EXAMEN PROGRAMMATION RÉSEAU 25/6/2018 12^{00}-15^{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.
- Pour répondre aux questions, pensez à utiliser le code des réponses précédentes.
- 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).
- 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

Contexte

Le but de cet examen est de programmer des entités mettant en place un service de diffusion de messages de 140 caractères. Ces messages seront envoyés par des diffuseurs. Des utilisateurs pourront transmettre au diffuseurs des messages à envoyer.
Les diffuseurs enverront en multi-diffusion des messages en continu, ils pourront aussi envoyer des messages d'utilisateurs et renvoyer les derniers messages à un utilisateur sur demande.
Les clients devront être capable d'écouter les messages des diffuseurs, de leur envoyer des messages particuliers et de leur demander une liste des derniers messages.

Communication avec les diffuseurs

Les caractéristiques d'un diffuseur sont les suivantes :

- Un port TCP pour recevoir les messages d'utilisateurs (inférieur à 9999)
- Une adresse IPv4 de multi-diffusion
- Un port de multi-diffusion (inférieur à 9999)

Il est important de comprendre qu'un diffuseur gérera sa liste de messages à diffuser. Ces messages correspondent à ceux que les utilisateurs ont demandé à transmettre. Commençons tout d'abord par décrire comment un diffuseur diffuse ses messages. Il envoie sur son adresse et port de multi-diffusion des messages de la forme `[DIFF␣num-mess␣id␣mess]` où `num-mess` est le numéro du message, qui va de 0 à 9999, `id` est l'identifiant de l'utilisateur ayant rédigé le message et `mess` est le message contenant 140 caractères. Les numéros de message seront incrémentés à chaque envoi de message (et quand on arrive à 9999 on repart à 0).

Un diffuseur peut donc recevoir des messages d'utilisateur à diffuser. La transmission d'un message par un utilisateur à un diffuseur se fait en mode connecté. Pour transmettre un message, un utilisateur se connecte au port de réception du diffuseur et lui envoie un message de la forme `[MESS␣id␣mess]` où `id` est l'identifiant de l'utilisateur et `mess` est le message à transmettre contenant 140 caractères. Le diffuseur renvoie alors un message de la forme `[ACKM]` et ferme la connexion.
Un utilisateur peut aussi demander une liste des derniers messages diffusés. Ce mode de communication a également lieu en mode connecté. Pour ce faire un utilisateur se connecte au port de réception du diffuseur et lui envoie un message de la forme `[LAST␣nb-mess]` où `nb-mess` est un entier (compris entre 0 et 999). L'utilisateur demande ainsi au diffuseur les `nb-mess` derniers messages diffusés. À ce message, le diffuseur répond par une séquence d'au plus `nb-mess` messages de la forme `[OLDM␣num-mess␣id␣mess]` où `num-mess` , `id` et `mess` sont identiques aux champs des messages de type `MESS`. Cette séquence contient les derniers messages diffusés. Leur nombre peut être inférieur à `nb-mess`. Lorsque le diffuseur a fini de transmettre sa séquence de messages, il le signale à l'utilisateur en envoyant un message de la forme `[ENDM]` et il ferme ensuite la connexion.

Spécification de la forme des messages

Tout d'abord chaque message terminera nécessairement par un octet servant à encoder le caractère `'\n'` (nouvelle ligne).
Les quatre premiers octets d'un message serviront à encoder dans une chaîne de caractères son type. Les types des messages étant : `DIFF`, `MESS`, `ACKM`, `LAST`, `OLDM`, `ENDM`.
Voilà maintenant les caractéristiques pour chacun des champs contenus dans les messages :

- `num-mess` sera codé sur 4 octets et contiendra la chaîne de caractères correspondant au numéro du message. Par exemple, pour le message 120, `num-mess` vaudra `0120`.
- `id` sera codé sur 8 octets et contiendra une chaîne de caractères. Si l'identifiant contient moins de 8 caractères, alors on complètera la fin de la chaîne avec des caractères #. Par exemple si l'identifiant est RADIO, alors `id` vaudra `RADIO###`.
- `mess` sera codé sur 140 octets et contiendra une chaîne de caractères. Si le message contient moins de 140 caractères, `mess` sera complété comme `id` avec des caractères # à la fin.
- `nb-mess` sera codé sur 3 octets et contiendra la chaîne de caractères correspondant au nombre de messages. Si la chaîne de caractères contient moins de 3 caractères, on la complètera avec des 0 au début comme pour `num-mess`.

On remarque que chaque message a une unique taille fixe.

Questions

Le but de cet examen est de programmer les diffuseurs ainsi que des clients.

La première étape consiste à programmer la liste des messages qui est donc une liste avec une capacité maximale de 10000 éléments.
Tout d'abord les messages seront stockés dans des objets de la classe suivante.

```
public class Message {
  //Numero du prochain message cree
  private static int next=0;

  //Numero du message
  public int num;

  //Identite du redacteur
  public String id;

  //Texte du message
  public String mess;
}
```

Dans cette classe, le champ statique ~~?~~ next sert à attribuer le prochain numéro au message, il faut donc le modifier à chaque fois qu'un nouveau message est créé en faisant attention au fait que les numéros sont bornés (inférieurs ou égaux à 9999). Donc si on a créé 10000 messages, le prochain message créé aura le numéro 0.

1. Écrire un constructeur pour la classe `Message` qui prendra comme arguments une chaîne de caractères représentant l'identité d'un rédacteur de message ainsi qu'une chaîne de caractères représentant le texte d'un message. On supposera que dans les objets de cette classe, les identités et les messages seront respectivement composés d'exactement 8 et 140 caractères, donc le constructeur devra prendre cela en compte (quitte à rajouter des # aux chaînes de caractères données en argument si nécessaire).
2. Écrire une méthode `numChaine` de la classe `Message` qui renvoie une chaîne de caractères d'exactement 4 caractères correspondant au numéro du message complété éventuellement avec des 0.

Nous allons maintenant donné la structure de données servant à stocker les messages. Il s'agit de la classe suivante.

```
public class ListeMessage {
  //Derniere position modifiee, vaut -1 au debut
  public int pos;

  //Nombre message dans le tableau
  public int nomb;

  //Tableau des messages
  public Message[] mess;
}
```

Lorsqu'un message est ajouté on stocke sa position dans `pos` et on stocke dans `nomb` le nombre de messages ajoutés (ces informations permettront en particulier d'envoyer la liste des derniers messages). Attention lorsque le tableau est plein on peut encore insérer des messages mais le champ `nomb` n'évolue plus. Ainsi on sait qu'il y a au moins `nomb` messages dans la liste, que le dernier message a été inséré à la position `pos` et les derniers messages ajoutés sont ceux avant `pos` (en considérant que le tableau `mess` est utilisé de façon circulaire).

3. Écrire un constructeur pour la classe `ListeMessage` qui ne prend pas d'argument et initialise tous les champs de l'objet. On rappelle qu'une liste n'a pas vocation à stocker plus de 10000 messages.

4. Écrire une méthode `insere` de la classe `ListeMessage` qui prend en arguments une chaîne de caractères correspondant à l'identité d'un rédacteur et une chaîne de caractères contenant le texte d'un message. Cette méthode crée l'objet de la classe `Message` correspondant l'insère dans le tableau à la position donnée par le numéro du message. On veut que cette méthode puisse être utiliser de façon sûre de façon concurrente.
5. Écrire une méthode `derniersMess` de la classe `ListeMessage` qui prend en arguments un entier `num` correspondant à un nombre de messages et renvoie un objet de la classe `LinkedList<Message>` contenant les `num` derniers messages (cette liste peut contenir moins de `num` messages si moins de `num` messages ont été ajoutés à la liste globale). On veut là encore que cette méthode puisse être utiliser de façon sûre de façon concurrente. On supposera de plus que les messages sont ajoutées dans l'ordre de leur numéro dans le tableau (en se rappelant qu'après le numéro 9999 on passe au numéro 0).

On va maintenant s'intéresser aux diffuseurs qui seront codés par la classe suivante.

```
public class Diffuseur {
  //Port  TCP
  public int portTcp;

  //Port Udp de multidiffusion
  public int portUDP;

  //Adresse multi-diffusion
  public String multip;

  //Liste des messages
  ListeMessage  lisM;
}
```

6. Écrire un constructeur de la elasse `Diffuseur` qui prend comme arguments un numéro de port TCP, un numéro de port UDP et une adresse IP de multi-diffusion donnée sous la forme d'une chaîne de caractères et qui initialise tous les champs de la classe.
7. Écrire une méthode `diffuse` de la classe `Diffuseur` qui prend comme arguments une chaîne de caractères correspondant à un message de la forme |`DIFF␣num-mess␣id␣message`| et le multi-diffuse sur l'adresse et le port de multi-diffusion du diffuseur.
8. Écrire une méthode `nouveauMess` de la classe `Diffuseur` qui prend comme arguments une chaîne de caractères correspondant à un message TCP envoyé par un utilisateur de la forme |`MESS␣id␣message`| et un objet `PrintWriter` correspondant à une socket TCP ouverte. Cette méthode ajoute le message à la liste des messages, multi-diffuse le message correspondant et envoie |`ACKM`| sur la socket TCP.
9. Écrire une méthode `extraitInt` de la classe `Diffuseur` qui prend comme arguments une chaîne de caractères correspondant à un message |`LAST␣nb-mess`| et renvoie l'entier correspondant à `nb-mess`.
10. Écrire une méthode `derniers` de la classe `Diffuseur` qui prend comme arguments une chaîne de caractères correspondant à un message |`LAST␣nb-mess`| et un objet `PrintWriter` correspondant à une socket TCP ouverte. Cette méthode enverra sur la socket les réponses à la requête |`LAST␣nb-mess`|.
11. Écrire une méthode `traiteTCP` de la classe `Diffuseur` qui prend comme arguments une chaîne de caractères correspondant à un message TCP envoyé par un utilisateur et un objet `PrintWriter` correspondant à une socket TCP ouverte et répond au message comme décrit précédemment. Dans cette question, on traite donc tous les types de message.

On va vouloir que notre diffuseur puisse accueillir plusieurs clients en même temps pour cela on va le rendre *multi-threadé*. Le code exécuté par chaque thread sera donc dans la classe suivante.

```
public class EntiteClient implements Runnable {
  //Socket associe au client
  Socket s;

  //Diffuseur associe
  Diffuseur dif;

  public EntiteClient(Socket s,Diffuseur d){
    this.s=s;
    this.d=d;
  }
}
```

12. Écrire la méthode run de la classe `EntiteClient` qui ouvre les flux associés à la socket, attend en boucle les messages TCP des utilisateurs et les traite.
13. Écrire la méthode go de le classe `Diffuseur` qui lance le serveur d'écoute attend les connexions des clients et traite ses connexions de façon concurrente.
14. Écrire dans une classe `TestDiffuseur` un programme qui lance un diffuseur. À vous de choisir les ports et l'adresse de multi-diffusion.

Passons maintenant à une utilisation du code précédent. On suppose que le diffuseur de la question précédente tourne sur une machine dont le nom complet est `mac.resoperso.pr6.fr`.

15. Écrire en C un programme qui affichera à l'écran les messages multi-diffusés par le diffuseur précédent.
16. Écrire en C un programme qui donnera 100 messages à diffuser au diffuseur de la question 14. Ces messages auront la forme `HELLO i` pour `i` allant de 1 à 100.
17. Écrire en C un programme qui demandera les 10 derniers messages diffusés par le diffuseur de la question 14 et les affichera à l'écran.

Quelques fonctions de Java pouvant être utiles

— Méthodes de la classe `LinkedList<E>`

```
boolean add(E e)

ListIterator<E> listIterator()
```

— Méthodes de la classe `ListIterator<E>`

```
boolean hasNext()

E next()
```


— 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.

static String valueOf(int i)
//Returns the string representation of the int argument.
```

— Méthodes de la classe `Integer`

```
static int parseInt(Sting s)
/*Parses the string argument as a signed decimal integer*/
```