Cahier de Travaux Dirigés et Pratiques

Construction d'Applications Réparties (CAR)

Master Sciences et Technologies

Mention Informatique – M1

2013

Équipe enseignante
Damien Cassou
Laurence Duchien
Martin Monperrus
Daniel Romero
Romain Rouvoy
Lionel Seinturier

Calendrier

Séances	Cours	Sujets TD	Sujets TP
Semaine 1	Introduction aux applications réparties		
Semaine 2	Applications réparties en mode message	TD 1 : Programmation client/serveur en mode message	
Semaine 3	Applications réparties en mode message	TD 2 : Serveur FTP	TP 1 : Serveur FTP
Semaine 4	Web Services	TD 3 : Architecture d'applications Internet	TP 1 : Serveur FTP
Semaine 5	Client/serveur et objets	TD 4 : Représentation des données	TP 1 : Serveur FTP
Semaine 6	Java RMI	TD 5 : RMI – Élection dans un anneau	**Évaluation TP 1** TP 2 : Passerelle REST
Semaine 7	Java RMI	TD 6 : RMI – Élection dans un anneau	TP 2 : Passerelle REST
Semaine 8	CORBA	TD 7 : RMI – Annuaire	TP 2 : Passerelle REST
Semaine 9	Java EE – JSP-servlet	TD 8 : RMI – Instanciation à la demande	**Évaluation TP 2** TP 3 : RMI – Transfert de données
Semaine 10	Java EE – EJB	TD 9 : CORBA – Répartiteur de charge	TP 3 : RMI – Transfert de données
Semaine 11	Java EE – EJB	TD 10 : CORBA – Protocole de validation à deux phases	**Évaluation TP 3** TP 4 : Java EE
Semaine 12	Composants pour la répartition	TD 11 : Révision	TP 4 : Java EE
Semaine 13		TD 12 : Révision	TP 4 : Java EE **Évaluation TP 4**

Les quatre TP sont à rendre selon les modalités qui vous seront indiquées. **Ils compteront dans la note de contrôle continu**.

TD 1 – Programmation client/serveur en mode message

L'objectif du TD est d'étudier la définition de protocoles de communication client/serveur pour la mise en œuvre d'applications réparties.

1. Définitions

Donner les définitions des concepts suivants : protocole, pile de protocoles, entrée/sortie bloquante, entrée/sortie non bloquante, mode de communication par envoi de message, mode de communication requête/réponse.

2. Serveur calculette

En utilisant un protocole de transport fiable (TCP), on souhaite réaliser un serveur qui implante les fonctionnalités d'une calculatrice élémentaire fournissant quatre opérations (`+ - * /`). Des clients doivent donc pouvoir se connecter à distance et demander au serveur la réalisation d'une opération. Le serveur leur répond en fournissant le résultat ou un compte rendu d'erreur.

1. Définir un protocole applicatif (i.e. un ensemble de messages et leur enchaînement) aussi simple que possible implantant cette spécification.

2. Recenser les cas d'erreur pouvant se produire avec ce protocole applicatif et proposer un comportement à adopter lorsque ces erreurs surviennent (on ne s'intéresse ici ni aux erreurs de transport du style message perdu, ni aux pannes de machines).

3. En utilisant les primitives fournies ci-dessous donner le pseudo-code du serveur et le pseudo-code d'un client demandant la réalisation de l'opération `8.6*1.75`.

- `ouvrirCx(serveur)` demande d'ouverture de connexion vers serveur
- `attendreCx()` attente bloquante et acceptation d'une demande de connexion
- `envoyer(données)` envoi de données
- `recevoir()` attente bloquante et réception de données
- `fermerCx()` fermeture de connexion

4. Le serveur est-il avec ou sans état ?

5. Le protocole est-il avec ou sans état ?

6. On souhaite que plusieurs clients puissent se connecter simultanément sur le serveur. Cela pose-t-il un problème particulier ? Modifier le pseudo-code du serveur pour prendre en compte ce cas.

7. On souhaite maintenant que les clients puissent enchaîner plusieurs demandes d'opérations au sein d'une même connexion. Proposer deux solutions pour cela (une en modifiant le protocole précédent et une sans modification du protocole).

TD 2/TP 1 – Serveur FTP

1. API *socket* Java

Le but de cet exercice est d'implanter une partie du protocole FTP avec l'API *socket* du langage Java. Ce TD sert de préparation au TP 1.

1. Quelle méthode Java permet d'attendre l'arrivée de demandes de connexions sur un port TCP ?

2. Comment lire des données sur une *socket* en Java ?

3. Comment écrire des données sur une *socket* en Java ?

2. Le protocole FTP

Le protocole FTP permet le transfert de fichiers d'une machine vers une autre machine. Le protocole date de 1971, mais est resté très populaire. Il est décrit dans le RFC 959.

Dans une session classique, l'utilisateur est devant une machine (la machine locale) et souhaite transférer des fichiers vers ou à partir d'une machine distante. L'utilisateur interagit alors avec FTP via un programme FTP. L'utilisateur fournit le nom de la machine sur laquelle il souhaite se connecter, le processus client FTP établit une connexion TCP avec le processus serveur FTP. Pour accéder à un compte distant, l'utilisateur doit s'authentifier à l'aide d'un nom et d'un mot de passe. Après avoir fourni ces informations d'authentification, l'utilisateur peut transférer des fichiers de sa machine vers la machine distante et vice-versa.

Les commandes pour le client sont les suivantes :

`USER username` : utilisé pour l'authentification de l'utilisateur
`PASS password` : utilisé pour le mot de passe de l'utilisateur
`LIST` : permet à l'utilisateur de demander l'envoi de la liste des fichiers du répertoire courant
`RETR filename` : utilisé pour prendre un fichier du répertoire distant et le déposer dans le répertoire local
`STOR filename` : utilisé pour déposer un fichier venant du répertoire local dans le répertoire distant
`QUIT` : permet à l'utilisateur de terminer la session FTP en cours

Chaque commande est suivie par une réponse envoyée du serveur vers le client. Les réponses contiennent un nombre, sur trois positions, suivi d'un message optionnel. Cette structure de réponse est similaire à la structure des codes de réponse du protocole HTTP.

Les réponses sont des chaînes de caractères au format suivant : trois chiffres suivis d'un espace suivi d'un message. Les trois chiffres représentent le code de retour (commande effectuée ou code erreur).

1. Écrire le pseudo-code d'un serveur FTP traitant les commandes citées ci-dessus.

Implantation en Java

Votre serveur doit pouvoir fonctionner avec un client FTP (FileZilla, commande `ftp`, ou tout autre client FTP respectant la RFC).

2. Écrire le code Java du serveur FTP en respectant les noms de classes suivantes.

- Une classe `Serveur` avec une méthode `main`
  - écoutant les demandes de connexion sur un port TCP > 1023
  - donnant accès aux fichiers présents dans un répertoire du système de fichier. La valeur de ce répertoire est précisée et initialisée par une valeur passée en argument au moment du lancement du serveur FTP.
  - délégant à l'aide d'un thread le traitement d'une requête entrante à un objet de la classe `FtpRequest`
- Une classe `FtpRequest` comportant
  - une méthode `processRequest` effectuant des traitements généraux concernant une requête entrante et déléguant le traitement des commandes
  - une méthode `processUSER` se chargeant de traiter la commande `USER`
  - une méthode `processPASS` se chargeant de traiter la commande `PASS`
  - une méthode `processRETR` se chargeant de traiter la commande `RETR`
  - une méthode `processSTOR` se chargeant de traiter la commande `STOR`
  - une méthode `processLIST` se chargeant de traiter la commande `LIST`
  - une méthode `processQUIT` se chargeant de traiter la commande `QUIT`

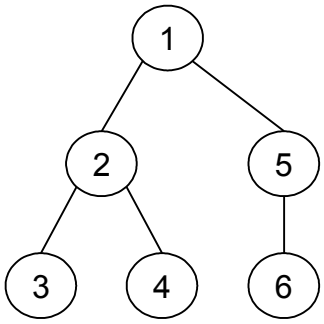
3. Enfin, rajouter les requêtes `PWD, CWD, CDUP`.
`PWD` : permet à l'utilisateur de connaître la valeur du répertoire de travail distant.
`CWD directory` : permet à l'utilisateur de changer de répertoire de travail distant.
`CDUP` : cette commande est équivalente à `CWD ..`

Pour plus d'informations :
http://www.faqs.org/rfcs/rfc959.html
http://www.w3.org/Protocols/rfc959/

TD 3 – Architectures d'applications Internet

1. Diffusion de messages

On considère un protocole client/serveur qui permet de diffuser des messages à un ensemble de sites organisés selon une topologie en arbre. Chaque nœud de l'arbre propage le message à ses fils. Par exemple, dans la figure ci-dessous, pour diffuser son message, le site 1 l'envoie en parallèle à 2 et 5, puis 2 l'envoie en parallèle à 3 et 4 tandis que 5 l'envoie à 6.



1. Quel peut-être l'avantage d'une diffusion en arbre par rapport à une diffusion habituelle (par exemple de type Multicast IP) ?

On considère que les sites communiquent en UDP sur le port 5000 en envoyant des messages de 2048 octets (contenu quelconque), et que chaque site a :
- une variable `pere` qui contient l'adresse de son père dans l'arbre (`null` si pas de père),
- un tableau `fils` qui contient les adresses de ses fils (tableau vide si pas de fils).

2. Écrire en Java le code d'un nœud intermédiaire (dans l'exemple 2 et 5 sont des nœuds intermédiaires).

On suppose maintenant que les nœuds feuille (3, 4 et 6 dans l'exemple) renvoient à l'émetteur un accusé de réception (message de 8 octets contenant des données quelconque) lorsque le message arrive. Lorsqu'un nœud intermédiaire a reçu les accusés de réception de tous ses fils, il renvoie à son propre père (1 dans l'exemple) un accusé de réception (message de 8 octets contenant des données quelconque).

3. Compléter le code Java de la question précédente en y incluant ce comportement.

2. Conception d'un protocole client/serveur et RPC

On s'intéresse à la conception d'un protocole client/serveur de transfert de fichiers. Les serveurs gèrent un ensemble de fichiers. Les clients se connectent sur le serveur pour récupérer un fichier. Le protocole comporte un seul message `TRANSFERT` *`nom`* où *`nom`* est le nom du fichier à transférer.

1. Le protocole est-il avec ou sans état ? Si la réponse est avec état, quel est l'état ? Sinon, pourquoi n'y a-t-il pas d'état ?

2. Plusieurs clients peuvent-ils transférer simultanément le même fichier ? Justifier.

En cas d'interruption du transfert, suite à une panne réseau ou à une panne de serveur, on souhaite ne pas avoir à reprendre le transfert depuis le début lorsque le service redevient opérationnel (pour que les clients ne perdent pas le contenu des fichiers déjà transférés). On se base pour cela sur un mécanisme de « point de reprise majeur ». Tous les 100Ko transférés, le serveur envoie au client un message `PRep` (point de reprise majeur). Un point de reprise majeur est aussi envoyé au début, avant tout transfert de donnée. Chaque point est numéroté à partir de 0. À la réception d'un message `PRep`, le client acquitte la réception en envoyant au serveur un message `PRepAck` pour indiquer au serveur qu'il a bien reçu le point de reprise.

3. Lors d'une demande de transfert de fichier par un client, quelle modification cette fonctionnalité entraîne-t-elle au niveau du protocole ? Que doit indiquer en plus le client ?

4. Lors de l'envoi d'un message `PRep`, le serveur attend d'avoir reçu un acquittement (`PRepAck`) avant de continuer. Citer deux fonctionnalités qu'un tel mécanisme permet de réaliser.

On introduit maintenant un nouveau mécanisme dit « point de reprise mineur ». Entre deux points de reprise majeurs, le serveur envoie un message `PRepMin` (point de reprise mineur) après 50 Ko transférés. Les points de reprise mineurs ne sont pas acquittés (i.e. les clients ne renvoient rien suite à leur réception).

Étude des messages échangés par le protocole client/serveur

On considère les **primitives** suivantes :

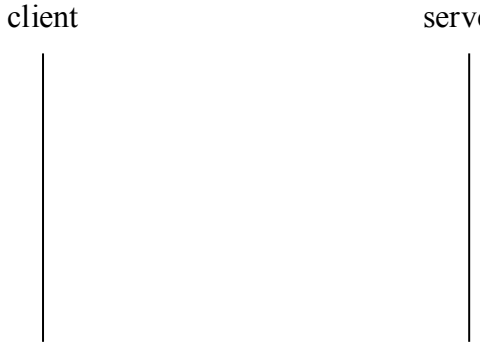
- `ouvrirCx(serveur)` primitive de demande d'ouverture de connexion vers `serveur`
- `attendreCx()` primitive d'attente bloquante et acceptation d'une demande de connexion
- `envoyer(message)` primitive d'envoi d'un message
  `message` peut prendre une des valeurs suivantes :
  - `TRANSFERT` *`nom i j`* demande de transfert du fichier *`nom`*
    message envoyé 1 fois à chaque demande d'un nouveau transfert
    si `j=0` demande de transfert à partir du point de reprise majeur `i`
    si `j=1` demande de transfert à partir du point de reprise mineur suivant le point de reprise majeur `i`
  - `DATA` *`i j`* envoi d'un bloc de données de au plus 50 Ko
    si `j=0` envoi à partir du point de reprise majeur `i`
    si `j=1` envoi à partir du point de reprise mineur suivant le point de reprise majeur `i`
  - `PRep` *`i`* envoi du point de reprise majeur numéro `i`
  - `PRepAck` envoi de l'acquittement d'un point de reprise majeur
  - `PRepMin` envoi du point de reprise mineur
  - `FIN` pour signaler la fin du transfert
- `recevoir()` primitive d'attente bloquante et réception d'un message

(`message = recevoir()`)

- `fermerCx()` fermeture de connexion
- `nbBloc100K(fic)` retourne le nombre de bloc de au plus 100 Ko du fichier `fic` (3 pour un fichier de 225 Ko)

5. Le protocole est-il maintenant avec ou sans état ?

6. Dans le cas particulier où le fichier s'appelle `fic` et contient 225 Ko, indiquer sur un diagramme la séquence de **messages** échangés par le client et le serveur pour un transfert complet du fichier. Les messages sont ceux définis ci-dessus et uniquement ceux-là. On ne demande pas de faire apparaître les primitives sur ce diagramme.



Implantation des services

On se replace dans le cas où la taille du fichier est quelconque. On considère les primitives suivantes :

- `client(fic,i,j)` exécutée par le client pour demander le transfert du fichier `fic` à partir de :
  - si `j=0` du point de reprise majeur `i`
  - si `j=1` du point de reprise mineur suivant le point de reprise majeur `i`
- `serveur()` exécutée par le serveur pour envoyer le fichier au client.

Dans un but de simplification, on suppose que :
- en cas de réception d'un message non attendu, les procédures sont interrompues et une erreur signalée à l'utilisateur,
- on ne s'intéresse pas à la façon dont le client écrit les données du fichier qu'il reçoit.

7. En utilisant les primitives de la question précédente, écrire le pseudo-code ou le code Java des primitives `client(fic,i,j)` et `serveur()`.

TD 4 – Représentation des données

1. Petit boutiste et grand boutiste

Le but de cet exercice est de mettre en lumière les différences de stockage des données entre les formats petit boutiste (*little endian*) et grand boutiste (*big endian*). Ce problème acquiert son importance dès lors que l'on cherche à transférer de l'information entre des clients et des serveurs qui ont fait des choix de représentation de données différents : il faut que le protocole s'assure de la bonne conversion des données.
On considère un flux d'entrée/sortie en mode binaire (i.e. constitué d'octets) dans lequel on veut envoyer des mots de 16 bits. Chaque mot se décompose en octet de poids fort et octet de poids faible (la valeur du mot est 256*octet poids fort + octet poids faible).
- le format petit boutiste consiste à écrire d'abord l'octet de poids faible puis l'octet de poids fort,
- le format grand boutiste consiste à écrire d'abord l'octet de poids fort puis l'octet de poids faible.

Remarque : ces définitions se généralisent lorsqu'il s'agit d'écrire des mots de plus de 16 bits (par exemple 32 ou 64 bits).

1. On considère la chaîne de caractères « hello world » codée de la façon suivante :
- 2 octets pour sa longueur,
- 1 octet par caractère.

Soit A une machine utilisant la convention petit boutiste et B une machine utilisant la convention grand boutiste. Représenter l'ordre de stockage des octets dans les deux cas.

2. Que se passe-t-il si la machine A envoie la chaîne de caractères telle quelle à la machine B ?

3. Dans un deuxième temps, on décide d'inverser l'ordre des octets pour chaque couple d'octets arrivant sur B. Que se passe-t-il ?

2. Mécanisme d'encodage des données en Java

Le but de cet exercice est d'étudier le mécanisme d'encodage des données (*Object Serialization Stream Protocol*) utilisé par Java pour écrire/lire de l'information sur un flux d'entrée/sortie quelconque (fichier binaire, *socket*, tube, etc.). On se reportera pour cela au document reproduit en annexe. On notera de plus que la longueur de codage des entiers en Java est la suivante :

Type	Longueur (en octet)
`byte`	1
`short`	2
`int`	4
`long`	8

Les chaînes de caractères sont représentées selon la norme UTF. De façon simplifiée, pour des chaînes ne comportant que des caractères ASCII non accentués, le codage est le suivant :
- 2 octets pour la longueur
- 1 octet par caractère

1. Décoder la séquence d'octets suivante sachant que 2 entiers `(int)` ont été écrits dans le flux.

```
AC ED 00 05 77 08 00 00 00 11 00 00 00 02
```

Quelle est la valeur de ces entiers ?

2. Décoder la séquence d'octets suivante sachant qu'une chaîne de caractères a été écrite dans le flux.

```
AC ED 00 05 77 07 00 05 48 65 6C 6C 6F
```

3. Chaque bloc de données (`blockdata`) encodé avec ce protocole comporte la taille du bloc (soit sous forme `d'unsigned byte` pour un `blockdatashort`, soit sous forme d'un `int` pour `blockdatalong`). Quels en sont les avantages et les inconvénients ?

4. Coder un objet de la classe Point2D suivante ayant pour valeur de x 18 et pour valeur de y 20.

```
class Point2D implements Serializable {
      private long x;
      private long y;
}
```

Indications :
- le codage ASCII hexadécimal de la chaîne `Point2D` est `50 6F 69 6E 74 32 44`
- le codage ASCII hexadécimal de la chaîne x est `78`
- le codage ASCII hexadécimal de la chaîne y est `79`
- le codage ASCII hexadécimal du caractère J est `4A`
- `serialVersionUID` est un long qui permet d'identifier une classe Java. On supposera qu'il est ici égal à `00 00 00 00 00 00 00 00`
- une classe n'héritant d'aucune autre (cas de `Point2D`) a pour description de super-classe une référence nulle

5. Par rapport au codage des blocs de données vu aux questions 1 et 2, quelle différence majeure présente le codage de l'objet de la question 4 ?

6. Décoder les données sérialisées suivantes :

```
ac ed 00 05 73 72 00 08 45 74 75 64 69 61 6e 74
99 7d 3c 17 6f 9d 44 e2 02 00 02 49 00 04 6e 6f
74 65 4c 00 03 6e 6f 6d 74 00 12 4c 6a 61 76 61
2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 78 70 00
00 00 10 74 00 06 44 75 72 61 6e 74
```

Vous prendrez soin de bien commenter/expliquer chaque étape de votre décodage.

7. Pour finir, vous donnerez la définition de la classe de l'objet que vous avez décodé ainsi que la valeur des différents attributs.

Annexe : Java Object Serialization Stream Protocol

Les éléments du flot d'octets

Un minimum de structuration est nécessaire pour représenter les objets dans le flot. Chaque attribut de l'objet doit être présent : sa classe, les champs et les données. Ceux-ci pourront être lus par des méthodes spécifiques à la classe. La représentation des objets dans le flot est décrite par une grammaire. Une représentation particulière est prévue pour les objets nuls, les nouveaux objets, les classes, les tableaux, les chaînes de caractères et les références sur des objets déjà dans le flot. Chaque objet écrit dans le flot est référencé de façon à pouvoir être accessible. Les références commencent à 0.

Une classe d'objet est représentée par son objet `ObjectStreamClass`.

Un objet `ObjectStreamClass` est représenté par :
- un SUID (*Stream Unique Identifier*) de classes compatibles,
- un drapeau indiquant si la classe a des méthodes de lecture/écriture des objets,
- le nombre de champs non statiques ou non temporaires (*transients*),
- le tableau de champs de la classe qui est sérialisé par le mécanisme par défaut. Pour les tableaux et les champs de l'objet, le type du champ est inclus comme une chaîne de caractères.
- les enregistrements de blocs de données ou les objets optionnels écrits par une méthode `annotateClass`.
- l' `ObjectStreamClass` de son super-type (`null` si la superclasse n'est pas sérialisable).

Les chaînes sont représentées par leur encodage UTF (*Unified Text Format*).

Les tableaux sont représentés par :
- leur objet `ObjectStreamClass`,
- le nombre d'éléments,
- la séquence de valeurs. Le type des valeurs est implicite par le type du tableau. Par exemple les valeurs d'un tableau d'octets sont de type octet.

Les nouveaux objets dans le flot sont représentés par :

- la classe la plus dérivée de l'objet,
- les données pour chaque classe sérialisable de l'objet, avec la superclasse la plus haute en premier. Pour chaque classe, le flux contient :
  - les champs sérialisés par défaut (les champs ni statiques ni temporaires comme décrits dans la `ObjectStreamClass` correspondante).
  - si la classe a des méthodes `writeObject`/`readObject`, il peut y avoir des objets ou des blocs de données optionnels des types de primitives écrits par la méthode `writeObject` suivi par un code `endBlockData`.

La grammaire du format flot d'octet

Nous suivons les règles typographiques suivantes : les symboles non terminaux sont en minuscule, les symboles terminaux sont en majuscule.

Un flot d'octet est représenté par n'importe quel flot satisfaisant la règle `stream`.

```
stream:
  magic version contents
contents:
  content
  contents content
content:
  object
  blockdata
object:
  newObject
  newClass
  newArray
  newString
  newClassDesc
  prevObject
  nullReference
  exception
  TC_RESET
newClass:
  TC_CLASS classDesc newHandle
classDesc:
  newClassDesc
  nullReference
  (ClassDesc)prevObject      // an object required to be of type
                             // ClassDesc
superClassDesc:
  classDesc
newClassDesc:
  TC_CLASSDESC className serialVersionUID newHandle classDescInfo
  TC_PROXYCLASSDESC newHandle proxyClassDescInfo
classDescInfo:
  classDescFlags fields classAnnotation superClassDesc
className:
  (utf)
serialVersionUID:
  (long)
classDescFlags:
  (byte)                    // Defined in Terminal Symbols and
                            // Constants
proxyClassDescInfo:
  (int)<count> proxyInterfaceName[count] classAnnotation
      superClassDesc
proxyInterfaceName:
  (utf)
fields:
  (short)<count>  fieldDesc[count]
fieldDesc:
  primitiveDesc
  objectDesc
primitiveDesc:
  prim_typecode fieldName
objectDesc:
  obj_typecode fieldName className1
fieldName:
  (utf)
className1:
  (String)object             // String containing the field's type,
                             // in field descriptor format
```

```
classAnnotation:
  endBlockData
  contents endBlockData      // contents written by annotateClass
prim_typecode:
  'B'   // byte
  'C'   // char
  'D'   // double
  'F'   // float
  'I'   // integer
  'J'   // long
  'S'   // short
  'Z'   // boolean
obj_typecode:
  '[`   // array
  'L'   // object
newArray:
  TC_ARRAY classDesc newHandle (int)<size> values[size]
newObject:
  TC_OBJECT classDesc newHandle classdata[]  // data for each class
classdata:
  nowrclass                 // SC_SERIALIZABLE & classDescFlag &&
                            // !(SC_WRITE_METHOD & classDescFlags)
  wrclass objectAnnotation  // SC_SERIALIZABLE & classDescFlag &&
                            // SC_WRITE_METHOD & classDescFlags
  externalContents          // SC_EXTERNALIZABLE & classDescFlag &&
                            // !(SC_BLOCKDATA  & classDescFlags
  objectAnnotation          // SC_EXTERNALIZABLE & classDescFlag&&
                            // SC_BLOCKDATA & classDescFlags
nowrclass:
  values                    // fields in order of class descriptor
wrclass:
  nowrclass
objectAnnotation:
  endBlockData
  contents endBlockData     // contents written by writeObject
                            // or writeExternal PROTOCOL_VERSION_2.
blockdata:
  blockdatashort
  blockdatalong
blockdatashort:
  TC_BLOCKDATA (unsigned byte)<size> (byte)[size]
blockdatalong:
  TC_BLOCKDATALONG (int)<size> (byte)[size]
endBlockData    :
  TC_ENDBLOCKDATA
externalContent:          // Only parseable by readExternal
  (bytes)                // primitive data
    object
externalContents:         // externalContent written by
  externalContent         // writeExternal in PROTOCOL_VERSION_1.
  externalContents externalContent
newString:
  TC_STRING newHandle (utf)
  TC_LONGSTRING newHandle (long-utf)
prevObject
  TC_REFERENCE (int)handle
nullReference
  TC_NULL
exception:
  TC_EXCEPTION reset (Throwable)object   reset
magic:
```

```
  STREAM_MAGIC
version
  STREAM_VERSION
values:          // The size and types are described by the
                 // classDesc for the current object
newHandle:       // The next number in sequence is assigned
                 // to the object being serialized or deserialized
reset:           // The set of known objects is discarded
                 // so the objects of the exception do not
                 // overlap with the previously sent objects
                 // or with objects that may be sent after
                 // the exception
```

Les symboles et les constantes terminaux

Les symboles suivants dans `java.io.ObjectStreamConstants` définissent les valeurs terminales et les valeurs des constantes attendues dans un flot :

```
final static short STREAM_MAGIC = (short)0xaced;
final static short STREAM_VERSION = 5;
final static byte TC_NULL = (byte)0x70;
final static byte TC_REFERENCE = (byte)0x71;
final static byte TC_CLASSDESC = (byte)0x72;
final static byte TC_OBJECT = (byte)0x73;
final static byte TC_STRING = (byte)0x74;
final static byte TC_ARRAY = (byte)0x75;
final static byte TC_CLASS = (byte)0x76;
final static byte TC_BLOCKDATA = (byte)0x77;
final static byte TC_ENDBLOCKDATA = (byte)0x78;
final static byte TC_RESET = (byte)0x79;
final static byte TC_BLOCKDATALONG = (byte)0x7A;
final static byte TC_EXCEPTION = (byte)0x7B;
final static byte TC_LONGSTRING = (byte) 0x7C;
final static byte TC_PROXYCLASSDESC = (byte) 0x7D;
final static  int   baseWireHandle = 0x7E0000;
```

L'octet `classDescFlags` peut inclure les valeurs suivantes :

```
final static byte SC_WRITE_METHOD = 0x01;  //if SC_SERIALIZABLE
final static byte SC_BLOCK_DATA = 0x08;       //if SC_EXTERNALIZABLE
final static byte SC_SERIALIZABLE = 0x02;
final static byte SC_EXTERNALIZABLE = 0x04;
```

The flag `SC_WRITE_METHOD` is set if the `Serializable` class writing the stream had a `writeObject` method that may have written additional data to the stream. In this case a `TC_ENDBLOCKDATA` marker is always expected to terminate the data for that class.
The flag `SC_BLOCKDATA` is set if the `Externalizable` class is written into the stream using `STREAM_PROTOCOL_2`. By default, this is the protocol used to write `Externalizable` objects into the stream in JDK™ 1.2. JDK™ 1.1 writes `STREAM_PROTOCOL_1`.
The flag `SC_SERIALIZABLE` is set if the class that wrote the stream extended `java.io.Serializable` but not `java.io.Externalizable`, the class reading the stream must also extend `java.io.Serializable` and the default serialization mechanism is to be used.
The flag `SC_EXTERNALIZABLE` is set if the class that wrote the stream extended `java.io.Externalizable`, the class reading the data must also extend `Externalizable` and the data will be read using its `writeExternal` and `readExternal` methods.

Annexe : Table ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

TD 5/TD 6 – RMI

1. Service RMI de réservation

On considère une chaîne hôtelière offrant un service de réservation accessible à distance via Java RMI. Le service gère plusieurs hôtels. Chaque hôtel dispose d'un certain nombre de chambres pouvant être réservées. Les données concernant chaque hôtel sont stockées indépendamment les unes des autres sur le serveur.

Le service offre trois opérations (méthodes) :
- `reserver` : à partir d'un nom de client (chaîne), d'un nom d'hôtel (chaîne), d'une date (chaîne) et d'un nombre de chambre (entier), cette opération renvoie un entier qui correspond au numéro de réservation si celle-ci peut être effectuée ou à –1 sinon
- `annuler` : à partir d'un numéro de réservation (entier), cette opération annule la réservation correspondante. Cette opération ne retourne rien. Si le numéro de réservation n'est pas valide, cette opération ne fait rien.
- `lister` : à partir d'un numéro de réservation (entier), cette opération retourne une chaîne de caractère qui fournit les caractéristiques de la réservation correspondante (nom du client, nom de l'hôtel, date, nombre de chambres). Si le numéro de réservation n'est pas valide, cette opération ne fait rien.

1. De manière générale (sans rentrer dans des détails syntaxiques), quelle est la différence entre une interface et une implantation ? Pourquoi la notion d'interface est importante en client/serveur ?

2. Expliquer quelles sont les règles syntaxiques que doit suivre une interface Java RMI. Même question pour une implantation d'un objet serveur Java RMI.

3. En Java, donner l'interface RMI définissant les trois opérations spécifiées ci-dessus.

4. L'objet RMI jouant le rôle de la chaîne hôtelière peut-il exécuter simultanément plusieurs fois la méthode `reserver` ? Si oui, dans quelles conditions ? Si non, pourquoi ? Mêmes questions avec lister.

En plus de la chaîne hôtelière et de ses clients, on introduit maintenant deux nouveaux intervenants : une compagnie aérienne et une agence de voyage (tous deux également des objets RMI). L'agence de voyage permet aux clients de réserver une formule complète avion + hôtel.

5. Représenter sur un schéma les quatre intervenants. Représenter par des flèches les invocations de méthodes mises en jeu lorsqu'un client demande à une agence de voyage de réserver une formule complète avion + hôtel.

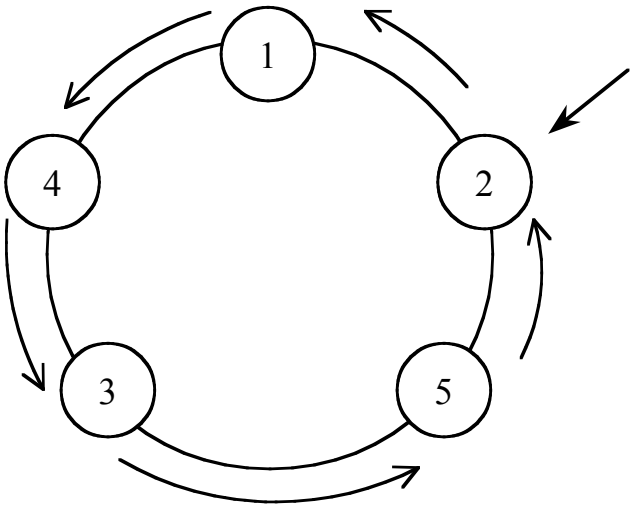
6. À partir du schéma précédent, dire pour chaque intervenant s'il est client (de qui), serveur (pour qui) et client/serveur (de qui et pour qui).

7. L'agence de voyage peut maintenant proposer à ces clients une réservation complète hôtel + avion. L'objet RMI représentant l'agence de voyage gère la double réservation en émettant deux requêtes en parallèle vers les deux objets représentant la compagnie aérienne et la chaîne hôtelière.
- Donner le (ou les) cas où la réservation ne pourra avoir lieu.
- Proposer un fonctionnement pour la mise en place de cette réservation hôtel+avion. Faire une description de ce fonctionnement sous forme d'un diagramme d'interactions entre les objets RMI représentant l'agence de voyage, la compagnie aérienne et la chaîne hôtelière.

8. Donner la liste des opérations à mettre en œuvre et leur enchaînement pour permettre la double réservation ?

2. Election sur un anneau d'objets RMI

On considère un nombre quelconque d'objets RMI reliés selon une topologie logique en anneau. Chaque objet connaît un voisin droit et un voisin gauche avec qui il est capable de communiquer. Chaque objet gère donc deux variables (`voisinDroit` et `voisinGauche`) qui sont des références d'objets RMI. Un identifiant unique (un entier) est attribué à chaque objet. L'élection est un mécanisme qui consiste à élire l'objet d'identifiant le plus élevé (pour par exemple, lui attribuer un privilège). L'élection est réalisée à l'aide d'un message qui « fait le tour » de l'anneau. Elle peut être déclenchée par n'importe quel objet de l'anneau qui dans ce cas, s'appelle un initiateur. Deux types d'élection sont possibles : en tournant dans le sens des aiguilles d'une montre ou dans le sens inverse. À la fin de l'élection, tous les objets de l'anneau doivent connaître l'identifiant de l'élu.



1. Quel est le test d'arrêt de l'élection i.e. du « tour de l'anneau » ? Quelle information faut-il inclure dans le message qui fait le « tour de l'anneau » pour pouvoir réaliser ce test ?

2. Représenter sur un diagramme les échanges de messages engendrés par l'exécution d'une élection avec l'objet 2 comme initiateur.

3. Quel est le test de décision de l'élu ? Quelle information faut-il inclure dans le message qui fait le « tour de l'anneau » pour pouvoir réaliser ce test ?

4. Reprendre le schéma de l'anneau avec cinq objets donné ci-dessus et indiquer sur chaque message la valeur de ces deux informations.

5. À la fin d'un tour, quel(s) objet(s) connaî(ssen)t l'identifiant de l'élu ? Pourquoi ?

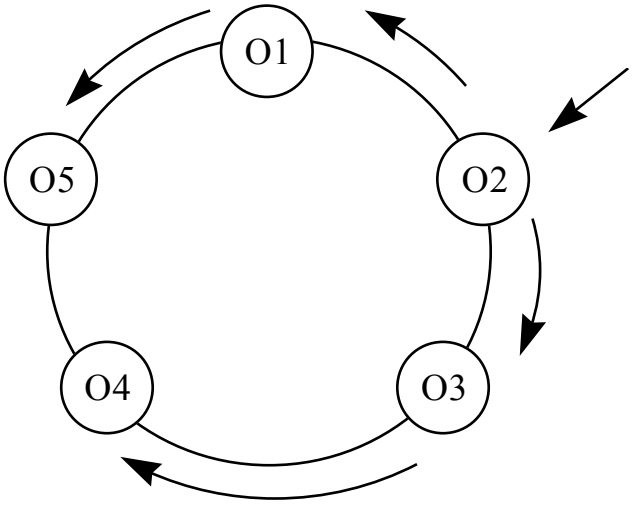
6. Proposer deux solutions, une à base de messages synchrones, une à base de messages asynchrones, pour que tous les objets de l'anneau connaissent l'identifiant de l'élu.

7. Sur chaque objet, le « tour de l'anneau » correspond à une méthode `election` prenant en entrée les informations définies aux questions 1 et 3. Pour chacune des solutions proposées à la question précédente, définir l'interface de la classe Java correspondant à un objet de l'anneau.

8. Pour chacune des solutions proposées à la question précédente, donner en Java l'implantation de la méthode `election`.

3. Election contrarotative sur un anneau d'objets RMI

On considère un nombre quelconque d'objets RMI reliés selon une topologie logique en anneau. L'élection consiste à désigner un site particulier au sein de cet anneau. On souhaite étendre l'algorithme d'élection de l'exercice précédent. On appelle « vague » le mécanisme de propagation d'objet en objet qui permet de réaliser l'élection. **L'élection n'est maintenant plus réalisée par une seule vague comme précédemment, mais par 2 vagues « qui tournent » en sens inverse** (voir figure). Ainsi, l'objet initiateur d'une élection propage simultanément une vague vers la droite et une vague vers la gauche. Les objets intermédiaires propagent les vagues dans le sens où ils les reçoivent. Lorsque les deux vagues se rencontrent, elles refluent (via le message de retour associé à l'invocation de méthode) chacune de leur côté vers l'initiateur. Sauf pour la question 8, on suppose qu'il n'y a simultanément qu'un seul objet initiateur sur l'anneau.



Dans cet exemple, O2 est initiateur.
Il propage 2 vagues :
- une à gauche vers O1
- une à droite vers O3
Lorsque les vagues se rencontrent, elles refluent vers O2.

Chaque objet visité par une vague, choisit un nombre aléatoire. L'élu est l'objet qui a tiré le plus grand nombre. A la fin, l'initiateur proclame les résultats (identité de l'élu et valeur tirée).

Test d'arrêt de la propagation

On étudie le mécanisme à mettre en place pour décider si un objet doit continuer à propager une vague ou si la vague doit être retournée.
On appelle antipodes, l'objet (dans le cas d'anneaux contenant un nombre pair d'objets) ou les objets (dans le cas d'anneaux contenant un nombre impair d'objets) positionnés sur l'anneau de façon symétrique par rapport à un objet donné. Par exemple, sur un anneau à 4 objets (O1, O2, O3, O4), l'objet aux antipodes de O2 est O4. De même, sur un anneau à 5 objets (O1, O2, O3, O4, O5), les objets aux antipodes de O2 sont O4 et O5.

1. Peut-on supposer que les deux vagues se rencontrent toujours sur des objets situés aux antipodes de l'initiateur ? Pourquoi ?

2. Expliquer comment le test à mettre en place pour décider si un objet doit continuer à propager une vague ou si la vague doit être retournée peut être réalisé. Il est conseillé d'illustrer cette explication par un schéma.

Diagramme de séquence

3. Représenter sur un diagramme les échanges de messages engendrés par l'exécution d'une élection avec l'objet O2 comme initiateur.

Evaluation de performances

On souhaite maintenant comparer les performances de cette nouvelle version de l'élection avec la version en une seule vague. On s'intéresse aux performances en nombre d'invocations de méthodes (1 invocation = 1 message d'appel + 1 message de retour) et en temps.

4. Par rapport à la version en une seule vague, la nouvelle version diminue-t-elle ou augmente-t-elle le nombre d'invocations de méthodes nécessaires à une élection ? De combien d'unités ? Il est conseillé de détailler explicitement le nombre d'invocations, éventuellement à l'aide d'un exemple. Par rapport à la version en une seule vague, la nouvelle version améliore-t-elle ou détériore-t-elle le temps nécessaire à une élection ?

Interfaces

5. Définir à l'aide du langage Java, l'interface des objets membres de l'anneau.

6. Cette interface doit-elle obligatoirement être différente de celle définie pour l'élection avec une seule vague ou peut-on réutiliser la même interface ? Justifier votre réponse.

Implantation d'un objet de l'anneau

7. En supposant, qu'il n'y a qu'un **seul objet initiateur simultanément**, donner en Java, le code de la classe qui implante le comportement d'un objet de l'anneau.

8. On suppose maintenant que plusieurs objets peuvent initier simultanément une élection. Y a-t-il des problèmes nouveaux à gérer ? Si oui, le(s)quel et expliquer alors (sans forcément donner un nouveau code), comment votre algorithme de la question précédente peut être modifié.

TP 2 – Passerelle REST

Le but de ce TP est de permettre d'accéder au serveur FTP programmé lors du TP 1 via une passerelle REST.

1. installation

Nous utiliserons l'IDE NetBeans avec le serveur d'applications GlassFish qu'il sera nécessaire d'installer (GlassFish servira également pour le TP 4 sur Java EE).

1. Télécharger le serveur « GlassFish Server Open Source Edition » à l'adresse suivante :
   http://glassfish.java.net/public/downloadsindex.html
2. Procéder à l'installation dans votre répertoire personnel
3. Dans NetBeans, onglet Services > Servers > clic droit > Add Server…

Vous pouvez tester le fonctionnement de l'installation avec, par exemple, un des exemples REST fournis par NetBeans, onglet Projects > clic droit > New Project… > Samples > Web Services > REST: Hello World, puis onglet Projects > HelloWorld > clic droit > Run.

2. Travail à réaliser

Ce TP met en œuvre une architecture répartie à trois niveaux : client REST, passerelle REST/FTP, serveur FTP. Les questions qui suivent ont pour but de concevoir et d'implanter progressivement la passerelle REST/FTP.

Vous réutiliserez le serveur FTP que vous avez développé pour le TP 1. Pour l'accès au serveur FTP depuis la passerelle, vous pourrez utiliser par exemple la librairie Apache Commons Net (`http://commons.apache.org/net`) qui fournit dans le package `org.apache.commons.net.ftp` un ensemble de classes implantant un client FTP. Pour les questions 1 à 5, l'authentification vers le serveur FTP pourra être codé en dur dans la passerelle.

1. Dans une architecture REST, chaque répertoire et chaque fichier est modélisé comme une ressource accessible via les différents verbes du protocole HTTP. Dans un premier temps, implanter l'accès aux ressources fichiers en utilisant le type MIME `application/octet-stream` et le verbe GET. Tester le fonctionnement à l'aide d'un navigateur.

2. Ajouter les méthodes supportant les verbes POST, PUT, DELETE en précisant un choix de comportement pour les méthodes associées au POST et au PUT.

3. Ajouter les ressources répertoires en mode `application/html` via le verbe GET. La passerelle retourne alors le contenu du répertoire sous la forme d'un document HTML. Tester le fonctionnement à l'aide d'un navigateur.

4. Faire en sorte que le contenu du répertoire retourné par la passerelle contienne des liens HTML qui permettent d'accéder aux ressources (fichier ou répertoire) correspondantes.

5. Faire en sorte que le contenu retourné par la passerelle contienne en plus un formulaire permettant de déposer un nouveau fichier dans le répertoire.

6. Proposer et implanter une solution pour gérer l'authentification de l'utilisateur depuis HTTP.

Références

- Style architectural REST : http://en.wikipedia.org/wiki/REST
- Types MIME : http://en.wikipedia.org/wiki/MIME
- Authentification HTTP : http://fr.wikipedia.org/wiki/HTTP_Authentification

TD 7/TD 8 – RMI

1. Annuaire

L'objectif de cet exercice est d'étudier la mise en œuvre d'un annuaire de serveurs Internet.

1. Quel est le rôle d'un annuaire dans une application répartie ?

L'annuaire de serveurs Internet sera géré par un serveur TCP accessible depuis les processus clients à travers un objet souche (voir Figure 1).

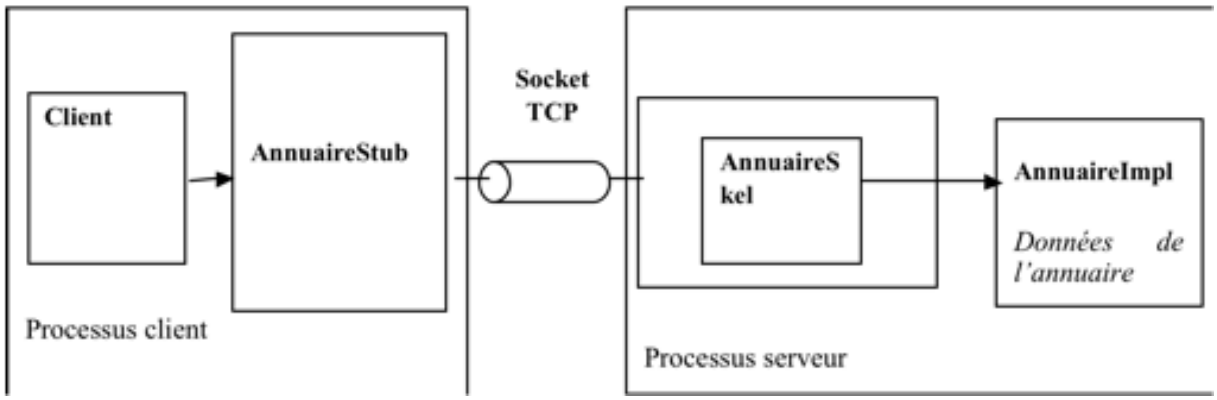


Figure 1 : Relation entre les objets de l'application répartie

On considère que les adresses physiques sont représentées à l'aide de la classe `Adresse` suivante :

```
public class Adresse {
  public String adresse; // Adresse IP de la machine
  public int port;       // Numéro de port TCP
  public Adresse(String a,int p) { this.adresse=a; this.port=p; }
}
```

L'interface d'accès à l'annuaire est la suivante :

```
public interface Annuaire {

  /**
   * Enregistre une association entre une adresse logique et une adresse
   * physique.
   * @return false si adresseLogique est déjà utilisée
   */
  boolean ajouter( String adresseLogique, Adresse adressePhysique );

  /**
   * Modifie une association (adresse logique – adresse physique).
   * @return false si adresseLogique n'existe pas.
   */
  boolean modifier( String adresseLogique, Adresse adressePhysique );
```

```
  /**
   * Retire une association (adresse logique – adresse physique).
   * @return false si adresseLogique n'existe pas.
   */
  boolean retirer( String adresseLogique );

  /**
   * @return l'adresse physique associée à l'adresse logique fournie
   *         ou null si l'adresse logique n'existe pas.
   */
  Adresse chercher( String adresseLogique );

   /**
    * @return la liste des adresses logiques enregistrées.
    */
   String[] lister();
}
```

On considère que l'on utilisera les classes `DataInputStream` et `DataOutputStream` pour lire et écrire sur les flux d'entrée et de sortie de la *socket* TCP.

```
// Création d'un flux de lecture de données binaires.
java.io.DataInputStream fluxIn =
        new java.io.DataInputStream(unInputStream);

// La lecture de données sur un flux java.io.DataInputStream
int unEntier = fluxIn.readInt();
boolean unBooleen = fluxIn.readBoolean();
String uneChaine = fluxIn.readUTF();

// Création d'un flux d'écriture de données binaires.
java.io.DataOutputStream fluxOut =
  new java.io.DataOutputStream(unOutputStream);

// L'écriture de données sur un flux java.io.DataOutputStream
fluxOut.writeInt(unEntier);
fluxOut.writeBoolean(unBoolean);
fluxOut.writeUTF(uneChaine);
```

2. Définir le protocole permettant d'accéder à l'annuaire. Pour cela, lister les différents messages de requête et de réponse échangés, identifier l'ordonnancement de ces messages, leur structuration et leur contenu en terme de types et valeurs des données encodées et désencodées.

3. La figure 1 suggère une architecture côté client comprenant deux parties : la classe `Client` proprement dite et la classe `AnnuaireStub`. Quel peut être l'intérêt d'un tel découpage ?

On considère un client qui effectue la séquence d'appels suivante sur un annuaire hébergé sur la machine annuaire.lifl.fr, port 89 :
- ajouter adresse logique « car », adresse physique rac.lifl.fr port 5896,
- ajouter adresse logique « m1 », adresse physique master.lifl.fr port 698,
- lister.

4. Écrire le code des classes `Client` et `AnnuaireStub` correspondant au comportement précédent. La classe `AnnuaireStub` doit implémenter l'interface `Annuaire`.

5. La figure 1 suggère une architecture côté serveur comprenant deux parties : la classe `AnnuaireSkel` et la classe `AnnuaireImpl`. Quel est l'intérêt d'un tel découpage ?

6. Écrire le code des classes `AnnuaireSkel` et `AnnuaireImpl`. La classe `AnnuaireImpl` doit implémenter l'interface `Annuaire`.

7. Parmi les 4 classes précédentes, quelles sont celles dont le code pourrait être généré automatiquement ? Quelle information minimale a-t-on besoin pour cela ? Proposer en pseudo-code un algorithme pour cela.

2. Instanciation à la demande

Cet exercice s'intéresse à l'instanciation à la demande d'objets serveurs RMI. L'idée est de ne pas laisser actifs en permanence tous les objets serveurs RMI, mais de les activer en fonction des besoins des clients. Dans cet exercice, l'activation est réalisée par un serveur Web. L'activation peut être une vraie création ou une récupération d'objet existant dans un pool. Les objets serveurs RMI exécutent des calculs mathématiques quelconques. Chaque type de calcul est identifié par un nom sous forme d'une chaîne de caractères (ex. : intégration, dérivation, équation différentielle, etc.). Les utilisateurs demandent donc au serveur Web d'activer des objets serveurs RMI d'un certain type en fonction de leurs besoins.

On dispose de trois catégories de programmes : programme utilisateur, serveur Web et objets serveurs RMI. Le scénario de fonctionnement est le suivant :
- étape 1 : le programme utilisateur demande au serveur Web à l'aide d'une commande GET d'activer un objet serveur RMI en lui fournissant un nom (correspondant au type de calcul dont il a besoin).
- étape 2 : le serveur Web retourne au programme utilisateur la référence de l'objet serveur RMI.
- étape 3 : le programme utilisateur invoque l'objet serveur RMI.

3. Dans le scénario précédent, et pour les trois catégories de programme, dire qui est client (de qui), qui est serveur (pour qui) et qui est client/serveur (de qui et pour qui) ?

Interface RMI

Les objets serveurs RMI offrent les méthodes suivantes :
- `submit` : permet d'effectuer un calcul. Cette méthode prend en paramètre une chaîne de caractères qui contient les arguments du calcul et retourne un objet Java contenant le résultat du calcul. Cet objet Java est transmis par valeur.
- `getHost` : fournit l'adresse de la machine sur laquelle est instancié l'objet RMI. Cette méthode ne prend pas de paramètres. Elle retourne une chaîne de caractères contenant l'adresse de la machine.
- `getPort` : retourne un port TCP sur lequel l'objet RMI est capable de lire des messages. Cette méthode ne prend pas de paramètres. Elle retourne un entier correspondant au port TCP en question.

4. Quelle est la différence entre transmission par valeur et transmission par référence ? Vis-à-vis de RMI, quelle condition doit respecter la classe d'un objet transmis par valeur ? Même question pour un passage par référence ?

5. Ecrire l'interface `CalculItf` de ces objets serveurs RMI.

Programme utilisateur

On souhaite que les programmes clients puissent périodiquement scruter les objets RMI auxquels ils ont soumis des calculs pour savoir s'ils sont toujours en fonctionnement. En même temps qu'ils appellent la méthode `submit`, les programmes client testent l'activité de l'objet RMI. Pour cela, ils utilisent un thread (classe `ClientPingThread`) pour envoyer un message TCP contenant les 4 octets P I N G, sur l'adresse fournie par la méthode `getHost` et le port fourni par `getPort`. Les objets serveurs RMI utilisent un thread (classe `ServeurPingThread`) pour recevoir ce message et y répondre avec un message contenant les 3 octets A C K. Si 10s après l'envoi du message P I N G le client n'a pas reçu de réponse ou si la réponse ne correspond pas à A C K, il renvoie le message P I N G. Si au bout de 3 envois, il n'a toujours pas de réponse, il lève une exception pour signaler la panne de l'objet serveur RMI.

6. Un thread Java peut s'écrire de 2 façons. Donner en une.

7. Le message P I N G permet de détecter si l'objet RMI est en panne ou toujours en activité. Il existe une autre technique de détection de pannes. Donner son nom et expliquer brièvement son fonctionnement.

8. Proposer une solution simple, n'utilisant pas de thread supplémentaire, pour savoir que l'attente de 10s du message A C K est arrivée à échéance.

9. Ecrire le code Java de la classe `ClientPingThread` qui teste l'activité du serveur.

10. Ecrire le code Java du programme client qui invoque la méthode `submit("12.5")` sur l'objet RMI `obj` et qui en même temps, teste l'activité du serveur.

Serveur RMI

11. Ecrire le code Java de la classe `ServeurPingThread` qui traite les messages P I N G du client. On utilisera le numéro de port TCP 2003.

12. Ecrire le code Java de la méthode `submit`. Vous laisserez une zone de commentaire pour indiquer l'emplacement du calcul et la valeur retournée par la méthode.

Serveur Web

13. Est-il nécessaire que les objets serveurs RMI créés par le serveur Web soient enregistrés dans le service de noms de RMI (`rmiregistry`) ? Si oui, pourquoi ? Si non, comment les programmes utilisateurs peuvent invoquer des méthodes sur ces objets ?

14. Si l'on veut que le serveur Web puisse retourner la même référence d'objets RMI à deux clients s'exécutant simultanément, de quoi faut-il s'assurer ?

Le serveur Web reçoit des requêtes de la forme :
`GET /index?calcul=`*nom* `HTTP/1.1`

où *nom* est le nom du calcul mathématique demandé.

15. Décrire en français l'algorithme de fonctionnement du serveur Web.

3. Patron Observateur/Sujet

Le patron de conception Observateur/Sujet permet de faire prendre en compte les changements d'un objet (le sujet) par d'autres objets (les observateurs). Cela assure, par exemple, la mise à jour des données sur les objets observateurs. Cette relation entre deux objets peut être explicitement codée dans la classe représentant le sujet, mais cela demande une connaissance sur la façon dont les observateurs doivent être mis à jour. Ce type de réalisation fait que les objets deviennent fortement couplés et ne peuvent pas être réutilisés.

Nous proposons donc une approche plus faiblement couplée par l'intermédiaire de deux classes `Observer` et `Subject` qu'il sera possible d'utiliser ou d'hériter. Un changement dans un objet devra être signalé aux autres par une notification, leur permettant ainsi de se tenir à jour.

L'utilisation de ce patron se fait donc en deux temps :
- un observateur s'abonne et se désabonne d'un sujet par l'intermédiaire de deux méthodes offertes par l'objet sujet (`attach` et `detach`),
- une fois l'observateur abonné au sujet, ce dernier le prévient lors de la modification d'un événement dans son environnement par l'appel de la méthode `notification` offerte par l'objet observer.

Interfaces

1. Les deux objets `Observer` et `Subject` ont des relations client/serveur. Décrire à quels moments ces objets ont une fonction de client et à quels moments ils ont une fonction de serveur.

2. Écrire l'interface Java RMI des deux objets `Observer` et `Subject`.

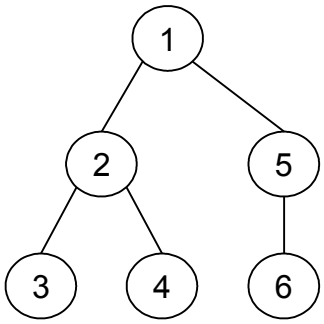
Implantation

3. Écrire le code Java RMI des classes associées aux interfaces `Subject` et `Observer` permettant la mise en place du patron observateur/sujet. Une notification sera émise toutes les secondes.

TP 3 – RMI – Transfert de données

On considère une application répartie qui permet de transférer en RMI des données à un ensemble d'objets organisés, dans un premier temps, selon une topologie en arbre. Chaque nœud de l'arbre s'exécute dans une machine virtuelle différente et propage les données à ses fils. Par exemple, dans la figure ci-dessous, pour diffuser son message, l'objet 1 l'envoie à 2 et 5, puis 2 l'envoie à 3 et 4 et 5 l'envoie à 6.

Chaque objet a une variable `pere` qui est une référence d'objet RMI et un tableau `fils` qui est un tableau d'objets RMI. Les données transférées sont des tableaux d'octets.



1. Écrire l'interface RMI `SiteItf` et la classe `SiteImpl` implémentant cette interface qui mettent en œuvre le mécanisme de transfert décrit ci-dessus. Vous ferez en sorte de fournir des messages de trace permettant de visualiser la propagation des données dans l'arbre.

2. On souhaite maintenant faire en sorte que les diffusions puissent être initiées à partir de n'importe quel objet de l'arbre (pas uniquement à partir de l'objet racine).

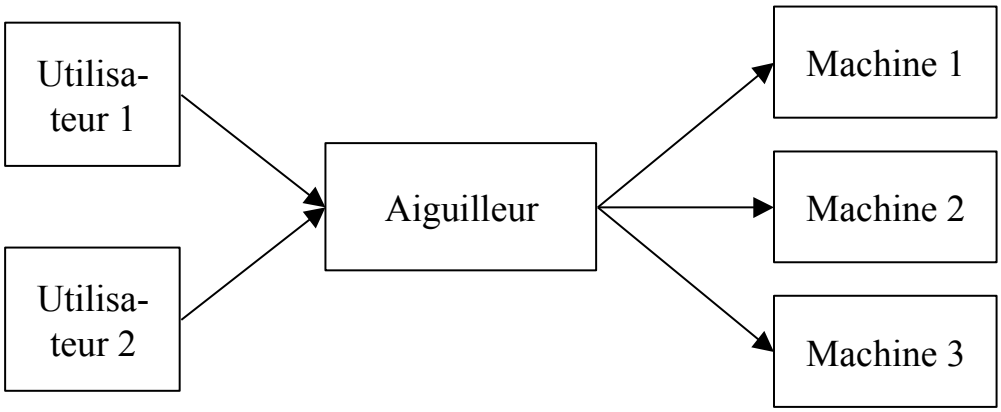
3. On souhaite maintenant faire en sorte que les diffusions s'effectuent concurremment dans chacune des branches de l'arbre.

4. On suppose maintenant que les objets sont organisés selon une topologie qui n'est plus un arbre, mais un graphe. Par exemple, on ajoute un arc entre les nœuds 4 et 6 de l'exemple précédent. Modifier le programme précédent pour gérer ce nouveau cas.

TD 9 – CORBA – Répartiteur de charge

1. Répartiteur de charge

Cet exercice a pour but d'étudier dans le cadre d'un environnement de programmation répartie un aiguilleur de requêtes. Celui-ci permet de répartir la charge de traitement entre plusieurs machines.
La figure ci-dessous présente l'architecture envisagée qui comprend deux utilisateurs, un aiguilleur et trois machines identiques (i.e. fournissant les mêmes services). Les utilisateurs, l'aiguilleur et les machines sont des objets CORBA.
Les utilisateurs soumettent des requêtes à l'aiguilleur. Pour chaque requête, l'aiguilleur choisit de la rediriger à une machine et une seule. La machine choisie traite la requête, fournit le résultat à l'aiguilleur qui le retransmet à l'utilisateur. Pendant le temps de traitement de la requête, l'aiguilleur peut aiguiller une nouvelle requête vers une autre machine ou vers la même. L'aiguilleur est donc un objet concurrent. Le degré de concurrence des machines est étudié à partir de la question 9.



Interfaces

1. Pour chacun des trois types d'objets CORBA (utilisateur, aiguilleur, machine), dire s'ils sont client (de qui), serveur (pour qui) ou client/serveur (de qui et pour qui).

Chaque objet machine fournit deux services :
- un service dit `lecture` qui à partir d'un nom de fichier (une chaîne de caractères) fourni en entrée, retourne les données (un tableau de taille non déterminée d'octets) correspondant au contenu du fichier,
- un service dit `ecriture` qui à partir d'un nom de fichier (une chaîne de caractères) et de données (un tableau de taille non déterminée d'octets) fournis en entrée, écrit les données dans le fichier. Ce service retourne un booléen qui vaut vrai en cas de succès de l'opération d'écriture.

On suppose dans un premier temps que tous les objets machines ont accès aux mêmes fichiers et que l'écriture dans un fichier est répercutée de façon immédiate et automatique sur toutes les machines.

2. Donner en IDL CORBA la définition de l'interface `Machine` contenant ces 2 services.

Aiguilleur

3. Sans anticiper sur les questions suivantes et sans en donner la définition explicite, expliquer à partir des indications fournies jusqu'à présent, quelle doit être l'interface de l'aiguilleur.

On souhaite maintenant que des objets machines puissent être ajoutés et retirés en cours d'exécution au *pool* d'objets géré par l'aiguilleur.

4. Quelle(s) méthode(s) faut-il définir pour prendre en compte ces fonctionnalités ? Pour quel(s) objet(s) ? Définir en IDL CORBA, l'interface `Controle`, réunissant ces fonctionnalités. Donner en français la signification précise de chaque élément (méthodes, paramètres, etc.) défini.

5. Proposer en français ou en pseudo code un algorithme, le plus simple possible, pour effectuer la répartition de charge. L'algorithme doit répartir équitablement les requêtes entre toutes les machines sans tenir compte du fait que certaines requêtes peuvent être plus longues que d'autres.

On souhaite maintenant prendre en compte les charges des machines. Périodiquement les machines informent l'aiguilleur de leur niveau de charge. Celui-ci est représenté par un entier donnant le nombre de requête en attente sur la machine. On suppose que ces notifications sont des informations non prioritaires dont la perte est sans conséquence.

6. Quelle(s) méthode(s) faut-il définir pour prendre en compte cette fonctionnalité ? Pour quel(s) objet(s) ? Définir en IDL CORBA, l'interface `Notification` comprenant cette fonctionnalité.

7. Finalement, comment construit-on l'interface complète de l'aiguilleur prenant en compte l'ensemble des fonctionnalités le concernant ?

Concurrence des traitements

8. Expliquer quel peut être le degré de concurrence acceptable pour une machine en fonction des deux types de requêtes lecture et écriture.

On considère maintenant que lors d'une opération d'écriture, la mise à jour du fichier sur toutes les machines est à la charge de la machine qui traite la requête d'écriture.

9. Faut-il ajouter des définitions aux interfaces précédentes pour prendre en compte ce cas ? Si oui, le(s)quelle(s) ? Si non, pourquoi ?

On s'intéresse maintenant aux incohérences qui peuvent survenir lorsque l'aiguilleur répartit successivement deux requêtes d'écriture pour un même fichier sur deux machines différentes.

10. Expliquer en quoi consiste cette incohérence. Proposer pour gérer ce problème au niveau de l'aiguilleur, une solution simple qui ne modifie aucune interface.

Performances

11. L'aiguilleur étant un point de passage obligé et de ce fait un goulet d'étranglement pour les requêtes et les réponses, proposer une solution n'introduisant pas de nouvelles entités pour alléger sa charge, notamment en ce qui concerne la gestion des réponses.

12. Expliquer sans forcément les définir explicitement quelle(s) conséquence(s) cela entraîne sur les interfaces.
Pour palier au problème du goulet d'étranglement, on propose en plus de la solution précédente, de mettre en place plusieurs aiguilleurs dont la répartition des rôles sera prédéfinie (codée "en dur" chez les utilisateurs).

13. Proposer deux façons de répartir le travail entre les aiguilleurs, autres que celle de l'algorithme que vous avez proposé en 5.

TD 10 – CORBA – Protocole de validation à deux phases

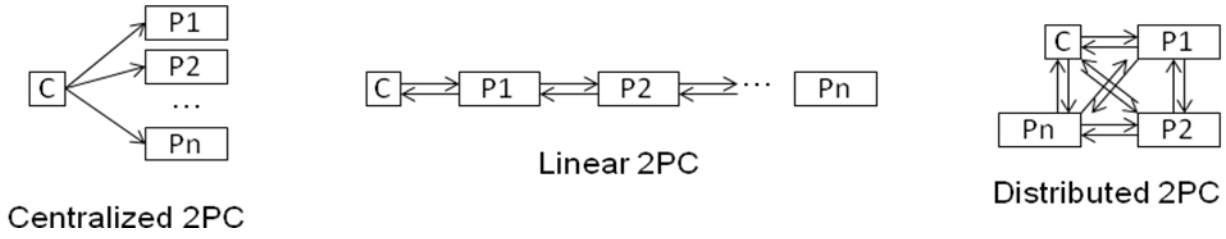
1. Protocole de validation à deux phases

Cet exercice a pour but d'étudier la conception d'un protocole transactionnel de validation à deux phases avec des objets CORBA.

1. Donner la définition d'un protocole de validation à deux phases.

Centralized 2PC

Dans un premier temps, nous étudierons la version dite *Centralized 2PC* de ce protocole (voir figure ci-dessous). Dans cette version, on considère qu'un objet coordinateur, noté C, dispose à un moment donné de la référence de tous les objets participants à la transaction. Les participants sont en nombre quelconque et sont notés P1, P2, … , Pn.



Dans cette version, trois interfaces sont définies : `TransactionItf`, `GestionItf` et `ParticipantItf`.

L'interface `TransactionItf` permet à un objet utilisateur noté U d'effectuer une transaction. Elle comporte les opérations suivantes :
- `begin` : demande de démarrage d'une transaction. Retourne un entier représentant l'identifiant de transaction.
- `commit` : demande de validation d'une transaction. Prend en paramètre un identifiant de transaction. Retourne un booléen pour indiquer si la transaction a pu être engagée.
- `rollback` : demande d'annulation d'une transaction. Prend en paramètre un identifiant de transaction. Retourne un booléen pour indiquer si la transaction a pu être annulée.

L'interface `GestionItf` permet d'enregistrer un participant auprès du coordinateur. Elle comporte les opérations `register` et `remove` prenant chacune en paramètre un participant et un identifiant de transaction et permettant respectivement d'enregistrer et de retirer un participant.

L'interface `ParticipantItf` est une interface permettant d'identifier un participant et ne comporte pour l'instant aucune opération.

2. Définir en CORBA IDL les interfaces `TransactionItf`, `GestionItf` et `ParticipantItf`.
7.

2. Parmi les objets U, C, P1, P2, … Pn, indiquer quels objets implémentent quelle(s) interface(s).

On considère une transaction avec des opérations `credit` et `debit` permettant de créditer et débiter un compte bancaire d'un certain montant. Ces opérations sont définies dans une interface `BanqueItf` qui étend `ParticipantItf`. Elles permettent de réaliser une transaction qui correspond au transfert d'un certain montant entre deux comptes bancaires.

On s'intéresse à la conception du protocole client/serveur permettant de mettre en œuvre une telle transaction. Pour cela, vous pourrez être amené à étendre les interfaces définies ci-dessus. Votre solution doit faire en sorte de respecter les contraintes suivantes :

- Le coordinateur doit conserver un comportement général et ne pas comporter de spécificité due au fait que la transaction correspond à un transfert entre comptes bancaires.
- Le coordinateur ne connait pas a priori les participants. Ceux-ci doivent donc lui être indiqués par l'utilisateur pour chaque transaction.

3. Proposer une solution permettant de mettre en œuvre une telle transaction. Votre présentation peut être accompagnée de quelques phrases expliquant les choix effectués, d'un diagramme de séquence illustrant les invocations d'opérations entre les objets U, C, P1, P2, … Pn et le cas échéant de la définition en CORBA IDL des nouvelles interfaces que vous proposez.

4. Écrire en Java le code des objets coordinateur et participant.

Linear 2PC

On s'intéresse maintenant à la version dite Linear 2PC. Comme illustré sur la figure, dans cette version, le coordinateur dispose à un moment donné de la référence du premier participant, qui lui-même dispose de la référence du participant suivant, etc. jusqu'au dernier participant.

5. Expliquer comment cette version permet de réduire le nombre d'invocations d'opérations en effectuant la transaction en une seule phase.

6. Proposer une solution permettant de mettre en œuvre une telle transaction. Votre présentation peut être accompagnée de quelques phrases expliquant les choix effectués, d'un diagramme de séquence illustrant les invocations d'opérations entre les objets U, C, P1, P2, … Pn et le cas échéant de la définition en CORBA IDL des nouvelles interfaces que vous proposez.

Distributed 2PC

On s'intéresse maintenant à la version dite *Distributed 2PC*. Comme illustré sur la figure, dans cette version, le coordinateur et tous les participants disposent à un moment donné de la liste des références de tous les participants et du coordinateur.

7. Expliquer comment cette version permet d'effectuer la transaction en une seule phase.

8. Proposer une solution permettant de mettre en œuvre une telle transaction. Votre présentation peut être accompagnée de quelques phrases expliquant les choix effectués, d'un diagramme de séquence illustrant les invocations d'opérations entre les objets U, C, P1, P2, … Pn et le cas échéant de la définition en CORBA IDL des nouvelles interfaces que vous proposez.

TP 4 – Java EE

Ce TP a pour but de manipuler Java EE.

Nous utiliserons l'IDE NetBeans avec le serveur d'applications Java EE GlassFish qui a été installé pour le TP 2.

1. JSP

1. Écrire un fichier HTML contenant un formulaire permettant de saisir les informations relatives à un livre représenté par un titre, un nom d'auteur et une année de parution.

2. Écrire une JSP, adossée au formulaire précédent, qui affiche un récapitulatif des informations saisies dans le formulaire.

3. Proposer une solution pour que, suite à l'affichage du récapitulatif précédent, le formulaire soit réaffiché pré-rempli avec ces informations saisies.

2. EJB

1. Implanter un EJB permettant de stocker les informations (titre, auteur, année) relatives à un livre. Le titre est la clé primaire qui permet d'identifier chaque livre.

2. Développer un EJB session offrant deux services : un service d'initialisation permettant d'enregistrer quelques livres prédéfinis, et un service retournant la liste de tous les auteurs enregistrés.

3. Développer deux servlets, chacune permettant d'invoquer respectivement, l'un des deux services précédents.

4. Compléter votre application en offrant à l'utilisateur la possibilité d'ajouter un livre (via un formulaire) et d'obtenir la liste des livres existants.

5. Ajouter à votre programme une fonctionnalité de panier électronique afin qu'un utilisateur puisse sélectionner un ou plusieurs livres et passer commander. Chaque commande est matérialisée par un numéro et la liste des livres sélectionnés. Elle n'est enregistrée en base que lorsque l'utilisateur indique qu'il passe commande.

TD 11 – Révision

1 Question de cours

1.1 Dans les mécanismes d'invocation de méthodes à distance, à quoi sert une souche cliente ?
1.2 Comment partager simplement de l'information entre plusieurs EJB *session bean* sans passer par un *entity bean* ?
1.3 Quelle est à votre avis la différence conceptuelle fondamentale entre RMI et REST ?

NB : lorsqu'il est demandé de comparer des notions, il ne s'agit pas de recopier les définitions des notions respectives, mais d'argumenter et d'expliquer en quoi ces notions sont similaires ou différentes.

2 Java EE – Vidéo-club en ligne

Cet exercice s'intéresse à la conception d'une application client/serveur de diffusion de films à base d'EJB et d'objets RMI. Elle met en relation un utilisateur et un vidéo-club en ligne. L'utilisateur se connecte au vidéo-club, demande un film et visualise en temps réel le film sur sa machine. Avant cela, il faut que l'utilisateur se soit abonné auprès du vidéo-club.

Gestion des abonnés et du catalogue de films

La gestion des abonnés et du catalogue de films est réalisée à l'aide de composants EJB.
Un abonné possède un numéro, un nom et une adresse.
Un film possède un titre, un genre (policier, SF, comédie) et une année de sortie.
On veut pouvoir créer un abonné et rechercher un abonné à partir de son nom.
On veut pouvoir créer un film et rechercher tous les films d'un genre donné.

Visualisation d'un film

La visualisation d'un film se déroule de la façon suivante :
1. l'utilisateur appelle la méthode `visualiser` en fournissant son numéro d'abonné et le titre du film qu'il souhaite visualiser,
2. si le numéro d'abonné et le titre du film existent, la méthode `visualiser` :
   - crée un objet RMI `Projecteur` qui va envoyer les images du film à l'utilisateur,
   - retourne à l'abonné la référence de cet objet.
3. l'utilisateur :
   - récupère la référence de l'objet `Projecteur`,
   - crée un objet RMI `Ecran`,
   - appelle la méthode `run` de l'objet `Ecran` lorsqu'il souhaite commencer la visualisation.

L'interface `ProjecteurItf` de l'objet RMI `Projecteur` possède les méthodes suivantes :
- `setEcran` : prend en paramètre un objet RMI de type `EcranItf`. Ne retourne rien.
- `play` : aucun paramètre, ne retourne rien. Permet de commencer la visualisation d'un film.

L'interface `EcranItf` de l'objet RMI `Ecran` possède les méthodes suivantes :

- `frame` : prend en paramètre un tableau de 1024 octets correspondant à une frame du film à visualiser. La visualisation complète d'un film correspond à plusieurs invocations de la méthode `frame`.

2.1 Quel type de composants EJB faut-il utiliser pour les abonnés ? Pourquoi ? Mêmes questions pour les films ?

2.2 On souhaite mettre en place le *design pattern* Façade pour accéder à l'application. Donner le code Java de l'interface `VideoClubFacade` correspondant à cette façade. Pour cela, vous pourrez être amené à définir de nouvelles interfaces ou classes dont vous donnerez la signification (sans donner leur code).

2.3 Proposer un diagramme d'échange de messages entre l'utilisateur, l'objet `Ecran` et l'objet `Projecteur` correspondant à la visualisation d'un film de 3*1024 octets. Proposer une solution, que vous expliquerez en français, pour que l'objet `Ecran` sache que le film est terminé.

2.4 Ecrire le code Java des interfaces `ProjecteurItf` et `EcranItf`.

2.5 Ecrire le code de la classe `Projecteur`. Le contenu du film à visualiser est fourni via le constructeur.

2.6 On souhaite mettre en place une méthode `pause` pour arrêter temporairement la visualisation d'un film. La reprise de la visualisation se fait en appelant de nouveau la méthode `pause`. Expliquer en français quelles modifications au(x) interface(s) et au(x) classe(s) précédente(s) vous proposez pour mettre en place cette fonctionnalité.

2.7 Plutôt que RMI, on propose maintenant de programmer la méthode `frame` de la classe `Ecran` à l'aide de sockets UDP. Expliquer les avantages et les inconvénients de cette solution par rapport à RMI. Donner le code Java de cette méthode (on ne demande pas le code correspondant à l'affichage que vous remplacerez par un commentaire).

3 CORBA – Service de diffusions d'informations

Cet exercice s'intéresse à la conception d'un service de diffusion d'informations (de type `String`). Les trois entités suivantes sont présentes dans l'application : les producteurs qui produisent l'information, les consommateurs qui la consomment et les canaux de diffusion qui servent à mettre en relation les producteurs et les consommateurs. Un consommateur s'abonne auprès d'un canal en mentionnant qu'il est intéressé par un type d'information représenté par un identifiant (de type `String`). Les consommateurs peuvent s'abonner à plusieurs types d'informations et un producteur peut produire plusieurs types d'informations.

Les canaux fonctionnent selon deux modes : push et pull. Dans le mode push, un producteur ayant une information à produire la transmet au canal qui la retransmet à tous les consommateurs abonnés à ce type d'information. Dans le mode pull, un consommateur interroge le canal pour savoir si une information d'un type donné est disponible, le canal interroge les producteurs qui, si ils disposent d'une information de ce type, la retourne au canal, qui la ou les (si

plusieurs producteurs ont une information à produire) retransmet à tous les consommateurs abonnés.

3.1 Définir une ou plusieurs interfaces CORBA IDL pour gérer les abonnements et les désabonnements des consommateurs. Indiquer quelles entités implémentent quelles interfaces. Préciser le type que vous utilisez pour représenter les consommateurs.

3.2 Définir une ou plusieurs interfaces CORBA IDL pour gérer le mode push. Indiquer quelles entités implémentent quelles interfaces.

3.3 Définir une ou plusieurs interfaces CORBA IDL pour gérer le mode pull. Indiquer quelles entités implémentent quelles interfaces.

3.4 On suppose maintenant que les canaux sont accessibles via REST. Décrire en français la façon dont vous mettriez en place une telle solution.

En plus de push et de pull, un troisième mode de fonctionnement correspondant à une hybridation de push et de pull est envisageable.

3.5 Proposer une description en français du fonctionnement de ce troisième mode.

On considère maintenant que les producteurs sont organisés selon une topologie en anneau et que un seul producteur, qui joue le rôle de point d'entrée dans l'anneau, est relié au canal. La recherche d'information se fait en parcourant l'anneau. On suppose que l'anneau existe et que chaque producteur sait s'il est un point d'entrée ou non. On suppose également que chaque producteur dispose d'une méthode privée `poll` qui retourne vrai si une nouvelle information est disponible pour la production et d'une méthode privée `get` qui retourne l'information à produire.

3.6 Définir une ou plusieurs interfaces CORBA IDL pour gérer le mode pull. Justifier en quoi la solution est similaire ou différente de la solution proposée pour la question 3.3. Écrire en Java l'implémentation d'un producteur membre de l'anneau (les méthodes privées `poll` et `get` peuvent être utilisées, mais on ne demande pas d'écrire leur code).

Pour les anneaux comportant un très grand nombre de producteurs, le tour complet de l'anneau peut prendre un temps élevé. On introduit donc la notion de raccourci : en plus de son voisin dans l'anneau, un nœud peut connaître un raccourci, c'est-à-dire la référence d'un nœud situé plus loin dans l'anneau. Tous les nœuds n'ont pas forcément à leur disposition un raccourci. On fait les mêmes hypothèses qu'à la question précédente, et on ajoute le fait que les raccourcis sont connus.

3.7 Écrire en Java l'implémentation d'un producteur membre de l'anneau en exploitant cette notion de raccourci.

TD 12 – Révision

1. Question de cours

1. Qu'est-ce qu'un *pool de threads* ? En quoi est-ce intéressant pour programmer des serveurs ?

2. Soit une classe Java RMI étendant `UnicastRemoteObject`, y a-t-il un intérêt à programmer un *pool de threads* dans cette classe ? Pourquoi ?

3. Au-delà du fait que REST s'affranchit de la contrainte XML, expliquer quelle est la différence fondamentale entre SOAP et REST.

2. Service de diffusion d'événements

On considère un service CORBA permettant de diffuser des événements entre des producteurs et des consommateurs via un canal. Chaque canal représente un type d'événement à diffuser et propose un mécanisme d'abonnement. Deux modes de fonctionnement sont envisagés : *push*, les producteurs produisent des événements et les « poussent » vers le canal qui les redistribue aux consommateurs, et *pull*, un consommateur interroge le canal qui répercute la requête vers les producteurs pour savoir si un événement est disponible. Il s'agit dans cet exercice de proposer une solution pour la conception de ce service.

1. Représenter sur un diagramme de séquence de messages les invocations de services pour le mode *pull* et le mode *push*.

2. Proposer des interfaces CORBA IDL pour gérer les abonnements, le mode push et le mode pull. Expliquer brièvement en français, le rôle de chacune de ces interfaces et indiquer qui (producteur, consommateur, canal) l'implante.

3. Pour le mode *pull*, expliquer, dans votre solution, comment se passe la transmission des événements au consommateur.

4. Proposer un troisième mode de fonctionnement pour le canal. Expliquer en français les conséquences, si il y en a, que cela entraine sur les interfaces que vous avez défini.

5. Écrire la classe Java de l'objet CORBA qui implante le mode de fonctionnement précédent.

3. Élection dans un réseau d'objets

Cet exercice a pour but d'étudier le mécanisme d'élection sur des topologies logiques de réseaux d'objets : arbre et graphe. Il s'agit de pouvoir désigner l'objet d'identifiant le plus élevé (l'élu), et de faire en sorte que cet identifiant devienne connu de tous les objets. Dans tous les cas, l'élection est déclenchée par un objet appelé initiateur.

1. Cette question s'intéresse à l'élection dans un arbre. Donner le pseudo-code d'un nœud de l'arbre participant à l'élection.

2. Cette question s'intéresse à l'élection dans un graphe quelconque. Expliquer en français le fonctionnement de l'algorithme (on ne demande pas le pseudo-code).

3. Proposer une évaluation de la complexité de ces algorithmes. Quelle topologie est la plus efficace ? Justifier.

4. Serveur Web annuaire RMI

On considère que l'on dispose d'un serveur Web écrit en Java et conforme au protocole HTTP version 1.1. On souhaite que ce serveur Web puisse remplacer l'annuaire `rmiregistry` et que les programmes RMI continuent à utiliser la même interface d'accès à cet annuaire.

1. Décrire en français une solution pour pouvoir réaliser cela en précisant ce que vous allez mettre en place au niveau du serveur Web, des programmes RMI client et des programmes RMI serveur et en précisant ce que vous allez transférer avec HTTP pour les méthodes `bind` et `lookup`.

2. On souhaite maintenant adopter une approche REST pour le remplacement de l'annuaire. Commenter en français les conséquences de ce choix sur l'interface d'accès à l'annuaire.

5. Serveur de fichiers RMI

On souhaite mettre en place un serveur RMI permettant de transférer des fichiers selon les mêmes principes que FTP : connexion avec login/mot de passe, *upload*/*download* de fichier, changement de répertoire, fermeture de connexion. Afin de limiter les invocations avec des volumes trop importants de données, on souhaite par ailleurs que les *upload*/*download* se fassent par blocs de 1024 octets. Plusieurs transferts, éventuellement entre le même client et le même serveur, doivent pouvoir se dérouler simultanément.

1. En présence de plusieurs clients simultanés sur le même serveur, comment faites-vous pour distinguer les requêtes des différents clients ?

2. Proposer une solution en RMI pour reproduire le mécanisme de double canal de communication, un pour l'envoi des commandes, un pour l'envoi des données, utilisé par FTP lors d'un *upload*/*download* de fichier.

3. Proposer une ou plusieurs interfaces RMI permettant de faire cela.

4. Donner le code Java de la ou des classes qui implantent les interfaces précédentes.

5. On souhaite que le client puisse détecter les pannes du serveur. Expliquer en français les modifications que vous proposez d'effectuer pour cela dans le code précédent.