REactive Java Objects
Reactive Operating System

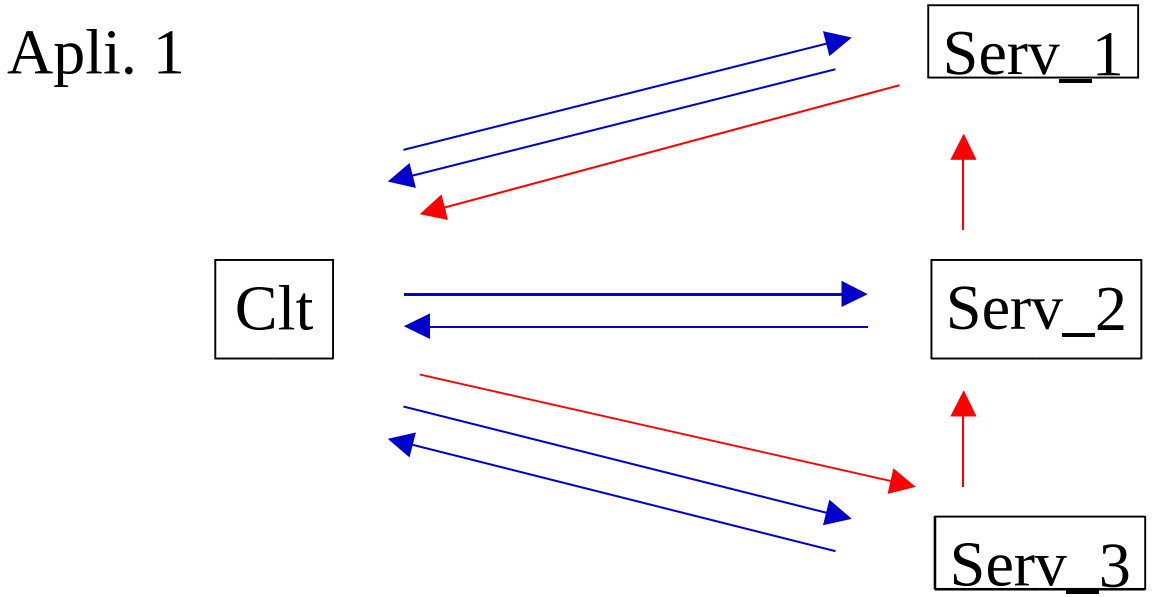
Raúl ACOSTA BERMEJO

Index

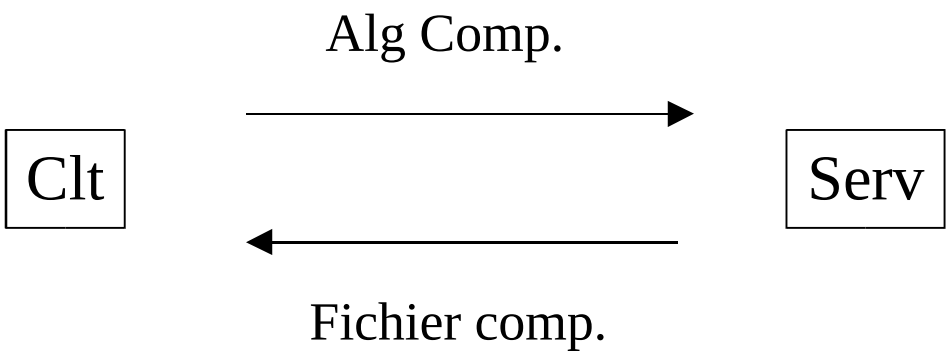
- Agents Mobiles
- RAMA
- REJO
- ROS
- Démos
- Conclusions

Agents Mobiles

Un Agent Mobile c'est un programme qui peut migrer d'une machine à une autre dans un réseau hétérogène. Le programme choisit lui−même quand et où il veut migrer.



Apli. 2



Avantages

- Réduction de la bande passante utilisée.
- Réduction du temps de latence (latency).
- Continuité de fonctionnement en cas de déconnexion.
- Equilibrage de la charge.
- Développement dynamique des applications.

Conception d ’un Système d ’Agents Mobiles

- Mobilité:

  Forte ou faible.

- Routage

  Transfert d’adresses. ou Liste de sites et de services.

- Communication.

  Message Passing, RMI (RPC), Publish Suscribe, Broadcast.

  Com. Inter−groupe ou Intra−groupe, Local Inter−groupe ou Glo.

- Nommage

  Nom local+Nom machine,Domain Name Server, Nommage global.

- Langage: interprété ou compilé.

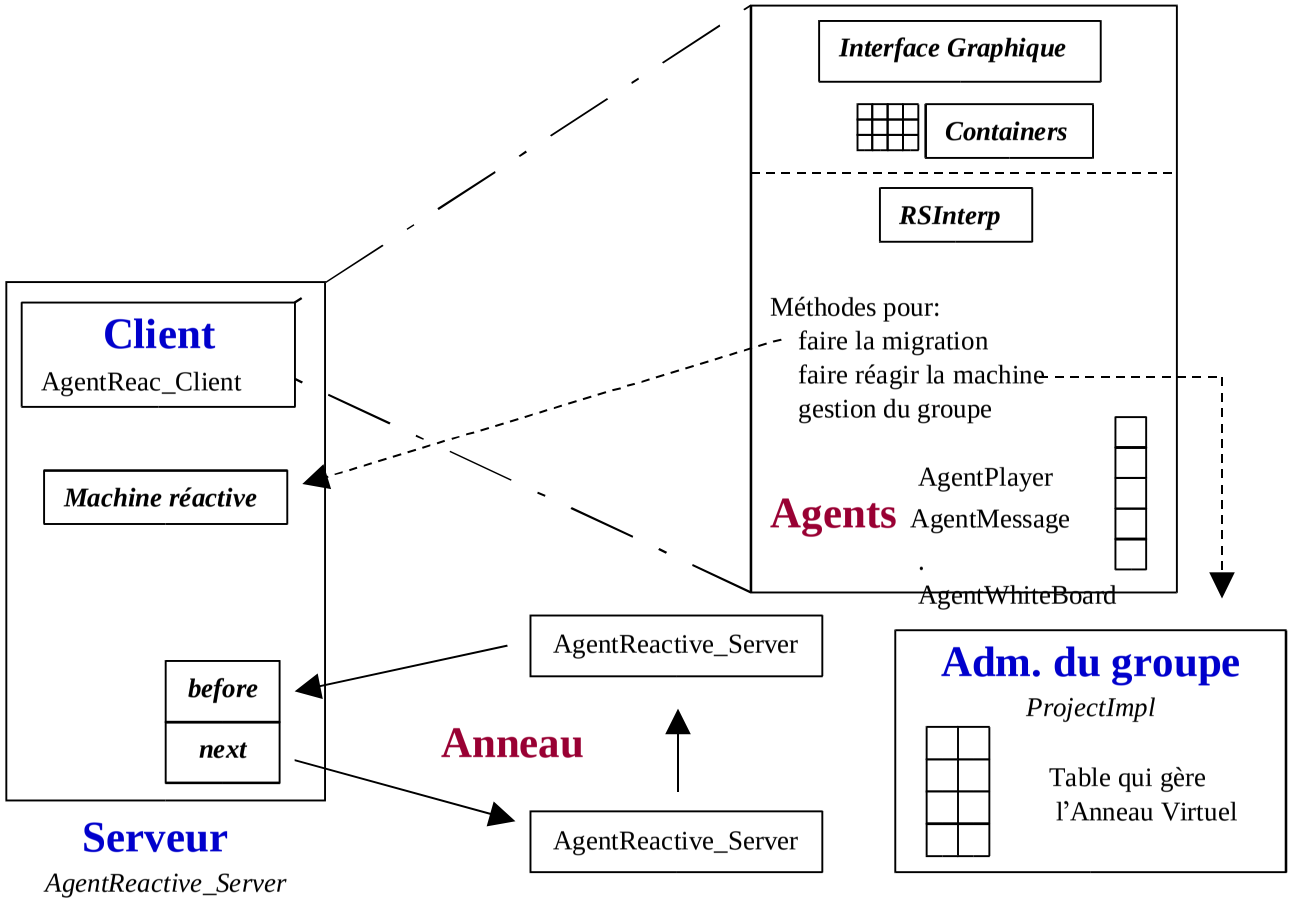
  Portabilité, robustesse, sécurité, efficacité.

RAMA

Reactive Autonomous Mobil Agent

- Créé par Navid Nikahaein en Sep/99, DEA RSD ESSI.
- RAMA est un ensemble de ***bibliothèques*** Java.
- Ces bibliothèques servent à construire des ***Agents Mobiles*** en utilisant l ’***Approche Réactive***.
- Les agents s ’exécutent sur une ***plate−forme*** qui offre, entre autres choses, la notion de groupe.

Architecture



Exemple

```
package rama;
import  inria.meije.rc.sugarcubes.*;
import  java.awt.Label;

public class WhiteBoardAgentCode extends AgentPrimitive
{
   public void start(String home)
   {
      try{
         Instruction WhiteBoardAgent=
            new Cube(new JavaStringValue("WhiteBoard_Agent_" +(counter++)+home),
                     new JavaObjectValue(
                         new WhiteBoardAgent("WhiteBoard_Agent_"+(counter-1),home)),
                     new Until("kill", new Seq(
                                       new Await("Migrate"),
                                       new Loop(
                                             new Until(new OrConfig(
                                                         new PosConfig("Migrate"),
                                                         new PosConfig("Return")),
                                                       new Merge(new Seq(
                                                                   new Await("Return"),
                                                                   _migrationToHome()    )
                                                                ,new Seq(
                                                                   new Await("Migrate"),
                                                                   _migrationCode()      )
                                                       )
                                             )
                                       )
                         ))
                     ,new JavaInstruction(){ public void execute(Link self){
                         ((Agent)self.javaObject()).disposeMessage();
                     }}

                     . . .
            );
         AgentReactive_client.currentSite.machine.add(WhiteBoardAgent);

      }catch(Exception e){
         System.out.println("Error when sending the agent:"+e); }
   }
}
```

Nom

Linked Obj

Programme

Handlers

Bilan

Caractéristiques:

- Concurrence.
- Interactivité
  Diffusion et Inst. Réactifs.
- Mobilité.
- Dynamicité (agents, platforme).
- Multi−plateforme.

Grâce a **SugurCubes** et **Java**

Désavantages:

- Architecture Monolithique.
- Programmation (style) à la SugarCubes.
- Problèmes de nommage.
- Diffusion local et donc il faut migrer
  pour communiquer.

Avantages:

- Routage d'anneau
  mais aussi une désavantage
  Performance et le seul Alg. de routage
- Modèle d 'agent réactif.
- Possibilité d'implémenter les services
  avec les agents.

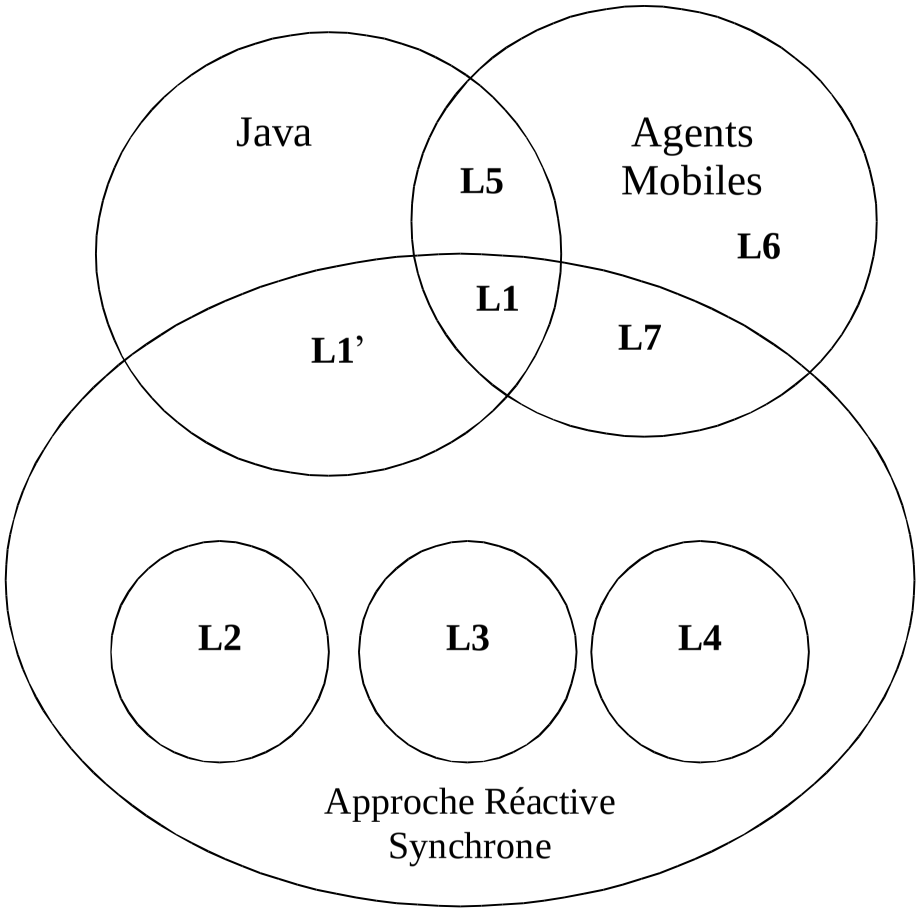
Agent (Cube) = Link + Shell + Freeze

Motivation

On voudrait pouvoir:

- Programmer les systèmes réactifs à la Reactive Scripts
  mais C'est pas du Java, pas de plate−forme, ...
- Avoir des services qui concerne seulement aux agents:
  migration, nommage, routage, etc.
- Ajouter et enlever des modules à la plate−forme pour:
  - Choisir si on veut se synchroniser avec d'autres sites,
    Distributed Reactive Machines (DRM).
  - Avoir ou pas la notion de groupe.
  - Choisir le mécanisme de routage.
  - ...

Motivation



L1 : ***Langages Réactifs d'Agents en Java***
REJO, MARS
L1' : ***Langages Réactifs en Java***
SugarCubes, Reactive Scripts, Junior
L2 : ***Langages Impératifs Réactifs***
Esterel
L3 : ***Langages Déclaratifs Réactifs***
Signal, Lustre
L4 : ***Langages " High Order " Réactifs***
Bibliothèque en SML [RP98]
L5 : ***Langages d'Agents en Java***
Aglets, Concordia, Voyager.
L6 : ***Langages d'Agents***
Messengers
L7 : ***Langages Reactifs d'Agents***
ROL [GM99]

Pourquoi Java?

Parce que grâce à Java on

- a tous les avantages de la programmation orientée objet.
- peut construire un modèle d’objet réactif comme ROM.
- génère du code portable.
- implémente facilement la migration en utilisant les RMI.

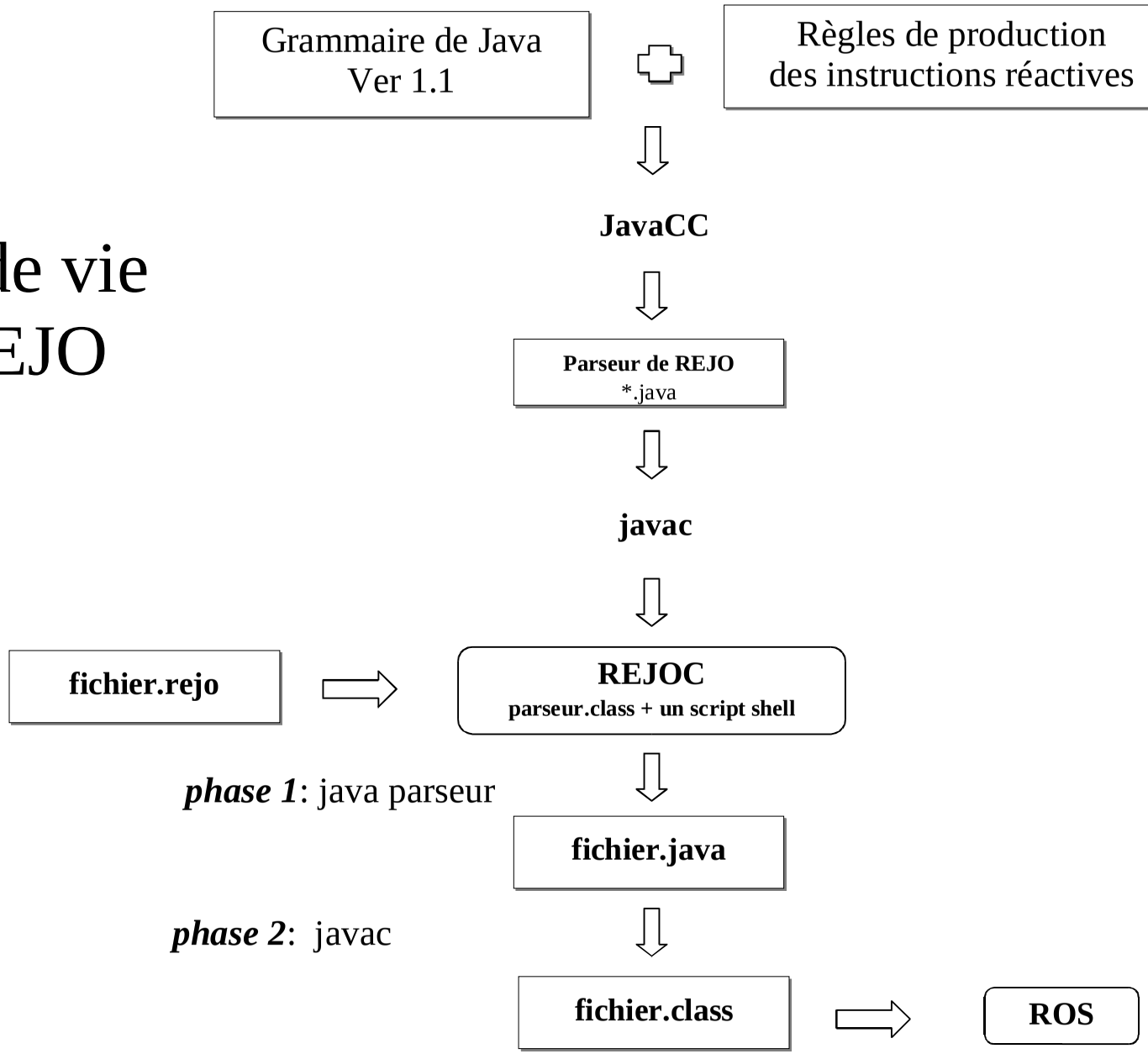
REJO

REactive Java Object

REJO est une extension à Java qui génère des objets qui peuvent être considérés comme des ***Objets Réactifs***, avec leurs données et leurs instructions réactives.

Ces objets peuvent être considérés comme des ***Agents Mobiles*** car ils peuvent migrer sur une plate−forme (ROS) qui offre toutes les fonctionnalités dont ils ont besoin.

Cycle de vie de REJO



Structure d'un REJO

```
import ros.kernel.bin.*;
import java.awt.*;

public class rejo implements Agent
{
 int cont;

      public void Methodes_1()
      {
         body
      }

      public reactive rmain(String[] args)
      {
       int ini=5;

         Merge{
            cont=ini;
            Methodes_1();
            Until("Preemption1" && "Preemption2")
              Loop{
                Methodes_1();
                Stop;}
         ||
            call Methodes_2();
         }
      }

      public reactive Methodes_2()
      {
        body
      }
}
```

REJO = { **Données** + **Code Java** + **Code Réactif**

Exemples de traduction

```
react Method( param )
{
   var loc;
       body
}
```

= >

```
Program Method()
{
     obj = new _Method();
     obj.name = init;
     prg = Jr.Link( obj,  Trad(body) )
}

Class _Method()
{
   var loc;
   param;
}
```

Identifier si on utilise des:

Var Loc	_Method.name
ou ***Glo***	name

Exemples de traduction

```
i1;                 = >     Jr.Seq( i1, Jr.Seq(
i2;                             i2, Jr.Seq(
… ;                             …, Jr.Nothing()  ) ) )
```

```
Merge               = >     Jr.Par(
{
   i1;                              i1, Jr.Par(
||
   i2;                              i2, Jr.Par(
||
   …                               …, Jr.Nothing()  ) ) )
}
```

Exemples de traduction

```
Loop {                          = >     Jr.Loop ( body )
  body
}
```

Assig: i=5; = > Jr.Atom(new Action(){

i= Var Loc ou Glo public void execute(Environment env)

Exp. Arit, logique, etc. {

Invo: meth(); Inst;

obj.meth(); }}

)

```
Atom{
   Expressions;
   Structures de Control de Flow
        if, for, while.
}
```

Identifier si on utilise des:

Var Loc ((_meth)env.linkedObject()).name

ou ***Glo*** name

Exemples de traduction

```
Repeat( Cte ){        = >        Jr.Repeat( Cte , body )
   body
}
```

```
Repeat( Var ){        = >        Jr.Seq(Jr.Atom(new Action(){
  body                                    public void execute(Environment env)
}                                         {
                                              IntWrap = Var;
                                          }}),
                                      Jr.Repeat( IntWrap,
                                                 body)
                                  )
```

La même chose pour l’If, When.

Exemples de traduction

```
Wait  "eve" + var + met()   = >   Jr.Seq(Jr.Atom(new Action(){
                                          public void execute(Environment env)
                                          {
                                             StrWrap = "eve" + var + met();
                                          }}),
                                        Jr.Await( StrWrap )  )
```

```
Wait  "eve" && var          = >   Jr.Seq(Jr.Atom(new Action(){
Wait  var ||  met()                       public void execute(Environment env)
Wait  ! "eve "                            {
                                             sw_N = Exp;
                                          }}),
                                       Jr.Await(
                                           Jr.And( Jr.Presence( sw_1 ),Jr.And(  ) )
                                           Jr.Or( Jr.Presence( sw_N ),Jr.Or(  ) )
                                           Jr.Nor( … )        ) )
```

Bilan

REJO ne seulement fait plus facile l'utilisation des instructions réactives en évitant l'utilisation de parenthèse, il également permet:

- Mélanger le model réactif avec les ***variables en Java***,.
  Dans les instructions Repeat, If, Wait, etc.
  En cachant l'utilisation de wrappers.
- Mélanger le model réactif avec les ***instructions en Java***.
  Des expressions arithmétiques et des invocation à méthodes Java.
  Exécution atomique de ces instructions Java.
- Introduire la notion de ***Méthodes Réactives*** qui permettent:
  Faire une programmation modulaire.
  Réutiliser du code en faisant des invocations à ces méthodes.
- Introduire la notion de ***variable local*** dans les méthodes réactifs.

En conclusion, tous ces éléments définissent un **Model d'Objet Réactif** particulier qui n'existe pas en Junior. Les autres models définit en Java sont Rhom et SugarCubes.

ROS

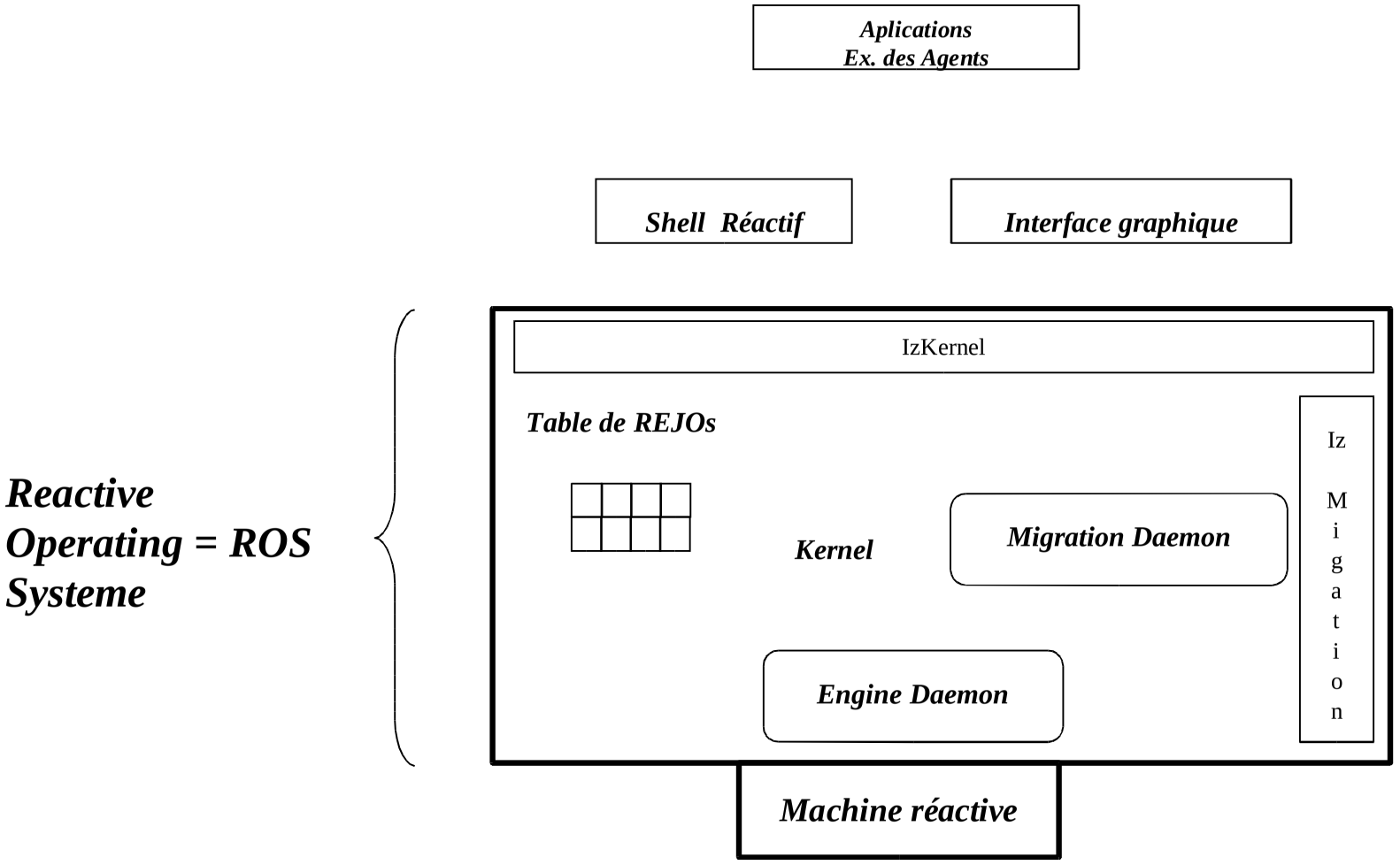
Reactive Operating System

ROS est un système construit sur le modèle réactif qui permet l'exécution des objets réactifs REJO.

L'architecture de ROS est similaire à celle d'un Système d'Exploitation Réparti (SER), le ***micro−kernel***.

ROS a été implémenté avec SugarCubes et Junior et donc il a tous les avantages de RAMA.

Architecture



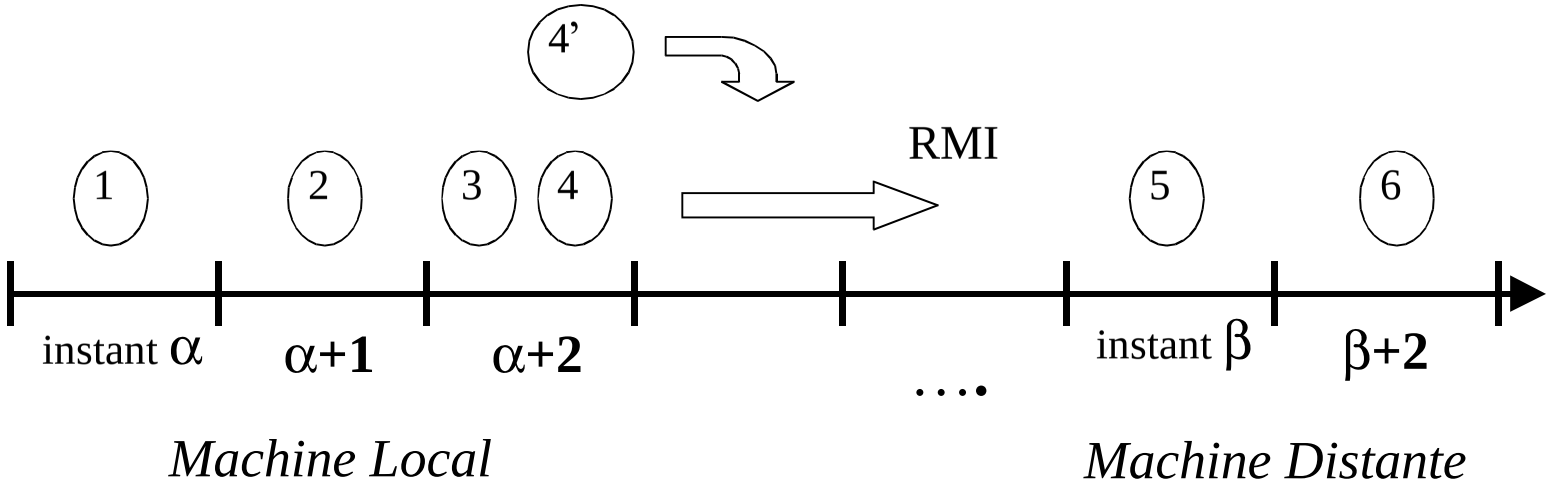
Freezable

t,E −− ß −−> t', E' ß != STOP
=> Freezable(S,t),E −−ß−−>Freezable(S,t'),E'

t,E −−STOP−−> t',E ' Freezable*(S,t'),E' −− ß−−> v,E''
=> Freezable(S,t),E −−ß−−> v,E''

S ∈ E eoi(E)=false
=>Freezable*(S,t) −−TERM−−> Nothing, E(S:=Par(t,E(S)))

Mécanisme de migration



1. Exécuter les instruction qui provoque la migration.
   Inst. dans l ’agent ou Rajoute des instructions
2. Geler les instructions qui restent à exécuter.
3. Enlever les instructions de la machine.
4. Envoyer une copie des instructions (tout l ’objet) => 2 ou 3 inst.
   4 ’. Si la migration échoue il faut rajouter les instructions.
5. Charger les instructions reçues dans la machine distante.
6. Exécuter les instructions.

Conclusions

Un ***nouveau langage*** appelé REJO qui:

- Rend la programmation des systèmes réactifs plus facile (syntaxe).
- Garde la plupart des propriétés de SugarCubes et Junior.
- Permet la programmation des systèmes réactifs en utilisant un langage orienté objets. Le résultat est un ***langage réactif basé objets*** qui peut devenir orienté objets.
- Permet la programmation en Java d'agents migrants (code portable).

Conclusions

Une ***nouvelle plate−forme*** appelé ROS qui:

- Exécute des objets réactifs (REJOs).
- Permet la migration des objets réactifs.
- Montre l'utilisation des systèmes réactifs comme une alternative a l'utilisation traditionnelle de threads.
- Pose une architecture modulaire et minimale analogue à celle d'un SE auquel on peut ajouter et enlever des modules selon les besoins.

Travail Futur

La suite des travaux porte sur les points suivants :

- Ajouter la propriété de persistance aux agents.
- Ajouter au ROS la possibilité de se synchroniser avec des autres ROS.
- Créer automatiquement l’automate équivalent de la partie réactive d’un programme REJO.
- Ajouter d’autres propriétés à ROS (securité, ...).

Questions?

Raul.Acosta_Bermejo@sophia.inria.fr
http://www.inria.fr/mimosa/rp/ROS/