Communication avec TCP

Dans ce TP, tout comme le précédent, il faut écrire deux programmes ; un émetteur et un récepteur, pour s'échanger un fichier, mais cette fois en utilisant le protocole TCP.

Classes à utiliser

- `ServerSocket` : Cette classe représente le point d'accès d'un serveur au réseau. Le programme récepteur doit créer un `ServerSocket` avec un entier représentant le numéro de port de réception (entre 1024 et 65535). La méthode `accept()` met le programme en attente d'une connexion TCP demandée par un autre programme.
- `Socket` : Cette classe représente le point d'accès d'un client au réseau vers un serveur donné. Le programme émetteur doit créer un `Socket` avec l'adresse IP de la machine où s'exécute le programme serveur ("localhost" par exemple) et le numéro de port de réception. Attention, un objet de la classe `Socket` est également retourné par la méthode `accept()` du récepteur lorque la demande de connexion TCP à été faite.

Description du fonctionnement

Le récepteur doit créer un `ServerSocket` et attendre une demande de connexion par l'émetteur. Ce dernier la réalise en créant un `Socket` qui va entrainer la création dans le programme récepteur de l'autre `Socket`. Les deux `Socket` sont maintenant reliés par l'intermédiaire du protocole TCP. Tout ce qui sera envoyé sur un des deux `Socket` sera reçu par l'autre.

Pour émettre et recevoir des données d'un `Socket`, il faut utiliser les flux entrant et sortant des `Socket` (méthodes `getInputStream()` et `getOutputStream()`). Les flux permettent d'envoyer et de recevoir des `byte[]`) avec les méthodes également appelées `read(byte[])` et `write(byte[])`. Le programme émetteur devra donc lire un fichier avec un `read(byte[])` puis écrire avec un `write(byte[])` sur le `Socket`. De son coté, le récepteur devra lire depuis le `Socket` avec un `read(byte[])` et écrire dans le fichier avec un `write(byte[])`.

A noter que l'émission d'un `byte[]` n'entraine pas forcément la réception d'un `byte[]` de la même taille, mais plutôt la réception de plusieurs `byte[]` de plus petite taille. En effet, comme dans la plupart des cas le réseau local est de l'Ethernet (avec ou sans fil) dont la taille maximale des trames est d'environ 1500 octets [1], TCP va découper à l'émission le `byte[]` pour générer plusieurs segments TCP de cette taille. Dans le TP précédent, les datagrammes UDP, qui avaient une taille supérieure à 1500, ont été découpés par IP et son mécanisme de fragmentation. UDP n'a pas eu "conscience" de ce fait.

Des problèmes de synchronisation peuvent se poser entre l'émetteur et le récepteur :
- Une fois que le récepteur a reçu la demande de connexion (passage de méthode `accept()`) et a récupéré le `Socket` lui permettant de communiquer avec l'émetteur, il faut attendre que les premières données soient arrivées. Pour cela la méthode `available()` de la classe

1. Si émetteur et récepteur sont sur la même machine, la taille maximale est généralement plus grande

`InputStream` peut être utilisée. Le résultat de cette méthode retourne le nombre d'octets pouvant être lu ; si le résultat est 0, c'est que le flux n'est pas encore prêt, ou que le transfert est terminé. Le résultat de la méthode `available()` doit aussi servir à diensionner correctement le `byte[]` de réception.

- Durant le transfert, une condition d'arrêt possible est la surveillance de la présence de données dans le flux (la méthode `read(byte[])` retourne ne nombre d'octets lus).
- Lorsque le récepteur reçoit plusieurs segments (la plupart du temps), il est souvent nécessaire d'ajouter un arrêt d'une miliseconde (`Thread.sleep(1)`) [2] dans la boucle de réception. Le récepteur lisant trop rapidement par rapport à l'émetteur, il ne trouve plus rien à lire et quitte la boucle de réception. Comme l'émetteur continue d'émettre, il se retrouve en situation d'erreur.

2. Pour des transferts de fichiers plus important, cet arrêt doit être plus long (5 par ex.)