EXAMEN 1ère SESSION | PROGRAMMATION RÉSEAU | 19/05/2021 09^{00}-11^{00}

Les seuls documents autorisés sont les documents « sur papier » ;
à l'exclusion de la copie ou des brouillons des voisins.

Consignes

- Lire **très attentivement** tout le sujet.
- Il est tout à fait possible de répondre à une question sans avoir répondu à toutes les questions précédentes et aussi d'utiliser les méthodes des questions précédentes même si on n'a pas donné leur code.
- Quand des consignes manquent de précision (par exemple le type de retour des méthodes), libre à vous de proposer la solution qui vous semble le mieux.
- On supposera que tous les messages circulant sur le réseau sont corrects (on ne vous demande pas de traiter les cas où un message ne respecte pas la spécification).
- Les messages circulant sont signalés entre crochets `[...]` et **les crochets ne font pas partie du message**.
- On supposera qu'un message envoyé dans un paquet **UDP** est reçu dans un paquet **UDP** (c'est-à-dire qu'il n'est pas coupé).
- On ne demande pas de traiter les exceptions et on ne demande pas d'écrire les `import` d'API.
- À la fin du sujet, on vous rappelle quelques méthodes `Java` pouvant être utiles.

Contexte

On désire développer une architecture sous forme d'abre binaire où chaque entité du réseau a au plus deux fils et exactement un père sauf l'entité racine qui elle n'a pas de père. À tout moment des entités peuvent s'ajouter au réseau en faisant la demande aux noeuds n'ayant pas deux fils déjà attribués. On peut de plus envoyer un message à un noeud qui le transmet alors à ses fils qui eux-mêmes le transmettront à leurs fils, etc.
Nous ne traiterons pas dans ce sujet les demandes de déconnexion du réseau. Les noeuds écoutent les messages qui viennent de leur père sur le port **UDP** `1234`. Les noeuds communiquent sur le port **TCP** `5678` pour la communication venant de l'extérieur de l'arbre. Un noeud peut également envoyer des messages vers l'extérieur en multi-diffusion en utilisant l'adresse de multi-diffusion `233.222.222.1` et le port `4242`.

Nous détaillons maintenant les différentes fonctionnalités de notre application réseau.

La construction de l'arbre

Lorsqu'une entité souhaite devenir le fils d'un noeud `n` de l'arbre, elle se connecte en **TCP** sur le port `5678` de ce noeud et elle envoie un message `[CONNECT\n]`. Si le noeud `n` a déjà deux fils (c'est à dire qu'il a déjà accepté deux demandes d'insertion), il répond par le message `[FULL\n]`, sinon il répond `[OKSON\n]` pour signaler que cette entité est maintenant enregistrée comme l'un de ses deux fils. Après avoir reçu `[FULL\n]` ou `[OKSON\n]`, l'entité ayant fait la requête ferme la connexion et est prête à recevoir les messages de son père sur son port **UDP** (si l'insertion a été acceptée).
Une entité extérieure peut demander également l'adresse d'un noeud ayant un fils libre. Pour cela, elle se connecte à la racine de l'arbre sur le port **TCP** et elle envoie le message `[LEAF\n]` et se déconnecte. Si la racince a un fils libre, elle attend une seconde (avec un *sleep*) et elle envoie par multi-diffusion (à l'IP `233.222.222.1` et au port `4242`), le message `[L ip]` de taille au plus 21 octets où `ip` est une chaîne de caractères correspondant à son adresse IP. Si la racine n'a pas de fils libre, après un délai d'une seconde, elle envoie par **UDP** à ses deux fils le message `[FREE]`, qui suivent alors le même comportement, selon si ils ont ou n'ont pas de fils libre, dans le premier cas ils multi-diffusent un message `[L ip]` (avec leur adresse IP) et dans le deuxième cas, ils envoient le message `[FREE]` à leurs fils, etc.

La diffusion de messages dans l'arbre

Pour diffuser un message dans l'arbre, un client extérieur peut se connecter au port **TCP** d'un noeud et il lui envoie un message de la forme `[M mess\n]` où `mess` est une chaîne de caractères à transmettre (qui ne contient pas de caractères `\n` et qui fait au plus 500 caractères). À la réception d'un tel message, le noeud de l'arbre le traite puis ferme la connexion **TCP**. Ensuite il transmet sur le port **UDP** `1234` de ses fils (si il en a) le message `[T mess]`. Lorsqu'un noeud de l'arbre reçoit sur son port **UDP** `1234` un message de la forme `[T mess]`. Il envoie à ces fils (si il en a) sur leur port **UDP** le message `[T mess]`.

Questions

Programmation en C

On commence par supposer qu'un tel arbre existe déjà et que sa racine se trouve sur la machine `machine1.progreseaux.fr` qui est accessible depuis votre machine.

1. Écrire en C un client qui transmet à l'arbre le message `Hello World!`.
2. Écrire en C une fonction `requestLeaf` qui se connecte en **TCP** à la racine de l'arbre et envoie le message `[LEAF\n]` et se déconnecte.
3. Écrire en C une fonction `getLeaf` qui attend un message `[L ip]` émis par multi-diffusion sur l'IP `233.222.222.1` et sur le port `4242` et, quand elle l'a reçu, renvoie la chaîne de caractères contenant l'adresse IP `ip`.
4. Écrire une fonction `nextLeaf` qui se connecte en **TCP** à une adresse IP (donnée sous-forme de chaîne de caractères en arguments de la fonction) sur le port 5678 et envoie le message `[CONNECT\n]`. Elle attend ensuite une réponse et renvoie l'entier 0 si elle reçoit un message `[OKSON\n]` et -1 si elle reçoit une autre réponse. Ensuite la fonction se déconnecte.
5. Écrire un programme C (i.e. une fonction `main`) qui se rajoute comme feuille de l'arbre dont la racine se trouve sur la machine `machine1.progreseaux.fr` et affiche un message sur le terminal de la forme `I am leaf` quand cela est fait. *Attention :* certains programmes peuvent faire la même demande en parallèle et donc il n'est pas garanti, que entre le moment où l'on demande un noeud de l'arbre avec un fils libre (avec le message `[LEAF\n]`) et le moment où l'on essaie de se connecter à ce noeud le noeud renvoie `[OKSON\n]`. Entre temps, les noeuds avec un fils de libre peuvent avoir changé. Il faut donc traiter ce cas quitte à demander plusieurs fois de connaître les noeuds avec un fils de libre. En revanche, on ne vous demande pas de traiter les messages **UDP** qui arrivent de l'arbre une fois le programme ajouté comme feuille.

Programmation en Java

Nous créons d'abord la classe `Root` qui correspond au noeud racine d'un arbre. Cette classe a un champ `myIp` pour stocker son adresse IP, deux champs `ipSon1` et `ipSon2` correspondant aux IP des deux fils du noeud. Nous rappelons qu'un noeud racine ne reçoit pas de messages **UDP** car il n'a pas de père.

```
class Root{
  public String myIp;
  public String ipSon1;
  public String ipSon2;

  //A completer
}
```

6. Écrire le constructeur de la classe `Root` qui remplit le champ `myIp` avec l'adresse IP de la machine sur laquelle on exécute cette méthode et met les adresses IP des deux fils à `null`.

7. Écrire une méthode `getIP` qui prend en argument une socket **TCP** et renvoie la chaîne de caractères correspondant à l' l'IP de la machine connectée à l'autre bout de la socket.
8. Écrire une méthode `setSon` qui prend en argument une socket `TCP` et qui met l'adresse IP de la machine connectée à l'autre bout dans `ipSon1` ou `ipSon2` si l'un des deux vaut `null` et renvoie `true` si c'est le cas, et `false` si les deux champs `ipSon1` et `ipSon2` sont différents de `null`.
9. Écrire une méthode `transMess` qui prend comme arguments une chaîne de caractères `mess` et envoie en **UDP** le message `[T mess]` aux fils de cette racine (si elle en a).
10. Écrire une méthode `treatLeaf` qui ne prend pas d'argument et qui envoie en multidiffusion le message `[L ip]` sur l'IP et le port donné de multi-diffusion de l'arbre où `ip` est l'adresse IP du noeud si la racine a encore un fils libre et sinon elle envoie en **UDP** le message `[FREE]` aux fils de cette racine.
11. Écrire une méthode `treatTCPmess` qui prend en arguments une socket **TCP** et une chaîne de caractères contenant un message venant de la communication **TCP** et traite le message selon si il s'agit d'un message de la forme `[M mess\n]` ou `[CONNECT\n]` ou `[LEAF\n]`. Cette méthode ne renverra rien.

Nous allons maintenant créer la classe `TCPService` qui contiendra la méthode en charge de la communication **TCP** du noeud racine. Cette classe a deux champs, un champ `ro` contenant une référence vers l'objet `Root` associé au service et un champ `sock` contenant une référence vers la socket **TCP**.

```
class TCPService implements Runnable {
  public Root ro;
  public Socket sock;

  public TCPService(Root _ro,Socket _so){
   this.ro=_ro;
   this.sock=_so;
  }

  //A completer
}
```

12. Écrire la méthode `run` de la classe `TCPService`. Cette méthode est appelée après une connexion à la racine et elle est responsable de la communication **TCP** entre l'entité extérieure et le noeud racine.
13. Écrire une classe `ProgRoot` qui contient un programme principal (c'est-à-dire une méthode `main`) pour un noeud racine . Ce programme crée tout d'abord un objet `Root` et attend ensuite en boucle des connexions sur le port TCP et pour chaque connexion lance un thread pour gérer la communication.
14. Quel problème pourrait-il se passer si deux clients demandent à être successeur du noeud racine « en même temps » ? Comment éviter ce problème en modifiant votre code ?

Quelques fonctions de Java pouvant être utiles

— Méthodes de la classe `Socket`

```
InetAddress getInetAddress()
\\ Returns the address to which the socket is connected.
```

— Méthodes de la classe `InetAddress`

```
String getHostAddress()
//Returns the IP address string in textual presentation.
```

```
static InetAddress getLocalHost()
//Returns the address of the local host.
```

— Méthodes de la classe `InetSocketAddress`

```
public InetSocketAddress(String hostname, int port)
//hostname is either the name of the machine or the IP address
```

— Méthodes de la classe `String`

```
int length()

String substring(int beginIndex, int endIndex)
//Returns a new string that is a substring of this string.
// The substring begins at the specified beginIndex
// and extends to the character at index endIndex - 1.
// Thus the length of the substring is endIndex-beginIndex.
// The first index is 0.
```

— Méthodes de la classe `Integer`

```
Integer(String s)
/*Constructs a newly allocated Integer object that represents the int
value  indicated by the String parameter. */

int intValue()
```