EXAMEN 2ème SESSION — PROGRAMMATION RÉSEAU — 23/06/2015 08^{30}-11^{30}

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.
- À la fin du document, on rappelle quelques méthodes Java qui pourraient être utiles.
- 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).

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 entités feuilles. On peut de plus envoyer un message à un noeud qui le transmet alors à ses fils qui eux-même les transmettrons 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 s'ajouter à l'arbre, elle se connecte en **TCP** sur le port `5678` d'un noeud `n` de l'arbre 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 un noeud de l'arbre sur le port **TCP** et elle envoie le message `[LEAF\n]` et se déconnecte. Si le noeud correspondant a un fils libre, il attend une seconde (avec un *sleep*) et il envoie par multi-diffusion (à l'IP `233.222.222.1` et au port `4242`), le message `L ip` qui de taille au plus 21 octets où `ip` est une chaîne de caractères correspondant à son adresse IP. Si ce noeud n'a pas de fils libre, après un délai d'une seconde, il envoie par **UDP** à ces 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` 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 ferme la connexion TCP. Ensuite il envoie le message `[D num mess]` en multi-diffusion et il transmet sur le port UDP 1234 de ces fils (si il en a) le message `[T num mess]` où `num` est la chaîne de caractères égale à `0`.
Lorsqu'un noeud de l'arbre reçoit sur son port UDP 1234 un message de la forme `[T num mess]`. Il commence par envoyer `[D num2 mess]` en multi-diffusion et il envoie à ces fils (si il en a) sur leur port **UDP** le message `[T num2 mess]`. Dans ces deux cas `num2` est la chaîne de caractères correspondant à l'entier stocké dans la chaîne `num` augmenté de 1. Notons que grâce à la multi-diffusion, il est possible de suivre les messages circulant dans l'arbre.

Questions

Nous créons tout 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
}
```

1. 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`.
2. Écrire une méthode `getIP` qui prend en argument une socket **TCP** et renvoie la chaîne de caractères correspondant à son IP.
3. Écrire une méthode `setSon` qui prend en argument une socket `TCP` et qui met son adresse IP 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`.
4. Écrire une méthode `multidiffMess` qui prend comme arguments une chaîne de caractères `mess` et un entier `i` et envoie en multidiffusion le message `[D num mess]` sur l'IP et le port de multi-diffusion où `num` est la chaîne de caractères correspondant à l'entier `i`.
5. Écrire une méthode `transMess` qui prend comme arguments une chaîne de caractères `mess` et un entier `i` et envoie en **UDP** le message `[T num mess]` aux fils de cette racine (si elle en a) où `num` est la chaîne de caractères correspondant à l'entier `i`.
6. 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.
7. 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 renverra `true` si le message est de la forme `[M mess\n]` ou `[LEAF\n]` et lorsque le message est de la forme `[CONNECT\n]`, cette méthode renverra `true` si elle a ajouté un nouveau fils et `false` sinon.

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
}
```

8. 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.
9. 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.

On va maintenant procéder au code des noeuds internes de l'arbre. Pour se faire on définit une classe `Node` qui hérite de `Root`,

```
class Node extends Root{
  //A completer
}
```

10. Écrire une méthode `connectTree` qui prend en argument une chaîne de caractères correspondant au nom ou à l'IP d'une machine sur laquelle s'exécute une entité noeud ou racine, et demande à se connecter à l'arbre. Cette méthode renvoie `true` si le noeud concerné accepte de prendre ce nouveau noeud comme fils et `false` sinon.
11. Écrire une méthode `treatUDPmess` qui prend comme argument une chaîne de caractères contenant un message venant de la communication **UDP** et traite le message selon si il s'agit d'un message de la forme `[T num mess]` ou `[FREE]`.

Nous allons maintenant créer la classe `UDPService` qui contiendra la méthode en charge de la communication **UDP** du noeud . Cette classe a un unique champ `no` contenant une référence vers l'objet `Node` associé au service.

```
class UDPService implements Runnable {
  public Node no;

  public UDPService(Node _no){
   this.no=_no;
  }

  //A completer
}
```

12. Écrire la méthode `run` de la classe `NodeService`. Cette méthode est appelée après la connexion du noeud à un arbre et elle gère la communication UDP, en attendant en boucle les messages et en les traitant.

13. Écrire une classe `ProgNode` qui contient un programme principal (c'est-à-dire une méthode `main`) pour un noeud. Ce programme crée tout d'abord un objet `Node`, puis il connecte ce noeud au noeud d'un abre situé sur la machine `machine.progreseaux.fr` (on suppose ici que ce dernier noeud a au moins un fils libre), ensuite il crée un thread pour gérer les communications UDP et finalement attend en boucle des connexions sur le port TCP et lance pour chaque connexion un thread pour gérer la communication.

On va maintenant écrire différents clients en C.

14. Écrire en C un client qui veut transmettre à tout l'arbre le message `Hello World!` en supposant que la racine de l'arbre tourne sur la machine `machine2.progreseaux.fr`.
15. Écrire un client en C qui s'adresse au même arbre et affiche à l'écran l'adresse IP d'un noeud libre de l'arbre.

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()
```