Master 2 Informatique I2A

Master DVL Master ITVL

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

Master mention Informatique

Parcours Informatique Avancé et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

Filière Informatique

VVIXECSD

Communication dans les systèmes distribués

Mr PHILIPPE - LAURENT
laurent.philippe@univ-fcomte.fr

Mr MAZOUZI - KAMEL
kamel.mazouzi@univ-fcomte.fr

Université de Franche-Comté
UBFC Université Bourgogne Franche-Comté

UNIVERSITÉ DE FRANCHE-COMTÉ

Module INFO-CSD
Laurent Philippe

Communication dans les Systèmes Distribués

Master Informatique Avancée et Applications
2ème Année

EXERCICES

Centre de Télé-enseignement Universitaire

Centre de télé enseignement
Filière Informatique
Domaine Universitaire de la Bouloie
25030 Besançon Cedex (France)

Table des matières

Chapitre 1

Sockets

Les programmes sockets sont des programmes Java classiques, composés de classes, regroupées dans un même fichier ou dans des fichiers indépendants. Ces programmes peuvent être codés à partir d'un éditeur de texte ou d'un EDI (Environnement de Développement Intégré).

Toutes les versions de Java (au moins jusqu'à la version 17) disposent des bibliothèques nécessaires à faire fonctionner les sockets. Pour compiler les programmes avec un EDI il suffit donc de disposer d'une installation de Java et de paramétrer l'accès. La compiltion est alors automatique. Pour compiler les programmes en ligne de commande, il suffit d'utiliser le compilateur Java (`javac`) qui génère les `.class`.

L'exécution des programmes sockets nécessite le lancement simultané d'à minima deux programmes (le client et le serveur). La procédure de lancement peut varier suivzant l'EDI utilisé. Il peut être nécessaire de configurer la possibilité de lancement plusieurs programmes en parallèle. Depuis la ligne de commande, le plus simple est de disposer d'autant de fenêtres que de programmes à lancer et de lancer chacun dans une fenêtre indépendante.

À noter que, pour que les exemples et les corrections marchent il faut, bien sûr, que le nom de la machine soit adapté. Si vous faites le test sur une seule machine il peut-être plus simple d'utiliser ``` ``localhost'' ``` comme nom de machine. Comme son nom l'indique, ce nom réfère à la machine locale.

Exercice 1 : Modes de communication

Dans cet exercice nous allons utiliser les extraits de code vu en cours pour réaliser une première application avec des sockets.

Pour exécuter les programmes, créez deux fenêtres shell, positionnez-vous dans le répertoire `ex01` et lancez l'exécution de chacun des programmes dans des fenêtres différentes. En mode non-connecté, le programme récepteur doit être lancé en premier. En mode connecté, c'est le serveur qui doit être lancé en premier. Nous vous conseillons enfin d'utiliser `localhost` comme nom d'hôte.

Question 1.1

Composer les parties de code données dans le cours pour passer une valeur entière entre deux sockets non-connectées et une chaîne de caractère entre deux sockets connectées.

Correction de la question 1.1

La réponse à cette question se trouve dans le répertoire `ex01` dans les fichiers dont le nom finit par `UDP1.java` et `UDP2.java`.

Question 1.2

Faire l'inverse : la chaîne en mode non-connecté puis la valeur en mode connecté.

Correction de la question 1.2

La réponse à cette question se trouve dans le répertoire `ex01` *dans les fichiers dont le nom finit par* `TCP1.java` *et* `TCP2.java`*.*

Exercice 2 : Plusieurs échanges

Dans l'exercice précédent, nous nous sommes contentés de faire un échange entre les deux programmes. Dans cet exercice, nous mettons en place plusieurs échanges pour établir une vraie communication. Nous utilisons le mode connecté.

Question 2.1

Compléter les progammes échangeant la chaîne de caractères de manière à ce que le récepteur de la chaîne retourne la taille de celle-ci à l'émetteur. Nous utilisons les classes `DataInputStream, DataOutputStream` pour les échanges des données.

Correction de la question 2.1

La réponse à cette question se trouve dans les fichiers `ex02/Receiver2.java` *et* `ex02/Sender1.java`*.*

Question 2.2

Écrire les programmes de manière à ce que l'émetteur de la chaîne saisisse celle-ci au clavier avant de l'envoyer. Nous supposerons que la taille maximale de la chaîne est de 256 caractères. Répéter l'échange tant que la chaîne envoyée n'est pas «fin». Dans ce cas, fermer proprement la connexion entre les deux programmes sans retourner de réponse.

Correction de la question 2.2

La réponse à cette question se trouve dans les fichiers `ex02/Receiver2.java` *et* `ex02/Sender2.java`*. Pour envoyer "t recevoir les chaînes de caractères nous utilisons les méthode* `writeUTF(String)` *et* `readUTF()`*.*

Ces méthodes détectent automatiquement la taille des chaînes de caractères dans un flux.

Si nous avons à utiliser les méthodes de base `read(byte[] b)` *et* `write(byte[] b)`*. L'expéditeur doit envoyer la taille de la chaîne en premier pour informer le récepteur de la quantité de données à allouer.*

Question 2.3

Modifier le premier programme de l'exercice de manière à ce qu'il envoie successivement cinq chaînes de caractères, sans attendre de retour du récepteur. Une fois les chaînes reçues, le récepteur retourne un tableau contenant les différentes tailles des chaînes.

Correction de la question 2.3

La réponse à cette question se trouve dans les fichiers `ex02/Sender3.java` *et* `ex02/Receiver3.java`*.*

Il n'existe pas une méthode Java pour envoyer une tableau d'entier. Dans ce cas, nous enverrons les entiers se forme de flux en utilisons la méthode `writeInt(int)`*. Nous recevons le résultat dans une boucle en utilisant la méthode* `readInt()`*.*

Exercice 3 : Communication à plusieurs

Écrire trois programmes qui communiquent sous la forme d'un anneau : le premier envoie ses messages au deuxième, qui les envoie au troisième. C'est le premier programme qui initie la communication. Par la suite, un programme ne communique avec le suivant que s'il a reçu un message de son prédécesseur.

Question 3.1

Quel mode de communication doit-on utiliser ?

Correction de la question 3.1

Puisque chacun des programmes ne communique qu'avec le suivant, le mode connecté est tout à fait adapté pour permettre plusieurs communications.

Question 3.2

Comment garantir que les adresses utilisées pour les communications sont correctes si les programmes sont lancés par des personnes différentes sur des machines différentes ?

Correction de la question 3.2

Ce problème est relativement difficile à résoudre dans l'absolu. Une solution ici consiste à attribuer le même numéro de port à l'ensemble des programmes et de connaître le nom de la machine avec laquelle nous communiquons. Pour fixer le numéro de port, une solution consiste à l'enregistrer dans la base des services.

Question 3.3

Comment permettre qu'aucun des programmes ne soit bloqué par les autres ?

Correction de la question 3.3

Il faut que chaque programme envoie un message même s'il n'a rien à dire pour passer la main au suivant.

Question 3.4

Écrire les programmes.

Correction de la question 3.4

La réponse à cette question se trouve dans le répertoire `ex03`.

Exercice 4 : Communication objet

On veut écrire deux programmes Java qui échangent un message composé de plusieurs informations : nom de l'émetteur, date d'émission et texte du message.

Question 4.1

Comment réaliser cet échange de manière à ce que l'ensemble des données soit transmis en une seule fois ?

Correction de la question 4.1

On encapsule les données dans une classe. On peut alors transmettre un objet de cette classe en une seule fois.

Question 4.2

Comment structurer les fichiers de programmes de manière à permettre une évolution des données contenues dans le message ?

Correction de la question 4.2

On peut passer par une interface Java. Le client peut alors implémenter cette interface comme bon lui semble et envoyer l'objet. Le serveur reçoit l'objet mais l'utilise via son interface.

Question 4.3

Écrire les programmes.

Correction de la question 4.3

La réponse à cette question se trouve dans le répertoire `ex04`*.*

Exercice 5 : Serveur de temps concurrent

Question 5.1

Écrire un serveur qui permet de répondre à plusieurs clients en leur retournant la date courante du serveur.

Correction de la question 5.1

La réponse à cette question se trouve dans le répertoire `ex05`*.*

Question 5.2

Modifier l'exercice précédent pour que le serveur puisse traiter plusieurs clients en parallèle.

Correction de la question 5.2

La réponse à cette question se trouve dans le répertoire `ex06`*. Une phase d'attente a été ajoutée pour tester la création de plusieurs threads simultanéement.*

Exercice 6 : Serveur de calcul

Écrire un programme serveur qui permet de réaliser les quatre opérations arithmétiques de base $(+, -, *, /)$ avec des entiers pour le compte d'un client. Le serveur ne traite qu'un client à la fois.

Question 6.1

Quel problème se pose pour la constitution d'un message et quelles solutions ? Proposer plusieurs solutions.

Correction de la question 6.1

Le client doit envoyer plusieurs données au serveur : l'opérateur et les deux opérandes.

Une solution consisterait à écrire la requête sous une forme textuelle, par exemple «4 + 5». Cette solution implique cependant que le serveur réalise une analyse syntaxique du message reçu pour comprendre la demande du client, ce qui est trop lourd à gérer côté serveur.

Une autre solution consiste à envoyer les données les une après les autres et de les recevoir sous cette forme en respectant un ordre établi, par exemple d'abord l'opérateur puis l'opérande un, puis l'opérande deux. Cette solution est envisageable mais demande plusieurs envois.

Il faut plutôt générer un message structuré pour que le serveur reçoive toutes les données en une seule fois. En java le message structuré peut facilement être implémenté sous la forme d'un objet. Ici nous définissons l'objet `Calcul`*.*

Nous voyons plus loin que la réponse pose le même type de problème.

Question 6.2

Quels sont les cas d'erreur possibles ? Apporter des solutions de manière à garantir la sécurité du serveur.

Correction de la question 6.2

Il y a deux cas d'erreurs : la division par zéro et une mauvaise opération. Dansd l'objet de réponse il est donc nécessaire d'ajouter un code d'erreur qui indique que l'opération s'est bien passée ou qu'il y a eu une erreur.

Question 6.3

Écrire le programme serveur qui ne traite qu'une opération par connexion pour le compte d'un client, puis le programme client.

Correction de la question 6.3

La réponse à cette question se trouve dans le répertoire `ex06/qu3`*.*

Question 6.4

Comment permettre à un client d'effectuer une suite d'opérations telle que $(4 * 5) + (34/2)$?

Correction de la question 6.4

Il est possible soit d'envoyer le message en une seule fois mais nous sommes encore confrontés à des difficultés d'encodage de la requête et d'analyse de son contenu. Il est possible d'envoyer plusieurs opérations à réaliser au serveur soit en faisant autant de connexion que de demande d'opérations, soit en maintenant la connexion ce qui pose le problème évoqué à la question suivante.

Question 6.5

Comment permettre à un client d'envoyer un nombre quelconque d'opérations à effectuer et de terminer sans affecter le serveur.

Correction de la question 6.5

Il faut maintenir la connexion tant que le client envoie des opérations. Ceci nécessite alors de gérer une fin de connexion. Cette fin peut être réalisée avec un code d'opération spécifique (par exemple en ajoutant l'opérateur `FIN`*). Le client peut aussi simplement se déconnecter sans prévenir le serveur quand il a fini. Dans ce cas le serveur rec*

La réponse à cette question se trouve dans le répertoire `ex06/qu5`*.*

Question 6.6

Le serveur traite plusieurs clients. Adapter le serveur pour qu'il puisse servir d'autres clients avant que le premier ne soit terminé.

Correction de la question 6.6

Pour traiter plusieurs client en même temps nous utilisons la même méthode que dans l'exercice sur le serveur de temps : nous créons un thread dès qu'un nouveau client se connecte.

La réponse à cette question se trouve dans le répertoire `ex06/qu6`*.*

Chapitre 2

RMI

Les applications utilisant les Java RMI sont des programmes Java classiques, composés de classes, regroupées dans un même fichier ou dans des fichiers indépendants. Ces programmes peuvent être codés à partir d'un éditeur de texte ou d'un EDI (Environnement de Développement Intégré).

Toutes les versions de Java (au moins jusqu'à la version 17) disposent des bibliothèques nécessaires à faire fonctionner les RMI. Pour compiler les programmes avec un EDI il suffit donc de disposer d'une installation de Java et de paramétrer l'accès. La compiltion est alors automatique. Pour compiler les programmes en ligne de commande, il suffit d'utiliser le compilateur Java (javac) qui génère les .class. À noter que dans les versions anciennes de Java (avant 5) les classes talon et squelette devaient être générées par compilation à partir de la classe d'interface grâce à la commande `rmic`. Depuis la version 5 tout est générés automatiquement à la compilation de l'interface.

L'exécution RMI nécessite une phase préalable, celle d'installation des fichiers sur les différentes machines de l'environnement d'exécution (cette étape n'est pas exigée si le déploiement est effectué sur un même système de fichiers). L'interface de l'objet distant (donc le fichier `.class` généré) devra être connue des deux côtés (client et serveur). L'implémentation de l'objet est connue uniquement par le serveur.

Lancement du serveur de noms

La commande `rmiregistry` lance un serveur de noms destiné à RMI ; celui-ci doit être démarré sur la même machine que le programme effectuant la publication (l'application serveur).

Le rôle du serveur de noms est donc de faire le lien entre le serveur et le client en permettant le passage de l'un à l'autre de l'objet distant de l'interface. Pour cela il est nécessaire que le serveur de noms sache où trouver la classe correspondante à l'interface. Il existe plusieurs moyens d'informer le serveur de noms de l'emplacement où se trouve la classe interface. Les moyens les plus simples, pour un premier test, sont soit de lancer le serveur de noms dans un répertoire où se trouve la classe, soit de mettre le répertoire où se trouve la classe dans la variable $CLASSPATH$, soit de donner l'information au lancement de la commande `rmiregistry` en ajoutant le paramètre $-J - Djava.class.path = ...$ Nous expliquons en **??** le principe du chargement dynamique de classes qui est à privilégier pour le déploiement d'une application à plus grande échelle.

À noter que, à partir de la version 7, Java permet un lancemet automatique du `rmiregistry` depuis le code de l'objet serveur, en utilisant la classe `LocateRegistry` de la manière suivante :

```
Registry registry = LocateRegistry.createRegistry(int port);
```

Dans les exercices, pour des questions de simplicité, nous utilisons le lancement manuel.

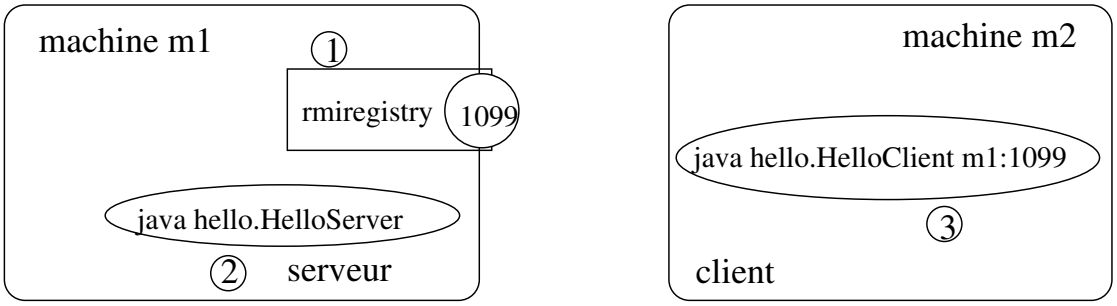


FIGURE 2.1 – Exécution de l'application RMI Hello

Lancement de l'application serveur

L'application serveur réalisant la création et la publication de l'objet distant peut être lancée une fois le serveur de noms en exécution. Supposons que ce lancement s'effectue sur la machine *m1* comme dans la figure 2.1.

Le lancement du serveur pour l'exemple proposé est :

```
java HelloServer
```

La méthode d'exposition d'objet distant a pour effet de créer et démarrer le serveur d'objets dans un processus léger concurrent. Ce serveur d'objets est automatiquement placé en attente d'appels à distance de méthodes. Pour cette raison, l'exécution du serveur ne rend pas la main, même après l'affichage des messages (le serveur d'objets reste en attente). Ainsi, le serveur peut être lancé en arrière-plan afin de pouvoir réutiliser le terminal de son lancement.

Lancement de l'application cliente

L'application cliente s'exécute généralement sur une machine distincte de la machine serveur. Dans notre exemple, le client nécessite connaître l'adresse de la machine sur laquelle le serveur de noms tourne (où l'objet distant est publié). L'adresse sera définie par le nom de la machine suivi de `:` et du numéro du port de lancement du serveur de noms.

Le lancement du client pour l'exemple proposé est :

```
java HelloClient m1:1099
```

A noter que, comme pour le serveur de noms, il est nécessaire, pour que le client puisse s'exécuter correctement, qu'il ait accès à la classe d'interface, c'est-à-dire en mettant éventuellement à jour son classpath.

Les étapes du lancement d'une application distribuée RMI sont résumées dans la figure 2.1.

Exercice 7 : Application Hello distribuée

Question 7.1

Analyser et développer l'application `Hello` présentée en cours. Déployer l'application sur un environnement distribué.

Correction de la question 7.1

L'application `Hello` en distribué se déroule sur le schéma suivant : le client saisit une chaîne de caractères qu'il envoie à un serveur. Le serveur, à la réception du message de la part du client, l'affiche sur son terminal, précédé par la chaîne `Hello`.

La description de l'approche à suivre pour le développement en distribué de cette application est donnée dans la section 2.3 du cours et les étapes du déploiement dans la section 2.4.

Exercice 8 : Calcul distribué d'une approximation du nombre Pi

L'intérêt de cet exercice est de trouver une approximation du nombre π d'une manière distribuée. La formule la plus simple est celle déterminée par l'allemand Leibniz en 1674 :

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

On supposera qu'un serveur dispose du savoir-faire pour calculer une approximation du nombre π à l'aide de cette méthode et les clients peuvent l'interroger afin de récupérer cette estimation.

Question 8.1

Développer l'application distribuée permettant d'approximer la valeur du nombre π à partir de la formule de Leibniz.

Correction de la question 8.1

Le schéma de communication dans ce problème est similaire à l'application `Hello` *vue précédemment. En plus, le serveur rend une information au client. Un objet distant réalise les calculs d'approximation du nombre π demandés par les clients et leurs renvoie les résultats.*

Définition de l'interface de l'objet distant *La première question qu'on doit se poser en réalisant une application répartie est : quels sont les services que l'objet distant fournit ?*

Un serveur est capable de réaliser l'approximation du nombre π s'il accueille un objet fournissant ce service de calcul. Malgré le fait que la formule se base sur une somme infinie de termes, l'algorithme de calcul devra imposer une limite sur le nombre de termes à sommer. Plus le nombre de termes est grand, plus l'approximation est exacte. Ainsi, le service de l'objet distant prendra en paramètre le nombre de termes à sommer et retournera la valeur approximée de π.

L'interface de l'objet distant peut être définie ainsi :

```
package calculpi;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ApproxPi extends Remote {
  public double valPi(int nbTermes) throws RemoteException;
}
```

Définition de l'implémentation de l'objet distant *Le serveur accueillant l'objet distant doit connaître l'implémentation de cet objet, c'est-à-dire la description du service réalisé par l'objet distant. La deuxième étape de la réalisation de l'application est donc le développement de l'implémentation de l'objet distant.*

La formule de calcul de π donnée dans l'énoncé de l'exercice,

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots,$$

peut être réécrite sous la forme :

$$\frac{\pi}{4} = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{1}{(2n-1)}$$

ou encore, en considérant que l'algorithme de calcul utilise un nombre fini de termes,

$$\frac{\pi}{4} = \sum_{i=1}^{nbTermes} (-1)^{i-1} \frac{1}{(2i-1)}$$

où `nbTermes` *est un paramètre donné à l'appel du calcul.*

L'implémentation de l'objet distant doit spécifier une méthode pour réaliser ce calcul et en plus, un constructeur qui aide à l'exportation de l'objet.

```
package calculpi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ApproxPiImpl extends UnicastRemoteObject
implements ApproxPi {
  public ApproxPiImpl() throws RemoteException {
    super();
  }
  public double valPi(int nbTermes) throws RemoteException {
    double pi = 0;
    int i;

    for (i=1; i <= nbTermes; i++)
    pi = pi +  Math.pow(-1, i-1)/(2*i-1);
    return 4 * pi;
  }
}
```

Le comportement de l'objet distant peut être défini dans une classe indépendante, comme dans l'exemple, mais il est possible également de le définir directement dans la classe du serveur. La séparation proposée dans cette solution est la plus convenable en général.

Définition du serveur *La tâche du serveur consiste à créer un objet distant et à l'enregistrer auprès du serveur de noms afin de publier sa référence et la rendre accessible aux clients.*

```
package calculpi;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;

public class PiServer {
  public static void main(String[] args) {
    try {
      ApproxPiImpl objPi = new ApproxPiImpl();
      Naming.rebind("Pi", objPi);
    } catch(RemoteException e) {
      System.out.println("PiServer exception " +
      e.getMessage());
    } catch(MalformedURLException e) {
      System.out.println("PiServer : url malformed " +
      e.getMessage());
    }
  }
}
```

Définition du client *L'application cliente demande le service d'approximation du nombre π auprès du serveur. Afin d'obtenir cette approximation, le client doit tout d'abord chercher la référence de l'objet distant capable de lui rendre le service, et ensuite il peut invoquer la méthode de calcul de l'objet et récupérer le résultat.*

```
package calculpi;
```

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class PiClient {
  public static void main(String[] args) {
    String url = args[0];
    int nbTermes = Integer.parseInt(args[1]);
    try {
      ApproxPi objPi;
      objPi = (ApproxPi)Naming.lookup("//" + url + "/Pi");
      System.out.println("Approx pi = " + objPi.valPi(nbTermes));
    } catch(RemoteException e) {
      System.out.println("PiClient exception " +
      e.getMessage());
    } catch(NotBoundException e) {
      System.out.println("PiClient : objet not bound " +
      e.getMessage());
    } catch(MalformedURLException e) {
      System.out.println("PiClient : url malformed " +
      e.getMessage());
    }
  }
}
```

L'adresse du serveur de noms (le nom de la machine du lancement suivi du numéro de port de l'écoute des requêtes) est donnée en paramètre dans la ligne de commande. Le deuxième argument de la ligne de commande indique le nombre de termes du développement mathématique de la formule de Leibniz.

Déploiement de l'application distribuée Pi *Toutes les classes développées doivent être compilées grâce à l'outil* `javac`. *De plus, des souches seront générées pour l'implémentation de l'objet distant,* `calculpi.ApproxPiImpl`, *grâce à l'outil* `rmic`.

```
javac -d . *.javaApproxPi.java ApproxPiImpl.java PiClient.java
  PiServer.java
rmic -d . calculpi.ApproxPiImpl
```

Le lancement du serveur est précédé par le lancement du serveur de noms :

```
rmiregistry
java calculpi.PiServer
```

Le client doit être lancé avec deux arguments :

- *le premier est une partie de l'url utilisé pour accéder au bon serveur de noms (nom de machine :numéro de port) ;*
- *le deuxième est le nombre de termes pour le calcul de l'approximation de π selon la formule de Leibniz.*

```
java calculpi.PiClient localhost:1099 2000000000
```

L'exemple précédent montre l'exécution au cas où la machine locale sert en même temps de client et de serveur.

Le résultat de cette exécution est l'affichage chez le client du message :

```
Approx pi = 3.141592658505181
```

Le nombre d'itérations choisi permet d'approximer exactement 8 décimales du nombre π. La valeur de π en Java est celle donnée par la constante `Math.PI`, 3.141592653589793.

Le code source correspondant à cet execice est donné dans le répertoire `ex08`.

Exercice 9 : Encore une autre méthode d'approximation du nombre Pi

Sachez qu'aujourd'hui, il est connu 1 241 100 000 000 décimales de π ; c'est un japonais, Yasumasa Kanada, qui détient le record. Les méthodes d'approximation ont considérablement évolué. On ne s'intéresse pas à battre le record, mais à réaliser une comparaison entre des approximations de π calculées par deux méthodes : une qui calcule l'approximation à base de la formule de Leibniz, et une autre à base de la série Euler, définie de la manière suivante :

$$\frac{\pi}{2} = 1 + \sum_{n=1}^{\infty} \frac{1 \times 2 \times \cdots \times n}{1 \times 3 \times 5 \times \cdots \times (2n+1)}$$

Question 9.1

Développer un schéma d'application distribuée permettant, à partir d'un même ordre de développement des formules, de préciser laquelle des deux méthodes (Leibniz ou Euler), est plus exacte par rapport au référentiel donné par Java (`Math.PI`).

Correction de la question 9.1

L'intérêt de cette application est de concevoir une application distribuée permettant de rendre deux services, les deux méthodes d'approximation du nombre π. L'énoncé ne précise pas d'information particulière sur le nombre d'objets distants de l'application. Deux solutions sont envisageables :

- *soit un seul objet distant est disponible d'offrir les deux services ;*
- *soit un objet distant connaît une seule formule de calcul, donc on aura deux objets distants créés.*

Un seul objet distant *Si un même objet fournit les services de calcul du nombre π selon les deux formules, l'interface de l'objet doit contenir deux méthodes, une réalisant l'approximation selon la formule de Leibniz, une autre selon la formule d'Euler :*

```
package methodepi;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ApproxPi extends Remote {
  public double valPiLeibniz(int nbTermes) throws RemoteException;
  public double valPiEuler(int nbTermes) throws RemoteException;
}
```

L'implémentation spécifie donc le code pour les deux méthodes de l'interface de l'objet distant :

```
package methodepi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ApproxPiImpl extends UnicastRemoteObject
implements ApproxPi {
  public ApproxPiImpl() throws RemoteException {
    super();
  }
  public double valPiLeibniz(int nbTermes) throws RemoteException {
    double pi = 0;
    int i;

    for (i=1; i <= nbTermes; i++)
    pi = pi + Math.pow(-1, i-1)/(2*i-1);
```

```
    return 4 * pi;
  }

  public double valPiEuler(int nbTermes) throws RemoteException {
    double pi = 0;
    int i;
    double terme = 1.0;
    for (i=1; i <= nbTermes; i++){
      pi = pi + terme;
      terme = terme * i/(2*i+1);
    }
    return 2 * pi;
  }
}
```

Le code du serveur reste exactement le même, tandis que le code du client change uniquement lors de l'invocation de la méthode à distance, qu'elle soit `valPiLeibniz` *ou* `valPiEuler`*.*

Deux objets distants *Une deuxième solution consiste à considérer deux objets distants différents, un qui permet de calculer l'approximation selon la formule de Leibniz, et l'autre selon la formule d'Euler.*

Dans ce cas, deux serveurs s'exécuteront sur deux machines différentes (si un seul serveur instancie les deux objets, cela signifie qu'il connaît les deux implémentations, donc la première solution qui crée un seul objet serait préférable). L'interface de l'objet distant est la même que pour l'exercice précédent (une seule méthode permettant de rendre une approximation du nombre π*), par contre deux implémentations de l'objet distant seront créées. Etant donné qu'elles sont sur des machines différentes, elles peuvent porter le même nom.*

Donc sur une machine `m1`*, on peut avoir :*

```
package calculpi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ApproxPiImpl extends UnicastRemoteObject
implements ApproxPi {
  public ApproxPiImpl() throws RemoteException {
    super();
  }
  public double valPi(int nbTermes) throws RemoteException {
    double pi = 0;
    int i;

    for (i=1; i <= nbTermes; i++)
    pi = pi +  Math.pow(-1, i-1)/(2*i-1);
    return 4 * pi;
  }
}
```

et sur une autre machine `m2`*, on aura :*

```
package calculpi;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ApproxPiImpl extends UnicastRemoteObject
implements ApproxPi {
  public ApproxPiImpl() throws RemoteException {
    super();
  }

  public double valPi(int nbTermes) throws RemoteException {
    double pi = 0;
```

```
        int i;
        double terme = 1.0;
        for (i=1; i <= nbTermes; i++){
          pi = pi + terme;
          terme = terme * i/(2*i+1);
        }
        return 2 * pi;
      }    }
```

Les codes du serveur et du client restent aussi identiques aux codes serveur et client de l'exercice précédent. Cette fois-ci c'est l'exécution des clients qui est modifiée : sur la machine cliente, si on veut avoir le résultat de l'estimation selon la formule Leibniz, la ligne de commande sera :

```
java calculpi.PiClient m1:1099 1000
```

et si l'estimation est réalisée selon la formule Euler, la ligne de commande sera :

```
java calculpi.PiClient m2:1099 1000
```

En fonction de l'adresse passée à l'exécution du client, l'objet distant sera recherché dans le serveur de noms soit de la machine `m1` soit de la machine `m2`. Étant donné que dans le serveur de noms s'exécutant sur chacune de ces machines est enregistré un objet d'implémentation différente, le résultat dépendra de la machine spécifiée à l'exécution du client.

Exercice 10 : Boîte à messages électroniques

Nous souhaitons dans cet exercice réaliser une application permettant de gérer, à distance, une boîte à messages électroniques envoyés de la part de plusieurs clients. Le fonctionnement du serveur de boîte à messages est le suivant :

- Avant tout dépôt de message de la part d'un client, le client doit s'inscrire auprès du serveur. Le serveur lui retourne un numéro unique qui lui permettra de s'identifier auprès du serveur lors de l'envoi des messages.
- Un client accompagne le dépôt de son message du numéro d'inscription reçu de la part du serveur. Le message contient, à part le corps du message proprement dit, le nom du client et la date de l'envoi.
- Le serveur accepte le dépôt d'un message d'un client, en conservant uniquement son dernier message déposé.
- Un client peut consulter le message d'un client donné (lui-même ou un autre) : le serveur lui renvoie le dernier message du client ; un client est identifié par son numéro d'inscription.

Question 10.1

Concevoir un schéma de mise en œuvre de la boîte à messages. Représenter sur un schéma les deux acteurs : le serveur et le client. Représenter par des flèches les invocations de méthodes mises en jeu lorsqu'un client souhaite envoyer un message au serveur de boîte à messages.

Correction de la question 10.1

La boîte à messages a la fonction d'un forum de messages, où un ensemble de participants peut déposer et consulter des messages. Le forum est donc une ressource partagée par plusieurs potentiels clients. Ainsi, la boîte à messages est un objet accessible à distance. Les clients peuvent consulter et déposer des messages uniquement s'ils s'inscrivent à ce service d'accès aux messages. L'inscription donne une information à la boîte à messages sur l'identité de chaque client et constitue la première opération qu'un client doit réaliser (opération 1 dans la figure 2.2). À la réception de son identifiant de la part de la boîte à messages, le client peut envoyer (opération 3) le message construit (opération 2) accompagné de son identifiant, afin que la boîte à messages puisse répertorier correctement le message venant du client.

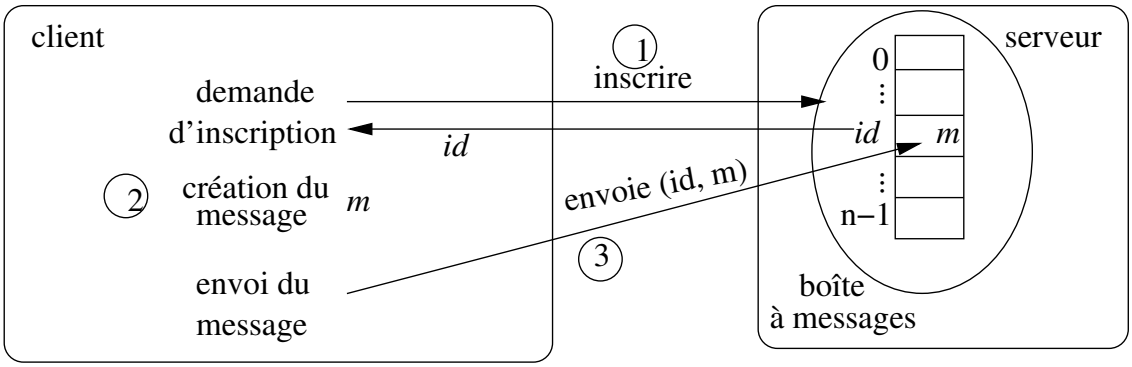


FIGURE 2.2 – Schéma de communication entre le client et la boîte à messages

Question 10.2

Écrire une classe `Message` qui contient le nom de l'émetteur, la date de l'envoi et un texte de message.

Correction de la question 10.2

Le message envoyé par le client est un paramètre à la méthode distante d'envoi, appliquée à l'objet distant boîte à messages. Tout objet passé en paramètre à une méthode à distance doit être sérialisable. L'objet `Message` *doit donc être déclaré dans une classe implémentant l'interface* `java.io.Serializable`*.*

La méthode à prévoir sur l'objet de type message est celle imposée par les opérations du cahier des charges, et notamment la possibilité de récupérer un message qui peut être donné sous la forme : «Le `date`*,* `nom` *a ecrit :* `message`*».*

D'autres méthodes peuvent être prévues pour avoir accès en lecture et modification aux attributs privés de la classe. Celles-ci ne sont pas illustrées ici afin de ne pas encombrer le code.

```
package boite;
import java.util.Calendar;
import java.io.Serializable;

public class Message implements Serializable {
  protected String nom;
  protected String message;
  protected Calendar date;

  public Message(String nom, String message, Calendar date) {
    this.nom = nom;
    this.message = message;
    this.date = date;
  }

  public String getCorpsMessage() {
    return "Le " + date.getTime() + " " + nom
    + " a ecrit : " + message;
  }
}
```

Question 10.3

Écrire un serveur de boîte à lettres pour qu'il permette l'inscription d'un client, l'envoi et la consultation des messages.

Correction de la question 10.3

Le principe de l'application nécessite un objet distant qui gère la boîte à messages. Cet objet sera créé et enregistré auprès du serveur de noms par une application serveur.

- *Le client qui veut déposer un message dans la boîte à messages contacte d'abord le serveur pour demander une inscription (méthode `inscrire`). Le serveur lui renvoie un numéro d'inscription (un entier, unique pour chaque client).*
- *Lorsque le client détient son numéro d'inscription auprès de la boîte à messages, il peut alors envoyer des messages, étiquettés par le numéro (méthode `envoieMessage`).*
- *Un client peut récupérer le dernier message envoyé par un client donné, lui-même inclus (méthode `getMessage`).*

L'interface de l'objet distant s'écrit de la forme suivante :

```
package boite;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BoiteMessages extends Remote {
  public int inscrire() throws RemoteException;
  public void envoieMessage(int client, Message mess) throws RemoteException;
  public Message getCorpsMessage(int client) throws RemoteException;
}
```

L'implémentation de l'objet distant utilise un vecteur d'objets pour stocker les derniers messages de chacun des clients (l'attribut `tabMess`), un entier permettant de compter le nombre de clients courants, donc attribuer un identifiant unique à tout client venant de s'inscrire (l'attribut statique `nbClients`).

```
package boite;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.util.Vector;

public class BoiteMessagesImpl extends UnicastRemoteObject
implements BoiteMessages {
  protected static int nbclients;
  protected Vector tabMess;

  public BoiteMessagesImpl() throws RemoteException {
    super();
    nbclients=0;
    tabMess = new Vector(10);
  }

  public int inscrire() throws RemoteException {
    return nbclients++;
  }

  public void envoieMessage(int client, Message mess)
  throws RemoteException {
    tabMess.add(client, mess);
  }

  public Message getMessage(int client) throws RemoteException {
    return (Message)tabMess.get(client);
  }
}
```

Le serveur est chargé de la gestion de la boîte à messages, donc de la création et de l'enregistrement de celle-ci auprès du serveur de noms.

```
package boite;

import java.rmi.Naming;
import java.rmi.RemoteException;
```

```
import java.net.MalformedURLException;

public class BoiteServer {
  public static void main(String[] args) {
    try {
      BoiteMessagesImpl objBoite = new BoiteMessagesImpl();
      Naming.rebind("Boite", objBoite);
    } catch(RemoteException e) {
      System.out.println("BoiteServer exception " +
      e.getMessage());
    } catch(MalformedURLException e) {
      System.out.println("BoiteServer : url malformed " +
      e.getMessage());
    }
  }
}
```

Le client réalise tout d'abord son inscription, suite à laquelle il peut envoyer et consulter les messages.

```
package boite;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import java.util.Calendar;

public class Client {
  public static void main(String[] args) {
    String url = args[0];
    try {
      BoiteMessages boiteMess;
      boiteMess = (BoiteMessages)Naming.lookup("//"+url+"/Boite");

      int nbclient = boiteMess.inscrire();
      Message m = new Message("vio", "hi", Calendar.getInstance());
      boiteMess.envoieMessage (nbclient, m);
      System.out.println("Mon message sur le serveur :\n" +
      boiteMess.getMessage(nbclient).getCorpsMessage());
    } catch(RemoteException e) {
      System.out.println("Client exception " + e.getMessage());
    } catch(NotBoundException e) {
      System.out.println("Client : objet not bound " +
      e.getMessage());
    } catch(MalformedURLException e) {
      System.out.println("Client : url malformed " +
      e.getMessage());
    }
  }
}
```

L'exécution du client (après le lancement du serveur de noms et du serveur) réalise l'affichage suivant :

```
Mon message sur le serveur :
Le Fri Sep 15 12:27:55 CEST 2006 vio a ecrit : hi
```

Question 10.4

Supposons qu'un client qui vient d'envoyer un message se rend compte que la date du message est erronée. Il doit pouvoir modifier la date du message, mais sans le retransmettre. Est-il possible de le faire dans la solution actuelle ? Justifier votre réponse, et en cas négatif, proposer une solution.

Correction de la question 10.4

Pour avoir la possibilité de changer la date d'un message, l'objet de type message doit disposer d'une méthode de mise à jour de la date. Ainsi, la méthode suivante est rajoutée au code de la classe `Message` *:*

```
public void setDate(Calendar date) {
  this.date = date;
}
```

Le code suivant (rajouté dans le client) permet de tester si le changement de date effectué sur l'objet de type message en local, chez le client, engendre le changement de date chez la boîte à messages pour le client en question :

```
System.out.print("Pour modifier la date (appuyer une touche)");
Clavier.readString();
Calendar date = Calendar.getInstance();
System.out.println("Nouvelle date " + date.getTime());
m.setDate(date);

System.out.println(
"Message sur le serveur apres modif date :\n"
+ boiteMess.getMessage(nbclient).getMessage());
```

L'exécution affiche le message suivant :

```
Mon message sur le serveur :
Le Fri Sep 15 12:34:42 CEST 2006 vio a ecrit : hi
Pour modifier la date (appuyer une touche)
Nouvelle date Fri Sep 15 12:34:44 CEST 2006
Message sur le serveur apres modif date :
Le Fri Sep 15 12:34:42 CEST 2006 vio a ecrit : hi
```

Le schéma actuel des classes ne permet donc pas d'actualiser la date d'un message sans le retransmettre au serveur. Si le client réémet le message, celui-ci sera actualisé chez la boîte à messages :

```
boiteMess.envoieMessage(nbclient, m);
System.out.println(
"Mon message sur le serveur apres retransmission :\n"
+ boiteMess.getMessage(nbclient).getCorpsMessage());
```

Le nouveau code du client affichera :

```
Mon message sur le serveur :
Le Fri Sep 15 12:37:09 CEST 2006 vio a ecrit : hi
Pour modifier la date (appuyer une touche)
Nouvelle date Fri Sep 15 12:37:10 CEST 2006
Message sur le serveur apres modif date :
Le Fri Sep 15 12:37:09 CEST 2006 vio a ecrit : hi
Mon message sur le serveur apres retransmission :
Le Fri Sep 15 12:37:10 CEST 2006 vio a ecrit : hi
```

Ce comportement s'explique par le fait que les objets non distants sont passés par copie, donc lors de la transmission d'un message, c'est une copie de l'objet local qui est transmis chez le serveur.

Solution avec un objet message distant *La solution pour pouvoir faire des modifications directement sur l'objet enregistré chez le serveur est de rendre l'objet de type message distant.*

L'interface de l'objet message distant est la suivante :

```
package boite;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Calendar;

public interface MessageRef extends Remote {
  public String getCorpsMessage() throws RemoteException;
  public void setDate(Calendar date) throws RemoteException;
}
```

L'implémentation de l'objet distant message fournit du code pour les méthodes de l'interface distante :

```
package boite;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Calendar;

public class MessageImpl extends UnicastRemoteObject
implements MessageRef {
  protected String nom;
  protected String message;
  protected Calendar date;

  public MessageImpl(String nom, String message, Calendar date)
  throws RemoteException {
    super();
    this.nom = nom;
    this.message = message;
    this.date = date;
  }

  public String getCorpsMessage() throws RemoteException {
    return "Le " + date.getTime() + " " + nom
    + " a ecrit : " + message;
  }

  public void setDate(Calendar date) throws RemoteException {
    this.date = date;
  }
}
```

Les lignes de code de l'interface de l'objet boîte à messages distant qui changent sont les suivantes :

```
public void envoieMessage(int client, MessageRef mess)
throws RemoteException;
public MessageRef getMessage(int client) throws RemoteException;
```

Les lignes de code de l'implémentation de l'objet boîte à messages distant qui changent sont :

```
public void envoieMessage(int client, MessageRef mess)
throws RemoteException {
  tabMess.add(client, mess);
}

public MessageRef getMessage(int client) throws RemoteException {
  return (MessageRef)tabMess.get(client);
}
```

Comme pour tout objet distant, seul le serveur (celui qui le crée), manipule l'implémentation de l'objet, les autres parties de l'application doivent toujours manipuler l'interface de l'objet. Pour cette raison, le paramètre ou le résultat de type message passé pour les méthodes distantes a le type de l'interface, `MessageRef`.

Et dans le client, la création de l'objet message change :

```
MessageRef m = new MessageImpl("vio", "hi",
Calendar.getInstance());
```

Ce problème est un exemple où l'application cliente joue, dans un premier temps, le rôle de client, parce qu'elle fait appel à l'objet boîte à messages pour récupérer sa référence, et dans un deuxième temps, le rôle de serveur, car il crée un objet distant de type message qu'il enregistre, cette fois-ci pas dans un serveur de noms, mais dans la boîte à messages.

Le résultat de l'exécution du client est maintenant :

```
Mon message Le Fri Sep 15 12:54:15 CEST 2006 vio a ecrit : hi
Pour modifier la date (appuyer une touche)
Nouvelle date Fri Sep 15 12:54:17 CEST 2006
Message sur le serveur apres modif date :
Le Fri Sep 15 12:54:17 CEST 2006 vio a ecrit : hi
```

La date du message a été modifiée, sans que le client réémette le message, parce que lors du passage du paramètre de type message, c'est la référence de l'objet (plus exactement son talon) qui est envoyée à la boîte à messages. Donc le serveur détient aussi une référence sur le même objet distant de type message.

Exercice 11 : Téléchargement dynamique de classes en RMI

Question 11.1

Déployer correctement l'application boîte à messages afin de tester le téléchargement dynamique de classes dans le mécanisme d'invocation distante RMI (voir section 3.5 du cours).

Correction de la question 11.1

Pour que le téléchargement de bytecode puisse fonctionner, il faut que le client et le serveur installent chacun un gestionnaire de sécurité, instance de la classe `java.rmi.RMISecurityManager` *:*

```
if (System.getSecurityManager() == null) {
  System.setSecurityManager(new SecurityManager());
}
```

L'application boîte à messages utilise :

- *l'interface de l'objet distant boîte à messages (classe disponible chez le serveur et accessible par le client) ;*
- *l'implémentation de l'objet distant boîte à message (classe disponible chez le serveur) ;*
- *la classe du message à transmettre (classe accessible par le client) ;*
- *l'application cliente (classe disponible chez le client) ;*
- *l'application serveur (classe disponible chez le serveur).*

Le talon de l'objet distant développé du côté serveur doit être rendu accessible chez le client. Cette disponibilité est réalisée grâce au téléchargement dynamique de bytecode.

1. *Le codebase de l'objet distant est spécifié par le serveur grâce à la propriété* `java.rmi.server.codebase`*. Le serveur enregistre l'objet distant dans le serveur de noms, sous un nom donné. Le codebase donné par la JVM du serveur est annoté à la référence de l'objet distant dans le serveur de noms.*
2. *Le client demande la référence d'un objet distant enregistré sous un nom donné. La référence de l'objet distant (du talon de l'objet) lui servira à réaliser les appels de méthodes à distance sur l'objet distant.*

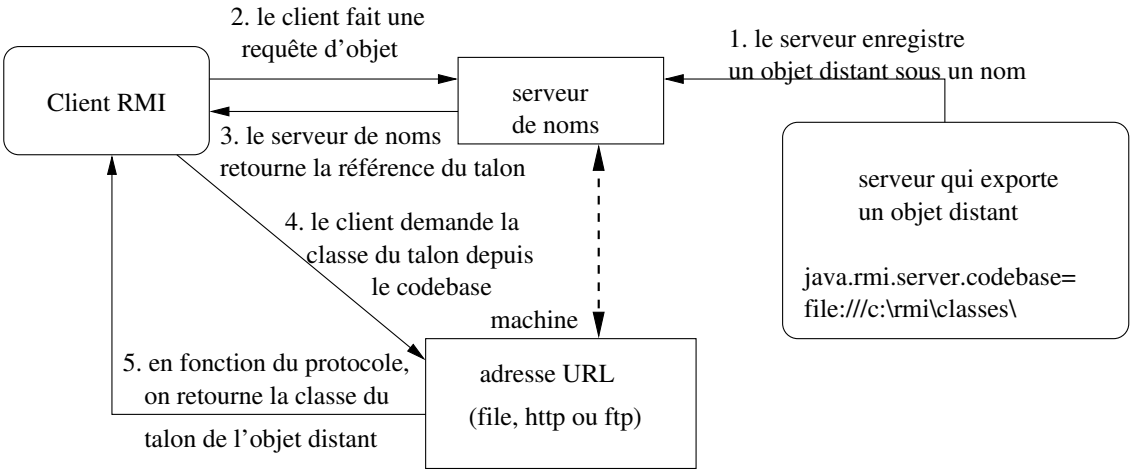


FIGURE 2.3 – Téléchargement dynamique de talons en RMI

3. *Le serveur de noms renvoie une référence (une instance du talon) au client. Si la définition de la classe talon de l'objet distant n'est pas trouvée localement chez le client (dans son CLASSPATH) - recherche réalisée en priorité par rapport à codebase - le client charge cette classe localement. Au contraire, si la définition du talon n'est pas trouvée dans CLASSPATH, le client tente de récupérer la définition de la classe depuis le codebase de l'objet distant.*
4. *Le client demande la définition de la classe depuis le codebase. Le codebase que le client utilise est l'URL annoté dans l'instance du talon quand la classe du talon a été chargée dans le serveur de noms.*
5. *La définition de la classe du talon (et de toutes les autres classes dont il a besoin) sont chargées chez le client.*

Les étapes 4 et 5 sont les mêmes que le serveur de noms utilise quand il charge la classe de l'objet distant, lors de son enregistrement. Quand le serveur de noms essaie de charger la classe talon de l'objet distant, il nécessite la définition de la classe qu'il récupère depuis le codebase associé à l'objet distant.

Le client dispose maintenant de l'information nécessaire pour invoquer des méthodes à distance sur l'objet distant. L'instance du talon joue le rôle de proxy de l'objet distant qui réside sur le serveur.

Le lancement du serveur de noms doit être réalisé tel que celui-ci n'ait pas accès aux classes nécessitant d'être téléchargées. En supposant que les classes à télécharger sont dans le répertoire `/home/csd/rmi/classes` *(et encore dans le sous-répertoire* `boite`*, à cause du paquetage utilisé), le serveur est lancé :*

```
java -Djava.security.policy=java.policy
  -Djava.rmi.server.codebase=file:///home/csd/rmi/classes/
  boite.BoiteServer
```

Dans le répertoire `/home/csd/rmi/classes/boite`*, on doit retrouver les deux classes* `BoiteMessagesImpl`*, le talon de l'objet distant et* `Message`*, classe utilisée par le talon. Le protocole de téléchargement utilisé dans l'exemple est* `file:///`*, si le client et le serveur résident sur la même machine physique. D'autres protocoles (*`ftp` *ou* `http`*) peuvent être utilisés.*

Quand la valeur de la propriété codebase est un URL vers un répertoire, la valeur doit être terminée par "/" .

Le client est lancé comme précédemment, en rajoutant les permissions de téléchargement :

```
java -Djava.security.policy=java.policy
  boite.Client localhost:1099
```

Erreurs d'exécution possibles *Une erreur classique se produit lors du lancement du serveur : la levée de l'exception* `ClassNotFoundException`*. Cette erreur apparaît quand un objet est enregistré dans le serveur de noms, à cause d'une valeur de codebase mal construite, ce qui empêche le serveur de noms de localiser les talons des objets distants ou des autres classes utilisées par le talon.*

L'erreur ressemble à cet affichage :

```
BoiteServer exception RemoteException occurrend in server thread;
  nested exception is :
  java.rmi.UnmarshalException: error unmarshalling arguments;
    nested exception is:
    java.lang.ClassNotFoundException: boite.BoiteMessageImpl_Stub
```

Une autre exécution incorrecte peut apparaître lors du lancement du client, par la levée de la même exception lors de la recherche de la référence d'un objet distant dans le serveur de noms. Cette erreur est causée par la valeur de la variable `CLASSPATH` *lors du lancement du serveur de noms. Cette erreur se ressemble à :*

```
java.rmi.UnmarshalException: Return value class not found;
nested exception is:
  java.lang.ClassNotFoundException: BoiteMessageImpl_Stub
  at sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:109
  at java.rmi.Naming.lookup(Naming.java:60)
  at Client.main(Client.java:19)
```

Le code source de cet exercice est donné dans le répertoire `ex10/qu4`.

Chapitre 3

JMS

Les applications utilisant les JMS sont des programmes Java classiques, composés de classes, regroupées dans un même fichier ou dans des fichiers indépendants. Ces programmes peuvent être codés à partir d'un éditeur de texte ou d'un EDI (Environnement de Développement Intégré).

Pour pouvoir développer, compiler et exécuter les programmes des exercices, il est nécessaire d'installer un bus de messages sur votre machine. Le bus utilisé est Artemis de la fondation Apache. Les instructions ci-dessous sont données pour une installation sur un système Linux, natif ou virtuel. Le bus de message proposé fonctionne cependant également sur les systèmes windows. Il est alors nécessaire d'adapter les commandes données.

Installation et lancement du bus de messages

1. Télécharger une version binaire du bus de messages qui correspond à votre système à partir de cette page.
2. Lancer une fenêtre de commande, décompresser l'archive dans le répertoire où vous souhaitez l'installer à l'aide de la commande :
   `tar zxvf apache-artemis-2.17.0-bin.tar.gz`.
   Dans la suite nous ferons référence au répetoire d'installation sous le nom de *monRep*.
3. Créer une instance de serveur avec les commandes suivantes :
   `cd monRep ; mkdir run_dir ; bin/artemis create run_dir`.
4. Mettre à jour la variable `PATH` pour y ajouter le chemin d'accès à l'exécutable `artemis` : `monRep/run_dir/bin` :
   `export PATH=$PATH:monRep/run_dir/bin`
5. Mettre à jour la variable CLASSPATH pour y ajouter les bibliothèques de JMS et le répertoire courant : `export CLASSPATH="$CLASSPATH:monRep/lib/*:."`
6. Le lancement du bus de messages Artemis se fait avec la commande `artemls run`.
7. La compilation et l'exécution des programmes se fera de manière classique avec les commandes `javac` et `java`.

Il est recommandé de lancer le bus de messages dans une fenêtre à part pour voir plus facilement les traces affichées en cas d'erreur.

Lancement des applications

Les applications JMS utilisent les bibliothèques données par le $CLASSPATH$. Elles peuvent être lancées en ligne de commande à partir d'un shell dont le $CLASSPATH$ est correctement configuré ou depuis un EDI dans lequel l'accès aux bibliothèques JMS est ajouté.

Pour les exercices suivants l'accès au bus de messages (`Connection Factory`) dans les programmes se fait à partir de l'adresse `tcp://localhost:61616`. Vous pouvez observer ce qui

se passe sur le bus de message en vous connectant à la console d'administration à l'adresse `http://http://localhost:8161/admin/`, avec le compte *admin* et le mot de passe *admin*.

Dans les exercices suivants, lorsque la réponse est constituée du code des programmes, celui est donné en correction sur le site du cours.

Exercice 12 : Programmes simples

Question 12.1

À partir des informations dont vous disposez dans le cours écrire un programme *Sender* qui envoie un message de type texte contenant la chaîne *Hello* à un programme *Receiver* qui affiche le contenu du message une fois qu'il l'a reçu.

Question 12.2

Écrire les programmes *Publisher* et *Subscriber* qui échangent la même chaîne à travers un topic.

Question 12.3

Que se passe-t-il si plusieurs programmes cherchent à recevoir en même temps sur la queue ? Que se passe-t-il si plusieurs programmes cherchent à recevoir en même temps sur le même topic ?

Correction de la question 12.3

Si deux programmes se mettent en attente sur une même queue, le premier reçoit le premier message émis, le second reçoit le second message. Le message n'est donc délivré qu'à un seul programme. Sur un topic tous les programmes en attente reçoivent le message.

Exercice 13 : Envoi de lettre

Les messages qui ont été échangés à l'exercice précédent sont de simples textes. L'objectif de cet exercice est de modifier les programmes prédédents pour réaliser l'envoi, sur la queue et sur le topic, d'un objet lettre. Une lettre est un objet caractérisé par un émetteur, un texte et une date.

Question 13.1

Comment réaliser un message contenant les différents champs de la lettre ?

Correction de la question 13.1

Pour envoyer une lettre il faut créer un objet Java qui contient les différents champs et utiliser un `ObjectMessage`*.*

Question 13.2

Réaliser l'envoi d'une lettre entre un émetteur et un destinataire.

Question 13.3

Comment permettre au destinataire de répondre quel que soit l'émetteur (il ne le connaît pas à priori) ?

Correction de la question 13.3

Il faut que l'émetteur dispose lui-même d'une queue et en donner le nom sous la forme d'un message sur la queue initiale.

Question 13.4

Modifier vos programmes pour que le récepteur reçoive le nom d'une nouvelle queue et réponde au message de l'émetteur sur cette queue.

Exercice 14 : Application de chat

On souhaite réaliser une application de chat (discussion en ligne) avec inscription durable : lorsqu'un participant se connecte il récupère les messages qu'il n'a pas encore lu. Pour cela, on utilise un seul topic sujet qui reçoit et diffuse des `TextMessage`.

Question 14.1

Au début l'application est découpée en deux programmes : l'émetteur et le récepteur. L'émetteur est appelé à chaque fois qu'un message doit être transmis, tandis que le récepteur s'exécute de façon continue. Dans l'émeteur le texte est saisi à l'aide de code suivant :

```
   java.util.Scanner in = new java.util.Scanner(System.in);
String input = in.nextLine();
```

Question 14.2

Modifier le récepteur pour qu'il réalise une inscription durable. L'identificateur du récepteur sera passé en paramètre de son lancement. Vérifier la durabilité de l'inscription en interrompant le récepteur, puis en le relançant après avoir émis quelques messages. Le récepteur devrait alors afficher les messages reçus entre temps.

Question 14.3

Modifier l'émetteur en intégrant la saisie des messages dans une boucle afin d'avoir deux programmes tournant en continu : l'émetteur et le récepteur. Puis fusionner les deux applications avec une réception asynchrone des messages (listener) pour que les réceptions aient lieu même lorsque l'application attend la saisie de l'utilisateur.

Exercice 16 : Mode transactionnel

Le but de cet exercice est de vous faire tester le mode transactionnel dans JMS. Reprendre l'exemple de la queue et les programmes `Sender` et `Receiver`.

Question 16.1

Ne définir aucun mode transactionnel, ni chez le `Sender`, ni chez le `Receiver`. Introduire des temporisations entre les envois de messages à l'aide de la commande `Thread.sleep(3000)` entre chaque envoi et chaque réception. Simuler ensuite une panne en arrêtant l'un ou l'autre des programmes, puis relancer les programmes pour qu'ils finissent l'exécution commencée. Que notez-vous ?

Question 16.2

Modifier les programmes `Sender` et `Reveiver` pour qu'ils travaillent en mode transactionnel. Recommencer la simulation des pannes chez l'un ou l'autre des programmes comme à la question précédente. Quelle différence notez-vous ?

Question 16.3

Les objets `Session` permettent de valider une transaction par la fonction `commit`. Que se passe-t-il si vous ajoutez un appel à la fonction `commit` après l'envoi des messages ? Que se passe-t-il si vous ajoutez un appel à la fonction `commit` après la réception des messages ?

Question 16.4

Que se passe-t-il pour le `Receiver` si le `Sender` envoie des messages sans commiter, ferme puis réouvre la session, renvoye les mêmes messages puis commite.

Question 16.5

Que se passe-t-il si le `Receiver` reçoit des messages sans commiter, ferme puis réouvre la session, redemande à recevoir des messages puis commite.

Question 16.6

Les objets `Session` permettent d'invalider une transaction par la fonction `rollback`. Que se passe-t-il si, après avoir envoyé des messages le `Sender` fait appel à la fonction `rollback` ?

Question 16.7

Que se passe-t-il si, après avoir reçu des messages, le `Receiver` fait appel à la fonction `rollback` ?

Question 16.8

Faire le même exercice avec les programmes `Publisher` et `Subscriber`. Qu'elle différence notez vous ?

Remarque : il est recommandé de suivre l'exécution des programmes `Sender` et `Receiver` à l'aide de l'outil d'administration pour mieux voir ce qui se passe.

Exercice 17 : Boîtes aux lettres

Cet exercice consiste à mettre en place des boîtes aux lettres sur la base des objets `Sender`, `Receiver` et `Letter` que vous avez utilisés pour le second exercice de JMS puis de diffuser une annonce de création de boîte aux lettres sur un topic afin que le client puisse recevoir des messages.

Question 17.1

Modifier le programme `Receiver` pour qu'il créé une boîte aux lettres (queue) et diffuse le nom de cette boîte sur un topic avant de se mettre en attente de réception des messages.

Question 17.2

Modifier le programme `Sender` pour qu'il attende l'annonce de création de la queue avant d'envoyer un message au client.

Question 17.3

Modifier le programme `Receiver` pour qu'il reçoive les messages de manière asynchrone. A la réception d'un message le listener doit en tenir informé le programme principal qui compte les messages reçus sur la boîte aux lettres.

Question 17.4

Mettre en place un sélecteur pour ne recevoir que les messages dont le type (propriété ou JMSType) est `important`.

Chapitre 4

Services Web

Pour la partie des Services WEB nous recommandons l'utilisation d'un EDI pour faciliter les phases de développement, de déploiement et d'exécution.

Attention, à partir de la version 9 de Java, une part des classes existantes jusque là a été mise en "deprecated" puis supprimée de la version 11. Pour les utiliser il est alors nécessaire d'ajouter les packages manquants, ce qui est pénible et risqué en terme de stabilité du code. Il est donc recommandé pour les nouvelles versions de Java d'utiliser un outil de compilation comme maven.

Les paquets support des web-services font partie des paquets supprimés. Dans les corrections nous donnons, dans les fichiers `README.txt`, une solution "classique" avec le jdk1.8 et une version adaptée au jdk12 avec maven.

Exercice 18 : Questions du cours

Question 18.1

Qu'est ce que les services Web SOAP ?

Correction de la question 18.1

Les services Web de type SOAP permettent l'appel de méthodes distantes (services) dans un environnement hétérogène et distribué. Ils reposent sur un ensemble de protocoles et de standards pour la communication et l'échange de données.

Question 18.2

Quels sont les principaux avantages des services Web SOAP ?

Correction de la question 18.2

Les avantages des services Web SOAP sont les suivants :
- *L'interopérabilité,*
- *L'utilisation de normes et standards,*
- *Couplage faible entre applications,*
- *Flexibilité et extensibilité.*

Question 18.3

Peut-on faire communiquer une application Java avec une application Python en SOAP ? Si oui, pourquoi et comment ?

Correction de la question 18.3

Il est possible de réaliser la communication entre des applications écrites dans des langages différents. Les deux applications sont toutes deux basées sur le protocole SOAP. Les données échangées, avec le protocole HTTP, sont représentés en XML.

Question 18.4

Quel est le rôle du protocole WSDL ?

Correction de la question 18.4

WSDL est un contrat entre le client le fournisseur de services, il permet de décrire, en XML, pour les applications clientes, des interfaces d'accès à des services distantes (les méthodes et leurs paramètres par exemple).

Exercice 19 : HelloWorld avec WS-SOAP

Le but de cet exercice est de vous faire tester un simple service Web en utilisant JAX-WS. Pour cela reprendre le code source, HelloWorld, vu dans le cours.

Question 19.1

Compiler et déployer le service Web

Question 19.2

Analyser le WSDL généré en accédant à l'adresse du serveur.

Question 19.3

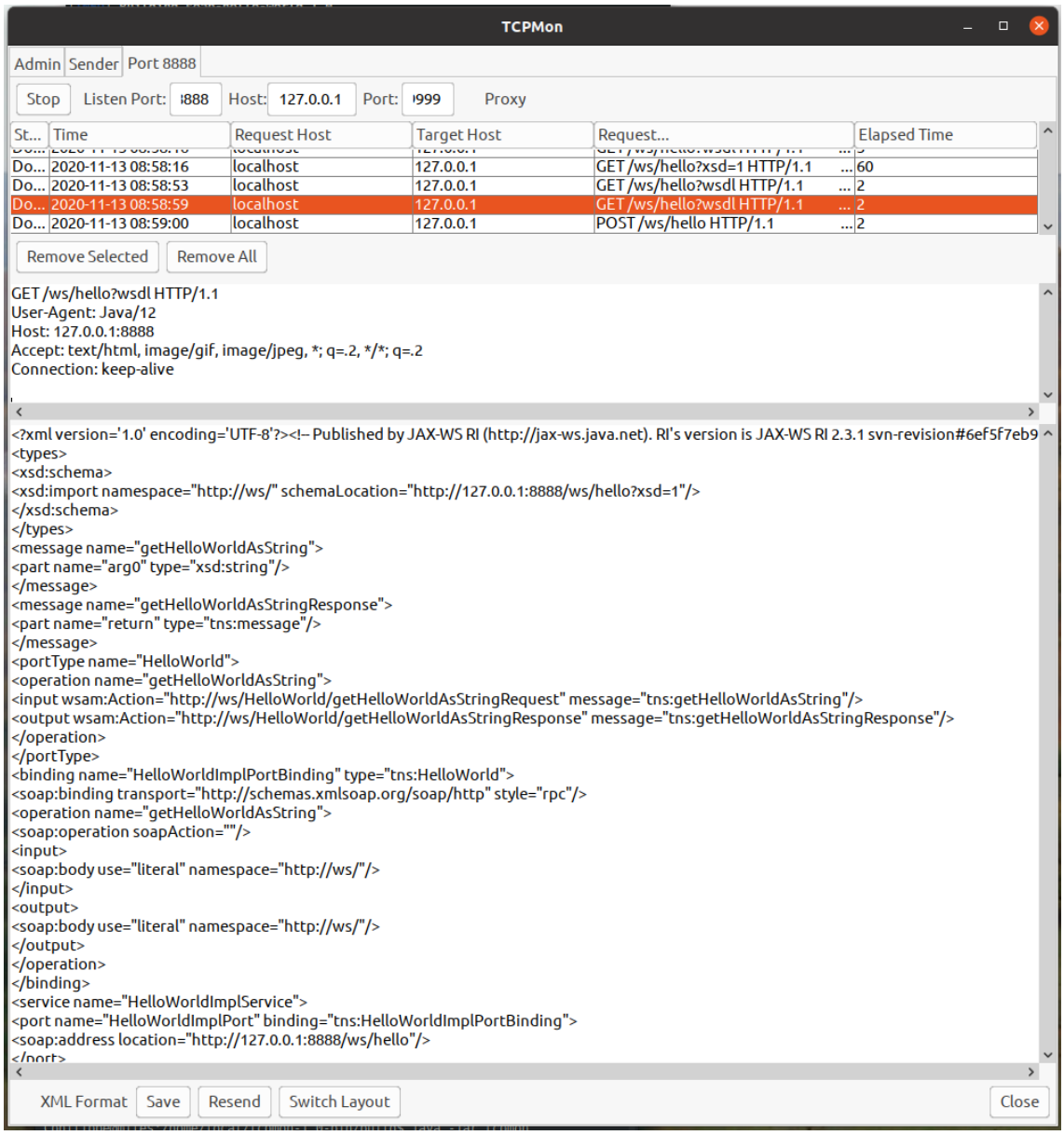
Utiliser le logiciel TCPMon pour analyser les messages SOAP échangés.

Une archive du logiciel TCPMON est disponible sur le moodle du cours dans la partie exercice. Pour l'utiliser vous devez decompresser l'archive et utiliser le lanceur (`tcpmon.sh` sous linux).

Dans l'interface, ouvrir le TAB admin et compléter : 8888 pour le `Listen port`, cocher la case `Listener` et donner l'adresse du web service (`localhost` et 9999), puis cliqer sur `add`. Un tab marqué `Port 8888` apparaît sur lequel vous devez vous placer.

Ensuite modifier le client pour qu'il s'adresse au port 8888 (voir fichiers README.txt), le compiler et l'exécuter. Regarder le contenu des messages échangés.

Correction de la question 19.3



Question 19.4

Remplacer Style.RPC par Style.DOCUMENT. Quelles différences notez-vous au niveau du WSDL généré et des messages SOAP ?

Question 19.5

Modifier le code du serveur pour que le serveur retourne un objet Java `Message` contenant une chaîne de caractère et un entier

Question 19.6

Analyser de nouveau le WSDL généré et les messages SOAP échangés

Question 19.7

Écrire le code python du client

Exercice 20 : Serveur d'images avec WS-SOAP

Le but de cet exercice est de mettre en place un service Web pour télécharger des images depuis un serveur. Le client demande une image en spécifiant son nom, le serveur retourne l'image.

Nous utilisons les classes : `java.awt.Image` et `javax.imageio.ImageIO` pour la manipulation des images.

Par exemple, pour lire une image en Java :

```
File file = new File("/home/kmazouzi/Images/" + name);
Image image ImageIO.read(file);
```

Le client peut afficher l'image en se basant sur le code suivant :

```
...
JFrame frame = new JFrame();
frame.setSize(300, 300);
JLabel label = new JLabel(new ImageIcon(image));
frame.add(label);
frame.setVisible(true);
```

Question 20.1

Proposer une interface pour ce service Web

Question 20.2

Comment sont transmises les données non XML ?

Correction de la question 20.2

Les données non XML sont transmises en d'attachement se forme de MIME (données binaires)

Question 20.3

Écrire le code du serveur et du client

Question 20.4

Que-se passe-t-il lorsque le client demande une image non disponible sur le serveur ?

Correction de la question 20.4

Le serveur déclenche une exception, il renvoi une erreur se forme de message SOAP avec un attribut FAULT.

Question 20.5

Analyser les messages SOAP avec TCPMon

Exercice 21 : Serveur de Météo avec WS-SOAP

Examiner la description WSDL du service : https://www.dataaccess.com/webservicesserver/NumberConversion.wso?wsdl

Question 21.1

Que propose ce service Web ?

Correction de la question 21.1

Ce service Web permet fr convertir des valeurs numériques en chaîne de caractères.

Question 21.2

Quels sont les noms des méthodes proposées par ce service ?

Correction de la question 21.2

NumberToWords et NumberToDollars

Question 21.3

Quels sont les paramètres des méthodes proposées ?

Correction de la question 21.3

NumberToWords : unsignedLong et NumberToDollars ; decimal

Question 21.4

Écrire le code du client

Correction de la question 21.4

```
from suds.client import Client

url= "https://www.dataaccess.com/webservicesserver/NumberConversion.wso?wsdl''

client = Client(url)

#pour voir l'objet client
print(client)

response = client.service.NumberToWords(22)
print(response)

response = client.service.NumberToDollars (130.8)
print(response)
```

Exercice 22 : Serveur de contacts avec WS-REST

Réaliser un serveur de contacts en utilisant l'architecture des Web Services REST. Les contacts sont identifiés par un identificateur et chaque contact contient le nom et l'adresse email d'une personne. La gestion des contacts et leur lecture sera faite à l'aide des opérations CRUD.

Question 22.1

Réaliser le serveur. Une fois lancé vous utiliserez la commande `curl` pour l'interroger.

Correction de la question 22.1

Le code d'implémentation est donné.