

Adaptation de Composants (et Agents) Répartis

Module ALADYN 2004 (2004-2005)
(Architectures Logicielles pour l'Auto-Adaptabilité)

Jean-Pierre Briot

Thème OASIS
(Objets et Agents pour Systèmes d'Information et Simulation)
Laboratoire d'Informatique de Paris 6
Université Paris 6 - CNRS
Jean-Pierre.Briot@lip6.fr



Jean-Pierre Briot



Module ALADYN 2004

1

Besoins d'Adaptation du Logiciel

- Nouvelles applications informatiques :
 - (informatique nomade, objets communicants, travail coopératif, multimedia, etc.)
- Profonds besoins en matière de dynamicité :
 - dynamicité des services offerts,
 - adaptation à des environnements et contraintes d'exécution évoluant dynamiquement (et éventuellement non prédictibles),
 - » ex : contraintes de débit,
 - » de taille
 - » de sécurité
 - » de robustesse
 - » de ressources
 - » de qualité de service...

Jean-Pierre Briot



Module ALADYN 2004



3

Supports de cours

Copie transparents sur :

Site Web ALADYN

et en :

<http://www-poleia.lip6.fr/~briot/cours/>

Aladyn04-05.pdf

(également autres supports et biblio)

Jean-Pierre Briot



Module ALADYN 2004



2

Problématique de l'adaptation du logiciel

- Adaptation (statique et dynamique) individuelle
 - Réflexion
- Adaptation d'un ensemble de logiciels (recomposition)
 - Composants
- (vers une) Auto-Adaptation logicielle
 - Agents

Jean-Pierre Briot



Module ALADYN 2004



4

Plan

- Rappels sur la réflexion
- L'architecture CodA
 - Objets, Méta-objets
 - Adaptation statique d'un objet
- L'architecture Comet (exemples à programmer en TME)
 - Composants, Rôles, Protocoles
 - Adaptation dynamique d'un composant et d'un ensemble de composants
- Le projet DarX
 - Auto-Adaptation dynamique de la réplication d'agents

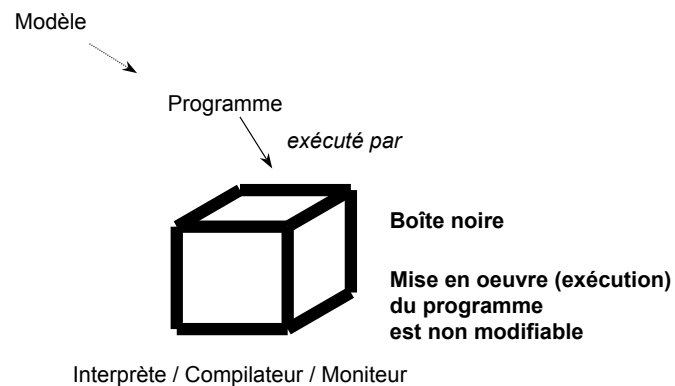


(Retour à un vieux) Dilemne

- Ecrire de BEAUX programmes
 - lisibles
 - concis
 - modulaires
 - abstraits
 - génériques
 - réutilisables
- Ecrire des programmes EFFICACES
 - spécialisés
 - choix optimaux de représentation interne des données
 - contrôle optimisé
 - gestion des ressources adéquate
- DILEMNE : Spécialiser/optimiser des programmes tout en les gardant génériques



Boîte noire

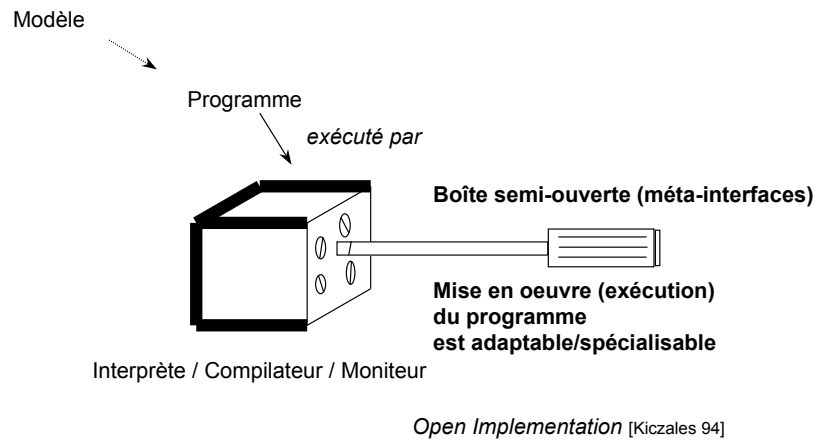


Solutions Ad-Hoc

- Coder "entre" les lignes
 - difficile à comprendre
 - difficile à maintenir (hypothèses cachées)
 - peu réutilisable
- Annotations/Directives (déjà mieux)
 - ex : High Performance Fortran (HPF)
 - mais
 - » notations de plus ou moins bas niveau
 - » ensemble/effet des annotations non extensible/adaptable

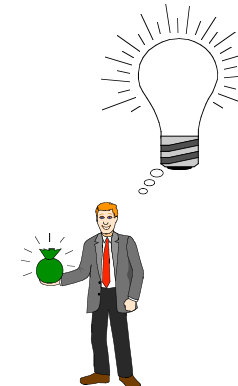


Réflexion (3)



Réflexion

- Le concept de *réflexion* (méta-programmation, architectures réflexives...) offre ainsi un cadre conceptuel permettant un découplage des *fonctionnalités* d'un programme des caractéristiques de sa *mise en œuvre*



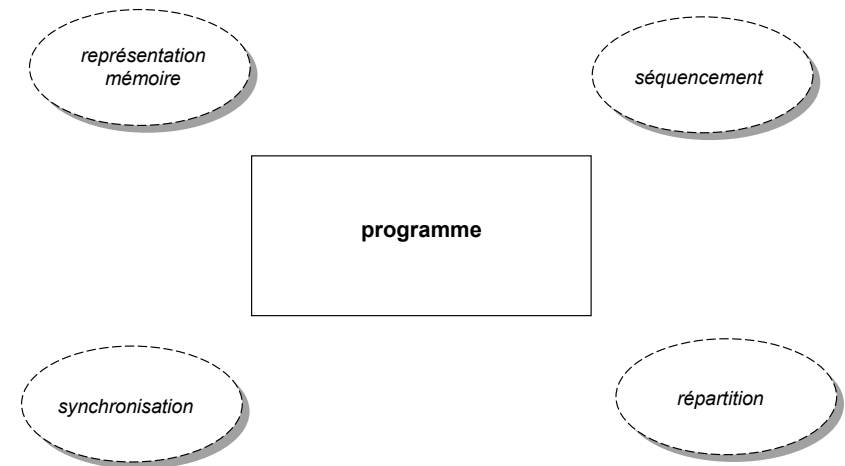
Réflexion (2)

- Diverses caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes (*réifiées*) sous la forme de *méta-programmes*.
 - Habituellement elles sont invisibles et immuables (interprète, compilateur, moniteur d'exécution...)
- La spécialisation de ces méta-programmes permet de *particulariser* (éventuellement dynamiquement) l'exécution d'un programme
 - représentation mémoire
 - modèle de calcul
 - contrôle de concurrence
 - séquencement
 - gestion des ressources
 - protocoles (ex : résistance aux pannes)

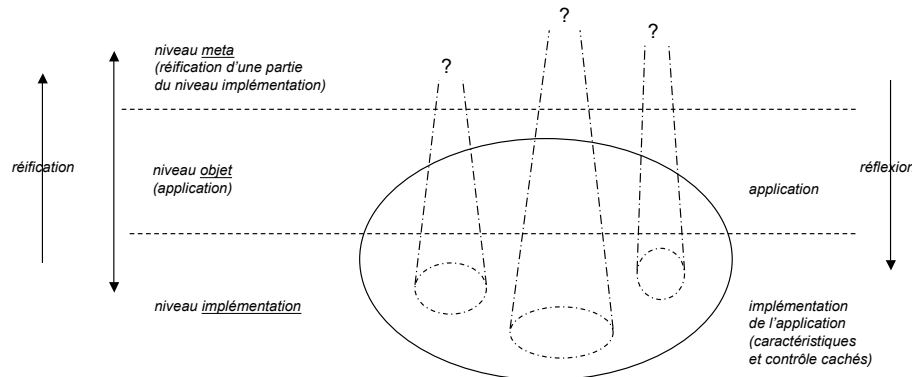
avec le minimum d'impact sur le programme lui-même



Contexte d'exécution



Réification/réflexion



Jean-Pierre Briot



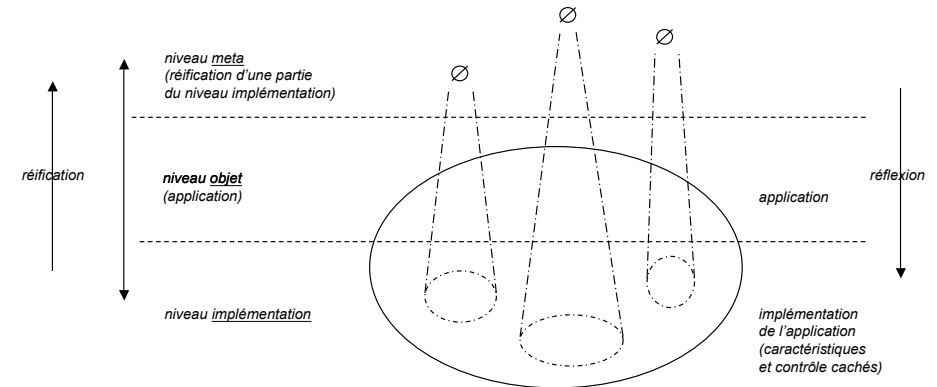
Module ALADYN 2004



13

Réification/réflexion

- Réification numérique (potentiomètres)
 - Ex : Options de compilation d'un compilateur
 - Efficacité vs taille du code généré



Jean-Pierre Briot



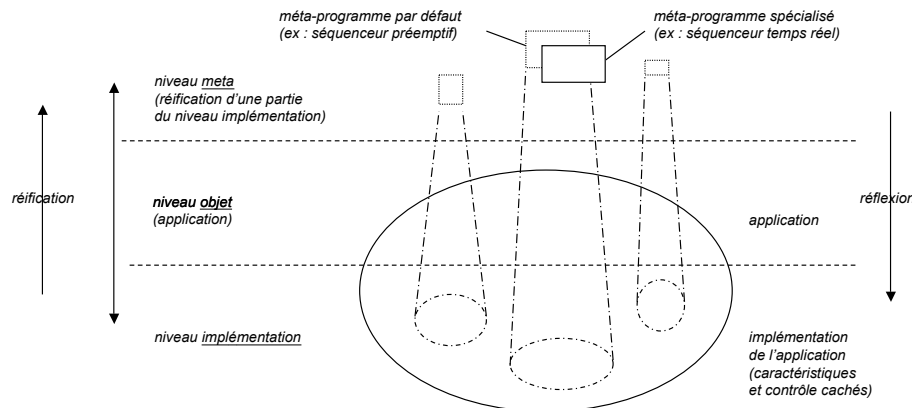
Module ALADYN 2004



14

Réification/réflexion (2)

- Réification logicielle (méta-programmes)
 - Ex : algorithme de séquençement (scheduler)
- plus général/flexible que des potentiomètres*



Jean-Pierre Briot



Module ALADYN 2004



15

Réflexion (4)

- Découplage des *fonctionnalités* d'un programme des caractéristiques de sa *mise en œuvre (exécution)*
- Séparation entre programme ET méta-programme(s) favorise :
 - généricité et réutilisation des programmes
 - et des méta-programmes
- Ex :
 - changer la stratégie de contrôle pour un programme donné
 - réutiliser une stratégie de contrôle en l'appliquant à un autre programme

Jean-Pierre Briot



Module ALADYN 2004



16

Structure / Dynamique

- Structure (représentation)
 - spécialiser la création des données
 - » méthodes de classe (= méthodes de métaclases) en Smalltalk
 - » constructor member functions en C++, en Java
 - spécialiser un gestionnaire de fenêtres
 - » implantation d'une feuille de calcul en Silica [Rao]
 - introspection
 - » représentation d'entités du langage Java (ex : entiers, classes) sous forme d'objets Java
- Dynamique (comportement/exécution)
 - implémenter des coroutines via la manipulation de continuations
 - » call/cc en Scheme
 - spécialiser le traitement d'erreur
 - » doesNotUnderstand: en Smalltalk
 - changer l'ordre de déclenchement de règles de production
 - » méta-règles en NéoOpus



Approche réflexive

- Réflexion permet d'intégrer intimement des (méta-)bibliothèques de contrôle avec un langage/système
- Offre ainsi un cadre d'interface entre approche applicative et approche intégrée
- La réflexion s'exprime particulièrement bien dans un modèle objet
 - modularité des effets
 - encapsulation des niveaux
- méta-objet(s) au niveau d'un seul objet
- méta-objets plus globaux (ressources partagées : séquençement, équilibre de charges...)
 - *group-based reflection* [Watanabe'90]

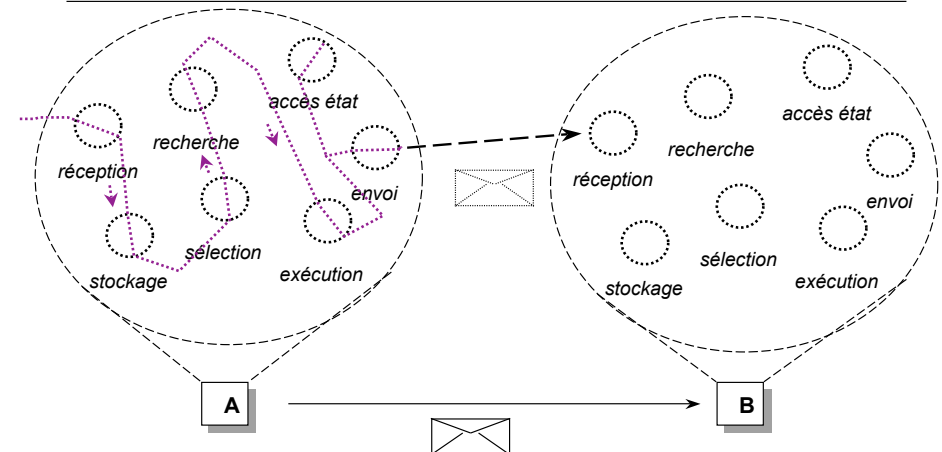


Méta-objets/composants

- *CodA* [McAffer ECOOP'95] est un exemple de modèle relativement général d'architecture réflexive
- Sept méta-objets/composants de base :
 - envoi de message
 - réception de messages
 - stockage des messages reçus
 - sélection du premier message à traiter
 - recherche de méthode correspondant au message
 - exécution de la méthode
 - accès à l'état de l'objet
- Les méta-composants sont :
 - spécialisables
 - (relativement) combinables



CodA

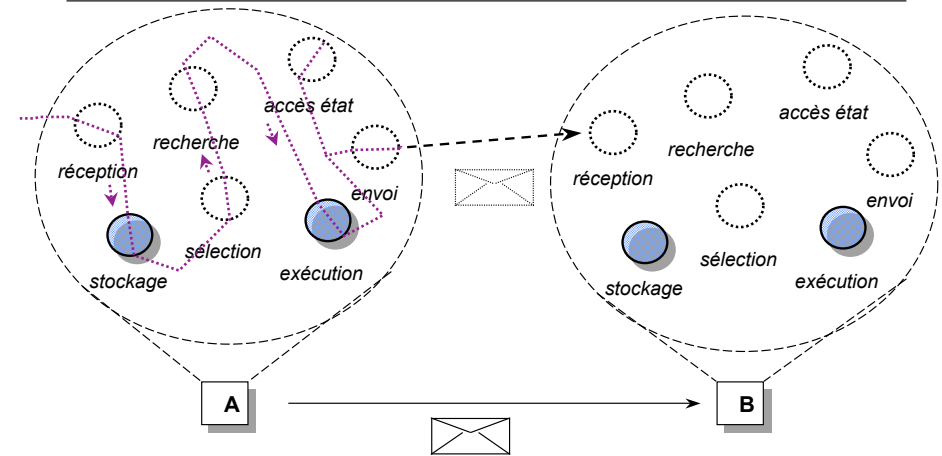


Ex : Exécution concurrente

- envoi de message
- réception de messages
- **stockage des messages reçus**
 - » file d'attente (FIFO)
- sélection du premier message à traiter
- recherche de méthode correspondant au message
- **exécution de la méthode**
 - » processus associé
 - » boucle infinie de sélection et traitement du premier message
- accès à l'état de l'objet



Exécution concurrente (2)

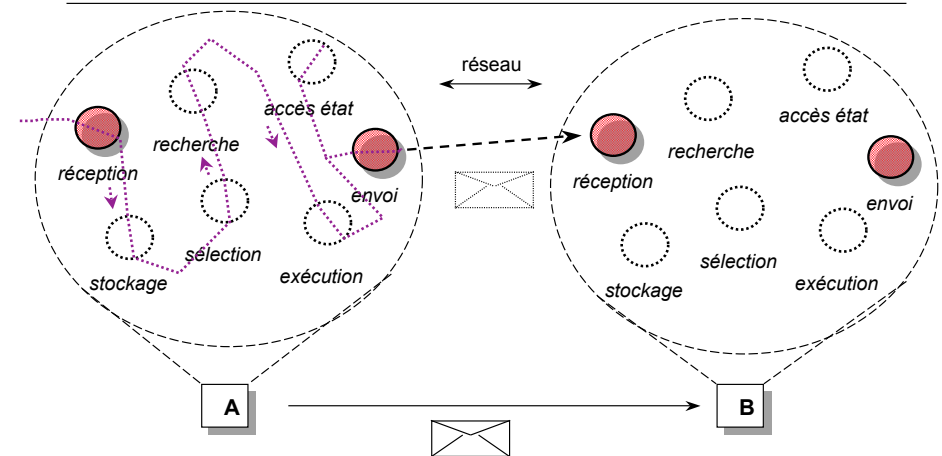


Ex : Exécution répartie

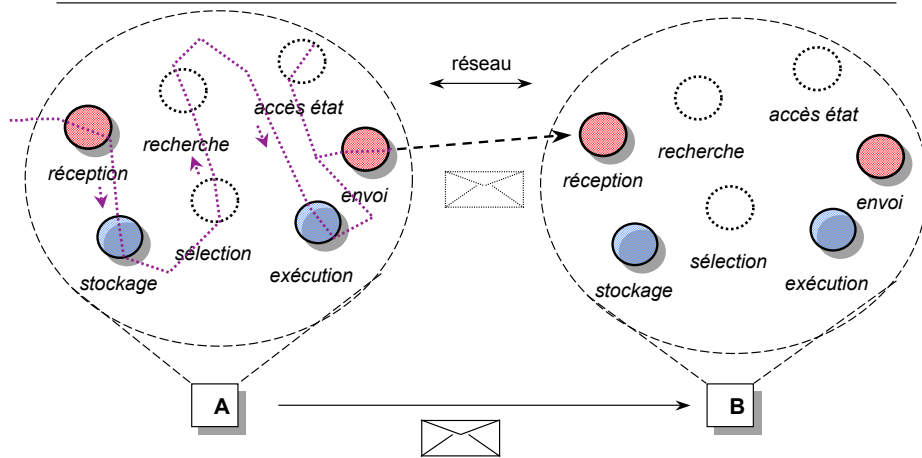
- **envoi de message**
 - » encodage des messages, envoi via le réseau
- **réception de messages**
 - » réception via le réseau, décodage des messages
- stockage des messages reçus
- sélection du premier message à traiter
- recherche de méthode correspondant au message
- exécution de la méthode
- accès à l'état de l'objet
- **encodage**
 - » discipline d'encodage (marshal/unmarshal)
- **référence distante**
- **espace mémoire**



Exécution répartie (2)

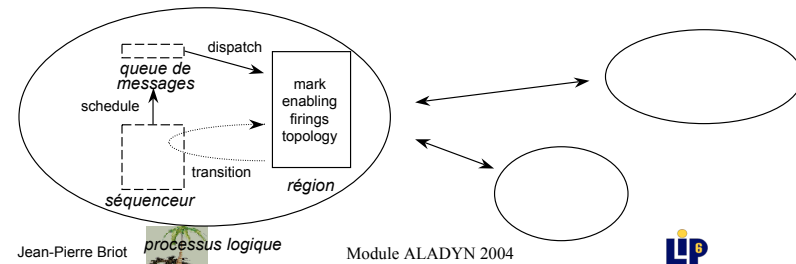


Exécution concurrente et répartie (composition)



Temps réel

- méta-acteurs associés à des acteurs
 - contrôle du temps pour du «soft real time» [Honda'92]
- machine de contrôle [Nigro et al., FMOODS'97] pour un ensemble d'acteurs
 - méta-composants :
 - » horloge, queue de messages (= liste d'événements), contrôleur (période de simulation), séquenceur
 - » permettent de modifier les aspects temporels de l'application indépendamment de l'application elle-même
 - » ex : simulation distribuée optimiste (Time Warp) de réseaux de Petri temporels (timed Petri nets)

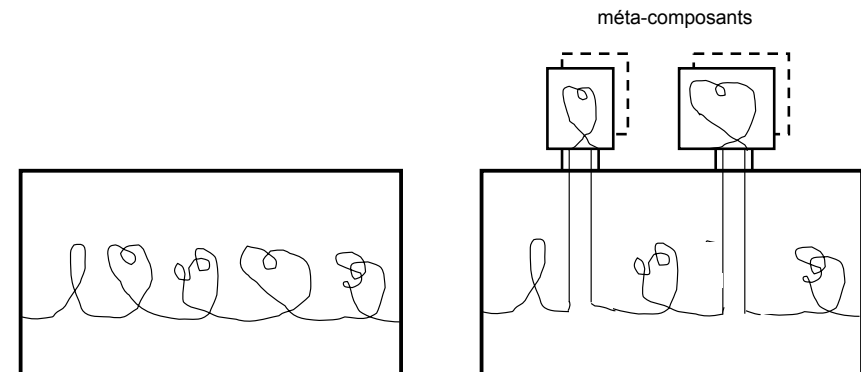


Systèmes commerciaux

- *Muse (ex-Aperios)* [Yokote OOPSLA'92]
 - spécialisation dynamique de la politique de séquençement (ex : passer au temps réel)
 - application au «video on demand» et aux robots-chiens Aibo (Sony)
- Moniteur de transaction [Barga et Pu '95]
 - Incorporation de protocoles transactionnels étendus (relâchant certaines des propriétés standard : ACID)
 - dans un système existant
 - réification a posteriori via des upcalls
 - » (délégation de verrou, identification de dépendances, définition de conflits)



Moniteur de transaction rendu réflexif/ouvert



Exemple : CORBA

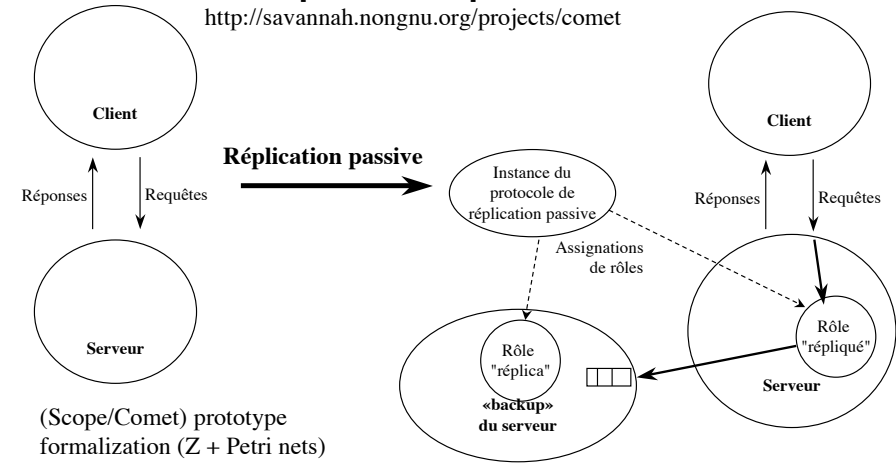
- approche réflexive
 - réification de certaines caractéristiques de la communication
 - » ex : smart proxies de Orbix (IONA)
 - ex d'utilisation : implantation de transmission de messages asynchrone
 - intégration des services avec la communication distante



Ex : Installation dynamique et transparente d'un protocole de réplication passive

COMET [Peschanski '99]

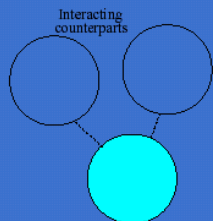
<http://savannah.nongnu.org/projects/comet>



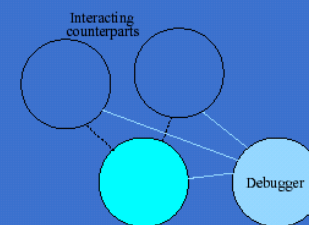
(Scope/Comet) prototype
formalization (Z + Petri nets)
Scheme/Java implementations



Example 2 : Beyond mechanisms



Example 2 : Beyond mechanisms



Issues:

→ interaction consistency

→...

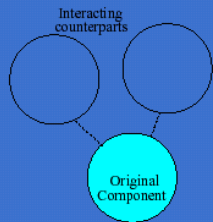
In our opinion, important prerequisites:

- Structural decoupling = explicit (and dynamic) links
- Control-level decoupling = asynchronism

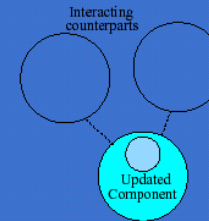
(Strong) claim : Object-orientation is not adequate



Example 3 : Fine-grained Changes



Example 3 : Fine-grained Changes



Issues:

- impact on component internals
- security
- ...

- Design abstractions?
- **Language constructs?**
- **Operational mechanisms?**



The Comet Middleware

[Frédéric Peschanski, Middleware'2003]

A (Free Software) Distributed Environment
for Dynamic Distributed Systems

- Architectural reconfigurations :
 - Adding/replacing/removing components
 - Modifying interconnexions
 - Mobility (strong migration is on the way)
- **Finer-grained changes**
 - Whithin component boundaries
 - Behavioral changes



The Comet Middleware

A (Free Software) Distributed Environment
for Dynamic Distributed Systems

- **The Scope composition language**
 - DSL constructs embedded in a mother language (Java/Scheme)
 - Source-to-source compiler
- **The Comet Middleware**
 - Efficient distributed execution of Scope programs
 - Implemented in Java (JDK >=1.2) above Sockets



Scope Example : Client/Server

Component Definitions :

```
component MMClient {
  receive MMEvent; send MMRequest;
  when(MMEvent event) { show(event); }
  void askServer(MMRequest req) { send(req); }
```

```
component MMServer {
  receive MMRequest; send MMEvent;
  when(MMRequest request) {
    // subclasses refine this method
    doRequest(request); } }
```



Scope Example : Client/Server

Component Definitions :

```
component MMClient {
  receive MMEvent; send MMRequest;
  when(MMEvent event) { show(event); }
  void askServer(MMRequest req) { send(req); }
```

```
component MMServer {
  receive MMRequest; send MMEvent;
  when(MMRequest request) {
    // subclasses refine this method
    doRequest(request); } }
```

Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
```



Scope Example : Client/Server

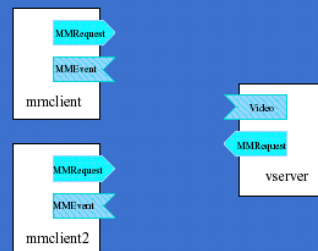
Component Definitions :

```
component MMClient {
  receive MMEvent; send MMRequest;
  when(MMEvent event) { show(event); }
  void askServer(MMRequest req) { send(req); }
```

```
component MMServer {
  receive MMRequest; send MMEvent;
  when(MMRequest request) {
    // subclasses refine this method
    doRequest(request); } }
```

Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
mmclient2 = instantiate(MMClient, <location2>);
vserver = instantiate(VideoServer, <location3>);
```



Scope Example : Client/Server

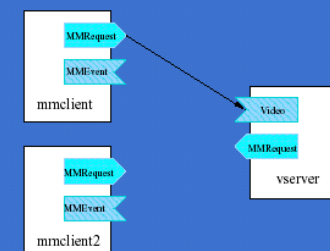
Component Definitions :

```
component MMClient {
  receive MMEvent; send MMRequest;
  when(MMEvent event) { show(event); }
  void askServer(MMRequest req) { send(req); }
```

```
component MMServer {
  receive MMRequest; send MMEvent;
  when(MMRequest request) {
    // subclasses refine this method
    doRequest(request); } }
```

Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
mmclient2 = instantiate(MMClient, <location2>);
vserver = instantiate(VideoServer, <location3>);
connect(mmclient, vserver, MMRequest);
```



Scope Example : Client/Server

Component Definitions :

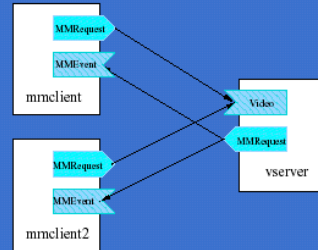
```
component MMClient {
  receive MMEvent; send MMRequest;
  when(MMEvent event) { show(event); }
  void askServer(MMRequest req) { send(req); }
```

```
component MMServer {
  receive MMRequest; send MMEvent;
  when(MMRequest request) {
    // subclasses refine this method
    doRequest(request); } } send(event);
```

Configuration console (could be a program/script):

```
mmclient = instantiate(MMClient, <location>);
mmclient2 = instantiate(MMClient, <location2>);
vserver = instantiate(VideoServer, <location3>);

connect(mmclient, vserver, MMRequest);
connect(vserver, mmclient, MMEvent);
... // and so on
```



Replication Protocol (2)

```
role ReplicatedRole {
  prehook MMRequest ;
  send MMRequest ;
```

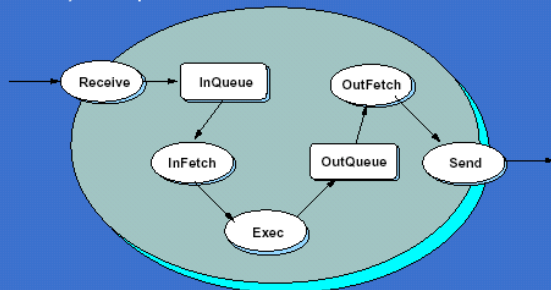
```
before MMRequest(event) {
  send (event); }
```

```
protocol ReplicationProtocol {
  public void createReplica(String repClass, Address machine) {
    ComponentRef ref = upload (repClass, machine);
    instantiate (ref); }
  public void doReplication(ComponentRef server, ComponentRef replica) {
    assign(ReplicatedRole, server);
    connect(server, replica, Event); } }
```



Component Internals

Within (top-level) Components : Micro-architecture of sub-components



Fine-grained Dynamic Adaptability :

- Add/change/remove sub-components
- Change interconnections



Protocols and Roles

Language-level abstractions

- **Protocols** : Description of Adaptive Interactions
- **Roles** : What should provide/require the involved components

Categories of adaptation:

- **Functional roles** : Add/Replace functionalities
- **Hooks** : pluggable wrappers around functionalities
- **Filters** : Type/Content-based filtering



Functional Roles : an Example

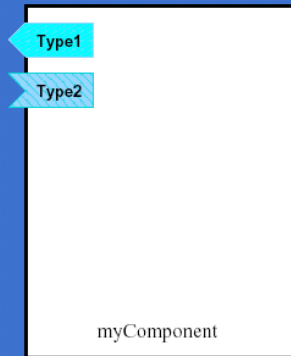
```

event ConfigEvent {
  slot key type String;
  slot val type Object;
  String getKey() { return key; }
  Object getVal() { return val; }

  role RecordRole {
    receive RecordEvent;
    HashMap config = new HashMap();
    when(RecordEvent re) {
      config.put(re.getKey(), re.getVal());
    }

    protocol ConfigAdapt {
      void adapt(ComponentRef comp) {
        assign(comp, RecordRole);
      }
    }
  }
}

```



Functional Roles : an Example

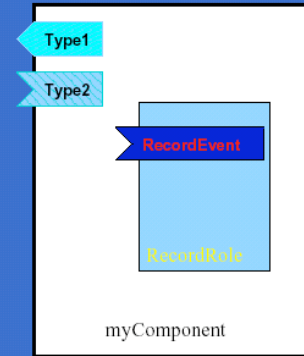
```

event ConfigEvent {
  slot key type String;
  slot val type Object;
  String getKey() { return key; }
  Object getVal() { return val; }

  role RecordRole {
    receive RecordEvent;
    HashMap config = new HashMap();
    when(RecordEvent re) {
      config.put(re.getKey(), re.getVal());
    }

    protocol ConfigAdapt {
      void adapt(ComponentRef comp) {
        assign(comp, RecordRole);
      }
    }
  }
}

```



Functional Roles : an Example

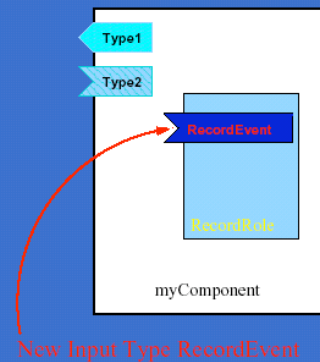
```

event ConfigEvent {
  slot key type String;
  slot val type Object;
  String getKey() { return key; }
  Object getVal() { return val; }

  role RecordRole {
    receive RecordEvent;
    HashMap config = new HashMap();
    when(RecordEvent re) {
      config.put(re.getKey(), re.getVal());
    }

    protocol ConfigAdapt {
      void adapt(ComponentRef comp) {
        assign(comp, RecordRole);
      }
    }
  }
}

```

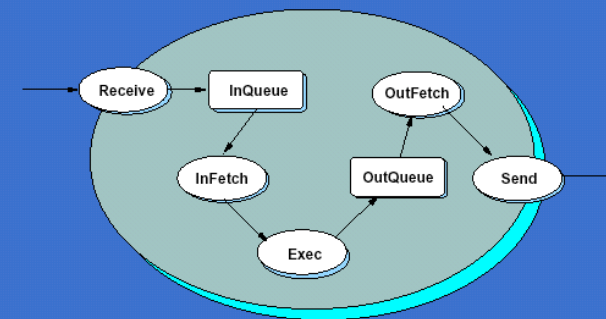


New Input Type RecordEvent



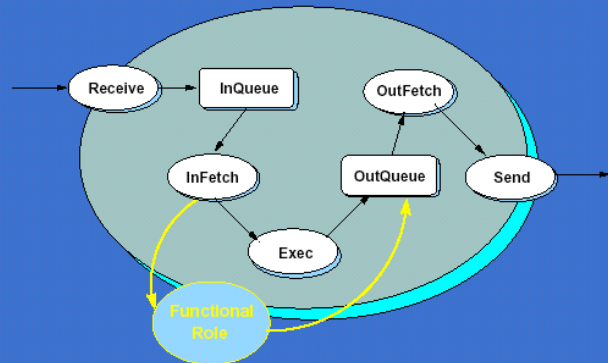
Functional Roles : Assignment

Internally : Derivation as “Exec-like” sub-component



Functional Roles : Assignment

Internally : Derivation as “Exec-like” sub-component

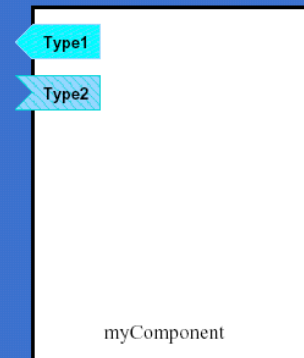


Hook Roles : an Example

```
role CountRole {
  prehook Event;
  int count = 0;
  when(Event e) {
    count++;
  }

  protocol CountAdapt {
    void adapt(ComponentRef comp) {

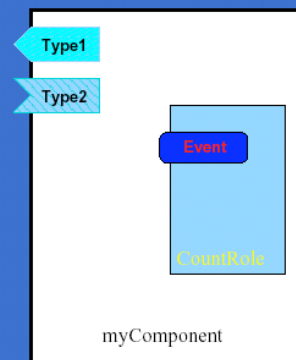
    }
  }
}
```



Hook Roles : an Example

```
role CountRole {
  prehook Event;
  int count = 0;
  when(Event e) {
    count++;
  }

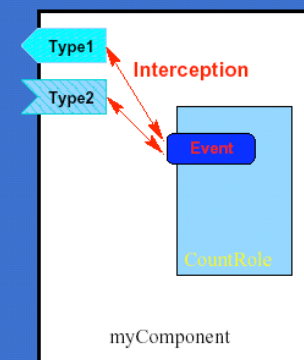
  protocol CountAdapt {
    void adapt(ComponentRef comp) {
      assign(comp, CountRole);
    }
  }
}
```



Hook Roles : an Example

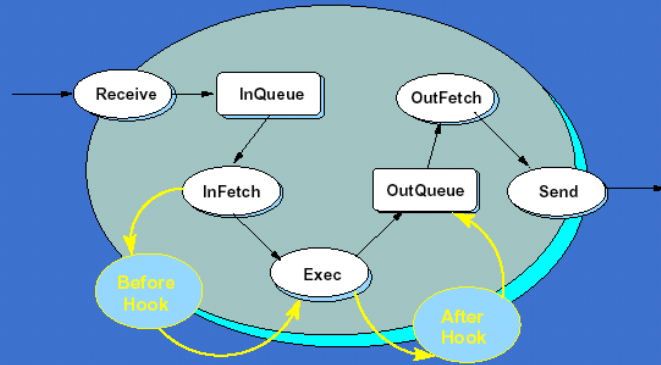
```
role CountRole {
  prehook Event;
  int count = 0;
  when(Event e) {
    count++;
  }

  protocol CountAdapt {
    void adapt(ComponentRef comp) {
      assign(comp, CountRole);
    }
  }
}
```



Hook Roles : Assignment

Internally : Wrappers around the Exec sub-component

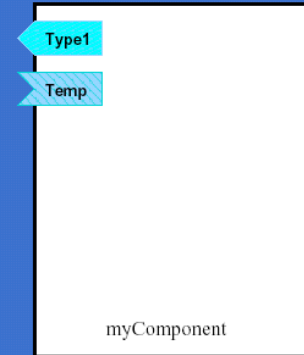


Filter Roles : an Example

```
event TempEvent {
  slot temp type float;
  float getTemp() { return temp; }}
```

```
role TempFilter {
  infilter TempEvent;
  int count = 0;
  when(TempEvent t) {
    if(t.getTemp()>0.0) return t;
    else return null; }}
```

```
protocol TempAdapt {
  void adapt(ComponentRef comp) {
    myComponent
  }}
```

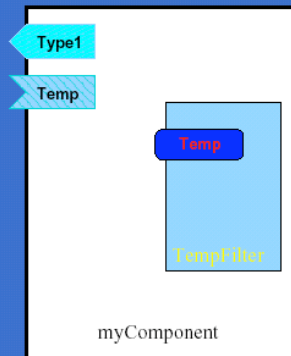


Filter Roles : an Example

```
event TempEvent {
  slot temp type float;
  float getTemp() { return temp; }}
```

```
role TempFilter {
  infilter TempEvent;
  int count = 0;
  when(TempEvent t) {
    if(t.getTemp()>0.0) return t;
    else return null; }}
```

```
protocol TempAdapt {
  void adapt(ComponentRef comp) {
    assign(comp, TempFilter);
  }}
```

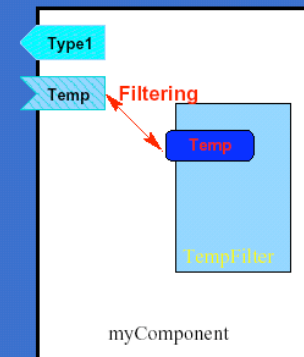


Filter Roles : an Example

```
event TempEvent {
  slot temp type float;
  float getTemp() { return temp; }}
```

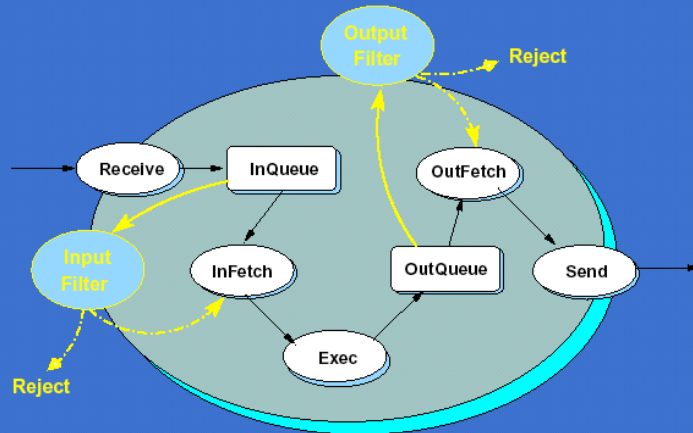
```
role TempFilter {
  infilter TempEvent;
  int count = 0;
  when(TempEvent t) {
    if(t.getTemp()>0.0) return t;
    else return null; }}
```

```
protocol TempAdapt {
  void adapt(ComponentRef comp) {
    assign(comp, TempFilter);
  }}
```



Filter Roles : Assignment

Internally : Pre-fetching sub-components



Role composition : an example

Issue: Asynchronous control requires special care when :

- source and destination components do not work at the same speed
- network slowdowns

Risk : event queue “explosions”



Role composition : an example

Default Event flow management algorithm :

- Send and queue event until **ceil capacity** exhausted on either source or destination
- Stop exceptions sent to senders (no propagation though)
- Restart exceptions sent when **floor capacity** reached



Role composition : an example

Efficient **optimistic** algorithm for most systems But some systems needs more pessimistic algorithms (perhaps temporarily)

Idea: Dynamic adaptation for pessimistic flow regulation

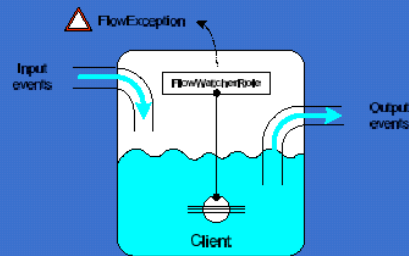


Flow Watching Role

```

role FlowWatcher {
  prehook Event; send FlowException;
  long _event_count=0; long _avg_limit=100;
  long _avg_time=0;
  before(Event event) {
    _event_count++;
    long current = System.currentTimeMillis();
    _total_time = _total_time + current;
    _avg_time = _total_time / _event_count;
    if(_avg_time > _avg_limit)
      send(new FlowException(_avg_time));
  }
}

```



Si le temps de traitement (temps / nbre d'événements traités) dépasse une certaine limite, une exception est émise indiquant cette nouvelle limite, qui deviendra la cadence minimale

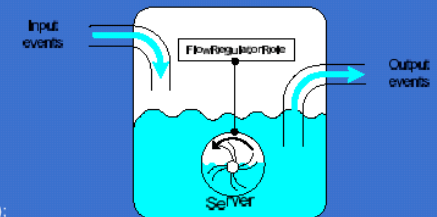


Flow Regulation Role

```

role FlowRegulator {
  prehook Event; posthook Event;
  receive FlowException;
  long _start_time; long _end_time;
  long _rate=1000;
  before(Event event) {
    _start_time = System.currentTimeMillis();
  }
  after(Event event) {
    _end_time = System.currentTimeMillis();
    if(_end_time - _start_time < _rate)
      Thread.sleep(min_rate - (_end_time - _start_time));
  }
  when(FlowException except) {
    _rate = except.getRate(); // Detected anomaly
  }
}

```

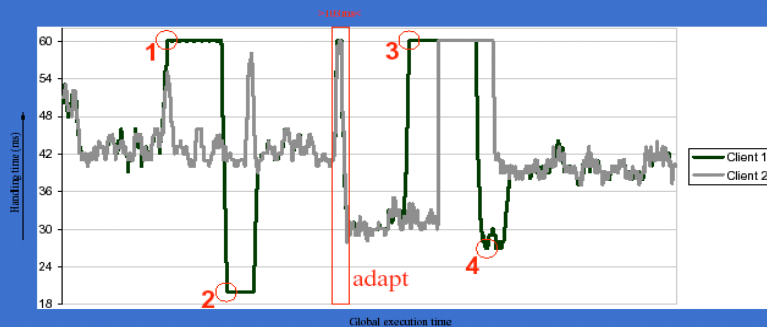


On garantit une cadence minimale

Mise à jour de la cadence minimale



Flow Regulation in Practice



```

Protocol FlowSyncProtocol {
  void sync(ComponentRef server ; ComponentRef client) {
    assign(client, FlowWatcher) ;
    assign(server, FlowRegulator) ;
    connect(server, client, FlowException) ;
  }
}

```



Security: Verification Contracts

"Hand-shake" contracts (using XML):

- Component host verifies assigned role
- Role verifies host

Contract categories:

- **Typing contracts** : check compatibility of role and components input, output, filter and hook types
- **Access contracts** : allow/forbid references/method calls within role bodies (e.g. `outer` reference)
- **Security contracts** : authentication, encryption, certification (Java Security API)



Conclusion and Future Work

Comet/Scope = Adaptation in Practice:

- Architectural Reconfigurations (see bibliography)
- **Finer-grained dynamic adaptations**
- A Free-software Project at GNU Savannah : <http://www.non-gnu.org/comet>
- Various Applications : MAS (DIMA), Dist. CAD (SALOME), Load-balancing, Dist. Debugging, ...

Ongoing work:

- **Mobility** : Self-stabilizing forwarding algorithm, Strong migration (integration with JavaGO)
- **Formal insights** : Essentially about Multicast Interactions
- **Real-world** : RMI/Corba Compararison/Interoperability
- **Longer term** : Low-cost mobility for large-scale reactive multi-agent systems



DARX : Self-Adaptation of Distributed Agents to Faults Towards Fault-Tolerant Agents

Starting points :

- Multi-agent systems aimed at distributed problem solving, distributed control...
 - but (paradoxally) yet rare large scale distribution
- Large scale distribution implies
 - taking partial faults (processor, network) into account
- A distributed system is one on which I cannot get any work done, because a machine I have never heard of has crashed. [Leslie Lamport]
- <http://www-src.lip6.fr/projets/darx/>



Bilan - Conclusion

- Approche réflexive prometteuse
- Architectures réflexives encore plus ou moins complexes, mais méthodologie s'établit et s'affine
- Validations en vraie grandeur en cours
- Retour du problème clé de la composition arbitraire (de méta-composants)
- (In)Efficacité
 - réduction de la portée de la réflexion (compilation)
 - » ex : OpenC++ version 2 [Chiba, OOPSLA'95]
 - » Javassist [Chiba, ECOOP'2000] - liaison au chargement des bytecodes
 - transformation de programmes - évaluation partielle
 - » [Masuhara et al., OOPSLA'95] [Consel et al. 2000]
- Ne dispense pas du travail nécessaire à l'identification des bonnes abstractions



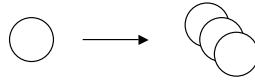
Examples of Applications

- Crisis management systems
 - Human agents
 - firemen, policemen, medics...
 - Various organizations and commanding chains
 - Information servers
 - stable
 - mobile
 - (Mobile) laptops and PDAs
 - (Assistants) software agents
- Computer-Supported Cooperative Work (CSCW)
- Community-based computing
 - A simple scenario : scheduling meetings through digital assistant agents
 - if an assistant machine (PDA, PC...) fails, scheduling process may be blocked
 - presence of *some* participants to a meeting may be critical
 - dynamic process (simultaneous meetings, evolving...)
- Workflow, etc.



Motivations

- To increase robustness and availability of agents (and their services), replication strategies have been proposed
 - mimicking nature (redundancy for robustness)
- But
 - they are **statically** applied
 - to which components they are applied is the **explicit decision** of the programmer
 - (ex : data bases servers)
- Problem :
 - does not apply to highly **dynamic** applications
 - criticality of software components may highly vary in time (ever changing focus of activity)
 - ex : crisis management systems, CSCW, community-based computing...
 - costly protocols (redundancy of data or/and activities, consistency synchronization)
 - one cannot apply them all the time to all components



Objectives

- Apply replication protocols
 - **dynamically**
 - and **automatically** : **where**, **when** and **how** it is useful
 - where : to which agents
 - when : at the right time (interval)
 - how :
 - which replication policy
 - how many replicas
 - where to create replicas
 - what consistency policy
- Using information :
 - run-time information
 - load, latency, etc.
 - application semantics
 - criticality of agents
 - potential communications
 - message criticality (e.g., exploiting KQML/ACL performatives)
 - ...



Replication policies (strategies)

- active
 - all replicas compute request
- passive
 - only primary replica computes request
- semi-active
 - as for active, but only one replica replies
- auto-stabilizing quorum-based
 - only a subset of replicas is maintained consistent
- each policy has its pros and cons :
 - reliability
 - availability
 - recovery delay
 - various overheads :
 - » checkpoint management
 - » processing
 - » communication



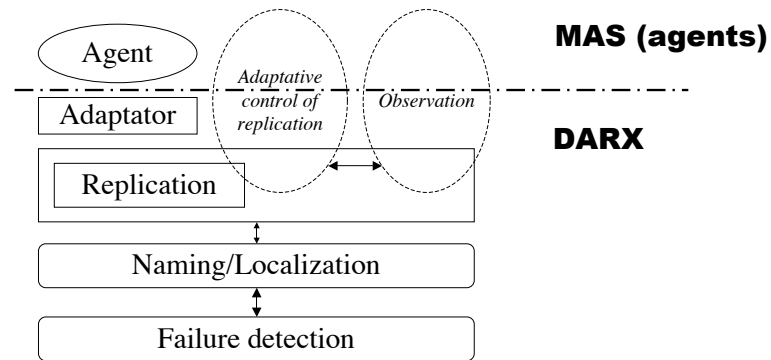
Dynamic adaptation of policies (meta-policies)

- Dynamically changing and tuning the replication policy
 - add/remove a replica
 - change strategy
 - tune strategy :
 - add/remove replicas
 - update frequency
 - change number of consistent replicas
 - where to create (placement) of replica(s)
- Criteria
 - system level
 - load, latency, etc.
 - application level
 - criticality of agents, communications...
 - specified by designer
 - automatically extracted/updated
 - Forces:
 - optimize the robustness and availability
 - minimize the overhead
 - Future work : adaptive learning (stochastic)



DARX

Framework + low level middleware



Jean-Pierre Briot



Module ALADYN 2004



73

Adaptive failure detection

- Adapting the suspicion window to the network speed (load)
- « heartbeats » diffusion
- 2 layers :
 - basic detection
 - QoS adaptation
- Adaptability :
 - Dynamic estimation
 - Emission interval
- IP-multicast

Jean-Pierre Briot



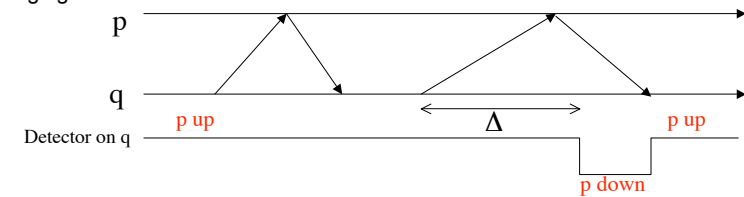
Module ALADYN 2004



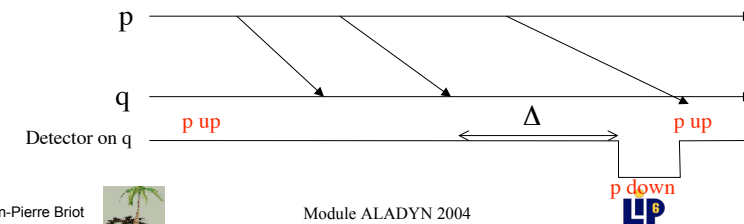
75

Failure detection techniques

- Applicative (denial of service)
- Pinging



- Heartbeat



Jean-Pierre Briot



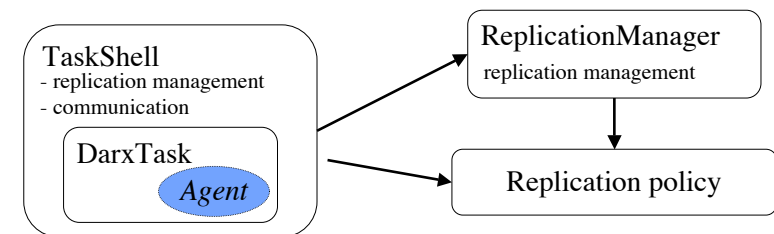
Module ALADYN 2004



74

Agent replication management

- Encapsulation of agents in system tasks
 - state (code + data) + threads
 - control primitives
- Encapsulation of system tasks in replication groups
 - unique leader unique
 - policy chosen
 - replication transparency (for other agents)



Jean-Pierre Briot



Module ALADYN 2004



75

Jean-Pierre Briot

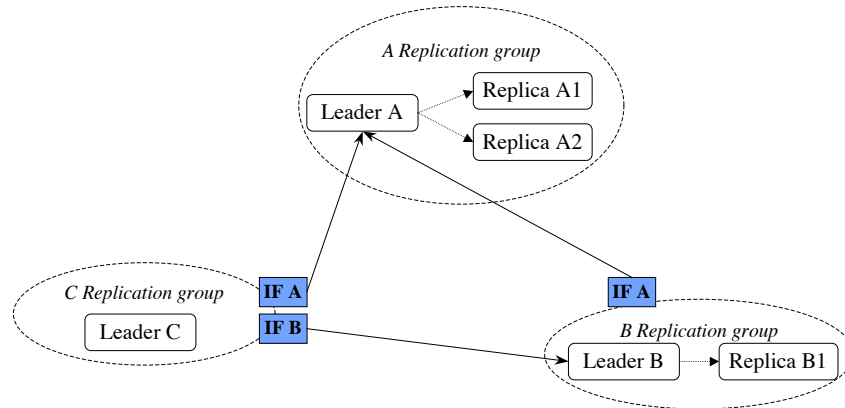


Module ALADYN 2004



76

DARX seen from the application



Jean-Pierre Briot

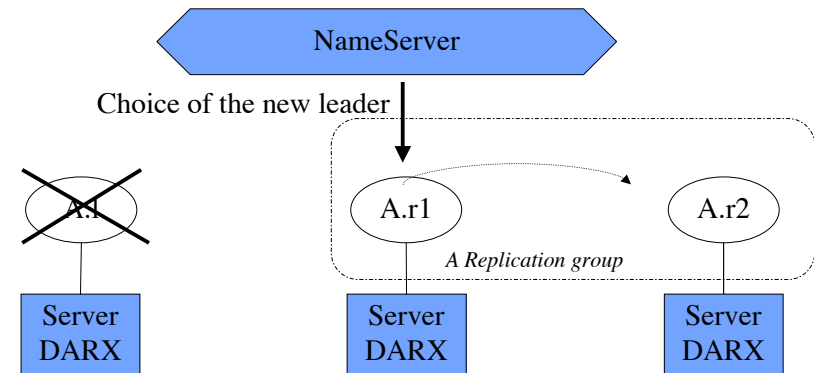


Module ALADYN 2004



77

Leader failure



Jean-Pierre Briot

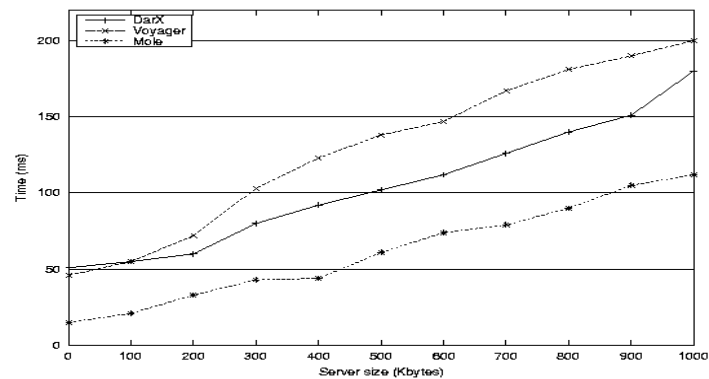


Module ALADYN 2004



78

Performances : (dynamically) Adding replicas



Jean-Pierre Briot

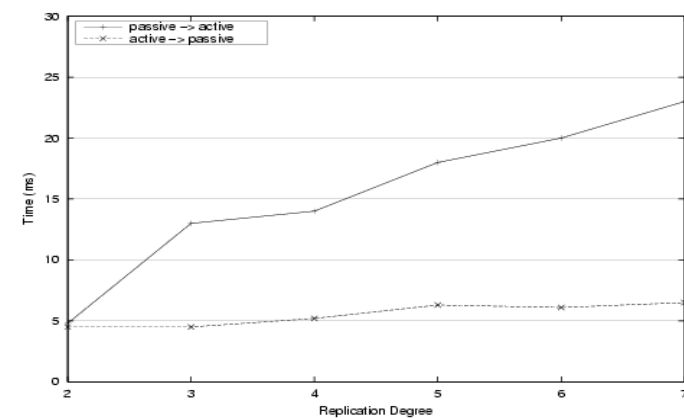


Module ALADYN 2004



79

Performances : (dynamically) changing replication policy



Jean-Pierre Briot



Module ALADYN 2004



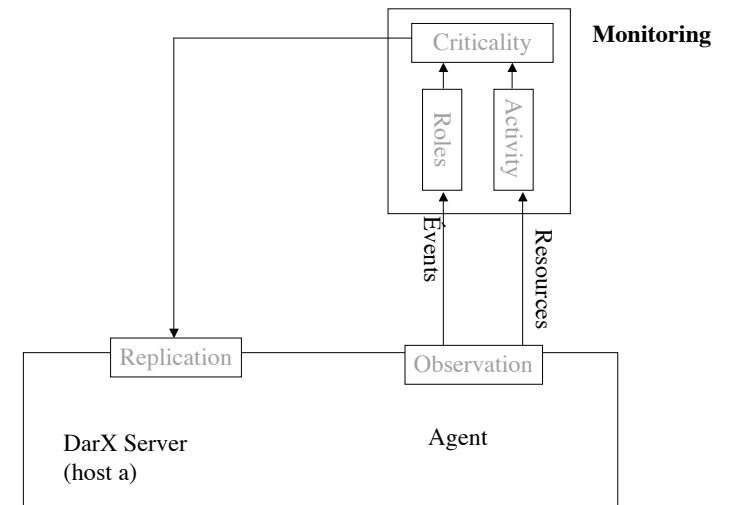
80

Criticality analysis

- Criticality : Impact of an individual failure on the agent organization(s)
- Semantic informations
 - Interdependency of agents (C. Castelfranchi)
 - Importance of messages (priorities, KQML/ACL performatives...)
- Roles
 - Implicit
 - » Role recognition (observing protocols)
 - Explicit
 - » Role declaration (intentions/commitments)



Architecture



Roles

- Electronic Agendas
 - digital assistant for each human user
 - Meetings (scheduling) management



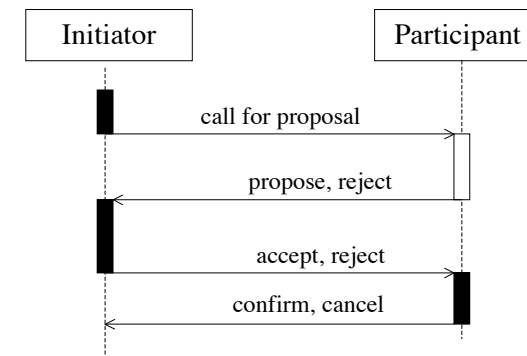
- Exemples of roles and their weights

Roles	Weight
Initiator	0.7
Participant	0.4



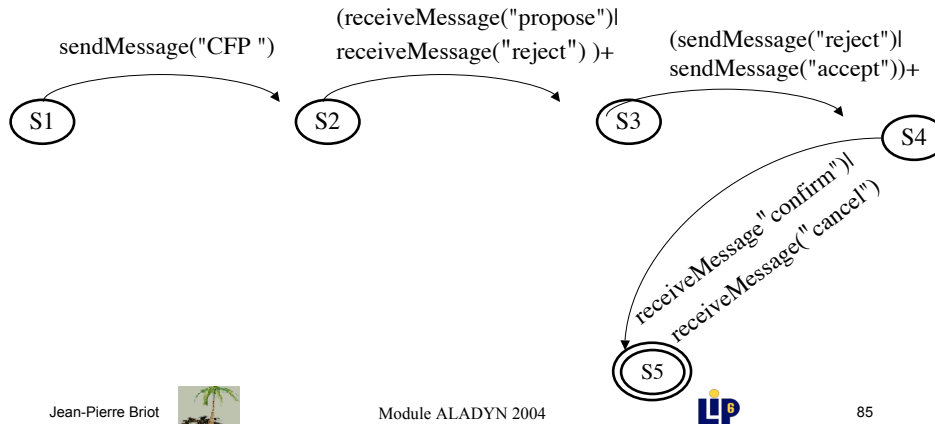
Roles in a Protocol

- Example: Contract Net



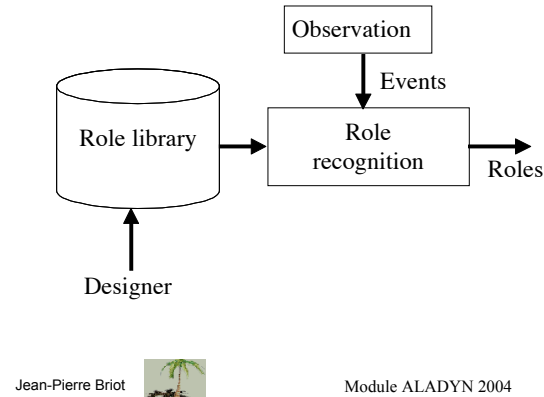
Roles

- Exemple of role : Initiator



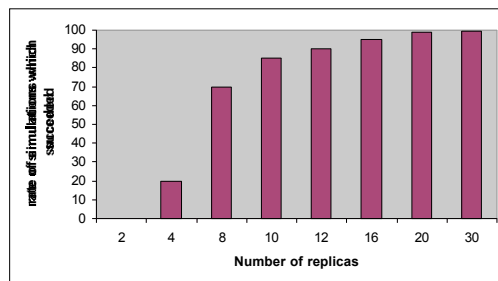
Roles

- Role observation/analysis
 - explicit (specified by agent)
 - implicit (from observation of events)



Performances : Agenda application

- Agenda
 - 20 agents on 8 machines
 - 100 faults injected (randomly killing one leader agent)



Another example : Dining Philosophers

- classical example in concurrent programming
- Small :()

Experimentation parameters

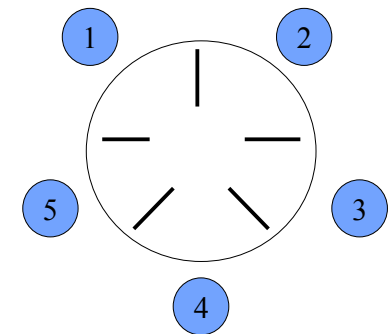
- 1 table
- N philosophers
- N chopsticks
- R = 100 meals to be served

Performance measurement

- Total execution time
- Global processing time

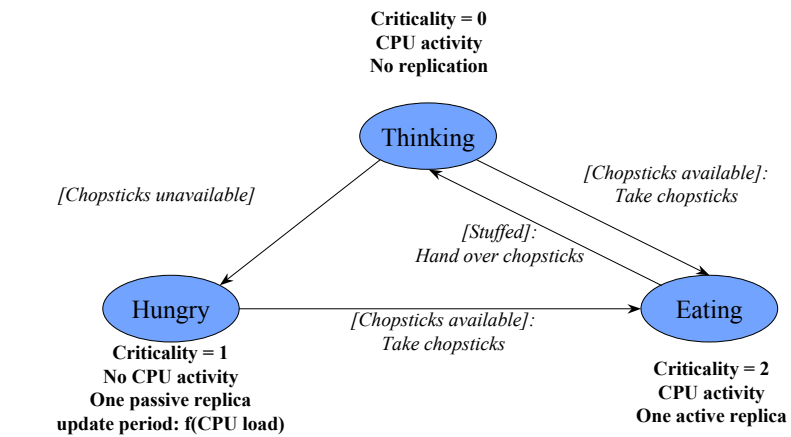
Environment

- DAS-2 Cluster (Vrije Universiteit, Amsterdam)
- 8 dedicated nodes: PIII 1Ghz bi-processors
- Myrinet-2000



Philosopher's behavior

• Dining Philosopher (Meta-) Behavior



Jean-Pierre Briot



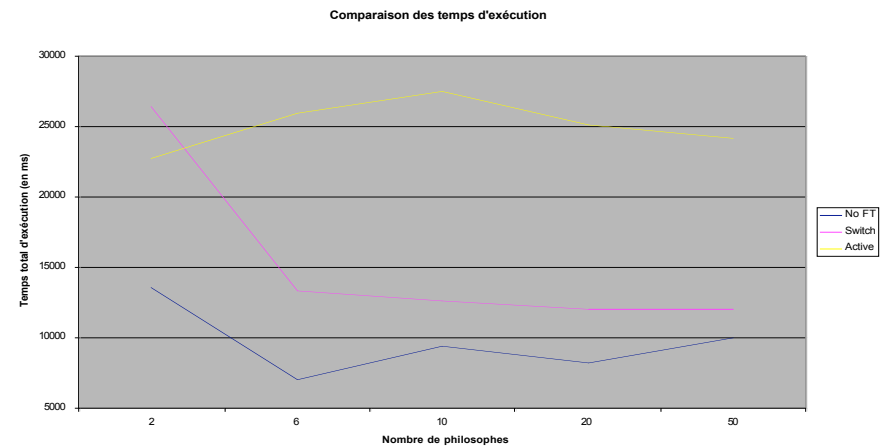
Module ALADYN 2004



89

Dining Philosophers - Performances

• Performance: Execution Time



Jean-Pierre Briot



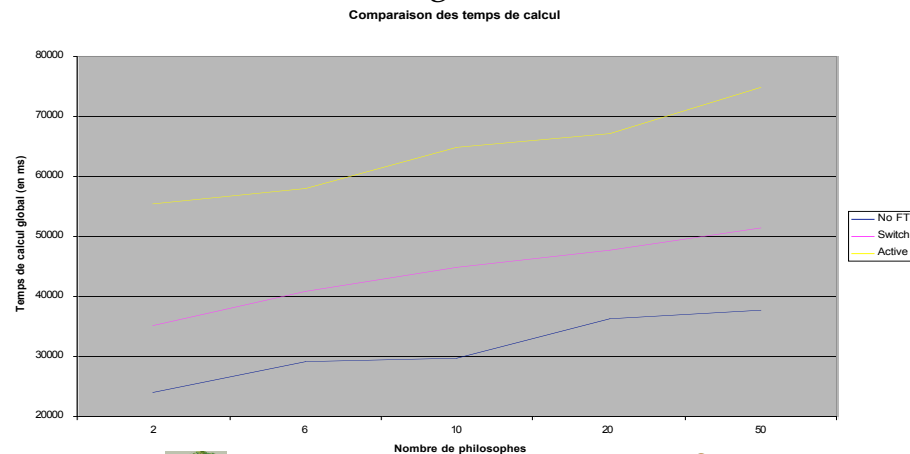
Module ALADYN 2004



90

Dining Philosophers - Performances (2)

• Performance: Processing Time



Jean-Pierre Briot



Module ALADYN 2004



91