Cahier de Travaux Dirigés et Pratiques

Construction d'Applications Réparties (CAR)

Master Sciences et Technologies

Mention Informatique – M1

2016

Équipe enseignante
Clément Ballabriga
Laurence Duchien
Hanaë Rateau
Giuseppe Lipari
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 (REST, SOAP)	TD 3 : Architecture d'applications Internet	TP 1 : Serveur FTP
Semaine 5	Objets et répartition	TD 4 : Représentation des données	TP 1 : Serveur FTP
Semaine 6	Java RMI	TD 5 : RMI	**Évaluation TP 1** TP 2 : REST
Semaine 7	Akka	TD 6 : Annuaire	TP 2 : REST
Semaine 8	Java EE – JSP	TD 7 : Élection sur un anneau	TP 2 : REST
Semaine 9	Java EE – servlet	TD 8 : Élection contrarotative sur un anneau	**Évaluation TP 2** TP 3 : Akka
Semaine 10	Java EE – JDBC	TD 9 : Répartiteur de charge	TP 3 : Akka
Semaine 11	Java EE – EJB	TD 10 : Protocole de validation à deux phases	**Évaluation TP 3** TP 4 : Java EE
Semaine 12	Java EE – EJB	TD 11 : Vidéo-club en ligne	TP 4 : Java EE
Semaine 13		TD 12 : *Map Reduce*	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.

Protocole : échange structuré de messages entre entités distantes. Vous pouvez prendre l'exemple de HTTP et de ses différentes commandes (GET, POST, HEAD, PUT, etc.).

Pile de protocole : un protocole de niveau inférieur est utilisé pour véhiculer les informations d'un protocole de niveau supérieur (ex. IP, TCP, HTTP).

Entrée/sortie (primitive envoi/réception) bloquante ou non bloquante. Bloquant : la primitive reste bloquée tant que l'opération n'est pas terminée. Exemple envoi bloquant : la primitive d'envoi se termine lorsque toutes les données ont été envoyées. Exemple réception bloquante : primitive bloque tant que l'on n'a pas reçu de message. Respectivement, non bloquant, l'envoi retourne même si toutes les données n'ont pas été envoyées, réception récupère le message si il est arrivé sinon rend la main.

Mode de communication par envoi de message, parfois qualifié d'asynchrone parce qu'il n'y a pas d'entité commune entre les communicants qui leur permettrait de se synchroniser.

Mode de communication requête/réponse, parfois qualifié de synchrone parce que l'émetteur de la requête attend la réponse avant de continuer son traitement. Il y a d'autres propriétés de l'interaction requête/réponse qui font qu'elle peut être dite asynchrone (*oneway*) ou semi-synchrone (*deferred synchronous*).

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.

- un message de la forme `operation` *`operateur opérande1 opérande2`* (par exemple `operation + 3 2`) envoyé par le client au serveur pour lui demander de réaliser une opération,

- un message de la forme `resultat` *`valeur`* (par exemple `resultat 5`) envoyé par le serveur au client pour lui retourner le résultat de l'opération.

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).

Les erreurs pouvant survenir au niveau protocole applicatif sont essentiellement dues :
- à des erreurs de codage du protocole dans le programme du client et/ou du serveur,
- à des tentatives malintentionnées du client et/ou du serveur pour trouver des failles dans le programme de son vis-à-vis.

Erreur dans la transmission de la demande du client vers le serveur :
- commande inconnue (ne commence pas par `operation`),
- opérateur inconnu (ex : le client envoie `operation % 5 8`),
- opérande1 n'est pas un nombre,
- opérande2 n'est pas un nombre,
- message mal formé (il manque au moins une opérande et/ou l'opérateur et/ou la commande).

Demande correcte mais erreur suite au traitement demandé :
- division par zéro,
- dépassement de capacité dans l'opération.

On définit un nouveau message permettant au serveur de signaler ces erreurs au client. :
- message de la forme `erreur transmission|traitement cause` (par exemple erreur traitement « division par zéro »).

On peut également envisager des erreurs dans la transmission de la réponse du serveur au client :
- commande inconnue (ni résultat, ni erreur),
- message mal formé.

Dans ces cas, le client doit resoumettre sa demande au serveur.

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

```
Serveur
tant que vrai
      attendreCx()
      commande := recevoir()
      si commande ok alors
            effectuer opération
            si résultat ok alors
                  envoyer( « resultat » + résultat opération )
            sinon
                  envoyer( « erreur traitement » + cause )
            fin si
      sinon
```

```
            envoyer( « erreur transmission » + cause )
      fin si
      fermerCx()
fin tant que

Client
ouvrirCx(serveur)
envoyer( « operation * 8.6 1.75 » )
reponse := recevoir()
si reponse = « resultat .. » alors
      afficher( valeur resultat )
sinon si reponse = « erreur .. » alors
      afficher( cause erreur )
sinon
      afficher( « message de reponse inconnu » )
fin si
fermerCx()
```

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

Aucune donnée n'a besoin d'être sauvegardée sur le serveur. Il se contente d'effectuer les opérations et de retourner le résultat. Il s'agit d'un serveur de calcul « pur ». Le serveur est donc sans état.

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

Un protocole est avec état lorsque les messages reçus (respectivement émis) doivent s'enchaîner dans un ordre précis sur le serveur ou le client. Il n'y a ici qu'un seul message émis et un seul message reçu, donc il n'y a pas de problématique d'état et le protocole et sans état.

Le protocole est avec état lorsque typiquement, le serveur et/ou le client doivent gérer un diagramme état/transition pour orchestrer les échanges et qu'il est donc nécessaire de sauvegarder l' « état » de ce diagramme (i.e. la position atteinte dans ce diagramme).

Le but de la question est d'expliquer cette double acceptation du terme état selon que l'on parle du serveur ou du protocole.

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.

Le fait que le serveur soit sans état ne pose pas de problème particulier : deux clients ne risquent pas de faire des modifications concurrentes sur la même donnée.
Il suffit de faire en sorte que le bloc d'instructions allant de `commande := recevoir()` à `fin tant que` exclus s'exécute dans un processus ou un *thread*.

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).

Le serveur ne peut plus fermer autoritairement la connexion après avoir traiter une opération. Il se peut en effet que le client ait besoin de lui transmettre d'autres demandes dans la foulée.

Solution avec modification du protocole

Dans un premier temps on peut demander aux clients de signaler la fin des demandes d'opérations par un message. Le serveur doit donc boucler sur les réceptions de commande tant que la commande `fin` n'est pas envoyée.

```
Serveur
tant que vrai
      attendreCx()
      commande := recevoir()
      tant que commande != « fin »
            si commande ok alors
            ..
            commande := recevoir()
      fin tant que
      fermerCx()
fin tant que

Client
ouvrirCx(serveur)
envoyer( « operation * 8.6 1.75 » )
reponse := recevoir()
si reponse = ..
envoyer( « operation - 13 15 » )
reponse := recevoir()
si reponse = ..
envoyer( « fin » )
fermerCx()
```

Solution sans modification de protocole

La solution précédente présente l'inconvénient de modifier le protocole. Les « anciens » clients ne seront donc plus utilisables.

Après réception d'une commande, le serveur garde sa connexion ouverte et se met en attente de commandes. Il se peut que le client ait à en envoyer de nouvelles ou qu'il ait terminé. Conserver une *socket* ouverte ayant un coût, le serveur ne peut le faire indéfiniment. On décide donc qu'à l'issue d'un délai de garde fixé (par exemple 10s), le serveur ferme sa connexion. Le client a donc 10s pour envoyer une nouvelle commande. Dans la majorité des cas, ce délai est suffisant et permet d'économiser les temps d'ouverture et de fermeture de connexion (dans le cas de TCP, pas si négligeable que ça par rapport au coût de transfert d'un volume faible de données – la majeure partie des fichiers transférés sur le Web font moins de 10K). Les clients dépassant l'expiration du délai doivent rouvrir une nouvelle connexion (comme pour un transfert complètement différent).

```
Serveur
tant que vrai
      attendreCx()
      commande := recevoir() || attendre délai
      tant que non( expiration du délai )
            si commande ok alors
            ..
            commande := recevoir() || attendre délai




      fin tant que
      fermerCx()
```

```
fin tant que
```

On notera que c'est la solution retenue par HTTP/1.1 (le délai de garde est appelé *keep-alive*).

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 ?

La classe `java.net.ServerSocket` fournit une méthode bloquante `accept` qui attend l'arrivée de demandes de connexions sur un port TCP. Dès qu'une demande arrive, cette méthode l'accepte et retourne une instance de la classe `java.net.Socket`.

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

La classe `java.net.Socket` fournit une méthode `getInputStream` qui retourne une instance de la classe `java.io.InputStream`. Les lectures peuvent se faire alors selon les modalités habituelles de gestion des entrées/sorties en Java.

On remarquera en particulier, qu'une opération courante consiste à vouloir lire ligne à ligne en mode caractère à partir d'une *socket* (ou de tout autre `java.io.InputStream`). Il faut pour cela :
- créer un flux en mode caractère à partir de l'instance de `InputStream` (qui est en mode binaire),
- créer un flux bufférisé en mode caractère.

Soit :

```
InputStream is = aSocket.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
```

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

La classe `java.net.Socket` fournit une méthode `getOutputStream` qui retourne une instance de la classe `java.io.OutputStream`. Les écritures peuvent se faire alors selon les modalités habituelles de gestion des entrées/sorties en Java.

On remarquera en particulier, qu'une opération courante consiste à vouloir écrire une chaîne de caractères dans une *socket* (ou tout autre `java.io.OutputStream`). Il faut pour cela créer une instance de la classe `java.io.DataOutputStream` qui fournit la méthode `writeBytes(String)`.

```
OutputStream os = aSocket.getOutputStream();
DataOutputStream dos = new DataOutputStream(os);
dos.writeBytes("maChaine\n"); // \n nécessaire pour vider le buffer d'envoi
```

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 en français ou à l'aide d'un *framework* (par exemple JUnit) les scenarii de test qui permettront de s'assurer du bon fonctionnement de votre TP.

2. 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).

Tests. Votre code doit posséder des tests automatiques pour vérifier son bon fonctionnement. Le *framework* de tests unitaires JUnit est conseillé, mais d'autres choix sont possibles. Vous montrerez votre suite de tests lors de la séance d'évaluation.

3. 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`

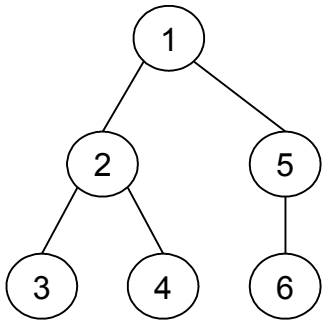
4. Enfin, rajouter les requêtes `PWD, CWD, CDUP`.
`PWD` : permet à l'utilisateur de connaître la valeur du répertoire de travail distant.
`CWD directory` : permet à l'utilisateur de changer de répertoire de travail distant.
`CDUP` : cette commande est équivalente à `CWD ..`

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 évite l'inondation, et on ne contacte que les sites concernés.

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).

```
DatagramSocket ds = new DatagramSocket(5000);
DatagramPacket dp = new DatagramPacket(new byte[2048],2048);
ds.receive(dp);
for( int i=0 ; i < fils.length ; i++ ) {
   new Thread() {
     public void run() {
       byte[] data = dp.getData();
       DatagramPacket dp2 =
         new DatagramPacket(data,data.length,fils[i],5000);
       ds.send(dp2);
   }} . start();
}
ds.close();
```

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.

```
DatagramSocket ds = new DatagramSocket(5000);
DatagramPacket dp = new DatagramPacket(new byte[8],8);
for( int i=0 ; i < fils.length ; i++ ) {
  ds.receive(dp);
}
dp = new DatagramPacket(new byte[8],8,pere,5000);
ds.send(dp);
```

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 ?

Un protocole est avec état si les requêtes provenant d'un même client doivent s'enchaîner selon un certain ordre (conduisant côté serveur à la gestion d'un graphe de transitions entre états). Inversement, un serveur est sans état si deux requêtes provenant d'un même client sont indépendantes.
Ici, les requêtes des clients sont des demandes de transfert de fichiers : elles sont indépendantes les unes des autres. Le protocole est sans état. Il n'y a aucune donnée à stocker d'une requête à l'autre.

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

Oui. Le transfert d'un fichier ne modifie pas les données du serveur. Il n'y a donc pas de problèmes à effectuer plusieurs transferts (y compris d'un même fichier) simultanément.

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 ?

Le client doit indiquer s'il souhaite un transfert de fichier depuis le début du fichier ou depuis le point de reprise numéro i.

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.

Tolérance aux pannes : ce mécanisme permet au serveur de détecter les pannes de client.
Contrôle de flux : le fait d'attendre l'acquittement du client, permet de s'assurer que celui-ci a eu le temps de traiter le message `PRep` avant de continuer le transfert.

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 ?

Il y a maintenant un 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.

Deux solutions ont été acceptées :

```
client -> serveur : TRANSFERT fic 0 0
serveur -> client : DATA 0 0, PRepMin, DATA 0 1, PRep 0
client -> serveur : PRepAck
serveur -> client : DATA 1 0, PRepMin, DATA 1 1, PRep 1
client -> serveur : PRepAck
serveur -> client : DATA 2 0, Fin
```

ou

```
client -> serveur : TRANSFERT fic 0 0
serveur -> client : PRep 0
client -> serveur : PRepAck
serveur -> client : DATA 0 0, PRepMin, DATA 0 1, PRep 1
client -> serveur : PRepAck
serveur -> client : DATA 1 0, PRepMin, DATA 1 1, PRep 2
client -> serveur : PRepAck
serveur -> client : DATA 2 0, Fin
```

La 1ère solution sous-entend qu'une demande de transfert à partir de `0 0` est en fait une demande de transfert pour le point de reprise précédant `0 0`. Ce n'est pas forcément la solution la plus claire.
La 2ème solution pose moins de problème puisqu'une demande de transfert à partir de `0 0` signifie bien que l'on souhaite transférer le fichier depuis le début. L'énoncé ne comporte pas néanmoins la mention explicite que tout transfert commence par la pose d'un point de reprise majeur : d'où la prise en compte des deux solutions.

Par contre, il s'agit bien ici d'indiquer les messages échangés entre le client et le serveur dans le cadre de **ce protocole**. Les primitives (par exemple celle de demande d'ouverture de connexion) génèrent certainement des messages au niveau transport, mais l'énoncé ne mentionne en aucun cas ces messages. Les seuls messages cités sont : `TRANSFERT`, `DATA`, `PRep`, `PRepAck`, `PRepMin` et `FIN`. Il n'y a donc pas lieu de mentionner sur le diagramme de messages correspondant à l'exécution des primitives.

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()`.

Rq : la solution ci-dessous ne vérifie pas les différents cas d'erreur qui peuvent se produire dans les messages attendus.

```
client(fic,i,j)
ouvrirCx(serveur)
envoyer("TRANSFERT "+fic+" "+i+" "+j)
message = recevoir()
tant que message != "FIN" faire
  si message.startsWith("PRepMin") alors
    // rien
  sinon si message.startsWith("PRep") alors
    envoyer("PRepAck")
  sinon si message.startsWith("DATA") alors
    // récupérer x et y (entiers suivant DATA)
    // écrire dans fic à l'emplacement x*100+y*50 le block de données
  fin si
  message = recevoir()
fait
fermerCx()

serveur()
cx = attendreCx()
message = recevoir()  // on doit recevoir TRANSFERT fic i j
nb = nbBloc100K(fic)
for( int n=i ; n < nb ; nb++ )
  // lire le n-ème bloc de 100KB de fic
  si j=1 && n=i alors
    // on saute cette partie si on a demandé un transfert à partir du point
    // de reprise mineur
    envoyer("PRep "+n)
    message = recevoir()  // on doit recevoir PRepAck
    envoyer("DATA "+n+" 0 "+ /* la 1ère moitié du bloc */)
  finsi
  envoyer("PRepMin")
  envoyer("DATA "+n+" 1 "+ /* la 2ème moitié du bloc */)
envoyer("FIN")
fermerCx()
```

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.

11 caractères dans la chaîne (y compris l'espace). Les caractères sur 1 octet ne sont pas affectés par le format (seuls les mots de 2 octets le sont).

```
A           11 00 hello world
B           00 11 hello world
```

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

B extrait la longueur de la chaîne, et étant grand boutiste, considère que le 1er octet (11) est l'octet de poids fort, tandis que le 2nd (00) est l'octet de poids faible. Elle s'attend donc à trouver à la suite 11*256 + 0 caractères (ce qui bien évidemment n'est pas le cas).

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 ?

Après inversion, la machine B obtient la suite `00 11 ehll oowlr d`. Le nombre de caractères attendu est bien de 11 cette fois, mais les caractères sont inversés 2 à 2.

Conclusion : un mécanisme d'encodage/désencodage des données ne peut se contenter de faire des conversions automatiquement lorsqu'il s'agit de transférer de l'information entre machines ayant des conventions de représentation différentes. Le type des données (ici entier ou caractère) doit être pris en compte pour savoir quelle règle appliquer. Les mécanismes

adoptent en général un format pivot et font (ou ne font pas) la conversion en fonction des machines.

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 ?

NB : le but est de faire en sorte que les étudiants arrivent à décoder la grammaire fournie à partir de la page 52 de l'annexe.

Cela semble implicite dans le document : l 'encodage est grand boutiste.

```
AC ED magic : en-tête invariable identifiant le flux comme un flux Java
00 05       version du protocole de sérialisation
77          TC_BLOCKDATA (en-tête de la règle blockdata)
08          longueur du bloc de données (unsigned byte)
00 00 00 11       1er int (il faut savoir que l'on s'attend à trouver un
int – le type n'est pas codé)
00 00 00 02       2nd int (idem)
```

Le 1er entier vaut 0x11 (soit 17 en décimal) et le second 0x02.

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
```

```
AC ED magic
00 05       version du protocole de sérialisation
77          TC_BLOCKDATA (en-tête de la règle blockdata)
07          longueur du bloc de données (unsigned byte)
```

```
00 05       longueur de la chaîne
48 65 6C 6C 6F    la chaîne (codage ASCII de hello)
```

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 ?

Avantage : on connaît dès le début du bloc la taille des données à venir
Inconvénient : on doit attendre de disposer de toutes les données pour calculer la taille du bloc et renseigner la taille et l'octet contenant soit `TC_BLOCKDATA`, soit `TC_BLOCKDATALONG`.

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

```
AC ED magic
00 05       version du protocole de sérialisation
73          TC_OBJECT         content object newObject
72          TC_CLASSDESC      classDesc newClassDesc
00 07 50 6F 69 6E 74 32 44    className               (Point2D)
00 00 00 00 00 00 00 00       serialVersionUID        (long)
02          SC_SERIALIZABLE   classDescFalgs          (byte)
00 02       fields count                              (short)
4A          fieldDesc[0] primitiveDesc prim_typecode  ('J' pour long)
00 01 78    fieldName                                 (x)
4A          fieldDesc[1] primitiveDesc prim_typecode  ('J' pour long)
00 01 79    fieldName                                 (y)
78          TC_ENDBLOCKDATA   classAnnotation endBlockData
70          TC_NULL           superClassDesc classDesc nullReference
00 00 00 00 00 00 00 12       valeur de x             (18 décimal)
00 00 00 00 00 00 00 14       valeur de y             (20 décimal)
```

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 ?

Alors que les blocs ne contiennent aucune information sur les types de données présents, le codage d'un objet embarque une représentation de la classe, de ses champs et de leurs types. On est ainsi à même de reconstituer une instance en lisant le flux (il faut néanmoins être capable de refaire l'association entre un `serialVersionUID` et le bytecode de la classe), alors

que la lecture d'un flux comportant un bloc ne nous apprend rien si on ne connaît pas les types stockés.
C'est une différence majeure entre :
- des souches compilées ou statiques, qui savent quels types de données extraire du flux,
- des souches interprétées ou dynamiques, qui tirent les informations concernant les types du flux lui-même.
Les premières sont plus performantes mais dédiées à un profil de types donné : si l'on veut décoder un nouveau profil de types, il faut une nouvelle souche. Les secondes sont moins performantes, mais génériques : elles permettent de décoder n'importe quel profil de données.

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.

```
magic:  ac ed
version du protocole de serialisation: 00 05
TC_OBJECT  content object newObject: 73
TC_CLASSDESC  classDesc newClassDesc: 72
className: chaine de carac donc 2 partie
longueur de la chaine sur 2 octets -> 00 08
les caracteres correspondant a Etudiant, 1 octet par caractere
 ->  45 74 75 64 69 61 6e 74
serialVersionUID (long): 99 7d 3c 17 6f 9d 44 e2
SC_SERIALIZABLE   classDescFalgs (byte): 02
fields count (short): 2 champs donc -> 00 02
fieldDesc[0]: primitiveDesc prim_typecode, on a un int donc I ->  49
fieldName pour "note" (longueur + carac) :  00 04 6e 6f 74 65
fieldDesc[1]: objectdesc obj_typecode : L -> 4c
fieldName pour "nom" (longueur + carac) : 00 03 6e 6f 6d
(String)objet
TC_STRING 74
Longeur + carac 00 12 4c 6a 61 76 61  2f 6c 61 6e 67 2f 53 74  72 69 6e 67
3b
12 Ljava/lang/String
TC_ENDBLOCKDATA   classAnnotation endBlockData: 78
TC_NULL           superClassDesc classDesc nullReference: 70
```

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

```
class Etudiant implements Seriablizable {
      private int note;
      private String nom;
}

valeur de note (16 en hexa)    :  00 00 00 10
TC_STRING:  74

valeur de nom (6 caractères, Durant): 00 06 44 75  72 61 6e 74
```

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 – RMI

1. Service 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 ?

Une interface déclare des services tandis que l'implantation fournit le code associé à ces services. En client/serveur, les interfaces permettent de déclarer les services accessibles à distance par les clients sans que ces derniers en connaissent les détails internes de réalisation.

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.

Une interface Java RMI doit étendre l'interface `java.rmi.Remote` (∈ aux librairies internes du JDK) et chaque méthode de l'interface doit lever l'exception `java.rmi.RemoteException`.
Un objet serveur Java RMI doit étendre la classe `java.rmi.server.UnicastRemoteObject` et implanter une interface Java RMI. Elle doit fournir au moins un constructeur levant l'exception `java.rmi.RemoteException`.

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

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

interface HotelItf extends Remote {
   int reserver( String client, String hotel, String date, int nbChambres )
            throws RemoteException;
   void annuler( int numReservation ) throws RemoteException;
   String lister( int numReservation ) throws RemoteException; }
```

4. Expliquer le rôle et le fonctionnement de l'annuaire `rmiregistry`.

L'annuaire stocke des associations entre des noms logiques et des adresses (références) physiques d'objets RMI. Les objets serveur s'y enregistrent, et les objets clients viennent y rechercher les références des objets serveur avec lesquels ils souhaitent interagir.

5. 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.

La méthode `reserver` modifie l'état du serveur. Elle ne concerne néanmoins qu'un seul hôtel à la fois. Les données concernant les hôtels étant stockées indépendamment les unes des autres, on peut donc avoir simultanément plusieurs réservations pour des hôtels différents, mais pas deux réservations pour le même hôtel.
La méthode `lister` peut quant à elle, être exécutée simultanément dans n'importe quelle condition car elle ne modifie pas l'état. Cependant elle ne peut pas être exécutées en même temps que la méthode `reserver` qui, elle, modifie l'état.

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.

6. 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.

client -> agence de voyage
        |-> compagnie aérienne
        |-> chaîne hôtelière

7. 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).

Client est client de l'agence de voyage.
L'agence de voyage est client/serveur : serveur pour le « client » et client de la compagnie aérienne et de la chaîne hôtelière.
La compagnie aérienne est serveur pour l'agence de voyage.
La chaîne hôtelière est serveur pour l'agence de voyage.

8. 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 plus avion. Faire une description de ce fonctionnement sous forme d'un diagramme de séquences entre les objets RMI représentant l'agence de voyage, la compagnie aérienne et la chaîne hôtelière.

2. 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.

En phase d'abonnement/désabonnement; l'observer est client de `subject` et `subject` est serveur d'Observer.
En phase d'observation, observer est serveur de `subject` (ou plusieurs) et `subject` est client d'observer.

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

```
interface Observer extends Remote {
  public void notification( String msg ) throws RemoteException;
}

interface Subject extends Remote {
  public void attach( Observer ref ) throws RemoteException;
  public void detach() throws RemoteException;
}
```

L'interface `Observer` dispose de la méthode `notification` qui sera appelée par le sujet auprès duquel l'observateur s'abonne par la méthode `attach`. Si un observateur s'abonne à plusieurs sujets, il est nécessaire de passer en paramètre de notification quel sujet a été notifié.

L'interface `Subject` dispose de deux méthodes `attach` et `detach` qui permettent aux `observers` de d'abonner et de se désabonner du `subject`. Il est nécessaire de passer en paramètre l'observer souhaitant s'abonner pour pouvoir appeler par la suite la méthode `notify`.

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.

```
public class SubjectImpl extends UnicastRemoteObject implements Subject {
      protected List observers = new ArrayList();
      public void attach(Observer obs) {
            synchronized(observers) {
                  observers.add(obs);
            } }
      public void attachAll(Observer[] obs) {
            synchronized(observers){
                  int i;
                  for(i=0;i<obs.length;i++)
                        observers.add(obs[i]);
            } }
      public void detach(Observer obs) {
            synchronized(observers) {
                  observers.remove(obs);
            } }
      public void detachAll(Observer[] obs) {
            synchronized(observers) {
                  int i;
                  for(i=0;i<obs.length;i++)
                        observers.remove(obs[i]);
            } }
      public void notifications() {
            synchronized(observers) {
                  int i;
                  for(i=0;i<observers.length;i++)
                        observers[i].notification("Un message",this);
            } }

public class ObserverImpl extends UnicastRemoteObject implements Observer {
      public void notification(String msg, Subject sub) {
            System.out.println("notification de "+sub+": "+msg);
      } }
```

Il s'agit ici d'écrire les programmes qui permettront d'instancier sur deux machines différentes les objets `subject` et `observer`.

```
public class InstallSubject {
      public static void main(String[] args){
            Subject obj = new SubjectImpl();
            Naming.rebind("subject",obj);
      } }

public class InstallObserver {
      public static void main(String[] args){
            Observer obj = new ObserverImplImpl();
            Naming.rebind("observer",obj);
      } }

public class Client {
      public static void main( String[] args ) throws Exception {
            Subject subject = (Subject) Naming.lookup("subject");
```

```
            Observer observer = (Observer) Naming.lookup("observer");
            subject.attach(observer);
            for(int i=0;i<10;i++)
                  subject.notifications();
            subject.detach(observer);
      } }
```

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. Ce TP est à réaliser avec le squelette de code car-tp2.zip disponible en téléchargement à partir du module Moodle associé à l'UE.

1. 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. Pour tester et débugger votre passerelle REST, vous pouvez utiliser un client HTTP en ligne de commande, par exemple `curl`.

1. Écrire en français ou à l'aide d'un *framework* (par exemple JUnit) les scenarii de test qui permettront de s'assurer du bon fonctionnement de votre TP.

2. 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.

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

POST dépôt d'un nouveau fichier, PUT remplacement d'un fichier existant. Notons que dans les deux cas, cela se traduit par un PUT FTP.

4. 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.

Utiliser une expression régulière pour récupérer le nom du répertoire : `@Path("{path: .*}")`

5. 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.

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

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

Formulaire avec login et mot de passe vers une ressource, authentification en mode POST. Le formulaire peut être retourné par la ressource en mode GET.

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 6 – 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'essentiel est de leur faire comprendre l'utilité d'une adresse logique par rapport à une adresse physique et le besoin qui en découle de stocker des associations entre une adresse logique et une adresse physique. Ce qui est le rôle d'un annuaire. Arrivé à ce stade, j'ai abordé plusieurs fois cette notion en cours et elle ne devrait pas poser de problème.

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

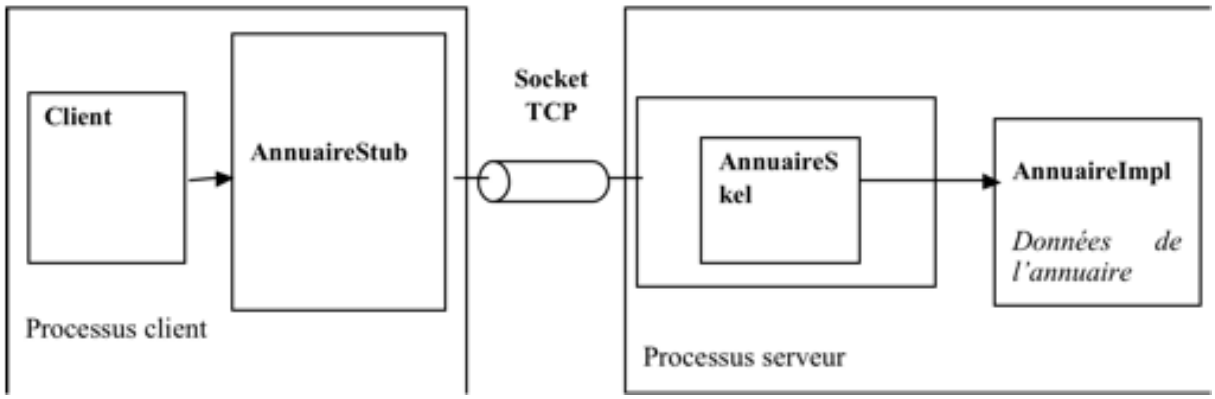


Figure 1 : Relation entre les objets de l'application répartie

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

```
public class Adresse implements Serializable {
  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 `java.io.ObjectInputStream` et `java.io.ObjectOutputStream` 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
ObjectInputStream fluxIn = ObjectInputStream(unInputStream);

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

// Création d'un flux d'écriture de données binaires
ObjectOutputStream fluxOut = new ObjectOutputStream(unOutputStream);

// L'écriture de données sur un flux
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 décodées.

Assez intuitivement, il suffit de définir un message d'appel et un message de retour par méthode définie dans l'interface Annuaire. On choisit de faire débuter les messages d'appel par une chaîne fournissant le nom de la méthode appelée et on les fait suivre par les paramètres de l'appel. Quant au message de retour, il suffit de transmettre la valeur retournée.

Message d'invocation de la méthode ajouter :
- `String "ajouter"`
- `String adresse logique`
- `String adresse physique.addresse`
- `int adresse physique.port`

Message de retour de la méthode ajouter :

- `boolean résultat`

Même principe pour modifier, retirer, chercher. Pour lister, seule difficulté, transmettre la taille du tableau :

Message d'invocation de la méthode lister :
- `String "Lister"`

Message de retour de la méthode lister :
- `int taille du tableau`
- autant de `String` qu'il y a d'élément dans le tableau à retourner.

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 ?

Il s'agit de séparer le code générique permettant d'accéder à l'annuaire (`AnnuaireStub`), du code particulier contenant le comportement de ce client. La classe `AnnuaireStub` est ainsi réutilisable dans tout programme devant accéder à distance à l'annuaire.

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`.

```
class Client {
  public static void main( String[] args ) {
    Annuaire a = new AnnuaireStub() ;
    a.ajouter("car",new Adresse("rac.lifl.fr",5896);
    a.ajouter("m1",new Adresse("master.lifl.fr",698);
    a.lister();
  }
}

class AnnuaireStub implements Annuaire {
  Socket s;
  InputStream is;
  OutputStream os;

  public AnnuaireStub() {
    s = new Socket("annuaire.lifl.fr",89);
    is = s.getInputStream();
    os = s.getOuputStream();
  }

  public boolean ajouter(String adresseLogique, Adresse adressePhysique) {
    os.writeUTF("ajouter");
    os.writeUTF(adresseLogique);
    os.writeUTF(adressePhysique.adresse);
    os.writeInt(adressePhysique.port);
    boolean ret = is.readBoolean();
    return ret;
```

```
  }

  public String[] lister() {
    os.writeUTF("lister");
    int l = is.readInt();
    String[] tab = new String[l];
    for( int i=0 ; i < l ; i++ )
      tab[i] = is.readUTF();
    return tab;
  }

  // ... pour les autres méthodes
}
```

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 ?

Il s'agit de séparer le code générique permettant de recevoir des requêtes pour l'annuaire (`AnnuaireSkel`), du code particulier implémentant le comportement « réel » de l'annuaire (`AnnuaireImpl`). On peut noter qu'il y a de nombreuses façons d'implémenter l'annuaire : avec un `HashMap` en mémoire, avec un fichier, etc. La classe `AnnuaireImpl` peut ainsi varier tandis que la classe `AnnuaireSkel` reste inchangée.

Note : le suffixe `skel` est inspiré de CORBA qui parle de *skeleton* côté serveur.

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

Rq : il s'agit de faire comprendre qu'ici la classe `AnnuaireSkel` joue le rôle d'un serveur et que l'on peut donc retrouver toutes les problématiques des codes serveurs : *multi-threading*, *pool*, etc.

```
class AnnuaireSkel {
  public static void main( String[] args ) {
    Annuaire annuaire = new AnnuaireImpl();
    ServerSocket serv = new ServerSocket(89);
    while(true) {
      Socket s = serv.accept();
      InputStream is = s.getInputStream();
      OutputStream os = s.getOutputStream();
      String method = is.readString();
      if( method.equals("ajouter") ) {
        String adresseLogique = is.readUTF();
        String ap_adresse = is.readUTF();
        String ap_port = is.readInt();
        Adresse a = new Adresse(ap_adresse,ap_port);
        boolean ret = annuaire.ajouter(adresseLogique,a);
        os.writeBoolean(ret);
      }
      else if( method.equals("lister") {
        //  ...
      }
    }
  }
}
```

```
class AnnuaireImpl extends HashMap<String,Adresse> implements Annuaire {
  ...
}
```

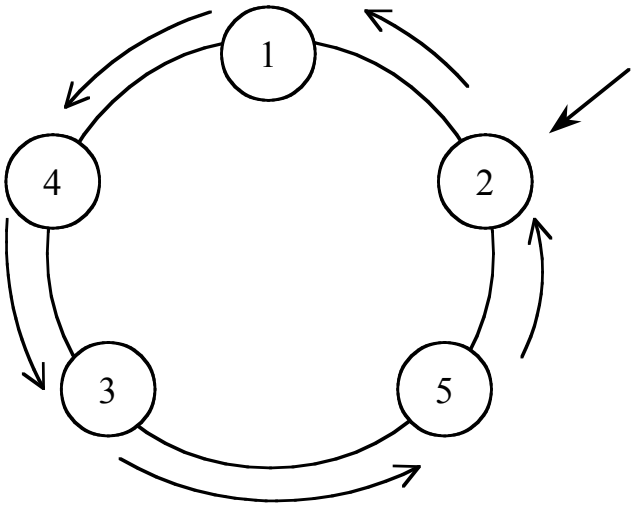
7. Parmi les quatre classes précédentes, quelles sont celles dont le code pourrait être généré automatiquement ? Quelle information minimale a-t-on besoin pour cela ? Proposer en pseudo-code un algorithme pour cela.

Il s'agit des classes `AnnuaireStub` et `AnnuaireSkel`. On a besoin simplement de l'interface `Annuaire`.

Générer prologue
Itérer sur chaque méthode …
Générer épilogue

TD 7 – Élection sur un anneau

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 ?

Le test d'arrêt correspond à la mise en évidence que le successeur du site est l'initiateur.

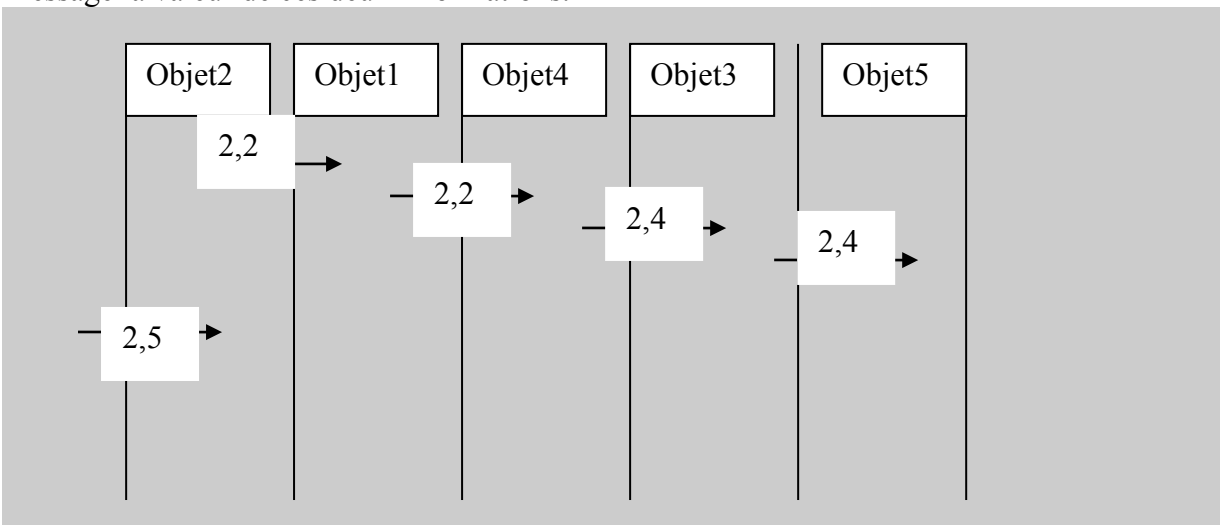
2. Représenter sur un diagramme les échanges de messages engendrés par l'exécution d'une élection avec l'objet 2 comme initiateur.

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 ?

Le test de décision de l'élu est quand le site initiateur reçoit le message contenant son propre identifiant + l'identifiant du plus grand. Il faut donc inclure dans le message l'identifiant du plus grand site.
L'algorithme est alors le suivant :
N'importe quel processus peut commencer une élection. Il est alors initiateur. Il marque son état comme participant et il place son identifiant dans un message « élection ». Il envoie celui-ci à son voisin.
Quand un processus reçoit un message « élection », il compare l'identifiant reçu avec son propre identifiant. Si l'identifiant reçu est supérieur à son propre identifiant, alors il envoie le message au voisin, sinon il échange l'identifiant reçu avec son propre identifiant. Il fait

uniquement une substitution s'il n'est pas encore participant. En faisant suivre le message, il marque sont état à »participant ».

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 connaît/connaissent l'identifiant de l'élu ? Pourquoi ?

Seul, l'initiateur connaît l'élu. Il faut continuer le tour jusqu'à l'élu lui-même de façon à ce que tous sachent qu'il est élu ainsi que lui-même.

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.

Il faut continuer le tour jusqu'à l'élu lui-même de façon à ce que tous sachent qu'il est élu ainsi que lui-même. Dans le cas synchrone, il faut que le retour de la méthode contienne le résultat (faire le schéma au tableau).

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.

```
public interface Participant extends java.rmi.Remote {
    public int election( int max, int initiateur )
    throws java.rmi.RemoteException; }
```

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

Je n'ai pas fais la solution asynchrone… à vous de la faire.

```
public class ParticipantImpl extends UnicastRemoteObject
implements Participant {
 /** identifiant du site */
 protected int noInterne;

 /** site suivant */
```

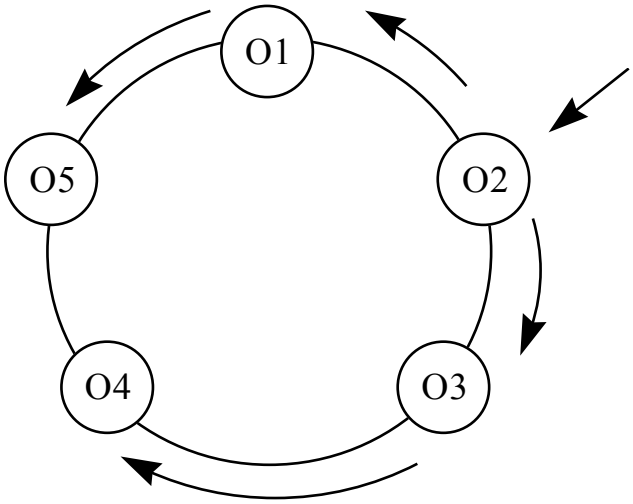
```
 protected int suivantInterne;

 /**
  * Initialisation de l'anneau
  *
  * @param              no du site
  * @param Participant  reference de l'objet suivant
  * @return             le participant elu
  */
 public ParticipantImpl( int no, Participant suivant )
 throws java.rmi.RemoteException {
    noInterne = no;
    suivantInterne = suivant; }

  /**
   * Election d'un participant de plus grand no
   *
   * @param max le no du participant de no max
   * @return    le participant elu
   */
 public int election( int max, int initiateur )
 throws RemoteException {
    if ( max < noInterne ) {
        max = noInterne;
    }
    if (suivantInterne == initiateur) {
      return max;
    }
    else {
      return suivantInterne.election(max,initiateur); } } }
```

TD 8 – Élection contrarotative sur un anneau

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 deux 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).

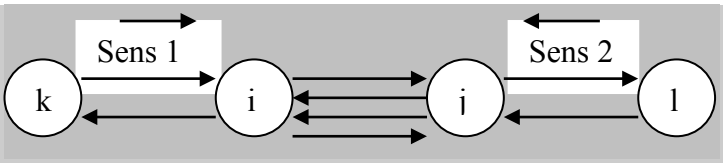
Test d'arrêt de la propagation

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 ?

Non, car cela dépend à la fois de la vitesse de transfert des messages sur le réseau, mais aussi de la vitesse de traitement du message par chaque objet.

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.



Le site i a reçu la visite du sens 1 avant la visite du sens 2
Il propage la visite du sens 1 vers le site j et il termine la propagation du sens 2,
Aucun site k en amont de i sur le sens 1 ne recevra la visite du sens 2
Le site j avait propagé vers le site i du sens 2. Il avait donc reçu ce sens là. Il recevra obligatoirement la visite du sens 1 après la visite du sens 2
Aucun site l en amont de j ne recevra la visite du sens 1.

Remarque : cette approche organise la rencontre des vagues en général. Un cas particulier consiste à ce que, sur un site j, il y ait détection simultanément des visites 1 et 2. Le site j serait alors capable de terminer les deux vagues en organisant le reflux.
Une telle vision est possible mais rare.
Soit on reçoit une visite et on se bloque en attente de l'autre. Ceci va limiter probablement le parallélisme des propagations (puisqu'on produit une attente). On pourrait le faire si on connaissait exactement la topologie de l'anneau sur le site diamétralement opposé à l'initiateur.
Soit on laisse à la chance le soin de faire débuter à peu près ensemble les deux rotations. Ceci permet de détecter leur simultanéité avant qu'elles soient propagées. Ce qui est peu probable. L'une des deux vagues aura été propagée et l'on retombe dans la solution des croisements sur deux objets.

Échanges de messages

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

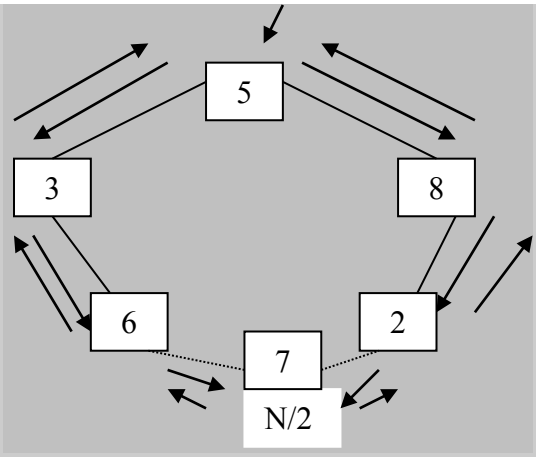
Evaluation de performances

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 ?

Si on lance l'élection dans les deux sens on va avoir deux vagues contrarotatives (tournant en sens contraire) qui vont se rencontrer en un point de l'anneau qui est plus ou moins diamétralement opposé au site de l'initiateur.

Le schéma de parcours devient le suivant.



Il faut tenir compte des vitesses de propagation des appels en cascade sur l'anneau et il est possible que l'un des sens se propage plus vite que l'autre de sorte que le site de rencontre n'est pas forcément le site diamétralement opposé à l'initiateur. Mais il s'agira d'un site dont la position moyenne sur un grand nombre d'expérience est le site diamétralement opposé (sauf si l'anneau n'est pas homogène dans son fonctionnement)

Le site diamétralement opposé est à la position N/2. On aura donc un temps de réponse en O(N/2)T et par contre un nombre d'appels distants en O(N).
Si on se lance dans une élection dans un seul sens on doit boucler un tour de l'anneau pour connaître la solution. Si on a N sites on exécute un appel local et N-1 appels distants soit un temps de réponse de l'ordre de (N-1)T si T est le temps moyen d'exécution d'un appel distant.
On a dons un nombre d'appels distants en O(N) et un temps de réponse en O(N)T.

Interfaces

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

```
public interface Election extends Remote {

    /** un des deux sens **/
    public static final int SENS1 = 0;

    /** l'autre sens **/
    public static final int SENS2 = 1;

    /** choix du voisin de droite
     * @param others le voisin de droite
     */
    public void fixerVoisinDroite( Election other ) throws RemoteException;

    /** choix du voisin de gauche
     * @param others le voisin de gauche
     */
    public void fixerVoisinGauche( Election other ) throws RemoteException;

    /** permet de participer à l'élection
     * @param num un candidat pour l'élection
     * @return le meilleur candidat
     */
    public int election( int num, int sens ) throws RemoteException;

    /** fixe le serveur comme initiateur et lance l'election */
```

```
    public void initiateur() throws RemoteException;


    /** envoie le résultat a tous les participants
     * @param res le résultat de l'élection
     */
    public void envoieResultat( int res ) throws RemoteException;

    /** permet de récuperer l'identifiant du serveur
     * @return l'ID du serveur
     */
    public int getID() throws RemoteException;

     /** pour savoir si le serveur a deja participé a l'election
      * @return vrai si il a déjà participé et faux sinon
      */
    public boolean participant() throws RemoteException; }
```

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.

Il faut différencier l'initiateur de l'objet classique. En effet l'initiateur va lancer en parallèle l'élection alors qu'un objet classique ne lance l'élection que dans un seul sens.

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.

Pour construire le nouvel algorithme, il faut donc :

- lancer deux vagues contrarotatives en parallèle si on est l'initiateur. On passe en paramètre pour chaque vague un identificateur du sens dans lequel elle doit tourner

- lorsque l'on reçoit une vague et que celle-ci correspond à une première visite on est dans le cas d'une propagation simple. Il faut donc enregistrer en exclusion mutuelle l'ordre des visites. On propage cette vague dans le sens où elle circulait.

Lorsque sur un site on a déjà été visité une première fois on a donc à coup sûr propagé la vague. Il faut alors s'arrêter de propager et faire refluer la dernière vague.

```
public class ElectionImpl extends UnicastRemoteObject implements Election {

    private int id;
    private Election voisin_g, voisin_d;
    private visite = false;
    private int max;

    public ElectionImpl() throws RemoteException  { super(); }

    public void init( int id, Election voising_g, Election voisin_d )
    throws RemoteException {
      this.id = id;
      this.voisin_d = voisin_d;
      this.voisin_d = voisin_d;
    }
```

```
    public int election( int max, int sens ) throws RemoteException {

      synchronized(this) {
        this.max = max > this.max ? max : this.max;
        if( visited == true ) {
          return max;
        }
        visited = true;
      }

      Election suivant = sens == SENS1 ? voisin_g : voisin_d;
      return suivant.election(max,sens);
   }

    public void initiateur() throws RemoteException {
        Vague vague1 = new Vague(voisin_g,SENS1);
        Vague vague2 = new Vague(voisin_d,SENS2);
        vague1.join();
        vague2.join();
        int resultVote = Math.max(vague1.result(),vague2.result());
        System.out.println(
            "Recu : " + vague1.result() + "-" + vague2.result() );
    }

    private class Vague extends Thread {
      private Election e;
      private int sens;
      private int res;
      public Vague( Election e,int sens ) {
          this.e = e;
          this.sens = sens;
          this.start();
      }
      public void run(){
          try {
            res = e.election(ID,sens);
          }
          catch ( RemoteException exc ) {
            System.err.println(exc);
          }
      }
      public int result() { return res; } } }
```

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

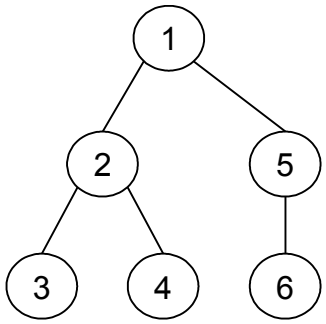
Nous avons par exemple trois initiateurs 3, 5, et 8. Le principe d'extinction sélective consiste à laisser se propager une vague qui est initiée par un site d'identifiant plus grand et de faire refluer la vague d'identifiant plus petit. On va éviter ainsi à des élections de se terminer en faisant la totalité des visites alors qu'une partie suffit.

Le côté paradoxal est atteint puisque 5 va refluer contre 8 mais va se propager au dépend 3. En tout état de cause 8 rencontre 3 qui reflue.

TP 3 – Akka – Transfert de données

Le but de ce TP est de mettre en œuvre des acteurs répartis avec Akka. Ce TP est à réaliser avec le squelette de code car-tp3.zip disponible en téléchargement à partir du module Moodle associé à l'UE.

On considère une application répartie qui permet de transférer en Akka des données à un ensemble d'acteurs organisés selon une topologie en arbre. Chaque nœud de l'arbre propage les données à ses fils. Par exemple, dans la figure ci-dessous, pour diffuser son message, l'acteur 1 l'envoie à 2 et 5, puis 2 l'envoie à 3 et 4 et 5 l'envoie à 6. Les messages sont des chaînes de caractères.



On considère dans un premier temps que les acteurs sont colocalisés dans le même système d'acteurs sur la même machine.

1. Écrire en français ou à l'aide d'un *framework* (par exemple JUnit) les scenarii de test qui permettront de s'assurer du bon fonctionnement de votre TP.

2. Écrire le programme Akka qui met 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.

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

```
// voir archive car-tp3-correction

public class NodeActor extends UntypedActor {

  private ActorRef parent;
  private ActorRef[] children;

  public void onReceive( Object message ) throws Exception {
    if( message instanceof NeighborsMessage ) {
      NeighborsMessage nm = (NeighborsMessage) message;
      this.parent = nm.parent;
      this.children = nm.children;
    }
    else if( message instanceof BroadcastMessage ) {
      BroadcastMessage bm = (BroadcastMessage) message;
      ActorRef self = getSelf();
      ActorRef origin = getSender();
      System.out.println( self.path()+" : "+bm.message);
```

```
      // Propagation vers le parent et les enfants, sauf l'emetteur
      if( parent != null && parent != origin ) {
        parent.tell(bm,self);
      }
      for (ActorRef child : children) {
        if( child != origin ) {
          child.tell(bm,self);
      } } }
      else {
        unhandled(message);
} }
```

4. On suppose maintenant que chaque acteur est localisé dans un système d'acteurs différent. Configurer votre application pour qu'elle s'exécute de façon répartie.

5. On suppose maintenant que les acteurs 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.

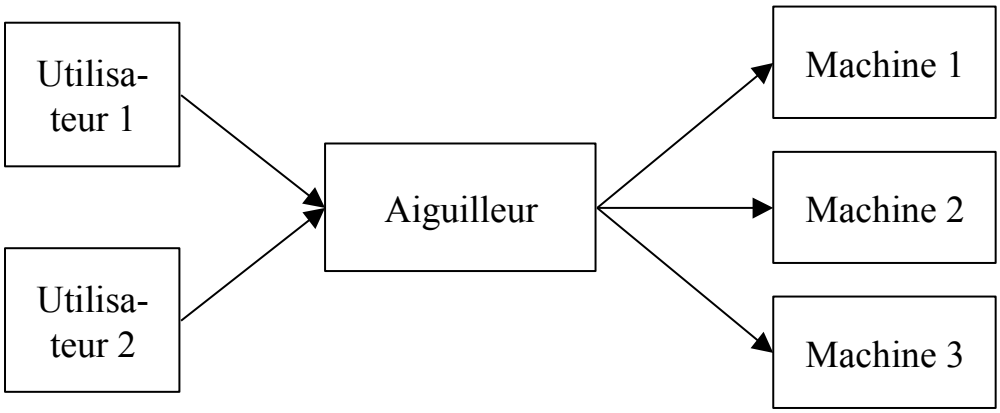
TD 9 – Répartiteur de charge

Cet exercice utilise le langage de définition d'interface Apache Thrift dont une description est fournie en annexe.

Thrift n'aura pas été vu en cours. L'idée est de demander aux étudiants, la semaine précédente, de prendre connaissance de l'annexe, puis éventuellement de répondre à quelques questions en début de TD. À vous de voir comment vous souhaitez faire : pas de question des étudiants, pas de commentaire, vous enchaînez sur l'exercice (bien évidemment il faudra les avoir prévenu de ce *modus operandi*), ou si vous êtes dans un jour magnanime.

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 répartis.
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 (utilisateur, aiguilleur, machine), dire s'ils sont client (de qui), serveur (pour qui) ou client/serveur (de qui et pour qui).

Note : cet exercice permet de vérifier que certains étudiants se mélangent complètement les pinceaux et ont du mal à assimiler les notions, en apparence pourtant simples, de client et de serveur.

Utilisateur est client de aiguilleur. Aiguilleur est serveur pour utilisateur et client de machine. Machine est serveur de aiguilleur.

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 la définition de l'interface `Machine` contenant ces 2 services.

Remarque Clément : Thrift conseille d'utiliser `typedef binary donnees` plutôt que `list<bytes>`

```
// @author Damien Cassou
service Machine {
      Donnees lecture( 1:string nomFichier ),
      bool ecriture( 1:string nomFichier, 2:Donnees donnees ) }
```

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.

L'aiguilleur doit avoir une interface identique à celle des machines. En effet, il doit être capable d'accepter les mêmes requêtes que les machines. Le traitement des requêtes est par contre différent : l'aiguilleur aiguille, tandis que les machines traitent réellement les requêtes.

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 l'interface `Controle`, réunissant ces fonctionnalités. Donner en français la signification précise de chaque élément (méthodes, paramètres, etc.) défini.

Il faut ajouter des méthodes `ajouterMachine` et `retirerMachine` au niveau de l'aiguilleur. C'est en effet lui qui gère le pool.

Remarque Damien : il n'est pas possible d'avoir un paramètre de type `Machine` avec Thrift. Réponse Romain : passer par une structure de données qui encode l'adresse sur laquelle le service distant est disponible.

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.

L'algorithme le plus simple que l'on puisse imaginer et celui du tourniquet (round-robin en anglais). Chaque machine reçoit à tour de rôle une requête (même si elle n'a pas fini de traiter la ou les requêtes qu'elle a reçues précédemment). Quand on arrive à la dernière machine, on recommence avec la première.

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 l'interface `Notification` comprenant cette fonctionnalité.

L'aiguilleur doit pouvoir être notifié. Il lui faut donc une méthode charge indiquant le niveau de charge et la machine. Cette méthode ne nécessite pas de retour et sa perte, bien que diminuant la qualité du service rendu, ne le met pas en péril : on peut donc sans problème, la déclarer `oneway`.

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

Malheureusement Thrift ne supporte pas l'héritage multiple, sinon il aurait été possible de construire une interface héritant des interfaces précédentes.

```
// @author Clément Ballabriga
service Aiguilleur {  // éventuellement on peut hériter de Machine

  //acces client
  Donnees lecture( 1:string nomFichier ),
  bool ecriture( 1:string nomFichier, 2:Donnees donnees )

  //gestion pool
  bool ajouterMachine( 1:NomMachine machine )
  bool retirerMachine( 1:NomMachine machine )

  //gestion charge/notification
  oneway void charge( 1:NomMachine machine, 2:i32 charge )
}
```

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.

Comme pour tous les problèmes d'accès en lecture/écriture sur une ressource partagée : plusieurs lecteurs ou (exclusif) un seul écrivain.

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, laquelle/lesquelles ? Si non, pourquoi ?

On peut considérer que la mise à jour d'un fichier par une machine sur une autre machine s'apparente à une opération d'écriture sur cette dernière, initiée par la première. Néanmoins il faut savoir si l'écriture provient d'un utilisateur (via l'aiguilleur) ou d'une machine (mise à jour de copie). Sinon, dans le 2ème cas, on risquerait de rediffuser la modification aux autres machines. On peut modifier l'interface `Machine` de la façon suivante :

```
enum Source {utilisateur, machine}

service Machine {
  Donnees lecture( 1:string nomFichier ),
  bool ecriture( 1:string nomFichier, 2:Donnees donnees, 3:Source src )
}
```

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.

Les deux machines font leurs mises à jour sur le fichier de façon concurrente. On aboutit à deux versions différentes du même fichier.
Pour palier à cet inconvénient, l'aiguilleur peut bloquer une requête d'écriture pour un fichier, tant qu'il y en a une autre en cours pour le même fichier (i.e. tant que la première méthode `ecriture` n'est pas terminée). Dans ce cas, aucune interface n'est modifiée, seule l'implantation de l'aiguilleur l'est.

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.

L'aiguilleur ne fait aucun traitement utile sur les réponses : il se contente de les renvoyer aux utilisateurs. On peut donc imaginer que les machines répondent directement aux utilisateurs, sans passer par l'aiguilleur.

12. Expliquer sans forcément les définir explicitement quelle(s) conséquence(s) cela entraîne sur les interfaces.

Il faut pour cela qu'elles connaissent l'identité de l'utilisateur émetteur de la requête et que celui-ci soit capable de recevoir un message d'une machine (ce n'est plus simplement un retour suite à un appel à l'aiguilleur). Il faut donc ajouter un paramètre de type `Machine` aux méthodes lecture et `ecriture`, et faire en sorte que les utilisateurs implantent une nouvelle interface avec une méthode `reponse` invoquée par les machines et contenant la réponse à leur requête.
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.

Il s'agit de faire en sorte que tous les clients ne s'adressent pas au même répartiteur sans que chaque envoi de requête nécessite une coordination entre clients.
On peut imaginer une répartition géographique : on partitionne l'ensemble des utilisateurs et en fonction de leur localisation, on assigne chaque sous-ensemble à un répartiteur qui traite les requêtes de tous les membres de ce sous-ensemble.
Une autre solution consiste à imaginer une répartition thématique : plutôt que de partitionner les utilisateurs, on partitionne l'ensemble des fichiers. Chaque sous-ensemble de fichiers est associé à un répartiteur qui traite les requêtes de tous les utilisateurs, mais uniquement pour les fichiers de ce sous-ensemble. Cela nécessite que les utilisateurs connaissent tous les répartiteurs et sachent à quel répartiteur s'adresser en fonction du fichier. On peut imaginer une organisation de type file system, dans laquelle un préfixe dans le nom du fichier permet à l'utilisateur de déterminer le répartiteur à utiliser.
On peut finalement imaginer une troisième solution combinant les deux précédents dans laquelle il y a deux niveaux de répartiteur : un premier géographique et un second thématique.

Annexe : Apache Thrift

TD 10 – Protocole de validation à deux phases

1. Protocole de validation à deux phases

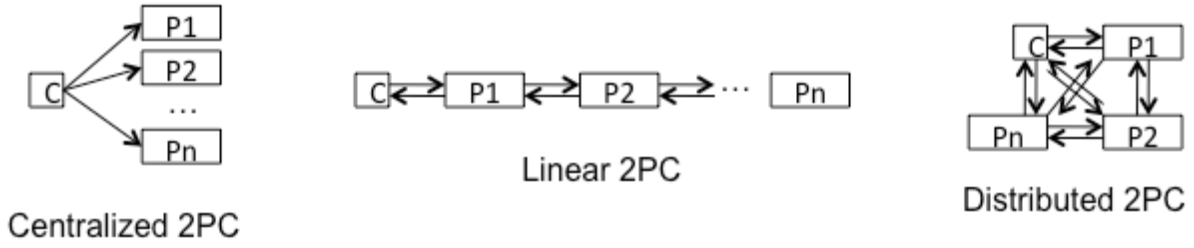
Cet exercice a pour but d'étudier la conception d'un protocole transactionnel de validation à deux phases avec des objets répartis.

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

Phase de vote puis phase d'engagement ou d'abandon.

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 :

- `beginTransaction` : 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.

Damien : `beginTransaction` a été préféré à `begin` qui est un mot-clé réservé pour Apache Thrift.

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 IDL les interfaces `TransactionItf`, `GestionItf` et `ParticipantItf`.

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

C implémente `GestionItf` et `TransactionItf`. P1, P2, … Pn implémentent `ParticipantItf`.

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.

4. 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 IDL des nouvelles interfaces que vous proposez.

Pour que le coordinateur reste générique, il faut que les appels à `credit` et `debit` aillent directement du client aux participants. Il faut que chaque participant sache quand une transaction démarre, puisse indiquer le résultat de son vote et engager ou annuler la transaction. On peut donc définir une interface `RessourceItf` avec `beginTransaction`, `vote`, `commit`, `rollback` qui sera implémentée par chaque participant.

Thrift. Remarque Damien : limitation due à l'impossibilité de lancer plusieurs processeurs (un pour `AccountThrift`, un pour `ResourceThrift`) sur un même port. Voir mail du 1/4/2014.

```
U -> C.beginTransaction()
∀i U -> C.register(Pi,Txid)
      ∀i C -> Pi.beginTransaction(Txid)
∀i U -> Pi.credit/debit(xx)   // chaque Pi enregistre l'état de son vote
U -> C.commit(Txid)
    ∀i C -> Pi.vote(Txid)     // chaque Pi retourne l'état de son vote
    ∀i C -> Pi.commit/rollback(Txid)
```

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

Correction proposée par Damien.
Remarque Lionel : la synchronisation peut être plus fine que l'exclusion mutuelle. En particulier on est censé laisser progresser les transactions concurremment tant qu'il n'y a pas

de modification des données. L'exclusion mutuelle est néanmoins pédagogiquement une bonne approximation pour un cours dont l'objet n'est pas la théorie de la sérialisabilité. Une fois cette version assimilée, on peut faire toucher du doigt aux étudiants le fait qu'il faudrait mettre en place une synchronisation plus simple.

```
    public synchronized void beginTransaction( int transactionId )
    throws TException {
        if (currentTransaction == -1) {
            currentTransaction = transactionId;
            futureBalance = balance;
        } else {
            throw new InvalidState();
        }

    }

    private boolean isStateValid( int transactionId ) throws InvalidState {
        return futureBalance >= 0;
    }

    public void debit( int amount ) throws TException {
        futureBalance -= amount;
    }

    @Override
    public void credit( int amount ) throws TException {
        futureBalance += amount;
    }

    private synchronized void ensureCorrectTransaction( int transactionId )
    throws InvalidState {
        if (currentTransaction != transactionId)
            throw new InvalidState();
    }

public synchronized boolean vote( int transactionId )
    throws TException {
        ensureCorrectTransaction(transactionId);
        return isStateValid(transactionId);
    }

    public synchronized void commit( int transactionId )
    throws TException {
        ensureCorrectTransaction(transactionId);
        balance = futureBalance;
        resetTransaction();
    }

    public synchronized void rollback( int transactionId )
    throws TException {
        ensureCorrectTransaction(transactionId);
        resetTransaction();
    }

    private void resetTransaction() {
        currentTransaction = -1;
        futureBalance = 0;
    }
```

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.

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

Préciser que cela concerne la phase de vote, pas l'invocation des méthodes `debit` ni `credit`.

Chaque participant transmet l'état du vote à son successeur en effectuant un ET-logique entre le résultat de son vote et l'état qu'il a reçu de son prédécesseur. Le dernier participant connait ainsi le résultat de la transaction qui peut être engagée ou annulée par chaque participant lorsqu'il reçoit le retour de l'invocation.

7. 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 IDL des nouvelles interfaces que vous proposez.

La méthode `vote` doit pouvoir prendre en paramètre l'état de la transaction.

```
U -> C.beginTransaction()
∀i U -> Pi.reqister(Pi+1,Txid)
U -> C.register(P1,Txid)
      C -> P1.beginTransaction(Txid)
      P1 -> P2.beginTransaction(Txid)
            P2 -> Pn.beginTransaction(Txid)
∀i U -> Pi.credit/debit(xx)   // chaque Pi enregistre l'état de son vote
U -> C.commit(Txid)
    C -> P1.vote(Txid,true)
      P1 -> P2.vote(Txid,bool)      // bool = résultat du vote de P1
        P2 -> Pn.vote(Txid,bool)    // bool = bool ET résultat du vote de P2
          Pn -> this.commit/rollback(Txid)
          // Pn retourne l'état du vote
        P2 -> this.commit/rollback(Txid)
        // P2 retourne l'état du vote
      P1 -> this.commit/rollback(Txid)
      // P1 retourne l'état du vote
```

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.

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

Chaque participant diffuse le résultat de son vote à tous les autres. Quand un participant a reçu le résultat du vote de tous ses pairs, il sait si il peut engager ou annuler la transaction.

9. 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 IDL des nouvelles interfaces que vous proposez.

La méthode `register` doit pouvoir prendre en paramètre une liste de participants et le coordinateur doit pouvoir être considéré comme un participant.

```
U -> C.beginTransaction()
∀i U -> Pi.reqister(<C,P1,P2,…,Pn>-Pi,Txid)
U -> C.register(<P1,P2,…,Pn>,Txid)
      ∀i C -> Pi.beginTransaction(Txid)
∀i U -> Pi.credit/debit(xx)   // chaque Pi enregistre l'état de son vote
∀i Pi -> (<P1,P2,…,Pn>-Pi).vote(Txid)    // chaque Pk retourne à tous les autres son vote
∀i Pi -> this.commit/rollback(Txid)
```

TP 4 – Java EE

Le but de ce TP est de mettre en œuvre une application répartie client/serveur 3 tiers avec Java EE. Ce TP est à réaliser avec le squelette de code car-tp4.zip disponible en téléchargement à partir du module Moodle associé à l'UE. Lire les instructions contenues dans le fichier README.txt de

On souhaite mettre en place une application de gestion d'une bibliothèque de livres.

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. La clé primaire est un identifiant unique auto-généré de type entier.

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.

Gestion du panier dans la session. Le passage de la commande déclenche l'enregistrement dans la table `Commande`. Envisager un EJB *entity* `Commande` « simple » : par exemple, autant d'enregistrements pour une commande qu'il y a de livres commandés (i.e. si je commande trois livres pour la commande 12, cela fait trois enregistrements avec 12 comme numéro de commande).

Gestion d'un *login* avec *password*. Suggérer une `Map` codée en dur pour les utilisateurs. Éventuellement, pour les plus hardis, un EJB *entity*.

TD 11 – Vidéo-club en ligne

1 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`.

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

EJB *entity*, ce sont des données.

1.2 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).

```
@Stateless  // important EJB session stateless
public interface VideoClubFacade {
  void creerAbonne( int num, String nom, String adresse );
  void creerFilm( String titre, String genre, int annee );
  AbonneDAO rechercherAbonne( String nom );
  List<FilmDAO> rechercheFilms( String genre );
  ProjecteurItf visualiser( int num, String titre );
}
```

1.3 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.

```
Utilisateur -> run :: Ecran
                      Ecran -> setEcran(this) ::
                      Ecran -> play           ::
                            :: frame          <- Projecteur
                            :: frame          <- Projecteur
                            :: frame          <- Projecteur
                            :: frame {}       <- Projecteur
```

Fin : invocation de `receive` avec un tableau vide.

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

```
interface ProjecteurItf extends Remote {
  void setEcran( EcranItf ecran ) throws RemoteException;
  void play() throws RemoteException;
}
interface EcranItf extends Remote {
  void frame( byte[] data ) throws RemoteException;
  void run() throws RemoteException;
}
```

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

```
class Projecteur extends UnicastRemoteObject implements ProjecteurItf {
  private byte[] film;
  private EcranItf ecran;
  public Projecteur( byte[] film ) { this.film = film; }
  public void setEcran( EcranItf ecran ) { this.ecran = ecran; }
  pubic void play() {
    for( int i-0 ; i < film.length / 1024 ; i+=1024 ) {
      byte[] buf = System.arraycopy(film,i,buf,0,1024);
      ecran.frame(buf);
    }
    if( film.length % 1024 != 0 ) {  // dernière frame
      byte[] buf =

System.arraycopy(film,(film.length/1024)*1024,buf,0,film.length%1024);
      ecran.frame(buf);
    }
    ecran.frame(new byte[]{});
} }
```

1.6 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.

Méthode `pause` dans l'interface `ProjecteurItf`. Dans la classe `Projecteur`, booléen `paused`. À chaque itération de la méthode `play`, test du booléen `paused`.

1.7 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).

Avantage : performance ; Inconvénients : non garantie de livraison, plus compliqué à programmer.

```
void frame() {
  DatagramSocket s = new DatagramSocket(PORT);
  byte[] buf = new byte[1024];
  DatagramPacket p = new DatagramPacket(buf);
  s.receive(p);
  while( buf.length != 0 ) {
    // affichage des données contenues dans buf
    s.receive(p);
} }
```

2 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.

2.1 Définir une ou plusieurs interfaces 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.

`CanalAboItf` implémentée par les canaux. `ConsoItf` implémentée par les consommateurs.

```
interface CanalAboItf {
  void abonner( in ConsoItf c, in String ressourceId );
  void desabonner( in ConsoItf c, in String ressourceId );
}
```

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

`PushItf` implémentée par les canaux et les consommateurs.

```
interface PushItf {
  void push( in String ressourceId, in String content );
}
```

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

`PullItf` implémentée par les canaux et les producteurs.

```
typedef sequence<string> tstringarray;
interface PullItf {
  tstringarray pull( in String ressourceId );
}
```

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

```
abonner -> PUT
desabonner -> DELETE
pull -> GET
push -> POST
```

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

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

Push-pull: les producteurs font du push, les consommateurs font du pull, le canal sert de buffer.

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.

2.6 Définir une ou plusieurs interfaces 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).

En ce qui concerne le point d'entrée, il faut pouvoir distinguer les appels qui proviennent du canal de ceux qui proviennent d'un pair. Ajout d'un paramètre à la méthode `pull`.

```
interface PullItf {
  tstringarray pull( in String ressourceId, in boolean init );
}

class PullImpl extends PullItfPOA {
  private PullItf vd;
  private isEntryPoint;  // valeur fixée par un constructeur
  public String[] pull( String id, boolean init ) {
    if( isEntryPoint && !init ) { return poll() ? get() : new String[0]; }
    return poll() ?
      get() + vd.pull(id,false) :  // + pour concaténation
      vd.pull(id,false);
  }
  private boolean poll() { return ... }
  private String get() { return ... }
}
```

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.

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

Effectuer une double propagation avec *thread* sur le raccourci et le voisin. Condition d'arrêt : point d'entrée ou nœud raccourci.

TD 12 – *Map Reduce*

Cet exercice vise à étudier la mise en œuvre d'un service dit MapReduce qui permet d'indexer de façon efficace un grand volume de données. Un tel service est utilisé par des librairies comme Yahoo! Hadoop qui sert de base pour des moteurs de recherche et des systèmes de fichiers distribués.

Le but de l'exercice est de définir un service pour pouvoir dénombrer les occurrences des mots contenus dans `n` fichiers texte. On s'intéresse dans un premier temps à une solution centralisée. Soit l'interface `MapReduceItf` qui définit la méthode `count` prenant en paramètre un tableau de `java.io.File` et retournant une `java.util.Hashtable` dont la clé est un mot et la valeur le nombre total d'occurrences de ce mot dans les `n` fichiers.

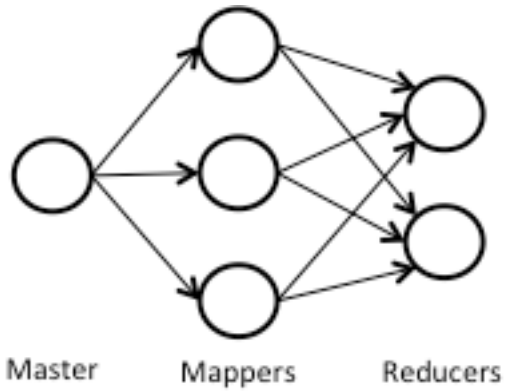
1. Écrire l'interface Java RMI `MapReduceItf`.

```
interface MapReduceItf extends Remote {
  Hashtable<String,int> count( File[] files ) throws RemoteException;
}
```

2. Écrire la classe Java RMI implémentant cette interface. Le corps de la méthode peut être écrit en pseudo-code.

```
class MapReduceImpl extends UnicastRemoteObject implements MapReduceItf {
  public Map<String,int> count( File[] files ) {
    val ht = new Hashtable<String,int>
    foreach( file in files ) {
      // on utilise un StringTokenizer sur le contenu du fichier
      // on itère sur le StringTokenizer
        // pour chaque token présent dans la hashtable +1
        // sinon ht.put(token,1)
    }
    return map
}}
```

On s'intéresse maintenant à une version répartie de la mise en œuvre de ce service. Pour cela on divise la tâche de comptage en deux sous-tâches dites `map` et `reduce`, implémentées respectivement par des objets `Mapper` et `Reducer`. Soit l'architecture illustrée dans la figure suivante et contenant 1 objet `Master`, 3 objets `Mapper` et 2 objets `Reducer`. L'objet `Master` distribue 1 fichier parmi `n` à chaque objet `Mapper`. Comme il y a potentiellement plus de 3 fichiers, chaque objet `Mapper` peut être sollicité plusieurs fois. À chaque sollicitation, l'objet `Mapper` compte les occurrences des mots du fichier qu'il reçoit et transmet pour chaque mot le nombre d'occurrences à un des objets `Reducer`. Le choix de l'objet `Reducer` se fait à l'aide d'une méthode `partition` dont on considère dans un premier temps qu'elle est fournie, et qui, à partir d'un tableau de `Reducer` et d'un mot, retourne la référence du `Reducer` à contacter pour ce mot. Chaque objet `Reducer` additionne les décomptes qu'il reçoit pour chaque mot et stocke le résultat dans une `java.util.Hashtable`.



Le décompte est ainsi distribué parmi les objets `Reducer`. Chaque objet `Reducer` fournit une méthode `getHt` qui retourne sa `java.util.Hashtable`. Plus le nombre d'objets `Mapper` est élevé, plus le volume de données traité efficacement va pouvoir être important.

3. Écrire les interface Java RMI `MapperItf` et `ReducerItf` des objets `Mapper` et `Reducer`.

```
interface MapperItf extends Remote {
  void map( File f ) throws RemoteException;
}

interface ReducerItf extends Remote {
  void reduce( String word, int count ) throws RemoteException;
  Hashtable<String,int> getMap() throws RemoteException;
}
```

4. Écrire les classes Java RMI implémentant ces interfaces. Le corps des méthodes peut être écrit en pseudo-code.

```
class MapperImpl extends UnicastRemoteObject implements MapperItf {
  ReducerItf[] reducers;  // prévoir get/set/ctor
  public void map( File f ) {
    val ht = new Hashtable<String,int>()
    // on utilise un StringTokenizer sur le contenu du fichier
    // on itère sur le StringTokenizer
      // pour chaque token présent dans la hashtable +1
      // sinon ht.put(token,1)
    // on itère sur map avec (key,value)
      ReducerItf reducer = partition(reducers,key)
      reducer.reduce(value)
} }

class ReducerImpl extends UnicastRemoteObject implements ReducerItf {
  ht = new Hashtable<String,int>
  public void reduce( String word, int count ) {
    // si word présent dans ht +count, sinon ht.put(word,count)
  }
  public Hashtable<String,int> getHt = return ht
}
```

5. Proposer en français une implémentation pour la méthode `partition`.

`hashcode` du mot modulo le nombre de `Reducer`.