RICM 4ème année

17/4/2013

Examen de Systèmes Répartis
Durée : 2h, Documents autorisés à l'exception des livres. Le barème est indicatif.

Partie A – Applications Web (3 pts)

Question 3. Soit le code d'une servlet ci-après.

a) Ce code contient deux erreurs qui seront détectées à la compilations (uniquement au niveau de la déclaration des deux méthodes). Dites lesquelles (no de ligne).
b) Soit l'erreur résolue (les deux méthodes sont toujours présentes). Que sera le résultat d'une requête de type GET?
c) Une requête de type POST générera une erreur, vrai ou faux? Produire la sortie si faux. Expliquer en deux lignes si vrai.

```
1. import javex.servlet.*;
2. import javax.servlet.http.*;
3. import java.io.*;
4.
5. public class MaServlet exends HttpSerlet{
6.
7.     public void service(HttpSerletRequest req, HttpSerletResponse resp){
8.       Writer out=resp.getWriter();
9.       out.write("Que penses tu de la methode service?");
10.    }
11.
12.     public void doGet(HttpSerletRequest req, HttpSerletResponse resp){
13.       Writer out=resp.getWriter();
14.       out.write("Que penses tu de la methode doet?");
15.     }
16. }
```

Partie B – Systèmes répartis à objets (9 pts)

On souhaite accéder par Java RMI à des données stockées sur un serveur distant. Chaque donnée, de type *java.io.Serializable*, est associée à une clé unique de type *String*. Le serveur souhaite savoir qui manipule quelles données à quels instants. Il définit pour cela une notion de connexion. Avant d'accéder à une donnée, un client potentiel doit invoquer la méthode *connect* du serveur. Cette méthode prend comme paramètre un nom d'utilisateur (*String*) et retourne un entier représentant un identifiant de connexion. Le serveur fournit également les méthodes suivantes :

- *consult(int id, String key)* : à partir d'un identifiant de connexion et d'une clé, retourne l'élément associé à la clé, ou *null* s'il n'y en a pas.
- *update(int id, String key, Serializable value)* : à partir d'un identifiant de connexion et d'une clé, met à jour la donnée associée à la clé.
- *close(int id)* : à partir d'un identifiant de connexion, ferme la connexion. Cette méthode ne retourne rien.

Par exemple, un scénario possible d'interrogation du serveur *s* va consister à invoquer les méthodes suivantes :

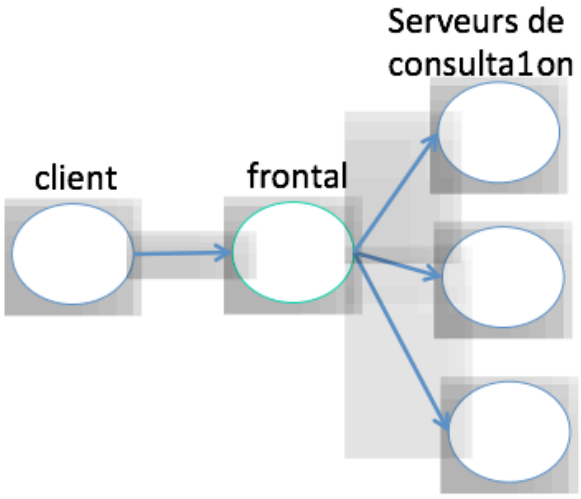
int id = s.connect("Bob");

```
Serializable obj = s.consult(id,"chauffagiste");
printResult(("chauffagiste ", obj) ;
s.close(id) ;
```

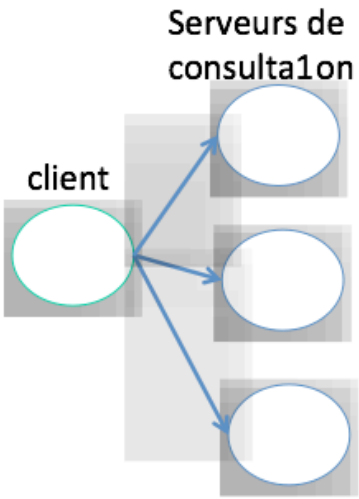
Question 1 (1 pt). Définir l'interface Java RMI *ServerItf* correspondant aux méthodes *connect*, *consult, update* et *close*.

Question 2 (1 pt). Proposez une mise en œuvre de la classe *Server*. On s'intéresse uniquement à la mise en œuvre de la méthode connect et de la méthode main. On supposera que tout serveur s'enregistre auprès du service de nommage de RMI. On supposera également qu'un serveur maintient les données, ainsi que les informations de connexion, dans des HashMap.

Le volume de données augmentant, il ne devient plus possible de les stocker sur un seul serveur. On décide de les répartir sur plusieurs serveurs, et on envisage les deux architectures suivantes. Dans la première, un frontal gère l'accès aux serveurs de consultation. Les clients ne connaissent que lui. Le frontal se charge de déléguer la requête aux serveurs de consultation. Le frontal connaît donc tous les serveurs de consultation. Dans la seconde, les clients connaissent tous les serveurs de consultation et s'adressent directement au serveur dont ils souhaitent consulter les données.



Architecture 1



Architecture 2

Question 3 (0.5 pt). En cas d'ajout ou de retrait d'un serveur de consultation, que faut-il faire avec la première architecture ? Avec la seconde ? Quelle est celle qui pose le moins de problème ? Pourquoi ? (1 point).

On se place maintenant dans le contexte de l'architecture 1.

Question 4 (1.5 pt). Précisez les rôles du frontal et d'un serveur (3 lignes max). Proposez une définition pour l'interface *FrontalItf*. L'interface *ServerItf* définie précédemment est-elle toujours valide pour les serveurs de consultation?

Question 5 (0.5 pt). Le frontal doit être capable d'aiguiller les requêtes de consultation vers le bon serveur. Proposer un mécanisme de répartition des données sur les serveurs qui permette de faire cela efficacement (on ne vous demande pas de l'implanter).

Question 6 (1 pt). Combien de communications distantes sont mise en œuvre lorsqu'un client réalise une consultation ? combien de threads sont sollicités ? soyez précis dans votre réponse.

Question 7 (1 pt). On souhaite maintenant pouvoir utiliser 2 frontaux. Un client peut se connecter à l'un ou à l'autre des frontaux. Proposez un principe pour gérer l'allocation des identifiants de connexion de manière à garantir leur unicité.

Question 8 (1 pt). Selon l'interface *ServerItf*, comment sont passées les données consultées? par référence? par copie?
Donnez un exemple de situation, dans le contexte courant, pour laquelle ce mode de transmission ne vous paraît pas adapté.

Question 9 (1.5 pt). Quelles sont les interfaces et classes qui doivent être présentes chez le client? chez le frontal ? chez le serveur? Un client peut il recevoir un objet de classe inconnue de lui lors d'une invocation de la méthode *consult* ? que se passe t-il dans ce cas ?

Partie C – NIO (3 pts)

Soit la classe ci-après mettant en œuvre une continuation sur écriture pour un serveur NIO. Cette classe permet d'envoyer un tableau d'octets (*data*) dans un canal donné (*socketChannel*). La méthode *handleWrite* est appelée par le sélecteur *nio* lorsqu'il y a de la place en sortie sur le canal.

a) La méthode *handleWrite* est conceptuellement mal programmée. Expliquer pourquoi (5 lignes max) en pointant les lignes du code qui posent problème.
b) Proposez une correction.

```
1. public class WriteCont{
2.   public byte[] data = null;   // the data to send
2.   private ByteBuffer bufData = null;  // buffer used to send the data
3.   private ByteBuffer bufLength = null;  // buffer used to send the length
4.
5. public void handleWrite(SelectionKey key, SocketChannel socketChannel)
throws IOException{
6.     if (data != null) {
7.       if (bufLength ==null)
8.         bufLength = ByteBuffer.wrap(intToByteArray(data.length));
9.
10.      while (bufLength.remaining() > 0)
11.         socketChannel.write(bufLength);
12.
13.       if (bufData ==null)
14.         bufData = ByteBuffer.wrap(data);
15.
16.       while (buf.remaining() > 0)
17.           socketChannel.write(bufData);
18.
19.       // reinitialize the continuation
20.       bufLength = null;
21.       bufData = null ;
22.       data = null;
23.    }
24.  ...
```

Partie D – Serveur HTTP (5 pts)

Le but de cet exercice est d'implanter une partie du protocole HTTP avec l'API TCP du langage Java. On rappelle que la commande GET a la syntaxe suivante : *GET url.* L'url à la forme suivante : *protocole"://"adresse[:port]["/"chemin]["/"nomdefichier]*. Les éléments entre crochets sont facultatifs. Les éléments entre guillemets sont des chaînes de caractères finales.

Cette commande GET peut être suivie de lignes d'entêtes ayant la forme suivante et, finalement d'un retour chariot qui indique la fin de la commande.

User-agent: Mozilla/4.0
Accept: text/html, image/gif,image/jpeg
Accept-language:fr

La réponse à la commande GET a schématiquement la syntaxe suivante qui indique le protocole utilisé par le serveur (soit HTTP/1.0) et un code réponse sous la forme d'un entier à trois positions :
HTTP/1.0 code

Les deux codes de réponse que nous vous demandons de considérer sont 200 (tout est OK) et 404 (non trouvé). Si le fichier a été trouvé, cette ligne de réponse est suivie des lignes d'entête suivantes, d'une ligne blanche et des données, soit la forme suivante :

Date: Thu, 06 jan 2004 12:00:15 GMT
Server: Apache/1.4.0 (Unix)
Content-Length: 6821
Content-Type: text/html (on se limitera à ce type de contenu dans le cas présent)

... data ....

Question 1 (1 pt). Ecrire une classe Java *HttpClient* avec une méthode *get(String url, String hostName, int port)* qui envoie une requête HTTP au serveur localisé à l'adresse indiquée et écoutant sur le port indiqué. Ce client affiche ensuite le code reçu en retour.

Question 2 (2 pts). Ecrire une classe Java *HttpServer* mettant en œuvre un serveur multi-threadé de 10 threads . La méthode une *main de ce serveur :*
- écoute les demandes de connexion sur le port TCP 1023
- délègue à un thread (d'un pool de threads) le traitement d'une requête. La définition de la classe de ce thread fait l'objet de la question suivante.

Question 3 (2 pts). Ecrire en Java le pseudo-code exécuté par le thread traitant une requête HTTP. On supposera disponible les primitives suivantes :

- *String getFilePath(String url)* retourne le chemin absolu permettant d'accéder au fichier demandé
- *String getDate()* retourne la date sous la forme attendue dans une réponse http (ex *Thu, 06 jan 2004 12:00:15 GMT)*
- *String getServerInfo()* retourne la description du serveur telle qu'attendue dans une réponse http (ex : *Apache/1.4.0 (Unix))*