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 :

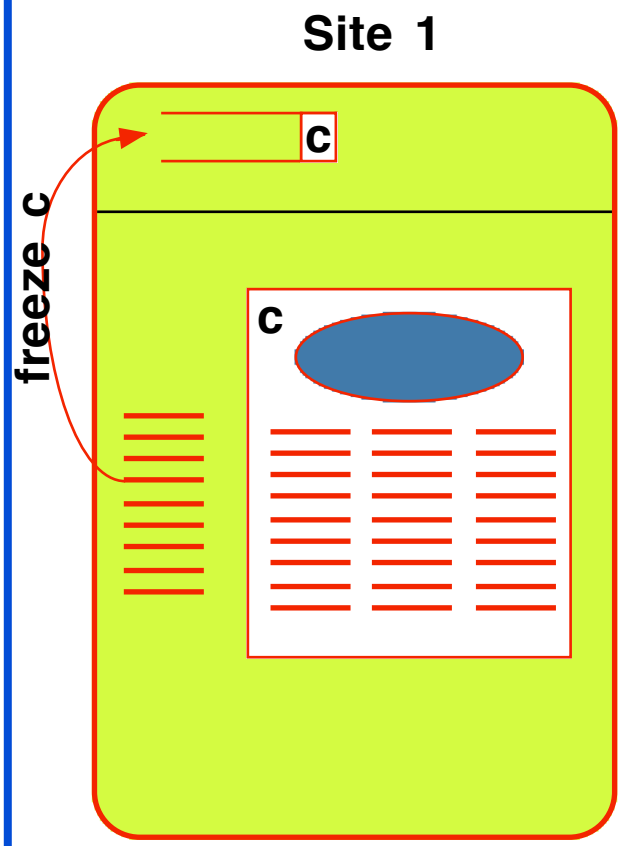
- Calcul de résidu du programme gelé à la fin d'un instant
- Sérialisation du résidu
- Envoie asynchrone à travers le réseau
- Ajout du programme migré dans la machine cible

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

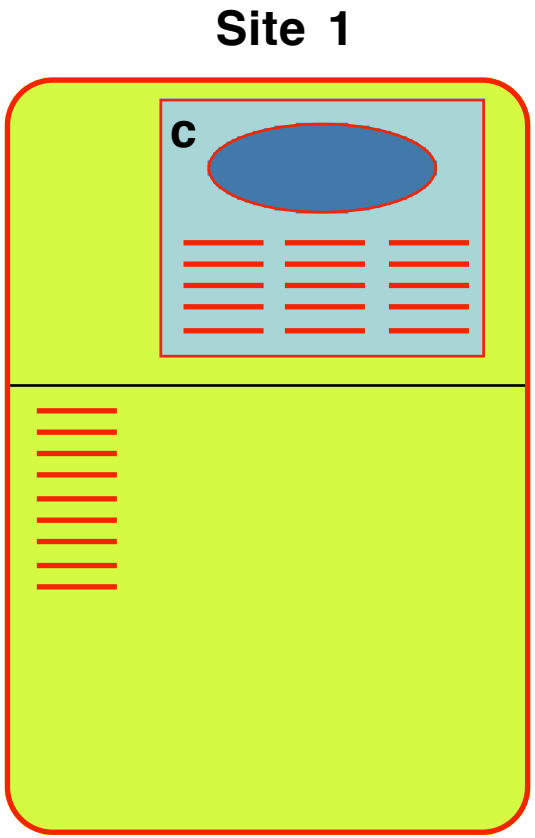
Migration des Cubes



Une requête de gel est émise

Elle est traitée à la fin de l'instant

Migration des Cubes

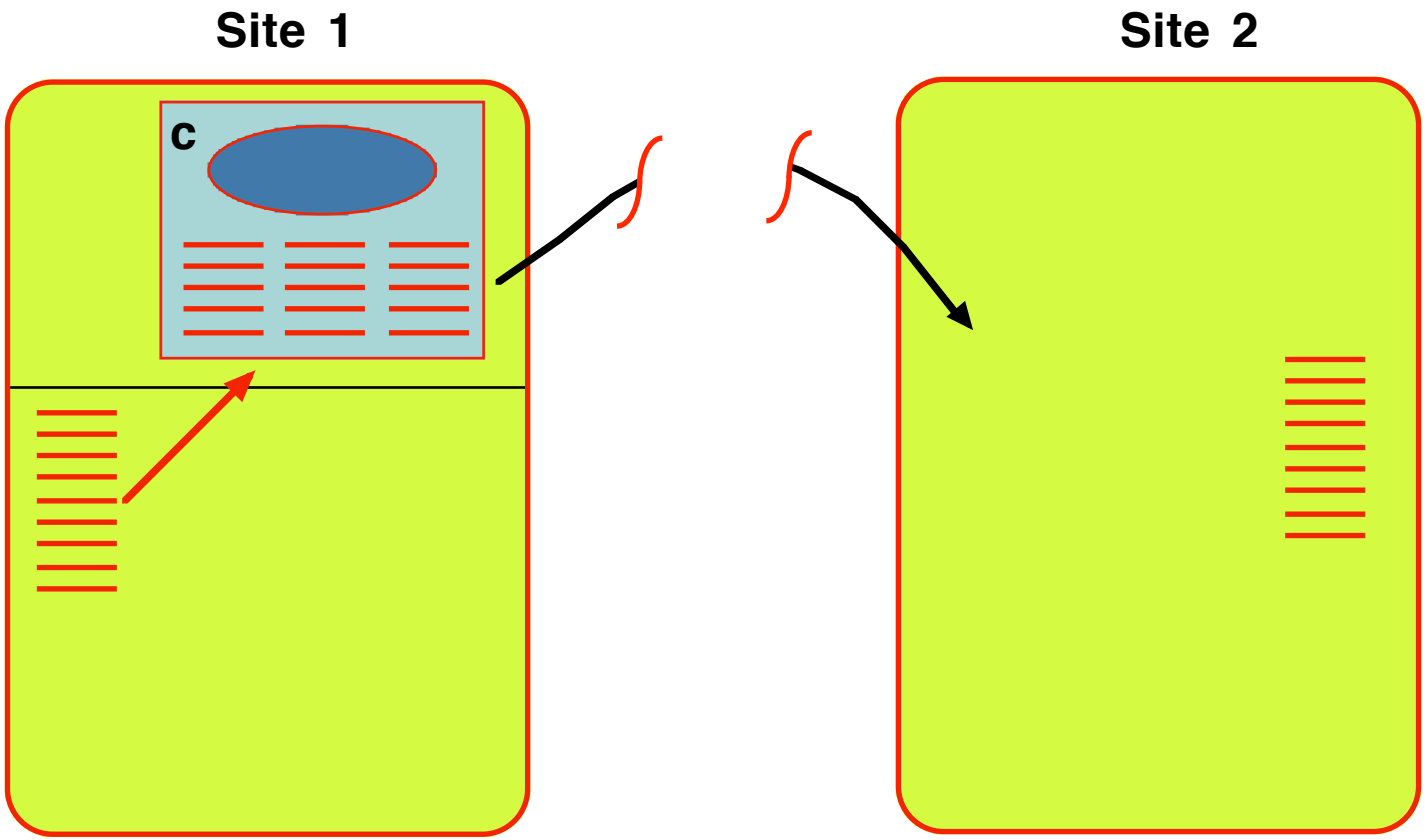


Une requête de gel est émise

Elle est traitée à la fin de l'instant

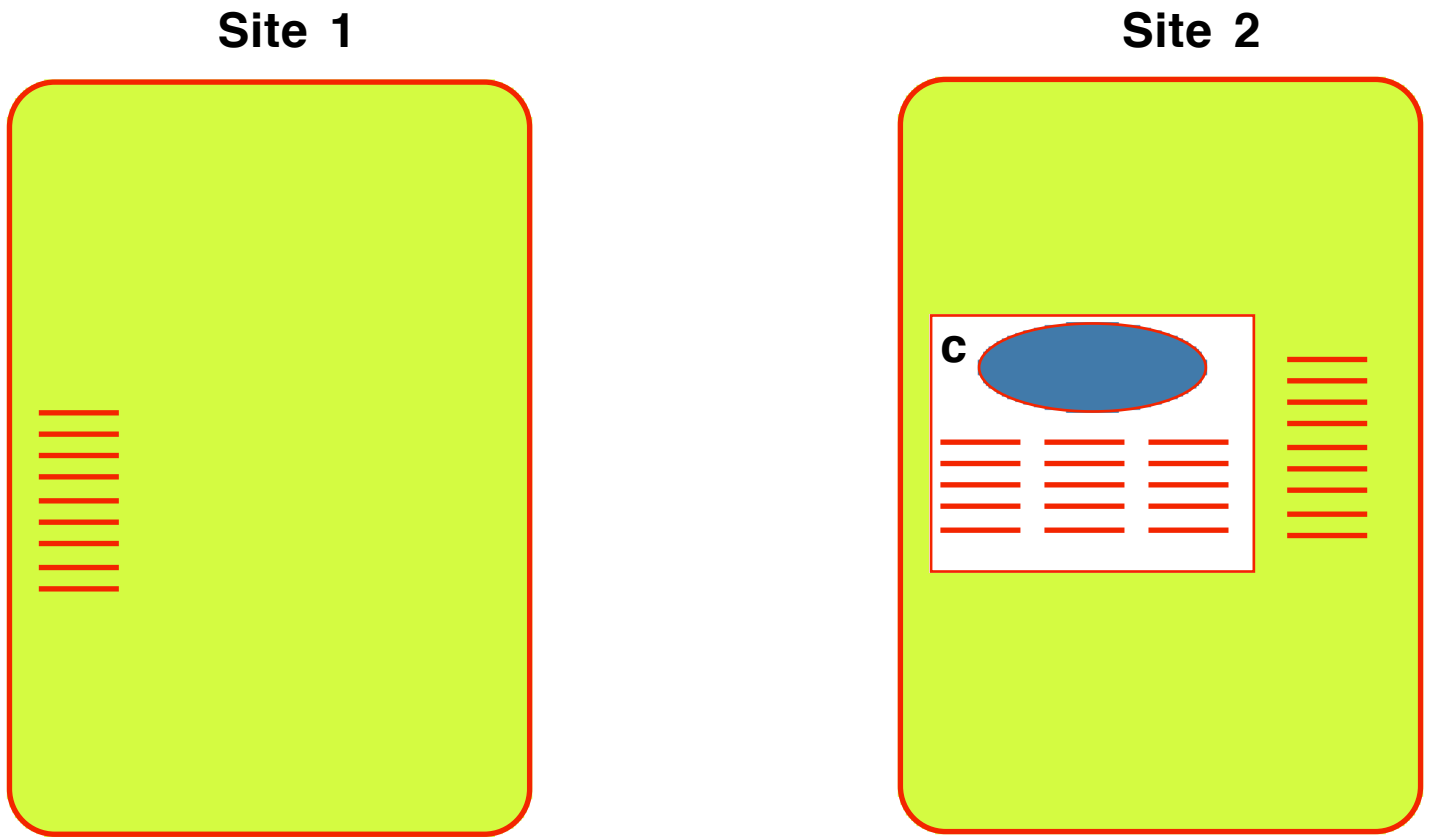
Migration des Cubes

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()
                    )
                );
}
```

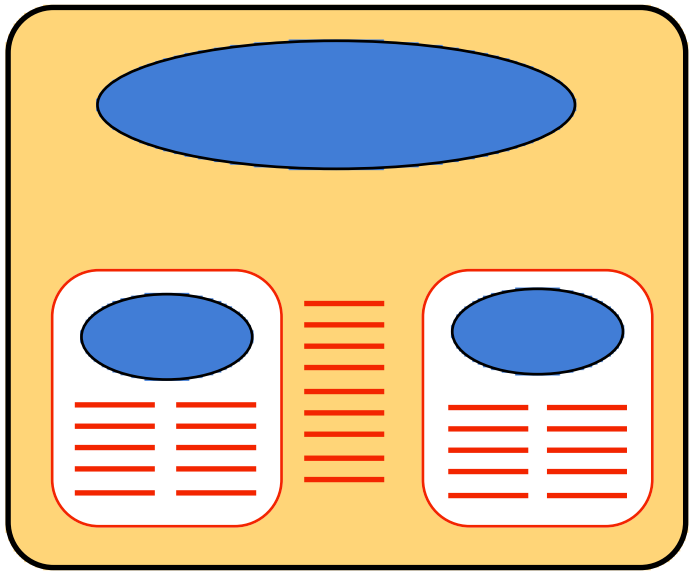
Implémentation

```
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

Conclusion

Les SugarCubes :

- **framework Java pour exprimer concurrence ordonnancement contrôle et communication d'entités parallèles.**

- **Modèle objet des Cubes, basé sur la notion d'acteurs interagissant par diffusion instantanée d'événements. Encapsulation structure de donnée, méthodes de traitement et fil de contrôle d'exécution.**

Cubes et migration de Code :

- **notion d'instant => État stable d'un système + sérialisation -> migration**

- **Cubes et structure hiérarchique => migration de groupe**

- **Protocoles de "dynamic binding" basés sur les événements diffusés**