
ACEEL : modèle de composants auto-adaptatifs

Application aux environnements mobiles

Djalel Chefrour* — Françoise André**

* INRIA Rennes

** Université de Rennes 1

IRISA, Campus de Beaulieu
35042 Rennes cedex, France

{chefrour,fandre}@irisa.fr

RÉSUMÉ. Les applications s'exécutant dans les environnements mobiles ont besoin de s'adapter aux fluctuations des niveaux de ressources disponibles comme, par exemple, les variations de la bande passante dans les réseaux sans fil. Pour supporter le développement de telles applications, nous proposons un modèle de composants auto-adaptatifs qui changent dynamiquement leurs comportements pour s'accommoder aux nouvelles conditions d'exécution. Notre modèle repose sur le schéma de conception Strategy et sur un mécanisme de notification par événement des variations de l'environnement. Il comporte un méta-niveau pour le contrôle de l'adaptation selon une politique écrite à part dans un langage de script. Nous présentons une implantation du modèle en Python avec son expérimentation sur une application de vidéo à la demande avec différents comportements écrits en Java.

ABSTRACT. Applications running on mobile environments need to adapt to the fluctuations of available resources levels as, for example, the variations of bandwidth in wireless networks. To support the development of such applications, we propose a model of self-adaptive components that change their behaviors dynamically to accommodate the new runtime conditions. Our model is based on the Strategy design pattern and on an event driven mechanism for the notification of environment variations. It contains a meta-level for the control of adaptation according to a policy written separately in a scripting language. We present an implementation of our model in Python and its experimentation for developing a video-on-demand application with different behaviors written in Java.

MOTS-CLÉS : Composants auto-adaptatifs, Schéma de conception Strategy, Réflexion, Environnements mobiles, Vidéo à la demande

KEYWORDS: Self-adaptive components, Strategy Design Pattern, Reflection, Mobile Environments, Video-on-Demand

1. Introduction

Au cours de la dernière décennie, l'important avancement des technologies des réseaux sans fils (e.g., IEEE 802.11 [IEE], Bluetooth [Blu], GSM [GSM]) et des équipements mobiles (PDA, Téléphones portables...), a permis un développement explosif des environnements mobiles [JOS 95]. Une importante partie des travaux de recherche dans ce domaine visent à rendre efficace l'exploitation d'applications usuelles (e.g., accès au Web, messagerie, vidéo) dans ces nouveaux environnements [SAT 96, DAV 96], ainsi que le développement de nouvelles applications dites nomades (e.g., commerce mobile [VAR 00b], guide touristique [CHE 00]).

Toutefois, contrairement aux environnements traditionnels statiques (réseau filaire et stations de travail fixes), les environnements mobiles sont caractérisés par une limitation des ressources disponibles (CPU, Bande passante, Énergie) ainsi qu'une importante variation dynamique de leurs caractéristiques [CHA 99, HOF 94]. En effet, afin d'être portables, les équipements utilisés sont limités en taille et en poids, ce qui induit une limitation de la source d'énergie et donc une faible puissance de calcul, une petite mémoire et des dispositifs d'interface utilisateur réduits. Par ailleurs, ces équipements utilisent souvent des réseaux sans fils pour leurs besoins de communications, or ces réseaux [VAR 00a] sont caractérisés par une faible bande passante qui de plus peut fluctuer considérablement, par de fréquentes déconnexions et un important taux d'erreurs.

Ces contraintes sur les ressources disponibles dans les environnements mobiles induisent une dégradation du comportement des applications traditionnelles car celles-ci ont été développées pour s'exécuter sous des conditions statiques. Il est reconnu par la majorité de la communauté des chercheurs du domaine que la solution de ce problème passe par l'adaptation du comportement de ces applications aux variations dynamiques des ressources de l'environnement [KAT 94, ALB 01, SAT 96].

L'objectif de notre travail est de fournir des mécanismes appropriés pour supporter le développement de logiciels adaptatifs pour les environnements mobiles. Pour cela nous proposons un cadre de conception (Framework) orienté-objet dont la base est constituée d'un modèle de composants auto-adaptatifs et d'un sous-système de détection-notification. Il permet à une application (ou service) de changer dynamiquement de comportement suite aux variations survenues dans l'environnement.

La suite de cet article est organisée comme suit : la section 2 présente l'approche que nous avons retenue pour prendre en charge l'adaptation d'applications dans les environnements mobiles. La section 3 détaille notre modèle de composants auto-adaptatifs et le sous-système de détection-notification des variations de l'environnement. La section 4 illustre l'utilisation de notre framework pour le développement d'une application vidéo adaptative réalisée avec Java et Python. Pour conclure nous mentionnerons les travaux futurs que nous envisageons.

2. Gestion de l'adaptation

L'adaptation aux changements de l'environnement touche pratiquement tous les niveaux depuis la pile des protocoles réseau jusqu'à l'utilisateur final en passant par l'intergiciel (middleware) et l'application elle-même [FRI 99]. Pour simplifier la tâche des développeurs d'applications adaptatives, nous adoptons le point de vue de [BLA 97] qui consiste à fournir l'accès aux diverses techniques d'adaptation des différents niveaux depuis l'intergiciel.

Par ailleurs, les services offerts par cet intergiciel ne doivent pas être implantés sous forme de boîtes noires qui rendent l'adaptation transparente aux applications. Par contre, ils doivent être implantés par des composants logiciel ouverts afin de permettre aux applications (qui le désirent) d'intervenir et faire les choix d'adaptation appropriés dans le cas où la solution fournie par défaut serait inadéquate [SAT 96]. Pour le développement de tels composants nous avons mis au point le modèle ACEEL (self-Adaptive ComponEnts modEL) décrit ci-après.

3. Modèle de composant auto-adaptatif

Nous utilisons le terme composant pour désigner toute entité logicielle qui fournit un service particulier via une interface séparée de l'implantation mettant en œuvre ce service. C'est une définition un peu différente de celles qu'on trouve dans la littérature [MEI 97] qui introduisent en plus des notions telles que *connecteur*, *conteneur* utiles à la génération (par composition) et au déploiement d'applications. Il faut noter que notre travail se situe dans un cadre système classique et que, comme [IND 00], nous distinguons les adaptations dites réactives dont l'objectif est de s'accommoder aux conditions de l'environnement, de celles dite évolutives dont le but est d'étendre les fonctionnalités existantes d'un composant et qui s'inscrivent dans le domaine du génie logiciel. Bien sûr ceci n'empêche pas d'utiliser les mêmes techniques pour réaliser l'une ou l'autre adaptation et de construire un système qui les utilise toutes les deux.

L'objectif de notre modèle est de permettre le changement dynamique du comportement d'un composant réalisant un service particulier. Ceci peut être réalisé par une large instruction conditionnelle (e.g., switch/case) qui sélectionne le comportement approprié en testant les variations de l'environnement d'exécution. Seulement ce type de code est difficile à maintenir et à réutiliser. Or, ces deux problèmes sont résolus par le recours aux techniques orientées objet. Plus précisément, dans notre cas il convient d'utiliser le Strategy Design Pattern [GAM 95].

3.1. Design Pattern Strategy

Il encapsule les variantes d'un algorithme dans une hiérarchie de classes et les rend interchangeables afin d'altérer le comportement d'une application sans changer son architecture. La racine de cette hiérarchie (la classe Strategy dans la figure 1) déclare

une interface commune à toutes les variantes de l'algorithme, celles-ci sont encapsulées dans les classes dérivées `ConcreteStrategy`. Les données manipulées par l'algorithme sont isolées dans la classe `Context` qui maintient une référence vers l'objet `ConcreteStrategy` sélectionné et lui transmet toutes les requêtes provenant des clients.

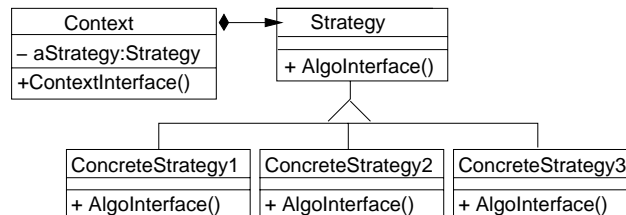


Figure 1. *Design Pattern Strategy*

Nous avons basé la construction de notre modèle sur le Strategy Design Pattern qui a l'avantage principal de permettre de modifier facilement l'une des variantes de l'algorithme ou d'en ajouter une nouvelle. Cependant, tel que ce pattern est décrit dans [GAM 95], cet ajout ne peut être fait dynamiquement en cours d'exécution car les variantes alternatives de l'algorithme sont définies à la phase de conception avant le déploiement de l'application. Par ailleurs, c'est le concepteur du composant qui détermine les différents comportements possibles et les changements de l'un vers l'autre. Nous souhaitons éviter qu'il mette ensemble le code qui gère le processus d'adaptation et les codes des différents comportements, afin d'obtenir une conception claire et réutilisable.

Pour à la fois permettre un ajout dynamique de nouveaux comportements et séparer les codes fonctionnels du code réalisant l'adaptation, nous avons recours à la technique de réflexion qui offre les moyens d'agir sur les comportements à un niveau extérieur à ceux-ci.

3.2. *Réflexion*

Un système logiciel est dit réflexif s'il est capable d'inspecter et de modifier sa propre interprétation de son domaine d'application. Pour ce faire, il doit manipuler une représentation de lui-même (i.e., des structures et opérations qui interprètent son domaine d'application) [MAE 87]. La partie de tout système réflexif qui traite le domaine d'application est appelée « niveau de base ». Tandis que le terme « méta-niveau » désigne la partie qui manipule la représentation de soi.

Une approche commune pour définir les services du méta-niveau consiste à utiliser un protocole méta-objet (MOP) [KIC 91] qui offre l'accès à la représentation de soi via un framework orienté-objet. Les services du méta-niveau sont généralement appelés par l'application elle-même ou par la plate-forme sous-jacente. Cette dernière peut par

exemple posséder des mécanismes de surveillance de la qualité de service qui peuvent déclencher des actions de contrôle situées dans le méta-niveau. Par conséquent, un processus réflexif peut adapter dynamiquement son comportement pour satisfaire des changements dans son environnement.

3.3. Composant auto-adaptatif

La figure 2 montre l'architecture de ACEEL (self-Adaptive ComponEnts model) notre modèle de composant auto-adaptatif et réflexif avec deux niveaux séparés :

1) Le **niveau de base** est conçu selon le Design Pattern Strategy. Il inclut l'objet **Context** qui réalise l'interface du composant avec ses clients et encapsule, dans l'attribut **State**, l'état commun à tous ses comportements alternatifs représentés par les objets dérivés de **Behavior**.

2) Le **méta niveau** contient l'objet **Adapter** qui prend en charge l'auto-adaptation du composant. Pour cela, il utilise une politique d'adaptation fournie sous forme d'un script et s'appuie sur un sous-système de détection-notification des changements de l'environnement (cf. sous-section 3.5). L'objet **Adapter** a un cycle de vie identique à celui de l'objet **Context** puisqu'il est fortement couplé avec ce dernier.

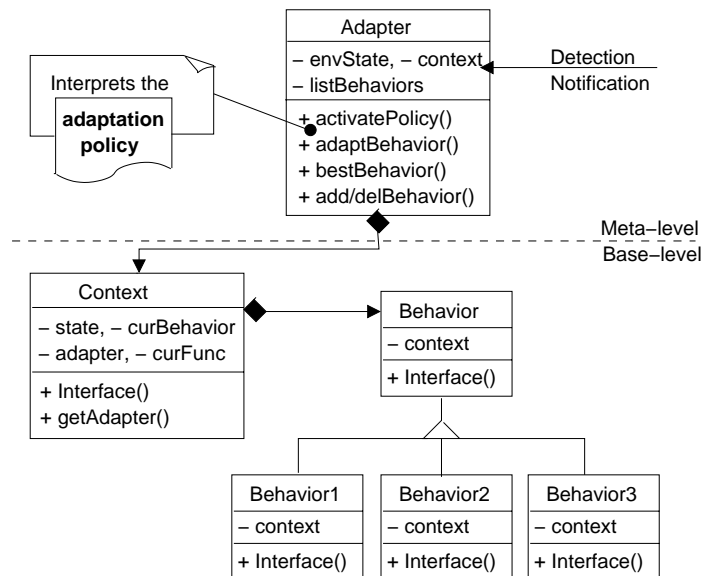


Figure 2. Modèle de composant auto-adaptatif ACEEL.

Écrite dans un script à part, la **politique d'adaptation** spécifie deux types d'informations : (1) Le cycle de vie de chaque objet dérivé de **Behavior**, i.e., s'il doit être créé / détruit en même temps que le composant ou seulement quand il sera activé

/ désactivé. (2) Les règles d'adaptation du comportement du composant qui sont sous la forme **On-Event : Action**. Le champ **Event** correspond à l'un des événements générés par le sous-système de notification des variations de l'environnement (e.g., $bandwidth \leq 16kbs$) et le champ **Action** précise l'une des *actions d'adaptation* possibles décrites dans la section 3.4 (e.g., **new-behavior = MPEG-codec**).

Le **méta-objet Adapter** fournit trois méthodes pour changer le comportement courant du composant qui est spécifié par la variable **context.curBehavior**¹. Premièrement, la méthode **activatePolicy** qui est appelée juste après la création du composant parcourt la politique d'adaptation afin : (1) d'établir, à partir des champs **Event** des règles d'adaptation, la liste des ressources dont les variations provoquent l'adaptation du composant, puis de s'enregistrer auprès des moniteurs de ces ressources qui sont gérés par le sous-système de détection-notification, (2) de savoir à quels moments les différents objets dérivés de **Behavior** seront créés et détruits.

Deuxièmement, la méthode **adaptBehavior** implémente le protocole d'adaptation décrit dans la sous-section 3.4 qui spécifie comment les opérations d'adaptation sont gérées. Enfin, **adaptBehavior** utilise la méthode **bestBehavior** qui sélectionne le comportement le mieux approprié parmi tous ceux listés dans l'attribut **listBehaviors**. Cette sélection est réalisée par interprétation de la politique d'adaptation et dépend, entre autres, de l'état de l'environnement qui est encapsulé dans le vecteur **envState**.

3.4. Protocole d'adaptation

Ce protocole spécifie les étapes précises par lesquelles passe le changement dynamique du comportement courant d'un composant ainsi et que les règles garantissant le succès de ce changement.

3.4.1. Étapes d'adaptation

Les étapes d'adaptation sont :

1) Activation de **adaptBehavior** par notification d'un changement survenu dans l'environnement (e.g., chute de la bande passante du réseau) ou par une demande explicite d'un autre composant généralement afin de coordonner les actions d'adaptation des deux. Cette activation peut avoir lieu à n'importe quel moment car l'événement qui la déclenche peut être asynchrone par nature (e.g., la perte de la connexion dans un réseau sans fils).

2) Gel provisoire de l'interface du composant et du mécanisme de notification des variations de l'environnement afin de préserver l'état du composant durant la phase d'adaptation. Elle consiste à mettre en attente, respectivement dans deux files de type

1. Bien que **curBehavior** soit une méta-information nous avons choisi de la mettre dans *Context* pour des raisons d'optimisation car elle sert à rediriger toutes les requêtes client parvenant à *Context* vers le comportement courant

Fifo, les requêtes des clients et les événements émis par le sous-système de détection-notification de façon transparente vis-à-vis de leurs émetteurs.

3) Interprétation de la politique d'adaptation. Généralement cette étape comporte (1) la mise à jour de la variable `envState` qui encapsule l'état de l'environnement et (2) le choix et l'exécution de l'action d'adaptation. En détails, la mise à jour de `envState` repose sur les informations associées à l'événement de notification et/ou celles recueillies plus tard pour être exploitées dans le choix d'adaptation. Par exemple, sur un terminal équipé avec plusieurs interfaces réseau, suite à une chute de la bande passante sur l'une de ces interfaces l'application pourrait demander l'état d'une autre interface pour basculer sur celle-ci. Concernant l'action d'adaptation, elle peut consister à :

a) Ne rien faire. Cela peut correspondre au cas d'une politique d'adaptation sous-spécifiée : Certains cas de variations (de l'environnement) envisageables ne sont pas traités².

b) Modifier certains paramètres du comportement en cours (e.g., changer le taux de compression d'un algorithme de codage vidéo).

c) Changer le comportement en cours. Ceci nécessite le choix d'un nouveau comportement, son (télé)chargement s'il n'est pas présent en mémoire et, éventuellement (selon la politique d'adaptation), la destruction de l'ancien comportement.

4) Dégel de l'interface du composant et du mécanisme de notification.

5) Reprise, si le composant était actif, du traitement au début de la méthode (référéncée par `context.curFunc`) qui s'exécutait au moment du déclenchement de l'adaptation.

3.4.2. Règles du protocole d'adaptation

Le processus d'adaptation défini dans la section précédente se base sur les hypothèses suivantes qui constituent pour le développeur du composant des règles de conception importantes à respecter.

1) **Les objets dérivés de Behavior doivent être des objets sans état (stateless objects)**, i.e., toute donnée rémanente utilisée dans les objets dérivés de **Behavior** doit être maintenue par l'objet **Context** (dans l'attribut **State**). Cette règle permet d'éviter le problème de transfert d'état entre les comportements alternatifs du composant dans le cas où cet état serait défini comme donnée membre de ces comportements. Ce transfert d'état est difficile à mettre en œuvre car il dépend à la fois des types de ces données, de la nature du traitement et des deux comportements interchangeable. Par conséquent, un tel mécanisme ne peut être fourni par défaut dans notre framework mais seul le développeur du composant peut le réaliser. Or nous voulons éviter cela pour la raison suivante :

Si le composant possède N comportements alors, théoriquement, on aura besoin

2. Ce genre de situation peut être évité par une vérification (formelle) préalable de la politique d'adaptation

d'écrire $N(N-1)$ procédures de transfert et ce nombre sera revu à la hausse à chaque fois que l'on rajoute un nouveau comportement.

2) **Les différentes implantations de toutes les méthodes appartenant à l'interface du composant doivent avoir le même effet de bord sur son état.** Cette règle concerne les opérations d'adaptation effectuées entre des appels de méthodes successives qui réalisent ensemble un traitement particulier. Elle permet au nouveau comportement activé après l'adaptation de poursuivre le traitement entamé par l'ancien comportement alors que ce dernier a déjà modifié partiellement l'état du composant. Dans la pratique, une telle règle n'est pas difficile à vérifier puisque les différentes implantations d'une méthode de l'interface réalisent la même fonction et ont la même signature. Par conséquent, il est facile d'éviter les cas où, faute d'inattention, le développeur écrit un comportement dont les méthodes rendent l'état du composant *inexploitable* par les autres comportements.

Cette notion d'*exploitabilité* de données manipulées par plusieurs méthodes de comportements différents est plus spécifique que la notion de *cohérence* relative au problème classique d'accès concurrents à une ressource partagée. Dans le second cas, les entités exploitant les données sont (généralement) indépendantes les unes des autres contrairement au premier cas où elles (i.e., les objets dérivés de **Behavior**) réalisent le même traitement sur un état unique de façon interchangeable. Ainsi un état modifié par l'une des méthodes d'un comportement pourrait être cohérent mais non exploitable par les autres comportements.

3) **Les modifications d'état doivent être atomiques de façon à garantir un état du composant cohérent et exploitable.** Cette règle traite le cas où le composant était actif et, éventuellement, son état était en cours de modification au moment du déclenchement d'une adaptation. Dans une telle situation, une question importante se pose : Comment reprendre le traitement interrompu, en utilisant le nouveau comportement, tout en assurant que ce dernier ne retrouve pas les données du composant dans un état inexploitable ?

Avant d'introduire la règle qui répond à cette question, notons que ce cas est similaire au précédent sauf qu'il est limité à l'échelle d'une seule méthode. Plus précisément, en plus de préserver la propriété de cohérence des données du composant, il faut mémoriser le point du traitement atteint au moment de l'interruption d'adaptation et s'en servir comme point d'entrée pour le nouveau comportement. Or, il est évident que cela ne peut être fait à n'importe quel endroit du code de la méthode interrompue car il peut être carrément différent du nouveau code qui va reprendre le traitement (même s'ils réalisent la même fonction). Toutefois, selon la nature du traitement réalisé, il reste souvent possible de suspendre le code d'une méthode lors de sections spécifiques dites *interruptibles* (à identifier par le développeur) et de poursuivre le traitement au début de la nouvelle méthode. Nous avons retenu cette approche comme solution car elle permet à la fois de garantir la propriété d'exploitabilité de l'état du composant par le nouveau comportement et d'utiliser les points d'entrée naturels des méthodes comme points d'entrée pour la reprise du traitement après l'adaptation. A titre d'exemple, mis à part les instructions qui réalisent l'échange de deux éléments d'un tableau, le reste du code d'une méthode de tri par insertion peut être interrompu

afin de lancer une nouvelle méthode de tri sur le même tableau sans problème particulier.

Pour résumer, la règle consiste donc à rendre non interruptibles (par une adaptation) les sections de code dans les objets dérivés de `Behavior` qui mettent l'attribut `Context.State` dans un état inexploitable. Ceci peut être fait par un mécanisme de synchronisation (e.g., un sémaphore) fourni par l'objet `Context` et utilisé d'une part, systématiquement, au début et à la fin de la fonction `adaptBehavior` et d'autre part, par le développeur, dans les méthodes des objets dérivés de `Behavior`.

3.5. Sous-système de détection-notification

Le processus de détection et notification des variations de l'environnement dans ACEEL est pris en charge par le *monitoring engine* montré dans la figure 3. Ce der-

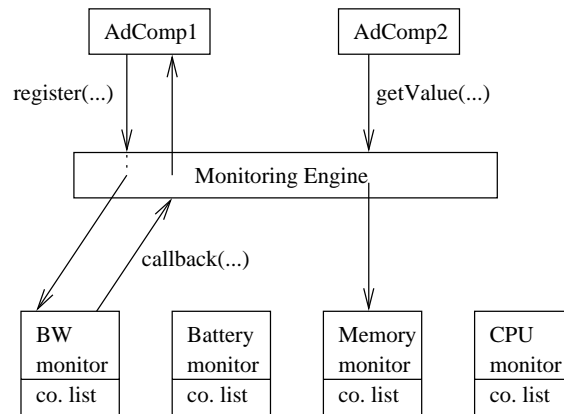


Figure 3. *Sous-système de détection notification.*

nier est un processus démon s'exécutant sur chaque machine hôte sur laquelle sont déployés des composants ACEEL. Il reçoit toutes les requêtes de surveillance émises par ces composants (plus précisément par leurs fonctions `activatePolicy`) et les redirige vers les moniteurs de bas niveau pour les traiter. Concrètement, chacune de ces requêtes est un appel à la méthode `register` ou à la méthode `getValue` fournies par le monitoring engine. La première correspond à une demande de surveillance des variations d'une ressource spécifique tandis que la deuxième permet la consultation de la valeur de l'un des paramètres de cette ressource (e.g. latence d'une liaison réseau). Dans les deux cas, cette demande est prise en charge par un moniteur de bas niveau dédié à la ressource en question (e.g., BW Monitor pour la bande passante réseau). S'il s'agit simplement d'une consultation de valeur alors le moniteur renvoie celle-ci au monitoring engine qui va l'acheminer jusqu'au composant ayant établi la demande. Dans l'autre cas (surveillance), le moniteur enregistre dans une liste l'identité du composant intéressé et l'événement qui, à son occurrence, sera notifié à ce composant (à la

méthode `adaptBehavior` de son objet `Adapter`) par un callback. Le recours au monitoring engine qui s'interpose entre les composants adaptatifs et les moniteurs de base se justifie par les avantages suivant :

1) Il allège les composants de la tâche complexe de gestion des moniteurs et la réalise de façon efficace. Notamment il prend en charge la création et la destruction de ces moniteurs selon les besoins de surveillance des composants s'exécutant sur sa machine hôte.

2) Il fournit à travers les deux fonctions `register` et `getValue` une interface homogène indépendante des types des ressources.

4. Expérimentation

Dans l'objectif de tester le modèle ACEEL nous avons développé un framework en Python [Pyt] offrant les trois classes `Adapter`, `Context` et `Behavior`, le monitoring engine et quelques moniteurs de base. Nous avons choisi Python car c'est un langage de script évolué qui permet le prototypage rapide de nouveaux concepts comme notre modèle de composant. Il est portable (multi-plateformes) et facile à interfacer avec d'autres langages (e.g., `C++`) qui peuvent être utilisés pour coder les comportements d'un composant (i.e., classes dérivées de `Behavior`). De plus, Python est un langage orienté-objet, à typage dynamique et surtout *réflexif* chose qui a facilité l'implantation du méta-niveau dans ACEEL.

Nous avons ensuite développé une application de vidéo à la demande composée d'un serveur et d'un client conçus selon le modèle ACEEL et écrits en Python. Le serveur de fichier vidéo s'exécute sur une station de travail et le client qui est un player vidéo s'exécute sur un portable relié à la station par un réseau sans fils du type WaveLAN comme le montre la figure 4. Le serveur vidéo possède deux comportements

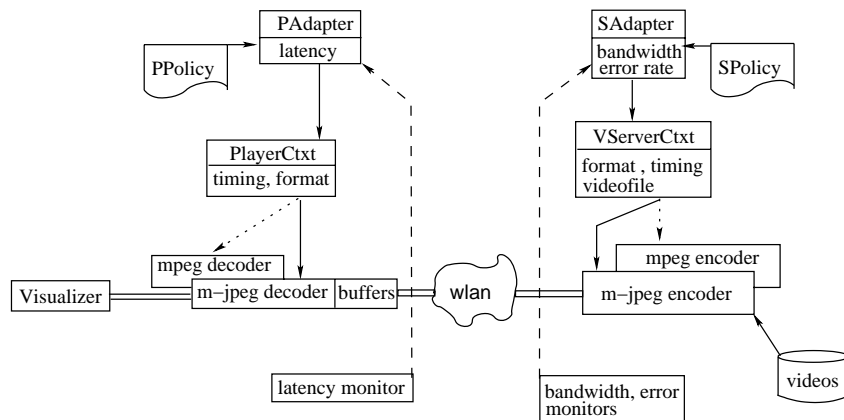


Figure 4. Exemple d'application vidéo avec des composants adaptatifs.

alternatifs implantés en Java qui emploient chacun un encodeur différent pour envoyer le flux vidéo vers le player. La politique d'adaptation du serveur illustrée par le code ci-après consiste à (1) utiliser un encodeur M-JPEG au lieu de MPEG si le taux d'erreur du réseau est important ; ceci permet d'éviter une propagation des erreurs due au fait que dans MPEG le codage d'une trame vidéo dépend de celles qui l'entourent, (2) modifier le taux de bits du flux vidéo émis proportionnellement au changement de la bande passante disponible. L'objet `Context` (`VServerCtxt`) maintient, dans l'attribut `State`, le nom du fichier vidéo, ses informations temporelles (i.e., le point atteint) et le format de codage utilisé. Tandis que les valeurs de la bande passante disponible et le taux d'erreur réseau sont maintenues dans l'attribut `envState` de l'objet `Adapter`.

```
<res name=Bandwidth>
  <exceeds value=100kbs action=change_param(compression_rate=1) />
  <falls value=100kbs action=change_param(compression_rate=7) />
</res>
<res name=ErrorRate>
  <exceeds value=10-6 action=change_to_behavior(M-JPEGCodec) />
  <falls value=10-6 action=change_to_behavior(MPEGCodec) />
</res>
```

Du côté client, l'adaptation consiste à réduire ou augmenter la taille des buffers (utilisés pour recevoir le flux vidéo) en fonction de la latence du réseau. Cette action est du type changement d'un paramètre du comportement courant. Par ailleurs, le player change aussi de décodeur quand le serveur vidéo change le format de codage ce qui constitue une coordination des adaptations entre les deux entités. Les informations temporelles et le format de la vidéo sont encapsulées dans l'objet `Context` du player et la valeur de la latence du réseau est stockée dans son objet `Adapter`.

Enfin, notons que les différentes adaptations ne peuvent avoir lieu durant l'envoi d'une trame vidéo (i.e., section non interruptible) mais avant ou après cet envoi.

5. Conclusion

Nous avons présenté un modèle de composants qui s'adaptent dynamiquement aux variations de leurs conditions d'exécution, notamment celles qui concernent les réseaux sans-fil, par un changement de comportement. La prise des décisions d'adaptation est réalisée par l'interprétation d'une politique d'adaptation écrite dans un script séparément du reste du code du composant. Notre modèle basé sur le design pattern Strategy et la réflexion constitue la brique de base d'un framework que nous sommes en train de développer afin de supporter la programmation d'applications adaptables sur les environnements mobiles. Ce framework contient aussi un mécanisme de détection et notification par événement des changements de l'environnement. Nous avons aussi montré, par une expérimentation sur une application du type vidéo à la demande, la faisabilité des idées que nous avons introduites, notamment la possibilité de réaliser un traitement particulier en basculant entre plusieurs algorithmes alternatifs qui manipulent un état commun tout en préservant l'intégrité de celui-ci.

Dans nos travaux futurs nous envisageons de traiter les problèmes liés à l'adaptation de plusieurs composants s'exécutant au sein d'une même tâche ou à un niveau multi-applicatif. Par exemple il faudrait assurer la cohérence des choix d'adaptation des différents composants, la synchronisation des processus d'adaptation ainsi que la stabilité des adaptations en *chaîne* de plusieurs composants. Par ailleurs nous envisageons aussi de raffiner l'exemple de l'application vidéo en la portant sur un terminal mobile expérimental équipé de plusieurs technologies de réseaux sans fils.

Enfin comme objectif à moyen terme nous envisageons d'utiliser le modèle ACEEL pour le développement de services d'un middleware dédié aux environnements mobiles.

6. Bibliographie

- [ALB 01] AL-BAR A., WAKEMAN I., « A Survey of Adaptive Applications in Mobile Computing », *IEEE International Workshop on Smart Appliances and Wearable Computing*, Phoenix, Az, USA, April 2001, p. 246-251.
- [BLA 97] BLAIR G., COULSON G., DAVIES N., ROBIN P., FITZPATRICK T., « Adaptive Middleware for Mobile Multimedia Applications », *8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97)*, St. Louis, Missouri, mai 1997.
- [Blu] BLUETOOTH SIG, « Bluetooth™ », URL : <http://www.bluetooth.com/>.
- [CHA 99] CHALMERS D., SLOMAN M., « A Survey of Quality of Service in Mobile Computing Environments », *IEEE Communications surveys*, , 1999, online : <http://www.comsoc.org/pubs/surveys/2q99issue/sloman.html>.
- [CHE 00] CHEVERST K., DAVIES N., MITCHELL K., FRIDAY A., EFSTRATIOU C., « Developing a Context-aware Electronic Tourist Guide : Some Issues and Experiences », *CHI 2000 Conference on Human Factors in Computing Systems*, Netherlands, avril 2000, p. 17-24.
- [DAV 96] DAVIES N., FRIDAY A., BLAIR G. S., CHEVERST K., « Distributed Systems Support for Adaptive Mobile Applications », *Mobile Networks and Applications (MONET)*, vol. 1, n° 4, 1996, p. 399-408.
- [FRI 99] FRIDAY A., DAVIES N., BLAIR G., CHEVERST K., « Developing Adaptive Applications : The MOST Experience », *Journal of Integrated Computer-Aided Engineering*, vol. 6, n° 2, 1999, p. 143-157.
- [GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [GSM] « GSM World, the website of the GSM Association », URL : <http://www.gsmworld.com/index.shtml>.
- [HOF 94] HOFMAN G. H., ZAHORJAN J., « The Challenges of Mobile Computing », *IEEE Computer*, vol. 27, n° 4, 1994, p. 38-47.
- [IEE] IEEE 802.11 WG, « IEEE 802.11 Wireless Local Area Networks », URL : <http://grouper.ieee.org/groups/802/11/>.
- [IND 00] INDULSKA J., LOKE S., RAKOTONIRAINY A., ZASLAVSKY A., « Adaptive Enterprise Architecture for Mobile computation », *Workshop on Reflective Middleware, Midd-*

eware 2000, April 2000.

- [JOS 95] JOSHI A., WEERAWARANA S., WEERASINGHE R. A., DRASHANSKY T. T., RAMAKRISHNAN N., HOUSTIS E. N., « A Survey of Mobile Computing Technologies and Applications », rapport n° TR-95-050, 1995, Dept. of Computer Sciences, Purdue University.
- [KAT 94] KATZ R. H., « Adaptation and Mobility in Wireless Information Systems », *IEEE Personal Communications*, vol. 1, n° 1, 1994, p. 6–17.
- [KIC 91] KICZALES G., DES RIVIERES J., BOBROW D. G., *The Art of the Metaobject Protocol*, The MIT Press, 1991.
- [MAE 87] MAES P., « Concepts and Experiments in Computational Reflection », *Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA'87)*, décembre 1987, p. 147-155.
- [MEI 97] MEIJLER T. D., NIERSTRASZ O., « Beyond Objects : Components », PAPAZO-GLOU M. P., SCHLAGETER G., Eds., *Cooperative Information Systems : Current Trends and Directions*, p. 49–78, Academic Press, 1997.
- [Pyt] « Python Language Home Page », url : <http://www.python.org/>.
- [SAT 96] SATYANARAYANAN M., « Mobile Information Access », *IEEE Personal Communications*, vol. 3, n° 1, 1996.
- [VAR 00a] VARSHNEY U., VETTER R., « Emerging Mobile and Wireless Networks », *Communications of the ACM (CACM)*, vol. 43, n° 6, 2000, p. 73-81.
- [VAR 00b] VARSHNEY U., VETTER R. J., KALAKOTA R., « Mobile Commerce : A New Frontier », *IEEE Computer*, vol. 33, n° 10, 2000, p. 32-38.