1) dans un script python, il faut préalablement importer le module « sys » et l'obtenir par sys.argv[1].

c. Sur h1, arrêtez vlc et exécutez netcat en mode serveur UDP. Exécutez votre programme sur h2 pour envoyer le message vers h1 et vérifiez que vous recevez correctement le message.

11) Modifiez votre programme pour que le client envoie le contenu du fichier video1.mpg (au lieu du texte « Hello »).

a. Sachant que le flux multimédia sur video1.mpg a été encodé en mode CBR à 638kbps, indiquez combien d'octets et à quelle fréquence vous devez envoyer le contenu afin que la lecture de video1.mpg se passe de manière optimale chez le client.

b. Codez votre proposition faite en a). Vous pouvez ouvrir un fichier binaire en python, en mode lecture, avec par exemple, « f = open('workfile', 'rb') ». Plus tard, vous pouvez lire « n » octets du fichier, et le garder dans une variable, avec « str1 = f.read(n) ».

c. Exécutez vlc à nouveau sur h1 et votre programme python sur h2. Vérifiez que la video est jouée comme si vlc lisait le fichier en local (pas de fragments de film coupés, lecture fluide de la vidéo).

Une version simplifiée de l'exercice ici. A vous de jouer pour obtenir la version complète et robuste demandée.

```
import socket
import struct
import time
import sys

DPORT= int(sys.argv[3])
DHOST = sys.argv[2]      # any available interfaces
LPORT = int(sys.argv[1])   # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', LPORT))
try:
    global f
    f = None
    f = open("video1.mpg",'rb')
except IOError:
    print "Error while opening the file. Exiting..."
    s.close()
    exit(-1)
data = f.read(7975)
while data != "":
    s.sendto(data,(DHOST,DPORT))
```

```
        time.sleep(0.1)
        data = f.read(7975)
    print "end"
    s.close()
    f.close()
```

12) Si la vitesse d'émission calculée en 6a est réduite de moitié, quel serait le résultat attendu ? Vérifiez votre réponse avec une expérience faite sur Mininet

La lecture de la vidéo s'arrête temporairement. Cependant, la vidéo est jouée dans sa totalitée.

13) Si la vitesse d'émission calculée en 6a est doublée, quel serait le résultat attendu ? Vérifiez votre réponse avec une expérience faite sur Mininet

Des segments de la vidéo sont perdus.

14) Pour cette question, utilisez le réseau mininet créé avec la commande suivante : « mn --topo single,3 --link tc,bw=0.7 ». Exécutez vlc sur h1 et sur h3, et depuis h2, exécutez votre script python pour envoyer la vidéo vers h1, attendre que la moitié de la vidéo se soit écoulé, et ensuite, envoyer aussi la vidéo sur h3.

a. Quelle commande sur h2 permet d'accomplir cet objectif ?

b. En prenant en compte les caractéristiques du réseau, expliquez le résultat de cette expérience.

En gros, très peu de bande passante pour une double émission de la vidéo. Conclusion : comme vu en cours, trop de flux qui passent par une seule liaison introduit de problèmes de congestion et dégrade la qualité de la communication.

3 Les sockets TCP - *Your virtual friend*

Dans cet exercice, vous allez déployer un mini-programme d'AI qui sera accessible depuis le réseau par le biais d'un serveur TCP.

15) Déployez un réseau de test Mininet, avec la commande « mn --topo single,3 », qui sera utilisé dans tous les tests effectués pour cette section.

16) Pour commencer, en reprenant le code des exemples fournis en cours, et en s'inspirant du code disponible à l'adresse https://docs.python.org/2/library/socket.html, créez un serveur TCP robuste qui prévoit le management d'erreurs pouvant se produire lors de la manipulation et communication par sockets, dans un fichier echoserver.py

a. Pour l'instant, ne prévoyez pas un serveur multiprocessus. Donc, lorsqu'un client arrive au serveur, le client suivant doit attendre que le premier finisse sa session.

b. Votre programme ne fera pour l'instant qu'attendre un message de la part du client, et ensuite, renvoyer le message reçu au client. Ainsi, vous

créerez le serveur d'écho, qui est l'équivalent du « Hello World » pour les tutoriaux d'apprentissage des langages de programmation.

c. Le port d'écoute du client devra être donné en paramètre lors de l'exécution du programme. E.g. « python echoserver.py 1234 »

17) Une fois que votre serveur est prêt sur h1, démarrez le, et sur h2 exécutez netcat en mode client TCP. Vérifiez que votre serveur d'écho fonctionne correctement.

18) Reprenez les parties nécessaires de votre programme echoserver.py et créez un serveur multithread qui repose sur les appels à fork() pour créer un processus chargé de fournir le service au client connecté, tandis que le processus principal reste dans l'attente d'un nouveau client.

a. Donc, 2 ou plusieurs client doivent pouvoir créer une session sur le serveur, en parallèle.

b. Prévoyez le management de processus zombie. Lorsqu'un client part, le processus zombie doit disparaître. Voir pages 31 et 32 du cours.

c. Le service fourni par votre serveur, pour l'instant, est toujours le service d'écho.

Voici une version très simplifiée du code attendu. Par exemple, la dernière ligne ne sera jamais exécutée très probablement. La solution serait l'installation d'un filtre du signal SIGINT (envoyé avec la combinaison de touches Ctrl+C très souvent) afin d'appeler une fonction qui exécute la dernière ligne à la réception d'un tel signal (ceci n'est pas montré en cours, ni discuté ici, car il fait partie d'autres cours de votre parcours). A vous de jouer pour obtenir la version complète et robuste demandée.

```
import socket
import signal
import os

def do_echo():
    s.close()
    while 1:
        data = conn.recv(1024)
        if not data: break
        conn.sendall(data)
    conn.close()
    os._exit(0)

HOST = ''
PORT = 5000
signal.signal(signal.SIGCHLD, signal.SIG_IGN); # Zombie proc mngmt
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
while 1:
    conn, addr = s.accept()
    pid = os.fork()
    if pid == 0:
        do_echo()
    else:
```

```
            conn.close()
    s.close()
```

19) Faites les expériences nécessaires pour vérifier que tout marche bien. Donnez la procédure suivi, commandes et sorties illustrant le bon fonctionnement de votre programme.

20) Maintenant, téléchargez le fichier eliza.py depuis la page web du cours. Ce fichier contient une implémentation du très connu « chatbot » Eliza en 1966 par Joseph Weizenbaum. Le code d' eliza.py a été publié par *evan* depuis la page web https://www.smallsurething.com/implementing-the-famous-eliza-chatbot-in-python/ et nous l'utiliserons sans modifications supplémentaires.

    a. Modifiez votre programme d'écho multithread pour mettre en place le service eliza. Comme vous le constatez, une fois que les sockets sont codées, n'importe quel service peut-être fourni.

    b. Pour mettre en place le service eliza, vous devez importer la fonction analyze() du fichier eliza.py (e.g. « from eliza import analyze », en supposant qu'eliza.py se trouve dans le même dossier de yourfiend.py)

    c. Sachez qu' analyze() prend comme argument une chaîne de caractères qui correspond à la réponse de l'utilisateur et renvoie une chaîne de caractères qui correspond à la réponse d'eliza à l'utilisateur.

    d. Le jeux marche de la manière suivante : eliza commence et demande comment ça va. L'utilisateur répond et eliza fournit une réaction. N'hésitez pas à lire les fonctions du code eliza.py (principalement le main, qui illustre l'interaction eliza<->utilisateur) et à exécuter eliza.py directement pour le tester.

Le programme avec eliza.py n'étant qu'une version « avancée » du serveur d'écho, il n'y aura pas de correction fournie pour cette partie du TP

4 Proposition d'une nouvelle architecture pour votre programme de diffusion TV

Dans la partie sockets UDP, vous avez découvert un problème de passage à l'échelle de votre application. Il est temps maintenant de mettre en place ce que vous avez appris à propos de la programmation multicast pour proposer une nouvelle architecture de votre programme de diffusion TV qui supporte mieux l'arrivée de plusieurs clients.

21) Déployez le réseau de test avec la commande « mn --topo single,3 --link tc,bw=0.7 ».

22) Expliquez avec des mots, en un court paragraphe, ce que vous devez faire afin de palier au problème détecté dans la section 1.1, exercice 9).

Utiliser une application Multicast. Un seul lien permettra maintenant de servir à plusieurs clients.

23)Traduisez votre proposition en un diagramme de flux

24)Codez votre proposition et testez-la avec vlc sur h1 et h2, et votre programme Python sur h3.

a. Attention, vous devez indiquer à VLC d'écouter l'adresse Multicast à laquelle votre serveur envoie des données. Donnez la commande.

sudo –u your_real_login vlc udp://@226.1.1.1:5000

b. Si vous avez bien rentré la commande, VLC doit vous dire que la route vers l'adresse multicast n'existe pas. Ajoutez une route vers le réseau multicast en spécifiant comme dispositif de sortie l'interface ethernet du host (e.g. h1-eth0 pour h1)

Cela dépend de la plage d'adresse multicast. Par exemple, pour adresser toutes les adresses multicast par la même interface on fera : route add –net 224.0.0.0 netmask 224.0.0.0 dev h1-eth0

c. Répétez la commande donnée pour la question 24a et vérifiez que tout marche bien cette fois-ci

d. Exécutez votre serveur et vérifiez la bonne lecture de la vidéo sur tous les clients. Expliquez pourquoi cette fois-ci tout marche correctement.

Voici une version très simplifiée du code attendu. A vous de jouer pour obtenir la version complète et robuste demandée.

```
import socket
import time

DHOST = '226.1.1.1'
DPORT = 5000
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.IPPROTO_IP,socket.IP_MULTICAST_IF,socket.INADDR_A
NY)
f = open("video1.mpg",'rb')
data = f.read(7975)
while data != "":
    s.sendto(data,(DHOST,DPORT))
    time.sleep(0.1)
    data = f.read(7975)
print "end"
f.close()
s.close()
```