Comet

Architecture Réflexive à Base de Composants pour la Construction d'Applications Concurrentes et Réparties

Frédéric Peschanski

Laboratoire d'Informatique de Paris 6 (LIP6)
8, rue du Capitaine Scott
75015 Paris

Frederic.Peschanski@lip6.fr

RÉSUMÉ. *Les systèmes d'information actuels rapprochent les problématiques de génie logiciel et de systèmes distribués. Nous proposons dans cet article une solution mixte, à la fois réflexive et basée sur les composants pour répondre aux problèmes liés à la conception de systèmes distribués hautement dynamiques et à large échelle. Notre architecture Comet met en œuvre des composants dont le contrôle est exprimé indépendamment des fonctionnalités. Ceci permet de réutiliser séparément les modèles de contrôle et les propriétés fonctionnelles des composants. A moyen terme, nous pensons que les applications seront également adaptatives, leur sémantique ne sera pas intégralement prédéfinie avant leur déploiement. Pour cette raison, les protocoles et rôles Comet autorisent les modifications contrôlées, à la fois des composants mais également des architectures elles-mêmes, qui deviennent ainsi dynamiques. Le framework Comet/J met en œuvre ces différents concepts dans une approche objet qui s'intègre efficacement dans l'environnement Java.*

ABSTRACT. *Today's information systems bring closer both software engineering and distributed systems concerns. We propose in this paper an hybrid solution, at the same time reflective and based on components, in order to address the design of large-scale, highly dynamic, distributed systems. Our Comet architecture implements components whose both control and functionalities are expressed independantly. This makes it possible to reuse separately their control models and their functional properties. With medium term, we think that future applications will be also adaptative, their semantics will not be completly preset before their deployment. Therefore, the Comet roles and protocols authorize the controlled modifications, at the same time of the components but also of the architectures themselves, which become thus dynamic. The Comet/J framework implements these concepts in an object-oriented approach which integrates smoothly into the Java environment.*

MOTS-CLÉS : *Composants Distribués, Réflexivité, Architectures Logicielles, Adaptabilité Dynamique.*

KEYWORDS: *Distributed Components, Reflection, Software Architectures, Dynamic Adaptability.*

1. Introduction

Les systèmes informatiques modernes entraînent, de par leur complexité croissante, la convergence de deux domaines autrefois séparés : génie logiciel et systèmes distribués. De fait, la vision centralisée des ordinateurs se révèle aujourd'hui inadéquate pour aborder la résolution de nombreux problèmes. L'univers centralisé et séquentiel semble pourtant sous-jacent des modèles objets courants [1] selon les considérations suivantes :

Contrôle implicite : l'immersion en environnement concurrent et distribué complexifie le contrôle des objets qui ne peut plus être intégralement délégué au support d'implémentation.

Invocations synchrones : le contrôle des objets repose invariablement sur l'invocation synchrone de méthodes. La distribution impose *a contrario* un fonctionnement asynchrone et adaptable (selon les sémantiques de communication sous-jacentes).

Anomalies d'héritage : l'explicitation d'une partie du contrôle des objets entre en conflit avec le principe de réutilisabilité par héritage. Le cas de la réutilisation de modèles de synchronisation est par exemple bien connu [MAT 93]. Les solutions passent alors invariablement par la remodularisation des programmes, intrusions peu souhaitables au niveau de la conception. En conséquence, le fossé entre problématique liée au domaine et implantation se creuse.

Trois familles de solutions pour résoudre les problèmes liés au caractère implicite du contrôle des objets sont proposées dans [BRI 98] :

L'approche applicative : la première solution envisageable concerne la construction de bibliothèques d'objets dont le domaine d'application est la programmation concurrente et répartie. Cette approche permet de ne pas remettre en question les systèmes et langages "traditionnels". Le principal problème émerge de la multiplication des objets et des rôles qui leurs sont attribués. Le programmeur doit alors manipuler des hiérarchies d'objets présentées de façon orthogonale mais sémantiquement liées.

L'approche intégrée : objets actifs, synchronisés et répartis sont autant de tentatives d'intégration des mécanismes de gestion de la concurrence et/ou de la répartition dans les langages orientés objet. Cette approche, attractive au premier abord, renforce le couplage peu souhaitable entre contrôle et fonctionnalités.

L'approche réflexive : l'idée de cette approche - celle que nous employons - est d'extraire et de rendre manipulable une partie du contrôle des objets. Le programme décrit les fonctionnalités de base du système et les méta-programmes expriment au moins en partie son contexte d'exécution. Il s'agit d'un pont possible entre les deux approches précédentes : réification par voie applicative puis ré-intégration au niveau système.

1. Le modèle objet ABCM/1[YON 89] représente une exception notable.

D'importants problèmes d'ordre structurel peuvent également être isolés dans l'approche objet :

Problème de granularité : le besoin de séparation entre unités d'implémentation, généralement de grain fin, et unités logiques de décomposition de problème, de grain plus important, est grandissant.

Couplage potentiellement fort : les références explicites sont possibles dans l'expression des fonctionnalités des objets. Ce couplage potentiellement fort empêche dans de nombreux cas d'assurer modularité et réutilisabilité des programmes.

Les approches à base de composants permettent à la fois de conserver le caractère neutre des objets, d'en élever le niveau de granularité et d'extraire la relation de couplage, au travers des connexions. Des modèles de composants industriels, comme les Enterprise JavaBeans [Sun 99] ou les Corba components [Obj 99] pour les applications distribuées sont en cours d'élaboration. Cependant, ces modèles font peu de distinction entre spécification et mise en œuvre, ce qui empèche par exemple l'élaboration d'un cadre réflexif associé. Notre architecture Comet propose donc un modèle de composant à la fois proche dans les principes, c'est à dire résultant de travaux dans le domaine des architectures logicielles [GAR 93], mais cependant différent puisque de nature réflexive.

Notre architecture explore également de façon approfondie la problématique des applications adaptatives. La propriété d'adaptation autorise les modifications dynamiques des programmes, pendant leur exécution. Il s'agit à notre avis d'une évolution naturelle et attendue des applications informatiques. Les problèmes, notamment sémantiques, posés par ces modifications dynamiques sont de taille. Il nous faut donc élaborer, au sein de Comet, un cadre strict à ces nouvelles fonctionnalités adaptatives.

De façon similaire à Rapide [LUC 96], Comet est basée sur la notion de composants typés, réunis au sein d'un système de communication basé sur les événements. Nous pouvons dans ce sens cibler différents types de systèmes distribués, sans pour autant poser des hypothèses trop fortes concernant le modèle de calcul sous-jacent ou le langage de programmation utilisé. Notre premier prototype d'implémentation, Comet/J est une instanciation basée sur Java de l'architecture Comet. Ce framework opérationnel est utilisé dans cet article pour décrire quelques exemples d'utilisation, dans le domaine de la résistance aux pannes.

La section 2 décrit plus en détail le modèle de composant utilisé. L'architecture de niveau méta associée est ensuite définie en section 3. Les protocoles Comet, entités logicielles indépendantes, sont introduits en section 4. Nous abordons ensuite, en section 5, les travaux connexes à notre approche avant de conclure et de préciser nos objectifs à plus ou moins long terme, en section 6.

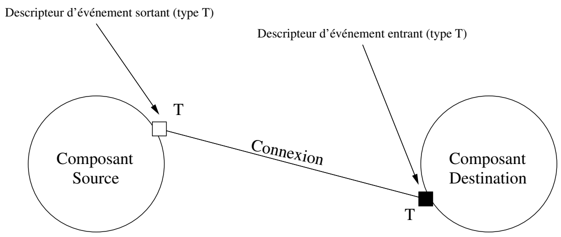


Figure 1. *Modèle événementiel de l'architecture Comet*

2. Modèle de composant

Le domaine des architectures logicielles introduit plusieurs concepts importants : composants, connecteurs et connexions. Les composants sont les entités logicielles de premier niveau, briques de base des architectures. Les connexions permettent de relier les composants entre eux, par le biais de leurs connecteurs. Nous proposons dans Comet un modèle événementiel pour représenter connecteurs et connexions. Dans ce modèle, les événements correspondent à des messages typés que peuvent s'échanger les composants. Les connecteurs sont représentés par des descripteurs d'événements qu'un composant peut recevoir ou émettre. Enfin, une connexion relie un composant source à un composant destination pour un type d'événement donné (cf. figure 1). Nous reviendrons sur cette notion de connexion en section 4, lorsque nous présenterons les protocoles Comet.

Dans notre architecture, les composants sont considérés comme unité de répartition/déploiement[2]. Ce sont donc des entités de granularité potentiellement importante, les objets étant considérés dans cette optique comme des entités distinctes, de grain plus fin. Les composants Comet disposent, au niveau base, des caractéristiques suivantes :

– Un identificateur ID, unique au sein du système,
– un état interne e,
– un ensemble de fonctionnalités $M = \{m_i\}$,
– un ensemble de descripteurs d'événements entrants $E = \{In_j\}$,
– un ensemble de descripteurs d'événements sortants $S = \{Out_k\}$,
– des ressources associées $R = \{Res_l\}$,

Il est possible de représenter ces composants dans des environnements objets traditionnels. Cependant, un descripteur de composant sera généralement nécessaire, notamment pour détailler le type du composant et les ressources associées (images, sons, textes, etc.). L'environnement Java sur lequel nous basons notre prototype Comet/J

2. Les acteurs, en comparaison, représentent plutôt des unités de concurrence, de grain fin.

permet toutefois, grâce aux interfaces, l'intégration simple du typage des composants. Voici en guise d'exemple un composant serveur minimaliste :

```
class Server extends comet.DefaultComponent
             implements ClientRequestEventReceiver,
                        ServerReplyEventSender {
  void ClientRequest(ClientRequestEvent cre) {
    String result = manageRequest(cre);
    ServerReplyEvent sre = new ServerReplyEvent(result);
    sendEvent(sre);
  }
}
```

Ce composant reçoit et traite des requêtes de clients (événements `ClientRequest-Event`), puis génère des réponses (événements `ServerReplyEvent`). Il est important de signaler que l'émission de l'événement (primitive `sendEvent`) n'identifie pas le destinataire puisque c'est la connexion qui se charge d'associer émetteur et récepteur(s).

Un descripteur d'événement entrant (comme `ClientRequestEventReceiver`) est pour sa part composé des informations suivantes :

– Un identificateur unique ID,
– un type T d'événement associé [3],
– une fonctionnalité attachée m_l.

Dans Comet/J, le descripteur d'événement entrant `ClientRequestEvent` est une interface Java :

```
interface ClientRequestEventReceiver
          extends comet.EventReceiver {
  public void ClientRequest(ClientRequestEvent cre);
}
```

Le lien entre le type d'événement entrant (`ClientRequestEvent`) et la méthode à exécuter dans le composant (`ClientRequest`) est donc décrit explicitement par l'interface descripteur de l'événement. En revanche, les descripteurs d'événements sortants n'ont pas de fonctionnalité attachée. Ceci se traduit en Java par une interface vide :

```
interface ServerReplyEventSender extends comet.EventSender {}
```

Les événements sont reliés dans une relation de sous-typage, permettant la subsomption [ABA 96]. Cette propriété autorise l'utilisation d'un événement de type T' dans un contexte où un événement de type T, super-type de T', est attendu. L'intérêt de la subsomption réside principalement dans l'assouplissement de la relation de cou-

3. Un événement sera généralement assimilé à un objet, avec un état interne et des méthodes.

plage entre composants. Ce modèle s'intègre parfaitement dans le système de typage de Java, nous pouvons donc associer événements et objets :

```
class ClientRequestEvent extends comet.Event {
  public ClientRequestEvent(String contents) {
    _contents = contents;
  }
 public String getContents() { return _contents; }
 private String _contents;
}
```

Cet événement de requête client gère un contenu représenté sous forme de chaîne de caractère.

Le typage des composants, défini au sens événementiel par le couple (E,S)[4], est également étendu à une relation de sous-typage. Ainsi, un événement de type T', sous-type de T, peut servir d'événement entrant pour un composant C_1 de type $(E = T \cup \tau_1, S)$. Un composant C_2, de type $(E, S = T'' \cup \tau_2)$, est alors partiellement compatible avec C_1 si T'' est sous-type, au sens large, de T. En d'autres termes, cela signifie qu'il est possible d'établir une connexion associée au type T'' entre les composants C_2 et C_1.

3. Niveaux sémantiques

Les composants Comet introduisent trois niveaux sémantiques causalement liés. Au **niveau base**, les développeurs peuvent exprimer les fonctionnalités statiques et prédéfinies du composant. Les fonctionnalités de contrôles localisés sont (au moins partiellement) réifiées au **niveau méta-comportement**. Ceci permet par exemple de fournir une vision centralisée et séquentielle au niveau base. Finalement, le **niveau méta-rôle** représente la couche de support des fonctionnalités adaptatives. La section 3.1 décrit le niveau méta-comportement puis la section 3.2 présente le niveau méta-rôle.

3.1. *Niveau méta-comportement*

Pour concevoir le niveau méta-comportement, nous avons décidé de décomposer l'ensemble des fonctionnalités relatives au cheminement des événements au sein d'un composant (cf. figure 2). Les fonctionnalités concernées sont les suivantes :

Receiving : Réception des événements entrants.

InQueuing : Mise en file d'attente de ces derniers.

InFetching : Extraction d'événement(s) dans le système de files d'attente.

4. Couple (types d'événement entrant, types d'événement sortant)

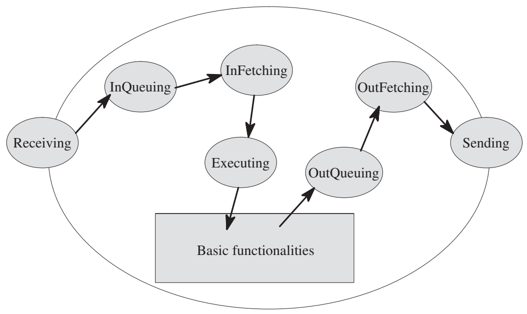


Figure 2. *Niveau méta-comportement*

Executing : Exécution d'une fonctionnalité de base associée.
OutQueuing : Mise en file d'attente d'un événement émis.
OutFetching : Extraction d'un événement sortant.
Sending : Envoi d'un événement sur la couche de communication.

En suivant l'approche de CodA [MCA 95], chacune de ces fonctionnalités de contrôle est réifiée sous forme de méta-objet. Nous présentons ci-dessous la sémantique par défaut et quelques spécialisations possibles pour chacun des différents méta-objets canoniques proposés :

MetaReceive : Arbitre choisissant un événement entrant dans les sources asynchrones du composant. Gestion des connexions (cf. section 4)
Spécialisations : adaptations à différentes sémantiques de communication.

MetaInQueue : file d'attente double : événements "normaux" et retours d'émissions synchrones, prioritaires.
Spécialisations : réseau complexe de files d'attente en entrée, gestion des files avec priorité.

MetaInFetch : choix de l'événement le plus ancien, communication au méta-objet *MetaExec*.
Spécialisations : synchronisation événementielle, heuristique de choix complexe, traitement d'événements spéciaux (synchrones, séquences atomiques, etc.).

MetaExec : Exécution dans l'activité courante de la méthode. *Wrappers* pour le niveau méta-rôle (cf. section 3.2).
Spécialisations : modèles de synchronisation interne liés aux méthodes (gardes, synchronization constraints, etc.). Sémantique d'exécution : activités multiples, propriétés d'ordonnancement (échéances temporelles).

MetaOutQueue : file d'attente simple et envoi "au plus tôt".
Spécialisations : émissions concurrentes, réordonnancement causal [5].

5. La perte de causalité peut par exemple résulter de l'emploi d'un système complexe de files d'attente en entrée.

MetaOutFetch : envoi immédiat de l'événement émis le plus ancien.
Spécialisations : heuristique de choix complexe, temporisation pour satisfaire la stabilité du système.

MetaSend : choix d'une sortie asynchrone et émission de l'événement. Gestion de la liste des destinataires.
Spécialisations : adaptations à différentes sémantiques de communication.

Nous séparons dans notre approche la nature des systèmes de files d'attente (méta-objets *MetaInQueue* et *MetaOutQueue*) et la gestion de ces derniers (*MetaInFetch* et *MetaOutFetch*). Cette vision "file d'attente" est notamment intéressante pour suppléer les discussions, habituellement basées sur la gestion des ressources, concernant la stabilité des sytèmes mis en œuvre. La réification du système d'émission des événements est principalement motivée par le coût potentiellement important des accès à la couche de communication en environnement distribué. Mais les traitements avant émission ne sont pas forcément liés aux performances (ex. restauration transparente d'un lien causal par une file de réordonnancement).

De façon comparable à CodA, l'introduction de nouveaux méta-objets est possible en fonction des besoins. Par exemple, la mise à disposition d'un méta-objet *MetaState*, gérant la sémantique d'accès à l'état interne, pourra se révéler utile dans le cadre de la migration[6] ou des systèmes transactionnels. La composition de ces différents méta-objets forme un modèle de comportement C. Dans le framework Comet/J, le modèle de comportement est la super-classe du composant. Par exemple, le composant `Server` défini précédemment hérite du modèle `DefaultComponent`.

Le niveau méta-comportement offre ainsi à l'architecture Comet une propriété de flexibilité nécessaire dans le cadre de systèmes complexes. Il est en effet possible de concevoir une large gamme de modèles de comportement par composition de méta-objets spécialisés. Plus important, l'intéropérabilité entre ces différentes stratégies de contrôles est possible, du fait de la nature asynchrone des communications mises en jeu.

3.2. *Niveau méta-rôle*

Les fonctionnalités adaptatives, dynamiques et non prédéfinies par les composants, sont gérées au niveau méta-rôle. De façon schématique, un rôle Comet réunit en son sein des fonctionnalités et des données (i.e. un état) permettant l'attribution d'une nouvelle propriété à un composant, déployé et opérationnel. Ce concept est généralement lié à la notion de protocole. Par exemple, on dira qu'un composant joue le rôle de serveur dans un protocole client/serveur (cf. section 4).

6. Pendant la phase de migration, il est possible d'augmenter la disponibilité du composant en rendant accessible à distance son état interne. Les accès à cet état interne doivent alors être sérialisés.

Un rôle est représenté au sein de Comet par un composant de niveau méta (i.e. méta-composant) proposant les caractéristiques suivantes :

– un état interne e_r,
– des fonctionnalités de rôle, $M_r = \{m_{r,i}\}$
– des descripteurs d'événements entrants $E_r = \{In_{r,k}\}$,
– des descripteurs d'événement sortants $S_r = \{Out_{r,j}\}$,
– des intercepteurs d'événements (*event hooks*) $H_r = \{Hook_{r,l}\}$,
– des ressources associées $R_r = \{Res_{r,m}\}$.

Les rôles sont structurellement assez similaires aux composants de niveau base (exception faite des intercepteurs d'événements, détaillés en section 4). Leur expression dans le framework Comet/J est en conséquence semblable :

```
public class RestoreServerRole extends comet.Role
             implements comet.RestoreServerEventReceiver,
                        comet.ValidateReplicaEventSender {
  public bool checkBase() {....}
  public void RestoreServer(RestoreServerEvent rse) {
    _base.restore(rse);
    sendEvent(new ValidateReplicaEvent());
  }
}
```

Ce rôle vise l'ajout dynamique d'une fonctionnalité de restauration à un serveur défaillant puis redémarré, à partir de requêtes sauvegardées sur un réplica. Une fois la restauration effectuée, le rôle émet un événement de validation pour les réplicas.

Le mécanisme de base mis en jeu au niveau méta-rôle concerne l'assignation dynamique d'un rôle, comme `RestoreServerRole`, à un composant pré-existant, déployé et opérationnel. L'assignation du rôle `RestoreServerRole` à un composant serveur est présentée en figure 3. Un intérêt fondamental de cette vision structurelle de l'adaptation dynamique concerne le contrôle de cohérence qui peut ainsi, en partie, être délégué au système de typage sous-jacent.

Pour valider une assignation de rôle au niveau opérationnel (et non structurel), nous proposons de mettre en œuvre deux systèmes de prédicats. Le premier, V_r, doit permettre au rôle de vérifier les propriétés du composant concerné par l'assignation. Les vérifications possibles portent sur le type du composant mais également sur son modèle de comportement, voir ses caractéristiques internes. Ce mécanisme est implémenté dans Comet/J par la classe `Role` et la méthode spéciale `checkBase`. Si cette méthode retourne `false`, alors l'assignation du rôle est refusée.

En plus de la vérification opérée par le rôle lui-même (règles V_r), un système V_C est également proposé pour permettre au composant, de son côté, de valider l'assignation. Ce système est associé au modèle de comportement C et pris en charge par

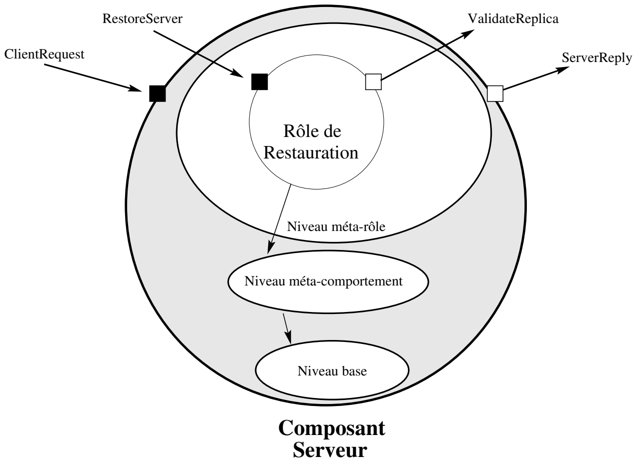


Figure 3. *Rôle de restauration assigné à un composant serveur*

le niveau méta-comportement. Ceci permet de gérer le cas des rôles conflictuels[7] en interdisant, par exemple, leur composition. Une autre possibilité serait de séquentialiser les rôles en conflit (introduction d'une relation d'ordre). Indépendamment de la résolution de conflit, un composant pourra également demander une authentification préliminaire des rôles avant assignation. Une autre vérification courante concernera la validité des intercepteurs d'événements, il faudra alors vérifier la relation $H_r \subseteq E$[8]. Après satisfaction de ces systèmes et assignation du rôle r à un composant Com, le composant Com' résultant sera défini de la façon suivante :

$$Com' = assign(r,Com) = \begin{cases} e' = e \cup e_r \\ M' = M \cup M_r \\ E' = E \cup E_r \\ S' = S \cup S_r \\ R' = R \cup R_r \end{cases}$$

Nous voyons ici clairement que le type des composants Comet est constitué d'une partie statique, définie par le composant lui-même, et d'une partie dynamique, union des types des rôles assignés.

7. Rôles qui interceptent ou gèrent les mêmes types d'événement.
8. L'ensemble des événements interceptés par le rôle doit être inclus, au sens large, dans l'ensemble des événements acceptés par le composant. Le modèle de comportement par défaut nécessite la vérification de cette relation.

4. Protocoles Comet

Les rôles assignés sont gérés par les composants mais les rôles eux-mêmes [9] sont définis au sein de protocoles. Suivant l'approche de DIL [STU 96], les protocoles sont introduits au sein de Comet en tant qu'entités logicielles autonomes. Cependant, contrairement à DIL, les protocoles Comet ne concernent pas uniquement les interactions entre composants. Considérons en guise d'exemple un protocole dit d'activabilité, responsable de l'assignation d'un rôle d'activation [10] à un ou plusieurs composants. Ces composants deviennent ainsi activables à distance sans pour autant participer à de nouvelles interactions.

Les besoins en terme d'adaptabilité nécessitent à la fois la possibilité de modifier dynamiquement les composants mais également de considérer la topologie du système comme un élément dynamique. Les protocoles Comet offrent donc également la possibilité de gérer des connexions dynamiques entre composants.

De façon plus pragmatique, les protocoles disposent des caractéristiques suivantes :

– des fonctionnalités de protocole (ex. ajouter un réplica),

– un état interne, optionnel (ex. gestion de la liste des réplicas),

– des rôles de protocole, qui peuvent être assignés dynamiquement, grâce à la primitive `assign` (ex. rôle répliqué).

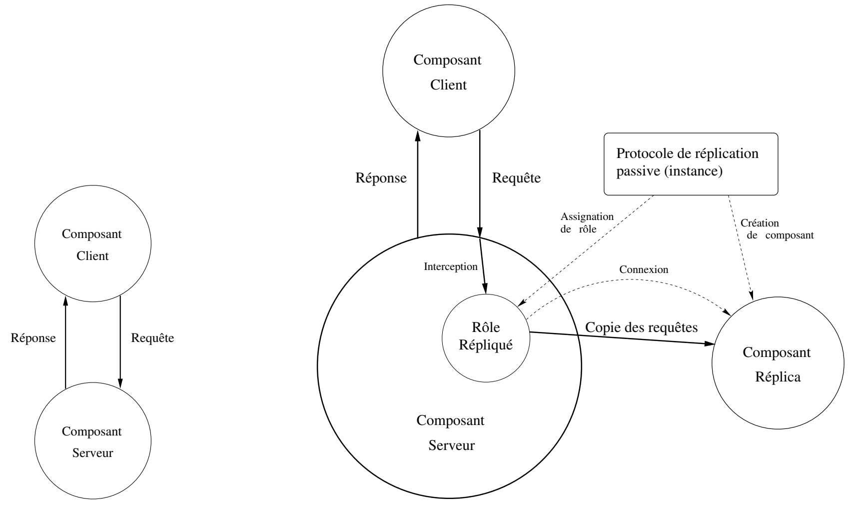
Prenons l'exemple d'une application client/serveur minimaliste dont l'architecture est présentée en figure 4(a). Supposons que durant certaines phases critiques, il soit nécessaire d'ajouter dynamiquement à cette application un protocole de réplication passive. L'idée de base du protocole est la création d'un (ou de plusieurs) réplica(s) et l'assignation dynamique du rôle de répliqué au serveur. La réalisation de ce protocole correspond à la figure 4(b).

Le protocole de réplication passive que nous nous proposons d'élaborer met en jeu le rôle `ReplicatedRole`, défini de la façon suivante :

```
public class ReplicatedRole extends comet.Role
             implements comet.ClientRequestEventPreHook,
                        comet.ReplicationEventSender {
  public void BeforeClientRequest(ClientRequestEvent cre) {
    ReplicationEvent re = new ReplicationEvent(cre);
    sendEvent(re);
  }
}
```

9. Dans Comet/J, les Rôles sont implémentés par des classes et les rôles assignés correspondent à des instances de ces classes

10. Gestion des événements de type `ActivateEvent`.



(a) Avant (b) Après instanciation

Figure 4. *Instanciation dynamique d'un protocole de réplication passive*

Ce rôle n'introduit pas de nouveaux événements mais intercepte les requêtes reçues par le serveur[11] et envoie au(x) réplica(s) une copie de ces requêtes. Nous pouvons maintenant exprimer, toujours selon une approche objet, notre protocole générique de réplication passive :

```
public class ReplicationProtocol extends comet.Protocol {
  public String createReplica(Address machine) {
  upload("Replica", machine);
  String id = instantiate("Replica", machine);
  return id;
  }
  public void replicate(String id_server, String id_replica) {
    assign("ReplicatedRole", id_server);
    connect("ReplicationEvent", id_server, id_replica);
  }
}
```

Ce protocole, une fois instancié, permet de créer des réplicas potentiels (`createReplica`) et de connecter ces réplicas à des serveurs (`replicate`). Nous pouvons voir sur cet exemple que la notion de protocole est étendue dans Comet aux problèmes de déploiement.

11. Intercepteur `ClientRequestEventPreHook`, associé à la méthode `BeforeClientRequest`.

Ainsi, les protocoles permettent également :

– d'envoyer le code de composants sur des machines distantes (`upload`),

– d'instancier des composants, quelle que soit leur localisation dans le système distribué (`instantiate`),

– de connecter des composants entre eux (`connect`).

Les protocoles gèrent donc les connexions entre composants, qui ne sont pas assimilées dans Comet à des entités logicielles autonomes. La réalisation d'une connexion correspond plutôt à l'invocation de la primitive de protocole `connect`.

Les primitives de base associées au protocoles peuvent de plus être augmentées, pour les modèles de composants mobiles, par une primitive de migration (`migrate`).

La caractéristique d'indépendance des protocoles et des rôles qu'ils gèrent permet la création de bibliothèques réutilisables dans de nombreux contextes. Par exemple, le protocole de réplication passive peut être ajouté à n'importe quelle application proposant un composant serveur acceptant des événements de type `ClientRequestEvent` en entrée [12].

5. Travaux connexes

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, basé sur les acteurs. Ceci confirme notre attachement au modèle de calcul par acteurs. D'importantes différences sont cependant notables, notamment au niveau de la granularité, plus importante dans le cas de Comet, même si des modèles de composants peuvent être construits au dessus des acteurs [AST 99]. Les composants Comet autorisent également les activités multiples, ce qui n'est pas le cas des acteurs. Dans ce sens, nos composants sont plus proches, au niveau comportemental, des objets actifs, par exemple modélisés dans le framework Actalk [BRI 96].

Concernant le modèle de composant, notre principale source d'inspiration provient des travaux dans le cadre des architectures logicielles [GAR 93]. Nous pensons cependant que l'approche événementielle permet de représenter de façon neutre d'autres types d'architectures. Le framework Rapide [LUC 96] est à ce titre comparable au nôtre mais sa portée se limite au prototypage d'algorithmes concurrents et à leur vérification par ordonnancement partiel d'événements.

Les composants Comet sont proches structurellement des composants JavaBeans [Sun 98] et au niveau opérationnel des Enterprise JavaBeans [Sun 99] ou Corba Com-

12. Sans oublier l'extension du typage des événements à une relation de sous-typage. N'importe quel sous-type de `ClientRequestEvent` ferait ici l'affaire.

ponents [Obj 99], mais ces différents modèles industriels ne permettent pas ou difficilement l'expression séparée des fonctionnalités et du contexte d'exécution. L'adaptabilité dynamique n'est pas non plus d'actualité dans ces approches.

La conception du niveau méta-comportement est inspirée de CodA [MCA 95] qui propose une vision fonctionnelle (et non structurelle comme c'est généralement le cas) de la composition des méta-objets. Les objectifs de CodA et Comet sont cependants assez différents. CodA permet la spécialisation du contrôle des objets alors que Comet introduit les composants et limite la portée du paradigme objet aux fonctionnalités de base. Dans ce sens, il s'agit d'une approche de plus haut niveau.

Le concept de rôle a été introduit dans un grand nombre de thèmes de recherche. Le plus récent étant le thème “Role Modelling” [RHI 98], focalisé sur l'expression, au niveau des classes, des intéractions entre objets. Dans notre approche, les rôles ne peuvent être planifiés au moment de la conception de l'application puisque, par définition, ils correspondent à des fonctionnalités non-prédéfinies des composants. Cependant, nous envisageons de rapprocher ces deux visions pour faciliter la tâche de l'ingénieur concernant la planification des modifications par modélisation des protocoles et rôles à assigner.

Contrairement aux architecture réflexive de type “middleware” comme OpenCorba [LED 99], les hypothèses concernant les communications sont peu nombreuses dans Comet. Nous pensons que dans le cadre de systèmes distribués hautement dynamiques, de nombreuses propriétés de contrôle devraient être associées aux entités logicielles et non déléguées à la couche de communication sous-jacente. Ceci permet à différents composants, associés à des stratégies de contrôles multiples, de coopérer. Le choix d'un mode de communication basé sur les événements est également plus proche de la nature asynchrone des systèmes distribués que les invocations synchrones de méthodes.

L'Aspect Oriented Programming [KIC 97] identifie clairement les problèmes de conception dans la construction de systèmes complexes. Les besoins en terme de séparation des préocupations sont précisément exposés : composants pour décomposer un problème et implémenter une solution, aspects pour expliciter une partie du contrôle des composants. En introduisant des générateurs de code en lieu et place de notre approche applicative, Comet/J serait comparable à un framework AOP comme D [LOP 97], qui ne s'intéresse cependant pas à la problèmatique des applications adaptatives.

6. Conclusion

Face à une problématique duale système/génie logiciel, nous proposons, par le biais de notre architecture Comet, une approche mixte, à la fois réflexive et basée sur les composants. Les niveaux sémantiques que nous introduisons permettent de résoudre le problème de l'expression du contrôle des composants, indépendamment de leurs fonctionnalités. Ceci facilite l'élaboration de systèmes distribués complexes, sans pour autant contraindre le modèle de développement. Les composants Comet, une fois immergés en environnement concurrent et réparti, conservent leurs (bonnes) propriétés de génie logiciel, ce qui est un résultat encourageant [GUE 99].

Les rôles et protocoles Comet couvrent également un vaste ensemble de besoins. La possibilité d'adapter dynamiquement les composants, de façon contrôlée, est dans ce sens la principale originalité de notre approche. Nous avons pour objectif de construire les systèmes d'information de demain de façon incrémentale, sans pour autant complexifier leur conception ni mettre en défaut leur disponibilité. Dans le cadre des architectures logicielles, les protocoles Comet (et avant eux les protocoles DIL) gèrent des connexions dont la nature dynamique ouvre de nouvelles voies très intéressantes. Les architectures ne sont plus des entités figées et fermées, elles évoluent, tout comme les applications qu'elles modélisent.

Notre vision des composants préserve également les acquis de la programmation orientée objet. Certes, les composants, événements, protocoles et rôles Comet ne sont pas des objets mais ils sont tous représentables, très naturellement, par des objets. Les différents exemples réalisés dans notre framework Comet/J sont dans ce sens assez explicites. La version actuelle de Comet/J offre la plupart des fonctionnalités détaillées dans cet article. Les protocoles sont en particuliers opérationnels, ce qui nous a permis de concevoir et tester quelques systèmes distribués simples : réplication adaptative, synchronisation d'horloges, etc. Pendant la finalisation de ce framework, nous validerons notre approche par des exemples applicatifs de taille significative :

– Système de diffusion vidéo à la demande.
– Applications industrielles de type workflow.
– Prototypage de systèmes multi-agents.

A plus long terme, nous envisageons de réaliser des langages et outils autour de Comet/J ou d'autres instanciations de notre architecture. Ces outils, liés à de nouveaux concepts méthodologiques, offriront un cadre intéressant pour la réalisation d'applications adaptatives d'envergure, notamment dans la communauté des systèmes multi-agents (applications dans la robotique collective, la gestion de crise, etc.).

Du fait de notre approche résolument descendante, un autre objectif majeur sera de développer une sémantique claire des concepts introduits par notre architecture : modèle de composant, cohérence des adaptations, vision "files d'attente", etc.

Remerciements

Cet article, sous sa forme actuelle, n'aurait pu voir le jour sans les relectures, nombreuses et approfondies, de Jean-Pierre Briot. Je l'en remercie donc chaleureusement. Un grand merci également aux autres membres du thème *Framework* de l'équipe OASIS - en particulier à Thomas Meurisse - ainsi qu'aux relecteurs anonymes pour leurs commentaires toujours très pertinents.

7. Bibliographie

[ABA 96] ABADI M., CARDELLI L., *A Theory of Objects*, Springer, 1996.

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

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

[BRI 96] BRIOT J.-P., « An Experiment in Classification and Specialization of Synchronization Schemes », *Proceedings of ISOTAS'96*, LNCS 1049, Springer Verlag, Mars 1996, p. 227-249.

[BRI 98] BRIOT J.-P., GUERRAOUI R., LÖHR K.-P., « Concurrency and Distribution in Object-Oriented Programming », *ACM Computing Surveys*, vol. 30, n^o 3, 1998, p. 291–329.

[GAR 93] GARLAN D., SHAW M., « An Introduction to Software Architecture », AMBRIOLA V., TORTORA G., Eds., *Advances in Software Engineering and Knowledge Engineering*, vol. I, World Scientific Publishing Company, 1993.

[GUE 99] GUERRAOUI R., « What Object-Oriented Distributed Programming may Be - and What it does Not have to Be », White paper, 1999.

[KIC 97] KICZALES G. et al., « Aspect-Oriented Programming », *Proceedings of the ECOOP'97*, LNCS 1241, Springer Verlag, Juin 1997.

[LED 99] LEDOUX T., « OpenCorba: a Reflective Open Broker », *Proceedings of Reflection'99*, Springer Verlag, Juillet 1999.

[LOP 97] LOPES C., KICZALES G., « D : A Language Framework for Distributed Programming », rapport , Février 1997, Xerox Palo Alto Research Center.

[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.

[MAT 93] MATSUOKA S., YONEZAWA A., « Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages », AGHA G., Ed., *Research directions in Concurrent Object-Oriented Programming*, p. 107-150, MIT Press, 1993.

[MCA 95] MCAFFER J., « Metalevel Programming with Coda », *Proceedings of ECOOP'95*, LNCS 952, Springer Verlag, Août 1995, p. 190-214.

[Obj 99] OBJECT MANAGEMENT GROUP, « Corba components model », `http://www.omg.org`, 1999.

[RHI 98] RHIEHLE D., GROSS T., « Role Model Based Framework Design and Integration », *Proceedings of OOPSLA'98*, ACM Press, 1998, p. 117–133.

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

[Sun 98] SUN MICROSYSTEMS, « JavaBeans Specifications 1.01 », `http://java.sun.com/beans`, 1998.

[Sun 99] SUN MICROSYSTEMS, « Enterprise JavaBeans Specifications 1.1PR2 », `http://java.sun.com/products/ejb`, 1999.

[YON 89] YONEZAWA A. et al., *ABCL, An Object-Oriented Concurrent System*, Series in Computer Systems, MIT Press, 1989.