Objectifs : Comprendre les principales différences entre TCP et UDP. Pour cela, vous allez réaliser des applications client-serveur en TCP et des équivalentes en UDP et observer le comportement via une série de scénarios à mettre en place.

Après avoir tester vos programmes sur une seule machine, choisissez un binôme avec qui vous aller permettre à vos programmes d'utiliser le réseau internet pour communiquer (contexte idéal). Avant, pour celles et ceux qui ne l'ont pas encore fait, créez une redirection de port sur votre box (pour la freebox par exemple, la procédure est décrite ici : `https://www.cartelectronic.fr/blog/?p=2167`).

Notations et rappel :
Le protocole de transport **TCP** permet de réaliser des communications en mode connecté. Un message envoyé en **TCP** est transféré/acheminé sous forme de flux d'octets. Enfin, **TCP** gère la duplication et la remise dans l'ordre des paquets à leur réception.
Le protocole de transport **UDP** permet de réaliser des communications en mode non connecté. Un message envoyé en **UDP** est transféré/acheminé en un seul paquet **UDP** et est indépendant de tout autre paquet. Enfin, **UDP** ne gère ni la duplication, ni la remise dans l'ordre des paquets à leur réception.

Téléchargez les fichiers fournis depuis Moodle. Ils contiennent des squelettes vous donnant des indications du comportement attendu.

1 Vais-je recevoir tous les messages et dans le bon ordre ?

Etape 1 : client-serveur TCP

Écrire deux programmes :

- un programme serveur qui accepte la demande de connexion d'un seul client, reçoit en boucle (infinie) des messages de type "long int". A chaque itération, le serveur compare la valeur reçue avec celle reçue à l'itération précédente : si elle est inférieure à la précédente, alors le serveur affiche un message le notifiant. A la fin de chaque itération, le serveur affiche le nombre total d'octets reçus et le nombre d'appels réussis de la fonction recv(..) depuis le premier message reçu. Le serveur termine lorsque le client se déconnecte.
- un programme client qui envoie au serveur des entiers (long int) allant de 1 à N, où N est un paramètre du programme. Chaque entier représente un message à part entière.

Le test effectué par le serveur, permet d'identifier une éventuelle arrivée de messages dans le désordre (est ce possible en TCP ?)
Dans cet exercice (et les suivants), vous utiliserez bien évidement les fonctions sendTCP(...) et recvTCP(...) du précédent TP. L'adresse de la socket du serveur est à passer en paramètre du programme client. Le numéro de port du serveur est à passer en paramètre de ce dernier.

Tester vos programmes avant de passer à la suite.

Etape 2 : et en UDP ?

Écrire un programme client et un programme serveur qui ont le même rôle qu'à l'étape 1 mais en utilisant les protocoles UDP (passage en mode datagramme, absence de socket d'écoute des demandes de connexion, absence de demande de connexion, remplacer les appels de sendTCP(..) par sendto(...) et recvTCP(...) par recvfrom(...)). Vous pouvez utiliser ici le mode asymétrique.

Tester vos programmes avant de passer à la suite.

Etape 3 : j'exécute, j'observe, je comprends et je compare

Dans cette partie, il est question, pour chaque scénario qui suit, d'exécuter vos programmes utilisant les protocoles TCP et UDP, d'observer ce qui se produit et d'expliquer la différence de comportement. Vous pouvez faire des tests en local (sur la même machine) puis en utilisant un réseau pour comprendre les éventuelles différences de comportement liées au réseau.

1. Exécuter vos programmes en faisant varier la valeur de N : 1, 10, 200, 1000, 50000, 500000. Que se passe-il pour chaque valeur de N et pour chaque protocole (TCP et UDP) ? Le nombre total d'octets reçus, le nombre d'appels à recv(...) ou recvfrom(...) et d'autres traces sont à prendre en compte.
   — Y a-t-il perte de messages ?

— Des messages arrivent-ils dans le désordre ?
— Y a-t-il des erreurs ? Si oui, les processus en sont-il notifiés (avez vous l'information) ?
— Y a-t-il un blocage/attente ?
— Le client s'exécute-il entièrement et sans erreur ?
— Le serveur s'exécute-il sans erreur ?

2. Modifier les programmes serveurs pour suspendre l'exécution après la réception du premier message et avant la boucle de réception de la suite (avec une saisie au clavier). Lancer vos programmes serveurs puis clients. Qu'observez vous pendant que l'exécution des serveurs est suspendue ?
   — Y a-t-il perte de messages ?
   — Des messages arrivent-ils dans le désordre ?
   — Y a-t-il des erreurs ? Si oui, les processus en sont-il notifiés (avez vous l'information) ?
   — Y a-t-il un blocage/attente ?
   — Le client s'exécute-il entièrement et sans erreur ?
   — Le serveur s'exécute-il sans erreur ?

   Et en poursuivant l'exécution des serveurs ?
3. Pour chaque protocole, lancer le client puis le serveur (oui, dans cet ordre). Que se passe t-il et pour quelle raison ?

2 La couche transport préserve t-elle les limites d'un message envoyé ?

Dans le programme serveur (TCP ou UDP) de la section précédente, au lieu de recevoir les messages, un après l'autre, il est possible d'effectuer la réception de la suite des messages par blocs d'octets. Le plus important est de recevoir tous les messages et de les interpréter correctement.

Cette partie se focalise sur ce qui se produit en effectuant des réceptions par blocs d'octets (sans interpréter le contenu des messages). La question qui se pose en particulier : la couche transport sera t-elle capable de délimiter chaque message (de type long int) envoyé par le client ?

Modifier le programme serveur (pour TCP et UDP) pour qu'il reçoive en boucle (infinie) des données de longueur K octets (taille d'un bloc), où K est un paramètre du programme. Le fichier fourni, vous donne un squelette répondant à cette spécification.

Aucune modification n'est à faire coté client.

Exécuter vos programmes avec N égale à 1, 2, 5 puis 1000 et pour chaque valeur de N, faire des tests avec K égale à 5 octets, 10 octets puis 20 octets.

Pour chaque test, comparer le nombre total d'octets reçus et le nombre total d'appels de la fonction recv(...) avec le nombre total d'octets envoyés et le nombre total d'appels de la fonction send ...) coté client. Répondez aux questions suivantes :

— Si un seul message est envoyé par le client, y a-il perte de données coté serveur ? Si oui, pourquoi ?
— Pour plusieurs messages envoyés, le serveur les reçoit-il entièrement ?
— Pour plusieurs messages envoyés, le serveur va t-il les extraire un après l'autre du buffer de réception, même s'il demande à recevoir plusieurs en un seul ?

3 Mode connecté Vs Mode non connecté et traitement de plusieurs clients

Dans les deux sections précédentes, le processus serveur TCP s'arrête à la fin du traitement du client, mais le processus serveur UDP se met en attente d'un nouveau message.

Etape 1 : traitement séquentiel de plusieurs clients

Si à la fin du traitement du premier client UDP, vous lancez un second client puis un troisième etc. vous allez constater que le serveur est capable de recevoir (via la même socket), des messages en provenance de différents clients (mais surtout de différentes sockets) : la communication se fait en mode non connecté.

En TCP, pour traiter plusieurs clients, le serveur doit créer une socket par client (lors de l'appel de la fonction accept(...)), cette socket sera connectée à celle du client et aucun autre client ne pourra utiliser la socket dédiée à un autre pour communiquer avec le serveur : la communication se fait en mode connecté.

Modifier le programme du serveur TCP pour qu'il traite en séquentiel plusieurs clients. Le serveur dans ce cas sera itératif. Remarque : le serveur doit pouvoir continuer son exécution même si une erreur se produit lors ce la communication avec un client (une erreur ou un arrêt d'un client, ne doit pas affecter le traitement des autres clients). Il est nécessaire de prendre les bonnes décisions lors du traitement des retours de fonctions utilisées pour communiquer avec un client.

Tester vos programmes TCP avec plusieurs clients.

Etape 2 : traitement simultanées (sans concurrence) de plusieurs clients

Si vous lancez le serveur UDP (sans aucune modification), si vous choisissez une valeur de N suffisamment grande et lancez un premier client et un deuxième sans attendre la fin du premier, vous constaterez que le serveur est capable de prendre en compte plusieurs clients en même temps.

Pour reproduire un comportement similaire en TCP, l'idée ici est de mettre en oeuvre le multiplexage des entrées sorties. En partant de la version itérative de l'étape 1, modifier le serveur pour traiter plusieurs clients en même temps en utilisant donc le multiplexage.

Tester vos programmes TCP avec plusieurs clients. Avec cette solution, un client ne sera pas obligé d'attendre la fin d'un autre client avant d'être pris en compte. L'ordre d'arrivée des messages en provenance des différents clients déterminera l'ordre dans lequel ils sont traités et le progression de chacun.

Pour terminer, si pour chaque message reçu, le serveur doit faire un traitement long (de quelques minutes), le multiplexage serait-il adapté ? Pour quelles raison cette solution convient à l'exercice actuel ?