# Spécification et ordonnancement dynamique d'applications multimédias : l'environnement POLKA

FRANK SINGHOFF, ISABELLE DEMEURE
Ecole Nationale Supérieure des Télécommunications - CNRS URA 820
Département Informatique
46, rue Barrault, 75634 Paris Cedex 13
{singhoff, demeure}@inf.enst.fr

**Résumé**

Cet article traite de la spécification et du support des applications multimédias, et plus particulièrement de celles dont les besoins en ressources sont difficiles à estimer et dont le comportement n'est pas entièrement prédictible. Nous définissons un modèle qui permet de spécifier une application multimédia en termes d'objets, de flux multimédias et de contraintes temporelles sur ces flux. Un algorithme d'ordonnancement qui prend dynamiquement en compte ces contraintes temporelles est décrit informellement. Une mise en œuvre de cet algorithme dans un environnement CORBA et un exemple d'application sont ensuite détaillés.

**Mots clefs**

QoS , Multimédia, CORBA, Ordonnancement temps réel, Contraintes temporelles

## 1 Introduction et motivations

Cet article[1] traite du support d'applications multimédias, et tout particulièrement de leur ordonnancement. Les tâches d'une application multimédia possèdent de nombreuses contraintes temporelles. Celles-ci peuvent être exprimées par le biais d'échéances et de dates d'exécution au plus tôt, qui sont alors utilisées pour modéliser les synchronisations entre et dans les flux de données multimédias[7].

Il est possible de séparer les applications multimédias en deux grandes classes : celles pour lesquelles le comportement, et donc les besoins en ressources, peuvent être précisément déterminés hors ligne, et celles dont le comportement est difficilement prédictible avant son exécution.

Pour la première classe d'applications, beaucoup de travaux appliquent des solutions employées dans les systèmes temps réels fortement contraints. Dans ces derniers, l'application se doit d'être bien maîtrisée ; ceci suppose une parfaite connaissance de l'application avant son exécution. Ces solutions sont très statiques et souvent hors ligne. Une système comme ARTS[27] en est un bon exemple. Toutefois, bien que les applications multimédias et les applications temps réel durs partagent l'obligation de respecter des contraintes temporelles, les systèmes fortement contraints

[1]Ce travail est réalisé dans le cadre d'une collaboration avec l'équipe de Jean-Bernard Stefani au Centre National d'Études des Télécommunications (CNET).

sont utilisés dans des domaines applicatifs où la violation d'une échéance peut entraîner de graves conséquences ; dans une large classe d'applications multimédias (ex : vidéo-conférences, enseignement à distance) on se contentera de noter une baisse de la QoS (Quality of Service) de l'application. L'utilisation de technologies temps réel durs reste toutefois satisfaisante pour ce type d'applications multimédias.

Nous nous intéressons dans cet article à l'ordonnancement d'applications appartenant à la deuxième classe. Pour celles-ci, l'utilisation de techniques issues du temps réel dur est beaucoup moins bien adaptée, et conduit généralement à une surréservation des ressources. En effet, ces applications sont caractérisées par un niveau de dynamicité important. Cette classe comprend les applications où l'utilisateur a la possibilité de modifier dynamiquement la charge du système en exécutant des applications supplémentaires ; ou encore, celles où l'utilisateur a la possibilité de redéfinir la qualité de service souhaitée pendant l'exécution de son application. Enfin, plus généralement, les applications où il est difficile d'évaluer efficacement les ressources nécessaires font aussi partie de cette classe (par exemple, certains flux de données multimédias possèdent des débits variables. C'est le cas des flux MPEG - Moving Pictures Experts Group [11]). Toutes ces applications ont, comme point commun, la difficulté de déterminer statiquement, et de façon précise, leur comportement et leurs besoins en ressources. L'utilisation d'algorithmes empruntés au domaine du temps réel fortement contraint n'est donc pas toujours adaptée (voir [25, 26, 28]).

Certaines équipes ont cependant proposé des solutions permettant le support d'applications dynamiques. C'est le cas de l'université du Massachusett avec le système Spring[21] et de l'université de Caroline du Nord avec Yartos[20]. Toutefois, même pour ces dernières solutions, les mécanismes proposés sont trop généraux et des systèmes plus spécifiques au multimédia doivent être utilisés. Par exemple, ceux-ci ne permettent pas de spécifier aisément la QoS et de gérer les changements de celle-ci ou de la charge du système.

D'autres travaux ont proposé des systèmes offrant une meilleur prise en compte des contraintes temporelles, et plus généralement de la gestion de la QoS. Citons par exemple le projet Pegasus[17] de l'université de Twente et le projet Maruti[4] de l'université du Maryland. Ces deux solutions offrent la possibilité de déterminer des "calendriers" pour l'allocation des ressources (du processeur entre autre). Dans le cas de Maruti, tous les calendriers sont évalués hors-ligne. Dans Pegasus, un nouvel ordonnancement et une renégociation de la QoS sont réalisés durant la vie du système, lors de l'arrivée ou du départ d'une application. Dans les deux cas, un certain niveau de garantie de QoS est donné. Toutefois, ces systèmes demandent encore une fois une connaissance trop précise des tâches et de leurs besoins.

Dans cet article, nous présentons l'environnement POLKA. Il permet une prise en compte dynamique des contraintes de QoS dans l'ordonnancement d'applications multimédias dont les besoins en ressources sont difficiles à déterminer. Le développeur ne précise aucune directive d'ordonnancement. Celles-ci sont automatiquement déduites par POLKA à partir des contraintes de QoS spécifiées par le développeur. Notre ordonnanceur fournissant un service de type "meilleur effort", c'est donc au

concepteur de prévoir plusieurs niveaux de QoS afin de permettre sa dégradation ou son amélioration. Enfin, cette technique permet la construction d'applications portables et évolutives puisque l'adjonction de tâches supplémentaires dans une application, n'oblige pas le développeur à revoir son ordonnancement, et donc son code.

Le plan de cet article est le suivant : nous définissons le modèle manipulé par l'utilisateur pour spécifier une application multimédia dans le chapitre 2. Dans le chapitre 3, nous expliquons succinctement comment l'ordonnanceur exploite ces informations. Le chapitre 4 est dédié à la description de notre implantation actuelle et nous illustrons dans le chapitre 5 son utilisation par la mise en œuvre d'une application multimédia. Nous concluons dans le chapitre 6.

# 2 Modèle de spécification d'une application multimédia

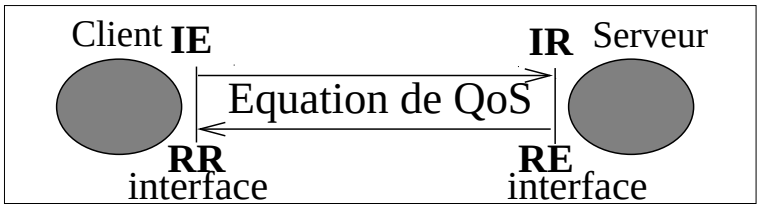


FIG. 1 – *Points d'observation durant une invocation de méthode*

Pour le modèle de spécification de POLKA (voir [5, 6]) une application est un ensemble d'objets qui coopèrent pour la mise en œuvre de différents flux multimédias[10]. Nous définissons un objet par une unité d'encapsulation de code et de données. Un objet peut être animé par une ou plusieurs activités. L'activité est l'unité d'ordonnancement utilisée par l'ordonnanceur du système d'exploitation. Un objet peut exporter une interface qui fournit le prototype des méthodes qui peuvent être invoquées par les autres objets. Par la suite, nous nommerons "client" les objets qui n'exportent pas d'interface, et "serveur" les autres. Les objets interagissent uniquement par appels de méthodes : ils ne communiquent pas par mémoire partagée.

Une fois les flux définis, l'utilisateur doit alors spécifier les interfaces d'objets ainsi que les contraintes de QoS sur les flux. Les équations de QoS sont rédigées dans la logique temporelle QL[23]. Ces équations expriment des contraintes d'échéance et de date d'exécution au plus tôt entre des événements observables dans le système. Les événements observables durant l'exécution d'une application sont définis par :

**Définition 1 (événements)**

- *Soit l'ensemble* $\mathcal{P} = \{IE, IR, RE, RR\}$. *Chaque élément de* $\mathcal{P}$ *correspond à un événement observable durant l'invocation d'une méthode d'un objet. La figure 1 nous montre ces événements qui sont, l'émission d'une invocation (événement*

*IE) par l'initiateur ou client, la réception chez le serveur de cette invocation (événement IR), l'émission de la réponse qui fait suite (événement RE), et la réception de la réponse par le client (événement RR). Notons que, dans le cas d'une invocation asynchrone,* $\mathcal{P} = \{IE, IR\}$.

- *Soit* $\mathcal{I} = \{i_1, ..., i_n\}$*, l'ensemble des interfaces d'objets définies dans une application donnée.*
- *Soit* $\mathcal{M}_i$*, l'ensemble des méthodes d'une interface* $i$.
- *Soit* $\mathcal{O}_i$*, l'ensemble des instances de l'interface* $i$.

*L'ensemble* $\mathcal{E}$ *des événements observables dans une application est défini par:*

$$\mathcal{E} = \bigcup_{\forall\ i\ \in\ \mathcal{I}} \mathcal{E}_i$$

*où* $\mathcal{E}_i$ *est l'ensemble défini par le produit cartésien des ensembles* $\mathcal{P}$, $\mathcal{O}_i$ *et* $\mathcal{M}_i$.

La notation qui sera utilisée dans cet article pour désigner un élément de $\mathcal{E}$ consiste à concaténer le nom de l'objet, le nom d'une de ses méthodes ainsi qu'un élément de $\mathcal{P}$. Par exemple, la notation $r.ope1.IE$ désignera l'événement associé à l'envoi d'une invocation de la méthode $ope1$ sur l'objet $r$. Les équations QL utilisées par POLKA utilisent les notations supplémentaires suivantes[24] :

$$\begin{cases} \forall\ n\ \tau(e, n+k) - \tau(e', n) < j_1\ (1) \\ \qquad\qquad ou \\ \forall\ n\ \tau(e, n+k) - \tau(e', n) > j_2\ (2) \end{cases}$$

Avec :

- $n$ est une variable entière . Une méthode $ope1$ d'un objet $r$ peut être invoquée plusieurs fois. Il y a donc plusieurs occurrences des événements $\{r.ope1.IE, r.ope1.IR, r.ope1.RE, r.ope1.RR\}$. La variable $n$ permet de désigner sans ambiguïté ces différentes occurrences.
- $\{j_1, j_2, k\} \in \mathcal{R}^+$, $\{e, e'\} \in \mathcal{E}$.
- $\tau(e, n)$ est un opérateur qui fournit la date absolue de l'occurrence $n$ de l'événement $e$.
- Les équations de la forme (1) sont appelées "équations d'échéances" et spécifient un délai maximal entre la fin d'un premier événement et la fin d'un deuxième. Les équations de la forme (2) sont appelées "équations de distance" et spécifient un délai minimal entre la fin d'un premier événement et le début d'un deuxième.

Ainsi, l'équation $\tau(r.ope1.RR, n+1) - \tau(r.ope1.RR, n) < 10\ ms$ spécifie que moins de 10 milli-secondes doivent s'écouler entre deux occurrences successives de l'événements $r.ope1.RR$. Ce modèle permet aisément la spécification de flux multimédias : un flux multimédia est alors une suite d'invocations d'une même méthode produisant différents événements contraints par un ensemble d'équations QL.

# 3 L'ordonnanceur de POLKA

Nous avons défini le modèle utilisé par le développeur d'application. Nous regardons dans ce chapitre comment les informations données par ce modèle sont exploitées par l'ordonnanceur. L'algorithme d'ordonnancement de POLKA est un algorithme orienté échéances (EDF [2][13]). L'ordonnancement est fait en ligne, sans contrôle d'admission. Il est non préemptif et non oisif. Nous nous plaçons uniquement, pour le moment, dans un environnement mono-processeur.

L'ordonnanceur utilise deux informations : les équations de QoS et des graphes de dépendances. Les graphes de dépendances sont construits par l'analyse du code des objets de l'application. Chaque nœud d'un graphe constitue un événement observable. Un graphe peut être défini de la façon suivante :

**Définition 2 (Graphe de dépendances)** *Un graphe de dépendances $\mathcal{D}$ est défini par le couple $\mathcal{D} =< \mathcal{S}, \mathcal{A} >$, où $\mathcal{S}$ est l'ensemble des nœuds du graphe et $\mathcal{A}$ est l'ensemble des arcs du graphe. Un graphe de dépendances est orienté et connexe. Chaque nœud de $\mathcal{S}$ est un élément de $\mathcal{E}$. Soit $\mathcal{S}_i$ et $\mathcal{S}_j$, deux nœuds du graphe ; un arc $\mathcal{A}_k$ d'origine $\mathcal{S}_i$ et de destination $\mathcal{S}_j$, exprime que les instructions associées au nœud $\mathcal{S}_j$ ne pourront s'exécuter qu'après la fin de l'exécution des instructions associées au nœud $\mathcal{S}_i$.*

A chaque activité est associé un graphe de dépendances. Celui-ci est construit en parcourant le code des activités et en isolant les invocations de méthodes dont les exécutions vont engendrer l'occurrence d'événements observables (nous en verrons un exemple avec la figure 2). Un graphe de dépendances découpe le code d'une activité en portions qui sont délimitées par deux événements observables. Le premier événement observable est le début de la portion de code. Le deuxième événement observable est la fin de la portion de code, tout en étant aussi le début de la portion de code suivante. La portion de code est l'unité d'ordonnancement. A chaque événement observable, nous associons une échéance et une date au plus tôt. Ces informations d'ordonnancement se rapportent donc à la portion de code suivant immédiatemment l'événement en question. Chaque activité possède un nœud courant qui signale quelle est la prochaine portion de code à exécuter. Par la suite, nous appellerons activité courante, l'activité qui détient le processeur.

[2] **EDF** pour **E**arly **D**eadline **F**irst.

Lorsqu'une unité d'ordonnancement se termine, l'ordonnanceur exécute trois traitements :

1. Le nœud courant de l'activité courante est modifié : il est initialisé avec son nœud successeur.

2. L'ordonnanceur met à jour les échéances des nœuds qui peuvent être déterminées suite à la fin de l'unité d'ordonnancement. Les échéances et dates au plus tôt associées aux unités d'ordonnancement sont des dates absolues. Elles sont calculées grâce aux équations de QoS.

3. L'ordonnanceur élit la prochaine unité d'ordonnancement selon une politique EDF. Toutes les unités d'ordonnancement dont la date d'exécution au plus tôt est inférieure à la valeur courante de l'horloge du système ne sont pas éligibles.

Avant de regarder un exemple, nous expliquons comment les échéances et les dates au plus tôt sont calculées lorsqu'une unité d'ordonnancement est terminée. Une équation de QoS relate un lien de causalité (au sens définit par Lamport dans [12]) entre les deux événements qui la compose. L'ordonnanceur exploite cette information pour évaluer les échéances et les dates au plus tôt. Considérons les notations suivantes :

- Soit $i$ une variable désignant la dernière occurrence d'un événement. Soit $i'$ une variable désignant la prochaine occurrence de ce même événement (on a donc $i' = i + 1$).
- Soit $\delta(i')$, l'échéance absolue de la portion de code associée à $i'$.
- Soit $\gamma(i')$, la date d'exécution au plus tôt de la portion de code associée à $i'$.
- Soit $\tau(i)$, la date de fin d'exécution de la portion de code associée à $i$.
- Soit $j_1$, le membre de droite d'une équation d'échéance et $j_2$, le membre de droite d'une équation de distance.

Alors, $\delta(i')$ et $\gamma(i')$ sont évalués de la façon suivante :

$$\begin{cases} \delta(i') = min(\delta(i'), \tau(i) + j_1) \\ \gamma(i') = max(\gamma(i'), \tau(i) + j_2) \end{cases}$$

La présence des opérateurs $min$ et $max$ permet de tenir compte des valeurs de $\delta(i')$ et de $\gamma(i')$ calculées antérieurement. Au démarrage de l'application, on a $\forall i \ : \ \delta(i) = +\infty$ et $\gamma(i) = 0$.

La difficulté consiste ensuite à affecter une échéance à chaque nœud du graphe tout en respectant les échéances de ses nœuds successeurs. Introduisons la relation de précédence $\prec$ telle que $i \prec j$ signifie que la tâche $j$ ne peut pas commencer son exécution avant que la tâche $i$ termine la sienne. Supposons qu'il existe deux nœuds $i$ et $j$ tel que $(i \prec j) \ \wedge (\delta(j) < \delta(i))$. Dans ce cas de figure, on voit que tout en

respectant $\delta(i)$, il est possible que $\tau(i) > \delta(j)$, et que donc, $j$ ne respectera pas son échéance. Ce problème fut étudié très tôt par Blazewicz[2]. La solution proposée est d'évaluer l'échéance du nœud $i$ par la formule suivante :

$$\forall\ i\ :\ \delta(i) = min(\delta(i), min(\forall\ j\ :\ \delta(j) \mid i \prec j))$$

en d'autres termes, cette solution force un nœud à acquérir la plus petite des échéances de ses nœuds successeurs, s'il en existe une qui est plus petite que la sienne. Cette technique est utilisée dans POLKA.

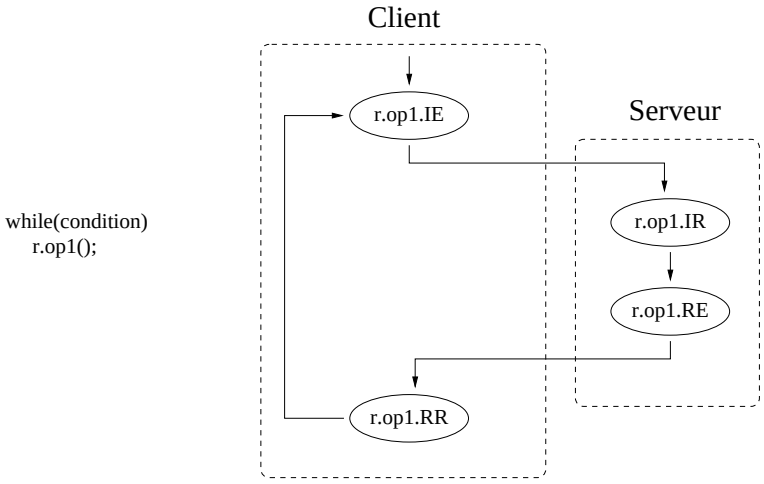


FIG. 2 – *Exemple d'un code et de son graphe de dépendances*

Illustrons les principes énoncés ci-dessus par un exemple simple. Regardons le graphe de dépendances de la figure 2 et supposons que ce graphe modélise une activité dont le code est constitué d'une boucle infinie sur l'invocation de la méthode $ope1$ d'un objet $r$. Supposons maintenant que le système d'équations suivant

$$\begin{cases} \tau(r.ope1.RR, n+1) - \tau(r.ope1.RR, n) < 10\ ms \\ \tau(r.ope1.RR, n+1) - \tau(r.ope1.RR, n) > 5\ ms \end{cases}$$

soit associé à cette activité et regardons comment l'ordonnanceur calcule les échéances et les dates au plus tôt des différents nœuds du graphe.

Lors du démarrage de l'activité, l'ordonnanceur ne peut calculer aucune échéance des événements $\{r.op1.IE, r.op1.IR, r.op1.RE, r.op1.RR\}$ et celles-ci sont alors initialisées par des valeurs arbitraires. Lorsque l'activité termine la première exécution du nœud $r.op1.RR$, l'ordonnanceur peut enfin évaluer les échéances des prochains événements $\{r.op1.IE, r.op1.IR, r.op1.RE, r.op1.RR\}$. Le calcul se fait alors en deux étapes :

- L'ordonnanceur met d'abord à jour la prochaine échéance du nœud $r.op1.RR$ en ajoutant 10 à la date de fin d'exécution de la première occurrence de

$r.op1.RR$. Ainsi, l'échéance de la deuxième occurrence de $r.op1.RR$ vaut $\delta(r.op1.RR) = \tau(r.op1.RR) + 10$.

- Puis, il applique la formule de Blazewicz pour répercuter cette nouvelle échéance jusqu'au nœud $r.op1.IE$. Cette deuxième étape consiste a parcourir le graphe de $r.op1.RR$ jusqu'à la racine ($r.ope1.IE$) et d'effectuer pour chaque nœud $i$ : $\delta(i) = min(\delta(i), \delta(r.op1.RR))$. La contrainte de distance n'est pas répercutée vers le haut du graphe, et on a donc $\gamma(r.op1.RR) = \tau(r.op1.RR) + 5$, et pour tous les autres nœuds $i$ du graphe : $\gamma(i) = 0$.

# 4 L'implantation actuelle et son fonctionnement

Notre solution consiste donc à spécifier une application en termes d'objets et d'équations de QoS. Pour l'implantation de ce modèle, nous utilisons un ORB[3] au standard CORBA[18][4].

CORBA est un standard qui a été défini par l'OMG[5], un consortium essentiellement constitué d'industriels. Ce standard définit un modèle d'objets distribués. Les objets interagissent entre eux par le biais d'invocations de méthodes et CORBA fournit les services nécessaires à la conduite de ces invocations, avec les deux objectifs suivants :

- L'interopérabilité entre objets. Les interfaces des objets sont définies dans un langage commun : l'IDL[6]. Toutefois, leurs implémentations sont libres. Elles peuvent être écrites dans des langages différents. CORBA spécifie alors la correspondance entre l'IDL et le langage utilisé pour implanter les objets. CORBA prend aussi en compte les problèmes d'hétérogénéité de machines, de systèmes et de réseaux.

- De transparence à la localité. Les clients accèdent aux objets d'une manière identique qu'ils soient sur la machine locale ou sur une machine distante.

Une application POLKA est donc constituée d'un ensemble d'objets CORBA. Dans notre implantation actuelle, ceux-ci sont écrits en C++ et nous utilisons un ORB disponible gratuitement, fonctionnant sur Solaris, et diffusé par le laboratoire commun de recherche d'Olivetti et d'Oracle.

Aujourd'hui, il est communément reconnu que le standard CORBA n'est pas adapté aux applications possédant des contraintes temporelles et de nombreuses équipes ont proposé des solutions pour le support des applications multimédias[16, 15, 1], et plus généralement pour les applications temps réels[3, 31]. Les principaux reproches faits à CORBA sont l'impossibilité de spécifier la QoS, le manque de

[3] **ORB** pour **O**bject **R**equest **B**roker.
[4] **CORBA** pour **C**ommon **O**bject **R**equest **B**roker.
[5] **OMG** pour **O**bject **M**anagement **G**roup.
[6] **IDL** pour **I**nterface **D**efinition **L**anguage.

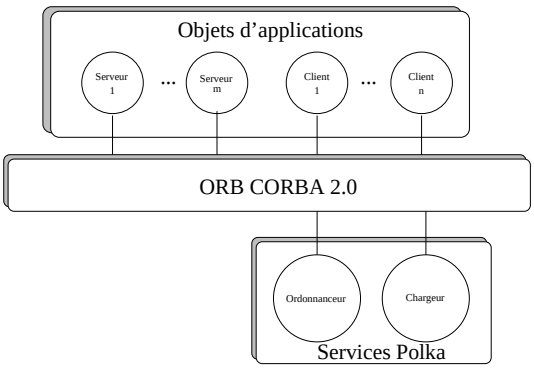


FIG. 3 – *Architecture de notre prototype*

mécanismes pour la garantir, le manque de services temps réel tels que les services d'événements temps réel, et plus généralement, l'impossibilité pour les applications de manipuler des informations sur les ressources de bas niveau. Par exemple, un ORB temps réel devrait fournir une gestion fine de la BOA[7], ce qui permettrait à un client de fixer la priorité de l'activité qui va traiter son invocation chez le serveur, de choisir les algorithmes d'ordonnancement, etc.

Pour les applications multimédias, il est important que l'ORB supporte la notion de flux. Ici, deux approches sont généralement suivies. Certaines équipes modifient le langage IDL pour permettre la spécification des flux. Ces solutions fournissent alors deux interfaces par flux : une interface de contrôle (pour créer les flux, les supprimer, les suspendre, etc ), une interface pour transmettre les données[16]. D'autres utilisent uniquement l'interface de contrôle et exploitent un autre support de communication pour les données afin d'obtenir de meilleures performances[15].

Notre approche actuelle consiste, pour l'instant, à modéliser un flux par une suite d'invocations d'une interface. Nous n'excluons pas toutefois l'utilisation d'interfaces de contrôle. Pour spécifier la QoS, nous utilisons bien évidemment le modèle donné dans le chapitre 2, mais nous ne nous préoccupons pas des garanties sur le respect de celle-ci. A cause des applications que nous visons, nous nous plaçons dans un contexte où le service offert est de type "meilleur effort".

L'environnement POLKA est constitué d'outils de compilation et d'un environnement d'exécution. Les outils de compilation ont pour but la génération des souches et des squelettes CORBA ainsi que l'analyse du code des objets et des équations de QoS. L'environnement d'exécution est implanté sous forme d'un démon. Ces outils sont les suivants :

- *i2p* pour IDL *to* POLKA. Ce compilateur génère les souches et squelettes CORBA des interfaces IDL spécifiées par l'utilisateur. Ces souches et squelettes sont instrumentés pour invoquer l'ordonnanceur de POLKA.

- *c2p* pour C++ *to* POLKA. Ce compilateur analyse le code C++ de l'application et les équations de QoS. Puis, il découpe le code en graphes de dépendances.

[7] **BOA** for **B**asic **O**bject **A**daptor.

```
module polka {
    interface scheduler {
        void callScheduler(in string nextNode);
        void notifyEndedClient(void);
        void resumeClient(in long uniqueId);
        void suspendClient(in long uniqueId);
        long getStatus(in long uniqueId);
        };
    interface loader {
        long subscribeServer(in string name);
        long subscribeClient(in string name, out long uniqueId);
        long deleteClient(in string name, in long  uniqueId);
        long deleteServer(in string name);
        };
    };
```

FIG. 4 – *Interface* IDL *de* POLKA

– Le démon *polkad*. Il contient deux objets qui sont l'ordonnanceur POLKA et un objet nommé chargeur. Ces deux objets partagent le même espace d'adressage. L'objet chargeur s'occupe d'initialiser les données nécessaires à l'ordonnanceur lors de l'arrivée de nouvelles activités. Ces données sont essentiellement constituées des équations de QoS et des graphes de dépendances. Le chargement en mémoire est réalisé lors de la création d'une activité ou d'un objet CORBA par le biais d'un "enregistrement" (invocation des méthodes *subscribeServer*() et *subscribeClient*() ). L'ordonnanceur est invoqué quand un client appelle une méthode d'un objet CORBA. Il gère l'allocation du processeur en respectant au mieux les contraintes exprimées par les équations de QoS données. L'appel de la méthode *callScheduler*() réalise l'invocation de l'ordonnanceur. Celle-ci intervient lors de l'exécution d'une souche ou d'un squelette CORBA. Les méthodes *suspendClient*(), *resumeClient*() et *getStatus*() permettent respectivement de suspendre une activité, de reprendre son exécution et d'obtenir son état courant. La méthode *notifyEndedClient*() permet de réallouer le processeur lorsqu'une activité est terminée. Notre ordonnanceur exploite l'ordonnancement préemptif à priorité fixe disponible sur Solaris.

# 5 Un exemple d'application

L'architecture définie dans le chapitre précédent a pu être testé par le développement d'une application qui émule une application de télévision interactive[8]. Nous décrivons maintenant cette application afin d'illustrer le fonctionnement de POLKA.

L'application simule la diffusion de la cérémonie des Césars. Les téléspectateurs


FIG. 5 – *Exemple d'une application utilisant* POLKA

peuvent intervenir lors du vote pour les films nominés. Ils disposent, à cet effet, d'une télécommande par laquelle ils peuvent choisir de visionner un ou plusieurs des films nominés, puis de voter pour l'un d'entres eux. Initialement, une seule fenêtre est active. Cette dernière délivre la séquence vidéo et audio MPEG de la cérémonie. L'utilisateur peut ouvrir d'autres fenêtres afin de visionner plusieurs films simultanément. Seule la bande sonore du dernier film ouvert est active. On voit ici l'importance de l'impact d'une réservation de ressources pour ce type d'applications. En effet, l'exécution d'un flux MPEG est coûteux en ressource processeur et effectuer une réservation au pire cas, c'est à dire pour le nombre maximum de films visibles en même temps, impliquerait une sur-réservation qui pénaliserait les utilisateurs ne regardant qu'un faible nombre de films.

```
module mpeg {
        interface mpegClient {
                long soundDecoder(in SoundFrame f);
                long imageDecoder(in ImageFrame f);
        };
        interface mpegServer {
                long getSoundFrame(out SoundFrame f);
                long getImageFrame(out ImageFrame f);
        };
};
```

FIG. 6 – *Interfaces IDL de l'application de télévision interactive*

L'application est constituée de deux processus (que nous nommerons *client* et *serveur* pour plus de commodité dans la suite de cet article). Les interfaces IDL des objets CORBA manipulés par ceux-ci sont données dans la figure 6. Le client

contient deux activités Solaris par film en cours. Pour chaque film, une activité est dédiée à la restitution du flux audio et l'autre, à la restitution du flux vidéo. Le serveur gère une instance de l'interface IDL *mpegServer* par film. Lorsqu'une activité du client souhaite obtenir un paquet de données du flux multimédia auquelle elle est associée, elle invoque l'interface *mpegServer* correspondante chez le serveur. Le processus serveur se contente alors de lire sur disque les données vidéo ou audio, puis de les renvoyer au client en retour de son invocation synchrone. Dans l'espace d'adressage du client, il existe une instance de l'interface *mpegClient* pour chaque film en cours. Cette interface encapsule un décodeur MPEG audio et vidéo. Une activité du client exécute durant toute la durée du film la séquence d'invocations suivante :

1. Invoquer la méthode *getSoundFrame*() (respectivement *getImageFrame*() ) de l'objet CORBA correspondant à son flux afin d'obtenir un paquet de données audio (respectivement vidéo).

2. Fournir ces données audio (respectivement vidéo) à son interface *mpegClient* par le biais de la méthode *soundDecoder*() (respectivement *imageDecoder*() ) afin que celles-ci soient décompressées, puis envoyées sur le périphérique de sortie.

3. Boucler sur ces deux invocations jusqu'à la fin du flux.

Lorsque cette application est compilée, puis exécutée comme une application CORBA normale, POLKA n'est pas invoqué et aucune gestion de la QoS n'est effectuée. Dès le lancement d'un troisième film, le flux audio de celui-ci n'est plus audible sur une station de travail Ultra Spark 1, car une grande partie des ressources processeurs sont consommées par la restitution des trois flux vidéo. De plus, aucune synchronisation n'est réalisée entre et dans les différents flux multimédias constituant l'application.

Nous regardons maintenant comment avec des équations de QoS écrites en QL, l'utilisateur peut spécifier les synchronisations intra et inter flux. La première étape consiste à choisir quels sont les événements de l'application qui seront utilisés pour exprimer les contraintes temporelles. Dans cette application, nous pouvons choisir, par exemple, les événements induits par l'invocation des méthodes de l'interface IDL *mpegServer*. Seuls les événements produits par les méthodes de *mpegServer* constitueront des événements observables par l'ordonnanceur, et les autres interfaces IDL seront compilées normalement par *i2p*[8]. La figure 7 nous donne le graphe de dépendances qui est exploité par l'ordonnanceur. Un graphe identique existe pour les flux vidéo. Par la suite, l'utilisateur doit spécifier les contraintes temporelles sur les différents événements identifiés. La figure 8 définit un système d'équations spécifiant pour le film dont l'objet CORBA correspondant est $m$, les synchronisations intra flux

[8]Un fichier permet de préciser à *i2p* quelles sont les interfaces et méthodes qui seront traitées par l'ordonnanceur, et qui doivent donc faire l'objet de la production de souches et de squelettes spécifiques.

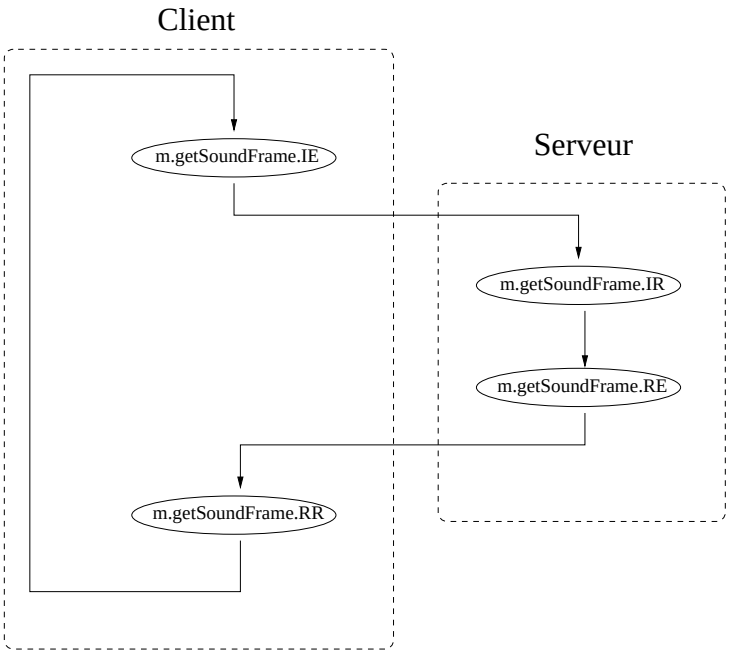


FIG. 7 – *Graphes de dépendances de l'application*

et la synchronisation voix-lèvres. Les équations (1) et (2) modélisent une gigue sur la livraison des paquets audio, de même que les équations (3) et (4) spécifient la gigue pour les paquets vidéo. Enfin, les équations (5) et (6) expriment la synchronisation voix-lèvres.

| n° | Équations |
|---|---|
| 1 | $\tau(m.getSoundFrame.RR, n+1) - \tau(m.getSoundFrame.RR, n) < x_1$ |
| 2 | $\tau(m.getSoundFrame.RR, n+1) - \tau(m.getSoundFrame.RR, n) > x_2$ |
| 3 | $\tau(m.getImageFrame.RR, n+1) - \tau(m.getImageFrame.RR, n) < x_3$ |
| 4 | $\tau(m.getImageFrame.RR, n+1) - \tau(m.getImageFrame.RR, n) > x_4$ |
| 5 | $\tau(m.getSoundFrame.RR, n) - \tau(m.getImageFrame.RR, n) < x_5$ |
| 6 | $\tau(m.getSoundFrame.RR, n) - \tau(m.getImageFrame.RR, n) > x_6$ |

FIG. 8 – *Synchronisations intra et inter flux sur une séquence MPEG*

Signalons que les équations données dans la figure 8, ne sont pas suffisantes si l'on souhaite spécifier un changement de QoS lors du lancement ou lors de la terminaison d'un film MPEG. Le basculement d'une qualité de service vers une autre est actuellement réalisé par deux techniques :

– Lors de l'occurrence d'événements spéciaux (appelés "déclencheurs"). Ces événements peuvent être l'arrivée ou la fin d'un flux multimédia par exemple.

– Par l'exploitation des informations retournées par l'ordonnanceur de POLKA (ex : respect ou non des échéances).

Supposons maintenant, qu'en plus des synchronisations inhérentes aux différents flux multimédias, l'utilisateur souhaite spécifier qu'en cas de modification du nombre de films regardés, un changement de QoS intervienne. Dans ce cas, l'utilisateur devra fournir plusieurs jeux d'équations. Chaque jeu d'équation sera activé à l'occurrence d'un événement déclencheur. Dans cette application, une solution possible est de considérer comme déclencheur le début et la fin d'un film. Aujourd'hui, POLKA permet de préciser que la valeur $i$ du membre de droite d'une équation doit être remplacée par une valeur $j$ à l'occurrence du dit événement. Cette fonctionnalité reste, dans certains cas, lourde à utiliser et dans l'avenir, l'utilisateur pourra manipuler des valeurs relatives : donner une valeur $j$ sur lequel on devra appliquer un opérateur[9] et $i$, pour obtenir la nouvelle contrainte de QoS.

Cette application a permi de montrer que POLKA est à même de gérer la qualité de service spécifiée par le concepteur. En particulier, POLKA peut privilégier un flux applicatif par rapport à un autre. Ainsi, dans l'application décrite dans ce chapitre, il a été possible de montrer que si l'on définissait convenablement les équations de QoS, on pouvait favoriser la qualité du flux audio et du dernier film lancé, au détriment bien sûr des flux vidéo lancés antérieurement. Toutefois, l'environnement utilisé (CORBA sur Solaris) et notre ordonnanceur ajoutent un surcoût non négligeable et nous n'avons pas pu montrer, pour l'instant, que POLKA offrait une précision suffisante pour une mise en œuvre satisfaisante de synchronisations de granularité fine comme la synchronisation voix-lèvres.

# 6 Conclusions

Dans cet article, nous avons décrit comment nous modélisons et exécutons une application multimédia avec POLKA. POLKA permet au concepteur, à partir d'une description IDL de son application et d'équations de QoS, d'ordonnancer automatiquement une application présentant des contraintes temporelles. L'ordonnancement offert par POLKA est de type "meilleur effort" et se limite, pour l'instant, à un environnement mono-processeur. POLKA répond bien aux besoins de dynamicité requis par les applications multimédias dont le comportement est partiellement prédictible et où les besoins en ressources sont difficiles à évaluer. Notre environnement ne fournit pas de garantie de qualité de service ; c'est donc au concepteur de préciser les informations qui, en cas de surcharge, permettront de faire évoluer l'ordonnancement vers le comportement souhaité.

Notre implantation actuelle est basée sur un bus à objets CORBA 2 sur Solaris. Ce choix a simplifié sa mise en œuvre, et facilitera l'implantation de notre version distribuée grâce aux propriétés de transparence à la localisation et d'interopérabilité de CORBA. En revanche, notre application de test qui manipule une importante quantité de données multimédias, nous a rappelé que le surcoût impliqué par CORBA

[9] Addition, soustraction, division ou multiplication.

et notre ordonnanceur sont loin d'être négligeables. Comme d'autres équipes (voir [9, 1]), nous pensons que la résolution de ces problèmes de performances passe, entre autre, par l'utilisation d'un bus à objets et d'un système d'exploitation mieux adaptés à notre problématique. Les micro-noyaux tels qu'Exokernel[14, 19] et L4[29], et les systèmes réflexifs comme Spin[22] ou Apertos[30], en sont de bons exemples. Dans ce cadre, POLKA devrait être intégré comme une bibliothèque système dans l'espace d'adressage utilisateur fournissant une abstraction de tâches et les outils d'ordonnancement associés. Enfin, nous travaillons actuellement sur une version permettant de distribuer sur plusieurs processeurs les activités d'une même application. Ces travaux sont réalisés en parallèle avec une description formelle de notre ordonnanceur et l'étude de ses conditions d'ordonnançabilité. Nous espérons que les résultats déboucheront vers une évolution de notre plate-forme actuelle permettant le support des applications distribuées avec une efficacité plus satisfaisante.

# Références


[1] Y. Y. Al-Salqan. Mediaware : A Distributed Multimedia Environment With Interoperability. In Proceedings of the 4th IEEE Workshop on Enabling Technologies : Infrastructure for Collaborative Enterprises. Los Alamitos, April 1995.

[2] J. Blazewicz. Scheduling Dependant Tasks with Different Arrival Times to Meet Deadlines. In. Gelende. H. Beilner (eds), Modeling and Performance Evaluation of Computer Systems, Amsterdam, Noth-Holland, 1976.

[3] D. Levine T. Harrison D. C. Schmidt A. Gokhale C. Cleeland. TAO: a High-performance End System Architecture for Real-time CORBA. Response to the OMG Real-time Special Interest Group Request for Information, 1997.

[4] M. Saksena J. da Silva A. K. Agrawala. Design and Implementation of Maruti-II. pages 72–102. In Principles of Real-Time Systems, Sang son (ed.). Chapter 4, 1994.

[5] J. Farhat-Gissler. Ordonnancement automatique d'applications présentant des contraintes de QoS temporelles. Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications, septembre 1996.

[6] I. Demeure J. Farhat-Gissler F. Gasperoni. A Scheduling Framework for the Automatic Support of Temporal QoS Constraints. In proceedings of the Fourth International Workshop on Quality of Services (IWQoS), Paris, March 1996.

[7] A. Vogel B. Kerhervé G. Von Bochmann J. Gecsei. Distributed Multimedia and QoS : A Survey. *IEEE Multimédia*, 2(2):10–19, 95.

[8] F. Girodengo. Création et développement d'une application multimédia interactive sur POLKA. Mémoire de fin d'études ENST, Juillet 1997.

[9] R. M. Edmundo H. S. Flavio. ODP-based QoS Specification for the Multiware Platform. pages 45–53. Andeas Vogel (Eds.), Quality of Service - Description Modelling and Management, Proceedings of the 4 international IFIP Workshop on Quality of Service, Paris 6-8 Mars, 1996.

[10] J.B. Stefani G.S. Blair G. Coulson M. Papathomas P. Robin F. Horn L. Hazard. A programming model and system infrastructure for real-time synchronization in distributed multimedia systems. *IEEE Journal on selected areas in communications*, 14(1), January 1996.

[11] ISO. Press Release, 29th Meeting of JTC 1/SC 29/WG 11. number 1110, March 1995.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jully 1978.

[13] C. L. Liu J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environnment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[14] J. Liedtke. Toward real Microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[15] B. Murphy G. Mapp. Integrating Multimedia Streams into a Distributed Computing System. Proceedings of Multimedia Computing and Networking, San Jose, CA, USA, January 1996.

[16] C. Trompette F. Merciol. Introducing Multimedia in COOL OODS : An ODP-based Approach. In ERSADS'95, 1995.

[17] P. Sijben S. Mullender. Quality of Service in Distributed Multimedia Systems. in Trends in distributed systems, Springer Lectures Notes on computing systems, TREVS 1161, 1996.

[18] OMG TC Document. The Common Object Request Broker : Architecture and Specification. June 1995.

[19] D. R. Engler M. F. Kaashoek J. O'Toole. Exokernel : An Operating System Architecture for Application-Level Ressource Management. pages 251–266. In Proceedings of the 15th ACM SOSP, December 1995.

[20] K. Jeffay D.L. Stone D.E. Poirier. YARTOS: Kernel support for efficient, predictable real-time systems. *Proc. joint Eighth IEEE Workshop on Real-Time Operating Systems and Software and IFAC/IFIP Workshop on Real-Time Programming, Atlanta. Real-Time Systems Newsletter*, 7(4):7–12, Mai 1991.

[21] J.A. Stankovic K. Ramamrithan. The Spring kernel : A New Paradigm for Hard-Real Time Operating Systems. *ACM Operating Systems Review*, 23(3):54–71, Jully 1989.

[22] B. Bershad S. Savage P. Pardyak E. Gun Sirer. Extensibily, Safety and Performance in the SPIN Operating System. pages 267–284. ACM SIGOPS'95, December 1995.

[23] J. B. Stefani. Computational Aspects of QoS in an object-based, distributed systems architecture. 3rd Workshop on Responsive Computer systems, Lincoln, NH, USA, September 1993.

[24] ARCADE team. Toward the integration of Real Time and QoS handling in ANSA architecture. Centre National d'Etudes des Télécommunications/Centre Paris A, technical note NT/PAA/TSA/TLR/3498, Jully 1993.

[25] C. W. Mercer S. Savage H. Tokuda. Applying Hard Real-Time Technology to Multimedia Systems. In the Proceedings of the Workshop on the Role of Real-time in Multimedia/Interactive Computing Systems, Raleigh-Durham, NC, December 1993.

[26] C. W. Mercer S. Savage H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Mai 1994.

[27] H. Tokuda and C. W. Mercer. ARTS kernel: A distributed real-time kernel. *ACM Operating Systems Review*, 23(3), 1989.

[28] V. Baiceanu C. Cowan D. McNamee C. Pu J. Walpole. Multimedia Applications Require Adaptive CPU Scheduling. Workshop on Resource Allocation Problems in Multimedia Systems, Washington DC, December 1996.

[29] H. Hartig M. Hohmuth J. Liedtke S. Schonberg J. Wolter. The Performance of micro-Kernel-Based Systems. 16th ACM Symposium on operating Systems Principles in Saint Malo, October 1997.

[30] Y. Yokote. The Apertos Reflective Operating System : The Concept and Its Implementation. Sony Computer Science Laboratoty Inc, Technical report number SCSL-TR-92-014, June 1992.

[31] L.C. DiPippo R Ginis M. Squadrito S. Wohlever V. F. Wolfe I. Zykh. Expressing and Enforcing Timing Constraints in a Real-Time CORBA System. University of Rhode Island, Technical Report number TR97-252, February 1997.