
Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

IV. Objets et répartition

Romain.Rouvoy@univ-lille1.fr

Objets et répartition

But : coupler la notion d'objet avec les concepts de programmation répartie

Objectifs

- meilleure modularisation des applications
- entités logicielles plus réutilisables
- applications mieux *packagées*, portables et maintenables plus facilement

Résultats

- unité de distribution = objet
- un objet = un ensemble de traitement fournis par ses méthodes
- interaction client/serveur = invocation de méthode

⇒ Nombreux aspects techniques à intégrer pour y arriver

Objets et répartition

1. Concepts généraux
 - 1.1 Nommage
 - 1.2 Ramasse-miettes
2. Langage de définition d'interfaces
3. Services non fonctionnels
 - 3.1 Contrôle d'accès
 - 3.2 Concurrency
 - 3.3 Transaction
 - 3.4 Migration
 - 3.5 Réplication

1.1 Nommage

Identifier les objets dans un environnement réparti

- deux objets ≠ sur le même site ou sur des sites ≠
ne doivent pas avoir la même identité
- généralisation de la notion de pointeur à un environnement réparti

Deux niveaux

- référence d'objet distant
 - adresse physique
 - ex. : pointeur + @IP
- annuaire
 - adresse logique
 - faciliter l'accès à l'objet
 - annuaire = serveur "bien connu" de couples <@ logique, @ physique>

1.2 Ramasse-miettes

Détruire les objets qui ne sont référencés par aucun autre

Objectif : récupérer les ressources (mémoire, CPU, disque) inutilisées

Difficulté / aux systèmes centralisés : suivre les références entre sites distants

Deux techniques

- **comptage** de références : on associe un compteur à chaque objet
 - +1 lorsque la référence est acquise
 - 1 lorsque elle est libérée

inconvénient : utilisation mémoire + pas de gestion des cycles de réf.
avantage : simple

- **traçage** (*mark and sweep*) : parcours graphe objets à partir d'une racine
objets non marqués détruits

inconvénient : temps CPU pour le traçage
avantage : pas de modification des objets

Client/serveur orienté objet

1. Concepts généraux

1.1 Nommage

1.2 Ramasse-miettes

2. Langage de définition d'interfaces

3. Services non fonctionnels

3.1 Contrôle d'accès

3.2 Concurrency

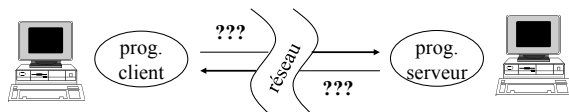
3.3 Transaction

3.4 Migration

3.5 Réplication

2. Langage de définition d'interface

Principe



- quels sont les services disponibles ?
- comment les invoquer ?

Langage de définition d'interface

en anglais : *Interface Definition Language* IDL

- serveur décrit les services
 - client les utilise
- => contrat d'utilisation entre le client et le serveur

2. Langage de définition d'interface

Syntaxe

2 cas pour la syntaxe de l'IDL

- même syntaxe que le langage de programmation (ex. : Java RMI)
- syntaxe propre (ex. : Apache Thrift, SOAP, CORBA IDL)

Exemple

```
interface CompteItf {  
    string getTitulaire();  
    float solde();  
    void deposter( float montant );  
    boolean retirer( float montant );  
};
```

2. Langage de définition d'interface

Syntaxe

IDL décrit

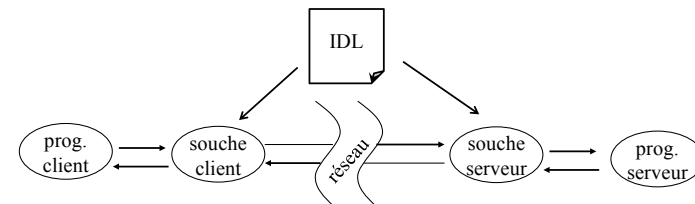
- des types de données
- des services/méthodes/procédures
- des exceptions
- des structures de données (tableau, struct, union, collection, etc.)
- éventuellement d'autres notions issues de langages de programmation
 - constantes, modules, héritage, etc.
- ou liées aux communications distantes
 - adressage, mode d'invocation, mode de passage de paramètres, etc.

⇒ les caractéristiques varient d'un IDL à l'autre

⇒ mais **PAS DE CODE**

2. Langage de définition d'interface

Génération des souches



- dans un langage de programmation donné (par ex. : Java, Python, C++, COBOL, etc.)
⇒ d'où l'intérêt d'une syntaxe IDL ≠ de celle des langages de programmation
- notion de mapping
 - traduction des notions de l'IDL vers le langage de programmation
 - si souche générique, pas de génération
- vocabulaire anglais pour le terme souche : *stub*, *proxy*

2. Langage de définition d'interface

Ecriture de programmes client/serveur avec IDL

1. Définition des services (contrat) ⇒ IDL
2. Ecriture du code des services ⇒ langage de programmation
3. Démarrage des services (serveur) ⇒ langage de programmation
4. Utilisation des services (client) ⇒ langage de programmation

Client/serveur orienté objet

1. Concepts généraux

1.1 Nommage

1.2 Ramasse-miettes

2. Langage de définition d'interfaces

3. Services non fonctionnels

3.1 Contrôle d'accès

3.2 Concurrency

3.3 Transaction

3.4 Migration

3.5 Réplication

3.1 Contrôle d'accès

Gérer le partage des objets dans un environnement réparti

Pour des raisons de sécurité, l'accès à certains objets peut être restreint

- en fonction de l'identité de l'objet appelant
ex : seuls les accès en provenance de l'intranet sont autorisés
- à partir d'une liste de contrôle d'accès
ex : mot de passe, mécanismes de clés de session...

La restriction peut

- interdire complètement l'accès à l'objet
- fournir une vue «dégradée»
ex : autoriser les méthodes qui fournissent un service de consultation
mais interdire celles qui modifient l'état de l'objet

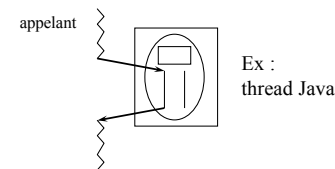
⇒ Nombreuses informations à ajouter aux objets

3.2 Concurrency

Deux types d'objets concurrents

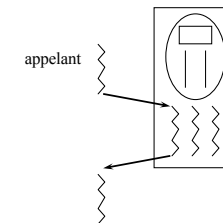
Objet passif

- L'activité
- manipulée de façon explicite
 - orthogonal à l'objet
 - se «plaque» sur les méthodes



Objet actif

Une (+sieurs) activité dédiée à l'objet exécutent les méthodes



3.3 Transaction

Assurer l'exécution cohérente d'une suite d'invocations de méthodes

Exemple : un transfert entre deux objets comptes bancaires

Assurer qu'un ensemble de 2 ou +sieurs invocations de méthodes

```
compte1.depot(50);  
compte2.retrait(50);
```

s'effectuent complètement ou pas du tout (+ propriétés ACID /* cf. BD */)

Problématique de la théorie de la sérialisabilité et des moniteurs transactionnels

- ⇒ intégration du moniteur dans le système réparti objet avec un
- protocole de **validation** (2PC ou 3PC)
 - mécanisme de **verrouillage** des ressources
 - mécanisme de **détection** et de **traitement** des conflits

3.4 Migration

Déplacer des objets

- d'un espace d'adressage à un autre
- d'une zone de stockage à une autre (mémoire, disque)
- d'un **site à un autre**

Objectifs

- réduire les coûts de communication entre objets «bavards»
- répartir la charge vers les sites inutilisés
- augmenter la disponibilité d'un service en le rapprochant de ses clients

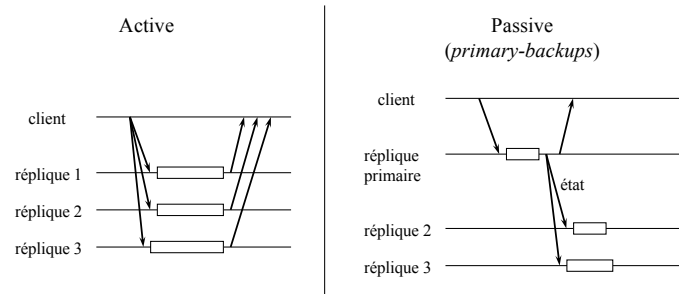
Nombreux problèmes à résoudre

- interrompre ? attendre la fin ? des traitements en cours avant de migrer
- impact de la migration sur la référence ?

3.5 Réplication

Dupliquer des objets sur +sieurs sites

Objectif principal : tolérance aux fautes



Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

IV. Java RMI

Romain.Rouvoy@univ-lille1.fr

Plan

1. Caractéristiques
2. Modèle de programmation
3. Services
4. Fonctionnalités additionnelles
5. Protocole

1. Caractéristiques

Java RMI

Un mécanisme d'invocation de méthodes distantes sur des objets Java

- inclus par défaut dans le JDK depuis 1.1 (évolutions dans JDK 1.2 puis 5)
- package `java.rmi`
- + outils : générateur de souches, serveur de noms, démon d'activation
- implantations alternatives
 - NinjaRMI, Jeremie, NRMI (copie-restauration)

2. Modèle de programmation

Principes

Chaque classe d'objets serveur doit être associée à une interface

⇒ seules les méthodes de l'interface pourront être invoquées à distance

1. Ecriture d'une interface
2. Ecriture d'une classe implantant l'interface
3. Ecriture du programme serveur
4. Ecriture du programme client

≡

1. déclaration des services accessibles à distance
2. définition du code des services
3. instanciation et enregistrement de l'objet serveur
4. interactions avec le serveur

2. Modèle de programmation

Ecriture d'une interface

1. interface Java normale
2. doit étendre `java.rmi.Remote`
3. toutes les méthodes doivent lever `java.rmi.RemoteException`

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

interface CompteInterf extends Remote {
    String getTitulaire() throws RemoteException;
    float solde() throws RemoteException;
    void deposter( float montant ) throws RemoteException;
    void retirer( float montant ) throws RemoteException;
    List historique() throws RemoteException;
}
```

2. Modèle de programmation

Ecriture d'une classe implantant l'interface

1. classe Java normale implantant l'interface
2. doit étendre `java.rmi.server.UnicastRemoteObject`
3. constructeurs doivent lever `java.rmi.RemoteException`
4. si pas de constructeur, en déclarer un vide qui lève `java.rmi.RemoteException`

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

public class CompteImpl extends UnicastRemoteObject
    implements CompteInterf {
    private String nom;
    private float solde;

    public CompteImpl(String nom) throws RemoteException {
        super();
        this.nom = nom;
    }

    public String getTitulaire() { return nom; }
    ...
}
```

2. Modèle de programmation

Ecriture d'une classe implantant l'interface

1. Compilation de l'interface et de la classe avec `javac`
2. Génération de la souche cliente avec `rmic`

```
javac CompteInterf.java CompteImpl.java
rmic CompteImpl
```

Fichier généré

`CompteImpl_Stub.class` : souche cliente

Options ligne de commande `rmic`

`-d path` répertoire pour les fichiers générés

2. Modèle de programmation

Ecriture du programme serveur

1. Instanciation de la classe serveur
2. Enregistrement de l'instance dans le serveur de noms RMI

```
public class Serveur {  
    public static void main(String[] args) throws Exception {  
        CompteInterf compte = new CompteImpl("Bob");  
        Naming.bind("Bob", compte);  
    }  
}
```

- compte prêt à recevoir des invocations de méthodes
- programme "tourne" en permanence tant que compte reste enregistré dans le runtime RMI

2. Modèle de programmation

Ecriture du programme client

1. Recherche de l'instance dans le serveur de noms
2. Invocation des méthodes

```
public class Client {  
    public static void main(String[] args) throws Exception {  
        CompteInterf compte = (CompteInterf) Naming.lookup("Bob");  
        compte.deposer(10);  
        System.out.println( compte.solde() );  
    }  
}
```

2. Modèle de programmation

Exécution des programmes

1. Lancer le serveurs de noms (rmiregistry)
 - une seule fois
 - doit avoir accès au *bytecode* de la souche cliente (→ CLASSPATH)
2. Lancer le programme serveur
3. Lancer le programme client

2. Modèle de programmation

Compléments

Passage de paramètres avec RMI

- types de base (int, float...) par copie
- objets implémentant `java.io.Serializable` passés par copie (sérialisés)
- objets implémentant `java.rmi.Remote` passés par référence
 - la référence RMI de l'objet est transmise
- dans les autres cas, une exception `java.rmi.MarshalException` est levée



2. Modèle de programmation

Compléments

Écriture d'une classe d'objet serveurs

- héritage `UnicastRemoteObject`
- pas toujours possible (si classe \in à une hiérarchie d'héritage)

→ méthode

```
static UnicastRemoteObject.exportObject (Remote)  
enregistrement côté serveur
```

→ méthode de désenregistrement

```
static UnicastRemoteObject.unexportObject (Remote, boolean)  
false : attente fin les requêtes en cours / true : immédiat
```

2. Modèle de programmation

Compléments

Objets serveur prévus pour fonctionner avec \neq types de liaisons

1. objet distant joignable en point à point
2. objets distants répliqués
3. objets distants joignables par diffusion

En pratique seul 1. mis en oeuvre (classe `UnicastRemoteObject`)

Références d'objets distants RMI avec `UnicastRemoteObject`

- adresse IP
- n° port TCP
- identifiant local d'objet (entier)

2. Modèle de programmation

Compléments

Invocations concurrentes de méthodes RMI

Un objet serveur RMI est susceptible d'être accédé
par plusieurs clients simultanément

⇒ toujours concevoir des méthodes RMI *thread-safe*
i.e., exécutable concurrence de façon cohérente

⇒ la création de *thread* est faite automatiquement par le *runtime* RMI

3. Services RMI

Services RMI

- service de nommage
- service d'activation d'objets
- ramasse-miettes réparti

3. Services RMI

Service de nommage

Permet d'enregistrer des liaisons
entre un objet serveur et un nom symbolique

- par défaut port 1099 (autre port : rmiregistry 12345)
- noms "plats" (pas de noms composés, pas de hiérarchies)

URL RMI `rmi://machine.com:1099/nomSymbolique`

`rmi://` et `:1099` facultatifs
`machine.com` par défaut `localhost`

Serveur de noms démarrable

- de façon autonome dans un *shell* avec l'outil `rmiregistry`
- dans un programme par appel de la méthode `static`
 `java.rmi.registry.LocateRegistry.createRegistry(int port)`

3. Services RMI

Service de nommage

Classe `java.rmi.Naming` pour l'accès local

Toutes les méthodes sont `static`

- | | |
|--|-------------------------------------|
| 1. <code>void bind(String,Remote)</code> | enregistrement d'un objet |
| 2. <code>void rebind(String,Remote)</code> | ré enregistrement d'un objet |
| 3. <code>void unbind(String)</code> | désenregistrement d'un objet |
| 4. <code>String[] list(String)</code> | liste des noms d'objets enregistrés |
| 5. <code>Remote lookup(String)</code> | recherche d'un objet |

Les paramètres `String` correspondent à une URL d'objet RMI

- | | |
|-------|--|
| 1 2 3 | accessibles localement uniquement |
| 1 | lève une exception si le nom est déjà enregistré |
| 4 | URL du <code>rmiregistry</code> |

3. Services RMI

Service de nommage pour l'accès distant

Classe `java.rmi.registry.LocateRegistry`
`static Registry getRegistry(String host, int port)`

`java.rmi.registry.Registry` : même méthodes que `Naming`

3. Services RMI

Service d'activation d'objets

Objets serveurs activables à la demande (depuis JDK 1.2)
en fonction demande clients

- démon `rmid`
- package `java.rmi.activation`

Avantages

- évite d'avoir des objets serveurs actifs en permanence
 (*ie.* "tournant" dans une JVM)
 → trop coûteux si beaucoup d'objets serveurs
- permet d'avoir des références d'objets persistantes
 → en cas de *crash* d'objet serveur
 le démon peut le relancer avec la même référence
 → les clients continuent à utiliser la même référence

3. Services RMI

Service d'activation d'objets

```
public class Serveur {  
    public static void main(String[] args) throws Exception {  
        ActivationGroupDesc gDesc = new ActivationGroupDesc(null, null);  
        ActivationGroupID gID =  
            ActivationGroup.getSystem().registerGroup(gDesc);  
        ActivationGroup.createGroup(gID, gDesc, 0);  
  
        MarshalledObject mo = new MarshalledObject("Bob");  
        ActivationDesc desc = new ActivationDesc("CompteImpl", "", mo);  
        CompteInterf compte = (CompteInterf) Activatable.register(desc);  
        Naming.bind("Bob", compte);  
    }  
}
```

Programme client inchangé

Lancer `rmiregistry` et **démon d'activation** (`rmid`)

3. Services RMI

Extension du principe : *pool* d'objets

Avoir un ensemble d'objets prêts à traiter les requêtes

Même problématique que le *pool* de *threads*

⇒ gestion de la taille du *pool* (fixe, variable)

⇒ à programmer

3. Services RMI

Ramasse-miettes réparti

Récupérer les ressources (mémoire...)

occupées par un objet que personne ne référence

→ *i.e.* que l'on ne pourra plus jamais accéder

Difficulté : environnement distribué donc référencement à distance

Dans une JVM : mécanisme *mark-and-sweep*

1. parcours du graphe de référencement des objets

2. destruction des objets non visités

Avec RMI : double mécanisme géré par le *Remote Reference Manager*

- comptage de référence : # de clients référençant l'objet

- bail (*lease*) : mémoire "louée" à l'objet pour un temps fini

3. Services RMI

Ramasse-miettes réparti

Comptage

- chaque transmission de référence +1
- chaque fin de référencement -1

Bail

- par défaut 10 min (réglable par la propriété `java.rmi.dgc.leaseValue`)
- but : se prémunir
 - partitions de réseaux
 - pertes de message de déréférencement

Si le compteur tombe à 0 ou si le bail expire,
l'objet devient candidat pour le ramassage local (*mark-and-sweep*)

3. Services RMI

Ramasse-miettes réparti

Attention : un objet serveur "normal"
instancié par un programme serveur qui "tourne" en permanence
est toujours référencé par ce programme

⇒ il n'est pas ramassé (même au delà des 10 min)

⇒ le ramassage concerne des objets créés dont on "perd" la référence

Client

```
BarRemote b = foo();  
...  
b = null;
```

Serveur

```
BarRemote foo() {  
    return new BarRemote();  
}
```

4. Fonctionnalités additionnelles

- chargement dynamique de classes (`RMIClassLoader`)
- personnalisation des communications
- génération dynamique des souches

4. Fonctionnalités additionnelles

Chargement dynamique de classes

- Java charge les `.class` à la demande à partir du disque (`CLASSPATH`)
- RMI introduit en + un mécanisme de chargement des classes
à distance par HTTP ou FTP

Avantage : classes déployées sur 1 seul site (+ rapide + simple à gérer)

Inconvénient : *single point of failure*

Utilisation

- propriété `java.server.rmi.codebase` : URL du serveur de téléchargement
- classe `RMISecurityManager`

4. Fonctionnalités additionnelles

Personnalisation des communications

RMI utilise des *sockets* TCP

- côté client et serveur
- attribuées automatiquement par défaut

Possibilité de personnaliser ces sockets pour

- forcer l'utilisation de *sockets* précises
- tracer les communications
- crypter et/ou signer les données
- introduire des traitements "à l'insu" des objets clients/serveurs RMI

⇒ redéfinir le constructeur

```
protected UnicastRemoteObject( int port,  
                               RMIClientSocketFactory csf, RMIServerSocketFactory ssf )  
- ou utiliser static UnicastRemoteObject.export( Remote, int, ... )
```

4. Fonctionnalités additionnelles

Personnalisation des communications

```
interface RMIClientSocketFactory {
    java.net.Socket createSocket( String host, int port );
}

interface RMIServerSocketFactory {
    java.net.ServerSocket createServerSocket( int port );
}
```

- ⇒ définir 2 classes qui implament ces interfaces
- ⇒ définir des sous-classes de `Socket` et `ServerSocket` pour personnaliser le fonctionnement des sockets
- ex : `javax.net.ssl.SSLSocket`, `javax.net.ssl.SSLServerSocket`

4. Fonctionnalités additionnelles

Génération dynamique des souches

- `rmic` génère les souches de communication
- A partir JDK 5, possibilité de générer les souches dynamiquement
 - génération de *bytecode* à la volée
 - chargement dynamique de la classe générée

5. Protocole

Protocole JRMP

2 possibilités pour acheminer les invocations de méthodes distantes

- JRMP : protocole Sun (`UnicastRemoteObject`) utilisé par défaut
- IIOP : protocole OMG pour CORBA

Messages JRMP sortants

- *Call* : véhicule une invoc. de méth. (+ *callData*)
- *Ping* : teste le bon fonctionnement d'un serveur

Messages JRMP entrants

- *Return* : véhicule le retour de l'invoc. (+ *returnValue*)
- *PingAck* : acquittement d'un message *Ping*

5. Protocole

Structure des paquets échangés par JRMP

4 octets	2 octets	1 octet	n octets
En-tête	Version	Protocole	Message(s)

En-tête : *magic number* (JRM1)

Version : numéro de version du protocole

Protocole : 3 possibilités

- `SingleOpProtocol` : 1 seule invoc. par paquet
- `StreamProtocol` : +sieurs invoc. **vers un même obj.** les unes à la suite
- `MultiplexProtocol` : +sieurs invoc. **vers une même machine** multiplexées sur la même connexion

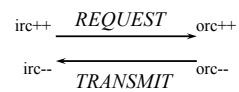
5. Protocole

Mécanisme de contrôle de flux JRMP

But : éviter qu'un buffer plein pour 1 connexion bloque toutes les autres
éviter qu'1 connexion qui ne se termine pas bloque toutes les autres
(par exemple en cas d'appels récursifs)

2 compteurs (en nombre d'octets) pour chaque cx RMI multiplexée

- *input request count* (irc)
- *output request count* (orc)



- irc et orc ne doivent jamais être négatifs
- irc ne doit pas dépasser une valeur (en nb d'octets) qui le bloquerait

Université Lille 1 Master mention Informatique – M1

Construction d'applications réparties

IV. Akka

Romain.Rouvoy@univ-lille1.fr

Plan

1. Caractéristiques
2. Notion d'acteur
3. Invocation à distance
4. Fonctionnalités avancées

1. Caractéristiques

Akka

- basé sur la notion d'acteur
 - notion créée en 1973 par Carl Hewitt
 - programmation concurrente et répartie
 - popularisée par le langage Erlang
- créé en 2009 par Jonas Bonér
- Java, Scala, .NET (C#, F#)

Objectifs

- programmes concurrents et répartis
 - légers
 - robustes
 - performants

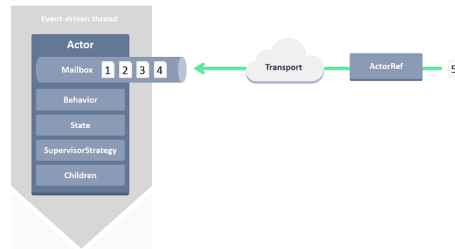
Applications dans de nombreux domaines

- systèmes bancaires, telecom, jeux en ligne, data mining, finance, rsx sociaux, etc.

2. Notion d'acteur

Acteur

- entités avec des capacités de traitement, stockage et communication
- communication par envoi de messages asynchrones
- transparence à la répartition
- état
- comportement
- boîte réception messages
- organisation hiérarchique
 - acteurs enfants
 - stratégie de supervision
 - détection de fautes
 - des enfants



2. Notion d'acteur

Acteur

- objectif : rendre la programmation concurrente plus simple et plus sûre
- chaque acteur est mono-threadé
- multi-thread* => plusieurs acteurs
- acteurs autonomes, indépendants, facilement redémarrables
- communiquent par messages asynchrones
- particulièrement bien adaptés
 - à des programmes répartis
 - à des processus (ensemble de tâches séquentielles ou concurrentes)
 - ex : processus de compilation, de génération de code, de tests, etc.
 - aux entités en lien avec la programmation concurrente
 - listener, callback*
 - à la gestion de machines à états finis

2. Notion d'acteur

Acteur en Java avec Akka

- classe qui hérite de `akka.actor.UntypedActor`
- implémente la méthode `onReceive`
 - définit le comportement de l'acteur
 - réception d'un message
 - tests de son type

```
public class GreetingActor extends UntypedActor {
    public void onReceive( Object message ) throws InterruptedException {
        if( message instanceof Greeting ) { ... }
        else if( message instanceof String ) { ... }
        else ...
        else { unhandled( message ); }
    }
}
```

- message : objet sérializable (sérrialization Java ou personnalisée)

2. Notion d'acteur

Messages

- message : objet sérializable (sérrialization Java ou personnalisée)

```
public class Greeting implements Serializable {
    public String who;
    public Greeting( String who ) { this.who = who; }
}
```

2. Notion d'acteur

Création des acteurs

1. notion de système d'acteurs (`akka.actor.ActorSystem`)
 - un ensemble d'acteurs qui partagent des caractéristiques communes
 - un annuaire pour enregistrer et retrouver des acteurs
2. chaque acteur est nommé
3. chaque acteur a une référence (`akka.actor.ActorRef`)

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter1, greeter2;

greeter1 = system.actorOf( Props.create(GreetingActor.class), "greeter1" );
greeter2 = system.actorOf( Props.create(GreetingActor.class), "greeter2" );

// ... envois de messages ...

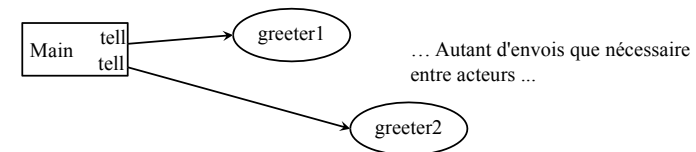
system.shutdown(); // arrêt du système (fin des acteurs)
```

2. Notion d'acteur

Envoi asynchrone de messages à un acteur

- la méthode `tell` permet d'envoyer un message à un acteur
- l'envoi est asynchrone et non bloquant
- les acteurs sont passifs
 - jusqu'à ce qu'ils reçoivent un message

```
greeter1.tell( new Greeting("Bob"), ActorRef.noSender() );
greeter2.tell( new Greeting("Robert"), ActorRef.noSender() );
```



2. Notion d'acteur

Quelques méthodes utiles à un acteur

- `getSelf()` : retourne la référence (`ActorRef`) de l'acteur courant
- `getSender()` : retourne la référence (`ActorRef`) de l'acteur ayant envoyé le message
- `ActorRef.noSender()` : le message n'a pas été émis par un acteur

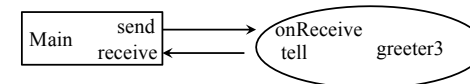
2. Notion d'acteur

Envoi semi-synchrone de messages à un acteur

- notion de boîte à lettres (`akka.actor.Inbox`)
- méthodes `send` et `receive`
- `receive` bloquant jusqu'à réception d'un message
 - à concurrence d'un timeout

```
Inbox inbox = Inbox.create(system);
inbox.send( greeter3, new PingMessage() );
// ... code qui ne nécessite pas reply ...

Object reply = inbox.receive(Duration.create(5, TimeUnit.SECONDS));
if( reply instanceof PongMessage ) { ... }
```



2. Notion d'acteur

Acteur typé

- implémente une interface (d'où le typage)
- envoi synchrone (requête réponse)

```
public interface CompteBancaireItf {
    double crediter( double montant ); }

public class CompteBancaireImpl implements CompteBancaireItf {
    private double solde;
    public double crediter( double montant ) {
        solde += montant; return solde; } }

CompteBancaireItf monCompte =
    TypedActor.get(system).typedActorOf(
        new TypedProps<PingImpl>(PingItf.class, PingImpl.class) );

double solde = monCompte.crediter(42);
```

2. Notion d'acteur

Acteur typé et envoi semi-synchronz

- notion de Future

```
public interface CompteBancaireItf {
    Future<List<String>> operations(); }

public class CompteBancaireImpl implements CompteBancaireItf {
    private List<String> operations;
    public Future<List<String>> operations() {
        return Futures.successful(operations); } }

Future<List<String>> f = monCompte.operations();
// ... éventuellement autres traitements
List<String> operations = Await.result(f, Duration.Inf());
```

Plan

1. Caractéristiques
2. Notion d'acteur
3. Invocation à distance
4. Fonctionnalités avancées

3. Invocation à distance

Remoting

- mécanisme d'envoi des messages à distance entre systèmes d'acteurs
- fichier de configuration application.conf

```
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
  }
  remote {
    enabled-transport = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "localhost"
      port = 2552
    }
  }
}
```

- plusieurs systèmes d'acteurs sur la même machine en utilisant des ports différents

3. Invocation à distance

Côté serveur

```
ActorSystem system = ActorSystem.create("MySystemServer");
ActorRef greeter4 =
    system.actorOf( Props.create(GreetingActor.class), "greeter4" );
```

- création de l'acteur greeter4 dans le système MySystemServer

3. Invocation à distance

Côté client

```
ActorSystem system = ActorSystem.create("MySystemClient");
String url = "akka.tcp://MySystemServer@localhost:2552/user/greeter4";
ActorSelection greeter4 = system.actorSelection(url);
greeter4.tell("Hello World!", ActorRef.noSender() );
```

- recherche de la référence de l'acteur distant greeter4 sur le port 2552 de localhost
- envoi d'un message à l'acteur greeter4
- format de désignation des acteurs distants
akka.<protocol>://<actorsystemname>@<hostname>:<port>/<actor path>

3. Invocation à distance

Déploiement à distance d'acteur

Programmatiquement

```
String host = "akka.tcp://MySystemServer@localhost:2552";
Address addr = AddressFromURString.parse(host);
ActorRef greeter5 =
    system.actorOf(
        Props.create(GreetingActor.class).
            withDeploy(new Deploy(new RemoteScope(addr))),
        "greeter5");
greeter5.tell("Hello World!", ActorRef.noSender());
```

- déploiement d'un acteur GreetingActor
sur le système d'acteurs installé sur le port 2552 de localhost

3. Invocation à distance

Déploiement à distance d'acteur

Via le fichier de configuration application.conf

```
akka {
  actor {
    deployment {
      /greeter5 {
        remote = "akka.tcp://MySystemServer@localhost:2552"
      } } }
}

ActorRef greeter5 =
    system.actorOf( Props.create(GreetingActor.class), "greeter5");
greeter5.tell("Hello World!", ActorRef.noSender() );
```

- déploiement d'un acteur GreetingActor
sur le système d'acteurs installé sur le port 2552 de localhost

Plan

1. Caractéristiques
2. Notion d'acteur
3. Invocation à distance
4. Fonctionnalités avancées

4. Fonctionnalités avancées

Fonctionnalités avancées

- cluster gestion de groupes d'acteurs
- persistance sauvegarde des acteurs, journalisation des messages envoyés
- agent entité permettant de partager un état global entre acteurs
- supervision d'acteurs
 - gérer les comportements en erreur (exceptions)
 - définition de la politique de reprise/rédémarrage de l'acteur
- utilisation de protocoles réseaux TCP, UDP avec des acteurs
- utilisation de bus de messages ZeroMQ, Camel avec des acteurs
- programmation de machines états/transitions
- fonctionnalités Java 8 (lambda) pour faciliter la programmation

4. Cluster

Programmation répartie

- communications entre acteur client et acteur serveur distant
 - perte de service en cas de panne
 - solution : réplication (1 acteur => n acteurs)
 - notion de cluster
 - cluster Akka peuvent être hétérogènes (différents types d'acteurs)
 - notion de rôle dans le cluster
- 2 besoins principaux
- détecter les pannes d'acteurs au sein du cluster
 - *failure detector*
 - on parle de défaillances
 - mécanisme heartbeat avec délai variable pour s'adapter aux contraintes du réseau
 - connaître la composition du cluster
 - *membership service*

4. Cluster

Membership service

Mécanisme *peer-to-peer* (P2P)

- ensemble de noeuds (acteurs) (en anglais *node*)
 - pas de point central
 - élection d'un *leader*
 - ré-élection d'un nouveau *leader* en cas de panne
 - chaque noeud est identifié par un triplet
 - `hostname:port:uid`
 - la composition du cluster est diffusée à tous avec un protocole dit épidémique (*gossip protocol*)
 - les noeuds peuvent rejoindre et quitter le cluster à tout moment
 - les arrivées et départs sont notifiés à tous les noeuds du cluster
 - structure de données réparties dite CRDT
- Convergent and Commutative Replicated Data Type*
<http://hal.upmc.fr/file/index/docid/555588/filename/techreport.pdf>
- garantie de convergence en cas de mises à jour concurrentes

4. Cluster

Exemple de cluster

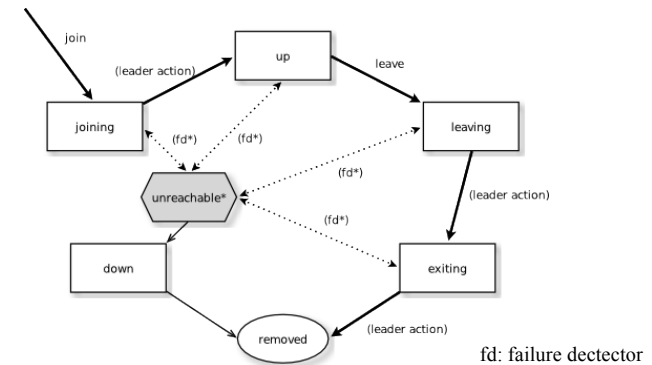
- cluster d'acteurs qui effectuent des transformations sur un texte
- 2 rôles pour les noeuds du cluster
 - *backend worker* effectue les transformations
 - *frontend* reçoit les requêtes et les transmettent à un *backend worker*
- des *frontend* et des *backend* peuvent être ajoutés dynamiquement au cluster

Acteurs : TransformationFrontend, TransformationBackend

Messages : TransformationJob, TransformationResult (un texte dans les deux cas)

4. Cluster

États pour les noeuds d'un cluster



4. Cluster

Événements liés à l'état du cluster

- événements générés lors de l'arrivée/départ d'un noeud dans le cluster

Exemple d'événements

- MemberUp
- MemberExited
- MemberRemoved
- UnreachableMember rq : par le détecteur de défaillances
- ReachableMember rq : après avoir été détecté en défaillance
- un événement = un message
- les noeuds du cluster peuvent s'abonner (ou pas) à ces messages
- message CurrentClusterState avec la composition actuelle du cluster envoyé lors de l'abonnement

4. Cluster

Acteur TransformationBackend

- reçoit un message TransformationJob
- modifie le texte
- renvoie le résultat

```
public class TransformationBackend extends UntypedActor {

    public onReceive( Object message ) {
        if( message instanceof TransformationJob ) {
            TransformationJob job = (TransformationJob) message;
            String text = job.getText();

            // ... transformation du texte ...

            Object result = new TransformationResult(...);
            getSender().tell(result);
        } }
    }
```

4. Cluster

Acteur TransformationFrontend

- reçoit un message TransformationJob
- choisit un backend
- transmet le message au backend choisi

Cela impose de

- connaître la liste des *backends*
 - abonnement aux messages d'état du *cluster*
 - tenir à jour la liste des *backends*
- politique de choix du *backend*
 - notion de routeur
 - différents types de routeurs fournis par Akka (+ programmation)
 - Round robin, Random, SmallestMailbox, etc.

Code complet <http://www.typesafe.com/activator/template/akka-sample-cluster-scala>

Conclusion

Akka

- un ensemble de concepts légers et performants pour la programmation
 - répartie
 - concurrente
 - tolérant les pannes
- librairie disponible dans plusieurs langages (Java, Scala, C#)
- site de référence (docs, etc.) : <http://akka.io>