EXAMEN 2ème SESSION — PROGRAMMATION RÉSEAU — 22/06/2017 15^{30}-18^{30}

Les seuls documents autorisés sont les documents « sur papier » ;
à l'exclusion de la copie ou les 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).
- Les messages circulant sont signalés entre crochets `[...]` et **les crochets ne font pas partie du message.**
- On ne demande pas de traiter les exceptions et on ne demande pas d'écrire les `import` d'API

Contexte

On désire développer un serveur de salon de discussion pour plusieurs clients souhaitant s'échanger des messages.
Les clients seront connectés à ce serveur et le serveur pourra en plus multi-diffuser certains messages sur l'adresse de multi-diffusion `233.000.000.001` sur le port `3333`.
Chaque client aura une identité qui sera une chaîne de 8 caractères et on suppose qu'il n'y a pas deux clients avec la même identité (autrement dit vous ne devez pas traiter ce cas).
Le serveur devra retenir la liste des clients connectés ainsi que les sockets qui leur sont associées. On supposera dans cet examen qu'**un client connecté ne se déconnecte pas.**
Un client avant de pouvoir envoyer et recevoir des messages, devra d'abord se connecter et s'enregistrer en envoyant un message de la forme `[R id\n]` où `id` sera son identité.

Un client connecté au serveur pourra ensuite effectuer les actions suivantes :

- Envoyer un message destiné à tous les clients connectés. Pour cela il enverra au serveur un message de la forme `[M mess\n]` où `mess` est le message qu'il souhaitera envoyer, il s'agit d'une chaîne d'au plus 250 caractères ne contenant pas le caractère `\n` ;
- Envoyer un message à un client particulier. Pour cela il enverra au serveur un message de la forme `[P id mess\n]` où `id` représentera l'identité du client, codé sur 8 caractères, auquel est destiné le message `mess` qui sera une chaîne d'au plus 250 caractères ne contenant pas le caractère `\n` ;
- Faire diffuser un message (sur l'adresse de multi-diffusion du serveur). Pour cela il enverra un message au serveur de la forme `[D mess\n]` où le message `mess` sera une chaîne d'au plus 250 caractères ne contenant pas le caractère `\n`.

En ce qui concerne le serveur :

- Quand il recevra un message de la forme `[R id\n]`, il ajoutera le client correspondant à sa liste de clients connectés.
- Quand il recevra un message de la forme `[M mess\n]`, il enverra un message `[SM mess\n]` à tous les clients connectés.
- Quand il recevra un message de la forme `[P id mess\n]`, il enverra un message `[SP mess\n]` au client avec l'identité `id` si celui-ci est présent dans sa liste et sinon il ne fera rien.
- Quand il recevra un message de la forme `[D mess\n]`, il multi-diffusera un message `[SD mess\n]`.

Questions

Dans un premier temps, il s'agit de développer le serveur en Java. Nous supposerons que les informations des clients sont stockées dans la classe `ClientInfo` qui aura les champs suivants :

```
class ClientInfo{
  public String id;
  public Socket s;
}
```

1. Écrire un constructeur de la classe `ClientInfo` qui prendra comme arguments une chaîne de caractères correspondant à l'identité et une Socket.

On aura ensuite une classe `ClientList` qui gèrera une liste d'objets de la classe `ClientInfo`. Cette classe aura donc un champ `public LinkedList<ClientInfo> list;`.

2. Écrire le constructeur de la classe `ClientList` en supposant qu'au début la liste est vide.
3. Dans cette classe, on va faire des méthodes pour ajouter des éléments à liste et aussi pour parcourir la liste et ces méthodes seront probablement utilisées par plusieurs *threads* en même temps. Comment garantir que l'utilisation de ces méthodes en parallèle ne posent pas de problème ?
4. Écrire une méthode `addClient` de la classe `ClientList` qui prend un argument une chaîne de caractères correspondant à l'identité d'un client ainsi qu'une Socket et qui ajoute les informations correspondantes à la liste (on rappelle qu'il n'est pas demandé de vérifier si le client est déjà dans la liste ou non).
5. Écrire une méthode `sendAll` de la classe `ClientList` qui prend en argument une chaîne de caractères correspondant à un message `mess` et qui l'envoie à tous les clients dans la liste (en respectant le protocole d'envoi de messages du serveur).
6. Écrire une méthode `sendClient` de la classe `ClientList` qui prend en argument une chaîne de caractères correspondant à un message `mess` et une chaîne de caractères correspondant à l'identité d'un client et qui envoie le message au client correspondant si il est dans la liste (en respectant le protocole d'envoi de messages du serveur) et qui ne fait rien si le client n'est pas dans la liste.
7. Écrire une méthode `newMess` de la classe `ClientList` qui prend en argument une chaîne de caractère correspondant à un message reçu par le serveur et qui la traite selon le type de messages (`M`, `P` ou `D`) en respectant le comportement du serveur décrit précédemment.

Comme nous l'avons dit, nous désirons que notre serveur puisse traiter plusieurs connexions à la fois. Nous aurons donc une classe `Connection` qui implémente l'interface `Runnable` et qui aura les champs et le constructeur suivant :

```
class Connection implements Runnable{
  public ClientList cList;
  public Socket so;

  public Connexion(ClientList _cList,Socket _so){
    cList=_cList;
    so=_so;
  }
}
```

8. Écrire la méthode `run` de la classe `Connection` qui attend des messages envoyés au serveur et arrivant sur la socket `so` et les traite (en supposant que la liste `cList` est la liste de clients du serveur). L'idée est que cette méthode `run` est appelé juste après qu'un client soit connecté. Il ne faut pas oublier ici qu'un client avant d'envoyer des messages doit s'enregistrer (et donc la méthode `run` devra aussi traiter l'enregistrement).

On s'intéresse maintenant à la classe `Server` qui contiendra le code principal du serveur. Cette classe aura les champs suivants :

```
class Server{
  public ClientList cList;
  public int port;
}
```

9. Écrire le constructeur de la classe `Server` qui prend en argument un numéro de port. On suppose qu'aucun client n'est enregistré dans la liste au début.
10. Écrire une méthode `listen` de la classe `Server` qui attend des connexions sur son port et traite les connexions. Bien entendu on souhaite que plusieurs clients puissent être connectés en même temps.
11. Dans la question précédente, on suppose que le serveur peut accepter un nombre non borné de connexions. Que faudrait-il faire pour que le serveur ne puisse pas accepter plus de `n` connexions (`n` étant un entier positif).
12. Écrire dans une classe `Prog1` un programme principal qui crée un objet `Server` (on pourra choisir le port que l'on souhaite) et le lance pour qu'il traite les clients souhaitant s'y connecter.
13. On suppose que le programme précédent s'exécute sur la machine `amachine.somewhere.tz`. Écrire en C un client pour notre serveur qui se connecte, s'enregistre (le choix de l'identité est libre) et demande au serveur d'envoyer un message "Hello !" à tous les clients connectés et également de multi-diffuser un message "Hello World !" et qui ensuite attend les messages qui lui sont envoyés personnellement par le serveur et les affiche dans le terminal.
14. Écrire en C un autre client pour notre serveur qui se connecte, s'enregistre (le choix de l'identité est libre) et qui envoie "Salut !" au serveur pour que celui-ci le transmette uniquement au client de la question précédente (que l'on suppose encore connecté).
15. Écrire en C un autre client pour notre serveur qui se connecte, s'enregistre (le choix de l'identité est libre) et affiche ensuite sur le terminal tous les messages qui lui sont envoyés par le serveur ainsi que tous les messages multi-diffusés par le serveur.

Quelques fonctions de Java pouvant être utiles

— Méthodes de la classe `LinkedList<E>`

```
boolean add(E e)

boolean remove(Object o)

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)
```