Base de la Conception Logicielle
TP04-05

DE MEIRA LIMA Otávio
CHAHIDI Hamza

1. Introduction

Les deux dernières séances de travaux pratiques se sont concentrées sur la communication de machine à machine sur le Web et le multitâche parallèle.

2. Protocole UDP

Le protocole de datagramme utilisateur (UDP) est un protocole de communication principalement utilisé pour établir des connexions à faible latence et tolérantes aux pertes entre les applications sur Internet. UDP accélère les transmissions en permettant le transfert des données avant qu'un accord ne soit fourni par la partie destinataire. Par conséquent, UDP est avantageux dans les communications sensibles au temps, y compris la voix sur IP (VoIP), la recherche de systèmes de noms de domaine (DNS) et la lecture vidéo ou audio. Comme il existe une certaine tolérance à la perte de données lors de la transmission, ce protocole est une alternative efficace mais peu fiable pour la transmission de données sensibles. Lorsque des données sont perdues sur le réseau, il n'y a aucune vérification qu'elles ont été perdues pour qu'une tentative de retransmission se produise.

Le langage Ada fournit des bibliothèques auxiliaires pour établir cette communication. La bibliothèque suggérée pour l'implémentation de ce travail est GNAT.Sockets. Mais d'abord, il est également nécessaire de comprendre le concept de « sockets » avant de commencer la mise en œuvre. Un socket est un point d'extrémité d'une liaison de communication bidirectionnelle entre deux programmes s'exécutant sur le réseau. Un socket est lié à une adresse et à un numéro de port afin que la couche de transport puisse identifier l'application à laquelle les données seront envoyées.

Il existe deux implémentations, le client et le serveur. Le serveur crée le socket, en liant son adresse et son port, tandis que le client essaie de s'y connecter. Au démarrage du serveur, il est possible de constater qu'il attend une connexion sur le port 7777. Une fois la connexion effective, il entre dans une boucle infinie où il reçoit les messages du client et les imprime sur le terminal.

Le client, quant à lui, lorsqu'il est exécuté, crée le socket, s'y connecte en fonction de l'adresse et du port du serveur, et lui envoie des messages d'accueil juste après. La valeur

imprimée X:Y représente respectivement l'adresse machine et le port correspondant à l'exécution du programme, séparés par un point-virgule. L'adresse, à cette occasion, est représentée par la suite de chiffres « 127.0.0.1 ». Cette adresse représente la machine locale, de l'anglais « localhost ». Comme le client et le serveur communiqueront via la même machine, il n'est pas nécessaire de communiquer via le réseau. Si une communication sur le réseau est nécessaire, les 4 chiffres de l'adresse vont de 0 à 255. L'adresse est représentée par 32 bits. Divisé en 4 positions, un nombre binaire de 8 chiffres maximum ne peut atteindre que la valeur 255.

La prochaine étape du travail est multilingue. Un programme Python a été fourni qui génère constamment des nombres aléatoires entre 0 et 20 et les imprime sur un graphique. Le travail effectué ensuite a été de créer les sockets en langage Python et de recevoir les nombres aléatoires générés par le client en Ada. De cette façon, la responsabilité de générer les nombres aléatoires devient la responsabilité du client dans Ada. Le serveur en Python commence à recevoir les informations par l'adresse Web et son port respectif. La génération du graphe s'effectue de la même manière que précédemment, en ne modifiant que la réception des données.

Le code client UDP dans Ada pour envoyer des nombres aléatoires au serveur Python est décrit ci-dessous. La génération des nombres aléatoires s'est faite à l'aide d'une fonction rand déjà présente dans le code initial de l'ouvrage:

```
loop
    Rand := Random;
    Send (Socket, Addr, Rand'Image, Last);
    Put_Line (Item => "Envoye : " & Integer'Image (Rand));
    delay (1.0);
end loop;
```

Ensuite, le code Python pour créer un socket avec l'adresse de la machine locale :

```
AdresseEcoute = ("127.0.0.1", 7777)
datagramSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
datagramSocket.bind(AdresseEcoute)
```

3. Protocole TCP

Le protocole de contrôle de transmission (en abrégé TCP) est l'un des protocoles de communication de la couche de transport du réseau informatique du modèle OSI, qui prend en charge le réseau mondial Internet, en vérifiant que les données sont envoyées dans l'ordre correct et sans erreur via le réseau. . Il est complété par le protocole Internet, communément appelé TCP/IP. Le protocole est fiable, fournit des données dans le bon ordre et dispose d'un contrôle d'erreur sur les paquets de données. Cela permet aux applications de s'assurer que les données sont livrées correctement. TCP est orienté connexion et une

connexion entre le client et le serveur est établie avant que les données puissent être envoyées. Le serveur doit écouter (passivement s'ouvrir) les demandes de connexion des clients avant qu'une connexion ne soit établie. L'établissement de la connexion se fait avec "3 poignées de main":

1. Le client envoie un signal au serveur pour se connecter.
2. Le serveur répond par un accusé de réception. Habituellement, un numéro de séquence est envoyé pour assurer une connexion correcte
3. Le client envoie un autre signal d'accusé de réception au serveur, assurant que le numéro reçu était correct et que la connexion est établie.

De cette façon, il est garanti que le client et le serveur sont connectés et qu'une connexion fiable a été établie. Si un paquet de données est perdu pendant la transmission, le client/serveur sait que les données n'ont pas atteint l'autre partie car il n'a pas reçu de signal d'accusé de réception après avoir reçu le message.

Un exemple de communication via le protocole TCP sont les chats en ligne. Il est intéressant que l'expéditeur du message sache que son destinataire a reçu le message. Et c'est sur cet exemple que s'applique la seconde partie de l'ouvrage. Dans la première version du chat, deux clients se connectent au serveur. La première modification à apporter est de permettre au client de dire son nom avant de pouvoir envoyer des messages à l'autre utilisateur. Cela a été possible en faisant le changement de code suivant :

```
Put ("Donner votre nom : ");
Get_Line (nom);
Send (Socket, To_String (nom) & ASCII.LF, Last);
```

Lorsque vous exécutez le programme, vous pouvez voir qu'il y a un problème majeur avec le code client : il ne peut recevoir des messages que s'il envoie un message en premier. La cause de ce problème est l'exécution séquentielle de l'envoi puis de la réception du message dans une boucle infinie. Comme le code s'exécute de manière séquentielle, le client ne peut recevoir un message que s'il en envoie également un autre. La solution passe par un concept bien connu lors de la programmation des sockets : le **multitâche**. La lecture de code n'est plus séquentielle et devient parallèle, permettant l'exécution simultanée de deux méthodes. Dans ce cas, les méthodes qui doivent s'exécuter en parallèle sont l'envoi et la réception de messages.

Deux tâches ont été implémentées pour exécuter les tâches décrites précédemment en parallèle. Ils sont déclarés dans une clause « declare ». Afin de ne pas gâcher l'impression et l'envoi des messages, chaque message lu depuis la borne ou reçu de l'autre client est envoyé dans une boîte de réception Boit à Lettre. Les messages sont lus dans une boucle et sont soit envoyés à l'expéditeur, soit imprimés sur le terminal avec un intervalle de 0,5 seconde entre chaque itération.

Le chat multiclient utilisant le protocole TCP fonctionne. Cependant, l'impression des messages dans le terminal de chaque client n'est pas bonne. Pour améliorer la visibilité du chat, une bibliothèque d'assistance appelée ANSI_Console a été fournie. Il fournit des méthodes pour imprimer sur une certaine ligne terminale et également pour effacer des lignes. Cela a été très utile car cela a aidé à rendre le chat plus visible. Voici un exemple :

La première ligne sert à écrire le message. Les lignes suivantes sont l'historique des messages déjà envoyés.