Plan

- Java-RMI
- Les Machines Réactives Distribuées
- Migration de code

Java-RMI

Objets en Java

Rappel :

Le modèle objet ne tient pas compte de la concurrence et du parallélisme

Introduction d'un modèle de thread

Interférence entre les notions

Proposition de modèles différents : acteurs, cubes...

Qu'en est il dans un cadre distribué?

Objets distribués

Objets présents sur des sites distants => parallélisme physique

Pas de mémoire partagée

L'asynchronisme du modèle des threads s'applique directement

Mise en œuvre de mécanismes (plate-forme) rendant transparente la distribution

- Référence distance
- Sérialisation et migration

Objets distribués

Mécanisme de distribution en Java :

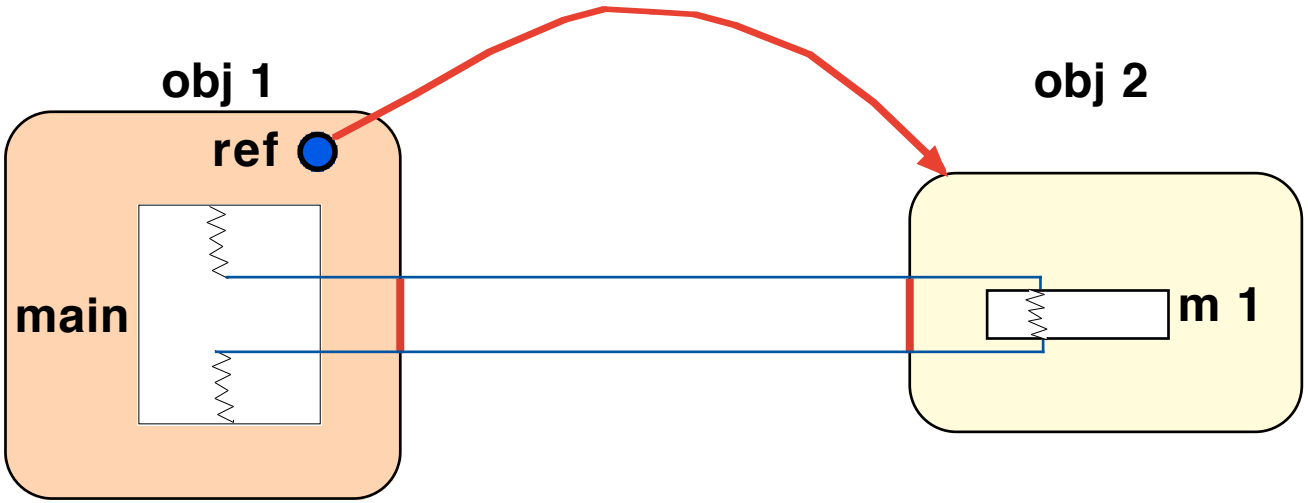
- **JavaRMI : standard**
- **JavaIDL : programmation sur ORB CORBA (modèle différent du modèle objet Java)**

JavaRMI-IIOP : programmation en RMI capable d'interagir avec un ORB CORBA (via IIOP)

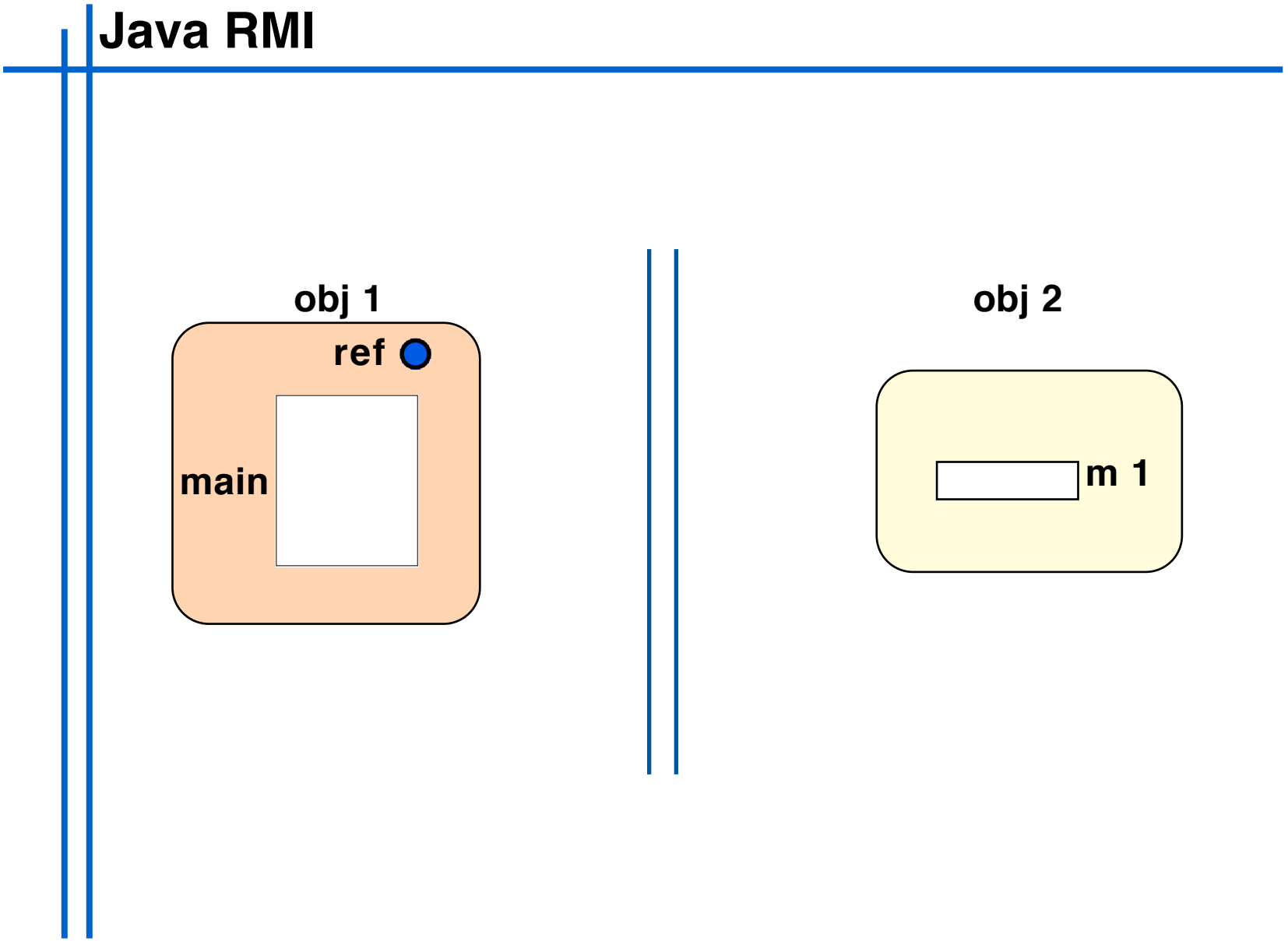
Parallélisme physique => Prise en compte au niveau du modèle objet

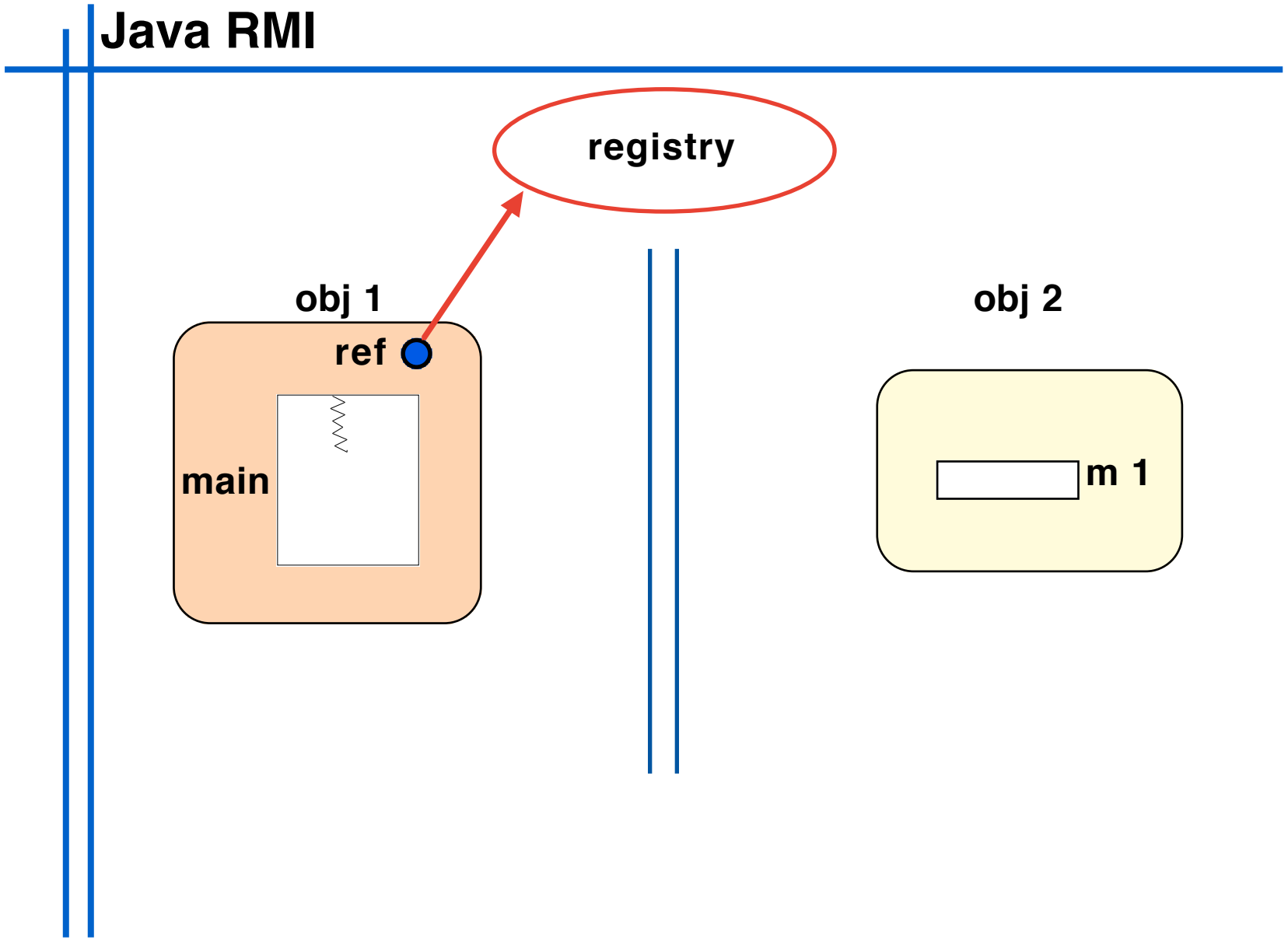
Impact sur les mécanismes de communication entre objets

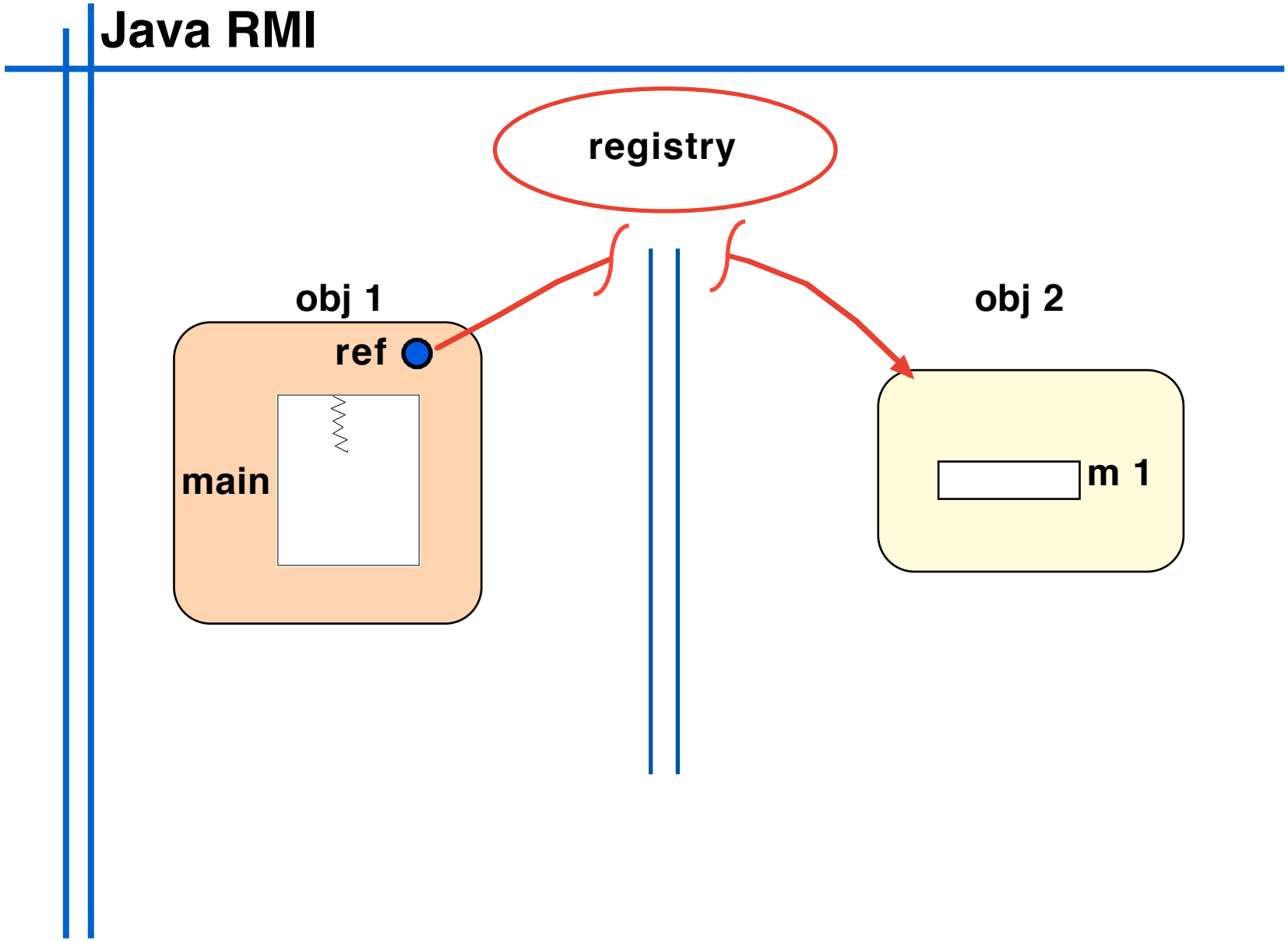
Java RMI



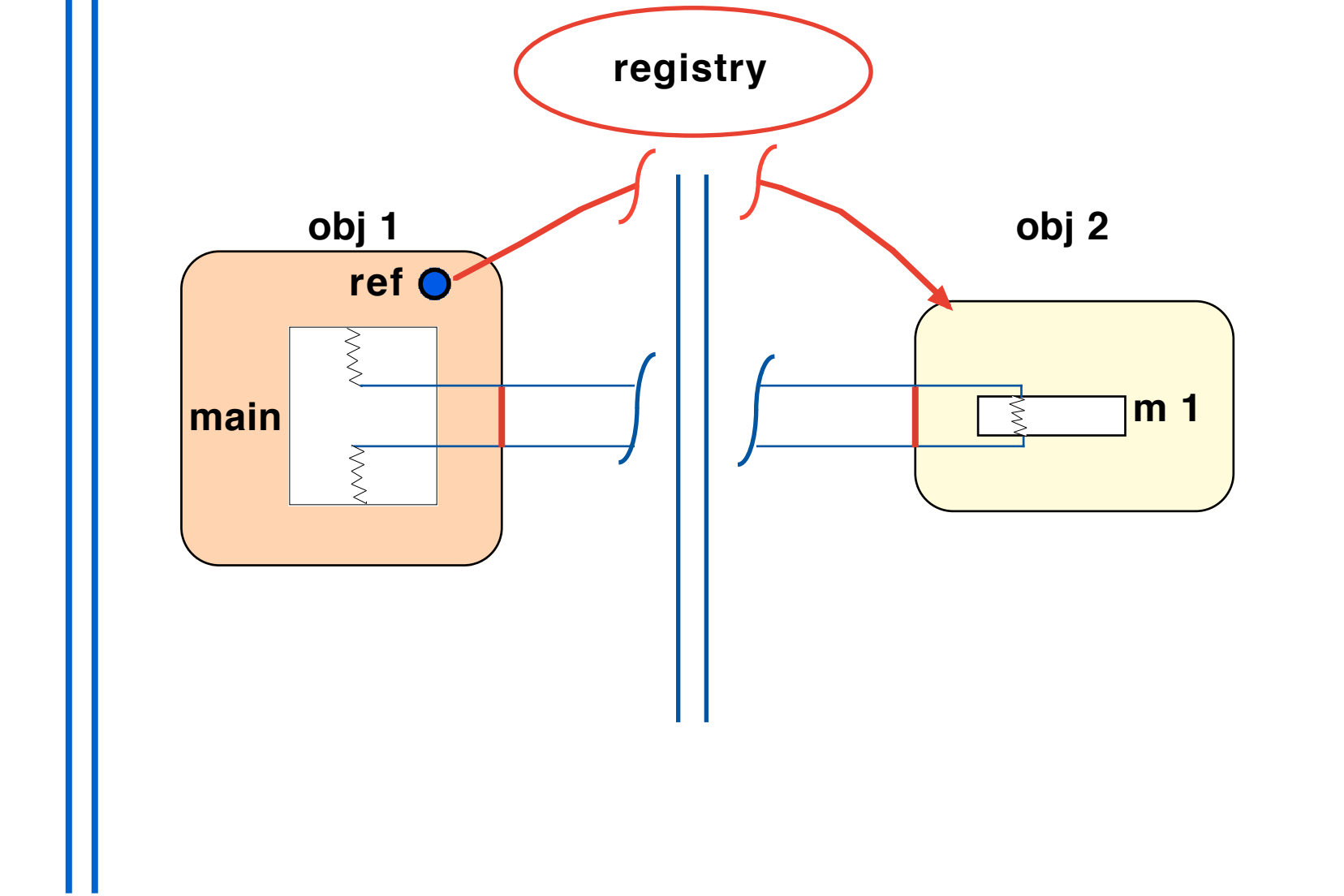
Java RMI




Java RMI
registry
obj 1
ref
main
obj 2
m 1

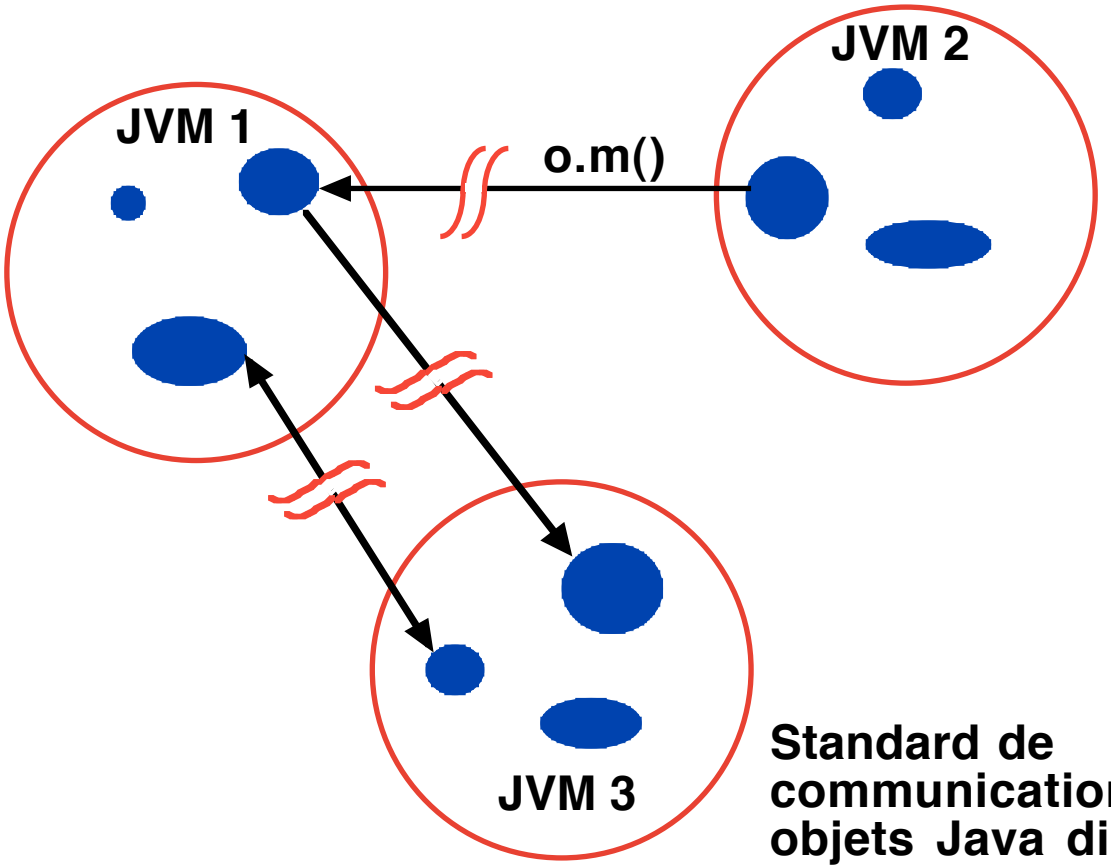

Java RMI
registry
obj 1
ref
main
obj 2
m 1

Java RMI



Java RMI

Remote Method Invocations



Standard de communication entre objets Java distants

Java RMI

- Architecture d'objets en mode client-serveur
- Communication par invocation de méthode synchrone type RPC
- Invocation transparente : respect de la sémantique des appels de méthodes
- Manipulation d'objets distants au travers d'Interfaces
- Passage d'arguments par copie des types de base ou des objets sérialisables ; sinon respect du passage par référence si référence distante
- Permet l'implémentation de politiques de sécurité personnalisées

Java RMI

- Notion de persistance à travers les mécanismes d'objets activables à la demande (rmid).

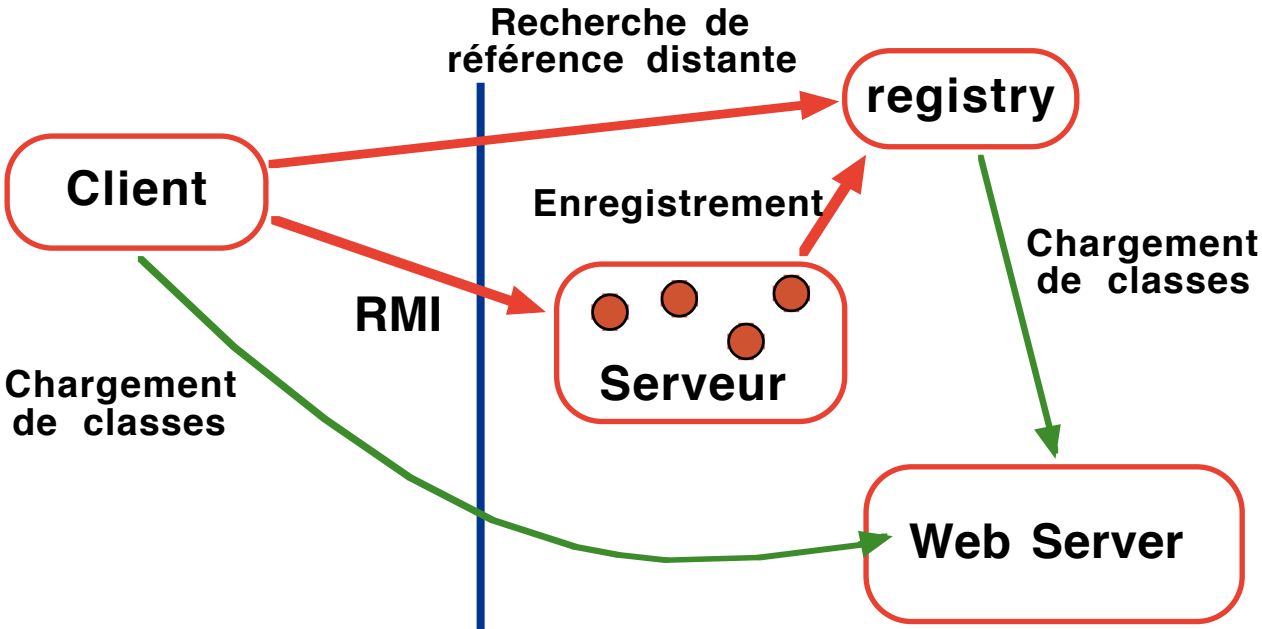
- Tunneling http

- Class loader dédié au téléchargement de code migrant ; utilise un serveur http pour gérer le téléchargement de code

- Interopérabilité avec les ORB via IIOP.

- Custom Socket Factory permettant la personnalisation des communication (encription des transmissions, connexions multiples, ...).

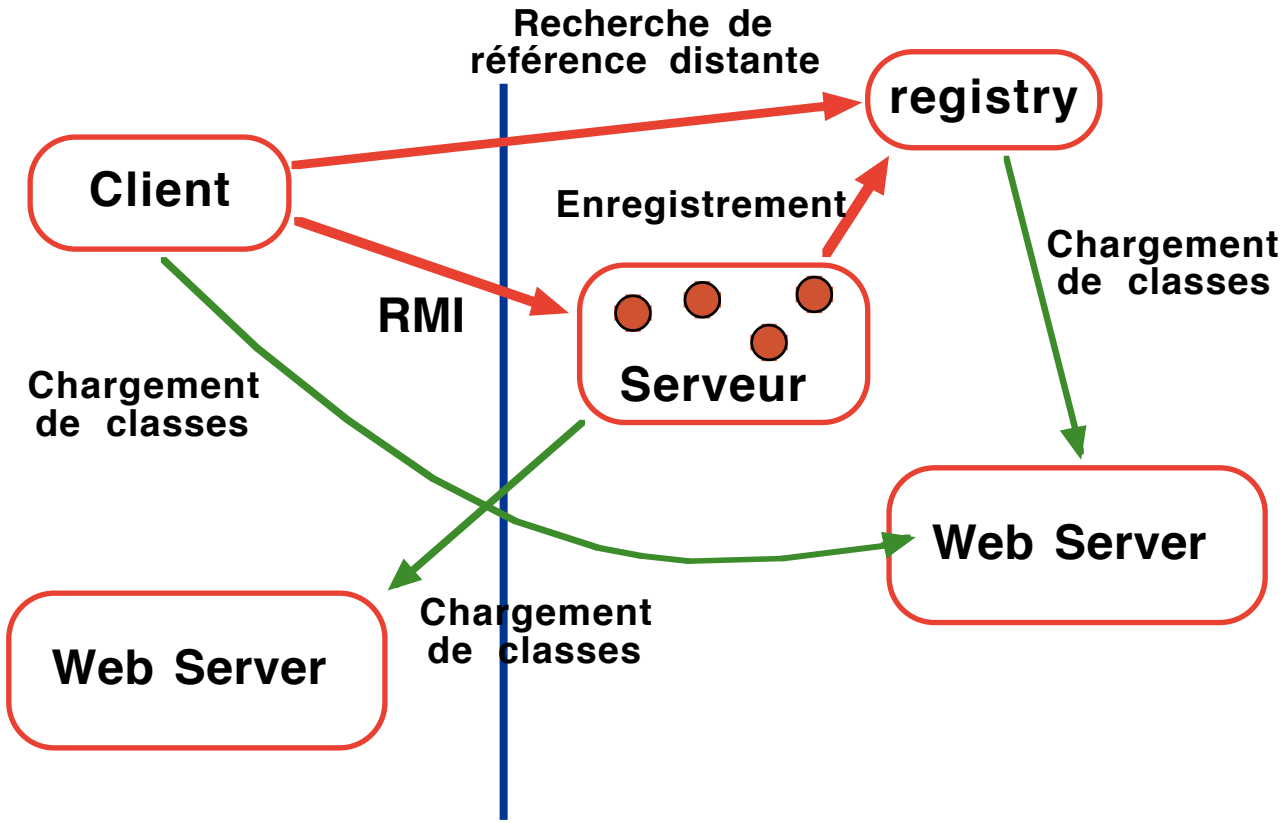
RMI : Application distribuée



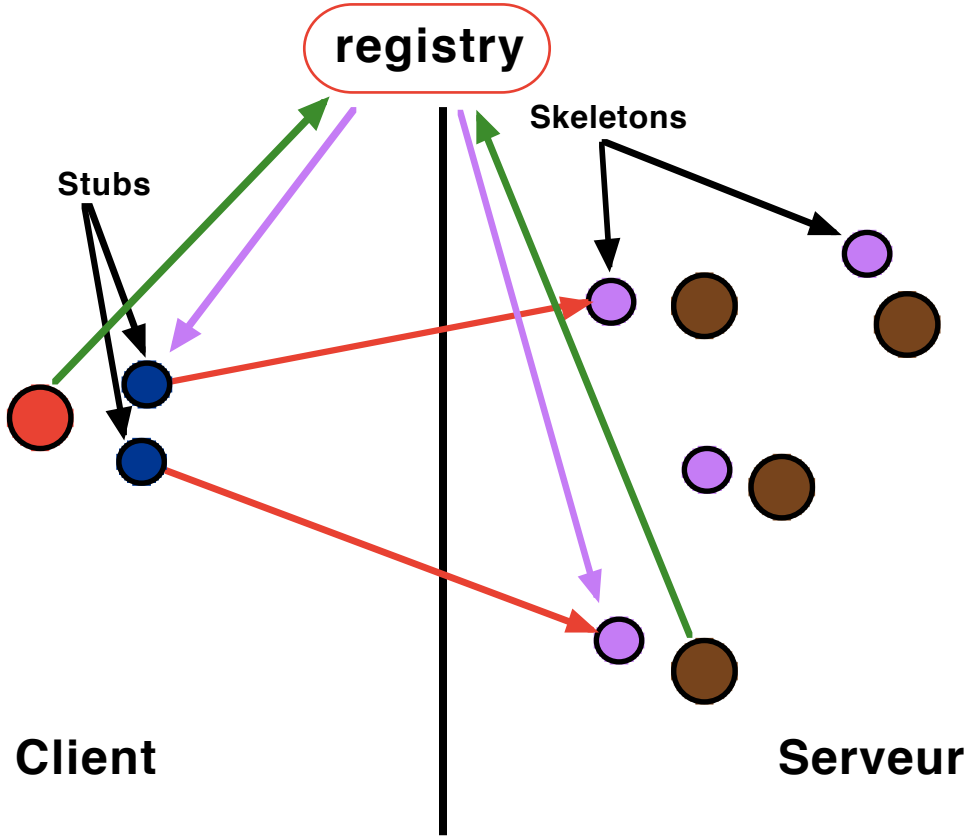
Services :

- recherche de références distantes
- chargement dynamique de bytecode
- communication par invocation de méthodes

RMI : Application distribuée



Références distantes



Serveur

On définit une interface pour le serveur :

```
import  java.rmi.*;
import  inria.meije.rc.sugarcubes.*;

public  interface  RemoteMachine  extends  Remote
{
   public  void  addProgram(Program  program)  throws  RemoteException;
}
```

L'interface doit étendre Remote

Exception levée en cas de problème durant l'invocation de la méthode distante

Implémentation du serveur :

```
public  class  RemoteMachineImpl  extends
java.rmi.server.UnicastRemoteObject  implements  RemoteMachine
{
      protected  Machine  itsMachine  =  SC.machine();
...
     public  void  addProgram(Program  program)  throws  RemoteException{
           itsMachine.addProgram(program);
     }
}
```

Implémente les services de bases d'un serveur RMI

Un Programme est un objet sérialisable

Génération des Stubs et des Skeletons

Les références distantes sont implémentées par des objets, servant à masquer le caractère distant d'un objet-serveur.

Un couple d'objets Stub/Skeleton est utilisé. Leur génération est automatisée par rmic à partir de l'implémentation de l'objet-serveur :

```
rmic  RemoteMachineImpl
```

```
RemoteMachineImpl_Skel.class

RemoteMachineImpl_Stub.class
```

Serveur

Mise en route du serveur :

n° de port standard du registry RMI

Enregistrement du serveur dans le registry

Création du Skeleton correspondant sur le serveur

```
try{
     if(1099  ==  portNumber){
          Naming.rebind("//"+hostName+"/"+name,server);
     }
     else{
               Naming.rebind("//"+hostName+portNumber+"/"+name,server);
     }
}
  catch(Exception e){
         System.out.println("RsiServer  error:"+e.getMessage());
        e.printStackTrace();
}
```

Client

On recherche une référence distante au serveur :

Création du Stub local

```
try{
      if(1099 == portNumber){
          target = (RemoteMachine)Naming.lookup("//"+hostName+"/"
                                                +targetName);
     }
      else{
             target = (RemoteMachine)Naming.lookup("//"+hostName
                                                +portNumber+"/"+targetName);
     }
}
  catch(Exception e){
        System.err.println("Client Error:"+e.getMessage());
       e.printStackTrace();
}
```

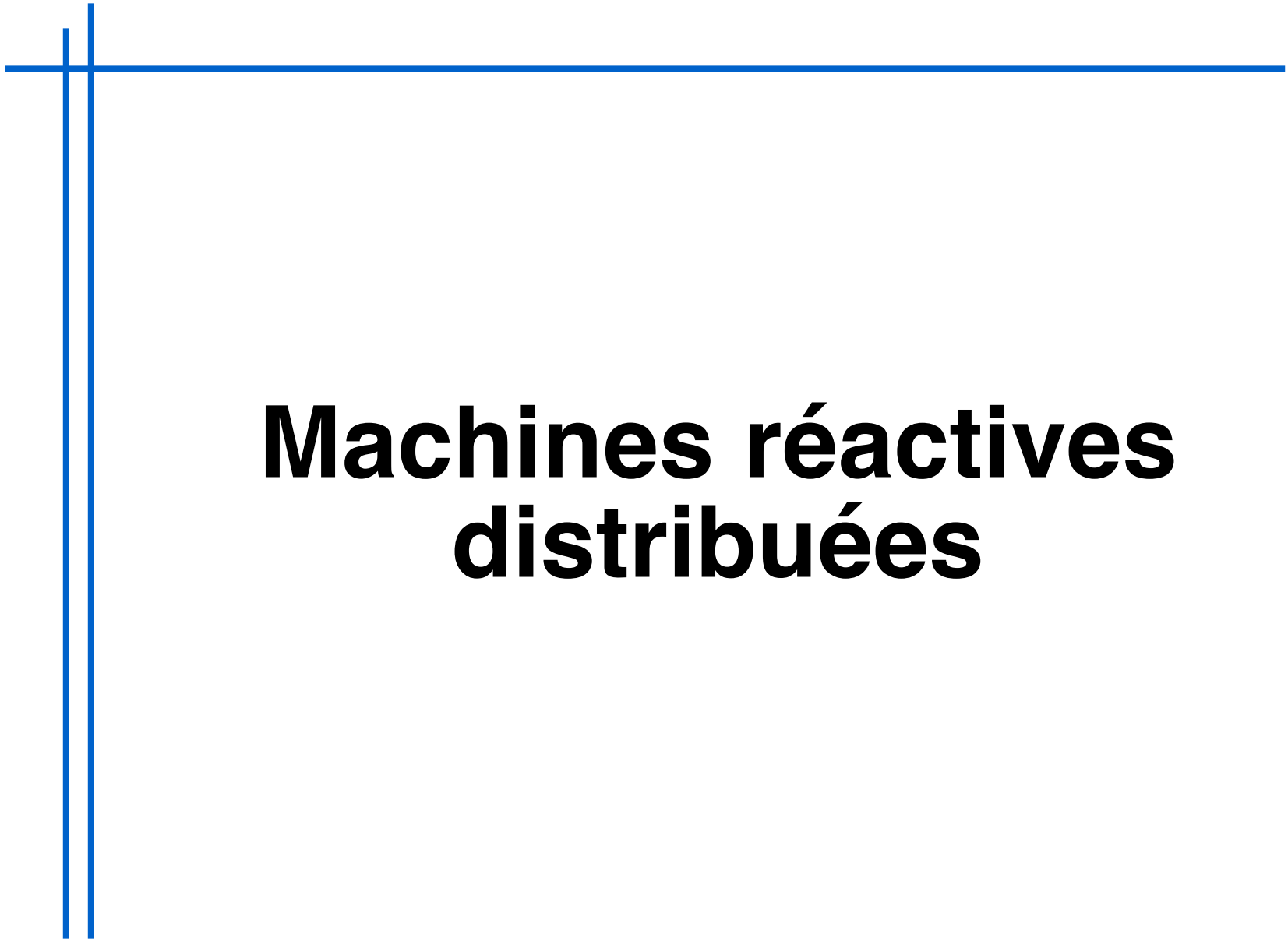
Utilisation de la référence :

```
try{
    Program p = SC.seq(SC.await("e"),SC.print("hello"));
   target.addProgram(p);
} catch(Exception e){ e.printStackTrace(); }
```

Sérialisation

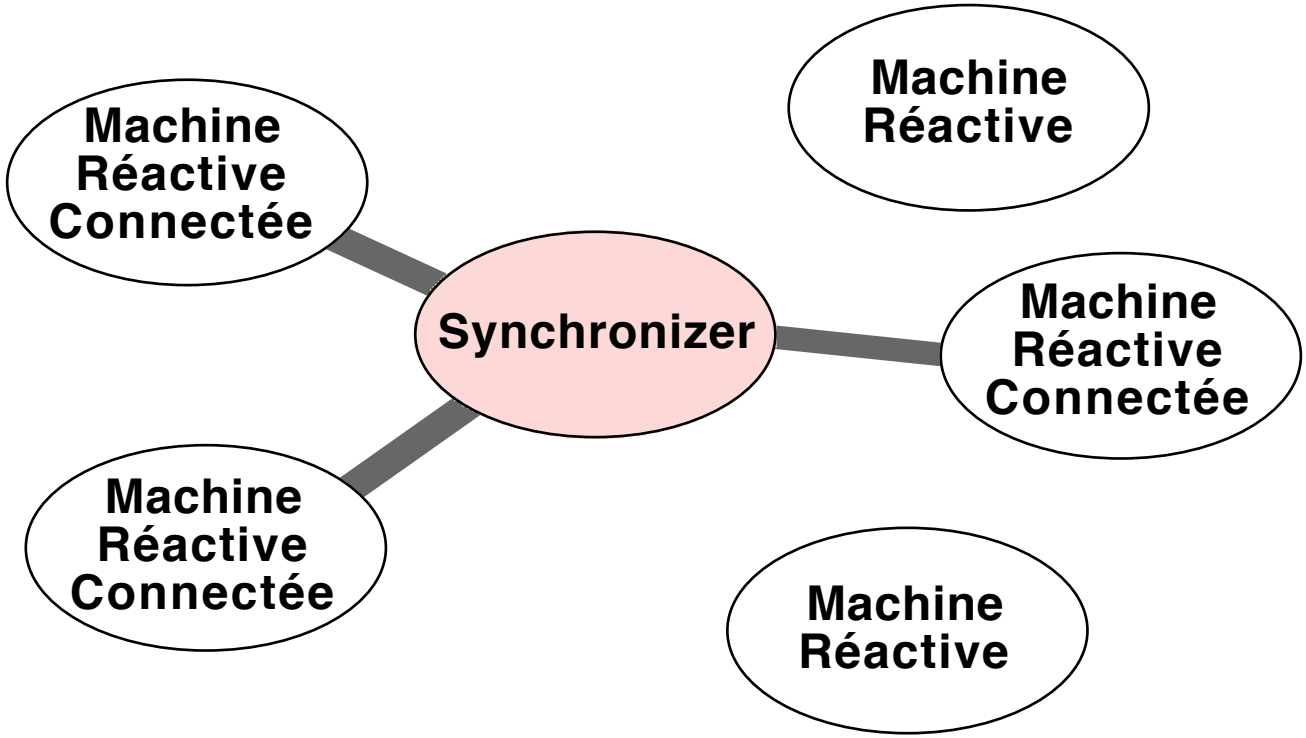
- Mécanisme transformant un objet Java en un flux d'octets
- Un objet est sérialisé avec l'ensemble des objets qu'il référence
- Le mot clé `transient` permet de limiter l'effet de la sérialisation
- Les types primitifs du langage sont sérialisables ainsi que la plupart des objets de l'API (ex : composants AWT)

Les instructions Junior ou SugarCubes sont sérialisables



Machines réactives distribuées

Machines Réactives Synchronisées

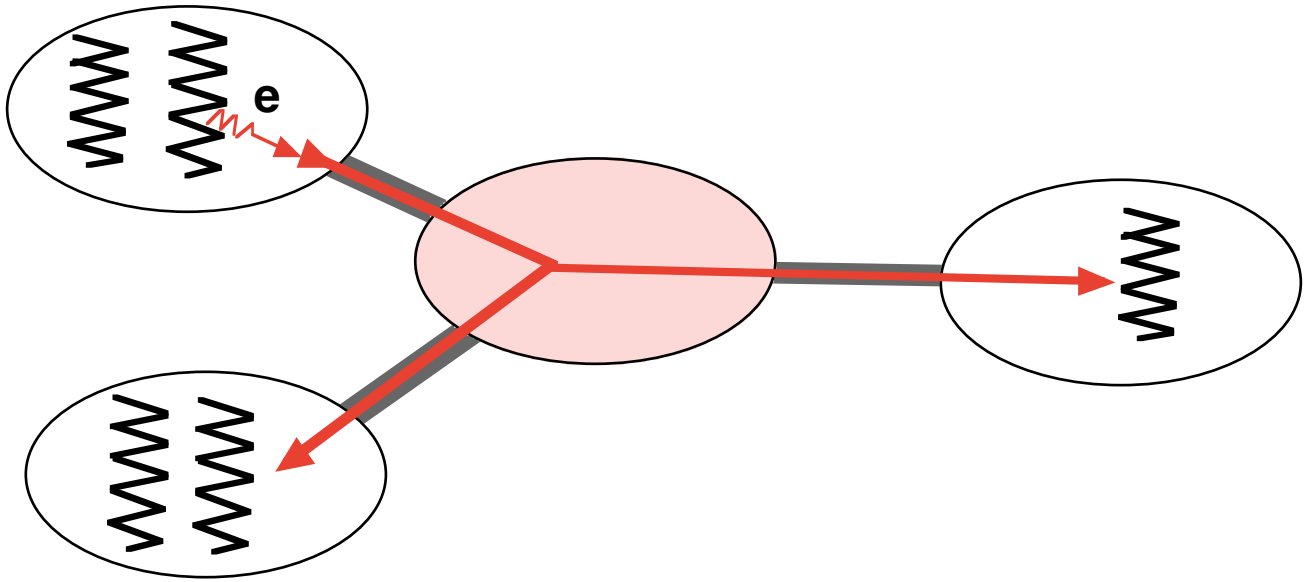


Zone d'exécution réactive distribuée

Intants partagés à travers le réseau

Connexion et déconnexion dynamique

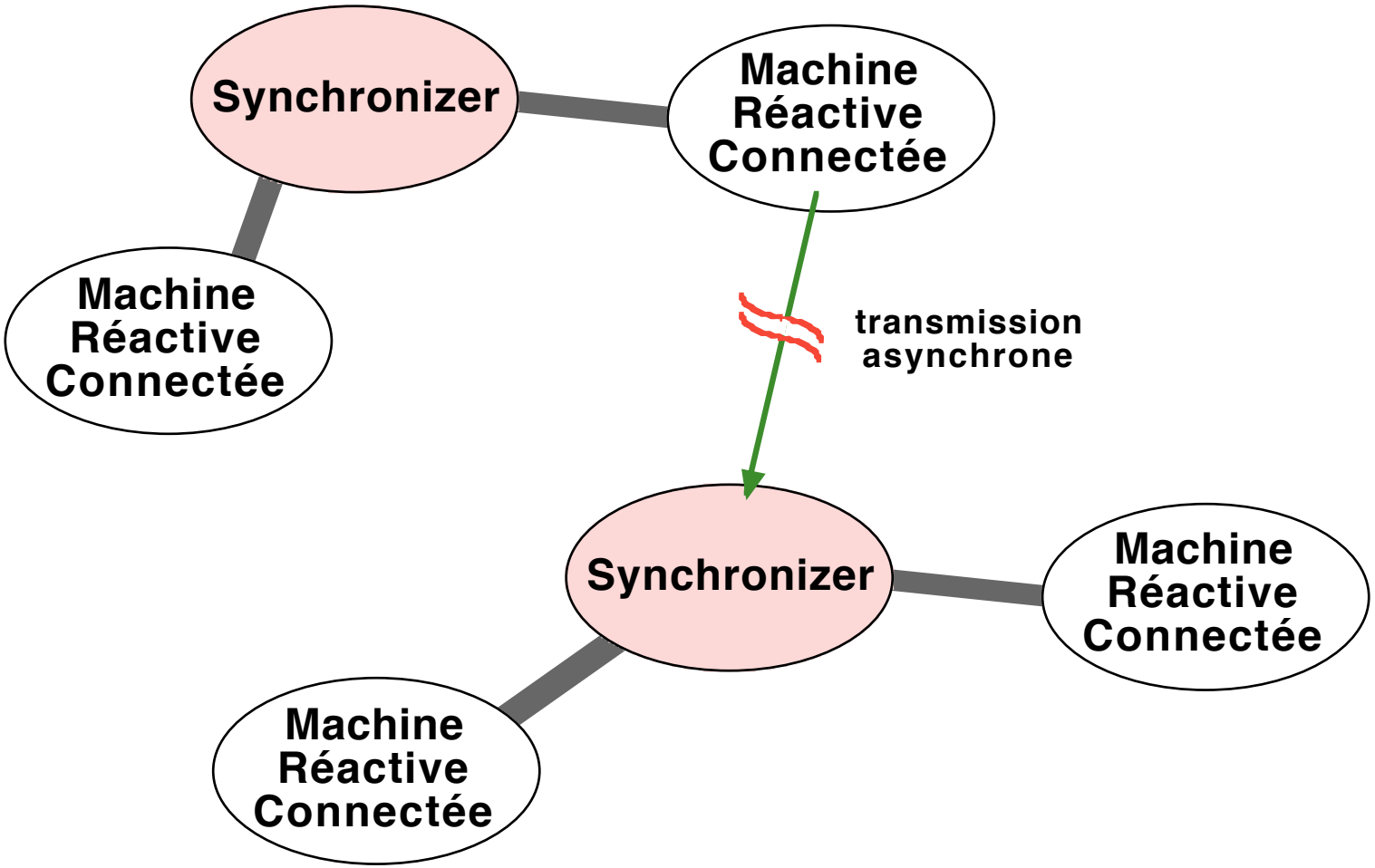
Diffusion d'événements à travers le réseau



Le synchroniseur diffuse les informations de génération

L'information sur l'absence de tous les événements non générés au cours d'un instant est factorisée avec la notification de fin d'instant

Diffusion asynchrone



Implémentation

synchroniseur : algorithme de terminaison des instants distribuée.

Deux algos proposés :

- **Two phase Comit**
- **à base de compteurs**

Deux plates-formes de distribution :

- **RMI**
- **Jonathan**

Synchronizer

```
public interface Synchronizer extends Remote
{
   public int connect(MachineSync mach) throws RemoteException;
   public void disconnect(int id)       throws RemoteException;

   public void broadcast(String event)  throws RemoteException;

   public void suspended(int id)        throws RemoteException;
   public void completed(int id)        throws RemoteException;
}
```

connexion et déconnexion dynamique de machine réactive

identificateur de la machine

génération d'événement à travers le réseau

la machine est suspendue

la machine a terminé pour l'instant courant

Synchronizer

```
public interface Synchronizer_Impl extends Remote
{
   int numberOfMachines = 0;
   MachineSync machine[] = new MachineSync[MaxMachineNumber];
   byte status[] = new byte[MaxMachineNumber];

   Vector broadcastDemands = new Vector();
   Vector broadcastSum = new Vector();
}
```

nb de machines connectées

tableaux des machines et leur statut d'exécution

Vecteur des événements à diffuser au cours d'une phase

Toutes les demandes de diffusion depuis le début de l'instant

Diffusion des événements

```
public void broadcastProcessing(){
   for(int i = 0; i < MaxMachineNumber; i++){
      if(status[i]!=disconnected && status[i] != completed){
        status[i] = undef;
        try{ machines[i].generate(broadcastDemands);}
        catch(Exception e){...}
     }
   }
  broadcastDemands.removeAllElements();
}
```

Synchronizer

Résolution d'un instant distribué

```
synchronized void step(){
   for(int i = 0; i < MaxMachineNumber;i++){
      if(status[i] == undef) return;
   }
   if(broadcastDemands.size() > 0){
      broadcastProcessing();
      return;
   }
   broadcastSum.removeAllElements();
   InstantIsOver();
}
```

Toutes les machines ont retourné leur statut

Il n'y a plus rien à diffuser

fin de l'instant global

Machine d'exécution distribuée

```
public interface MachineSync extends Remote
{
   public void instantIsOver()                throws RemoteException;
   public void generate(Vector eventList)  throws RemoteException;
}
```

Génération locale des événements diffusés

```
public void generate(Vector eventList){
   while(eventList.size() > 0){
       generate((String)eventList.firstElement());
       eventList.removeElementAt(0);
   }
   synchronized(this){ move = true; notifyAll(); }
}
```

Nouvelles instructions réactives

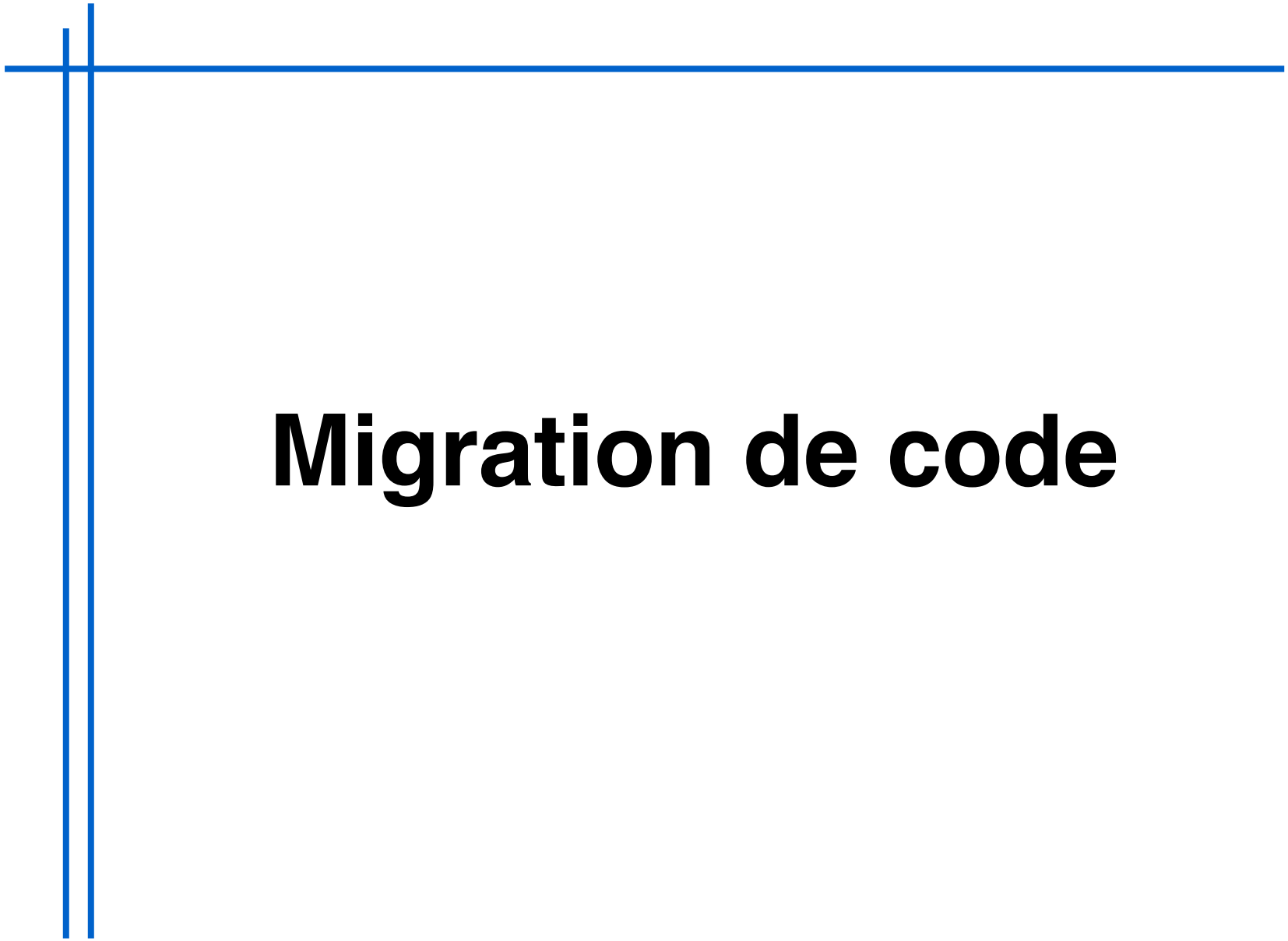
Trois nouvelles instructions réactives ont été introduites :

- **Connect** permet de demander une connexion à un synchroniseur et attend la fin d'instant.
- **Disconnect** permet de demander une déconnexion et attend la fin d'instant
- **Broadcast** permet de générer des événements diffusés à travers un synchroniseur à l'ensemble des machines connectées

Conclusion

Un programme réactif peut-être physiquement distribué et conserver une notion d'instant global.

Les zones réactives distribuées peuvent être dynamiquement reconfigurées afin de ne pas trop pénaliser les modules les plus autonomes.



Migration de code

Migration de code

migration d'une activité : "migration de thread logique"

migration objective, migration subjective

Pb : définir un état cohérent pour pouvoir migrer

Utilisation de la notion d'instant :

Fin d'instant = état stable

Notion de gel de programme réactif : instruction Freezable

Étapes de la migration :

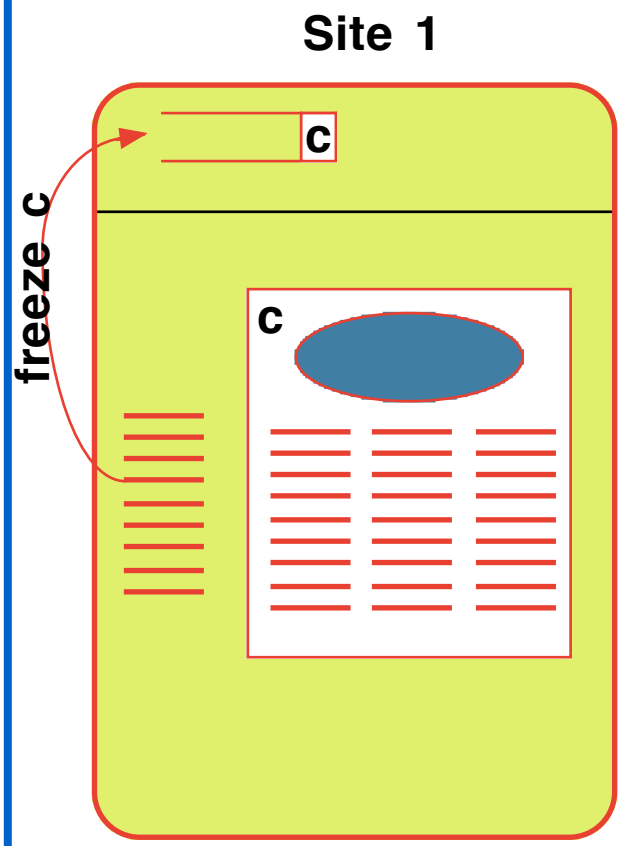
- Calcul de résidu du programme gelé à la fin d'un instant
- Sérialisation du résidu
- Envoie asynchrone à travers le réseau
- Ajout du programme migré dans la machine cible

Les Cubes

migration des Cubes :

- Un cube est une instruction Freezable
- Instruction Freeze : demande un gel d'instruction
- Calcul du résidu inter instant ; ajout dans l'environnement (instructions gelées)
- Notification de gel : propagée le long de l'arbre de programme
- Mise à profit de la structure hiérarchique
- Sérialisation des instructions
- Notification de réveil d'instruction propagée le long de l'arbre de programme : dynamic binding

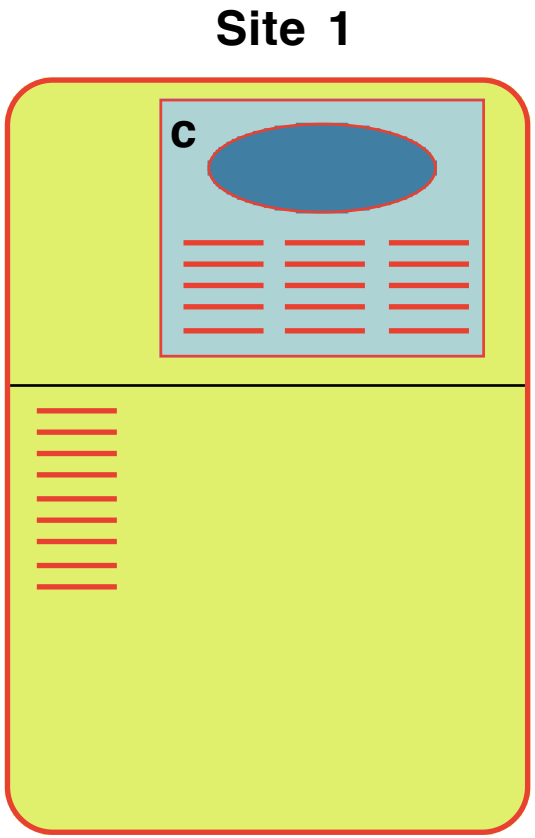
Migration des Cubes



Une requête de gel est émise

Elle est traitée à la fin de l'instant

Migration des Cubes

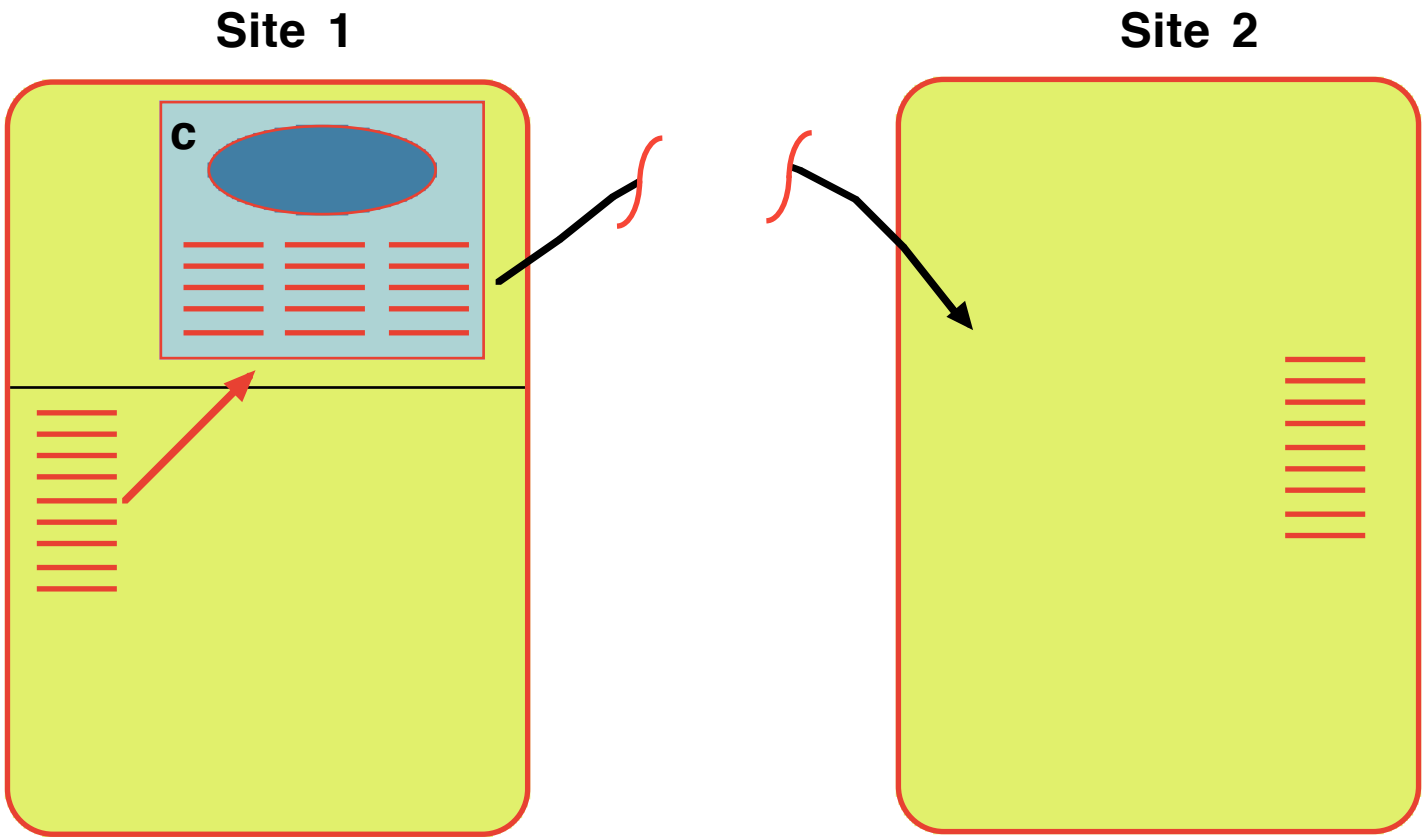


Une requête de gel est émise

Elle est traitée à la fin de l'instant

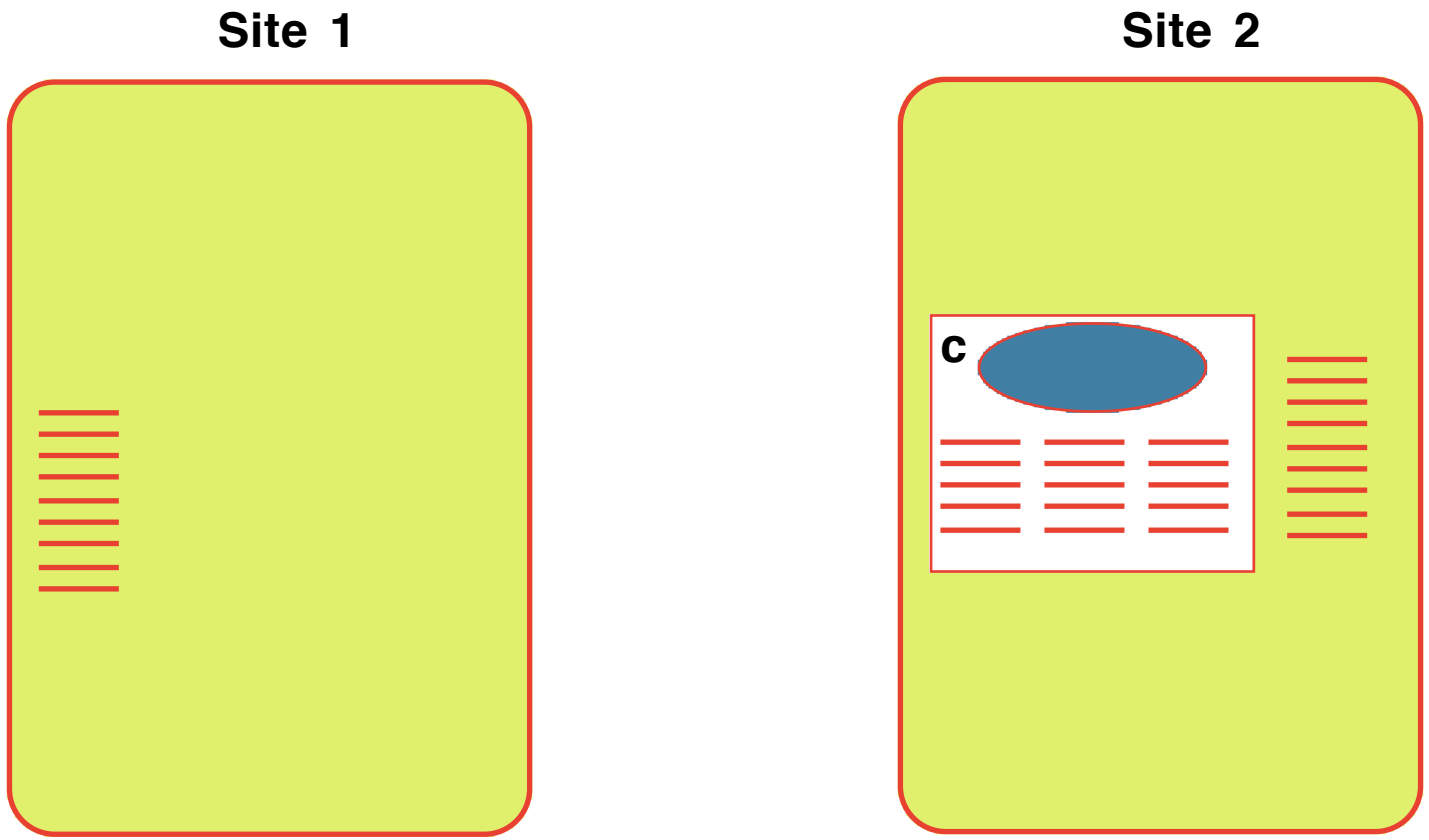
Migration des Cubes

Migration asynchrone à l'instant suivant



Migration des Cubes

Le Cube poursuit son exécution sur le site distant



Implémentation

```
static Program icobj(String name,DemoMainFrame f){
    Icobj i = new Icobj(name,200,200);
    return SC.cube(new JavaStringValue(name)
                        ,new JavaObjectValue(i)
                ,SC.seq(SC.action(new JavaAddIcobj())
                    ,SC.merge(migration(name)
                                ,SC.merge(sin(),cos())))
                 ,new JavaRemoveIcobj()
                 ,new JavaRemoveIcobj()
                ,new JavaAddIcobj());
}

static Program migration(String name){
    return SC.loop(
                SC.seq(
                    SC.await("migrate")
                    ,SC.action(new MoveTo())
                    ,SC.freeze(name)
                    ,SC.stop()
                    )
                );
}
```

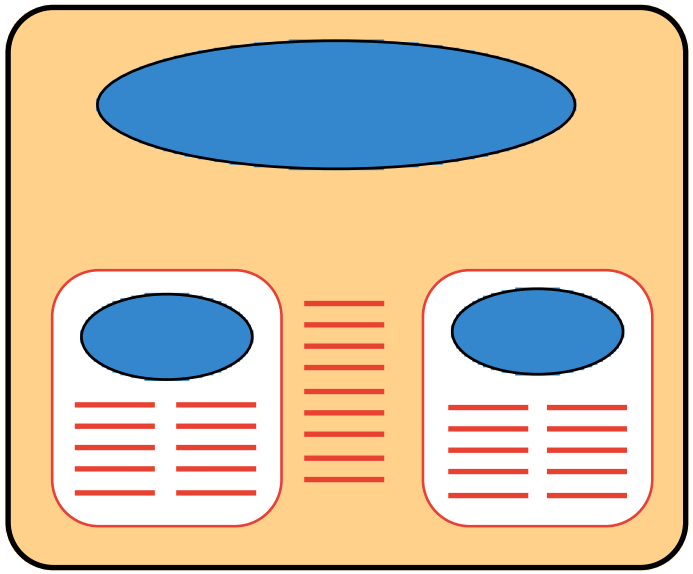
Implémentation

```
public class MoveTo implements JavaAction
{
    public void execute(Object self
                        ,LocalVariables vars
                        ,ReactiveEngine engine){
        String name = engine.getCube().getName();
        String dest = ((Icobj)self).nextTarget();
       engine.addProgram(SC.seq(SC.stop()
                    ,SC.action(
                        new RmiSynchronousSender(
                          ((DemoMachine)((Cube)engine.getMachine())
                              .getJavaObject()).registryHostName
                         , dest
                         , name)
                    )
                )
            );
    }
}
```

Construction hiérarchique de Cubes

Un cube peut contenir plusieurs autre cubes

Utilisation des constructions hiérarchiques des Cubes pour implémenter un mécanisme de migration de groupe



- Notification Java propagée

au Cubes imbriqués

- Machines réactive = cubes

particuliers

La synchronisation des Cubes imbriqués est conservées après migration