Université de Versailles Saint Quentin en Yvelines - UFR des sciences

Rapport de projet Réseaux L3

Programmation d'applications basées sur le modèle client/serveur

Réalisé par :
Mme Céline SAOUDI
Mme Amal BOUSHABA
Mme Lysa DERBAH
M. Pierre VERMEULEN
M. Remy CHIV

Version du
9 mai 2022

Table des matières

1 Introduction

Les sockets sont utilisés presque partout, et le but de ce projet est d'essayer de les comprendre. Pour cela dans ce projet on va programmer des applications basées sur le modèle client/serveur en utilisant les protocoles TCP et UDP qui sont les principaux protocoles de la couche transport du modèle TCP/IP.
Dans une première partie nous allons créer des sockets client /serveur en ajoutant les libraires nécessaires.
Puis dans la deuxième partie nous créerons des applications orientées connexion et sans connexion.

2 Présentation du projet

2.1 Partie TCP

2.1.1 Qu'est ce que TCP ?

TCP vient de l'anglais "Transmission Control Protocol" ou en français
"Protocole de Contrôle de Transmissions" est un protocole de transport en mode connecté, il permet la communication sûre entre un client et un serveur.
Ce protocole peut-être découpé en 3 phases :

- l'établissement de la connexion
- les transferts de données
- la fin de la connexion

Les caractéristiques principales de TCP :

- Il assure l'ordre des paquets
- TCP est orienté connexion : Il faut qu'une connexion soit faite avant de pouvoir transmettre des données
- TCP fournit la possibilité de détecter et corriger les erreur de transmissions.
- TCP fait du contrôle de flux : cela permet au client et au serveur d'envoyer et de consommer a une vitesse qui permet d'envoyer et de recevoir des informations de manière fiable.
- TCP fait du contrôle de congestion : cela permet d'éviter la cogestion (envoi de données trop rapidement et en quantité trop grande pour le réseau)

2.1.2 Côté client

- La première partie de toute programmation de socket consiste à créer le socket lui-même. Le socket client est créé avec un appel

```
sockfd = socket(AF_INET, SOCK_STREAM, 0)
```

  tel que sockfd est sockfd est le descripteur de fichier correspondant au socket.
  La fonction socket() renvoie un entier.
- Dans l'appel de socket, nous spécifions les paramètres suivants :
  Domaine : IPv4 (AF_INET)
  Type de socket : TCP (SOCK_STREAM)
  Protocole : IP (0)

— Ensuite dans une deuxième partie nous allons gérer et établir les connexions, sachant que la connexion à une adresse distante est créée avec l'appel

```
connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)).
```

Donc ici, nous spécifions l'adresse IP("127.0.0.1") et le port avec lequel nous allons nous connecter.
Si la connexion réussit, une valeur est renvoyée.
Sinon ça nous affiche "Connection Failed"

— On finit par envoyer des données avec la fonction send(sockfd, message, strlen(message), 0) et récupérer ces derniers avec un appel recv(sockfd, buffer, 2048, 0)

2.1.3 Côté serveur

— Comme pour la partie client, nous commençons par la création du socket.
Le socket client est crée avec un appel((sockfd =socket(AF INET, SOCK STREAM, 0) tel que sockfd est le descripteur de fichier correspondant au socket. La fonction socket() renvoie un entier.
Dans l'appel de socket, nous spécifions les paramètres suivants :
Domaine : IPv4 (AF INET)
Type de socket : TCP/UDP (SOCK STREAM)
Protocole : IP (0)

— Nous utilisons la fonction :

```
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)
```

Elle permet de rendre l'adresse réutilisable

— Permet de créer le lien entre la socket et l'adresse réseau :

```
bind(sockfd, (struct sockaddr *)&serv_address, addrlen_serv)
```

— Puis nous mettons en attente (d'une connexion) le serveur avec la fonction listen()

```
listen(sockfd, 10)
```

— Dans une boucle infinie, nous utilisons d'abord la fonction

```
cl = accept(sockfd, (struct sockaddr *)&client_address, (socklen_t
    *)&addrlen_client))
```

Elle extrait la première connexion de la file des connexions en attente de la socket sockfd à l'écoute, crée une nouvelle socket connectée, et renvoie un nouveau descripteur de fichier qui fait référence à ce socket qui est stocké dans le file descriptor cl_socket S'il réussit, accept() renvoie un entier non négatif si non, renvoie -1
Puis la fonction fork() qui permet de créer des processus fils identiques qui est stocké dans une variable childpid, ce qui permet donc de connecter plusieurs clients à un serveur

— Puis dans une autre boucle, le serveur reçoit un message du client avec la fonction

```
recv(cl_socket, buffer, sizeof(buffer), 0)
```

Le message est contenu dans le buffer. Si le message est 'exit' alors le client est déconnecté du serveur.
Puis le message du buffer est renvoyé au client avec la fonction

```
send(cl_socket, buffer, strlen(buffer), 0)
```

2.2 Partie UDP

2.2.1 Qu'est ce que UDP ?

Le User Datagram Protocol, abrégé en UDP, est un protocole non orienté connexion de la couche transport du modèle TCP/IP. La grande différence avec le protocole TCP est qu'il n'offre pas de contrôle d'erreurs.
Un paquet UDP est composé de 4 champs :

— Port Source : Port de l'appareil envoyant le paquet (peut être mit à zéro si cette appareil ne veut pas de réponse)
— Port Destination : Port de l'appareil recevant le paquet
— Longueur (Length) : Taille totale du paquet (limite fixé par le protocole IP utilisé pour transférer le paquet)
— Somme de contrôle (Checksum) : La somme de contrôle permet au receveur de vérifier l'intégrité du paquet entier (entête + données)

Caractéristiques d'UDP :

— Il est très simple, ce qui rend son utilisation intéressante pour construire d'autres protocoles au dessus de UDP.
— Il est orienté transaction, ce qui est utile pour les protocole simple de type requête-réponse.
— En cas de paquet perdu, il n'y a pas de délai (particulièrement intéressant pour la VoIP ou les jeux en ligne par exemple)
— Il est dit sans état, très utile si le nombre de client est très grand.
— Il supporte multicast(transmission de données d'un émetteur vers de multiples récepteurs), utile dans certains protocole pour procéder à la détection d'appareil ou le partage d'informations communes(l'heure avec PTP par exemple)
— Il fournit des datagrammes ce qui permet de créer des protocole ou on doit encapsuler des paquets(IP tunneling par exemple)

2.2.2 Côté client

— Tout d'abord on crée le socket avec l'appel :

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0)
```

— Nous spécifions les paramètres suivants :
AF_INET pour spécifier qu'il faut utiliser IPv4

Type de socket UDP : SOCK_DGRAM
Protocole : 0

— Ensuite on remplit la structure contenant les informations pour notre connexion :

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

Ici on précise le protocole utilisé, le port et avec INADDR_ANY on écoute sur toutes les interfaces.

— Ici on envoie le message stocké dans la variable message :

```
sendto(sockfd, message, strlen(message), 0, (struct serv_addr *)&serv_addr, slen)
```

On passe en argument de cette fonction le descripteur de fichier correspondant au socket, la taille de notre message et la structure contenant les informations pour la connexion.

— Ici on reçoit la réponse dans la variable buffer.

```
recvfrom(sockfd, buffer, 2048, 0, (struct serv_addr *)&serv_addr, &slen)
```

Comme l'envoi, on passe les même arguments sauf qu'ici on précise la taille maximale du message a recevoir et non la taille exacte de celui-ci

— On ferme le socket

```
close(sockfd)
```

2.2.3 Côté serveur

— Comme la première partie de la partie TCP, ou le client qui utilise UDP, on commence par créer un socket qui renvoie un descripteur de socket. Ce dernier est créé avec un appel

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0)
```

Tel que :
AF_INET spécifie le domaine de communication
SOCK_DGRAM le type de socket a créer
0 signifie utiliser le protocole par défaut pour la famille d'adresses.

— Le code coté serveur conserve les informations liées a l'adresse du serveur et du client dans la structure serv_addr (qui est un struct de type sockaddr_in)

— Ensuite on initialise l'adresse du serveur par le PORT et L'IP tel que :

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(PORT);
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

— On lie le descripteur de socket a l'adresse de serveur :

```
bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)
```

— Dans cette partie là contrairement a TCP, le coté serveur n'attend pas qu'un client se connecte et donc par conséquent il ne reçoit pas l'adresse du client avant d'envoyer et de recevoir des données.
Pour cela le serveur reçoit des informations sur le client lorsqu'il reçoit des données avec la fonction suivante :

```
recvfrom(sockfd, buffer, 2048 , 0, (struct sockaddr *) &serv_addr, &slen)
```

— Les informations du client stockées dans la variable de type sockaddr_in seront utilisées pour envoyer des données au même client avec la fonction :

```
sendto(sockfd, buffer, recv_len, 0, (struct sockaddr *) &serv\_addr, slen)
```

— On finit par fermer fermer le socket afin de mettre fin a la communication

```
close(sockfd)
```

3 Comment exécuter et tester le code

Pour compiler le code :

```
make // compile le code du client et du serveur
make server
make client
```

Pour tester le code exécutez :

```
./client protocole
./server protocole
// avec protocole => UDP ou TCP
// alternative :
make st // serveur TCP
make ct // client TCP
make su // serveur UDP
make cu // client UDP
```

4 Conclusion

Pour résumer TCP et UDP sont deux protocoles qui nous permettent de transférer des données sur la couche transport du modèle TCP/IP. Compte tenu de leurs différences, ces deux protocoles ne sont pas utilisés dans les mêmes situations.
La différence clé entre ces deux protocoles est la complexité. TCP est bien plus complet que UDP,

c'est pour cela qu'il est plus lent. La différence clé en terme de fonctionnalités est que UDP est un protocole plus simple et rapide mais non fiable car ce dernier peut juste détecter s'il y a une erreur, et abandonne les paquets corrompus.
Or TCP vérifie s'il y a une erreur pendant la transmission mais il est également capable de réparer ces erreurs et il a également un système d'acquittement qui permet de confirmer a l'envoyeur la bonne réception des données.

5 Sources

— https://en.wikipedia.org/wiki/User_Datagram_Protocol
— https://fr.wikipedia.org/wiki/User_Datagram_Protocol
— https://web.maths.unsw.edu.au/~lafaye/CCM/internet/tcp.htm
— https://web.maths.unsw.edu.au/~lafaye/CCM/internet/udp.htm