

**Cahier de Travaux Dirigés et Pratiques**  
**Construction d'Applications Réparties (CAR)**

**Master Sciences et Technologies**

**Mention Informatique – M1**

**2017**

**Équipe enseignante**

Clément Ballabriga  
Maxime Colmant  
Laurence Duchien  
Giuseppe Lipari  
Romain Rouvoy  
Lionel Seinturier

## Calendrier

Séances	Cours	Sujets TD	Sujets TP
Semaine 1	Introduction aux appli- cations 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	<b>Évaluation TP 1</b> 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	<b>Évaluation TP 2</b> 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	<b>Évaluation TP 3</b> TP 4 : Java EE
Semaine 12	Java EE – EJB	TD 11 : Vidéo-club en ligne	TP 4 : Java EE
Semaine 13		TD 12 : <i>Map Reduce</i>	TP 4 : Java EE <b>Évaluation TP 4</b>

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

## TD 1 – Programmation client/serveur en mode message

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

### 1. Définitions

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

### 2. Serveur calculette

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

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

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

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

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

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

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

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

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

## TD 2/TP 1 – Serveur FTP

### 1. API *socket* Java

---

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

1. Quelle méthode Java permet d'attendre l'arrivée de demandes de connexions sur un port TCP ?
2. Comment lire des données sur une *socket* en Java ?
3. Comment écrire des données sur une *socket* en Java ?

### 2. Le protocole FTP

---

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

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

Les commandes pour le client sont les suivantes :

`USER username` : utilisé pour l'authentification de l'utilisateur

`PASS password` : utilisé pour le mot de passe de l'utilisateur

`LIST` : permet à l'utilisateur de demander l'envoi de la liste des fichiers du répertoire courant

`RETR filename` : utilisé pour prendre un fichier du répertoire distant et le déposer dans le répertoire local

`STOR filename` : utilisé pour déposer un fichier venant du répertoire local dans le répertoire distant

`QUIT` : permet à l'utilisateur de terminer la session FTP en cours

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

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

1. Écrire en français ou à l'aide d'un *framework* (par exemple JUnit) les scénarii 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

**Client.** 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.
  - déléguant 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 ..`

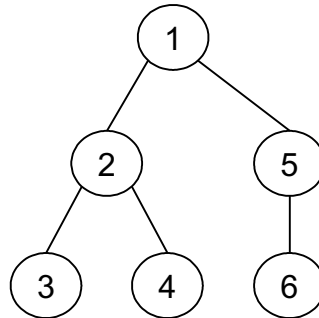
Pour plus d'informations :

1. <http://www.faqs.org/rfcs/rfc959.html>
2. <http://www.w3.org/Protocols/rfc959>
3. [https://fr.wikipedia.org/wiki/File\\_Transfer\\_Protocol](https://fr.wikipedia.org/wiki/File_Transfer_Protocol)

## TD 3 – Architectures d'applications Internet

### 1. Diffusion de messages

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



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

On considère que les sites communiquent en UDP sur le port 5000 en envoyant des messages de 2048 octets (contenu quelconque), et que chaque site a :

- une variable `pere` qui contient l'adresse de son père dans l'arbre (`null` si pas de père),
- un tableau `fils` qui contient les adresses de ses fils (tableau vide si pas de fils).

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

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

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

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

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

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

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

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

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

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

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

## Étude des messages échangés par le protocole client/serveur

On considère les **primitives** suivantes :

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

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

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

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



### **Implantation des services**

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

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

Dans un but de simplification, on suppose que :

- en cas de réception d'un message non attendu, les procédures sont interrompues et une erreur signalée à l'utilisateur,
  - on ne s'intéresse pas à la façon dont le client écrit les données du fichier qu'il reçoit.
7. En utilisant les primitives de la question précédente, écrire le pseudo-code ou le code Java des primitives client(fic,i,j) et serveur().



## TD 4 – Représentation des données

### 1. Petit boutiste et grand boutiste

Le but de cet exercice est de mettre en lumière les différences de stockage des données entre les formats petit boutiste (*little endian*) et grand boutiste (*big endian*). Ce problème acquiert son importance dès lors que l'on cherche à transférer de l'information entre des clients et des serveurs qui ont fait des choix de représentation de données différents : il faut que le protocole s'assure de la bonne conversion des données.

On considère un flux d'entrée/sortie en mode binaire (*i.e.*, constitué d'octets) dans lequel on veut envoyer des mots de 16 bits. Chaque mot se décompose en octet de poids fort et octet de poids faible (la valeur du mot est  $256 \times \text{octet poids fort} + \text{octet poids faible}$ ).

- le format petit boutiste consiste à écrire d'abord l'octet de poids faible puis l'octet de poids fort,
- le format grand boutiste consiste à écrire d'abord l'octet de poids fort puis l'octet de poids faible.

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

1. On considère la chaîne de caractères « hello world » codée de la façon suivante :

- 2 octets pour sa longueur,
- 1 octet par caractère.

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

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

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

### 2. Mécanisme d'encodage des données en Java

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

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

Les chaînes de caractères sont représentées selon la norme UTF. De façon simplifiée, pour des chaînes ne comportant que des caractères ASCII non accentués, le codage est le suivant :

- 2 octets pour la longueur
- 1 octet par caractère

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

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

Quelle est la valeur de ces entiers ?

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

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

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

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

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

Indications :

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

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

6. Décoder les données sérialisées suivantes :

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

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

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

## Annexe : Java Object Serialization Stream Protocol

### Les éléments du flot d'octets

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

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

Un objet `ObjectStreamClass` est représenté par :

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

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

Les tableaux sont représentés par :

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

Les nouveaux objets dans le flot sont représentés par :

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

### La grammaire du format flot d'octet

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

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

```

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

```

```

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

```

```

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

```

## Les symboles et les constantes terminaux

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

```

final static short STREAM_MAGIC = (short)0xaced;
final static short STREAM_VERSION = 5;
final static byte TC_NULL = (byte)0x70;
final static byte TC_REFERENCE = (byte)0x71;
final static byte TC_CLASSDESC = (byte)0x72;
final static byte TC_OBJECT = (byte)0x73;
final static byte TC_STRING = (byte)0x74;
final static byte TC_ARRAY = (byte)0x75;
final static byte TC_CLASS = (byte)0x76;
final static byte TC_BLOCKDATA = (byte)0x77;
final static byte TC_ENDBLOCKDATA = (byte)0x78;
final static byte TC_RESET = (byte)0x79;
final static byte TC_BLOCKDATAALONG = (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 caractères 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 ?

réponse : ié 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.

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

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

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`.

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».



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).

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.

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

### Implantation

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

## TP 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 déboguer votre passerelle REST, vous pouvez utiliser un client HTTP en ligne de commande, par exemple `curl`. Pour tester votre passerelle REST, vous utiliserez JUnit et éventuellement un *mock* du serveur FTP.

1. Écrire en français ou à l'aide d'un *framework* (par exemple JUnit) les scénarii 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.
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.
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.

## ***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\\_Authentication](http://fr.wikipedia.org/wiki/HTTP_Authentication)

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

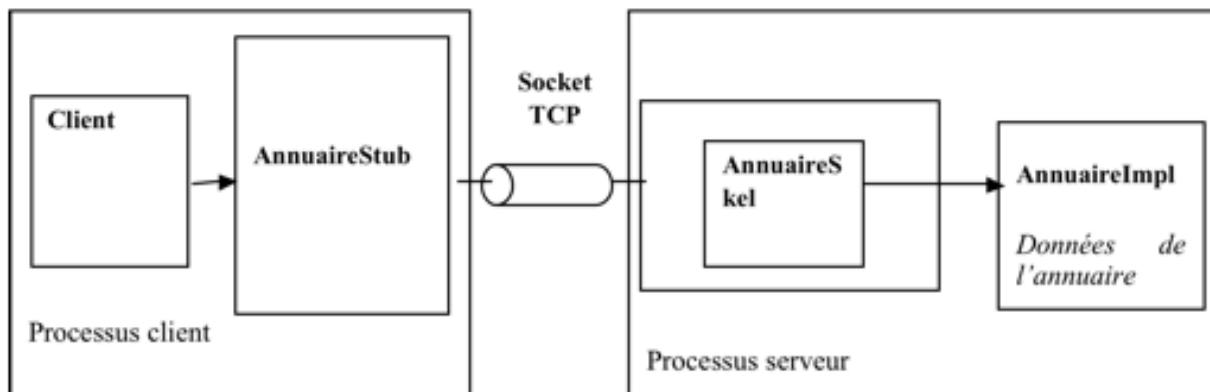


Figure 1 : Relation entre les objets de l'application répartie

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(unBooleen);
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.

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

On considère un client qui effectue la séquence d'appels suivante sur un annuaire hébergé sur la machine `annuaire.fil.fr`, port 89 :

- ajouter adresse logique « car », adresse physique `car.fil.fr` port 5896,
- ajouter adresse logique « m1 », adresse physique `master.fil.fr` port 698,
- lister.

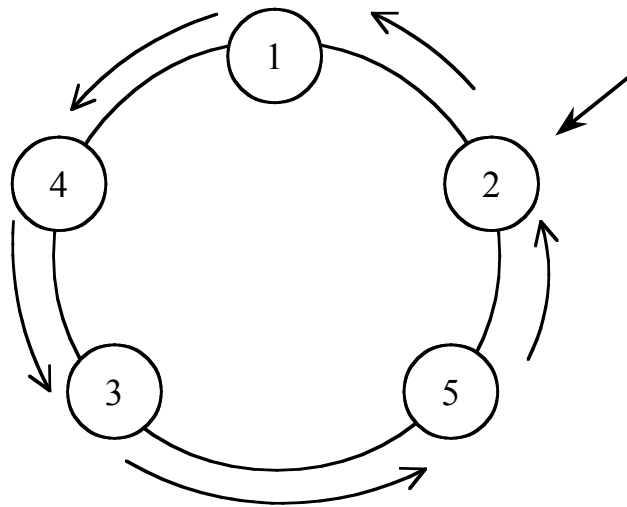
4. Écrire le code des classes `Client` et `AnnuaireStub` correspondant au comportement précédent. La classe `AnnuaireStub` doit implémenter l'interface `Annuaire`.

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

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

## 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.



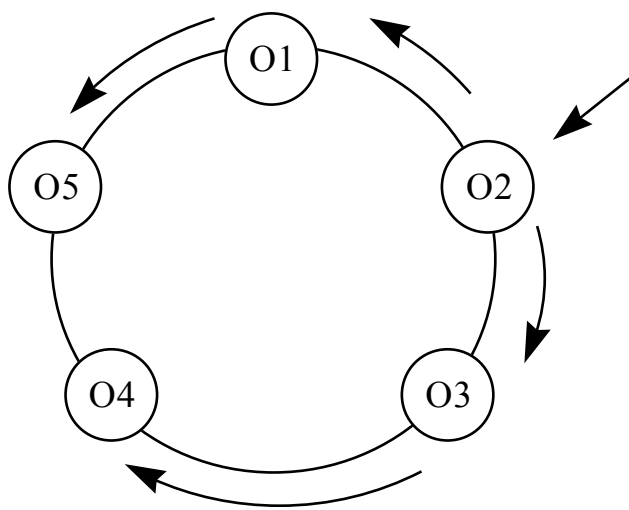
1. Quel est le test d'arrêt de l'élection i.e. du « tour de l'anneau » ? Quelle information faut-il inclure dans le message qui fait le « tour de l'anneau » pour pouvoir réaliser ce test ?
2. Représenter sur un diagramme les échanges de messages engendrés par l'exécution d'une élection avec l'objet 2 comme initiateur.
3. Quel est le test de décision de l' élu ? Quelle information faut-il inclure dans le message qui fait le « tour de l'anneau » pour pouvoir réaliser ce test ?
4. Reprendre le schéma de l'anneau avec cinq objets donné ci-dessus et indiquer sur chaque message la valeur de ces deux informations.
5. À la fin d'un tour, quel/s objet/s connaît/connaissent l'identifiant de l' élu ? Pourquoi ?
6. Proposer deux solutions, une à base de messages synchrones, une à base de messages asynchrones, pour que tous les objets de l'anneau connaissent l'identifiant de l' élu.
7. Sur chaque objet, le « tour de l'anneau » correspond à une méthode `election` prenant en entrée les informations définies aux questions 1 et 3. Pour chacune des solutions proposées à la question précédente, définir l'interface de la classe Java correspondant à un objet de l'anneau.

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



## 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 ?

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.

### 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 ?

### Interfaces

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

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

### Implantation d'un objet de l'anneau

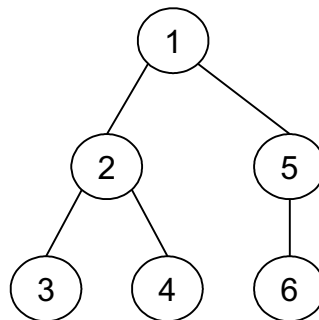
7. En supposant, qu'il n'y a qu'un **seul objet initiateur simultanément**, donner en Java, le code de la classe qui implante le comportement d'un objet de l'anneau.

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

## 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 les bibliothèques contenues dans l'archive 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 scénarii 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).
4. On suppose maintenant que les acteurs sont répartis dans deux systèmes d'acteurs différents. 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

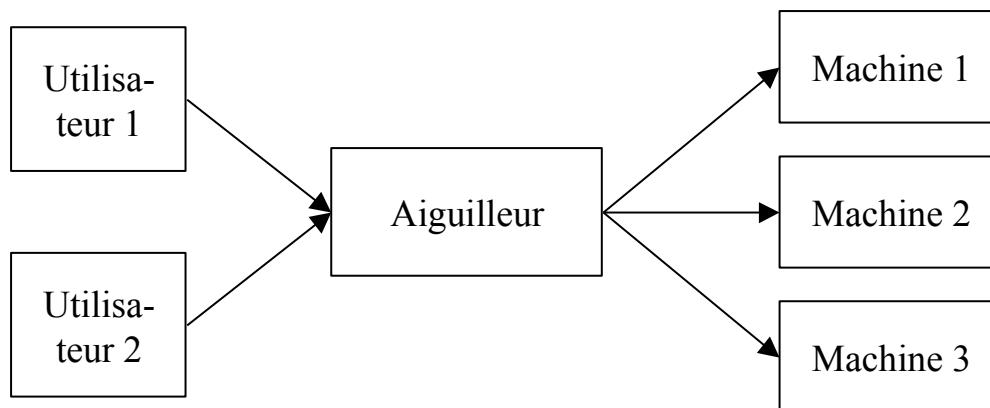
Cet exercice utilise le langage de définition d'interface Apache Thrift dont une description est fournie en annexe.

### 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).

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 `écriture` 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.

### **Aiguilleur**

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

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

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

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

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

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

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

### **Concurrence des traitements**

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

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

9. Faut-il ajouter des définitions aux interfaces précédentes pour prendre en compte ce cas ? Si oui, laquelle/lesquelles ? Si non, pourquoi ?

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

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

### **Performances**

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

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

Pour palier au problème du goulet d'étranglement, on propose en plus de la solution précédente, de mettre en place plusieurs aiguilleurs dont la répartition des rôles sera prédéfinie (codée "en dur" chez les utilisateurs).

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

## **Annexe : Apache Thrift**

---

## Apache Thrift™

- [Download](#)
- [Documentation](#)
- [Developers](#)
- [Tutorials](#)
- [About](#)
- [Apache](#)
  - [Apache Home](#)
  - [Apache License v2.0](#)
  - [Donate](#)
  - [Thanks](#)
  - [Security](#)

## Apache Thrift Features

- interface description language - Everything is specified in an IDL file from which bindings for many languages can be generated. See [Thrift IDL](#).
- language bindings - Thrift is supported in many languages and environments
  - C++
  - C#
  - Cocoa
  - D
  - Delphi
  - Erlang
  - Haskell
  - Java
  - OCaml
  - Perl
  - PHP
  - Python
  - Ruby
  - Smalltalk
- namespaces - Each Thrift file is in its own namespace allowing you to use the same identifier in multiple Thrift files
- language namespaces - Per Thrift file you can specify which namespace should be used for each programming language
- base types - Thrift has a small set of base types. See [Thrift Types](#)
- constants and enumerations - Constant values can be assigned logical names
- structs - Use structs to group related data. Structs can have fields of any type. See [Thrift Types](#)
- sparse structs - Optional base fields that have not been set and reference fields that are null will not be sent across the wire
- struct evolution - The addition and removal of fields is handled without breaking existing clients by using integer identifiers for fields
- containers - You can use sets, lists and maps of any type: base types, structs and other containers. See [Thrift Types](#)
- type definitions - Any type can be given a name that better describes it
- services - A service is a group of functions
- service inheritance - Subservices implement all functions of their base services and can have additional functions
- asynchronous invocations - Functions that do not return a result can be invoked asynchronously so the client is not blocked until the server has finished processing the request. The server may execute asynchronous invocations of the same client in parallel/out of order

- exceptions - If an error occurs a function can throw a standard or user-defined exception. See [Thrift Types](#)

## Non-features

The following are not supported by Apache Thrift:

- cyclic structs - Structs can only contain structs that have been declared before it. A struct also cannot contain itself
- struct inheritance - Use struct composition instead
- polymorphism - As there is no inheritance, polymorphism is also not supported
- overloading - All methods within a service must be uniquely named
- heterogeneous containers - All items in a container must be of the same type
- Null return - null cannot be returned directly from a function. Use a wrapper struct or a marker value instead

## Links

- [Download](#)
  - [Developers](#)
  - [Tutorials](#)
  - [Sitemap](#)
- ## Get Involved
- [Mailing Lists](#)
  - [Issue Tracking](#)
  - [How To Contribute](#)



Copyright 2012 Apache Software Foundation. Licensed under the [Apache License v2.0](#). Apache, Apache Thrift, and the Apache feather logo are trademarks of The Apache Software Foundation.

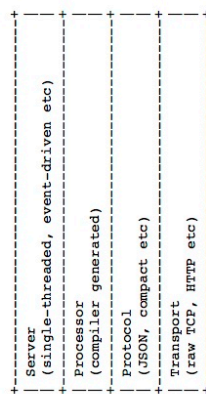


Apache Thrift™

- [Download](#)
- [Documentation](#)
- [Developers](#)
- [Tutorials](#)
- [About](#)
- [Apache](#)
  - [Apache Home](#)
  - [Apache License v2.0](#)
  - [Donate](#)
  - [Thanks](#)
  - [Security](#)

## Thrift network stack

Simple representation of the Apache Thrift networking stack



## Transport

The Transport layer provides a simple abstraction for reading/writing from/to the network. This enables Thrift to decouple the underlying transport from the rest of the system (serialization/deserialization, for instance).

Here are some of the methods exposed by the **Transport** interface:

- open
- close
- read
- write
- flush

In addition to the **Transport** interface above, Thrift also uses a **ServerTransport** interface used to accept or create primitive transport objects. As the name suggest, **ServerTransport** is used mainly on the server side to create new Transport objects for incoming connections.

- open
- listen
- accept
- close

Here are some of the transports available for majority of the Thrift-supported languages:

- file: read/write to/from a file on disk
- http: as the name suggests

## Protocol

The Protocol abstraction defines a mechanism to map in-memory data structures to a wire-format. In other words, a protocol specifies how datatypes use the underlying Transport to encode/decode themselves. Thus the protocol implementation governs the encoding scheme and is responsible for (de)serialization. Some examples of protocols in this sense include JSON, XML, plain text, compact binary etc.

Here is the **Protocol** interface:

```

writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeFieldStop()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
writeListBegin(etype, size)
writeListEnd()
writeSetBegin(etype, size)
writeSetEnd()
writeBool(bool)
writeByte(byte)
writeI16(i16)
writeI32(i32)
writeI64(i64)
writeDouble(double)
writeString(string)

name, type, seq = readMessageBegin()
readMessageEnd()
name = readStructBegin()
readStructEnd()
name, type, id = readFieldBegin()
readFieldEnd()
k, v, size = readMapBegin()
readMapEnd()
etype, size = readListBegin()
readListEnd()
etype, size = readSetBegin()
readSetEnd()
bool = readBool()
byte = readByte()
i16 = readI16()
i32 = readI32()
i64 = readI64()
double = readDouble()
string = readString()

```

Thrift Protocols are stream oriented by design. There is no need for any explicit framing. For instance, it is not necessary to know the length of a string or the number of items in a list before we start serializing them. Some of the protocols available for majority of the Thrift-supported languages are:

- binary: Fairly simple binary encoding – the length and type of a field are encoded as bytes

followed by the actual value of the field.

- compact: Described in [THRIFT-110](#)
- json

## Processor

A Processor encapsulates the ability to read data from input streams and write to output streams. The input and output streams are represented by Protocol objects. The Processor interface is extremely simple

```
interface TProcessor {
    bool process(TProtocol in, TProtocol out) throws TException
}
```

Service-specific processor implementations are generated by the compiler. The Processor essentially reads data from the wire (using the input protocol), delegates processing to the handler (implemented by the user) and writes the response over the wire (using the output protocol).

## Server

A Server pulls together all of the various features described above:

- Create a transport
- Create input/output protocols for the transport
- Create a processor based on the input/output protocols
- Wait for incoming connections and hand them off to the processor

---

## Links

- [Download](#)
- [Developers](#)
- [Tutorials](#)
- [Sitemap](#)

## Get Involved

- [Mailing Lists](#)
- [Issue Tracking](#)
- [How To Contribute](#)



Copyright 2012 [Apache Software Foundation](#). Licensed under the [Apache License v2.0](#). Apache, Apache Thrift, and the Apache feather logo are trademarks of The Apache Software Foundation.

Apache Thrift™

- [Download](#)
- [Documentation](#)
- [Developers](#)
- [Tutorials](#)
- [About](#)
- [Apache](#)
  - [Apache Home](#)
  - [Apache License 2.0](#)
  - [Donate](#)
  - [Thanks](#)
  - [Security](#)

Thrift interface description language

The Thrift interface definition language (IDL) allows for the definition of Thrift Types. A Thrift IDL file is processed by the Thrift code generator to produce code for the various target languages to support the defined structs and services in the IDL file.

Description

Under construction

Here is a description of the Thrift IDL.

Document

Every Thrift document contains 0 or more headers followed by 0 or more definitions.

(1) Document ::= Header\* Definition\*

Header

A header is either a Thrift include, a C++ include, or a namespace declaration.

(2) Header ::= Include | CppInclude | Namespace

Thrift Include

An include makes all the symbols from another file visible (with a prefix) and adds corresponding include statements into the code generated for this Thrift document.

(3) Include ::= 'include' Literal

C++ Include

A C++ include adds a custom C++ include to the output of the C++ code generator for this Thrift document.

(4) CppInclude ::= 'cpp\_include' Literal

Namespace

A namespace declares which namespaces/package/module/etc. the type definitions in this file will be declared in for the target languages. The namespace scope indicates which language the namespace applies to; a scope of '\*' indicates that the namespace applies to all target languages.

(5) Namespace ::= ( 'namespace' ( NamespaceScope Identifier ) | Identifier ) { 'smalltalk.prefix' Identifier } | { 'php\_namespace' Literal } | { 'xsd\_namespace' Literal }

(6) NamespaceScope ::= '\*' | 'cpp' | 'java' | 'py' | 'perl' | 'rb' | 'cocoon' | 'esharp'

N.B.: Smalltalk has two distinct types of namespace commands. *Can someone who knows Smalltalk verify two different kinds of namespaces?*

N.B.: The php\_namespace directive will be deprecated at some point in the future in favor of the scoped syntax, but the scoped syntax is not yet supported for PHP.

N.B.: The xsd\_namespace directive has some purpose internal to Facebook but serves no purpose in Thrift itself. Use of this feature is strongly discouraged

Definition

(7) Definition ::= Const | Typedef | Enum | Struct | Union | Exception | Service

Const

(8) Const ::= 'const' FieldType Identifier '=' ConstValue ListSeparator?

Typedef

A typedef creates an alternate name for a type.

(9) Typedef ::= 'typedef' DefinitionType Identifier

Enum

An enum creates an enumerated type, with named values. If no constant value is supplied, the value is either 0 for the first element, or one greater than the preceding value for any subsequent element. Any constant value that is supplied must be non-negative.

(10) Enum ::= 'enum' Identifier '{' ( Identifier ('=' IntConstant)? ListSeparator? )\* '}'

Seenum

Seenum (and Slist) are now deprecated and should both be replaced with String.

(11) Seenum ::= 'seenum' Identifier '{' ( Literal ListSeparator? )\* '}'

Struct

Structs are the fundamental compositional type in Thrift. The name of each field must be unique within the struct.

(12) Struct ::= 'struct' Identifier 'xsd\_all?' '{' Field\* '}'

N.B.: The xsd\_all keyword has some purpose internal to Facebook but serves no purpose in Thrift itself. Use of this feature is strongly discouraged

Union

Unions are similar to structs, except that they provide a means to transport exactly one field of a possible set of fields, just like union {} in C++. Consequently, union members cannot be required fields.

(13) Union ::= 'union' Identifier 'xsd\_all?' '{' Field\* '}'

N.B.: The xsd\_all keyword has some purpose internal to Facebook but serves no purpose in Thrift itself. Use of this feature is strongly discouraged

Exception

Exceptions are similar to structs except that they are intended to integrate with the native exception handling mechanisms in the target languages. The name of each field must be unique within the exception.

(14) Exception ::= 'exception' Identifier '{' Field\* '}'

Service

A service provides the interface for a set of functionality provided by a Thrift server. The interface is simply a list of functions. A service can extend another service, which simply means that it provides the functions of the extended service in addition to its own.

(15) Service ::= 'service' Identifier ( 'extends' Identifier )? '{' Function\* '}'

Field

(16) Field ::= FieldID? FieldReq? FieldType Identifier ('=' ConstValue)? RedefinitionOptions ListSeparator?

Field ID

(17) FieldID ::= IntConstant | '

Field Requiredness

(18) FieldReq ::= 'required' | 'optional'

XSD Options

N.B.: These have some internal purpose at Facebook but serve no current purpose in Thrift. Use of these options is strongly discouraged.

```
[19] xsd_optional? 'xsd_optional'? 'xsd_nullable'? xsd_attr?
[20] xsd_attr := 'xsd_attr' '{ ' field' '}'
```

Functions

```
[21] function := 'neway'? FunctionType Identifier '{ ' field* '}' Throws? ListSeparator?
[22] FunctionType := 'FieldType' | 'void'
[23] throws := 'throws' '{ ' field* '}'
```

Types

```
[24] FieldType := Identifier | BaseType | ContainerType
[25] DefinitionType := BaseType | ContainerType
[26] BaseType := 'bool' | 'byte' | 'i16' | 'i32' | 'i64' | 'double' | 'string' | 'binary' | 'uuid'
[27] ContainerType := MapType | SetType | ListType
[28] MapType := 'map' CppType? '<' FieldType ' ' FieldType '>'
[29] SetType := 'set' CppType? '<' FieldType '>'
[30] ListType := 'list' '<' FieldType '>' CppType?
[31] CppType := 'cpp_type' Literal
```

Constant Values

```
[32] ConstValue := IntConstant | DoubleConstant | Literal | Identifier | ConstList | ConstMap
[33] IntConstant := ('+' | '-' | '0')? Digit+
[34] DoubleConstant := ('+' | '-' | '0')? Digit* { '.' Digit+ }? ( 'E' | 'e' ) IntConstant ?
[35] ConstList := '{ ' (ConstValue ListSeparator)* '}'
[36] ConstMap := '{ ' (ConstValue ':' ConstValue ListSeparator)* '}'
```

Basic Definitions

Literal

```
[37] Literal := (''' {''}* ''') | ('"" {""}* "")
```

Identifier

```
[38] Identifier := ( Letter | '_' ) ( Letter | Digit | '.' | '_' ) *
[39] STIdentifier := ( Letter | '_' ) ( Letter | Digit | '.' | '_' | '_' ) *
```

List Separator

```
[40] ListSeparator := ',' | ';' | '\n'
```

Letters and Digits

```
[41] Letter := [ 'A'-'Z' ] | [ 'a'-'z' ]
[42] Digit := [ '0'-'9' ]
```

Examples

Here are some examples of Thrift definitions, using the Thrift IDL:

- [ThriftTest.thrift](#) used by the Thrift TestFramework
- [Thrift tutorial](#)
- Facebook's [fb.903.thrift](#)

- [Apache Cassandra's Thrift IDL: cassandra.thrift](#)
- [Extreme API](#)

To Do/Questions

Initialization of Base Types for all Languages?

- Do all Languages initialize them to 0, bool=false and string="" or null, undefined?
- Why does position of CppType vary between SetType and ListType?
- std::set does sort the elements automatically, that's the design, see [Thrift Types](#) or the [C++ std::set reference](#) for further details
- The question is, how other languages are doing that? What about custom objects, do they have a Compare function the set the order correctly?

Why can't the init function be the same as FieldType (i.e. include Identifier)?

Examine the smalltalk, prefix and smalltalk\_category status (esp smalltalk\_category, which takes str::identifier as its argument) ...

What to do about ListSeparator? Do we really want to be as lax as we currently are?


Should Field\* really be Field\* in struct, Enum, etc.?

Links

- [Download](#)
- [Developers](#)
- [Tutorials](#)
- [Slidemg](#)

Get Involved

- [Mailing Lists](#)
- [Issue Tracking](#)
- [How To Contribute](#)

 Copyright 2012 Apache Software Foundation. Licensed under the [Apache License, v2.0](#). Apache, Apache Thrift, and the Apache feather logo are trademarks of The Apache Software Foundation.



## Apache Thrift™

- [Download](#)
- [Documentation](#)
- [Developers](#)
- [Tutorials](#)
- [About](#)
- [Apache](#)
  - [Apache Home](#)
  - [Apache License v2.0](#)
  - [Donate](#)
  - [Thanks](#)
  - [Security](#)

## Thrift Types

The Thrift type system is intended to allow programmers to use native types as much as possible, no matter what programming language they are working in. This information is based on, and supersedes, the information in the [Thrift Whitepaper](#). The [Thrift IDL](#) provides descriptions of the types which are used to generate code for each target language.

### Base Types

The base types were selected with the goal of simplicity and clarity rather than abundance, focusing on the key types available in all programming languages.

- bool: A boolean value (true or false)
- byte: An 8-bit signed integer
- i16: A 16-bit signed integer
- i32: A 32-bit signed integer
- i64: A 64-bit signed integer
- double: A 64-bit floating point number
- string: A text string encoded using UTF-8 encoding

Note the absence of unsigned integer types. This is due to the fact that there are no native unsigned integer types in many programming languages.

### Special Types

binary: a sequence of unencoded bytes

N.B.: This is currently a specialized form of the string type above, added to provide better interoperability with Java. The current plan-of-record is to elevate this to a base type at some point.

### Structs

Thrift structs define a common object – they are essentially equivalent to classes in OOP languages, but without inheritance. A struct has a set of strongly typed fields, each with a unique name identifier. Fields may have various annotations (numeric field IDs, optional default values, etc.) that are described in the [Thrift IDL](#).

### Containers

Thrift containers are strongly typed containers that map to commonly used and commonly available container types in most programming languages.

There are three container types:

list: An ordered list of elements. Translates to an STL vector, Java ArrayList, native arrays in scripting languages, etc. set: An unordered set of unique elements. Translates to an STL set, Java HashSet, set in Python, etc. Note: PHP does not support sets, so it is treated similar to a List map: A map of strictly unique keys to values. Translates to an STL map, Java HashMap, PHP associative array, Python/Ruby dictionary, etc. While defaults are provided, the type mappings are not explicitly fixed. Custom code generator directives have been added to allow substitution of custom types in various destination languages.

Container elements may be of any valid Thrift Type.

N.B.: For maximal compatibility, the key type for map should be a basic type rather than a struct or container type. There are some languages which do not support more complex key types in their native map types. In addition the JSON protocol only supports key types that are base types.

### Exceptions

Exceptions are functionally equivalent to structs, except that they inherit from the native exception base class as appropriate in each target programming language, in order to seamlessly integrate with the native exception handling in any given language.

### Services

Services are defined using Thrift types. Definition of a service is semantically equivalent to defining an interface (or a pure virtual abstract class) in object oriented programming. The Thrift compiler generates fully functional client and server stubs that implement the interface.

A service consists of a set of named functions, each with a list of parameters and a return type.

Note that void is a valid type for a function return, in addition to all other defined Thrift types. Additionally, an async modifier keyword may be added to a void function, which will generate code that does not wait for a response. Note that a pure void function will return a response to the client which guarantees that the operation has completed on the server side. With async method calls the client will only be guaranteed that the request succeeded at the transport layer. Async method calls of the same client may be executed in parallel/out of order by the server.

### Links

- [Download](#)
  - [Developers](#)
  - [Tutorials](#)
  - [Sitemap](#)
- ### Get Involved
- [Mailing Lists](#)
  - [Issue Tracking](#)
  - [How To Contribute](#)

```

namespace cpp tutorial
namespace d tutorial
namespace java tutorial
namespace php tutorial
namespace perl tutorial
/**
 * Thrift lets you do typedefs to get pretty names for your types. Standard
 * C style here.
 */
typedef i32 MyInteger

/**
 * Thrift also lets you define constants for use across languages. Complex
 * types and structs are specified using JSON notation.
 */
const i32 INT32CONSTANT = 9853
const map<string,string> MAPCONSTANT = {'hello':'world', 'goodnight':'moon'}

/**
 * You can define enums, which are just 32 bit integers. Values are optional
 * and start at 1 if not supplied, C style again.
 */
enum Operation {
    ADD = 1,
    SUBTRACT = 2,
    MULTIPLY = 3,
    DIVIDE = 4
}

/**
 * Structs are the basic complex data structures. They are comprised of fields
 * which each have an integer identifier, a type, a symbolic name, and an
 * optional default value.
 *
 * Fields can be declared "optional", which ensures they will not be included
 * in the serialized output if they aren't set. Note that this requires some
 * manual management in some languages.
 */
struct Work {
    1: i32 num1 = 0,
    2: i32 num2,
    3: Operation op,
    4: optional string comment,
}

/**
 * Structs can also be exceptions, if they are nasty.
 */
exception InvalidOperation {
    1: i32 what,
    2: string why
}

/**
 * Ahh, now onto the cool part, defining a service. Services just need a name
 * and can optionally inherit from another service using the extends keyword.
 */
service Calculator extends shared.SharedService {
}

/**
 * A method definition looks like C code. It has a return type, arguments,
 * and optionally a list of exceptions that it may throw. Note that argument
 * lists and exception lists are specified using the exact same syntax as
 * field lists in struct or exception definitions.

```

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the license at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software distributed under the License is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
 * KIND, either express or implied. See the License for the
 * specific language governing permissions and limitations
 * under the License.
 */

# Thrift Tutorial
# Mark Slee (mcslee@facebook.com)

# This file aims to teach you how to use Thrift, in a .thrift file. Neato. The
# first thing to notice is that .thrift files support standard shell comments.
# This lets you make your thrift file executable and include your Thrift build
# step on the top line. And you can place comments like this anywhere you like.

# Before running this file, you will need to have installed the thrift compiler
# into /usr/local/bin.

/**
 * The first thing to know about are types. The available types in Thrift are:
 */
* bool Boolean, one byte
* byte Signed byte
* i16 Signed 16-bit integer
* i32 Signed 32-bit integer
* i64 Signed 64-bit integer
* double 64-bit floating point value
* string String
* binary Blob (byte array)
* map<tl,t2> Map from one type to another
* list<tl> Ordered list of one type
* set<tl> Set of unique elements of one type
*
* Did you also notice that Thrift supports C style comments?
*/

// Just in case you were wondering... yes. We support simple C comments too.

/**
 * Thrift files can reference other Thrift files to include common struct
 * and service definitions. These are found using the current path, or by
 * searching relative to any paths specified with the -I compiler flag.
 *
 * Included objects are accessed using the name of the .thrift file as a
 * prefix. i.e. shared.SharedObject
 */
include "shared.thrift"

/**
 * Thrift files can namespace, package, or prefix their output in various
 * target languages.
 */

```

```

*/
void ping() {
    i32 add(1:i32 num1, 2:i32 num2),
    i32 calculate(1:i32 logid, 2:Work w) throws (1:InvalidOperation ouch),
}
/**
 * This method has a oneway modifier. That means the client only makes
 * a request and does not listen for any response at all. Oneway methods
 * must be void.
 */
oneway void zip()
}
/**
 * That just about covers the basics. Take a look in the test/ folder for more
 * detailed examples. After you run this file, your generated code shows up
 * in folders with names gen-<language>. The generated code isn't too scary
 * to look at. It even has pretty indentation.
 */

```

## Apache Thrift™

- [Download](#)
- [Documentation](#)
- [Developers](#)
- [Tutorials](#)
- [About](#)
- [Apache](#)
  - [Apache Home](#)
  - [Apache License v2.0](#)
  - [Donate](#)
  - [Thanks](#)
  - [Security](#)

## Java Tutorial

### Introduction

All Apache Thrift tutorials require that you have:

1. Built and installed the Apache Thrift Compiler, see [installing Thrift](#) for more details.
2. Generated the `tutorial.thrift` file as [discussed here](#)

3. Followed all prerequisites listed

### Prerequisites

The Java tutorial includes an ant file; running `ant build` will bring in the libraries required for running the tutorial.

### Client

```
Transport transport = new TSocket("localhost", 9090);
transport.open();

Protocol protocol = new TBinaryProtocol(transport);
Calculator.Client client = new Calculator.Client(protocol);

client.ping();
System.out.println("ping()");

int sum = client.add(1,1);
System.out.println("1+1=" + sum);

Work work = new Work();

work.op = Operation.DIVIDE;
work.num1 = 1;
work.num2 = 0;
try {
    int quotient = client.calculate(1, work);
    System.out.println("Whoa we can divide by 0");
} catch (InvalidOperation io) {
    System.out.println("Invalid operation: " + io.why);
}

work.op = Operation.SUBTRACT;
work.num1 = 1;
work.num2 = 10;
try {
    int diff = client.calculate(1, work);
    System.out.println("15-10=" + diff);
} catch (InvalidOperation io) {
    System.out.println("Invalid operation: " + io.why);
}

SharedStruct log = client.getStruct();
System.out.println("Check log: " + log.value);
transport.close();
```

## Server

```
CalculatorHandler handler = new CalculatorHandler();
Calculator.Processor processor = new Calculator.Processor(handler);
TServerTransport serverTransport = new TServerSocket(9090);
TServer server = new TThreadPoolServer(new Args(serverTransport).processor(processor));
System.out.println("Starting the simple server...");
server.serve();
```

## Server Handler

```
public class CalculatorHandler implements Calculator.Iface {
    private HashMap log;

    public CalculatorHandler() {
        log = new HashMap();
    }

    public void ping() {
        System.out.println("ping()");
    }

    public int add(int n1, int n2) {
        System.out.println("add(" + n1 + ", " + n2 + ")");
        return n1 + n2;
    }

    public int calculate(int logId, Work work) throws InvalidOperation {
        System.out.println("calculate(" + logId + ", " + work.op + ", " + work.num1 + ", " + work.num2 + ")");
        int val = 0;
        switch (work.op) {
            case ADD:
                val = work.num1 + work.num2;
                break;
            case SUBTRACT:
                val = work.num1 - work.num2;
                break;
            case MULTIPLY:
                val = work.num1 * work.num2;
                break;
            case DIVIDE:
                if (work.num2 == 0) {
                    InvalidOperation io = new InvalidOperation();
                    io.what = work.op.getValue();
                    io.why = "Cannot divide by 0";
                    throw io;
                }
                val = work.num1 / work.num2;
                break;
            default:
                InvalidOperation io = new InvalidOperation();
                io.what = work.op.getValue();
                io.why = "Unknown operation";
                throw io;
        }
        SharedStruct entry = new SharedStruct();
        entry.key = logId;
        entry.value = Integer.toString(val);
        log.put(logId, entry);
        return val;
    }

    public SharedStruct getStruct(int key) {
        SharedStruct entry = (SharedStruct) log.get(key);
        return log.get(key);
    }

    public void zip() {
        System.out.println("zip()");
    }
}
```

## Additional Information



## TD 10 – Protocole de validation à deux phases

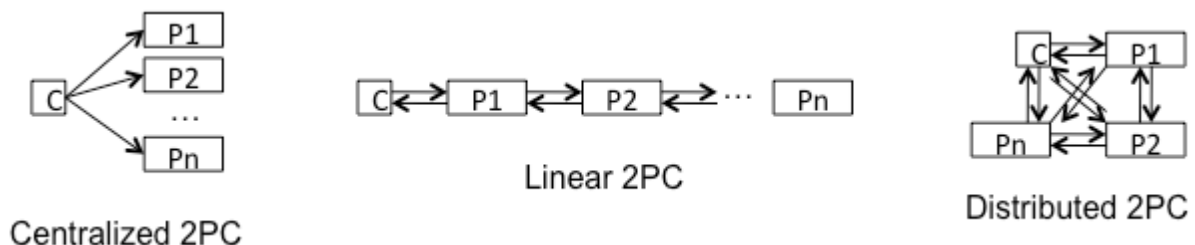
### 1. Protocole de validation à deux phases

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.

#### *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.

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).

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

### *Linear 2PC*

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

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

### *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.
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.

## 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.

## 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 ?
- 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).

- 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.
- 1.4 Ecrire le code Java des interfaces `ProjecteurItf` et `EcranItf`.
- 1.5 Ecrire le code de la classe `Projecteur`. Le contenu du film à visualiser est fourni via le constructeur.
- 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.
- 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).

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

- 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.

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.

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).

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.

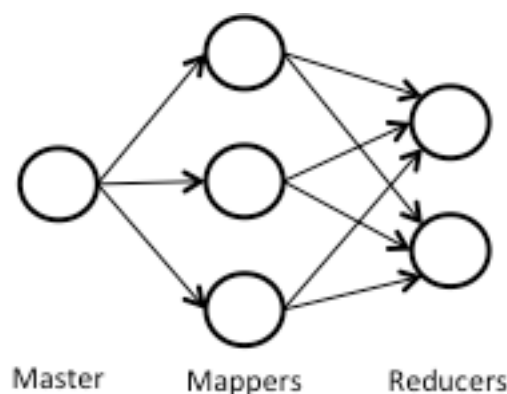
## TD 12 – Map Reduce

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

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 interfaces Java RMI `MapperItf` et `ReducerItf` des objets `Mapper` et `Reducer`.

4. Écrire les classes Java RMI implémentant ces interfaces. Le corps des méthodes peut être écrit en pseudo-code.
5. Proposer en français une implémentation pour la méthode `partition`.