TP Programmation Socket

Dino Lopez Pacheco dino.lopez@univ-cotedazur.fr

1 Introduction

Ce TP a donc pour objectif de réaffirmer vos connaissances sur la programmation de sockets UDP et TCP par la pratique, et mieux comprendre la différence entre ces 2 protocoles du point de vu de l'application. 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.

2 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). Dans ce TP, nous utiliserons netcat pour créer un serveur ou un client UDP ou TCP, qui peut être très pratique pour savoir si notre serveur/client TCP/UDP programmé en C s'exécute correctement. Mais d'abord, utilisons-le pour comprendre quelques différences clés entre les sockets TCP et UDP.

Pour exécuter netcat en mode serveur (càd, en attente des connexions ou 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 écoutant sur le port 1234 peut être créé avec la commande « nc -u -l -p 1234 ».

Un client se connectant vers un serveur est créé 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 ».

Vous pouvez également ajouter à netcat l'option « -v » à netcat si vous voukez voir ce que netcat fait « en backstage » pour se connecter à une machine distante. Ceci vous donnera des indications sur les actions prises par netcat et éventuellement, debugger les erreurs rencontrées.

- Tous les tests à faire dans cette section devront être exécutés sur le réseau single de Mininet. Déployez le réseau avec 3 hosts : mn --topo single,3 --switch lxbr
- Dans un terminal pour h1, créez un serveur UDP, à l'attente des requêtes sur le port 5555.

  nc -u -l -p 5555
- Dans un terminal pour h2, déployez netcat en mode client UDP et échangez des messages avec le serveur si possible. Déployez maintenant netcat en mode client TCP et faite de même. Expliquez le comportement de vos clients.

  Client UDP à créer avec « nc -u -v 10.0.0.1 5555 » et sans le « -u » pour TCP. Le client TCP échoue sa demande de connexion car même si l'adresse IP du serveur et le port sont correctes, le protocole de transport est différent.

- Cette fois-ci, déployez un serveur TCP sur h1 et déployez le bon client sur h2. Donnez les commandes.

  Serveur : « nc -l -p -v 5555 ». Client « nc -v 10.0.0.1 5555 »

- Avec le protocole TCP, le serveur doit être démarré après ou avant le client ? Utilisez netcat si nécessaire afin de donner votre réponse.

  Avant le client. Dans le cas contraire, l'essaie de connexion depuis le client échoue. TCP = communication en mode connecté.

- 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, si l'application ne met pas en place un système d'acquittement « fait maison ».

3 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 C assez 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 qu'un choix technologique a sur l'application.

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), utilisé surtout sur des canaux de communication très fiables (i.e. des canaux avec une très faible probabilité de perte de données), et un format TS (Transport Stream), conçu pour la transmission des fichiers dans des liens très peu fiabilité.

En effet, 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.

- Sachant que le fichier video1.ts 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 »

- Déployez avec Mininet un réseau de test avec la commande « mn --switch lxbr --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.

  - Quelles sont les caractéristiques du réseau déployé ?

  3 hosts, un switch

  - 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, qui est à l'attente d'un 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)

  - Sur h2 exécutez la commande « nc -q0 -w0 -u 10.0.0.1 5000 < /chemin/vers/video1.mpg » afin d'envoyer le contenu de video1.mpg au host h1 (port 5000) avec la commande netcat.
  - 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.

- En reprenant le code des exemples fournis en cours pour l'écriture d'un programme UDP, complétez le programme UDP difftv-test.py avec les caractéristiques suivantes.
  - Votre programme ne fera pour l'instant qu'envoyer le mot « Hello » et finira tout de suite son exécution.
  - L'adresse et port du récepteur devront être donné en paramètre lors de l'exécution du programme. E.g. « python3 difftv-test.py 10.0.0.1 1234 ».
  - 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.
- Complétez difftv.py pour que le client envoie le contenu du fichier video1.mpg. Pour cela, répondez et faite ce qui est indiqué dans les points suivants.
  - 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. Notez que la transmission de plus de 40000 octets par default déclenche une erreur « too big packet »

638000/8 = 79750 octest/s. Vu que cette valeur est trop large pour un datagramme UDP, on la divise en 2 et on envoie 39875 tous le 0,5 secondes

  - Codez votre proposition faite pour le point ci-dessus. Voici un exemple pour ouvrir en mode lecture un fichier nommé « workfile.bin » et lire 1000 octets

```
try:
     global f
     f = None
     f = open("workfile.bin",'rb')
except IOError:
     print("Error while opening the file...")
     exit(-1)
data = f.read(1000)
```

  - Exécutez vlc à nouveau sur h1 et votre programme C sur h2. Vérifiez que la vidéo est jouée comme si vlc lisait le fichier en local (pas de fragments de film coupés, lecture fluide de la vidéo).
- Si la vitesse d'émission calculée précédemment 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 (VLC lit trop vite par rapport à la vitesse de transmission du programme). Cependant, la vidéo est jouée dans sa totalité.

- 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.

- Pour cette question, utilisez le réseau mininet créé avec la commande suivante : « mn --switch lxbr --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.
  - Exemple de commande à utiliser : python3 difftv.py 10.0.0.1 5000 & sleep 30; python3 difftv.py 10.0.0.3 5000
  - 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.

4 Les sockets TCP

Dans cet exercice, vous allez déployer un serveur de chat qui sera accessible depuis le réseau par le biais d'un serveur TCP.

- Déployez un réseau de test Mininet, avec la commande « mn --switch lxbr --topo single,3 », qui sera utilisé dans tous les tests effectués pour cette section.
- Pour commencer, en reprenant le code des exemples fournis en cours, écrivez un serveur TCP echoserver-tcp.py
  - De type single. Donc, lorsqu'un client arrive au serveur, le client suivant doit attendre que le premier finisse sa session.
  - 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.
  - Uniquement lorsque le client part, le serveur ferme la connexion avec le client. Puis, le serveur attend le prochain client.
  - Le port d'écoute du serveur devra être donné en paramètre lors de l'exécution du programme. E.g. « python3 echoserver-tcp.py 1234 »
- Une fois que votre serveur est prêt sur h1, démarrez-le, et sur h2 exécutez netcat en mode client. Vérifiez que votre serveur d'écho fonctionne correctement.
- A partir de votre programme echoserver-tcp.py, créez un faux serveur HTTP (serveur de type single toujours) appelé fake-http-svr-single.py
  - Le serveur doit lire 4096 caractères maximum une seule fois, puis répondre de la manière suivante.

- Le serveur envoie un message HTTP de succès, en-tête + corps de réponse. Le corps de la réponse sera une page HTML minimale, contenant un message « Hello World »
- Puis le serveur ferme la connexion avec le client.
- Testez votre programme avec curl. E.g. supposant que le serveur tourne sur le port 5000 : « curl -v 10.0.0.1:5000 --output - »