Adaptations Dynamiques et Orthogonales de Composants Logiciels Distribués

Frédéric Peschanski *— Jean-Pierre Briot **

** Université de Tokyo*
*** Laboratoire d'Informatique de Paris 6*

RÉSUMÉ. *L'évolution logicielle de systèmes distribués en cours de fonctionnement, aujourd'hui peu maîtrisée, devra être prise en charge dans les plate-formes intergicielles de demain. Nous explorons cette direction dans le cadre d'une plate-forme componentielle exploratoire, Comet, en expérimentant le support dynamique de services distribués nommés protocoles. Les changements locaux nécessaires pour opérationnaliser ces protocoles sont décrits de façon orthogonale — indépendamment des composants applicatifs déjà déployés — sous la forme de rôles. Nous distinguons trois familles principales d'adaptations via les notions de rôles fonctionnels, de crochets et de filtres. L'application d'un protocole, par assignation dynamique des rôles le définissant, permet la mise à jour incrémentale des systèmes. Les protocoles que nous présentons dans cette étude concernent le déboguage adaptatif et la synchronisation du flux de communication dans le cadre d'une application simple de client/serveur multimédia. Les vérifications associées aux adaptations dynamiques présentées se divisent en trois catégories : contrats de typage, d'accès et de sécurité. Nous proposons également une première évaluation en conditions réelles de ce moteur adaptatif, dans le cadre de notre exemple applicatif.*

ABSTRACT. *The software evolution of distributed systems, nowadays hardly manageable, should be supported in future middleware platforms. In this perspective, we experiment the dynamic support of distributed services, or protocols in our terminology, in the realm of the exploratory Comet component-based middleware. The local modifications needed in the operationalization of such protocols —abstracted through the notion of role— are described orthogonally; that is, in an independent manner when considering the already deployed components. We distinguish three kind of dynamic adaptations through the notions of functional, hook and filter roles. The application of protocols, based on the dynamic assignment of the roles they define, allows incremental system updates. The protocols we present in this paper concern adaptive debugging and communication flow synchronization in the framework of a simple multimédia client/server application. These adaptations are verified from three complementary perspectives: type, access and security contracts. We also propose a preliminary evaluation of this adaptation engine under real-world conditions using our client/server application.*

MOTS-CLÉS : *Evolution logicielle, plate-forme intergicielle, composant*

KEYWORDS: *Software evolution, middleware, component*

1. Introduction

Les systèmes d'information distribués d'aujourd'hui — applications finales ou couches intergicielles — *évoluent* constamment, à un rythme qui s'accélère de jour en jour. A l'échelle de l'Internet, il semble clair que ce processus a lieu sans pour autant remettre en question le fonctionnement du système dans son intégralité. Cette *évolution logicielle*, aujourd'hui tangible mais *de fait* et (donc) peu maitrisée devra être supportée de façon *automatique* et *contrôlable* dans les plate-formes intergicielles de demain. Ces environnements du futur devront pour cela autoriser les changements dynamiques à la fois dans leur noyau fonctionnel (nouveaux services supportés), leur mode de contrôle (nouvelle façon de fournir un service), ou leur contexte opérationnel (environnement dynamique, utilisateurs mobiles), etc.

La tâche, prise globalement, semble titanesque, remettant profondément en question les méthodes de développement logiciel sous-jacentes des plate-formes intergicielles « classiques » (CORBA [Obj 01], RMI [Sun 01]) qui apparaissent assez démunies face au support effectif de changements dynamiques [GEI 01]. Certains travaux visent l'extension des middleware actuels, essentiellement par la conception de spécifications de services adaptatifs et l'implémentation de leur support opérationnel. Une autre voie, complémentaire, pose l'adaptation comme objectif fondamental via l'introduction de nouveaux langages et de nouvelles methodes ; remettant souvent en question les plateformes actuelles dans de plus larges mesures. C'est le cas de nos recherches — exploratoires — pour lesquelles nous avons conçu une plateforme open-source dédiée : Comet [com03]. Il est notable, en revanche, qu'une passerelle vers les applications RMI est fournie dans notre implantation.

Notre domaine d'application concerne le support adaptatif de services distribués, qui dénote la participation des composants du système à des services dont ils n'ont aucune « connaissance » *a priori* au moment de leur déploiement. Les modifications nécessaires, ou *adaptations*, doivent de plus se produire avec un minimum d'impact sur le fonctionnement des composants. La problématique qui émerge de cet objectif est duale. Pour chaque service adaptatif envisagé, il nous faut à la fois (1) permettre la description aussi simple que possible des adaptations nécessaires pour le support du service — et ce, à l'extérieur du système visé par la modification — et (2) assurer un cadre opérationnel sûr et efficace pour la mise en œuvre de ces descriptions. Notre approche se focalise sur des adaptations dont les descriptions sont donc *orthogonales* aux composants applicatifs concernés. En revanche, les effets de ces adaptations sont eux *transversaux* puisque se produisant à l'intérieure des composants. Dans de précédents travaux (compilés dans [PES 02a]), nous avons expérimenté les modifications architecturelles tels que l'ajout ou le retrait dynamique de composants. Dans cet article, nous explorons des changements se produisant à un niveau de granularité inférieur à celui des composants applicatifs. L'impact, ici essentiellement intrusif en terme comportemental, nécessite l'introduction de constructions linguistiques et de mécanismes opérationnels sous-jacents permettant d'interférer avec la fonctionnement interne des composants.

La section 2 de cet article décrit brièvement la plate-forme Comet et son modèle de type composant/événement au travers d'un exemple simple mais réaliste de client/serveur multimédia. Les principales abstractions permettant de décrire les adaptations — les *protocoles* et *rôles* — sont ensuite présentées (section 3). Nous illustrons leur usage en construisant les fondations d'un service de déboguage adaptatif basé sur trois mécanismes : inspection « à la demande » de composant, interception d'événements et mise en œuvre en mode pas-à-pas. Nous illustrons ici la complémentarité des trois principales formes de modifications disponibles via les rôles *fonctionnels*, *crochets* et *filtres*. En section 4, un exemple plus approfondi de synchronisation de flux événementiel est présenté et évalué, en terme de performances. Les importantes vérifications mises en œuvre au moment des modifications des composants sont ensuite présentées (section 5) selon trois points de vue : typage, accès interne et sécurité. Nous proposons ensuite quelques comparaisons avec des travaux similaires avant de conclure.

2. La plate-forme intergicielle Comet

Pour décrire les principes de base de la plate-forme Comet, nous allons construire une application dans le domaine du client/serveur multimédia, intéressant dans notre étude car focalisé sur des aspects essentiellement « système » [FIT 99]. Les composants mis en oeuvre dans notre application sont décrits dans le langage Scope pour lequel un traducteur source-à-source vers Java est disponible[1]. La définition suivante correspond à l'architecture de base des clients de notre application :

```
component MMClient {
  receive MMEvent ; send MMRequest ;
  when(MMEvent event) {
    // afficher le contenu multimedia
    show(event) ; }
  void askServer(MMRequest request) {
    // envoyer une requête au serveur
    send(request) ; } }
```

La déclaration `receive` et la construction `when` correspondant définissent le traitement des événements de type `MMEvent`. L'ensemble forme ce que nous appellerons un *bloc réactif* pour les événement de type `MMEvent`. Il s'agit ici d'afficher le contenu multimédia reçu (méthode `show` non détaillée). La méthode `askServer`, pour sa part, peut-être appelée pour émettre une requête de serveur (cf. la déclaration `send` préliminaire). Nous remarquons ici une propriété essentielle de l'environnement Comet : l'extraction de la relation de couplage entre composants au travers des types entrants (`MMEvent`) et sortants (`MMRequest`) qui leur sont associés. Notons particulièrement l'absence de référence explicite lors de l'émission d'un événement (primitive `send` dans la méthode `askServer`). L'architecture de base des serveurs est la suivante :

1. Il existe également un traducteur Scope pour Scheme ou CommonLisp.

```
component MMServer {
  receive MMRequest ; send MMEvent ;
  when(MMRequest request) {
    // Specialisee par les sous-classes
    doRequest(request) ;
  }
}
```

```
component VideoServer extends MMServer {
  receive MMRequest ; send VideoEvent ;
  double _frame_rate ; // Frame rate field
  void doRequest(MMRequest req) {
    ... // Pour toute la video ...
    send(req.sender, new VideoEvent(...)
    ... } }
```

La définition de gauche correspond aux serveurs générique dont une variante concrète pour la diffusion vidéo est décrite ci-dessus à droite. Un tel serveur video émet des événements de type image vers les clients connectés[2].

Comme dans le cadre des approches de type *publish/subscribe* [MEY 00], les *événements* dénotent en Comet les données échangées entre les composants au moment de l'exécution. De fait, le modèle proposé dans Scope est largement inspiré du langage de définition d'architecture Rapide [LUC 96]. Une relation de sous-typage étant proposée, nous ne donnerons ici qu'un aperçu d'un sous-type de `MMEvent` pour la gestion des événements vidéo :

```
event VideoFrameEvent is MMEvent {
  slot _contents type DeltaImage ;
  slot _serial type int ;
  slot _gentime type long ;
  VideoFrameEvent(DeltaImage ic, int is, long ig) {
   _contents = ic ; _serial = is ; _gentime = ig ;
  }
  DeltaImage getContents() { return _contents ;}
  int getSerial() { return _serial ; }
  long getGenTime() { return _gentime ; } }
```

Il est intéressant de noter dans cette définition le caractère *passif* des événements, simples regroupements de données ciblées (ici pour représenter une image différentielle dans un flux vidéo). La dichotomie composant/événement élève ainsi selon nous le degré de sémantique statique en comparaison d'objets plus « traditionnels », ce qui nous permet entre autre de générer automatiquement des procédures de linéarisation optimisées[3].

Nous devons maintenant établir la relation de couplage entre les composants clients et serveurs considérés. En d'autres termes, il nous faut *composer* notre architecture distribuée. Dans le cadre de Comet, les composants une fois déployés sont reliés entre eux par établissement dynamique de connexions *typées*, *unidirectionnelles* et *asynchrones*. Pour ce faire, il nous faut analyser leur

2. Lorsque les serveurs répondent à un client, la communication ne peut bien sûr plus être multipoint. Pour ce faire, nous utilisons le champs `sender` des événements et effectuons un envoi unipoint. Ceci fait partie des extensions de Scope/Comet pour le client/serveur (cf. [PES 02b]).
3. Si l'on compare avec RMI, par exemple, le compilateur Java ne "voit" aucune différence majeure entre une classe d'objet serveur et une class d'objet transmis par valeur (linéarisable). La différence entre composants et événements n'échappe pas au compilateur Scope.

type qui se décompose en une partie entrante (événements reçus) et une partie sortante (événements émis). Un composant client multimédia aura ainsi le type $MMClient = \langle MMEvent, MMRequest \rangle$. Un serveur générique aura, pour sa part, le type $MMServer = \langle MMRequest, MMEvent \rangle$ mais prenons le cas plus réaliste d'un serveur de flux vidéo proposant le type $VideoServer = \langle MMRequest, VideoFrameEvent \rangle$. Supposons de plus l'existence d'une instance `mmclient` de type $MMClient$ et d'une instance `vserver` de type $VideoServer$. Dans un premier temps, nous devons connecter le client au serveur pour autoriser l'émission des requêtes multimédia. Ceci se fait par l'intermédiaire d'une commande du langage Scope :

connect(mmclient, vserver, MMRequest)

De la même manière, nous devons relier en retour le serveur au client :

connect(vserver, mmclient, VideoFrameEvent)

Si deux clients sont ainsi liés à un serveur, nous obtenons l'architecture de la figure 1 qui, si ce n'est pas réellement perceptible dans le code proposé, est bel et bien distribuée. Les composants décrits sont effectivement intrinsèquement distribués et fonctionnent de plus de manière totalement découplée en terme de contrôle du fait de la nature asynchrone des connexions établies.

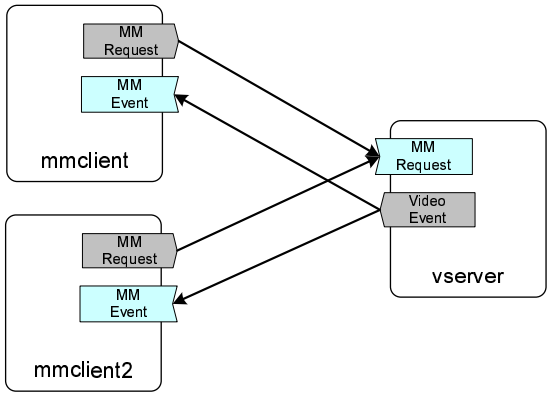


Figure 1. *Architecture client/serveur multimédia*

Remarquons que le type `VideoFrameEvent` est sous-type de `MMEvent`. Par subsomption, nous aurions pu utiliser ce dernier pour interconnecter nos composants. En fait, le langage Scope propose, via l'emploi d'un type « super-générique » `Event` d'inférer l'ensemble des interconnexions possibles entre composants. Dans notre exemple peu ambigü, nous aurions donc pu tout aussi simplement utiliser :

connect_infer(mmclient, vserver) ;
==> connect type = MMRequest

connect_infer(vserver, mmclient) ;
==> connect type = VideoFrameEvent

La réponse du système nous montre bien que du point de vue opérationnel, les deux jeux de connexions (inférées ou non) sont bien équivalentes.

Les composants clients et serveurs décrits ci-dessus sont en fait, au sens de la plate-forme Comet, des composants d'ordre supérieur (que nous dirons *composites*) : ils dénotent des sous-composants internes décrivant leur comportement. Par défaut, le modèle de comportement de chaque composant est décrit par une micro-architecture à cinq sous-composants réifiant le processus interne de cheminement des événements, comme indiqué sur la figure 2. Dans un premier temps, les événements sont reçus et mis en file d'attente (entrées asynchrones) par les composants *Receive* et *InQueue*. Ces mêmes événements sont ensuite extraits depuis la file d'attente dans une activité concurrente par le composant *InFetch*[4]. Le composant *Exec*, pour sa part, associe chaque événement à un bloc réactif associé (i.e. bloc `when`). Le composant *Send*, finalement, émet les événements selon un algorithme de *diffusion multi-point implicite en fonction du type* qui confère sa principale originalité au modèle de communication de la plate-forme Comet. Lors de l'émission d'un événement de type t par la primitive $send(t)$, une copie de cet événement sera diffusée sur chaque connexion établie pour le type t ou un super-type de ce dernier. Ceci permet la mise en œuvre simple d'architectures de nature multi-point (diffusion multimédia par exemple) sans pour autant lui sacrifier la communication point-à-point qui est vue comme un cas particulier (une seule connexion ou envoi explicite). Ce mode de communication particulier est illustré dans [PES 02b].

Le jeu initial de composants internes dote les composants d'un comportement proche (mais différent) de celui des acteurs [AGH 86]. D'autres sous-composants comportementaux internes peuvent être adjoints, par exemple pour extraire la sémantique d'accès à l'état interne du composant externe (sur la figure, composant $State$ optionnel).

Nous pouvons déduire de cette courte présentation de la plate-forme Comet qu'une modification comportementale au niveau du composant externe sera vue, au niveau opérationnel, comme une modification structurelle au niveau de l'architecture interne (et elle-même componentielle) du-dit composant. Ce changement de point de vue correspond à une évolution importante en comparaison du point de vue employé dans [PES 02a].

4. Les composants actifs, c'est-à-dire dénotant une (ou plusieurs) activité(s) autonome(s) en terme de contrôle, sont mis en relief sur la figure 2.

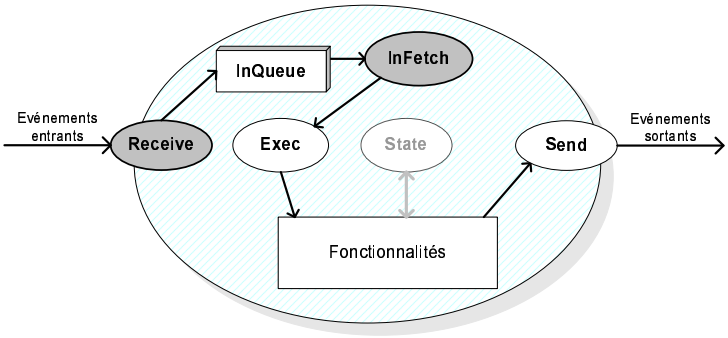


Figure 2. *Sous-composants comportementaux*

3. Protocoles et Rôles Comet

Dans le cadre de CORBA ou RMI, pour mettre en œuvre incrémentalement un *service commun*, il est nécessaire de stopper le système, d'en modifier le code puis de le recompiler/redéployer. Dans le cadre de notre application multimédia, nous allons considérer que le service fourni aux clients par les serveurs doit être *continu*. Il nous faut donc proposer un outil pour modifier le système sans pour autant l'interrompre. Ce medium d'adaptation correspond dans notre approche à une abstraction linguistique nommée *protocole*. Les descriptions protocolaires sont formées par l'association de *rôles* et de *fonctionnalités*. Chaque rôle décrit les conventions locales nécessaires pour qu'un composant participe au protocole envisagé. Une fonctionnalité, quant-à-elle, correspond à des traitements extérieurs aux entités mises en jeu (i.e. les rôles et composants). Il est possible d'adjoindre à de telles fonctionnalités un *état interne* de protocole, encore une fois indépendant de tout composant ou rôle mis en œuvre dans ce même protocole. Il existe plusieurs catégories de protocoles dont les plus simples ne concernent que des modifications structurelles des systèmes. Cependant, nous nous focalisons dans cet article sur des protocoles d'adaptations de grain plus fin, c'est-à-dire intervenant à un niveau intra-componentiel. De façon assez empirique, la déclaration d'un rôle ressemble fortement à celle d'un composant. D'un point de vue syntaxique, la principale différence réside dans l'usage du mot-clé `role` (en lieu et place de `component`). Il est toutefois important de noter qu'un rôle ne dénote aucun comportement propre ; ce dernier étant indissociable du composant auquel il sera assigné. Mais l'on définit également des propriétés spécifiques que l'on peut regrouper selon trois catégories non-exclusives de rôles : *fonctionnels*, *filtres* et *crochets*.

3.1. *Rôle fonctionnel : inspection de composant*

Un rôle fonctionnel permet d'ajouter une nouvelle fonctionnalité — ou bloc réactif — à un composant en cours d'exécution. Pour illustrer cette notion, nous allons

prendre l'exemple d'un protocole générique d'inspection interne de composant. La pièce maîtresse de ce service est le rôle `InspectorRole` défini ainsi :

```
role InspectorRole {
 receive InspectReq ; send InspectRep ;
 when(InspectReq req) {
  send(new InspectRep(outer.getSlotValue(req.getSlotName()))) ; } }
```

Ceci ressemble à une définition de composant mais il faut garder à l'esprit que l'identité opérationnelle d'un rôle (`this`) ne peut être considérée indépendamment de celle du composant auquel il est assigné (`outer`). Le bloc réactif pour les requêtes d'inspection fournit, via la référence `outer` et la méthode d'accès standard `getSlotValue()`, la valeur de l'attribut à inspecter (identifié par son nom et/ou son type). Pour communiquer avec ce rôle, il faut fournir des types entrants et sortants compatibles avec les types `InspectReq` (demande d'inspection) et `InspectRep` (résultat d'inspection), comme par exemple dans le composant suivant :

```
component InspectorClient {
 receive InspectReq ; send InspectReq ;
 Object inspect(String varname) {
  InspectRep rep = sendreceive(new InspectReq(varname)) ;
  return rep.getValue() ; } }
```

Ce client d'inspection définit une méthode `inspect` dont le but est d'inspecter à distance un composant (proposant le rôle `InspectorRole`). Nous pouvons y remarquer l'emploi de la primitive `sendreceive` qui autorise les échanges bidirectionnels, simplifiant considérablement la description des clients [PES 02b]. Les deux définitions précédentes peuvent être regroupées au sein d'un protocole d'inspection générique dont le contenu est le suivant :

```
protocol ComponentInspection {
 ComponentRef _inspector ; // Identification interne de l'inspecteur
 ComponentInspection() { // Création du composant interne
  _inspector = upload_instantiate_local(InspectorClient) ;
 }
 Object inspect(ComponentRef component, String varname) {
  // Adaptation du composant à inspecter (si nécessaire)
  if( !can_receive(component, InspectReq)) {
   assign(component, InspectorRole) ;
   connect_infer_all(component, _inspector) ; }
  return _inspector.getLocalRef().inspect(varname) ; // Effectue l'inspection
 } }
```

Ce protocole gère un composant interne, de type `InspectorClient` en communication directe avec des rôles d'inspection, ces derniers devant préalablement être assignés dynamiquement aux serveurs cibles. Pour ce faire, nous commençons dans la fonctionnalité de protocole `inspect` par nous renseigner si le composant peut recevoir les événements d'inspection. Ainsi, le prédicat `can_receive(comp, T)` retourne vrai si `comp` peut recevoir les événements de type `T`. Si le test est négatif alors

le rôle est assigné dynamiquement au serveur par l'intermédiaire de la commande `assign`. Cette assignation est bien sûr finement contrôlée (cf. section 5). Ensuite, le composant interne `InspectorClient` est connecté, en considérant tous les types possibles (`connect_infer_all`), au rôle assigné. Pour illustrer l'emploi de ce protocole, nous devons dans un premier temps en créer une instance :

```
ComponentInspection columbo = new ComponentInspection()
```

Il est ensuite possible d'inspecter les variables de composants comme notre serveur `vserver` précédemment déployé :

```
println(columbo.inspect(vserver, "_frameRate") ;
==> [float] 29.9673
```

Ici, la valeur de l'attribut interne `_frameRate` de `vserver` est inspectée. La figure 3 décrit l'impact de l'application du protocole d'inspection sur notre exemple. Nous pouvons remarquer le caractère autonome de ce protocole qui gère, comme état interne, le composant client d'inspection. Ce dernier est relié au serveur vidéo via le role `InspectorRole` préalablement assigné.

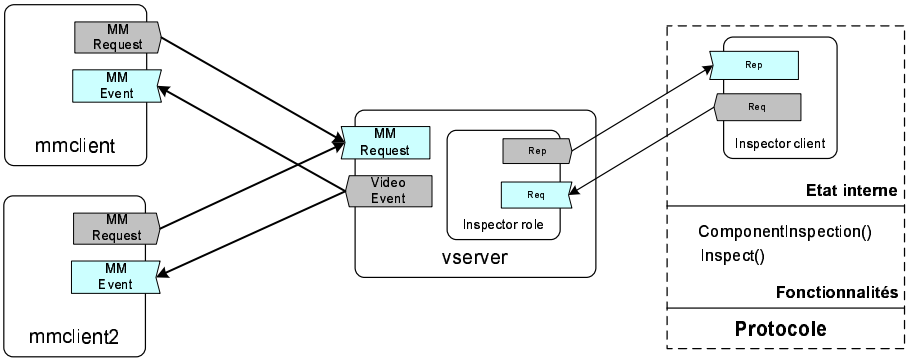


Figure 3. *Application du protocole d'inspection de composant*

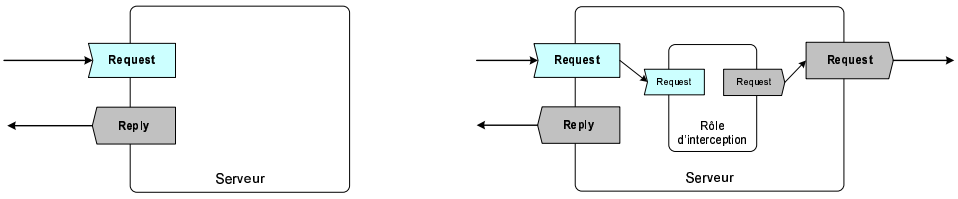
Notons qu'il n'est fait, au niveau du protocole `ComponentInspection`, aucune supposition sur la nature des composants concernés. Il est donc virtuellement possible d'adapter n'importe quel composant existant (serveur ou non) pour participer au protocole d'inspection, aux conflits de typage près et en établissant de plus des règles strictes quant aux possibilités d'accès interne depuis l'extérieur (cf. section 5). En terme de comportement, l'assignation d'un rôle fonctionnel est d'une grande simplicité puisqu'elle correspond en fait à une modification structurelle très simple de l'architecture interne du composant hôte. Sur la figure 5, nous pouvons voir que du point de vue interne, le rôle correspond en quelque sorte à une nouvelle « instance » de composant *Exec*.

3.2. *Rôle crochet : interception d'événements*

Le propos des rôles crochets (ou *hooks* en anglais) consiste à « encadrer » une fonctionnalité de composant par des pré et/ou post-traitements. Un crochet *avant* (ou *prehook*) sera ainsi déclenché immédiatement avant le traitement « normal » d'un événement reçu, puis ce sera le tour de l'éventuel crochet *après* (ou *posthook*). Nous allons illustrer ce type de rôle en construisant un mécanisme d'interception d'événements reçus par des composants en cours de fonctionnement. Dans l'exemple de la section 2, les événements reçus par le serveur multimédia sont des requêtes de type `MMRequest`. En supposant que ce type dérive d'un super-type `Request`, il nous est possible de décrire un rôle générique (pour le type `Request`) offrant le service attendu d'interception :

```
role InterceptorRole {
  prehook Request ; send Request ;
  before (Request request) {
    send(request) ; // Envoyer une copie
  } }
```

Ce rôle définit un crochet avant (bloc réactif `prehook/before`) pour les événements de type `Request` et dont le propos concerne leur réémission pure et simple. Le traitement de l'événement ne se trouvera bien sûr pas perturbé par ce « détour » (i.e. le sous-composant *Exec* recevra également une copie).



(a) Serveur en configuration normale

(b) Serveur « intercepté »

Figure 4. *Adaptation d'un serveur pour l'interception des événements reçus*

Après assignation de ce rôle (via la primitive `assign` ou dans le cadre d'une définition de protocole plus complète) au composant serveur de la figure 4(a), nous obtenons le composant de la figure 4(b), adapté également selon un aspect *fonctionnel* — le serveur dispose d'un nouveau type sortant, ici `Request`, émis par le rôle.

Comme montré sur la figure 5, les rôles crochets, contrairement aux rôles fonctionnels, encadrent le sous-composant *Exec* sans le « court-circuiter ». Encore une fois, nous pouvons constater que si pour le composant l'impact est bel et bien comportemental, le système lui ne fait que modifier des propriétés purement structurelles (ajouts/retraits de sous-composants et de connexions).

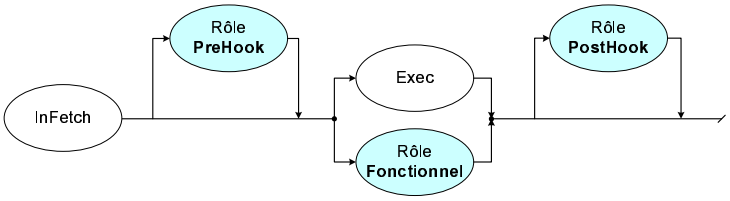


Figure 5. *Comportement interne, rôles fonctionnels et crochets*

Il est intéressant de noter que ce rôle d'interception d'évenements se trouve également à la base de nos précédents travaux sur la mise en œuvre de protocoles adaptatifs pour la réplication de serveur, comme présenté dans [PES 00].

3.3. *Rôle filtre : exécution en mode pas-à-pas*

Les rôles *filtres* dénotent comme les crochets avant des traitements ayant lieu lors de la réception des événements. Mais un filtre peut également bloquer la réception d'un événement, l'annuler ou bien encore la remettre à plus tard. Nous pouvons par exemple, par assignation d'un tel rôle, faire passer dynamiquement un composant du mode de contrôle réactif standard à un mode de fonctionnement *pas à pas*. Voici un exemple d'un rôle filtre servant ce propos :

```
role StepFilter {
 filter Event, Step ; // Filtre Event et Step
 LinkedList _stepList = new LinkedList(10) ; int _maxStep = 10 ;
 Event filter (Event event) {
  _stepList.addFirst(event) ; // Enregistrer
  if(_stepList.size()>=_maxStep-1) // Fairness
   return _stepList.removeLast() ; // Return the oldest event
  else // Mode pas-a-pas
   return null ; // Annuler la livraison
  }
 Event filter (Step s) {
  return _stepList.removeLast() ; // Remplacer par le plus ancien
 } } }
```

Dans cet exemple (et une fois le rôle assigné), pour tout événement reçu de type T (ou tout sous-type), nous proposons son enregistrement dans une file d'attente puis l'annulation de son traitement immédiat (en retournant un filtrage `null`). La réception d'un événement de type `Step`, quant-à-elle, permet le traitement unitaire de l'événement filtré le plus ancien. Pour assurer la *vivacité* de la modification, nous imposons une limite (`_maxStep`) sur la taille de la file d'attente. Au niveau de la micro-architecture interne, la modification effectuée est représentée en figure 6, le rôle filtre s'intercalant entre la file d'attente des événements entrants et le composant d'extraction.

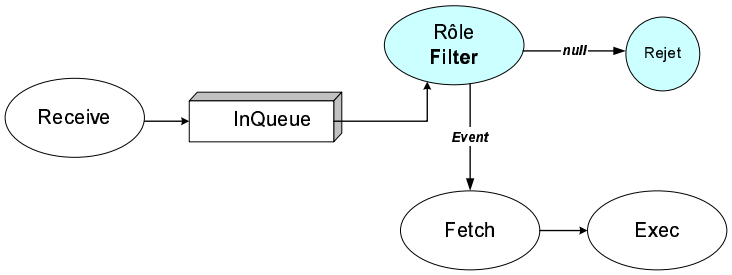


Figure 6. *Comportement interne et rôles filtres*

L'ensemble des rôles définis ci-dessus peuvent être regroupés dans la définition d'un protocole générique de déboguage distribué dont les principes sont décrits ci-dessous :

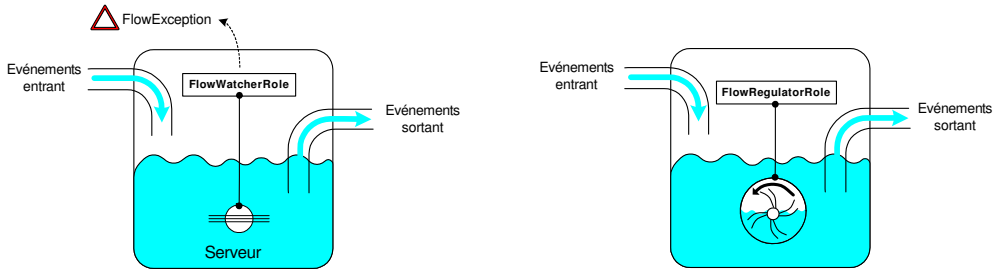
```
protocol DebugProtocol extends ComponentInspection {
 void stepMode(ComponentRef comp) { // Mode pas à pas
  assign(comp, StepFilter) ; }
 void step(ComponentRef comp) { // Traitement d'un événement
  send(comp, new Step()) ; }
 void reactiveMode(ComponentRef comp) { // Mode Réactif
  unassign(comp, StepFilter) ; } // Enlever le rôle
 void trace(ComponentRef comp, EventType type) { // Mode trace
  assign(comp, InterceptorRole) ;
  connect(comp, _inspector, type) ; } // Connexion au composant local hérité
 void unTrace(ComponentRef comp, EventType type) { // Fin de trace
  disconnect(_inspector, comp, type) ; // Déconnexion
  unassign(comp, InterceptorRole) ; } }
```

Cette définition montre bien que l'*essence* du protocole réside principalement dans les descriptions des rôles utilisés en paramètres des primitives `assign` et `unassign`. Le procole lui-même ne correspond qu'à un ensemble de fonctionnalités simples (séquences très courtes d'invocations de primitives). Remarquons cependant la possibilité de raffiner (par héritage dans notre implantation en Java) une définition de protocole existante (ici le protocole `ComponentInspection`). En plus de cet exemple simple et relativement puissant (pour le déboguage transparent d'applications distribuées), nous montrons dans [PES 02b] l'intérêt des rôles filtres dans le cadre de la mise en œuvre de protocoles de filtrage de contenu, suppléant à un niveau de granularité plus fin le mode original de filtrage en fonction du type proposé par la plate-forme Comet.

4. Service auto-adaptatif : synchronisation de flux

Les mécanismes construits dans les sections précédentes font partie d'un service plus général de déboguage transparent d'applications Comet qui peut être appliqué et

retiré dynamiquement par un opérateur, pendant le fonctionnement des applications[5]. Dans cette section, nous proposons un service plus autonome de synchronisation du flux des événements échangés entre les composants. Commençons par décrire un rôle de détection d'anomalies dans le flux événementiel (cf. figure 7(a)).



(a) Détection des anomalies (b) Régulation du flux

Figure 7. *Protocole de synchronisation de flux événementiel.*

Cette détection se base sur une donnée simple : l'intervalle de temps séparant le traitement de deux événements distincts au niveau du serveur. Il nous faut également donner un sens à l'expression « anomalie de flux » en relation avec cet indice d'intervalle temporelle de traitement. Nous établirons qu'un flux devient problématique si l'intervalle de traitement *moyen* dépasse une certaine limite fixée à l'avance (mais bien sûr adaptable à l'exécution). Nous donnons ci-dessous l'expression d'un tel rôle de détection :

```
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)) ;
 } }
```

Le rôle `FlowWatcher` met en œuvre un simple crochet avant traitement de tout événement au niveau du composant auquel il sera assigné. Lors du crochetage, le

5. Bien sûr lorsqu'une anomalie sérieuse est détectée, l'arrêt partiel ou total de l'application semble inévitable.

nombre d'événement traités `_event_count` est tout d'abord augmenté. Nous utilisons ensuite une méthode standard de Java pour récupérer le temps local immédiat en millisecondes. Ce temps est ajouté au temps total conservé dans `_total_time` puis une moyenne est calculée. Si cette moyenne `_avg_time` dépasse un certain seuil `_avg_limit`, alors une exception est simplement levée.

Pour compléter le protocole de synchronisation, il nous faut définir un rôle complémentaire permettant d'interpréter les exceptions signalées au niveau des serveurs. Dans notre exemple, nous employons une technique de régulation de flux événementiel en sortie du serveur adapté (cf. figure 7(b)). Lorsque ce rôle — nommé `FlowRegulator` — est assigné à un serveur, il impose une cadence minimale précise au traitement de chaque événement reçu. La définition proposée est la suivante :

```
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() ; // Anomalie detectee
 } }
```

Ce rôle, une fois assigné, intercepte tous les événements reçus par un serveur. Avant le traitement de chaque événement (crochet avant), le temps local précis est récupéré dans la variable `_start_time`. Après traitement (crochet après), le temps local est une nouvelle fois demandé (variable `_end_time`). Nous savons donc, à ce moment précis (et à la milliseconde près), combien de temps il a fallu pour traiter l'événement. Si ce temps est inférieur à la cadence minimale désirée (variable `min_rate`), alors on temporise pour que le prochain événement ne soit pas traité « trop tôt ». Une définition presque minimale pour notre protocole de synchonisation est donnée ci-dessous :

```
protocol FlowSyncProtocol {
 void sync(ComponentRef server, ComponentRef client) { // Synchroniser
  assign(client, FlowWatcher) ; // Surveillance du client
  assign(server, FlowRegulator) ; // Régulation du serveur
  connect(client, server, FlowException) ; // Connexion du protocole
 } }
```

Le graphique en figure 8 montre un scénario d'exécution de l'application client/serveur multimédia ainsi que de son adaptation pour la régulation du flux événementiel via le protocole décrit ci-dessus.

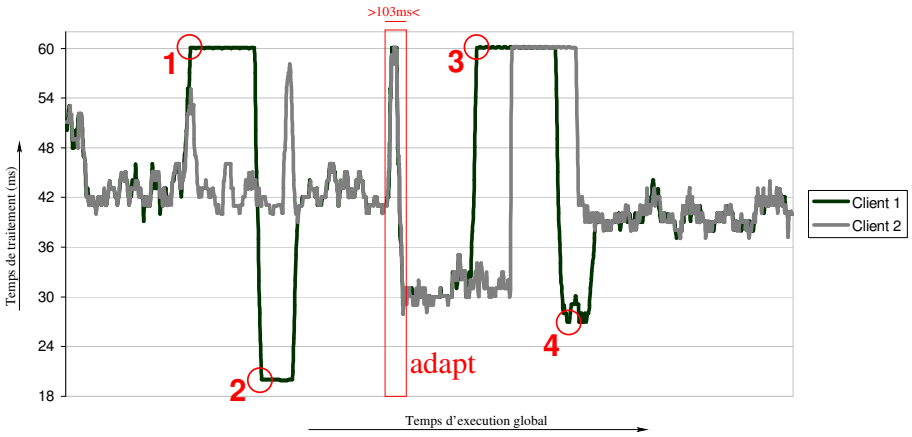


Figure 8. *Scénario d'exécution et adaptation du système multimédia*

L'échelle des ordonnées de ce graphe temporel représente l'intervalle de temps entre deux événements reçus en millisecondes. La courbe foncée correspond au client source d'anomalies, un autre client du système étant évalué par une courbe plus claire. Le repère (1) du graphe montre une première anomalie — déclenchée manuellement — avant adaptation du système. Le client en effet devient subitement beaucoup moins « productif » puisqu'il ne peut traiter les événements reçus en moins de 60 millisecondes[6]. Nous pouvons constater que dans le même temps, le deuxième client en profite et se voit, en quelque sorte, privilégié par le système puisque son temps de traitement diminue. Le repère (2) correspond à l'effet inverse, c'est-à-dire une augmentation drastique des performances du premier client. Nous observons en conséquence, un pic dans les temps de traitement du deuxième client.

La partie centrale du graphe (label *adapt*) correspond à la période d'adaptation du système ; c'est-à-dire à l'application manuelle (invocation de la fonctionnalité `sync`) du protocole de synchronisation. Nous pouvons voir l'impact de cette adaptation sur le système puisque les deux clients voient leurs performances diminuer sensiblement (sans pour autant s'arrêter de fonctionner). Mais, comme le montre la table 1, cette phase représente une durée d'environ 100 millisecondes du point de vue client, ce qui est très court puisque de l'ordre de grandeur du traitement de 2 ou 3 événements. La table 1 montre également que l'adaptation de ce système est nettement (de l'ordre de dix-huit fois) moins coûteuse que son déploiement. Le repère (3) reproduit l'anomalie du repère (1) mais nous pouvons constater, cette fois-ci, que quelques 150ms plus tard, le deuxième client a « rejoint » le niveau du client source d'anomalie. Nous sommes ici clairement dans le cadre d'une sous-utilisation du serveur, c'est pourquoi nous tentons

6. Cette anomalie est dirigée « à la main » par un opérateur extérieur. Durant l'expérience, l'intervalle nécessaire est monté jusqu'à plus de 110 millisecondes mais nous avons plafonné la courbe à 60 ms pour conserver les valeurs les plus significatives.

au repère (4) de forcer une surcharge du système en « accélérant » le premier client. Nous pouvons voir alors qu'en une très courte période (de l'ordre de 100ms), les deux clients finissent par se « retrouver » au niveau de performance qui correspond, du point de vue du serveur, au meilleur rendement partagé pour le système : la synchronization est active et autonome.

Operation	**Durée**
Déploiement de l'application	9934 ms
Temps d'adaptation (global)	541 ms
Temps d'adaptation (client)	103 ms
Temps de traitement moyen	42 ms

Tableau 1. *Mesures d'impact sur le système*

Le protocole ici décrit utilise une heuristique simpliste basée sur un indice immédiat de temps de traitement qui pourrait être associé à des indices en moyenne plus révélateurs du comportement du composant régulé sur le long terme. Mais encore une fois, les rôles Comet nous semblent particulièrement adaptés pour mettre en œuvre ces protocoles étendus de synchronisation de serveur de façon très simple. On peut même augmenter cette régulation non destructrice par une variante autorisant un filtrage des événements entrants pour réguler le flux. On pense alors tout naturellement aux rôles filtres pour prendre en charge cette spécialisation de notre protocole de synchronisation.

5. Vérifications et contrôle

Les *contrats rôle/composant* qui régissent les possibilités d'adaptation dynamique se décomposent en trois sections complémentaires : contrat de *typage*, d'*accès interne* et de *sécurité*. Notons que ces trois catégories de vérification sont complémentaires et que les règles de sécurité peuvent compléter les règles de typage et d'accès pour en restreindre l'applicabilité.

Le contrat de typage rôle/composant repose sur le système de vérification des types mis en œuvre dans la plate-forme Comet. D'un point de vue formel, ce contrat stipule qu'un rôle R de type $\langle R_{in}, R_{out}, R_{pre}, R_{post}, R_{filter} \rangle$ pourra être assigné dynamiquement à un composant C de type $\langle C_{in}, C_{out} \rangle$ si, et seulement si :

$$\forall t \in R_{pre} \cup R_{post} \cup R_{filter},\ \exists t' \in C_{in},\ t' \sim t$$

Ceci signifie que les types des événements interceptés par le rôle (via des crochets ou des filtres) doivent être « admissibles » par le composant[7].

7. La relation $t' \sim t$ exprime la comparabilité entre les deux événements, l'un étant sous-type de l'autre ($t' \preceq t$) ou vice-versa ($t' \succeq t$).

L'établissement du contrat de typage se fait, dans de larges mesures, par analyse statique du code source d'un rôle. Prenons l'exemple du rôle d'interception décrit en section 3. Le contrat de typage généré pour ce rôle correspond au document XML suivant :

```
<role type="InterceptorRole">
 <type>
  <prehook>Request</prehook>
  <out>Request</out>
 </type></role>
```

Cette définition doit être comparée à des propriétés liées aux composants hôtes éventuels. Le contrat de typage côté composant est ainsi généré de la même façon systématique. Par exemple, à partir de la définition des serveurs multimédias, nous obtenons :

```
<component type="MMServer">
 <type><in>MMRequest</in><out>MMEvent</out></type>
</component>
```

Il est évident que les deux parties du contrat de typage correspondent bien. En effet, le serveur reçoit des événements de type $MMRequest$ qui sont bien compatibles avec son super-type, $Request$, intercepté par le rôle (i.e. $MMRequest \preceq Request$).

L'assignation d'un rôle fonctionnel correspond en outre à une modification du contrat de typage pour le composant. Sur l'exemple précédent, après assignation d'un rôle d'interception, le serveur dispose d'un nouveau type sortant : $Request$. D'un point de vue plus formel, si l'on assigne à un composant de type $\langle C_{in}, C_{out}\rangle$ un rôle de type $\langle R_{in,} R_{out,} \ldots\rangle$ alors le type du composant deviendra : $\langle C_{in} \cup R_{in},\ C_{out} \cup R_{out}\rangle$. En cas de besoin, cette modification du type du composant peut être contrainte en dressant une liste des types potentiellement *augmentables* $C_{up_{in}}$ et $C_{up_{out}}$ ainsi qu'une liste de types « interdits » à toute mise à jour $C_{nup_{in}}$ et $C_{nup_{out}}$. Si par exemple $C_{up_{in}} = \{t_1, t_2\}$, seuls des sous-types de t_1 et t_2, types augmentables de C, pourront servir à la modification du type entrant du composant. De plus, si l'on pose $C_{nup_{out}} = \{t_3\}$ (avec $t_3 \prec t_2$) alors on interdit les sous-types de t_3 (mais pas les autres sous-types de t_2). Au niveau de la syntaxe concrète (toujours en XML), ceci se note :

```
<component type="Dummy">
 <type> ...
  <update><only>T1 T2</only><except>T3</except>
  </update> ...
 </type> ... </component>
```

En termes formels, ces vérifications supplémentaires (uniquement en cas de mise à jour du type du composant) se notent :

$$\begin{cases} C_{up_{in}} = \emptyset \vee \forall t \in R_{in}, \exists t' \in C_{up_{in}}, t \preceq t' \\ C_{up_{out}} = \emptyset \vee \forall t \in R_{out}, \exists t' \in C_{up_{out}}, t \preceq t' \\ \forall t \in R_{in}, \neg\exists t' \in C_{nup_{in}}, t \preceq t' \\ \forall t \in R_{out}, \neg\exists t' \in C_{nup_{out}}, t \preceq t' \end{cases}$$

Ces négociations peuvent conduire à d'éventuels *conflits de typage*. En effet, différents rôles assignés peuvent intercepter les mêmes types d'événements ou bien encore des événements déjà pris en compte au niveau du composant lui-même. Pour résoudre ces conflits potentiels, une heuristique de choix est proposée au niveau du composant hôte. En ce qui concerne les fonctionnalités, par défaut, la priorité va toujours au bloc le plus récemment défini et assigné au composant. L'activation est unique pour le couple fonctionnalité de composant/rôle fonctionnel. Ceci correspond à l'idée qu'un rôle doit être en mesure de substituer à une précédente version d'une nouvelle fonctionnalité pour un type donné. Les filtres et crochets sont en revanche activés de façon multiples. Les seuls conflits résultants étant d'ordre temporel puisqu'il faut choisir quel crochet et/ou quel filtre sera exécuté en priorité. Pour les filtres, nous opérons par ordre croissant de moment d'assignation (le filtre le plus ancien activé en premier), ce qui permet la réalisation de politiques de filtrage composées. Les crochets sont pour leur part triés comme les fonctionnalités (ordre croissant) en entrée, mais dans l'ordre inverse en sortie. Cette stratégie correspond à l'usage courant des primitives à effet de bord : acquisition de ressource, utilisation, puis mise à disposition.

Les contrats d'accès interne définissent les modes d'interaction entre les composants et leurs sous-composants internes, ces derniers pouvant être *intrusifs* ou *non-intrusifs* de ce point de vue. Par défaut, un rôle intrusif peut accéder à l'identité (et donc aux méthodes et à l'état interne) de son « sur-component » par l'intermédiaire de la référence `outer` (cf. rôle `InspectorRole` en section 3). Un rôle non-intrusif, en revanche, est « aveugle » et ne dispose donc pas de cette référence. Il est clair que les rôles intrusifs en terme d'accès interne sortent du cadre du contrat de typage et nécessitent une nouvelle forme de vérification : le contrat d'accès.

La première forme d'intrusion comportementale réside dans la possibilité pour un rôle d'accéder à tout ou partie de l'état interne des composants susceptibles de lui servir d'hôte. La partie rôle du contrat d'accès doit donner les informations les plus précises possibles concernant les champs ou attributs (dénommés *slots* au niveau du contrat) requis. Les informations nécessaires pour l'autorisation d'accès concernent : (1) le type d'attribut : nom de type ou de classe Java associé, (2) le nom du champ : identificateur Java, et (3) le mode d'accès : `readwrite`, `read` ou `write`. Prenons l'exemple du rôle `InspectorRole` dont les contrats de typage et d'accès (construction <access>) sont les suivants :

```
<role type="InspectorRole">
  <type> <in>InspectReq</in>
         <out>InspectRep</out></type>
  <access><all-slots mode="read"></access>
</role>
```

Puisqu'il s'agit d'un rôle d'inspection générique, l'accès est demandé ici pour la lecture de l'ensemble des attributs du composant hôte (déclaration `<all-slots>`). Pour satisfaire ce contrat côté composant hôte, il faut fournir l'accès à tout ou partie des attributs. Nous allons ici nous limiter au champ `_frameRate` qui est effectivement utilisé dans notre exemple concret de la section 3 :

```
<component type="VideoServer">
  <access><slot type="float" name="_frameRate" mode="read"></access>
</component>
```

La construction `<slot>` décrit l'accès à un unique attribut de nom `_frameRate`, de type `float` avec un mode d'accès en lecture seule (`read`). Ainsi, la requête d'inspection pour ce champ fonctionnera mais toute autre inspection sera refusée au niveau opérationnel. La deuxième possibilité d'accès, en supplément de l'état interne, concerne bien évidemment les invocations de méthodes internes du composant hôte. Pour le protocole de synchronisation (section 4), l'accès à la méthode statique `sleep` de la classe `Thread` est autorisée de la façon suivante :

```
<role type="FlowRegulator">
  <access>
    <method static="true" class="java.lang.Thread" name="sleep"/>
  </access>
</role>
```

Pour répondre à ce contrat d'accès du côté du composant, il est encore une fois possible d'autoriser l'accès partiel ou complet (construction `<all-methods>`) aux fonctionnalités internes.

Le contrat de sécurité représente la troisième catégorie de contrat rôle-composant. Les règles de sécurité qu'il propose sont nécessaires pour prévenir toute modification impromptue des systèmes lorsque les autres modes de contrôles ne suffisent pas. Par exemple, il semble important de ne pas autoriser « n'importe qui » à mettre en place le service de régulation de flux, étant donné l'effet de bord temporel potentiellement dangereux (imaginons un `sleep` infini !). Pour mettre en œuvre ce contrat de sécurité, nous nous reposons entièrement sur l'environnement Java. Les fonctionalités offertes permettent le support de la sécurité en terme d'authentification, de restriction de domaine, d'encodage de communication ou de certification de chargement dynamique [Sun 02].

6. Travaux connexes

De nombreux travaux scientifiques proposent la mise en œuvre de modifications de systèmes — distribués ou non — pendant leur exécution [LIU 00]. Cependant, dans le cadre des plate-formes intergicielles, la majeure partie de ces travaux se concentrent sur la reconfiguration structurelle du système et notamment le *remplacement à chaud* de composant. Dans X-RMI [XUE 02], par exemple, le remplacement à chaud d'objets serveurs RMI est proposée. La contribution est ici double car non seulement la pré-

servation de l'existant est assurée mais de plus les performances sont tout à fait appréciables étant donné l'étendue des modifications prises en charge. D'autres exemples similaires comme [JAC 01] ou [EMS 03], font état de résultats comparables. Cependant, il nous semble difficile d'identifier clairement le rapport entre la nature conceptuelle des besoins en terme d'adaptation avec la délimitation physique de la cible potentielle du changement : un composant particulier. Dans cet article, nous nous plaçons par exemple dans le cadre de composants qui assurent *parfaitement* leur propos — nous ne voulons surtout pas les remplacer — mais le système, lui, nécessite une modification.

L'approche de Drastic [EVA 99] est probablement plus proche, dans ses motivations, avec notre vision de l'adaptation dynamique des applications car la barrière d'abstraction entre spécification et opérationnalisation nous y semble plus tangible. Lors d'une modification du système, l'analyse de *contrats* entre objets distribués permet de délimiter des *zones* physiques de modification qui doivent être « endormies » pendant la mise en œuvre des modifications. Pendant ce temps, tout élément extérieur à ces zones identifiées poursuit son fonctionnement « comme si de rien n'était ». La principale différence dans notre cas concerne la granularité des changements qui ne nécessitent que des interruptions « microscopiques » du point de vue externe des systèmes, puisque la modification est confinée dans un composant.

Cette focalisation à un niveau de granularité très fin — et qui pourtant permet des modifications d'envergures en termes opérationnels — repose essentiellement selon nous sur la *similitude* entre le niveau interne ou micro-structurel des composants et leur niveau externe — la macro-structure — des systèmes distribués. L'identification de cette micro-architecture est fortement inspirée du framework CodA [MCA 95] de Jeff Mc Affer dont la cible applicative est différente puisque concernant l'élaboration de modèles objets variés pour le langage Smalltalk. L'avénement des modèles à base de composants permet, selon nous, de « démystifier » le côté quelque peu « magique » des procédés réflexifs. Les composants Comet et les sous-composants internes qui en régissent le fonctionnement ont, finalement, des propriétés très similaires lorsqu'on les observe en termes linguistiques (syntaxiquement similaires, sémantiquement proches).

Les intergiciels réflexifs, comme [PAR 00], proposent également la mise en œuvre de modifications dynamiques étendues. Cependant, notre travail "sous-jacent" de formalisation [PES 03] nous a conduit vers des chemins plus « directs » en encadrant les mécanismes evolutifs par des abstractions linguistiques, elles-même associées à une sémantique formelle. Ceci rejoint quelque peu certains concepts de la programmation orientée aspect [KIC 97] qui associe aux besoins de séparation des préoccupations, des constructions linguistiques, les aspects, et des outils pour les décrire et les composer. Mais si un parallèle avec des travaux AOP, comme par exemple [PAW 01], est possible d'un point de vue langage, il faut préciser notre attachement aux aspects systèmes, alors que l'AOP se focalise généralement sur des concepts de génie logiciel. Bien sûr, cette frontière est relativement floue puisque des travaux comme D [LOP 97] rejoignent notre point de vue point de vue système. Il est clair, cependant, que la ca-

tégorie que les mécanismes proposés dans Comet sont de type “weaving” dynamique, relativement nouvelle [POP 03].

Il est notable que le modèle de composant que nous proposons est différent des propositions industrielles comme Corba CCM [GRO 02] ou Java RMI [Sun 01]. De ce fait, il est plutôt inspiré des langages de définition d’architectures (ADL) et notamment [LUC 96]. Mais il est tout à fait possible d’employer les mécanismes adaptatifs de Comet pour modifier des composants RMI (ou CORBA mais implantés en Java). Des composants hybrides peuvent ainsi être développés, qui communiquent à la fois par événements asynchrones et par invocations synchrones de méthodes. En revanche, nous avons préféré situer notre implantation au dessous des sockets TCP et UDP pour une raison simple : les performances. D’un point de vue plus prospectif, l’intégration de constructions et mécanismes visant spécifiquement l’adaptation dynamique nous semble nécessaire à moyen terme, notamment dans le cadre de nouvelles applications distribuées comme le *Peer-to-peer* [ORA 01] ou le *grid computing* [BER 03]. Nous pensons que les besoins dans ces domaines dépassent le cadre des modèles de communication et implantations associées, ScalAgent/Joram [Obj 03] pour les Java Messaging Specification ou le modèle sources/puits des Composants Corba [GRO 02]. Bien sûr, ces infrastructures sont une étape importante mais nous pensons qu’une réflexion sur le long terme est également nécessaire. Cette réflexion nécessite selon nous une expérimentation préliminaire des nouvelles applications adaptatives ; nécessité qui motive nos travaux actuels.

Les composants Comet sont, au même titre que les acteurs [AGH 86], des entités logicielles communiquant de façon asynchrone. De plus, les concepts de protocole et de rôle que nous employons ont été introduits par le framework DIL [STU 96], basé sur les acteurs. Ceci confirme notre attachement au modèle de calcul par acteurs. D’importantes différences sont cependant notables, comme par exemple la mise en œuvre de l’extraction de la relation de couplage entre les composants et l’intervention connexe d’un système de typage omni-présent. Certains de ces aspects sont discutés dans [AST 99] où l’on comprend la différence entre un travail inspiré par les acteurs comme le notre et une approche prenant le modèle de calcul très spécifique des acteurs « au pied de la lettre ». Selon nous, certains traits de ce modèle comme l’identification acteur/activité nous semblent quelque peu discutables.

7. Conclusion

La plate-forme intergicielle Comet que nous avons conçue et implémentée propose un mode de développement selon nous novateur car complètement *incrémental*. D’un point de vue structurel (plus largement discuté dans [PES 02a]), la possibilité de modifier dynamiquement l’architecture des systèmes est rendue (1) *possible* par l’extraction de la relation de couplage entre composants et (2) *maîtrisable* par la sémantique sous-jacente, asynchrone et réactive. Dans cet article, nous montrons qu’en proposant une micro-architecture calquée sur ces principes — et donc elle aussi de nature composable — nous permettons la spécification et l’opérationalisation contrô-

lée d'adaptations dynamiques de fine granularité. Toute la « puissance » du système repose selon nous sur cette adéquation, voire similitude, entre les points de vue macro (externe) et micro (interne).

Nous décrivons nos adaptations sous la forme de *protocoles* autonomes et de *rôles* associés. Ces abstractions préfigurent selon nous ce que les langages et outils intergiciels *évolutifs* de demain devront proposer. En particulier, nous pensons que les trois catégories de rôles proposées forment un modèle très *expressif* pour l'adaptation logicielle de grain fin. Les rôles fonctionnels permettent par exemple la mise en place des systèmes de *plugins*, dans une approche plus structurée que la plupart des systèmes comparables[8]. De fait, il n'existe pratiquement aucune limite sur la portée de tels rôles, outre le fait qu'il faut proposer des mécanismes opérationnels pour vérifier l'impact de leur assignation sur les composants du système. Les rôles crochets permettent pour leur part de modifier l'interprétation d'événements déjà pris en compte par un composant. Le rôle d'interception décrit en section 3.2 montre leur intérêt en particulier lorsqu'il s'agit de rendre compte au niveau extérieur (émission d'un événement) d'un phénomène local (réception de l'événement « crocheté »). Le filtrage permet finalement d'avoir la main-mise sur le contrôle proprement dit des applications (cf. rôle « pas-à-pas » de la section 3.3), toujours selon un mode incrémental et dynamique.

Mais notre discours ne saurait être tenu sans les validations expérimentales, nombreuses, que nous avons menées. Dans notre exemple de la section 4, nous avons montré que non seulement, l'adaptation fine mais profonde du système était possible sans aucune interruption dans le fonctionnement de ce dernier et surtout, que l'impact sur ce même système en terme de performance était peu perceptible globalement. Il est cependant important de garder à l'esprit que l'évolution dynamique des systèmes reste très difficile à envisager *en toute généralité*. Dans notre exemple, nous « maîtrisons » nos modifications car elles se situent à notre niveau de compétence : la couche intergicielle. Si les rôles permettent d'autres modifications plus « arbitraires », les seules garanties que nous proposons dans ce cas sur les éventuelles conséquences opérationnelles restent à un niveau principalement « système ». Nous envisageons en conséquence la mise en œuvre d'une variante pessimiste des méthodes d'accès proposées précédemment : le cloisonnement ou *sand-boxing* transactionnel des rôles [SEL 96].

En parallèle, nous élaborons dans l'immédiat des extensions de la plate-forme avec entre autre application la mise en œuvre de politiques de filtrage à événements composites, c'est-à-dire des compositions temporelles d'événements atomiques. Ceci offrirait, en quelque sorte, une relation de couplage et un système de typage associé d'ordre supérieur dans lesquels la composante temporelle perdrait sa caractéristique implicite. Mais notre effort actuel le plus important concerne l'intégration de la mobilité dans la plateforme. Nous sommes également convaincu de l'intéret du modèle à connection dynamique que nous proposons dans le cadre de la mobilité, notamment pour les mécamismes de relayage des événements lors des phases de migration.

8. Les rôles et les protocoles fournissent un cadre de plus haut-niveau que le respect d'une interface, par exemple de bibliothèque utilitaire pour le langage sous-jacent.

8. Bibliographie

[AGH 86] AGHA G., *Actors : A Model of Concurrent Computation in Distributed Systems.*, Series in Artificial Intelligence, MIT Press, 1986.

[AST 99] ASTLEY M., « Customization and Composition of Distributed Objects : Policy Management in Distributed Software Architectures », PhD thesis, University of Illinois at Urbana-Champaign, Mai 1999.

[BER 03] BERNAM F., HEY A. J., FOX G., Eds., *Grid Computing : Making the Global Infrastructure a Reality*, John Wiley and Sons, 2003.

[com03] « The Comet Middleware Project », `http://www.non-gnu.org/comet`, 2003.

[EMS 03] EMSELLEM D., CHARFI A., RIVEILL M., « Dynamic Component Composition in .NET », *Proceedings of ECOOP'2003 Workshop on .NET : The Programmer's Perspective*, 2003.

[EVA 99] EVANS H., DICKMAN P., « Zones, Contracts and Absorbing Changes : An Approach to Software Evolution », *Proceedings of OOPSLA'99*, ACM Press, Novembre 1999.

[FIT 99] FITZPATRICK T., « Open Component-Oriented Multimedia Middleware for Adaptive Distributed Applications », PhD thesis, Computing Department, Lancaster University, Septembre 1999.

[GEI 01] GEIHS K., « Middleware Challenges Ahead », *IEEE Computer*, vol. 34, n° 6, 2001.

[GRO 02] OBJECT MANAGEMENT GROUP, « Corba Component Model v3.0 », `http://www.omg.org`, 2002.

[JAC 01] JACQUES MALENFANT MARIA-TERESA SEGARRA F. A., « Dynamic Adaptability : The Molène Experiment », *Proceedings of Reflection 2001*, vol. LNCS 2192, Springer Verlag, Septembre 2001.

[KIC 97] KICZALES G. et al., « *Aspect-Oriented Programming* », *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer Verlag, Juin 1997.

[LIU 00] LIU X., YANG H., Eds., *International Symposium on Principles of Software Evolution (ISPSE 2000), Kanazawa, Japan*, IEEE computer society, Novembre 2000.

[LOP 97] LOPES C. V., « D : A Language Framework for Distributed Programming », PhD thesis, Northeastern University, Boston, 1997.

[LUC 96] LUCKHAM D. C., « *Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events* », *DIMACS Partial Order Methods Workshop*, vol. IV, 1996.

[MCA 95] MCAFFER J., « *Metalevel Programming with Coda* », *Proceedings of the European Conference on Object-Oriented Computing (ECOOP)*, LNCS 952, Springer Verlag, Août 1995, p. 190-214.

[MEY 00] MEYER R., « State of the Art of Distributed Event Models », rapport, 2000, University of Dublin, Trinity College.

[Obj 01] OBJECT MANAGEMENT GROUP, « Common Object Request Broker Architecture (CORBA) 2.6 Specifications », `http://www.corba.org`, 2001.

[Obj 03] OBJECTWEB CONSORTIUM, « Joram 3.4 », `http://joram.objectweb.org`, 2003.

[ORA 01] ORAM A., Ed., *Peer-to-peer : Harnessing the Power of Disruptive Technologies*, O'Reilly and associates, 2001.

[PAR 00] PARLAVANTZAS N., COULSON G., CLARKE M., BLAIR G., « Towards a Reflective Component-based Middleware Architecture », *Proceedings of Workshop on Reflection and Metalevel Architectures*, Juin 2000.

[PAW 01] PAWLAK R., SEINTURIER L., DUCHIEN L., FLORIN G., « JAC : A Flexible Framework for AOP in Java », *Proceedings of Reflection 2001*, vol. LNCS 2192, Springer Verlag, Septembre 2001.

[PES 00] PESCHANSKI F., « Comet : Architecture Réflexive pour la Construction d'Applications Adaptatives en Environement Concurrent et Réparti », *Proceedings of Langages et Modèles Objets*, Editions Hermès, Janvier 2000.

[PES 02a] PESCHANSKI F., « Composition et Adaptation Dynamiques de Systèmes Distribués. Une Approche à base de Composants Asynchrones Typés », PhD thesis, Université Pierre et Marie Curie Paris VI, Juin 2002.

[PES 02b] PESCHANSKI F., « A Versatile Event-based Communication Model for Generic Distributed Interactions », *Proceedings of DEBS'02 (ICDCS International Workshop on Distributed Event-based Systems)*, IEEE, Juillet 2002.

[PES 03] PESCHANSKI F., JULIEN D., « When Concurrent Control Meets Functional Requirements or Z+Petri Nets », *Proceedings of ZB2003*, Springer Verlag, 2003, à paraître.

[POP 03] POPOVICI A., ALONSO G., GROSS T., « Just-In-Time Aspects : Efficient Dynamic Weaving for Java », *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, ACM Press, 2003.

[SEL 96] SELTZER M., ENDO Y., SMALL C., SMITH K., « Dealing with Disaster : Surviving Misbehaved Kernel Extensions », *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, USENIX, 1996.

[STU 96] STURMAN D. C., « Modular Specification of Interaction Policies in Distributed Computing », PhD thesis, University of Illinois at Urbana-Champaign, Mai 1996.

[Sun 01] SUN MICROSYSTEMS, « Java Remove Method Invocation (RMI) 1.4 Specifications », `http://java.sun.com/rmi`, 2001.

[Sun 02] SUN MICROSYSTEMS, « Java Security », `http://java.sun.com/security`, 2002.

[XUE 02] XUEJUN C., « Extending RMI to Support Dynamic Reconfiguration of Distributed Systems », *Proceedings of ICDCS'02*, IEEE computer society, Juillet 2002.