ECOLE NATIONALE SUPERIEURE DES TELECOMMUNICATIONS

PARIS

MEMOIRE

présenté en vue d'obtenir

le grade de docteur de l'Ecole Nationale Supérieure des Télécommunications

Spécialité informatique et réseaux

par

Frank SINGHOFF

Spécification temporelle modulaire et support pour les applications multimédias réparties

Soutenue le 14 décembre 1999

JURY

PRESIDENT :	Mr Daniel HAGIMONT de l'INRIA/Rhône-Alpes
RAPPORTEURS :	Mr Ken CHEN de l'université Paris 13
	Mr Gérard FLORIN du CNAM/CEDRIC
EXAMINATEURS :	Mme Isabelle DEMEURE de l'ENST Paris (directeur de thèse)
	Mr Eric GRESSIER-SOUDAN du CNAM/CEDRIC
	Mr Robert RANNOU de l'ENST Bretagne
	Mr Jean-Bernard STEFANI du CNET

« Pourquoi faire simple quand on peut faire compliqué. »

« Plus ça rate, plus on a de chances que ça marche. »

Le professeur Shadoko,

Remerciements

Je tiens tout d'abord à remercier ma voisine de bureau qui joua à temps plein le rôle de directeur de thèse : Mme Isabelle Demeure. Cette thèse a largement bénéficié de ses idées et suggestions. Je tiens tout particulièrement à la remercier pour la disponibilité dont elle a fait preuve mais aussi et surtout pour son accueil chaleureux, son amitié et ses conseils toujours pertinents.

Je remercie aussi les deux rapporteurs de ce mémoire, Mr Ken Chen de l'université Paris 13 et Mr Gérard Florin du CEDRIC/CNAM, ainsi que Mr Daniel Hagimont de l'INRIA Rhône-Alpes qui a bien voulu présider la soutenance.

De même, pour le temps qu'ils ont bien voulu consacrer à cette thèse, j'adresse mes remerciements aux membres du jury : Mr Eric Gressier-Soudan du CEDRIC/CNAM, Mr Robert Rannou de l'ENST-Bretagne et Mr Jean Bernard Stefani du CNET.

Leurs propositions et corrections ont permis d'améliorer significativement la qualité de ce mémoire.

Cette thèse a été financée pour une bonne partie grâce à un contrat de recherche liant l'équipe DTL/ASR d'Issy Les Moulineaux du CNET et le groupe ASTRE de l'ENST Paris. Je tiens donc à remercier le CNET pour sa participation financière mais aussi et surtout l'équipe d'Issy Les Moulineaux, et en particulier Mr François Horn, Mr Laurent Leboucher et Mr Nicolas Rivierre. Par les nombreuses réunions que nous avons eues, ils ont largement participé aux contributions de cette thèse. Pour leur soutien, leurs conseils et leurs réflexions, qu'ils en soient remerciés.

De la même façon, je tiens à remercier tous ceux qui ont contribué à ce mémoire par la relecture des versions préliminaires. Merci donc à Mme Basma Driss, Mme Jeannette Singhoff, Mme Magali Singhoff, Mr Xavier Blondel, Mr Philippe Callot, Mr Bertrand Dupouy et Mr Thierry Singhoff. Merci aussi aux personnes qui, aux travers de leurs stages, mémoires ou conseils, ont participé au Projet POLKA ; je pense essentiellement à Mme Basma Driss, Mr Philippe Callot et à Mr Franck Girodengo.

Pour terminer, j'exprime ma reconnaissance à tous ceux qui m'ont amené et permis de réaliser cette thèse. Je pense notamment à mes collègues du groupe ASTRE de l'ENST Paris qui ont pu m'aider ponctuellement. Je pense aussi à ceux qui m'ont aiguillé vers la recherche et l'enseignement ; c'est le cas de Mr Eric Gressier-Soudan et, plus généralement de l'équipe du CEDRIC au CNAM. Qu'ils en soient collectivement remerciés.

Résumé

Par le passé, les applications informatiques manipulaient essentiellement des données textuelles ou graphiques non animées. Aujourd'hui ces données, que l'on appelle parfois données "discrètes", sont mélangées à des données telles que le son ou les images animées. On parle alors d'applications multimédias (ou littéralement, applications mélangeant plusieurs média différents). Dans ces nouveaux systèmes, les média tels que le son ou l'image animé sont très souvent désignés sous le terme de flots de données "continues". En effet, contrairement aux données discrètes dont les éléments de données sont temporellement indépendants, les éléments de flots de données continues sont liés par des contraintes temporelles. On parle alors de contraintes de qualité de service temporelles (exemples : délai entre l'affichage de deux images successives, synchronisation voix-lèvres, synchronisation de l'affichage d'objets animés, etc). Ainsi, un flot de données audio est constitué d'une suite d'échantillons qui doivent être délivrés au périphérique audio régulièrement. Il existe donc une contrainte temporelle entre un échantillon donné, son prédécesseur et son successeur. Si ce type de contrainte n'est pas respecté durant la présentation du flot audio, il risque de ne pas être suffisamment intelligible et ainsi de perdre une part importante de son utilité.

Cette thèse traite du support des applications multimédias réparties qui mettent en œuvre des flots de données continues tels que l'audio et la vidéo. Nous nous intéressons tout particulièrement à leur support dans des environnements banalisés non spécifiquement conçus pour de telles applications (exemple : micro-ordinateur PC muni d'un système d'exploitation Windows ou Linux). Par ailleurs, cette thèse cible les applications multimédias dont le comportement temporel est partiellement prédictible ou encore celles dont les besoins en ressources sont difficiles à estimer.

Dans un premier temps, nous proposons l'utilisation d'un modèle orienté flots de données dans lequel nous spécifions les flots de données multimédias ainsi que les éléments importants du système. Ce modèle permet la spécification des contraintes temporelles associées aux flots de données et indirectement de modéliser des contraintes de débit. Les contraintes visées sont des contraintes déterministes.

Par la suite, nous décrivons des algorithmes qui exploitent ce modèle. Ces algorithmes fournissent principalement des services pour ordonnancer les tâches du système conformément aux contraintes temporelles spécifiées grâce au modèle. Ils offrent l'opportunité de réaliser automatiquement la traduction des contraintes de QoS en directives d'ordonnancement.

Le modèle ainsi que les algorithmes sont validés par une plate-forme (la plate-forme POLKA) et plusieurs applications multimédias. Nous montrons les impacts de ces propositions sur la gestion des ressources ainsi que les capacités de notre plate-forme à s'adapter à un changement des contraintes temporelles.

Enfin, nous terminons par une ouverture sur l'intégration des contraintes probabilistes.

Mots clefs

Applications multimédias réparties, modélisation, ordonnancement automatique, objets, contraintes temporelles

Abstract

In the past, applications used to be built with textual data and fixed pictures. Today, these data (usually called discrete data) are mixed with animated pictures and sound samples. These applications are called "multimedia applications" since they consist of several media. In these new systems, sound and picture flows are called "continuous data". Indeed, contrary to discrete data whose elements are independents, continuous data are composed of successive elements constrained by temporal relationships. These constraints express quality of service requirements such as delay between displaying two pictures, lips-synchronization, animated object synchronizations, etc. For instance, an audio flow is built with a sequence of audio samples which must be provided periodically to a device. So, each element of the flow has temporal constraints with its predecessor and its successor. If these temporal constraints are not futfilled, the audio flow could loose most of its utility.

This thesis deals with distributed multimedia applications composed of continuous flows. We focus on their support over general purpose systems (eg. a personal computer running an operating system like Windows or Linux). In addition, this thesis targets multimedia applications whose temporal behavior is partly unpredictable or whose requirements are difficult to estimate a priori.

At first, we define a data flow graph model to specify flows of a multimedia system. This model allows the specification of the temporal constraints associated to the application flows. It can describe throughput constraints as well. Constraints used in this model are mainly deterministic.

In the sequel, we propose several algorithms which make use of these specifications. Our algorithms automatically translate QoS temporal constraints specified in the model into scheduling information and allocate processors to the tasks of the system according to their temporal constraints.

The model and the algorithms are implemented in a middleware called Polka. They are tested with several multimedia applications. We provide measurements on these applications. They show that our propositions effectively schedule applications according to their temporal constraints, provided enough resources are available.

To end, we give some ideas about extension of our work in the case of probabilistic temporal constraints.

Key words

Distributed multimedia applications, modeling, real time scheduling, objects, temporal constraints.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction et contributions de cette thèse

Par le passé, les applications informatiques manipulaient essentiellement des données textuelles ou graphiques non animées. Aujourd'hui ces données, que l'on appelle parfois données "discrètes", sont mélangées à des données telles que le son ou les images animées. On parle alors d'applications multimédias (ou littéralement, applications mélangeant plusieurs média différents). Dans ces nouveaux systèmes, les média tels que le son ou l'image animée sont très souvent désignés sous le terme de flots de données "continues" [STE 95b]. En effet, contrairement aux données discrètes dont les éléments sont temporellement indépendants, les éléments de flots de données continues sont liés par des contraintes temporelles. Ainsi, un flot de données audio est constitué d'une suite d'échantillons qui doivent être délivrés régulièrement au périphérique audio. Il existe donc une contrainte temporelle entre un échantillon donné, son prédécesseur et son successeur. Si ce type de contrainte n'est pas respecté durant la présentation du flot audio, il risque de ne pas être suffisamment intelligible et ainsi de perdre une part importante de son utilité. On peut donc définir un flot de données continues comme un flot composé d'éléments de données qui doivent être présentés en respectant des contraintes de **qualité de service** (QoS, *Quality of Service*) temporelles (exemples : délai entre l'affichage de deux images successives, synchronisation voix-lèvres, synchronisation de l'affichage d'objets animés, etc).

Cette thèse traite du support des applications multimédias réparties qui mettent en œuvre des **flots de données continues** tels que l'audio et la vidéo. En effet, ces dix dernières années les applications multimédias se sont largement répandues au travers des jeux, de nouveaux outils d'information (exemples : bornes multimédias, médiathèques), ou de communication (exemple : vidéo-conférence), etc. Si de nombreuses applications bénéficient d'un environnement dédié (c'est le cas des applications embarquées dans des bornes multimédias ou des consoles de jeux), la tendance actuelle consiste à offrir comme support d'exécution un environnement logiciel et matériel dit "grand public" tel qu'un micro-ordinateur doté d'un système d'exploitation Windows ou Linux. Ces environnements sont capables d'exécuter des applications ayant des besoins variés et n'offrent pas de support

spécifique aux applications constituées de flots de données continues. Dans cette thèse, nous nous focalisons plus particulièrement sur les services qui doivent leur être ajoutés.

Pour prendre en compte les contraintes temporelles des flots de données continues dans de tels environnements, les solutions développées à ce jour utilisent, en général, les propriétés temps réel des ordonnanceurs du système sous-jacent et le support de contraintes de QoS temporelle qu'offrent les nouvelles générations de protocoles de communication (exemples : ATM [VET 95] (*Asynchronous Transfer Mode*), RTP/RTCP [SCH 96] (*Real Time Protocol, Real Time Control Protocol*)). Elles s'appuient également sur des composants spécifiques tels que les cartes audio et les décodeurs MPEG [ISO 95] (*Motion Picture Expert Group*) .

Ces solutions présentent plusieurs inconvénients : tout d'abord, les abstractions offertes par ces systèmes ne sont pas forcément adaptées à celles qui sont utilisées pour décrire les applications multimédias. Prenons l'exemple d'ordonnanceurs traditionnels tels que ceux du système UNIX SVR4. Ceux-ci n'offrent pas d'abstraction de flots de données continues car ils n'ont pas été conçus pour ce type d'applications. Le développeur d'applications multimédias devra donc manuellement traduire ses flots et leurs contraintes temporelles en tâches et priorités, puis, choisir dans quelles classes d'ordonnancement ses tâches devront s'exécuter. Ce travail est difficile à réaliser compte tenu des caractéristiques des ordonnanceurs SVR4 [NIE 93]. De plus, une petite modification dans le comportement temporel de l'application peut nécessiter la réestimation complète du choix des priorités et des classes d'exécution.

Ensuite, on ne peut pas se contenter de considérer chaque composant du système isolément : dans certains cas, il faut prendre en compte le comportement du système dans sa globalité, ou de bout-en-bout [CAM 98]. Pour cette raison, il est courant dans les plates-formes dédiées aux applications multimédias que la mémoire, le processeur et les ressources réseaux soient gérés de façon intégrée [TOK 92, SIJ 96]. Or, dans les systèmes ciblés par cette thèse, la prise en compte des contraintes temporelles est offerte de façon "ad hoc" : chaque composant offre (ou n'offre pas) des services de gestion de QoS avec ses caractéristiques propres (expression de la QoS, type de garantie offerte, etc). L'absence d'un modèle global du système rend difficile la construction d'applications multimédias complexes.

De plus, le développement de telles applications demande une connaissance très approfondie des systèmes d'exploitation sous-jacents, des protocoles de communication et des composants spécifiques utilisés (exemples : carte audio, décodeur MPEG). Il est donc important de pouvoir disposer de composants réutilisables permettant de cacher aux concepteurs la complexité de ces éléments.

Par ailleurs, cette thèse s'intéresse plus particulièrement aux applications multimédias dont le comportement temporel est partiellement prédictible ou à celles dont les besoins en ressources sont délicates à estimer :

- D'abord par la nature de leur environnement d'exécution qui est complexe. Par exemple, il est difficile d'obtenir une évaluation fine du temps d'exécution des tâches

dans les systèmes que nous ciblons, à la fois à cause du matériel utilisé (présence de cache mémoire, de mécanismes de DMA (*Direct Memory Access*), de processeurs utilisant de la prédiction de branchement pour des pipelines, etc) mais aussi à cause du système d'exploitation (occurrences d'interruptions, utilisation d'une mémoire virtuelle, présence de cache disque, etc) [HUA 97].

- Ensuite par la nature des données que les applications multimédias manipulent. Le format de compression MPEG 2 illustre bien ce cas de figure. En effet, il génère un débit variable qui est fonction du contenu du film ; il est alors difficile d'évaluer en ligne, pour un film donné, les besoins en ressources qui vont être requis [BAI 96].

- Enfin, du fait de l'utilisateur qui peut modifier ses besoins en ressources en lançant de nouvelles applications, ou qui peut changer la QoS d'une application durant son exécution. Une application utilisée par plusieurs personnes peut donc requérir une QoS différente pour chaque utilisateur (exemple : le besoin de synchronisation voix-lèvres est plus ou moins fort selon la sensibilité de l'utilisateur [STE 95a]).

Pour toutes ces raisons, dans de telles applications une réservation de ressources induirait une sur-réservation inacceptable [BAI 96] et il est nécessaire d'offrir des mécanismes souples, capables de s'adapter au comportement du système et des applications.

Or, dans les environnements ciblés, si la prise en compte de contraintes statiques est envisageable, celle de contraintes dynamiques, c'est à dire évoluant au gré de l'exécution de l'application, est souvent complexe, voire impossible. Par exemple, l'ajout dynamique d'une tâche dans un système utilisant un ordonnanceur SVR4 peut conduire à un comportement imprévisible du système [NIE 93]. Idéalement, on devrait pouvoir modifier les contraintes de QoS indépendamment du code de l'application. Cette fonctionnalité est nécessaire si l'on souhaite supporter des applications dynamiques, mais elle est aussi souhaitable pour atteindre un niveau de portabilité élevé de l'application. Malheureusement, il est courant dans les systèmes qui nous intéressent, que la prise en compte de la QoS soit réalisée au travers d'interfaces de programmation. Les contraintes temporelles sont alors incluses dans le code de l'application.

Les difficultés sont donc nombreuses si l'on souhaite construire les applications multimédias que nous ciblons dans des systèmes "grand public" et ce en offrant un certain niveau de portabilité, de flexibilité, d'adaptivité et de facilité de réalisation et de maintenance.

Contributions de cette thèse

Les contributions de cette thèse sont de trois ordres : sur la modélisation d'applications multimédias d'abord, sur le plan algorithmique et architectural ensuite, et enfin sur le support des contraintes probabilistes.

La première contribution porte sur le plan de la modélisation d'applications multimédias réparties

Nous proposons l'utilisation d'un modèle orienté flots de données dans lequel nous spécifions les flots de données multimédias ainsi que les éléments importants du système. Ce modèle permet la spécification des contraintes temporelles associées aux flots de données et indirectement de modéliser des contraintes de débit. Il permet aussi de modéliser le comportement temporel de l'environnement d'exécution. Les contraintes temporelles ainsi exprimées constituent des contrats engageant les différents éléments du système. Le modèle autorise une spécification modulaire du système où chaque élément peut être défini indépendamment, puis, être combiné avec d'autres éléments pour finalement constituer le système dans son intégralité. Il autorise donc un certain niveau de réutilisabilité des spécifications temporelles. L'utilisation d'un tel modèle tend à séparer, lors de la construction d'une application, les aspects fonctionnels de l'application (services offerts par l'application) de la Qualité de Service obtenue lors de la réalisation de ces services (exemple : contraintes temporelles). Ceci améliore aussi la portabilité de l'application. Enfin, un tel modèle offre au concepteur d'une application l'opportunité de se concentrer sur les caractéristiques temporelles de l'application et du système support, en laissant de côté les difficultés d'implantation liées à la complexité et à la diversité des composants matériels et logiciels utilisés dans une application multimédia.

La deuxième contribution porte sur des aspects algorithmiques et architecturaux

Nous proposons des mécanismes de gestion de ressources qui exploitent le modèle de spécification proposé ci-dessus. Ces mécanismes se présentent sous la forme d'algorithmes qui, à partir du modèle, génèrent de façon automatique les directives de gestion de ressources.

Dans cette thèse, nous regardons principalement la ressource processeur. Les algorithmes fournissent donc des services pour ordonnancer les tâches du système conformément aux contraintes temporelles spécifiées dans le modèle. Les algorithmes d'ordonnancement offrent l'opportunité de réaliser automatiquement la traduction des contraintes de QoS en directives d'ordonnancement (évitant ainsi au développeur d'effectuer cette traduction manuellement, comme c'est le cas si un ordonnanceur UNIX SVR4 est utilisé par exemple). Ils sont basés sur l'idée proposée par Jocelyne Farhat-Gissler dans sa thèse [FAR 96] et qui consiste à transformer une contrainte temporelle en échéance et en date d'exécution au plus tôt.

Ces algorithmes ne demandent pas à l'utilisateur de fournir le temps d'exécution des tâches du système, ce qui est important compte tenu des applications que nous ciblons et des environnements d'exécution choisis.

Nous proposons d'abord un algorithme capable d'ordonnancer les tâches d'une application multimédia centralisée. Puis, nous montrons comment, en exploitant des résultats connus et notre modèle, il est possible d'offrir un support pour des applications réparties.

Par la suite, nous proposons une architecture orientée objets offrant les services nécessaires à la spécification et à l'ordonnancement d'une application multimédia.

Ces propositions algorithmiques et architecturales sont validées par une plate-forme (la plate-forme POLKA) et plusieurs applications multimédias. Nous montrons les impacts de ces propositions sur la gestion des ressources ainsi que les capacités de notre plate-forme à s'adapter à un changement des contraintes temporelles. Que ce soit un changement des contraintes spécifiées par l'utilisateur ou un changement des caractéristiques temporelles des services du système (exemple : modification des caractéristiques temporelles d'un élément réseau).

La troisième et dernière contribution concerne le support des contraintes probabilistes

Dans cette dernière partie, nous menons une étude exploratoire sur les solutions techniques à mettre en œuvre pour offrir un support des contraintes probabilistes dans POLKA. Le support de ce type de contrainte est motivé par deux raisons principales. La première est qu'il existe dans les applications multimédias réparties des contraintes qui sont par nature probabilistes. Il est alors important d'offrir à l'utilisateur la possibilité d'exprimer de telles contraintes de QoS. La seconde est que le comportement de certains composants des systèmes que nous ciblons n'est pas déterministe, mais peut parfois être caractérisé par des contraintes probabilistes.

La prise en compte de ces contraintes pose des problèmes complexes non résolus à ce jour, mais qui font l'objet d'un effort important de la part de la communauté scientifique. Notre contribution consiste à montrer comment notre modèle de spécification peut être aménagé pour exprimer certaines formes de contraintes probabilistes qui nous semblent suffisantes dans un premier temps. Enfin, nous montrons au travers d'une application les gains obtenus en termes de consommation de ressources vis-à-vis d'une spécification déterministe.

Plan de ce mémoire de thèse

Le plan de ce mémoire est le suivant. Nous commençons dans le chapitre 2 à dresser un état de l'art des mécanismes de gestion de ressources disponibles aujourd'hui pour les applications multimédias. Au travers de ce chapitre, nous identifions les classes d'applications ciblées par nos propositions et nous montrons en quoi les solutions proposées aujourd'hui répondent ou non aux problèmes que nous soulevons dans cette thèse.

Dans le chapitre 3, nous donnons un bref aperçu de nos propositions que nous détaillons dans les chapitres suivants. Ainsi, nous décrivons dans le chapitre 4 notre modèle de spécification. Nous illustrons le modèle dans le chapitre 5 par plusieurs exemples couramment rencontrés dans les applications multimédias réparties. Le chapitre 6 est dédié aux aspects algorithmiques de cette thèse ; nous y détaillons les algorithmes d'ordonnancement ainsi

que leurs propriétés. Dans les chapitres 7 et 8, une large place est consacrée à la validation des concepts et algorithmes proposés dans les chapitres 4 et 6. Le chapitre 7 décrit le prototype réalisé. Le chapitre 8 présente plusieurs applications que nous avons exécutées sur notre plate-forme. Nous y montrons, en particulier, l'impact de nos propositions sur la gestion des ressources (principalement processeur et mémoire). Le chapitre 9 relate notre travail d'exploration concernant le support des contraintes probabilistes dans la plate-forme POLKA. Enfin, nous concluons cette thèse au chapitre 10. Nous en profitons pour évaluer les perspectives de recherche de nos travaux.

En annexe, le lecteur intéressé trouvera des compléments d'information. L'annexe A regroupe les démonstrations des résultats présentés dans les différents chapitres de cette thèse. L'annexe B contient les descriptions Esterel des composants décrits dans les chapitres 4 et 5. L'annexe C présente les descriptions de QoS utilisées pour les applications multimédias décrites dans le chapitre 8. Enfin, le lecteur souhaitant obtenir une vision plus synthétique des résultats décrits dans ce mémoire peut consulter l'annexe D qui énumère les publications associées à cette thèse.

Chapitre 2

Etat de l'art et motivations

Comme nous l'avons vu au chapitre 1, les applications multimédias manipulent à la fois des données discrètes et des données continues. Les éléments d'un flot de données continues sont soumis à des contraintes temporelles, qui, si elles ne sont pas respectées durant la présentation des données, impliquent la perte d'une part importante de l'utilité du flot de données.

La prise en compte des contraintes temporelles intrinsèques aux médias est donc importante dans ces applications. Mais quelles sont les différentes alternatives disponibles à ce jour? Il existe en fait une très large gamme d'applications multimédias, ayant des besoins en ressources et des contraintes variées. Le support de leurs contraintes temporelles dépend, bien sûr, de ces caractéristiques. Ceci explique l'existence de solutions diamétralement opposées, allant de l'utilisation de systèmes temps réel fortement contraints, à l'exploitation de systèmes temps partagé tel qu'UNIX. Nous nous proposons dans ce chapitre d'illustrer cette diversité des mécanismes de gestion de ressources.

Ainsi, dans la partie 2.1, nous décrivons deux exemples d'applications qui possèdent des contraintes et des besoins très différents. A travers ceux-ci, nous définissons les mécanismes abordés dans la suite du chapitre. Les parties 2.2, 2.3, et 2.4 sont dédiées à la présentation de mécanismes concernant respectivement la gestion du processeur, des ressources réseaux et mémoires. Enfin, nous concluons dans la partie 2.5 en revenant sur les caractéristiques des applications que nous ciblons ainsi que sur les motivations de cette thèse.

2.1 Exemples d'applications multimédias

Dans cette partie, nous donnons quelques exemples de contraintes de QoS couramment rencontrées dans les applications multimédias. Nous définissons les mécanismes de gestion de ressources employés dans les systèmes multimédias qui nous intéresseront par la suite. Puis, à travers deux exemples d'applications multimédias, nous montrons la diversité des besoins présentés par ces applications.

2.1.1 Principales contraintes de qualité de service

Il existe une multitude de contraintes de qualité de service, et les formalismes utilisés pour les spécifier sont tout aussi nombreux. Dans cette thèse, nous ne chercherons pas à énumérer exhaustivement celles qui existent. On peut toutefois séparer les contraintes de QoS en trois groupes : les contraintes de QoS spatiales, temporelles et de fiabilité [TOK 92].

- Les contraintes spatiales portent sur des aspects quantitatifs. L'utilisateur parlera de taille d'image, de nombre de couleurs, etc. Ces contraintes seront traduites en débit nécessaire, en taille de tampons ou en volume de données.

- Les contraintes temporelles expriment des relations d'ordre entre plusieurs événements. Selon que la contrainte utilise une horloge ou non, une contrainte peut être qualifiée d'absolue ou de relative [VEG 97]. Une contrainte absolue date un événement avec la valeur d'une horloge alors qu'une contrainte relative ordonne dans le temps deux événements.

  Le rythme d'affichage des images d'un film est une contrainte temporelle. Il en est de même pour la synchronisation d'un flot audio et d'un flot vidéo (on parle de synchronisation voix-lèvres). Les contraintes temporelles sur les flots de données continues sont généralement classées en deux grandes familles : les synchronisations "intra-flots" qui spécifient les contraintes temporelles qui existent entre les différents éléments d'un flot multimédia (ce peut être un délai maximal et un délai minimal entre deux images) et les synchronisations "inter-flots" qui décrivent les contraintes temporelles entre plusieurs flots (exemple : synchronisation voix-lèvres). Mais les contraintes temporelles ne portent pas uniquement sur les flots de données continues. Elles peuvent aussi concerner des événements quelconques du système (exemple : délai maximal entre le début d'un film et la mise à jour d'informations de contrôle sur un écran). On parle alors de contraintes temporelles événementielles [HAF 98].

  Dans cette thèse, nous nous focaliserons plus particulièrement sur les contraintes de qualité de service temporelles.

- Les contraintes de fiabilité, enfin, spécifient la marge d'erreurs acceptable pour le service demandé. Une contrainte de fiabilité peut exprimer un nombre d'images non délivrées, qui sera traduit en un nombre autorisé de paquets perdus ou en un taux d'erreurs sur bits. Elles peuvent aussi spécifier les pannes temporelles acceptables. Par exemple, un utilisateur peut tolérer une dérive de la synchronisation voix-lèvres sur un flot multimédia donné, mais pas sur d'autres.

Les contraintes de QoS peuvent être rencontrées à plusieurs niveaux d'un système multimédia. Le formalisme utilisé dépend alors de l'entité qui la spécifie. Une contrainte de QoS peut être spécifiée par l'utilisateur ou imposée par un élément du système.

Dans les deux cas, la contrainte spécifiée devra être éventuellement transmise à plusieurs éléments du système, et ce dans des formalismes parfois différents. Par exemple, un rythme d'affichage des images sera traduit en une contrainte de débit sur le composant réseau, en

échéance sur les tâches qui devront délivrer les images, en taille de mémoire, etc. C'est généralement l'infrastructure qui offre les moyens de convertir des contraintes de QoS de haut niveau en contraintes de QoS interprétables par les mécanismes de gestion de ressources (conversion manuelle ou automatique).

Dans ce chapitre, nous nous concentrons sur les mécanismes de gestion des ressources, en d'autres termes, sur les mécanismes qui permettent le partage des ressources du système en fonction d'une qualité de service donnée. Les formalismes utilisés pour l'expression de la QoS et les mécanismes de traduction de la QoS ne seront pas développés de façon détaillée dans ce chapitre. Nous renvoyons le lecteur intéressé vers d'autres publications pour ces aspects [VOG 95, CAM 98].

2.1.2 Services offerts par un système multimédia

Définir les contraintes d'un système n'est pas tout : il faut ensuite pouvoir les respecter grâce à des mécanismes de gestion de ressources. Que ces mécanismes soient fournis par le système d'exploitation ou les applications, il est possible d'en distinguer trois catégories : les mécanismes de réservation de ressources, les mécanismes d'allocation de ressources et les mécanismes de supervision.

Les mécanismes de réservation de ressources interviennent, par exemple, lors de l'arrivée d'une nouvelle application dans le système. Ils vérifient si toutes les ressources nécessaires à l'application sont disponibles (cette phase est souvent appelée contrôle d'admission), puis bloquent les ressources nécessaires à l'application. La réservation garantit à l'application qu'elle disposera des ressources durant toute son exécution. La phase de contrôle d'admission est parfois précédée d'une négociation entre l'application et le système. Dans la négociation les deux intervenants s'accordent sur une qualité de service commune en fonction des ressources disponibles, des besoins exprimés par l'application et de critères de coûts des ressources

Les mécanismes d'allocation décident à chaque instant à quelle application doit être allouée une ressource donnée.

Enfin, les mécanismes de supervision collectent des informations sur l'utilisation des ressources et sur le fonctionnement des applications. Les informations collectées peuvent alors être exploitées par les mécanismes de réservation ou d'allocation. Par exemple, ces informations peuvent être utilisées pour renégocier la qualité de service, limiter ou modifier la consommation en ressources de certaines applications, ou tout simplement renseigner les applications sur les ressources disponibles. On parle alors de mécanismes de rétroaction.

Il existe beaucoup de variantes des mécanismes présentés ci-dessus. Elles reflètent la grande variété des applications multimédias. Le lecteur intéressé par une classification des applications multimédias selon leurs fonctionnalités pourra consulter celle proposée par Hafid et al. [HAF 98].

Nous décrivons ici deux applications caractéristiques qui possèdent des propriétés diamétralement opposées. Elles illustrent les besoins en ressources que l'on peut rencontrer dans des systèmes ou des applications multimédias.

La première application est une application de vidéo-conférence. La seconde est un exemple de télévision interactive.

2.1.3 Une application de vidéo-conférence

Une application de vidéo-conférence a pour objet la transmission de flots audio et vidéo synchronisés d'un émetteur vers un récepteur [GEM 97]. On souhaite garantir de façon sûre l'acheminement et la restitution des données : aucune variation visible de la synchronisation voix-lèvres ne doit être observée par les deux utilisateurs. Nous supposerons que les flots d'informations sont compressés par la norme H.261 [CCI 90] et que les informations sont véhiculées par un réseau RNIS (*Réseau Numérique à Intégration de Services*). Le nombre de flots de données continues est fixé à deux : un flot vidéo et un flot audio. Enfin, hormis aux instants de démarrage et de terminaison de la vidéo-conférence, aucune interaction n'est faite avec les utilisateurs.

Il est relativement simple d'évaluer les besoins de cette application multimédia. Regardons, à titre d'exemple, le débit réseau nécessaire au bon fonctionnement de l'application. Tout d'abord, la norme de compression de données H.261 génère un débit constant multiple de 64 Kbits ; or, nous connaissons le nombre de flots qui vont être utilisés dans l'application. Comme le débit d'un film compressé est connu, il est facile de déterminer le débit réseau qui sera demandé par l'application (par multiples de 64 Kbits). Ensuite, l'application est par nature peu interactive ; les utilisateurs ne peuvent modifier les besoins en ressources ou la qualité de service de l'application. Le débit réseau de l'application est donc constant. Connaissant précisément le débit nécessaire, une application de ce type utiliserait très certainement un mécanisme de réservation de la bande passante. L'utilisation d'un réseau RNIS facilite d'autant plus la mise en œuvre de cette application que ce type de réseau permet la réservation de canaux de communication dont le débit est de 64 Kbits.

2.1.4 Les services de type "télévision interactive"

Cette deuxième application a pour objet la diffusion de films vers des postes informatiques banalisés. Contrairement à l'exemple précédent, la télévision interactive offre à l'utilisateur de très nombreuses possibilités d'agir sur l'application. Ainsi lors d'un match de football, un utilisateur peut consulter des informations sur les joueurs ou sur des résultats sportifs. Plus intéressant, il pourra sélectionner le commentaire sportif dans la langue de son choix: si le match se déroule en Allemagne, le téléspectateur français préférera, par exemple, écouter les commentaires en français émis de Paris, ce qui pose des problèmes non triviaux de synchronisation des flots audio et vidéo. De même, il est possible d'imaginer que l'utilisateur puisse sélectionner les images de la caméra de son choix, ou puisse regarder le match filmé à partir de plusieurs caméras (caméras qu'il pourra sélectionner au fur et à mesure du déroulement du match). En bref, contrairement à l'application de vidéo-conférence, l'utilisateur influe directement sur les besoins de l'application, en demandant la restitution d'un nombre variable de flots vidéo et audio.

De plus, il est parfois difficile d'évaluer précisément le débit nécessaire à chaque flot. C'est, par exemple, le cas des chaînes de télévision numérique "à péage" ; ces dernières offrent un service de vidéo à la demande et facturent leurs clients en fonction des films qu'ils ont visionnés. Ces réseaux utilisent le standard de compression MPEG-2 [ISO 94] pour la diffusion de leur programme [RIC 98]. Or, ce standard peut générer des flots vidéo à débits variables (le débit d'un film compressé en MPEG-2 dépend du codeur utilisé et du film lui-même). Il est alors difficile de déterminer a priori de façon précise les besoins en ressources d'un film. Dans ces conditions, une réservation au pire cas conduirait inévitablement à une sur-réservation des ressources, et donc à la perte d'une grande partie des gains obtenus lors de la compression du film.

On constate que contrairement à l'exemple de la vidéo-conférence, la télévision interactive est beaucoup plus dynamique et que ses besoins en ressources sont difficiles à évaluer a priori. Enfin, il est clair que les besoins en garantie de service sont différents de ceux demandés par l'application de vidéo-conférence : l'utilisateur d'un service de télévision interactive peut éventuellement se satisfaire d'une dégradation temporaire de la qualité de service ; si le système de vidéo-conférence est utilisé pour le suivi d'une opération chirurgicale, une dégradation temporaire de la qualité de service rendue par le système de vidéo-conférence est à exclure.

2.1.5 Des classes d'applications multimédias vers les mécanismes de gestion de ressources

A la lumière des exemples précédents, il est clair qu'il serait inadapté d'appliquer les mêmes mécanismes de gestion de ressources à toutes les applications multimédias. Alors que dans le cas de l'application de vidéo-conférence, une grande partie des informations concernant ses besoins en ressources peuvent être obtenues hors-ligne (c'est-à-dire avant son exécution), les ressources nécessaires à l'application de télévision interactive sont pour une grande part impossibles à déterminer précisément hors-ligne, ce qui peut interdire l'utilisation de mécanismes de réservation [BAI 96].

Ainsi, il est primordial lorsque l'on construit, évalue ou compare des mécanismes de gestion de ressources, de ne pas oublier de prendre en considération la classe d'applications pour laquelle le mécanisme étudié est conçu. Dans la suite de ce chapitre, nous allons décrire des exemples de mécanismes que l'on rencontre régulièrement dans la littérature, tout en montrant en quoi ces derniers sont plus ou moins bien adaptés à des applications multimédias dynamiques ou non.

2.2 L'allocation du processeur

Nous entrons maintenant dans le vif du sujet avec la description des solutions disponibles aujourd'hui pour l'allocation et la réservation des ressources processeurs. Nous nous focalisons uniquement sur les systèmes mono-processeurs (cf. partie 6.4 pour les systèmes répartis). Dans un premier temps, nous détaillons les algorithmes et les modèles de tâches utilisés dans les applications temps réel qui ont été appliqués dans les systèmes multimédias.

Puis, nous examinons quelques modèles de tâches plus spécifiques aux applications multimédias. Enfin, nous terminons en regardant comment les algorithmes d'ordonnancement ont été aménagés pour prendre en compte les aspects adaptatifs de certaines applications multimédias.

2.2.1 Les algorithmes classiques d'ordonnancement temps réel

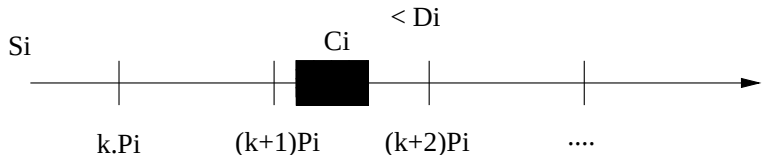


FIG. 2.1 – *Le modèle de tâche périodique de Lui et Layland*

Pour spécifier les contraintes temporelles et ordonnancer une application multimédia, on peut utiliser des techniques qui s'adressent aux applications temps réel en général. La littérature sur l'ordonnancement temps réel distingue les tâches répétitives des tâches non répétitives [LEB 98]. Les tâches répétitives sont celles qui font l'objet de plusieurs activations successives. Le modèle de tâches répétitives le plus répandu est celui de la tâche périodique proposé par Liu et Layland [LIU 73] (cf. figure 2.1). Dans ce contexte, une tâche périodique i est définie par le 4-uplet (S_i, D_i, C_i, P_i) où C_i constitue le temps d'exécution d'une activation de la tâche i (C_i est généralement une borne sur le temps d'exécution), P_i sa période d'activation et D_i son échéance. Enfin S_i est la date d'arrivée de la tâche dans le système. L'activation k de la tâche i intervient à la date $S_i + P_i(k-1)$. L'échéance D_i est relative aux dates d'activation. Dans le modèle initialement proposé par Liu et Layland, l'échéance d'une tâche était égale à sa période. Une variante de la tâche périodique est la tâche sporadique [MOK 83]. Une tâche sporadique est définie par les mêmes paramètres qu'une tâche périodique si ce n'est que ses activations ne sont plus contraintes par une période mais par un délai minimal. Enfin, on parle de tâches apériodiques pour désigner des tâches non répétitives définies par le triplet (S_i, D_i, C_i) où S_i est la date d'arrivée de la tâche i dans le système, D_i son échéance et C_i son temps d'exécution.

Pour ordonnancer les tâches périodiques, deux algorithmes ont été proposés par Liu et Layland : RM (*Rate Monotonic*) qui associe une priorité aux tâches de façon statique et EDF (*Earliest Deadline First*) où les priorités sont attribuées dynamiquement.

RM est un algorithme hors ligne : une priorité est affectée à chaque tâche avant l'exécution de l'application. La valeur de cette priorité est inversement proportionelle à la périodicité de la tâche. Durant l'exécution, l'allocation du processeur consiste à donner le processeur à la tâche prête de plus forte priorité. La réservation de la ressource processeur s'effectue grâce à une analyse hors-ligne de taux d'occupation du processeur. En effet, il est possible de garantir le strict respect des échéances des tâches du système si le taux d'occupation du processeur respecte une certaine borne. Dans le cas d'un ordonnanceur préemptif, Liu et Layland ont proposé la borne $n(2^{\frac{1}{n}} - 1)$ où n est le nombre de tâches

dans le système[1]. Le taux d'occupation du processeur est égal à la somme du rapport entre le temps d'exécution et la périodicité de chaque tâche, autrement dit, si l'inéquation (2.1) est vraie, alors les contraintes temporelles des tâches seront respectées.

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \tag{2.1}$$

Dans le cas d'EDF, l'allocation du processeur est réalisée selon l'échéance des tâches : l'ordonnanceur alloue le processeur à la tâche dont l'échéance est la plus proche. Dans le cas d'un ordonnanceur préemptif, la réservation de la ressource processeur peut être effectuée comme pour Rate Monotonic grâce à un test d'admission proposé par Liu et Layland et qui est basé sur le taux d'occupation du processeur :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{2.2}$$

Les deux algorithmes présentés ci-dessus sont les plus couramment utilisés au sein de la communauté temps réel. Le modèle de tâche qu'ils utilisent est contraignant. Toutefois, de nombreuses extensions ont été proposées [CAR 94, STA 95, GEO 96, HER 97, LEB 98, RIV 98]. Les principales extensions, qui portent sur le modèle de tâches, consistent en des généralisations telles que la suppression de la contrainte $D_i = P_i$, le support des contraintes de précédence ou encore le partage de ressources. Pour ce qui est des algorithmes d'ordonnancement, leurs propriétés ont largement été étudiées (test d'ordonnançabilité, optimalité, efficacité, temps de réponse des tâches et complexité), et ceci dans les cas préemptif et non préemptif.

Les modèles ci-dessus peuvent être utilisés pour des applications multimédias dont le comportement est connu a priori (notre application de vidéo-conférence par exemple). A titre d'exemple, Buddhikot et al. utilisent le modèle de Liu et Layland au travers du concept de RTU (*Real Time Upcall*) [BUD 96]. Un RTU est une fonction dans l'espace d'adressage de l'application qui peut être exécutée périodiquement par l'ordonnanceur. Le concept de RTU est particulièrement adapté à la mise en œuvre de protocole de communication applicatif : en effet, il peut aider à limiter les multiples recopies de données (l'exécution du RTU se faisant dans l'espace applicatif). L'ordonnanceur utilise une technique proche de RM. Un contrôle d'admission et une réservation du processeur sont effectués pour chaque RTU nouvellement créé.

Néanmoins, pour un nombre non négligeable d'applications multimédias, il n'est pas possible d'utiliser les mécanismes d'allocation et de réservation décrits ci-dessus sans les aménager. D'abord parce que les modèles de tâches ne sont pas adaptés aux abstractions rencontrées dans ces applications ; ensuite parce qu'il existe des applications ayant un

[1] Dans certains cas, cette borne peut être améliorée [LEH 90].

comportement "dynamique". Par comportement dynamique, nous entendons les applications dont les besoins en ressources peuvent évoluer en cours d'exécution. L'application de télévision interactive constitue un bon exemple d'application où ces mécanismes ne sont pas applicables. Nous allons maintenant traiter ces deux points.

2.2.2 Modèles de tâches pour le multimédia

Les modèles de tâches pour les applications multimédias sont généralement inspirés du modèle de Liu et Layland. Nous allons en présenter deux : les contraintes de distance et les cadences d'activation.

Dans une application multimédia, on s'intéresse souvent à des contraintes autres qu'une échéance relative à une période. Il est courant, par exemple, de spécifier des contraintes de délais minimaux et/ou maximaux entre deux fins de tâche. En effet, un utilisateur peut vouloir spécifier des délais maximum et minimum entre l'affichage de deux images successives. On parle alors de contraintes de "distance" [BUC 93, STE 95b, SAK 95, HAN 96].

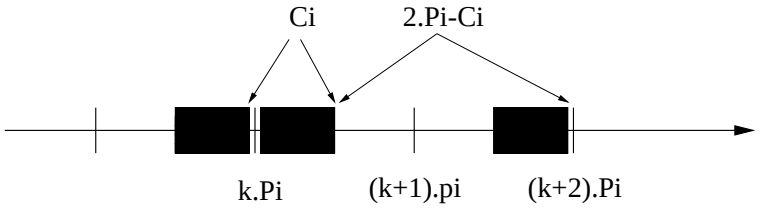


FIG. 2.2 – *Adéquation du modèle périodique au multimédia*

Pour définir et ordonnancer ce type de contrainte, plusieurs solutions ont été proposées. Si une tâche périodique est définie par les paramètres (S_i, D_i, C_i, P_i) tel que $D_i = P_i$, le modèle de tâche périodique spécifie uniquement que la terminaison de l'activation $k-1$ doit intervenir avant le début de la $k^{ème}$ période. Pour une tâche i, ceci a pour conséquence de soumettre les terminaisons de ses activations à la contrainte suivante [COT 98] :

$$C_i \leq i_{t+1} - i_t \leq 2 * P_i - Ci$$

où i_t est la date de terminaison de la $t^{ème}$ activation de la tâche i (cf. figure 2.2). Avec le modèle de tâche périodique, il est donc possible de spécifier implicitement une contrainte de distance en fixant P_i par :

$$P_i = \frac{\epsilon + C_i}{2}$$

où ϵ constitue un délai maximal entre deux fins d'activation successives. Cette première solution a pour avantage d'utiliser un modèle de tâches et des algorithmes d'ordonnancement bien connus. Toutefois, elle manque singulièrement de souplesse. Aussi, d'autres techniques ont elles été étudiées pour aménager le modèle périodique de Liu et Layland. Cottet et al. proposent une solution pour supprimer la gigue sur le délai entre deux fins d'activation successives d'une tâche [COT 98]. La solution consiste à modifier les dates

d'exécution au plus tôt ainsi que les échéances. G. Coulson et al. proposent de nouvelles conditions d'ordonnançabilité avec un ordonnanceur EDF sur des tâches qu'ils appellent "isochrones" ou "périodiques avec gigue" [COU 97]. Ces modèles de tâches sont des généralisations mieux adaptées aux applications multimédias que ne l'est le modèle de Liu et Layland. Les tâches isochrones sont définies par une échéance égale à leur temps d'exécution. De ce fait, la gigue sur le délai entre deux fins d'activation successives est nulle. Une tâche périodique avec gigue est définie par une échéance différente de la période.

Enfin, Han et al. proposent un modèle dont l'activation des tâches n'est plus déterminée par une période mais par la fin d'exécution de l'activation précédente [HAN 96]. Ce nouveau modèle de tâches n'est pas optimal s'il est ordonnancé par EDF. Han et Leboucher [LEB 98] suggèrent néanmoins un ordonnanceur basé sur EDF qui est optimal pour ce modèle : c'est l'algorithme de la dernière goutte ou LSD (*Last Single Drop*). L'algorithme consiste à retarder la fin d'exécution d'une activation jusqu'à son échéance. L'algorithme LSD est toutefois difficile à implanter. Une deuxième solution est présentée par Han et al. dans [HAN 96]. Elle utilise un algorithme d'ordonnancement différent et modifie le modèle de tâches en ajoutant un délai fixe entre la fin d'exécution d'une activation donnée et le début de l'activation suivante.

Nous terminons ce paragraphe par le modèle proposé par Jeffay et al. : les cadences d'activation [JEF 95]. Ici, les tâches d'une application multimédia sont contraintes par une échéance et sont définies par une loi d'arrivée plus générale que celle initialement décrite par Liu et Layland. Une tâche peut être activée x fois pendant une période de temps y. Aucune hypothèse particulière n'est faite sur la distribution des x activations sur la période de temps y. Le modèle est proche (bien que plus général) du modèle LBAP (*Linear Bounded Arrival Process*) utilisé par Anderson dans Dash [AND 93]. Les cadences d'activation ont été implantées dans YARTOS, un système d'exploitation pour les applications de vidéoconférence [JEF 92b].

2.2.3 Prise en compte des applications dynamiques

Pour offrir un support plus satisfaisant à des applications comme notre application de télévision interactive, de nombreuses solutions ont été proposées. Dans ce paragraphe, nous allons en donner quelques exemples. Nous commençons par les systèmes les moins adaptatifs pour finir par les plus adaptatifs.

Les premières solutions que nous pouvons citer sont celles qui ont tenté d'adapter les ordonnanceurs classiques RM ou EDF. Une des techniques les plus connues est celle du serveur sporadique introduite par Lehoczky et al. [SPR 89]. L'idée présentée ici est d'intégrer le support de tâches apériodiques dans une application constituée de tâches périodiques et ordonnancées par RM. Dans ces systèmes, l'exécution des tâches apériodiques est faite par une tâche périodique dédiée. Lehoczky et al. proposent plusieurs stratégies de gestion de cette tâche périodique. Chaque stratégie traite différemment le temps de réponse des tâches apériodiques, le taux d'occupation du processeur, etc.

La solution du serveur sporadique reste toutefois largement insuffisante dans le cadre d'une application telle que notre télévision interactive. Si elle permet, dans une certaine

limite, de prendre en compte l'arrivée de nouveaux traitements, elle est incapable de modifier les traitements périodiques déjà existants. Or, être capable de modifier les besoins en ressources des tâches pendant leur exécution est une fonctionnalité primordiale dans ce type d'application [2].

D'autres solutions plus satisfaisantes ont été proposées. Il existe principalement trois techniques pour adapter les besoins en ressources des tâches en cours d'exécution : renégocier la réservation de ressources, ne pas modifier la réservation mais adapter le comportement de l'application et enfin prévoir hors ligne les évolutions des besoins en ressources de l'application.

La première peut être illustrée par la solution de Mercer et al., où les tâches sont décrites par une période de réservation et un pourcentage de temps processeur [MER 94]. Ce pourcentage est aisément convertible en temps d'exécution d'une activation. Les tâches apériodiques sont dotées d'une période "artificielle" et les algorithmes utilisés sont soit EDF, soit RM. La solution est donc quasi-identique à celle utilisée dans un système temps réel classique. Mais cette fois-ci, aux mécanismes d'allocation et de réservation, les auteurs proposent d'ajouter un mécanisme de rétroaction. Les applications, en fonction des ressources disponibles peuvent alors renégocier leur réservation de ressources.

La technique de renégociation des ressources est particulièrement répandue. On la trouve dans YARTOS [JEF 92b]. Dans cette plate-forme, à chaque arrivée d'une tâche, un contrôle d'admission est réalisé. Le cas échéant, il est possible de renégocier les ressources allouées par une tâche (exemple : dans le cas où la source n'est plus conforme à la spécification utilisée lors de la réservation).

Toutefois, la renégociation n'est pas l'unique solution pour supporter des applications dynamiques. Une seconde possibilité consiste à demander à l'application d'adapter son comportement sans modifier la réservation effectuée préalablement. L'application effectue alors des traitements différents selon que les ressources sont disponibles ou non. Cette solution est celle choisie par Jones et al. [JON 97] ainsi que par Nieh et al. [NIE 97].

Jones propose que les contraintes de QoS soient directement spécifiées dans le code de l'application [JON 97]. Lors de son démarrage, l'application réserve des unités de temps sur une période donnée. Pendant son exécution, elle délimite les portions de code où doivent s'appliquer les contraintes de QoS. Les contraintes de QoS sont des contraintes d'échéance, de date de début d'exécution et de criticité. Elles ne sont pas spécifiées lors de la réservation du processeur mais au moment où la portion de code doit être exécutée. L'ordonnanceur prédit alors si la contrainte de QoS pourra être respectée (c'est-à-dire si la contrainte de QoS spécifiée reste compatible avec la réservation préalablement effectuée). Le développeur doit prévoir dans son code d'exécuter un code différent selon que la contrainte de QoS sera ou non respectée. L'ordonnanceur est basé sur EDF.

[2] Le serveur sporadique est par contre une solution intéressante pour notre application de vidéo-conférence dans le cas où l'on souhaite ajouter des traitements déclenchés par l'utilisateur n'ayant pas de contraintes temporelles fortes. Ainsi, les flots audio et vidéo peuvent faire l'objet de traitements par des tâches périodiques et la tâche servant les requêtes apériodiques peut être mise à contribution pour les traitements demandés par l'utilisateur (exemple : affichage à la demande d'informations sur les autres utilisateurs, sur l'application, etc).

SMART autorise également la spécification de contraintes de QoS (sous forme d'échéances) sur des portions de code [NIE 97]. L'application doit fournir une estimation du temps d'exécution pour chaque portion de code. Les informations temporelles sont également intégrées dans le code de l'application. Cette fois, l'ordonnanceur offre un mécanisme d'*upcall* permettant à l'application d'effectuer des traitements lorsqu'une contrainte temporelle ne peut être respectée. L'ordonnanceur de SMART est conçu pour faire coexister des tâches avec et sans contrainte temporelle. Pour ce faire, il utilise deux paramètres par tâche pour effectuer son allocation de ressources : une notion d'importance et une notion d'échéance. Schématiquement, l'élection d'une tâche par l'ordonnanceur est effectuée en deux étapes : d'abord en construisant l'ensemble des tâches de plus grande importance, puis en choisissant la plus urgente de cet ensemble. Les tâches de même importance partagent de façon équitable la ressource processeur. A cet effet, la notion d'importance est constituée d'une partie statique spécifiée par l'utilisateur et d'une partie dynamique, calculée par l'ordonnanceur.

Enfin, une troisième solution consiste à faire varier les besoins des applications dans un ensemble de valeurs déterminées hors ligne. Dans le projet Pegasus, Sijben et Mullender proposent d'utiliser cette technique en associant plusieurs niveaux de QoS pour chaque application [SIJ 96]. Chaque niveau comprend la liste des tâches et des connexions réseaux qui lui sont associées. Une tâche est définie par une périodicité, un temps d'exécution et un critère d'importance. L'ordonnanceur est capable d'exploiter des informations de dépendance entre les tâches. En effet, Pegasus cible tout particulièrement les applications multimédias dont les composants sont agencés en pipeline, architecture très courante dans ce type d'application. Concernant les connexions réseaux, les niveaux de QoS précisent le débit nécessaire et la périodicité entre chaque transmission de paquet (ici sur une couche de transport ATM/AAL5). Enfin, les niveaux de QoS précisent aussi la quantité de mémoire nécessaire. Une application fournit donc une liste de niveaux de QoS qu'elle est capable de supporter. La plate-forme Pegasus offre des services de réservation pour la mémoire, le processeur et le réseau. A tout moment, les applications sont donc assurées d'avoir les ressources qu'elles requièrent. Néanmoins, elles ne peuvent décider du niveau de QoS qui va leur être affecté. De plus, le niveau de QoS d'une application peut évoluer au cours de son exécution. En effet, lorsqu'une application entre dans le système, se termine ou modifie ses besoins, les traitements suivants sont déclenchés :

- Un gestionnaire de QoS choisit un niveau de QoS pour chaque application en maximisant le degré de satisfaction totale du système (grâce au critère d'importance).
- Puis, un ordonnanceur calcule l'ordre d'activation des tâches. Le résultat est stocké dans une liste. L'ordonnanceur est orienté échéance (EDF).
- Enfin, la liste est transmise au "répartiteur". Le répartiteur exploite la liste des tâches qu'il active séquentiellement. Durant les traitements effectués par l'ordonnanceur et le gestionnaire de QoS, le répartiteur exploite la liste élaborée ultérieurement.

Le mécanisme d'allocation du processeur est donc implanté dans deux entités séparées (l'ordonnanceur et le répartiteur).

Nous terminons ce paragraphe par la solution de Brandt et al., qui est plus radicale [BRA 98]. Brandt et al. n'offrent pas de mécanisme de réservation. Ils n'offrent pas non plus de mécanisme d'allocation du processeur. Comme pour Pegasus, plusieurs niveaux de QoS sont associés à chaque application. Chaque niveau de QoS comprend un pourcentage de processeur ainsi qu'un degré de satisfaction. Un gestionnaire de QoS affecte un niveau de QoS à chaque application en fonction des ressources disponibles et en maximisant un degré de satisfaction. La modification des besoins en ressources est effectuée par les applications qui s'adaptent au niveau de QoS dans lequel le gestionnaire les a placées. La gestion de la ressource processeur est donc réalisée au travers d'un mécanisme de supervision qui permet de détecter les surcharges et les "sous-charges".

2.3 La gestion des ressources réseaux

Nous regardons dans cette partie les mécanismes dédiés à la gestion des ressources réseaux. Les mécanismes de gestion de ces ressources peuvent être rencontrés à plusieurs niveaux dans les systèmes multimédias. On les trouve dans le système d'exploitation ; c'est principalement le cas des couches transport et réseau. Une autre partie du logiciel est embarquée dans les périphériques (exemple : la couche liaison parfois). Enfin, il arrive que les mécanismes soient conçus pour être directement intégrés dans les applications [CLA 90]. Cette tendance n'est toutefois pas spécifique aux systèmes multimédias. Selon les protocoles utilisés et les services offerts, les mécanismes de réservation bloquent des ressources dans les extrémités et/ou les nœuds du réseau. Ces ressources peuvent être des tampons mémoires, de la bande passante. Enfin, les contraintes de QoS généralement rencontrées sont des contraintes temporelles (exemples : délais, gigue, etc.) ou de fiabilité (exemple : taux de perte).

Dans une première partie, nous présentons des mécanismes bâtis sur des réseaux offrant des services synchrones ou isochrones. La deuxième partie traite des réseaux asynchrones [STE 95b].

Par services asynchrones, nous entendons des services de communication qui n'offrent pas de garantie temporelle sur l'acheminement des données. Ces services sont à opposer aux services synchrones qui garantissent un délai de communication borné et aux services isochrones qui assurent une gigue maximale sur les temps de communication. La notion de gigue maximale, que nous utiliserons régulièrement dans cette thèse, est définie par la différence entre le temps maximal et le temps minimal de communication [BOC 95, GAG 96].

Les services offerts par les réseaux isochrones et synchrones sont typiquement ceux que notre application de vidéo-conférence nécessite. Dans le cas de l'application de télévision interactive, les mécanismes préconisés sont plutôt ceux proposés pour les réseaux asynchrones.

2.3.1 Mécanismes rencontrés dans les réseaux synchrones ou isochrones

Les mécanismes rencontrés dans de tels réseaux sont généralement des mécanismes de réservation et d'allocation de ressources. Les services de communication offerts dans ce contexte sont orientés connexion. La phase de connexion permet de vérifier la disponibilité des ressources de bout en bout (ou au moins dans les nœuds du réseau), puis de les réserver. Les services d'allocation se chargent par la suite de délivrer un service de communication respectant les garanties promises à la connexion (exemple : les algorithmes d'ordonnancement de paquets dans les routeurs [ZHA 93]). Les ressources gérées dans les deux cas peuvent être la bande passante, les ressources mémoires et processeurs, etc.

Parmi les réseaux offrant des services synchrones, citons l'un des plus connus : le réseau FDDI. Ce dernier autorise le partage du médium par des communications asynchrones et synchrones (cf. partie 5.3.2). Il a bénéficié d'un intérêt prononcé de la part de la communauté scientifique proposant des solutions pour le support des applications multimédias réparties. Citons à titre d'exemple les travaux d'Anderson [AND 93] , de Zheng [ZHE 95] ou ceux de Tokuda et al. [TOK 92]. Ces derniers proposent un système de gestion de la QoS où l'utilisateur précise plusieurs niveaux de QoS. Comme pour Pegasus, à chaque niveau est associé un critère d'importance permettant à un gestionnaire de QoS de déterminer le niveau de QoS de chaque application, tout en maximisant le degré de satisfaction du système. Ces mécanismes sont implantés dans ARTS, un système pour les applications temps réel réparties. Le contrôle d'admission et la réservation de ressources sont effectués grâce au protocole CBSRP (*Capacity-Based Session Reservation Protocol*). Les ressources réservées sont la mémoire, la bande passante ainsi que le processeur. Le protocole autorise une renégociation des réservations. Les services de renégociation sont nécessaires puisque la plate-forme peut être amenée à changer le niveau de QoS de certaines applications lors de leur exécution. Tokuda et al. montrent comment, à partir des propriétés temporelles de FDDI, il est possible de traduire les contraintes de QoS utilisateur en contraintes sur la transmission des messages applicatifs. Ils proposent simultanément des formules pour effectuer le contrôle d'admission sur la bande passante synchrone. L'ordonnancement et le contrôle d'admission sur les ressources processeurs sont réalisées grâce à RM.

Si FDDI a par le passé fait l'objet de nombreuses propositions pour le support des applications multimédias réparties, il est aujourd'hui remplacé par des réseaux offrant des débits supérieurs et surtout une meilleure intégration de services (utilisation d'un même support physique par des trafics ayant des contraintes différentes). Son principal successeur est ATM [VET 95] (*Asynchronous Transfer Mode*). ATM propose plusieurs services de communication différents [FOR 96]. Dans le cadre d'applications multimédias, deux services sont particulièrement intéressants : le service CBR (*Constant Bit Rate*) et le service VBR-RT (*Variable Bit Rate-Real Time*). Les deux offrent des garanties sur les délais de communication (services isochrones) et sur les débits. Comme leur nom l'indique, le premier est adapté aux trafics à débits constants alors que le deuxième offre un support pour les trafics à débits variables (la réservation s'effectue alors sur des critères de débit crête, débit moyen et taille de rafales). A titre d'exemple, le service CBR est particulièrement bien adapté à un flot de données audio non compressé puisque son débit est constant. Par

contre, le service VBR s'applique mieux à des flots de données comme un film compressé par la norme MPEG 2 et qui génère un débit variable.

Malheureusement dans la pratique, selon les constructeurs, les plates-formes ATM n'offrent pas toujours l'intégralité de ces services. Ainsi, il est peu courant de disposer d'un service VBR. Celui-ci reste difficile à réaliser et à utiliser. Il existe toutefois des plates-formes opérationnelles. Les logiciels distribués par l'EPFL pour le support d'ATM sur Linux offre un service CBR [ALM 97]. Ce dernier est accessible soit grâce à une interface de programmation spécifique générant des trames AAL5, soit au travers d'un réseau virtuel Classical-IP [LAU 98] (dans ce deuxième cas, seul le débit crête peut être spécifié).

De nombreuses plates-formes multimédias architecturées autour d'un réseau ATM ont été proposées. C'est notamment le cas des travaux de Wray et al. avec leur plate-forme Medusa [WRA 94] ou du système basé sur Chorus proposé par Blair et al. [ROB 94].

Pour terminer, citons un standard émergeant qui offre des services isochrones : le bus IEEE 1394 (ou FireWire) [WIC 97]. Ce bus autorise la cohabitation d'un trafic asynchrone et d'un trafic isochrone. Il est particulièrement intéressant dans le cadre d'applications multimédias par sa capacité à interconnecter des machines et périphériques divers (imprimantes, caméras, télévisions, etc). Il offre de plus des débits importants (jusqu'à 400 Mbits/s).

2.3.2 Mécanismes rencontrés dans les services de communication asynchrones

Nous regardons maintenant le cas des services de communication dit asynchrones. De façon générale, il est difficile dans ces réseaux d'utiliser des mécanismes de réservation de ressources, on a donc plus couramment recours à des mécanismes de supervision. Le principe consiste à observer l'état du réseau afin d'adapter le comportement de l'application. Supposons que nous utilisions un réseau offrant des services de communication non connecté. Dans le schéma classique, on demande au récepteur de mesurer le taux de perte des paquets et de communiquer périodiquement cette information à l'émetteur. Le taux de perte des paquets étant un indice de congestion, l'émetteur adapte alors sa cadence d'émission aux ressources réseaux disponibles ; on parle de mécanismes de rétroaction. Ces mécanismes sont particulièrement bien adaptés à un protocole comme IP. Ils sont donc largement répandus dans les applications multimédias utilisées sur Internet. Ce principe a d'ailleurs été appliqué avec succès sur des applications multimédias telles que le logiciel IVS de l'INRIA qui offrent des services de vidéo-conférence sur Internet [BOL 94, DIO 95] ; ainsi que des décodeurs MPEG répartis (exemple : Cen et al. [CEN 95]).

Dans le cadre d'Internet, ces mécanismes ont fait l'objet d'une standardisation par l'IETF (*Internet Engineering Task Force*) au travers du protocole RTP [SCH 96] (*Real Time Protocol*). Ce protocole est très utilisé ; on pourra citer, par exemple, le projet FastWeb [FRY 96], l'outil Vic [CAN 94], etc. RTP est un protocole utilisateur appliquant le modèle ALF (*Application Level Framing*) [CLA 90]. Il est généralement implanté au dessus d'UDP. Il permet d'envoyer des paquets de données auxquels sont associés des

estampilles temporelles. Ces estampilles permettent alors d'effectuer des synchronisations (exemple : synchronisation voix-lèvres). RTP est couplé à un protocole de contrôle : le protocole RTCP (*Real Time Control Protocol*). Ce dernier offre des services de supervision. Avec RTCP, il est possible, par exemple, d'estimer un taux de perte des paquets [BUS 96]. Le protocole RTP n'exige pas que le réseau sous-jacent offre des garanties sur les services de communication (en termes de délai et de débit).

D'autres protocoles proposés ou en cours d'étude par l'IETF offrent des services de réservation de ressources. c'est notamment le cas du protocole RSVP [BRA 97a, BRA 97b] (*ReSerVation Protocol*) et du protocole ST-II [DEL 95].

2.4 Gestion de la mémoire

Comparativement aux ressources réseaux et processeurs, la gestion de la mémoire a fait l'objet de peu d'attention de la part de la communauté scientifique qui travaille sur les applications multimédias. Celles-ci sont pourtant très gourmandes en mémoire. Les besoins en mémoire interviennent pendant les phases de traitement des données (exemple : décompression d'une image), ou lors des transferts de données depuis ou vers des périphériques (tampons de lecture des disques [GEM 92, RAN 93], tampon utilisés lors de transfert réseaux, etc). De ce fait, les rares équipes qui ont proposé des mécanismes de réservation et d'allocation mémoire spécifiques aux applications multimédias ont essentiellement traité le problème de déterminer la quantité de mémoire nécessaire au respect d'une contrainte de QoS. C'est notamment le cas du projet Tenet où Ferrari et al. ont proposé des solutions pour calculer la taille mémoire nécessaire au respect de contraintes de QoS déterministes et probabilistes (taux de pertes de paquets) pour un trafic donné [FER 90]. Plus généralement, il existe plusieurs travaux traitant des algorithmes d'ordonnancement de paquets dans les réseaux à commutation où les besoins en mémoire ont été évalués pour chaque politique d'ordonnancement [GOP 96, PHI 96].

Les projets ci-dessus traitent des mécanismes de réservation. Pour ce qui est de l'allocation de la mémoire, celle-ci est presque toujours statique. Une allocation statique est satisfaisante dans une application comme celle de la vidéo-conférence. Pour l'application de télévision interactive, comme nous ne connaissons pas le nombre de flots, l'allocation statique est bien moins adaptée. Or, la gestion dynamique de la mémoire dans les applications temps réel est un sujet étudié depuis longtemps [BAK 78, LI 90, GAO 94, NIL 94]. Les problèmes habituellement abordés dans ce contexte résident dans la construction d'allocateurs mémoire dont le comportement temporel est prédictible (en général déterministe) ainsi que dans l'ordonnancement conjoint des applications et du ramasse-miettes. Par exemple, Henrikson propose une solution pour ordonnancer le ramasse-miettes et les applications grâce à Rate Monotonic [HEN 97]. Ces solutions ne sont toutefois pas complètement satisfaisantes pour les applications multimédias. En effet, elles ne prennent pas en compte leurs caractéristiques (besoin de garanties probabilistes, présence de dépendances entre les tâches qui se partagent des tampons, etc). Hormis quelques propositions, telles que celle de Johnstone qui a étudié des gestionnaires de mémoire probabilistes pour le temps réel mou [JOH 97],

à notre connaissance, il n'existe pas vraiment de travaux sur la gestion dynamique de la mémoire pour les applications multimédias.

2.5 Conclusions et motivations de cette thèse

Dans ce chapitre, nous avons présenté au travers de deux exemples d'applications multimédias, les principales solutions qui ont été proposées pour la gestion des ressources dans les systèmes multimédias. Nous avons cité des travaux relatifs à la gestion des ressources réseaux, des ressources processeurs et de la mémoire. Ces domaines ont bénéficié d'un effort important de la part de la communauté scientifique, et il existe aujourd'hui des solutions pour un grand nombre des besoins exprimés par les applications multimédias (hormis pour la gestion de la mémoire).

Les mécanismes d'allocation et de réservation de ressources qui ont été proposés à ce jour sont adaptés à la plupart des applications multimédias. Nous estimons toutefois qu'ils sont insuffisants pour construire des applications multimédias du fait de l'absence de leur intégration satisfaisante. En effet, chaque mécanisme propose ses abstractions propres. Le développement d'une application oblige alors le concepteur à traduire plusieurs fois la QoS requise par son application en fonction des abstractions associées à chaque ressource. Ainsi, une application qui doit afficher des images de taille fixe à un rythme donné contraint le concepteur à traduire la taille de l'image en débit pour la gestion des ressources réseaux, la cadence d'affichage en échéance pour le processeur, etc. L'absence d'un modèle global de haut niveau implique un effort important de la part du concepteur. Elle impose que le concepteur dispose de compétences étendues puisqu'il doit connaître pour chaque ressource les abstractions utilisées ainsi que le fonctionnement des directives de gestion de ressources qu'il doit invoquer. Tout ceci rend difficile la réalisation d'une application.

Nous préconisons l'utilisation d'un modèle de haut niveau qui offre des fonctionnalités de spécification modulaire permettant d'encapsuler la complexité des différents éléments du système. Ce modèle doit pouvoir être exploité de façon automatique, afin de générer les directives de gestion de ressources. La génération automatique des directives et l'utilisation d'un modèle global permet de masquer au concepteur une partie de la complexité des ressources qu'il souhaite utiliser. Selon l'environnement d'exécution, les directives peuvent être soit des directives d'allocation soit des directives de réservation, soit les deux à la fois.

Sans vraiment offrir un modèle de haut niveau, certains travaux ont proposé des solutions de gestion intégrée des ressources du système [TOK 92, SIJ 96]. Lorsqu'un tel modèle est proposé, les contraintes temporelles autorisées sont généralement peu flexibles. Ainsi Jeffay et Anderson proposent des solutions où toutes les tâches sont soumises à un même modèle de contraintes temporelles [JEF 92a, AND 93]. Cette solution facilite la construction des procédures de gestion automatique des ressources, mais il est alors difficile de modéliser efficacement tous les éléments d'un système. Il existe toutefois des propositions de modèle permettant la spécification de contraintes temporelles variées. Stefani et al. proposent un modèle orienté objets couplé à une logique temporelle [STE 96]. Le pouvoir d'expression de ce modèle est plus important que les propositions de Jeffay et Anderson.

Toutefois, leur plate-forme n'offre pas de service permettant la génération automatique des directives de gestion de ressources. Ainsi, les contraintes temporelles entre objets ne sont pas assurées de façon automatique par le système, mais le sont grâce à des objets de synchronisation construits par le concepteur à l'aide d'un langage synchrone (cf. chapitre 4).

Dans le cadre des applications et des systèmes d'exploitation que nous ciblons, nous estimons qu'à ce jour, les problèmes liés à l'intégration des mécanismes de gestion de ressources par un modèle de haut niveau ainsi que la génération automatique des directives de gestion de ressources posent des problèmes qui ne sont pas totalement résolus. C'est pourquoi nous allons proposer, dans les chapitres suivants, un modèle qui permet d'exprimer les contraintes de QoS d'une application et de générer de façon automatique les directives de gestion de ressources.

Dans cette thèse, nous nous limitons à exprimer des contraintes de QoS temporelles et nous nous intéressons à la génération des directives de gestion des ressources processeurs.

Chapitre 3

Survol de l'approche

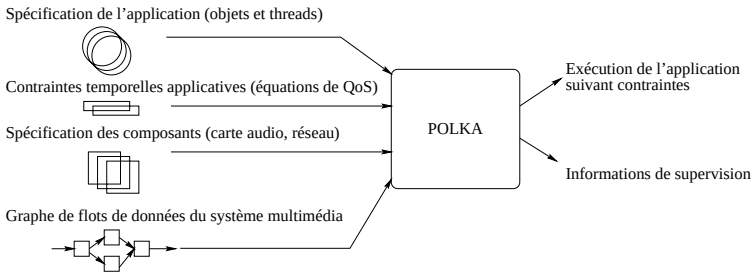


FIG. 3.1 – *Modèle et plate-forme* POLKA

Dans cette thèse, nous proposons un modèle de spécification et une plate-forme (la plate-forme POLKA) qui apportent des solutions aux problèmes soulevés dans les chapitres précédents. Le modèle de spécification que nous proposons, permet, d'une part, de décrire les aspects fonctionnels d'une application multimédia sous la forme d'objets et de threads ; il permet, d'autre part, de décrire le **comportement temporel** qu'elle doit observer (cf. figure 3.1).

La spécification temporelle d'une application est constituée d'un graphe de flots de données qui représente l'ensemble du système multimédia. Par système multimédia, nous entendons l'application ainsi que les éléments importants de la plate-forme support. Ces éléments peuvent être des éléments matériels (périphériques réseaux, carte audio, etc) mais aussi des composants logiciels (serveurs de fichiers, protocoles réseaux, décodeurs audio ou vidéo, etc). Les flèches du graphe symbolisent les flots de données continues multimédias et les nœuds modélisent les composants de l'application et du système. Ces composants sont annotés par des équations simples qui décrivent leur comportement temporel.

Cette spécification est alors exploitée par la plate-forme POLKA. Celle-ci ordonnance l'application conformément au comportement temporel voulu et fournit des informations de supervision sur l'ordonnancement ainsi calculé.

Dans cette thèse, nous nous focalisons donc sur les solutions à mettre en œuvre pour gérer automatiquement les ressources processeurs à partir d'une spécification temporelle

de haut niveau. La plate-forme a pour cible les applications partiellement prédictibles et les environnements d'exécution non dédiés (systèmes d'exploitation "grand public").

Nous donnons dans ce chapitre un bref aperçu de l'approche POLKA. Nous commençons donc par décrire rapidement le modèle de spécification que nous préconisons. Puis, nous présentons notre environnement de développement et d'exécution.

3.1 Un modèle de spécification pour les applications multimédias réparties

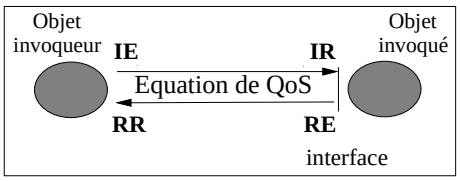


FIG. 3.2 – *Points d'observation durant une invocation de méthode*

Dans l'approche POLKA, une application est modélisée par un ensemble d'objets qui coopèrent pour traiter et présenter des flots de données continues ou **flots multimédias** [STE 96]. Un objet est une unité d'encapsulation de code et de données. Les objets interagissent uniquement par appel de méthodes exportées aux interfaces d'autres objets : ils ne communiquent pas par mémoire partagée. Ces objets sont animés par des threads. Comme dans Clouds [DAS 90], un thread POLKA est un flot d'exécution traversant éventuellement plusieurs objets et plusieurs machines au gré des invocations réalisées. Les objets et les threads d'une application constituent sa spécification fonctionnelle.

Un flot de données continues correspond à une suite d'invocations de méthode. Une invocation de méthode se décompose en événements (cf. figure 3.2). Ces événements sont l'émission d'une invocation par l'invoqueur (événement IE), la réception de cette invocation par l'objet invoqué (événement IR), l'émission de la réponse qui fait suite (événement RE), et la réception de la réponse par l'invoqueur (événement RR). Les invocations asynchrones donnent lieu aux seuls événements IE et IR. **Un flot multimédia correspond donc à une suite d'événements**. C'est sur ces événements que seront spécifiées les contraintes temporelles de l'application.

Illustrons notre modèle de spécification par une application (application qui sera détaillée dans le chapitre 8). Notre exemple est une application qui lit des éléments audio et vidéo d'un film stocké dans un ensemble de fichiers et qui les transmet par un réseau à un site distant où le film est visionné (cf. figure 3.3). On utilise la norme de compression MPEG-2 [ISO 95]. Cette application est constituée de trois objets : un objet *serveur* qui lit des trames MPEG sur un disque et les envoie à un objet distant : *sourceMpeg* ; un objet *sourceMpeg* qui consomme les trames qu'il a stockées dans ses tampons et les transmet à l'objet *decodeurMpeg* ; un objet *decodeurMpeg* qui décode des trames MPEG et les présente sur les périphériques de sortie (carte audio et écran géré par un serveur X11). Cet exemple

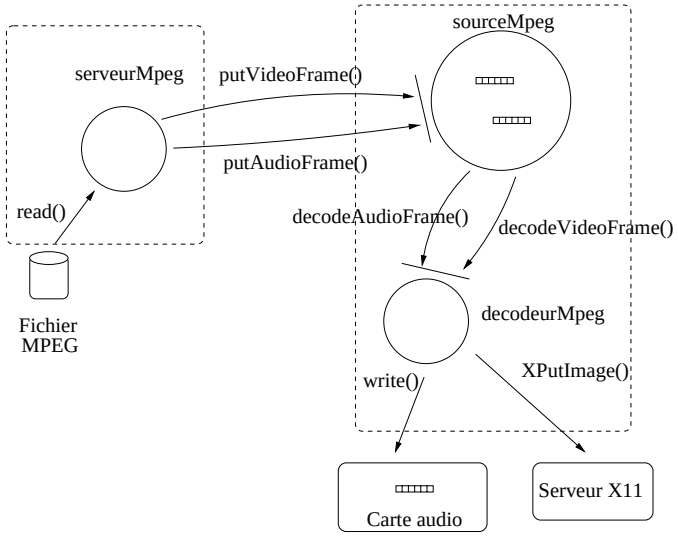


FIG. 3.3 – *Exemple d'une application répartie sur* POLKA

fait intervenir quatre threads. Ainsi pour la vidéo (respectivement pour l'audio), un thread remplit le tampon en bouclant sur l'invocation de la méthode $putVideoFrame$ (respectivement $putAudioFrame$) de l'objet $sourceMpeg$. Un deuxième thread consomme les trames du tampon et boucle sur l'invocation de la méthode $decodeVideoFrame$ (respectivement $decodeAudioFrame$) de l'objet $decodeurMpeg$.

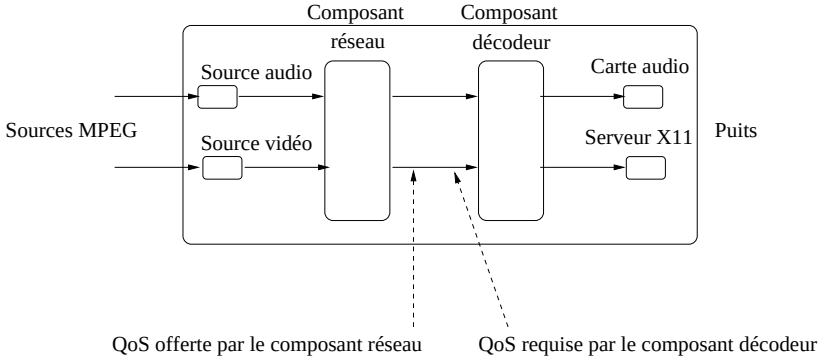


FIG. 3.4 – *Le graphe de flots de données d'un système multimédia*

Une fois les objets et les threads de l'application décrits, on peut spécifier les contraintes temporelles que celle-ci doit respecter. Pour ce faire, on modélise le système par un graphe de flots de données tel que celui de la figure 3.4. Le graphe de flots de données représente chaque flot multimédia, leurs sources, leurs puits et tous les éléments du système qui les manipulent. Chacun de ces éléments est modélisé par un composant dont le comportement temporel est décrit par un **contrat de QoS**. Un contrat est composé de deux jeux

d'équations de QoS : un jeu pour la QoS offerte et un jeu pour la QoS requise [BLA 98]. Les termes du contrat sont les suivants : **sous réserve du respect des contraintes de QoS requises par le composant en entrée, le composant s'engage à respecter une QoS offerte donnée**. Modéliser une application multimédia consiste donc à identifier les flots et les composants du système, puis, à spécifier les composants et les contrats de QoS.

Pour définir un contrat de QoS, on utilise des équations de QoS qui expriment des contraintes temporelles entre les événements (IE, IR, RE, RR) observables durant les invocations de méthode d'un objet. Les équations de QoS sont exprimées dans la logique temporelle QL [STE 93] (*QoS Language*). Regardons un exemple de contrainte qu'il est possible d'exprimer :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(o.decodeVideoFrame.RR, n+1) - \tau(o.decodeVideoFrame.RR, n) \leq \epsilon_2$$

où $o.decodeVideoFrame.IE$ est un événement observé lors de l'invocation de la méthode $decodeVideoFrame$ de l'objet o et $\tau(x, n)$, l'opérateur qui fournit la date de l'occurrence n de l'événement x. Cette inéquation stipule que les présentations de deux images successives doivent être séparées d'au moins ϵ_1 et d'au plus ϵ_2 unités de temps.

Il existe, bien sûr, d'autres formes de contraintes et de comportements exprimables dans notre modèle. Le chapitre 4 les décrit de façon plus complète. Nous nous limitons à ce jour à des contraintes déterministes. Une extension probabiliste de notre modèle est néanmoins proposée dans le chapitre 9.

3.2 Polka : une plate-forme pour le support des applications multimédias réparties

La plate-forme POLKA utilise les spécifications présentées ci-dessus pour **ordonnancer automatiquement** l'application de façon à respecter les contraintes temporelles spécifiées, dans la limite des ressources disponibles (exemples : processeur, mémoire et bande passante réseau).

Bien que les techniques de modélisation et d'ordonnancement proposées dans POLKA puissent être appliquées dans tous systèmes à objets où les interactions entre objets peuvent être observées, la plate-forme actuelle est basée sur un bus à objets au standard CORBA [OMG 99] (*Common Object Request Broker Architecture*). En pratique, les objets applicatifs sont donc des objets CORBA et les objets *decodeurMpeg* et *sourceMpeg* peuvent être définis par des interfaces IDL (*Interface Definition Language*).

La plate-forme est constituée d'un environnement d'exécution et d'un environnement de développement décrits de façon détaillé dans le chapitre 7.

L'environnement de développement offre un ensemble de compilateurs (les compilateurs *i2p*, *c2p* et *qc*) permettant de construire des spécifications de QoS et de générer des souches et des squelettes CORBA. Dans les plates-formes CORBA, les souches et les squelettes sont

des composants logiciels qui acheminent les invocations de méthode des clients vers les objets. Les souches et squelettes générées par le compilateur $i2p$ de POLKA offrent les mêmes services tout en interagissant avec notre environnement d'exécution pour ordonnancer les invocations de méthode conformément aux contraintes temporelles souhaitées.

L'environnement d'exécution est un démon composé d'objets CORBA qui analyse les descriptions de QoS, ordonnance les invocations de méthode et fournit des informations de supervision. Les algorithmes d'ordonnancement que nous utilisons pour traduire les spécifications de QoS en directives d'ordonnancement sont décrits dans le chapitre 6. Ceux-ci sont orientés échéances. Ils ne requièrent pas une connaissance a priori du temps d'exécution des tâches ; ce qui est intéressant compte tenu des applications et des systèmes que nous ciblons. L'environnement d'exécution est validé dans le chapitre 8 où nous présentons des mesures de performance. Ces mesures montrent que l'approche permet effectivement la prise en compte automatique des contraintes temporelles de façon significativement plus efficace qu'une plate-forme CORBA standard utilisant l'ordonnancement par défaut du système d'exploitation sous-jacent. Le surcoût engendré par la plate-forme est par ailleurs raisonnable.

Chapitre 4

Modélisation de systèmes multimédias

Dans ce chapitre, nous proposons un modèle de spécification pour les applications multimédias réparties. Ce chapitre est plus particulièrement axé sur la spécification qualitative des applications ; nous ne reviendrons donc pas sur la spécification fonctionnelle (objets et threads, cf. chapitre 3).

Les objectifs du modèle de spécification que nous proposons sont de permettre une spécification modulaire des contraintes de QoS des différents éléments d'un système multimédia. Le modèle doit refléter les dépendances temporelles existant entre les éléments du système. Il doit être suffisamment simple pour permettre une composition automatique et efficace des modules constituant une spécification. Il doit donc réaliser un compromis entre pouvoir d'expression et complexité. Nous espérons ainsi proposer des outils qui simplifient la phase de spécification grâce à une modélisation graduelle des applications ciblées : le concepteur définit le comportement temporel de son application en construisant des composants élémentaires qu'il va affiner et combiner. Par ce découpage en composants et par la réutilisation de composants déjà existants, le concepteur appréhende plus facilement la complexité de son application. La spécification d'une application s'en trouve simplifiée. Enfin, le modèle doit autoriser la spécification de bout en bout des contraintes temporelles associées aux flots de données multimédias [CAM 98].

Notre modèle ne dit rien sur le respect effectif des contraintes qu'il exprime. En effet, si garantie il y a, c'est la plate-forme qui exploite le modèle qui la fournit. Dans cette thèse, nous montrons dans les chapitres 6 et 7 qu'il est possible d'exploiter ce modèle dans le cadre d'applications partiellement prédictibles ; ceci n'interdit pas que le modèle proposé puisse être adapté à un environnement où une garantie de service est souhaitée.

Le plan de ce chapitre est le suivant. Dans la première partie, nous présentons notre modèle de spécification. Nous définissons le graphe de flots de données qui modélise un système multimédia ainsi que les équations de QoS prises en compte dans Polka. Puis, la définition du modèle est complétée par la proposition d'une bibliothèque de compo-

sants suffisante, dans un premier temps, pour modéliser les systèmes multimédias. Nous terminons le chapitre par une présentation des algorithmes de composition des composants.

4.1 Des composants au système : définition du modèle

Le modèle utilisé doit nous permettre de spécifier pour chaque flot de données multimédias les informations suivantes :

- Les informations qui définissent les flots : principalement les contraintes temporelles entre les éléments d'un flot ainsi que la taille des éléments. Nous nous limitons dans ce chapitre à des contraintes déterministes et nous ne nous préoccupons pas des contraintes de QoS spatiales ou relatives à la fiabilité des données (exemples : perte de paquets ou erreurs sur bits dans un flot de données transmis à travers un élément réseau).

- Les composants du système qui agissent sur des flots de données. Ces éléments peuvent être des composants matériels ou logiciels.

Chaque élément du système exprime ses contraintes temporelles sous la forme d'un ensemble de contrats de QoS. Un contrat est composé de deux jeux d'équations de QoS : un pour la QoS offerte et un pour la QoS requise [BLA 98]. Les termes du contrat sont les suivants : **sous réserve du respect des contraintes de QoS requises par le composant en entrée, le composant s'engage à respecter une QoS offerte donnée**. De façon générale, la composition d'un ensemble d'éléments annotés par des contrats de QoS est un problème complexe [LEB 98]. Nous verrons dans les parties 4.5 et 6.4.2 qu'il est néanmoins possible d'effectuer cette opération si certaines hypothèses simplificatrices sont utilisées. La principale hypothèse concerne la structure du système : lorsque l'on analyse la structure des applications multimédias (exemples : [BOU 95, POS 97, SIE 97]), on s'aperçoit qu'il en existe beaucoup dont les traitements sont organisés en pipeline ou en graphes de flots de données [ACK 82]. Par exemple, les différents processus de compressions/décompressions/présentations spécifiés dans la norme MPEG 2 s'expriment naturellement sous la forme de pipeline [ISO 94]. Il existe d'ailleurs de nombreux travaux qui offrent des mécanismes de gestion de ressources pour les applications multimédias et dont la spécification de ces applications est basée sur ces deux modèles [JEF 91, HOR 92, AND 93, JEF 95, MIC 97, MIT 99].

Ainsi, le modèle proposé par Anderson a pour objectif de gérer l'allocation combinée des ressources mémoires, réseaux et processeurs [AND 93]. Chaque ressource du système est modélisée par un composant. Le comportement de chaque composant est défini en termes de délais (principalement par un délai minimal et un délai maximal de transit dans le composant). Les composants sont connectés par des flèches qui modélisent les flots de données. Ce modèle a été implanté dans le système Dash [ROD 89].

La solution proposée par Jeffay dans YARTOS est proche de celle d'Anderson [JEF 95]. Mais celui-ci utilise une méthode intéressante pour évaluer les contraintes temporelles de

chaque nœud du graphe : seuls les nœuds sources du graphe sont définis par des contraintes temporelles, les contraintes des nœuds suivants sont calculées grâce à leur temps d'exécution et les contraintes de leurs nœuds successeurs.

Un autre exemple est le système DirectShow de Microsoft [MIC 97]. Là aussi, un flot de données modélise le système. DirectShow propose une solution pour gérer la QoS qui consiste à remonter des informations de supervision des nœuds puits vers les nœuds sources. Ces informations de QoS sont simples **et peu flexibles** ; elles sont essentiellement constituées d'une information précisant si le composant est en surcharge ou en "sous-charge", d'un rythme de traitement des données auquel les prédécesseurs doivent se conformer, ainsi que d'une estimation du retard ou de l'avance que le composant de présentation a pris. La QoS est gérée par chaque composant ; le code de l'application est donc fortement lié à sa QoS.

L'utilisation de graphe de flots de données est donc une méthode largement répandue et son utilisation pour représenter la composition de nos composants ne nous semble pas limitative. C'est donc tout naturellement que nous modélisons un système multimédia par un graphe de flots de données où les nœuds constituent les composants du système et où les flèches décrivent les flots de données multimédias.

Nos spécifications doivent manipuler le temps. Aussi, pour que l'on puisse raisonner facilement sur le temps, un système multimédia est défini comme un système réactif [HAR 85]. Un système réactif est un système qui réagit à des entrées provenant de façon répétitive de son environnement. A chaque jeu d'événements, il effectue un traitement et produit des sorties vers son environnement (par la suite, nous utiliserons indifféremment le terme d'événement ou de signal).

Pour modéliser un système réactif, nous utilisons, dans cette thèse, un langage basé sur le modèle synchrone[1]. Le modèle synchrone est fondé sur l'hypothèse du **synchronisme fort** [BER 87]. L'hypothèse du synchronisme fort stipule que l'exécution d'un système réactif synchrone nécessite zéro unité de temps. Dans la pratique, cette hypothèse se traduit par le fait qu'un système synchrone termine toujours l'exécution déclenchée par l'occurrence d'un jeu d'événements d'entrée avant l'arrivée d'un nouveau jeu d'événements. Le modèle de système réactif synchrone est particulièrement bien adapté pour spécifier des applications temps réel. Il permet, entre autres, de manipuler aisément la notion de temps. Un signal d'entrée du système réactif synchrone peut, par exemple, être utilisé pour modéliser une horloge. Chaque occurrence du signal est alors équivalente à un top de l'horloge. Le signal "horloge" constitue un signal d'activation du système réactif.

Le système réactif synchrone modélisant notre système multimédia possède trois groupes de signaux (cf. figure 4.1) :

- Le signal H_{phy} qui constitue un signal d'entrée modélisant le temps physique. Ce signal est diffusé à tous les composants de notre système multimédia.

[1] Nous utilisons le modèle synchrone pour clairement spécifier le comportement temporel d'un composant. Toutefois, la solution que nous décrivons dans ce chapitre ne requiert pas de façon absolue l'utilisation d'un tel modèle. La notion de système synchrone n'est d'ailleurs pas présente dans la mise en œuvre de notre plate-forme (cf. chapitre 7).

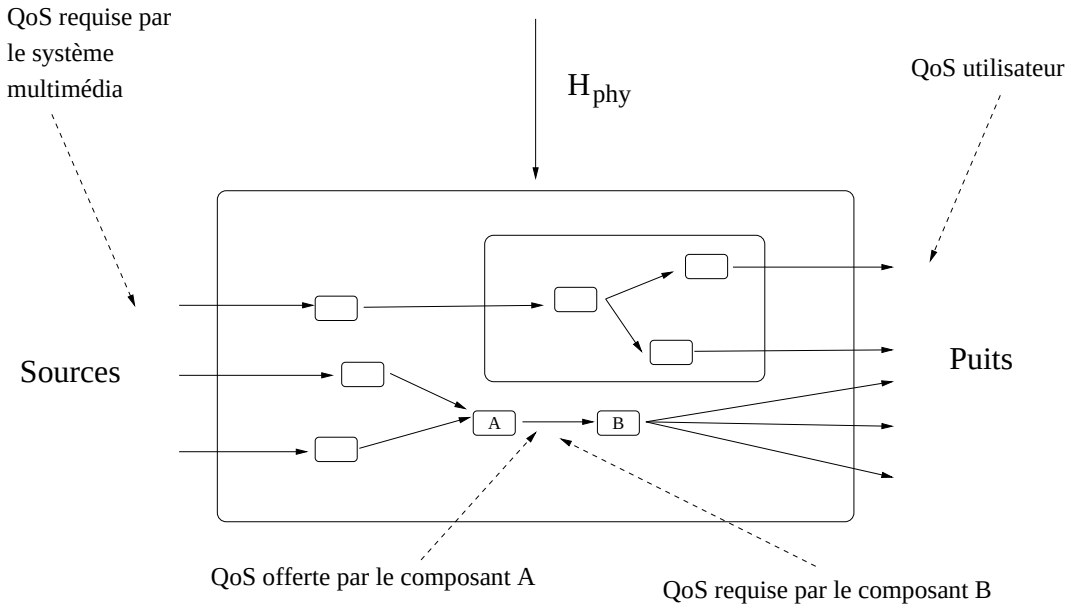


FIG. 4.1 – *Spécification d'un système multimédia*

- Les signaux d'entrée dont les occurrences modélisent les éléments des flots multimédias produits par les sources.

- Les signaux de sortie qui modélisent la production vers les puits d'éléments de flots multimédias.

Les signaux d'entrée et de sortie du système multimédia sont utilisés pour spécifier les caractéristiques temporelles existant entre les éléments des flots multimédias. Ces caractéristiques temporelles se matérialisent par des contraintes sur les occurrences des signaux du système réactif. Les équations de QoS spécifiées sur les signaux de sortie du système multimédia constituent la QoS utilisateur, c'est-à-dire la QoS que l'utilisateur souhaite obtenir du système multimédia. Les équations de QoS spécifiées sur les signaux d'entrée du système multimédia constituent la QoS que le système multimédia requiert pour satisfaire la QoS utilisateur.

Modéliser une application multimédia consiste d'abord à définir les flots multimédias ainsi que la QoS utilisateur. Puis, les éléments de la plate-forme multimédia sont modélisés par des composants. Le comportement temporel de chaque composant est défini par un ensemble de contrats de QoS [BLA 98]. **Grâce à la structure en graphe de flots de données de notre modèle, il est alors possible à partir de la QoS utilisateur et du comportement temporel des composants de déduire la QoS requise par le système multimédia. Cette dernière est évaluée composant par composant des puits jusqu'aux sources.**

4.2 Notion de composant

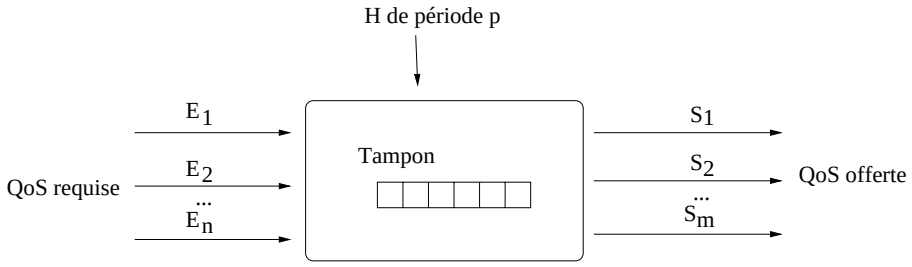


FIG. 4.2 – *Notion de composant*

Un composant est modélisé par un système réactif synchrone (cf. figure 4.2). Pour spécifier des systèmes réactifs synchrones, plusieurs langages synchrones ont été proposés par le passé. Citons, pour exemple, les plus connus qui sont Lustre [HAL 91], Esterel [BER 98], Signal [GUE 91] et les Statecharts [HAR 87]. Dans cette thèse, nous spécifions le comportement des composants en Esterel. Notons que les résultats présentés dans la suite de ce mémoire sont indépendants du langage synchrone choisi.

Le composant constitue la brique de base pour spécifier un système multimédia : le système multimédia est une composition de composants. Il est possible de combiner deux composants soit en séquence en connectant les signaux de sortie de l'un aux signaux d'entrée de l'autre, soit en parallèle en construisant un nouveau composant dont les signaux de sortie (respectivement d'entrée) sont les signaux de sortie (respectivement d'entrée) des deux composants combinés. Le modèle permet d'encapsuler plusieurs sous-composants combinés en un seul composant. Il offre ainsi la possibilité de modéliser par étapes un système complexe (en affinant successivement la description de ses composants). Les techniques de composition décrites ici sont proches de celles utilisées dans certains langages synchrones (tel qu'Esterel).

Un composant est constitué des signaux suivants :

- Le signal d'entrée H de période p. Ce signal modélise une horloge qui cadence l'activation du composant. La période de l'horloge H est exprimée en nombre de tops de l'horloge H_{phy} du système multimédia (cf. figure 4.1). Dans la suite de cette thèse, nous désignerons ces horloges par le terme "d'horloges logiques".

- Les signaux E_1, E_2, E_3, ..., E_n qui modélisent la livraison au composant des éléments des flots multimédias $1, 2, 3, ..., n$. Dans la suite de ce chapitre, les variables $QE_1, QE_2, QE_3, ..., QE_n$ dénoteront le volume d'informations véhiculées par une occurrence des signaux E_1, E_2, E_3, ..., E_n.

- Les signaux S_1, S_2, S_3, ..., S_m qui modélisent la sortie du composant des éléments des flots multimédias $1, 2, 3, ..., m$. Les variables $QS_1, QS_2, QS_3, ..., QS_m$ dénotent le volume d'informations véhiculées par une occurrence des signaux S_1, S_2, S_3, ..., S_m.

Les occurrences des signaux E et S sont utilisées pour spécifier la QoS offerte et la QoS requise par le composant. Enfin, nous verrons qu'un composant peut éventuellement contenir un tampon lui permettant de stocker les informations véhiculées par les signaux d'entrée.

Il existe, bien sûr, beaucoup de composants possibles ; ils diffèrent par un ou plusieurs paramètres : le composant possède ou non un tampon, exhibe plusieurs jeux d'équations de QoS ou tout simplement a un nombre variable de signaux d'entrée et de sortie. Il est ainsi possible de modéliser des composants effectuant du multiplexage de flots, du démultiplexage, du routage, etc (cf. figure 5.4 page 64). On peut néanmoins les classer en deux familles : ceux dont le comportement temporel est facile à déterminer hors ligne et ceux dont le comportement est difficile, voire impossible à déterminer a priori.

Pour cette deuxième famille de composants, il est possible d'utiliser un composant particulier pour obtenir une estimation d'un contrat de QoS. Pour ce faire, nous utilisons le composant dit "composant de rétroaction".

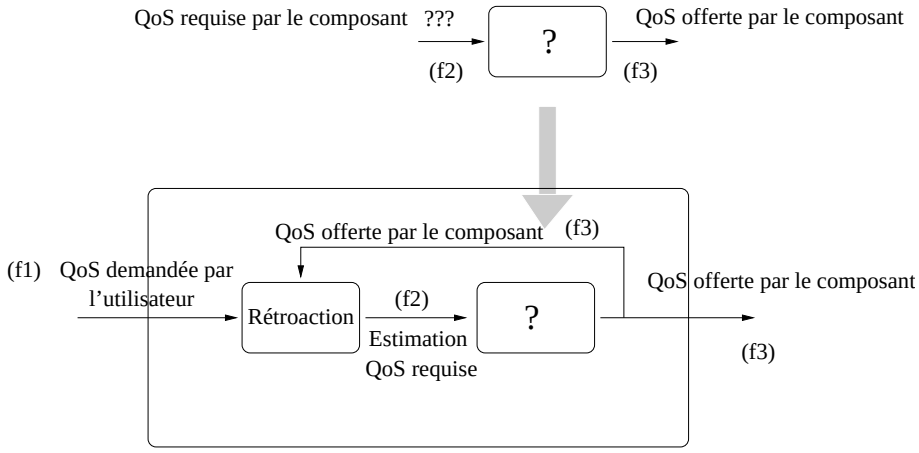


FIG. 4.3 – *Mécanisme de rétroaction*

D'une manière générale, un mécanisme de rétroaction est un mécanisme qui permet de piloter un système complexe et dont le comportement n'est pas connu a priori [CEN 97]. L'objectif du pilotage est d'amener le comportement du système au plus près d'un comportement voulu. Pour ce faire, le mécanisme de rétroaction observe le système piloté, puis agit sur celui-ci afin d'en modifier le comportement. Un mécanisme de rétroaction est donc constitué de plusieurs fonctions :

- Des fonctions d'observation du système.
- Des fonctions de "filtres". Ces fonctions déterminent l'état du système à partir d'informations fraîchement récoltées et d'un historique de son comportement. Par exemple, le débit disponible d'un réseau pourra être estimé en calculant une moyenne sur le débit disponible à des instants réguliers.

– Des fonctions de décision et de pilotage. A partir de l'état du système, ces fonctions lui appliquent des actions afin que son comportement s'approche du comportement souhaité par l'utilisateur.

Les mécanismes de rétroaction existant à ce jour sont nombreux. Dans [CEN 97], le lecteur trouvera une synthèse détaillée des mécanismes exploitables dans les systèmes multimédias. Appliqué à notre problématique, le composant de rétroaction observe le comportement d'un autre composant en collectant la QoS que celui-ci a offert, puis évalue la QoS requise du composant observé afin que ce dernier délivre à l'utilisateur une QoS la plus près possible de ce qu'il avait demandé.

Ainsi, le composant de rétroaction observe la QoS obtenue par le composant supervisé (équations (f3) de la figure 4.3), puis détermine la QoS requise (équations (f2)) qui permettra de se rapprocher au plus près de la QoS spécifiée par l'utilisateur (équations (f1)). En pratique, le composant de rétroaction est implanté par notre plate-forme d'exécution.

4.3 Expression des contraintes temporelles

Nous allons maintenant décrire les contraintes qu'il est possible d'exprimer sur les signaux d'entrée et de sortie de nos composants. Pour exprimer les contraintes temporelles d'applications temps réel et multimédias, de nombreuses méthodes ont été proposées dans la littérature.

Une première solution consiste à utiliser des constructions syntaxiques ajoutées aux langages de programmation des applications. C'est notamment la solution exploitée par le langage MPL (*Maruti Programming Language*) de Maruti [SAK 94], RTC (*Real Time Concurrency*) [WOL 91] ou Flex [LIN 88]. Une deuxième solution consiste à utiliser des techniques de spécification formelle. L'objectif visé ici est de spécifier des contraintes pouvant faire l'objet de preuves formelles automatisées. Les formalismes utilisables dans ce contexte sont, par exemple, les réseaux de Petri temporisés (solution exploitée par Diaz et al. [OWE 98] ainsi que par Al-Salqan et al. [CHO 96]). L'emploi de logiques adaptées aux applications temps réel est aussi rencontré [ALU 92] ; nous citerons les logiques RTL (*Real Time Logic*) [JAH 86] ou RTTL (*Real Time Temporal Logic*) [OST 92].

Dans cette thèse, nous optons pour cette dernière solution. Le formalisme utilisé pour l'expression des contraintes temporelles est fortement inspiré de la logique temporelle QL (*QoS Language*) proposée par Stefani et al. [BLA 98]. Notons que les propositions de cette thèse sont applicables quelle que soit la logique utilisée pour peu que celle-ci autorise l'expression de contraintes que nous qualifierons d'absolues et de relatives (classification basée sur celle proposée par Vega et Thomesse [VEG 97]) ; ce qui est le cas de QL. Par contraintes absolues, nous entendons celles qui datent l'occurrence d'un événement par la valeur d'une horloge (exemple : horloge du système). Les contraintes relatives sont celles qui mettent en relation les occurrences d'un ou plusieurs événements. Parmi les logiques qui satisfont à la contrainte ci-dessus, le choix entre l'une ou l'autre est motivé par un compromis entre le pouvoir d'expression de la logique choisie et ses possibilités de preuve

automatique. Nous ne traitons pas des problèmes liés à la réalisation de preuve dans cette thèse. QL est donc satisfaisant dans ce contexte.

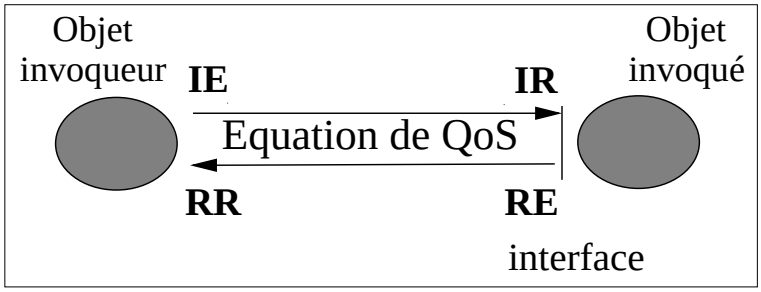


FIG. 4.4 – *Points d'observation durant une invocation de méthode*

Nos contraintes sont spécifiées sous la forme d'équations et d'inéquations. Par abus de langage, nous parlerons dans la suite de cette thèse **d'équations de QoS** pour désigner des équations ou des inéquations en QL. Les équations sont composées d'expressions construites par des constantes, des variables, les opérateurs d'addition, de soustraction, de multiplication et de division sur les entiers ainsi que par des expressions élémentaires de la forme $\tau(x, n)$ (où l'opérateur τ fournit la date de l'occurrence n de l'événement x). Les variables peuvent être quantifiées universellement ou existentiellement (opérateurs $\forall$ et $\exists$). Les événements référencés dans les équations peuvent être soit des tops d'horloges logiques (cf. partie 4.2), soit des événements observables durant les invocations de méthode d'un objet. Les horloges logiques permettent d'exprimer des contraintes absolues. La date d'occurrence du $n^{ème}$ top d'une horloge logique H_p de période p est donnée par $\tau(H_p, n) = n * p$.

Les événements observés lors des interactions entre objets sont utilisés pour exprimer des contraintes temporelles relatives. La figure 4.4 montre les événements captés lors des invocations de méthode. Ceux-ci sont l'émission d'une invocation par l'invoqueur (événement IE), la réception chez l'objet invoqué de cette invocation (événement IR), l'émission de la réponse qui fait suite (événement RE), et la réception de la réponse par l'invoqueur (événement RR). Le cas des invocations asynchrones permet d'observer uniquement les événements IE et IR.

Dans le cas général, l'expression des contraintes de QoS d'un composant consiste en une formule comprenant des conjonctions et des disjonctions d'équations (opérateurs $\wedge$ et $\vee$). La QoS requise ou offerte d'un composant est exprimée sous la forme d'une conjonction d'équations [BLA 98]. La disjonction d'équations est alors implicite lorsqu'un composant exporte plusieurs contrats de QoS (disjonction entre les contrats exportés par le composant).

A partir des définitions ci-dessus, il est possible de construire un nombre important d'équations de QoS. Nous décrivons ci-dessous les équations que nous rencontrerons le plus souvent dans cette thèse. Considérons d'abord l'équation suivante :

$$\forall\ n,k\ :\ \epsilon_1 \leq \tau(e,n+k) - \tau(e',n) \leq \epsilon_2 \tag{4.1}$$

où e et e' sont deux événements. Cette équation stipule que la $n^{ème}$ occurrence de e' doit être séparée d'au moins ϵ_1 et d'au plus ϵ_2 unités de temps de l'occurrence $n+k$ de e.

Un cas particulier intéressant de cette équation est celui où il existe un ϵ et une période p tels que $\epsilon_1 = p - \epsilon$ et $\epsilon_2 = p + \epsilon$. L'utilisation de cet artifice permet de spécifier une gigue maximale de $2 * \epsilon$ unités de temps entre les occurrences de e et de e'. Ici, la gigue désigne l'intervalle de temps maximum entre les occurrences de e et de e'. Il est possible d'utiliser une telle équation pour exprimer une contrainte de synchronisation intra-flot, mais on risque alors de voir le flot dériver (ainsi cette équation ne peut pas être utilisée pour spécifier un flot où les événements interviennent à un rythme régulier). Elle est en revanche bien adaptée pour exprimer des contraintes inter-flots (exemple : synchronisation voix-lèvres).

Une autre utilisation possible de cette équation consiste à borner des délais : si e' est un événement de début d'invocation et e la fin de cette invocation, notre équation permet de borner le temps de réponse de l'invocation par au plus ϵ_2 unités de temps et au moins ϵ_1 unités de temps.

Considérons maintenant l'équation :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(e,n) - \tau(H_p,n) \leq \epsilon_2 \tag{4.2}$$

où H_p désigne une horloge logique de période p. Comme auparavant, e désigne un événement. Cette équation peut modéliser une synchronisation intra-flot sur un flot audio ou vidéo par exemple. Les éléments successifs du flot doivent être présentés avec une gigue maximale de ϵ unités de temps (avec $\epsilon = \epsilon_2 - \epsilon_1$). Cette fois le flot ne peut dériver car il est asservi aux tops de l'horloge H_p. Une telle équation implique que deux occurrences de e soit séparées par au moins $p - (\epsilon_2 - \epsilon_1)$ et au plus $p + (\epsilon_2 - \epsilon_1)$ unités de temps. L'équation (4.2) implique donc que l'équation

$$\forall\ n\ :\ p - (\epsilon_2 - \epsilon_1) \leq \tau(e,n+1) - \tau(e,n) \leq p + (\epsilon_2 - \epsilon_1) \tag{4.3}$$

soit vraie. La réciproque est fausse (cf. démonstration A.1.1).

Il est bien sûr possible de combiner l'équation (4.2) avec l'équation (4.1) pour la mise en œuvre d'une synchronisation intra-flot, dont les événements du flot doivent être cadencés par une horloge, et où un délai plus contraignant entre les occurrences successives de e est nécessaire.

Enfin, une version plus contrainte de l'équation (4.2) peut prendre la forme suivante :

$$\forall\ n\ :\ \tau(e,n) = n * p \tag{4.4}$$

Cette équation modélise la contrainte intra-flot d'un flot de données continues où les éléments sont présentés toutes les p unités de temps (exemples : flot audio ou tout autre flot isochrone dans lequel on aura compensé la gigue pour que les occurrences de e interviennent à un rythme régulier).

4.4 Une bibliothèque de composants standards

Nous allons proposer dans cette partie plusieurs composants qui modélisent des comportements temporels fréquents dans les applications que nous ciblons. Bien sûr, il n'est pas possible de définir une liste exhaustive de composants, mais nous verrons dans les chapitres 5 et 8 que les trois composants que nous allons présenter permettent déjà de modéliser de nombreux éléments d'un système multimédia.

Les deux premiers composants sont relativement simples, ils ne possèdent pas de tampon et appliquent un temps de retard à chaque signal d'entrée. Le dernier utilise un tampon d'une taille donnée et permet de modéliser des composants qui créent, absorbent ou modifient une gigue. Une description Esterel de chacun des composants est fournie dans l'annexe B.

4.4.1 Le composant de "retard fixe"

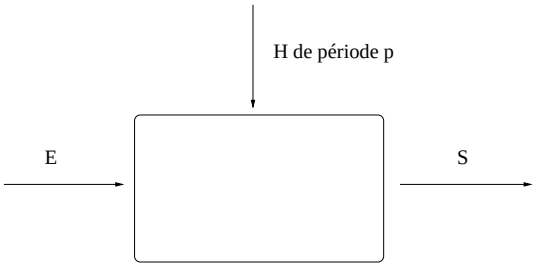


FIG. 4.5 – *Le composant "retard fixe"*

Le composant comprend deux signaux d'entrée et un signal de sortie (cf. figure 4.5 et annexe B.1). Il est cadencé par une horloge de période p dont chaque top est matérialisé par la réception d'un signal H. Le paramètre l du composant, ou retard, permet d'appliquer un retard fixe au signal E. Ainsi, l'occurrence d'un signal E est séparée par l tops d'horloge (soit l occurrences du signal H) du signal S correspondant.

On peut donc énoncer ceci :

Définition 1 (Temps de transit) *Le temps de transit d'un signal dans un composant de "retard fixe" qui applique un retard de l unités de temps est défini par :*

$$\forall\ n\ :\ \tau(S,n) - \tau(E,n) = l$$

La QoS offerte que nous allons étudier s'applique sur les signaux de sortie. Dans ce paragraphe, nous allons traiter deux QoS offertes différentes : une première qui utilise une horloge logique pour cadencer la livraison des signaux de sortie et une seconde qui n'utilise pas d'horloge logique. Nous supposerons ici que $QE = QS$.

4.4.1.1 QoS offerte avec une horloge logique

Lorsque la QoS offerte est de la forme :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'},n) - \tau(S,n) \leq \epsilon_2 \tag{4.5}$$

où $H'_{p'}$ est une horloge logique de période p' (avec p' multiple de p, la période de H). L'équation ci-dessus est une instance de l'équation (4.2). La QoS requise par le composant de retard fixe est alors de :

$$\forall\ n\ :\ \epsilon_1 + l \leq \tau(H'_{p'},n) - \tau(E,n) \leq \epsilon_2 + l \tag{4.6}$$

(cf. preuve A.1.2).

4.4.1.2 QoS offerte sans horloge logique

Nous regardons maintenant le cas où la QoS offerte est de la forme :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(S,n+r) - \tau(S,n) \leq \epsilon_2 \tag{4.7}$$

L'équation ci-dessus est une instance de l'équation (4.1). La contrainte de QoS requise par le composant est indépendante du retard fixe l et est de la forme :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(E,n+r) - \tau(E,n) \leq \epsilon_2 \tag{4.8}$$

(cf. preuve A.1.3).

4.4.2 Le composant de "retard variable"

Le deuxième composant (cf. annexe B.2) que nous allons regarder est proche du précédent mais cette fois, il applique un retard variable de l tops d'horloge, tel que $\alpha \leq l \leq \beta$; c'est donc une généralisation du composant de retard fixe.

Définition 2 (Temps de transit) *Le temps de transit d'un signal dans un composant de "retard variable" où le retard l vérifie $\alpha \leq l \leq \beta$ est de :*

$$\forall\ n\ :\ \alpha \leq \tau(S,n) - \tau(E,n) \leq \beta$$

Comme pour le composant précédent, nous regardons le cas d'une QoS offerte sans, puis avec horloge logique. Nous supposerons aussi que $QE = QS$.

4.4.2.1 QoS offerte sans horloge logique

Pour une QoS offerte de la forme :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(S,n+r) - \tau(S,n) \leq \epsilon_2 \tag{4.9}$$

la QoS requise de ce nouveau composant est :

$$\forall\ n,r\ :\ \epsilon_1 - \beta + \alpha \leq \tau(E,n+r) - \tau(E,n) \leq \epsilon_2 + \beta - \alpha \tag{4.10}$$

(cf. preuve A.1.4).

4.4.2.2 QoS offerte avec une horloge logique

La QoS offerte est de la forme :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'},n) - \tau(S,n) \leq \epsilon_2 \tag{4.11}$$

où $H'_{p'}$ est une horloge logique de période p' (avec p' multiple de p). La QoS requise par le composant de retard variable est alors de

$$\forall\ n\ :\ \epsilon_1 + \beta \leq \tau(H'_{p'},n) - \tau(E,n) \leq \epsilon_2 + \alpha \tag{4.12}$$

(cf. preuve A.1.5).

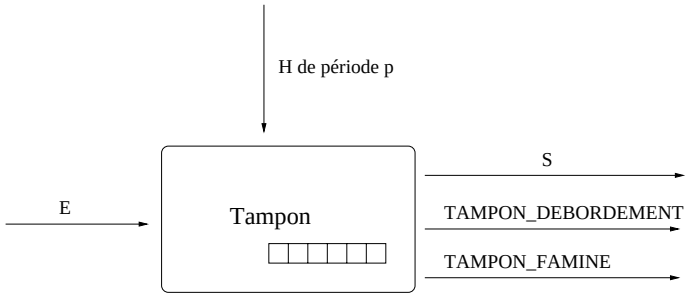


FIG. 4.6 – *Le composant compensateur de gigue*

4.4.3 Le composant compensateur de gigue

Le dernier composant est constitué d'un tampon de $TAILLE\_TAMPON$ octets (cf. figure 4.6 et annexe B.4). Il prend en entrée deux signaux H et E. Il génère trois sorties : les signaux S, $TAMPON\_FAMINE$ et $TAMPON\_DEBORDEMENT$.

Le composant est cadencé par une horloge. Chaque réception du signal H modélise un top de cette horloge. Les informations sont livrées au composant par le signal E. A la réception de ce signal, les données sont écrites dans le tampon du composant. Les signaux $TAMPON\_DEBORDEMENT$ (respectivement $TAMPON\_FAMINE$) permettent d'avertir l'environnement du composant qu'un débordement du tampon (respectivement qu'une famine du tampon) vient d'arriver.

La contrainte de QoS spécifiée par l'utilisateur porte sur une cadence de restitution des données ce qui s'exprime par l'équation :

$$\forall\ n\ :\ \tau(S,n) = n * CADENCE \tag{4.13}$$

où $CADENCE$ est la cadence de restitution de QS informations par le composant. QS (respectivement QE) est le volume de données véhiculées à chaque occurrence du signal S (respectivement E). En d'autres termes, l'équation de QoS spécifie que QS informations sont délivrées toutes les $CADENCE$ impulsions de l'horloge H. Cette équation est une instance de l'équation (4.4). Notons que contrairement aux composants précédents, nous n'avons plus nécessairement $QE = QS$. Dans la suite de ce paragraphe, nous supposerons qu'il existe une horloge Hs_{ps} qui cadence les occurrences des signaux S et dont la période est définie par $ps = CADENCE$.

L'équation (4.13) constitue donc la QoS offerte par le composant. Il faut maintenant déterminer les conditions à respecter pour que le composant puisse assurer cette QoS. Il faut trouver la contrainte sur les signaux d'entrée, et plus précisément sur le signal E, tel qu'il n'y ait jamais de débordement ou de famine du tampon du composant.

Ce problème est similaire à celui posé par la compensation de la gigue dans une couche AAL1 (*ATM Adaptation Layer*) (cf. preuve A.1.6). Des solutions de compensation dans ce contexte ont déjà été proposées [GAG 96].

Le problème soulevé est la restitution d'un trafic où les cellules, émises à une cadence régulière donnée, doivent être délivrées chez le récepteur à cette même cadence alors qu'elles sont transmises par un réseau isochrone. Plus précisément, on cherche à déterminer la borne sur la taille du tampon nécessaire pour compenser la variation sur les temps de communication (gigue). Pour cela, on détermine le temps maximum de transit d'une cellule ATM dans ce tampon en exploitant les temps maximaux et minimaux de transmission de celle-ci dans le réseau. Connaissant le débit de transmission, il est alors aisé d'en déduire la taille maximale du tampon qui permet d'éviter un débordement ou une famine. On montre que la durée de transit d'une cellule dans le tampon est bornée par $2 * g$ où g est la gigue maximale sur le temps de communication des cellules (ici, la gigue maximale est définie comme la différence entre le temps maximal de transmission et le temps minimal de transmission). La taille maximale du tampon est donc de $2 * g * d$ où d est le débit de la source.

Si nous appliquons ces résultats à notre composant, nous devons spécifier une équation de QoS qui détermine une gigue sur la livraison de signaux E qui soit compatible avec $TAILLE\_TAMPON$. Ce qui s'exprime par l'équation :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(He_{pe}, n) - \tau(E, n) \leq \epsilon_2 \tag{4.14}$$

où He_{pe} est une horloge de période pe multiple de p. Nous déterminons la valeur de pe (pour un QE et un QS donnés) aisément puisque le débit en entrée du composant doit être égal à son débit de sortie, soit $pe = \frac{ps*QE}{QS}$, où dans notre cas $pe = CADENCE * QE/QS$. Cette équation contraint la livraison des signaux E par rapport à la sortie des signaux S. Notons $\epsilon = \epsilon_2 - \epsilon_1$ la gigue maximale autorisée lors de la réception des signaux E afin de respecter la QoS offerte du composant. La gigue maximale constitue ici la différence entre l'instant d'arrivée au plus tôt et l'instant d'arrivée au plus tard d'un signal E. D'après [GAG 96], la taille du tampon est bornée par deux fois la gigue. Soit :

$$\begin{aligned} TAILLE\_TAMPON &= 2 * \epsilon * \frac{QS}{CADENCE} \\ \epsilon &= \frac{TAILLE\_TAMPON * CADENCE}{2 * QS} \end{aligned}$$

Ainsi, si $\epsilon \leq \frac{TAILLE\_TAMPON * CADENCE}{2 * QS}$, le composant pourra garantir sa QoS offerte.

Remarquons que l'on peut aussi effectuer le cheminement inverse, et à partir d'une contrainte de QoS, en déduire la taille du tampon. Cette information peut, par la suite, être avantageusement exploitée par un contrôleur d'admission.

4.4.4 Généralisation du composant compensateur de gigue

Nous donnons ici une généralisation du composant décrit dans le paragraphe précédent. Le nouveau composant offre une plus grande souplesse quant à la QoS qu'il offre, puisque

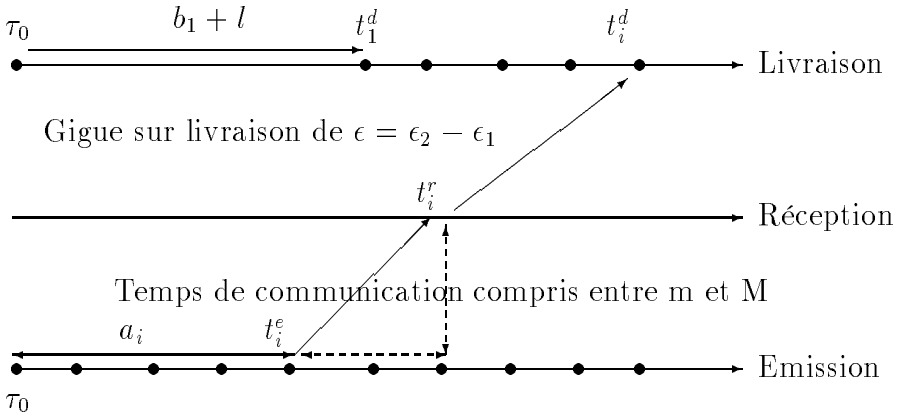


FIG. 4.7 – *Chronogramme d'un flot de cellules*

cette fois une gigue est spécifiable sur l'émission du signal de sortie S. Ainsi, la QoS offerte par le composant est de la forme :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(Hs_{ps}, n) - \tau(S, n) \leq \epsilon_2 \tag{4.15}$$

Où Hs_{ps} est une horloge de période ps (ps est multiple de p) qui rythme les émissions du signal S et où $\epsilon = \epsilon_2 - \epsilon_1$ constitue la gigue maximale autorisée entre les tops de l'horloge Hs_{ps} et les émissions du signal S. Notons que le compensateur de gigue du paragraphe précédent est un cas particulier de ce composant avec $\epsilon_2 - \epsilon_1 = 0$. Cette QoS offerte permet de modéliser des composants qui créent, absorbent ou modifient une gigue sur un flot de données continues.

Pour déduire la QoS requise, nous procédons en deux étapes :

1. D'abord, nous généralisons le résultat [GAG 96] utilisé dans la couche AAL1 d'ATM que nous avions exploité pour notre compensateur de gigue.

2. Puis, nous appliquons cette généralisation à notre nouveau composant.

4.4.4.1 Livraison d'un flot de cellules avec une gigue donnée

Le problème initialement traité consistait à absorber chez un récepteur la variation des délais de communication de cellules ATM émises à une cadence régulière, et que l'on souhaitait délivrer chez le récepteur à cette même cadence. Pour la mise en œuvre de ce mécanisme de compensation de gigue, il est nécessaire de déterminer deux informations :

– k, la taille du tampon chez le récepteur (taille minimale mais suffisante pour éviter tout débordement).

– l, le temps de retard appliqué à la livraison du flot de cellules chez le récepteur afin d'éviter une famine sur le tampon.

k et l sont déterminés pour une gigue maximale donnée sur les temps de communication (gigue que nous noterons $\triangle = M - m$ où M est le temps de communication maximal et m le temps de communication minimal).

Le problème adressé ici est proche du précédent. Nous autorisons toutefois l'existence d'une gigue $\epsilon = \epsilon_2 - \epsilon_1$ sur la livraison aux couches supérieures des cellules. Ainsi, si c est la cadence de livraison des cellules et t_1^d, la date de livraison de la première cellule chez le récepteur, la $i$ème cellule sera délivrée entre les instants $(i * c + t_1^d) + \epsilon_1$ et $(i * c + t_1^d) + \epsilon_2$ (cf. figure 4.7) alors que le problème initial spécifie que la cellule soit délivrée à l'instant $i * c + t_1^d$. Une solution possible à ce nouveau problème est la suivante :

Propriété 1 (Instant de livraison) *Soit τ_0, l'instant de début du flot de cellules (instant d'émission de la première cellule) ; soit l, le retard à appliquer à la livraison du flot ; soit $\epsilon = \epsilon_2 - \epsilon_1$, la gigue autorisée sur la livraison d'une cellule reçue précédemment ; soit $\triangle$ la gigue maximale sur la communication des cellules (avec $\triangle = M - m$). L'instant de livraison de la première cellule doit être retardé de $l = \triangle - \epsilon_1$ unités de temps après τ_0 pour éviter une famine du tampon chez le récepteur.*

Propriété 2 (Taille du tampon chez le récepteur) *Soit d le débit de la source ; soit $\epsilon = \epsilon_2 - \epsilon_1$, la gigue autorisée sur la livraison d'une cellule reçue précédemment ; soit $\triangle$ la gigue maximale sur la communication des cellules (avec $\triangle = M - m$). La taille minimale du tampon chez le récepteur afin d'éviter un débordement est de $d.(2.\triangle + \epsilon)$.*

Une démonstration de ces deux propriétés est donnée dans A.1.7.

4.4.4.2 Application à notre composant

Nous revenons maintenant à notre composant. A partir des résultats précédents, nous allons définir la QoS requise pour garantir la QoS offerte spécifiée par l'équation (4.15). La QoS requise est de la forme :

$$\forall\ n\ :\ \epsilon'_1 \leq \tau(He_{pe}, n) - \tau(E, n) \leq \epsilon'_2 \tag{4.16}$$

où He_{pe} est l'horloge qui cadence les occurrences du signal E, et dont la période pe est multiple de p. Le problème consiste donc à déterminer les valeurs possibles de pe et de $\epsilon' = \epsilon'_2 - \epsilon'_1$ dans le cadre suivant :

– En supposant la QoS offerte donnée (ou en d'autres termes ps et ϵ).

– En respectant la QoS offerte sans débordement et sans famine du tampon.

– En supposant que QE et QS soient connus.

On sait que le débit en sortie est égal à QS/ps informations par unité de temps. La propriété (2) permet de définir ϵ' tout en garantissant la QoS offerte et l'absence de débordement et de famine du tampon : soit k, la taille de ce tampon, on obtient :

$$k = \frac{QS}{ps} * (2 * \epsilon' + \epsilon)$$

et donc :

$$\epsilon' = \frac{\frac{k*ps}{QS} - \epsilon}{2}$$

Il ne nous reste plus qu'à évaluer pe ce qui est trivial puisque le débit en entrée du composant est nécessairement égal à celui en sortie, et donc $pe = \frac{ps*QE}{QS}$.

4.5 Composants et compositions

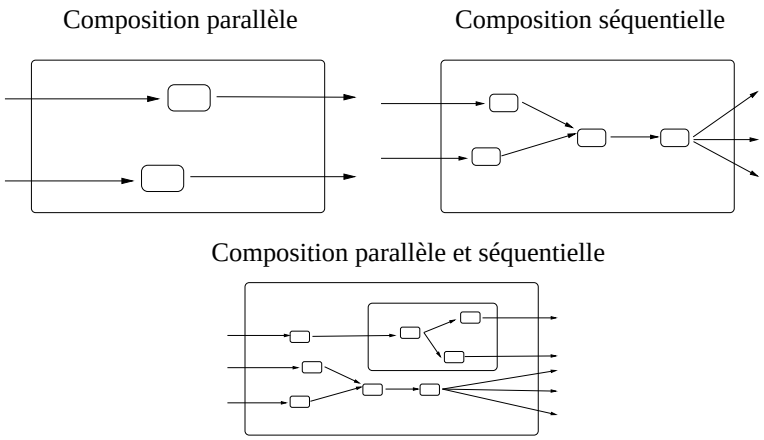


FIG. 4.8 – *Types de composition*

Dans ce paragraphe, nous détaillons la notion de composition. L'objectif des outils de composition est d'offrir des techniques permettant de faciliter la spécification temporelle d'une application. En effet, prise dans son intégralité, la modélisation du comportement temporel d'une application est difficile. Une modélisation modulaire, composant par composant, permet de simplifier la tâche du concepteur. D'abord parce que le concepteur peut se concentrer individuellement sur chacune des parties du système. Ensuite parce qu'il est possible de factoriser le travail de spécification par le biais de bibliothèques de composants (comme celle décrite précédemment).

Les méthodes de composition dans POLKA sont de deux types : les compositions parallèles et les compositions séquentielles (cf. premier et deuxième composants de la figure 4.8). Il est bien sûr possible de mixer les deux (cf. troisième composant de la figure 4.8). Réaliser une composition consiste, lors de la définition d'un composant, à lister ses sous-composants

en connectant leurs différents signaux. Les signaux non connectés constituent des signaux d'entrée et de sortie du composant ainsi défini. Chaque signal d'entrée (respectivement de sortie) non connecté d'un sous-composant constitue un signal d'entrée (respectivement de sortie) du composant global. La façon dont sont connectés les signaux définit si l'utilisateur réalise une composition parallèle, séquentielle ou les deux. Dans une composition parallèle, aucun des signaux des sous-composants n'est connecté. Tous les signaux des sous-composants sont donc des signaux du composant ainsi construit. Lors d'une composition séquentielle, certains signaux de sortie des sous-composants sont connectés à des signaux d'entrée d'autres sous-composants. Le nombre de signaux du composant résultat est donc inférieur au nombre total des signaux des sous-composants (sauf dans le cas ou le composant résultant introduit de nouveaux signaux). La composition, par la connexion des différents signaux, définit un graphe (éventuellement non connexe) où chaque nœud est un composant et chaque flèche modélise une connexion entre deux composants.

Lorsque l'on construit un composant par composition, définir la liste de ses sous-composants et spécifier leurs connexions n'est pas suffisant. En effet, comme pour tout composant, qu'il possède ou non des sous-composants, le concepteur peut éventuellement ajouter de nouvelles contraintes de QoS qui s'additionnent alors à celles de ses sous-composants. Prenons l'exemple d'un film sur lequel un traitement est effectué. Si le composant $film$ modélise le logiciel qui effectue ce traitement, il est possible de l'affiner en deux sous-composants. Chaque sous-composant traite un des flots du film (audio ou vidéo). Les sous-composants introduisent les équations de QoS définissant la synchronisation intra-flot pour le flot dont ils ont la charge. Le composant $film$ hérite donc naturellement des deux synchronisations intra-flots. Puis, le composant $film$ ajoute une équation qui définit la synchronisation inter-flots.

De cet exemple, il ressort que pour un composant, le contenu de chacun de ses contrats de QoS peut être disséminé dans plusieurs de ses sous-composants. Lors de la composition d'un ou plusieurs composants, les outils de POLKA doivent déterminer de façon automatique les contrats de QoS dans leur intégralité. Pour construire ces contrats de QoS, deux cas de figure peuvent se présenter ; soit la composition est parallèle, soit elle est séquentielle.

Dans le cas d'une composition parallèle, il est facile de trouver le contrat de QoS global puisqu'il suffit de faire l'union des contrats de QoS des sous-composants. En effet, une composition parallèle modélise finalement une conjonction de contraintes sur les flots entrant et sortant d'un composant.

```
struct composant {
    qos* requise;
    qos* offerte;
    struct composant* sous_composants[MAX_SOUS_COMPOSANTS];
};
```

FIG. 4.9 – *Structure décrivant un composant*

Dans le cas d'une composition séquentielle, le problème est nettement plus complexe. En effet, la QoS requise doit être déterminée en fonction du comportement temporel des sous-composants. **La QoS requise d'un composant correspond à sa QoS offerte sur laquelle est appliquée le comportement temporel de chacun de ses sous-composants en respectant l'ordre de leur connexion.**

```
composant* compose(composant c)
{
   composant* res;

   res→requise=c.requise;
   res→offerte=c.offerte;

   Pour chaque élément s du tableau c.sous_composants
      res→sous_composants[s]=compose(c.sous_composants[s]);
   Fpour;

   /* Phase de composition séquentielle */
   Tant qu'il reste des arcs dans le graphe g de res
      Choisir un composant k sans successeur;
      Pour chaque composant p prédécesseur de k
         Reporter k→offerte sur p→offerte;
      Fpour;
      Supprimer le composant k du graphe g ainsi que
         tous les arcs arrivant sur k;
   Ftant;

   /* Phase de composition parallèle */
   Pour chaque composant k restant dans g
      res→requise=res→requise ⋃ k.requise;
      res→offerte=res→offerte ⋃ k.offerte;
   Fpour;

   Retourner res;
}
```

FIG. 4.10 – *Algorithme de composition*

Construire le contrat de QoS d'un composant à partir des contrats de QoS de ses sous-composants peut être réalisé grâce à l'algorithme décrit par la fonction *compose*() de la figure 4.10. Nous utilisons une syntaxe proche du C. Le paramètre *c* de la fonction est le composant à traiter et la valeur de retour *res* pointe sur le composant obtenu. L'algorithme manipule une représentation des composants sous la forme d'une structure (cf. figure 4.9). Celle-ci comprend pour l'essentiel un ensemble d'équations formant la QoS requise (champ

requise), un ensemble d'équations formant la QoS offerte (champ $offerte$) ainsi qu'un ensemble de sous-composants (champ *sous_composants*). L'algorithme utilise un graphe orienté (nommé g dans la figure 4.10) qui modélise les connexions entre les sous-composants. L'algorithme procède en deux phases :

1. Il réduit d'abord le graphe g en un graphe sans arc. Cette étape consiste à remonter les contraintes de QoS de la QoS offerte vers la QoS requise ; nous parcourons donc le graphe des feuilles jusqu'aux racines. Les composants sont supprimés au fur et à mesure de leur traversée pour finalement obtenir un graphe constitué d'un ensemble de nœuds sans arc. Cette première phase se résume en fait par l'application sur les contrats de QoS de la composition séquentielle.

2. La phase suivante consiste à fusionner l'ensemble des contraintes de QoS. En effet, après la première phase, la composition est réduite à une composition parallèle. L'union des contrats de QoS est donc suffisante.

Notons qu'avant d'effectuer la phase de composition séquentielle et parallèle, la fonction *compose*() est invoquée de façon récursive sur chacun des sous-composants. En effet, rien n'empêche un sous-composant d'être lui-même constitué de composants plus élémentaires.

Evaluons graduellement la complexité de cet algorithme. L'application de la composition séquentielle est en $O(lhi)$. l est le nombre de sous-composants (et donc le nombre de nœuds du graphe g). L'opération de report du comportement temporel d'un sous-composant consiste à modifier les équations de QoS offerte de ses composants prédécesseurs dans le graphe g. Si h est le nombre d'équations maximal dans un contrat de QoS, cette opération est donc en $O(h)$. Enfin i est le demi-degré sortant maximal de g.

La composition parallèle est plus simple : l'union des contrats de QoS est en $O(l)$. Sans la composition récursive, la complexité est donc en $O(lhi) + O(l) = O(lhi)$. Toutefois dans le cas général, un sous-composant peut lui aussi être une composition ; d'où l'application récursive de la fonction *compose*(). La complexité d'un tel algorithme devient malheureusement exponentielle. Si x est la profondeur de composition, c'est-à-dire le nombre de compositions successives qu'il a fallu pour construire le composant, alors la complexité de notre algorithme est en $O(l^{(x-1)} * lhi) = O(l^x * hi)$. En pratique, l'utilisation d'un tel algorithme reste possible car la profondeur de composition n'est pas grande. De plus, il est possible d'effectuer ces opérations de composition au début de l'exécution d'une application, voire hors ligne. Dans ce contexte, la complexité de l'algorithme de composition est moins importante.

Pour terminer cette discussion sur la composition, nous donnons deux exemples de composition : le premier exemple illustre une composition parallèle, le second une composition séquentielle.

4.5.1 Exemple de composition parallèle

Le composant décrit ici (que nous appellerons m) est une composition parallèle de deux composants de retard m_1 et m_2 (cf. figure 4.11). Chaque composant de retard possède un

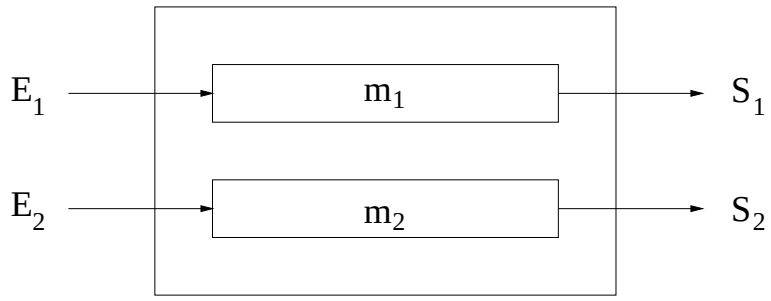


FIG. 4.11 – *Exemple de composition parallèle*

signal en entrée (respectivement E_1 et E_2) et un signal en sortie (respectivement S_1 et S_2). Nous supposerons que $QS_1 = QS_2 = QE_1 = QE_2$; rappelons que QS_i est le volume d'informations véhiculées par le signal S_i de même que QE_j est le volume d'informations véhiculées par le signal E_j. La QoS offerte par le composant m que nous allons étudier est de la forme :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(S_1, n) - \tau(S_2, n) \leq \epsilon_2 \tag{4.17}$$

Un tel composant peut être utilisé pour modéliser deux flots multimédias sur lesquels une synchronisation inter-flots est spécifiée. Les contraintes de QoS héritées des composants m_1 et m_2 modélisent les synchronisations intra-flots. Dans la suite, nous regardons deux cas de figure : le cas où m_1 et m_2 sont des composants de retard fixe et le cas où m_1 et m_2 sont des composants de retard variable.

4.5.1.1 Avec des composants de retard fixe

Dans ce premier cas de figure, si l'on note l_1 et l_2, les temps de retard appliqués par les composants m_1 et m_2, la QoS requise par le composant m pour satisfaire la QoS offerte décrite ci-dessus est évaluée de la façon suivante :

$$\forall\ n\ :\ \epsilon_1 - l_1 + l_2 \leq \tau(E_1, n) - \tau(E_2, n) \leq \epsilon_2 - l_1 + l_2 \tag{4.18}$$

(cf. démonstration A.1.8).

4.5.1.2 Avec des composants de retard variable

Dans ce deuxième cas de figure, les temps de retard l_1 et l_2 sont définis par :

$$\begin{cases} r_1 \leq l_1 \leq R_1 \\ r_2 \leq l_2 \leq R_2 \end{cases}$$

où r_1 et r_2 sont les retards minimaux appliqués par les composants m_1 et m_2 et où R_1 et R_2 sont les retards maximaux de ces mêmes composants. Dans ces conditions, la QoS requise par le composant m est la suivante :

$$\forall\ n\ :\ \epsilon_1 - R_1 + r_2 \leq \tau(E_1, n) - \tau(E_2, n) \leq \epsilon_2 - r_1 + R_2 \tag{4.19}$$

(cf. démonstration A.1.9).

4.5.2 Exemple de composition séquentielle

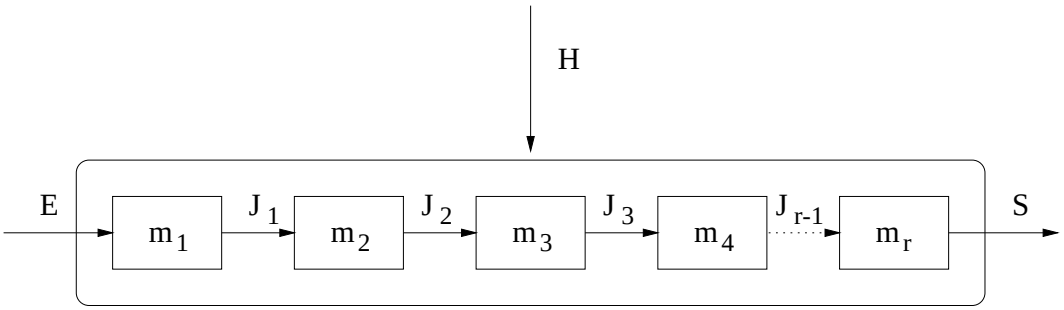


FIG. 4.12 – *Exemple de mise en séquence de composants*

Voyons maintenant le cas de la composition séquentielle. La figure 4.12 décrit un composant m constitué de r sous-composants (composants m_1 à m_r). La composition séquentielle autorise la combinaison de composants différents, néanmoins, ce paragraphe traite d'un exemple dont tous les composants sont identiques. Nous regardons successivement les cas où les composants sont des composants de retard fixe, puis le cas où les composants sont des composants de retard variable. La QoS offerte évaluée est la suivante :

$$\forall\ n, k\ :\ \epsilon_1 \leq \tau(S, n+k) - \tau(S, n) \leq \epsilon_2 \tag{4.20}$$

4.5.2.1 Avec des composants de retard fixe

Soit l_i, le temps de retard fixe appliqué par le composant m_i (tel que $1 \leq i \leq r$). La QoS requise par le composant m pour satisfaire la QoS offerte de l'équation (4.20) est :

$$\forall\ n, k\ :\ \epsilon_1 \leq \tau(E, n+k) - \tau(E, n) \leq \epsilon_2 \tag{4.21}$$

(cf. preuve A.1.10). Notons que si la QoS offerte et la QoS requise sont identiques à celles du paragraphe 4.4.1, le comportement du composant m reste différent de celui d'un unique composant à retard fixe. En effet, le composant m contient indirectement un

tampon de r cases puisque, à un instant donné, un signal peut être en attente d'émission dans chacun des sous-composants de m. La composition séquentielle peut donc être utilisée pour ajouter un tampon à un composant complexe sans pour autant être obligé de refaire l'analyse de son comportement temporel. C'est d'autant plus intéressant qu'il existe des composants qu'il est initialement difficile de modéliser avec un tampon.

4.5.2.2 Avec des composants de retard variable

Nous terminons avec le cas où les composants sont des composants à retard variable et où le retard l_i du composant m_i est de $\alpha_i \leq l_i \leq \beta_i$ unités de temps (avec $1 \leq i \leq r$). Dans ce cas la QoS requise est :

$$\forall\ n,k\ :\ \sum_{j=1}^{r}(-\beta_j+\alpha_j)+\epsilon_1 \leq \tau(E,n+k)-\tau(E,n) \leq \epsilon_2+\sum_{j=1}^{r}(\beta_j-\alpha_j) \tag{4.22}$$

(cf. preuve A.1.11).

4.6 Conclusion

Nous avons proposé dans ce chapitre un modèle orienté flots de données pour exprimer les contraintes temporelles qui existent entre les différents éléments d'un système multimédia. Chaque élément du système est spécifié par un composant. Son comportement temporel est alors décrit par un couple de jeux d'équations de QoS : la QoS offerte et la QoS requise du composant. Notre modèle permet de spécifier un système pas à pas, composant par composant. Ces derniers sont alors combinés de façon séquentielle ou parallèle afin de définir finalement le système dans son intégralité. Le modèle retenu est simple, nous montrons dans la partie 4.5 et dans le chapitre 6 qu'il autorise une manipulation automatique des spécifications avec une complexité raisonnable. Toutefois, de part le caractère déterministe des contraintes temporelles utilisées dans notre modèle, nous verrons que ce dernier ne permet pas de représenter des flots de données dont le débit est variable (cf. chapitres 5 et 9). Enfin, nous avons proposé un ensemble de composants constituant une bibliothèque qui nous semble suffisant, dans un premier temps, pour exprimer un nombre important des éléments présents dans un système multimédia.

Chapitre 5

Application du modèle : spécification d'éléments réseaux

L'objectif de ce chapitre est d'illustrer le modèle proposé dans le chapitre précédent. Pour ce faire, nous modélisons plusieurs éléments réseaux possédant des propriétés temporelles différentes.

Nous commençons par définir un composant de base : le composant élémentaire. Par la spécification de son comportement temporel, nous modélisons un réseau à délai de communication constant, puis un réseau synchrone, isochrone et enfin asynchrone [STE 95b]. Nous nous aidons de la bibliothèque de composants proposée dans la partie 4.4 ainsi que de la composition séquentielle.

5.1 Le composant élémentaire

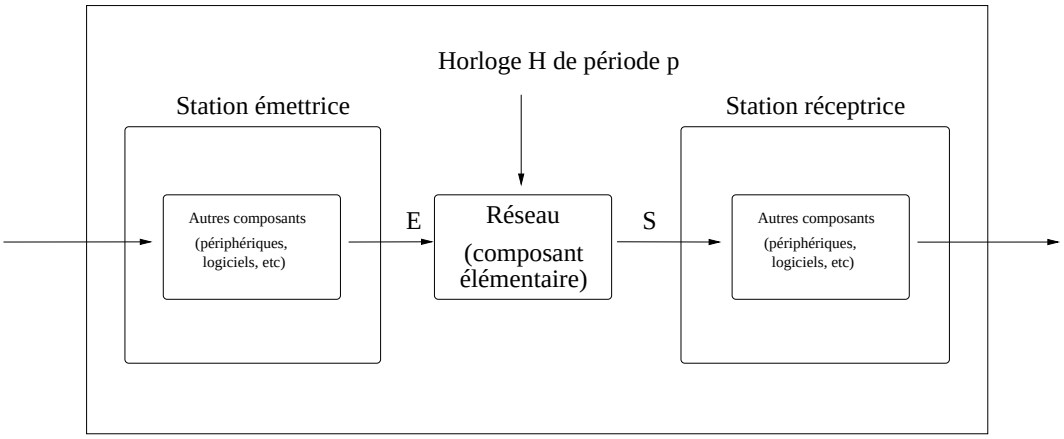


FIG. 5.1 – *Un composant réseau*

Le composant réseau élémentaire est défini comme un système réactif faisant intervenir trois signaux (cf. annexe B.5 et figure 5.1). Le composant est cadencé par une horloge H de période p. Le signal E modélise l'arrivée d'une donnée dans le réseau. Le signal S modélise la sortie d'une donnée du réseau. Ces données sont supposées être de taille fixe. Le composant ne possède pas de tampon. L'objectif du composant élémentaire est d'offrir un moyen simple de raisonner sur les occurrences de ces signaux. Ainsi, afin de simplifier son comportement temporel, nous supposons que le composant respecte les hypothèses suivantes :

Hypothèses :

1.a Le composant ne possède pas de tampon. Une donnée ne peut donc être émise que s'il n'existe aucune autre donnée présente dans le composant, ou encore : $\forall\ i\ :\ \tau(E,i) \geq \tau(S,i-1)$. De ce fait, le composant modélise des réseaux dont le mode de fonctionnement est exclusif (exemple : Token ring).

1.b Il n'y a ni perte, ni duplication de données dans le composant : chaque signal E déclenche l'émission en un temps fini d'un seul signal S.

1.c Il n'y a pas de déséquencement des données. Cette hypothèse est une conséquence directe de l'hypothèse 1.a.

Nous verrons dans les parties 5.2, 5.3 et 5.5 que, dans sa plus simple expression, la modélisation d'un réseau peut être réalisée avec un seul composant élémentaire. La partie 5.4 nous donnera l'occasion d'illustrer pourquoi et comment le composant élémentaire peut être utilisé pour des réseaux plus complexes. Par exemple, par raffinement et/ou composition du composant élémentaire, il est possible de construire des modèles de réseaux constitués de sous-réseaux utilisant des protocoles MAC (*Medium Access Control*) différents, des mécanismes de conversion de protocoles, des opérations de segmentation/assemblage, etc.

La QoS offerte par le composant élémentaire spécifie des contraintes temporelles sur la réception des données. Nous nous basons sur les équations de QoS définies dans la partie 4.3. Ces dernières offrent la possibilité à l'utilisateur de spécifier un grand nombre de contraintes temporelles. Toutefois, en première approche, nous nous restreignons à un jeu constitué des équations suivantes :

(e1) $\forall\ n,r\ :\ \epsilon_1 \leq \tau(S,n+r) - \tau(S,n) \leq \epsilon_2$

(e2) $\forall\ n,r\ :\ \epsilon_1 \leq \tau(S,n*r) - \tau(S,n) \leq \epsilon_2$

(e3) $\forall\ n\ :\ \tau(S,n) = n * p'$

Cette équation utilise une horloge $H'_{p'}$ dont la période est p' (tel que p' est multiple de p).

(e4) $\forall\ n\ :\ \epsilon_1 \leq \tau(S,n) - \tau(H'_{p'},n) \leq \epsilon_2$. Cette dernière est une généralisation de l'équation (e3) (l'équation (e3) est un cas particulier avec $\epsilon_2 - \epsilon_1 = 0$).

La question qui nous préoccupe maintenant est de savoir quelles doivent être les équations de QoS manipulées par la station émettrice (QoS requise par le réseau) si l'on souhaite que la QoS offerte par le réseau soit respectée sur la station réceptrice.

5.2 Modélisation d'un réseau à délai de communication constant

Afin de faciliter la lecture des exemples suivants, nous commençons par modéliser un élément réseau simple : un réseau qui offre un temps de communication constant pour un message d'une taille donnée. Le modèle de réseau utilisé dans cette partie est constitué d'une instance du composant élémentaire dont le temps de transit est défini comme suit :

Définition 3 (Temps de transit dans le réseau) *Le temps de transit d'une donnée dans le réseau est de λ unités de temps, c'est-à-dire :*

$$\forall\ n\ :\ \tau(S,n) - \tau(E,n) = \lambda$$

Le comportement de ce réseau est en fait celui d'un composant de retard fixe avec $l = \lambda$ (cf. partie 4.4.1). Nous étudions successivement les quatre équations que nous avons précédemment choisies.

5.2.1 Equation (e1)

Pour l'équation de QoS offerte :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(S,n+r) - \tau(S,n) \leq \epsilon_2$$

La QoS requise est :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(E,n+r) - \tau(E,n) \leq \epsilon_2$$

5.2.2 Equation (e2)

Pour l'équation de QoS offerte :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(S,n*r) - \tau(S,n) \leq \epsilon_2$$

La QoS requise est :

$$\forall\ n,r\ :\ \epsilon_1 \leq \tau(E,n*r) - \tau(E,n) \leq \epsilon_2$$

5.2.3 Equation (e3)

Pour la QoS offerte :

$$\forall\ n\ :\ \tau(S,n) = n * p'$$

La QoS requise est :

$$\forall\ n\ :\ \tau(E,n) = n * p' - \lambda$$

(cf. démonstration A.2.1).

5.2.4 Equation (e4)

Pour la QoS offerte :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(S,n) - \tau(H'_{p'},n) \leq \epsilon_2$$

La QoS requise est :

$$\forall\ n\ :\ \epsilon_1 - \lambda \leq \tau(E,n) - \tau(H'_{p'},n) \leq \epsilon_2 - \lambda$$

5.3 Modélisation de réseaux synchrones : exemple du protocole à jeton temporisé

Nous regardons maintenant un exemple de réseau synchrone. Comme pour le réseau à délai de communication constant, nous cherchons à déterminer la QoS qui doit être fournie par la station émettrice afin de respecter sur la station réceptrice la QoS offerte par le réseau. Par la suite, nous donnons une illustration avec le réseau FDDI (*Fiber Distributed Data Interface*).

5.3.1 Modèle d'un réseau synchrone

Cette fois encore, notre modèle de réseau consiste en un unique composant élémentaire. Toutefois, son temps de transit est maintenant défini par :

Définition 4 (Temps de transit dans le réseau) *Le temps de transit d'une donnée dans le réseau est de*

$$\forall\ n\ :\ \tau(S,n) - \tau(E,n) \leq \lambda$$

Ou en d'autres termes, nous regardons les réseaux dont le temps de traversée est borné par λ. La définition d'un réseau synchrone ne nécessite pas que le délai minimum soit connu[1]. Toutefois, nous supposerons dans cette partie, qu'un tel délai est connu, et que

[1] Physiquement, ce délai existe toujours.

sa valeur est de α unités de temps. Le temps de traversée d'une donnée est donc compris dans l'intervalle $[\alpha, \lambda]$. Finalement, un réseau synchrone est en fait un composant de retard variable.

5.3.1.1 Equation (e1)

Pour la QoS offerte :

$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(S, n+r) - \tau(S, n) \leq \epsilon_2$$

La QoS requise est :

$$\forall\ n, r\ :\ \epsilon_1 - \lambda + \alpha \leq \tau(E, n+r) - \tau(E, n) \leq \epsilon_2 + \lambda - \alpha$$

5.3.1.2 Equation (e2)

Pour la QoS offerte :

$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(S, n*r) - \tau(S, n) \leq \epsilon_2$$

La QoS requise est :

$$\forall\ n, r\ :\ \epsilon_1 - \lambda + \alpha \leq \tau(E, n*r) - \tau(E, n) \leq \epsilon_2 + \lambda - \alpha$$

5.3.1.3 Equation (e3)

Pour la QoS offerte :

$$\forall\ n\ :\ \tau(S, n) = n * p'$$

La QoS requise est :

$$\forall\ n\ :\ n * p' - \lambda \leq \tau(E, n) \leq n * p' - \alpha$$

(cf. démonstration A.2.2).

5.3.1.4 Equation (e4)

Pour la QoS offerte :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(S, n) - \tau(H'_{p'}, n) \leq \epsilon_2$$

La QoS requise est :

$$\forall\ n\ :\ \epsilon_1 - \lambda \leq \tau(E, n) - \tau(H'_{p'}, n) \leq \epsilon_2 - \alpha$$

5.3.2 Exemple d'application à FDDI

A titre d'exemple, nous appliquons les équations précédentes à un protocole bien connu : le protocole du jeton temporisé. Dans un premier temps, nous analysons le comportement temporel de la boucle FDDI. Puis, pour illustrer l'intérêt de la composition, nous montrons comment ajouter les interfaces réseaux du site émetteur et du site récepteur à la spécification. Finalement, nous discutons des extensions possibles de celle-ci.

5.3.2.1 La boucle FDDI et les interfaces réseaux

La couche MAC de FDDI utilise un mécanisme de jeton temporisé pour l'allocation du médium. Dans ce protocole, on cherche à faire cohabiter deux trafics différents : un trafic temps réel auquel la couche MAC garantit un délai de communication borné (dit trafic synchrone) et un trafic sans contrainte temporelle (dit trafic asynchrone).

Le protocole de jeton temporisé est utilisé sur un réseau de stations organisé en boucle. Sur celui-ci circule un jeton qui, lorsqu'il est reçu par une station, lui permet d'émettre du trafic. Un temps de rotation cible du jeton, le TTRT (*Target Token Rotation Time*) est déterminé lors du démarrage du réseau. Plus le TTRT est faible, plus la borne sur le temps de communication d'un message sera faible. Ainsi, plus le TTRT est petit et plus le réseau sera adapté au trafic temps réel (malheureusement, au détriment de son efficacité). L'allocation de la bande passante est réalisée de façons différentes pour le trafic synchrone et le trafic asynchrone. La bande passante du trafic synchrone est allouée une fois le TTRT connu. La bande passante synchrone d'une station i, que nous noterons SA_i, est en fait un pourcentage du $TTRT$. Lorsque le jeton arrive sur la station i, celle-ci est autorisée à émettre son trafic synchrone durant une période de $SA_i * TTRT$ unités de temps. L'émission du trafic synchrone est donc garantie une fois le SA_i réservé. Nous n'aborderons pas ici les problèmes d'allocation de la bande synchrone [TOK 92, ZHE 95]. Pour le trafic asynchrone, chaque station utilise un couple de chronomètres pour déterminer si le jeton est en retard ou en avance par rapport au $TTRT$. Une station ne peut émettre de trafic asynchrone que si le jeton est en avance. Il n'y a donc aucune garantie, pour une station donnée, de pouvoir émettre à chaque tour un volume fixe de trafic asynchrone. Notons que la norme FDDI stipule que le $TTRT$ doit permettre, au minimum, l'émission de la somme des trafics synchrones de chaque station additionnée de l'équivalent d'une trame asynchrone de taille maximale (soit 4500 octets).

Comme auparavant, nous modélisons le réseau par un système réactif synchrone (cf. figure 5.2). Chaque interface réseau est décrite par un module recevant et émettant les signaux suivants :

- $HORLOGE$. Le signal modélise l'horloge du système.
- $SORTIE\_MESSAGE$ et $ENTREE\_MESSAGE$ qui correspondent à la réception et à l'émission d'un message synchrone. Le message est segmenté en fonction de la bande passante allouée à l'interface réseau. Un message synchrone nécessite donc éventuellement plusieurs passages de jeton si sa taille dépasse SA_i. On suppose que les messages ont tous la même taille.

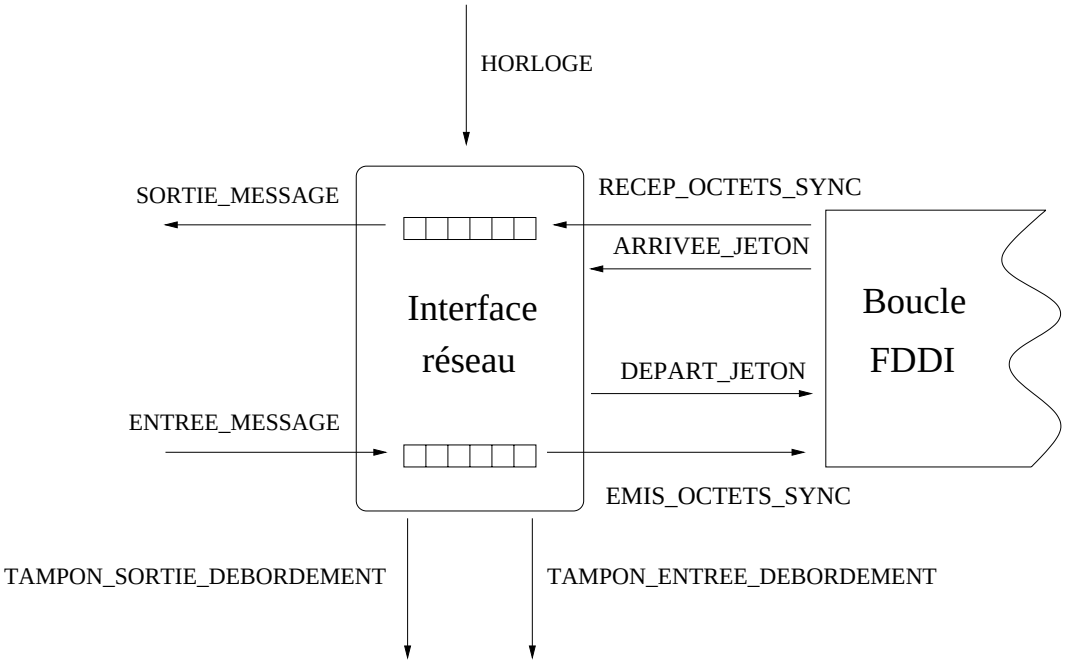


FIG. 5.2 – *Modélisation d'une interface réseau*

- $ARRIVEE\_JETON$ (respectivement $DEPART\_JETON$) dont l'occurrence signale l'arrivée (respectivement le départ) du jeton de l'interface réseau.
- $RECEP\_OCTETS\_SYNC$ et $EMIS\_OCTETS\_SYNC$ qui modélisent la réception et l'émission d'octets de messages synchrones.
- Enfin, les signaux $TAMPON\_ENTREE\_DEBORDEMENT$ et $TAMPON\_SORTIE\_DEBORDEMENT$. Chaque interface réseau contient deux tampons qui mémorisent les messages synchrones reçus et à émettre. Comme pour le composant compensateur de gigue de la partie 4.4.3, ces deux signaux permettent de détecter leur débordement. Nous supposerons, par la suite, que ces tampons sont gérés avec une politique FIFO (*First In, First Out*).

Le module Esterel de l'annexe B.6 spécifie le comportement temporel de l'interface réseau. Dans cette spécification, la gestion du trafic asynchrone est ignorée.

Le protocole à jeton temporisé possède deux propriétés importantes démontrées analytiquement par Johnson [SEV 87]. La première est une propriété essentielle pour le temps réel puisqu'elle énonce que, quelle que soit la charge du réseau, le temps de rotation du jeton (temps d'attente entre deux passages du jeton sur une station donnée) est borné par $2 * TTRT$. La deuxième propriété concerne le temps moyen de rotation du jeton. Ce dernier est borné par $TTRT$. Il peut être inférieur à $TTRT$ si la charge du réseau est faible. Par contre, si la charge du réseau est importante, le temps moyen de rotation du jeton sera

alors plus grand que $TTRT$. Par la suite, ce résultat a été généralisé pour k passages du jeton [AGR 92] : dans ce dernier cas, la borne est de $k * TTRT$ unités de temps.

Il est difficile d'exprimer facilement cette dernière propriété puisque notre modèle ne permet pas de spécifier une moyenne. Néanmoins, la première propriété peut s'exprimer de la façon suivante en QL :

$$\forall\, n \;:\; \tau(ARRIVEE\_JETON, n+1) - \tau(ARRIVEE\_JETON, n) \leq 2 * TTRT \tag{5.1}$$

Cette équation est particulièrement intéressante dans notre cas, puisqu'elle nous permet d'appliquer les résultats énoncés dans la partie 5.3.1. En effet, la boucle FDDI peut être modélisée comme un composant élémentaire dont le comportement temporel est celui d'un composant de retard variable (en substituant λ par $2 * TTRT$). De par le fonctionnement du protocole (cf. annexe B.6), l'équation (5.1) implique que le trafic synchrone émis à l'occurrence n du signal $ARRIVEE\_JETON$ est arrivé chez son destinataire avant l'occurrence $n+1$ du signal $ARRIVEE\_JETON$ chez l'émetteur. Ou encore, si l'occurrence n du signal $EMIS\_OCTETS\_SYNC$ constitue l'émission des octets synchrones par l'émetteur à la réception du $n^{ème}$ signal $ARRIVEE\_JETON$, et si l'occurrence n du signal $RECEP\_OCTETS\_SYNC$ modélise chez le récepteur la livraison par le réseau des octets émis, alors on a :

$$\forall\, n \;:\; \tau(RECEP\_OCTETS\_SYNC, n) - \tau(EMIS\_OCTETS\_SYNC, n) \leq 2 * TTRT \tag{5.2}$$

5.3.2.2 Exemple de composition : le réseau FDDI complet

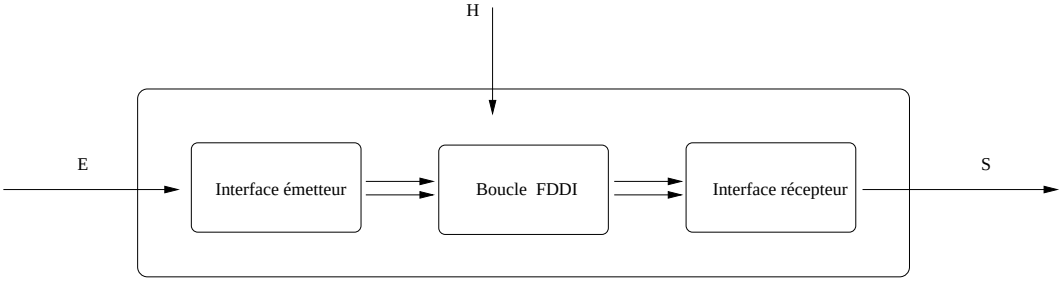


FIG. 5.3 – *Une boucle à jeton temporisé*

Si nous souhaitions utiliser la description d'un composant "FDDI" dans POLKA, le modèle précédent ne serait pas satisfaisant. En effet, nous avons vu qu'il était possible de considérer la boucle FDDI comme un composant élémentaire. Mais si l'on appliquait les résultats de la partie 5.3.1, les équations porteraient sur les occurrences des événements $RECEP\_OCTETS\_SYNC$ et $EMIS\_OCTETS\_SYNC$. De ce fait, les équations ne tiennent pas compte du comportement des interfaces réseaux émettrices et réceptrices. L'utilisateur ne peut donc toujours pas spécifier de contrainte de QoS sur l'émission (signal

$ENTREE\_MESSAGE$ de la figure 5.2) et la réception (signal $SORTIE\_MESSAGE$) de messages synchrones. Le modèle doit donc être complété.

Nous donnons maintenant un exemple de composition. Nous considérons le composant de la figure 5.3. Celui-ci est constitué de trois sous-composants : le composant central modélise la boucle FDDI et les deux autres, les interfaces réseaux. Le temps physique est toujours représenté par le signal d'entrée H. Enfin, les signaux E et S correspondent à l'émission et à la livraison d'un message synchrone (signaux équivalents aux signaux $ENTREE\_MESSAGE$ et $SORTIE\_MESSAGE$ de la figure 5.2). Nous modélisons l'interface réseau réceptrice comme un composant de retard fixe qui induit un retard de β unités de temps. β est le temps moyen nécessaire à la station réceptrice, lorsqu'elle reçoit les derniers octets d'un message synchrone, pour :

1. Copier ledit message dans l'espace d'adressage du noyau.
2. Avertir le système d'exploitation grâce à une interruption qu'un message peut être délivré (signal $SORTIE\_MESSAGE$).

De la même façon, nous allons modéliser l'interface réseau émetteur par un composant de retard fixe qui induit un retard moyen de ϕ unités de temps. Ce retard comprend le temps nécessaire à l'interface pour consommer et émettre des octets de son tampon (signal $EMIS\_OCTETS\_SYNC$).

A titre d'exemple, nous allons montrer comment propager une contrainte de QoS utilisateur sur les trois sous-composants de la figure 5.3. La QoS utilisateur étudiée est de la forme (équation (e1)) :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(S, n+1) - \tau(S, n) \leq \epsilon_2$$

Nous appliquons le résultat de la partie 5.3.1 à notre modèle de réseau FDDI. Nous cherchons donc l'équation de QoS sur la station émettrice qui contraint la réception des signaux E. L'équation constitue la QoS requise par le réseau FDDI permettant de garantir la QoS utilisateur sur la station réceptrice. Pour ce faire, il faut répercuter le comportement temporel des trois composants de la figure 5.3.

Par la suite, nous utilisons les notations suivantes :

- s est la taille (fixe) des messages synchrones en octets (et donne donc le volume de données transportées par les signaux E et S).
- SA_e est la bande passante synchrone allouée à la station émettrice (en pourcentage du $TTRT$).

Nous commençons par déterminer le nombre de rotations du jeton qui sera nécessaire pour transmettre chez le destinataire un message synchrone. Soit r ce nombre, il faudra alors

$$r = \left\lceil \frac{s}{SA_e * TTRT * 100} \right\rceil$$

rotations du jeton pour écouler le trafic généré par un message synchrone[2].

Maintenant, nous réalisons la propagation des contraintes de QoS. Nous procédons par étapes, en regardant d'abord le comportement de l'interface récepteur. Celle-ci a été modélisée par un composant de retard fixe. D'après les résultats de la partie 4.4.1.2, on en déduit que la QoS requise par l'interface récepteur est :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(RECEP\_OCTETS\_SYNC, n+r) - \tau(RECEP\_OCTETS\_SYNC, n) \leq \epsilon_2$$

Le paramètre β n'apparaît pas dans cette équation puisqu'un retard fixe ne joue pas sur les délais entre deux émissions successives. Nous reportons maintenant le comportement temporel de la boucle FDDI. Celle-ci induit à chaque message un retard variable compris entre α et $2 * TTRT$ unités de temps. D'après les résultats du paragraphe 4.4.2.1, la QoS requise par la boucle FDDI est donc :

$$\forall\ n\ :\ \epsilon_1 - 2*TTRT + \alpha \leq \tau(EMIS\_OCTETS\_SYNC, n+r) - \tau(EMIS\_OCTETS\_SYNC, n) \leq \epsilon_2 + 2*TTRT - \alpha$$

Finalement, la QoS requise pour garantir la QoS utilisateur est la suivante :

$$\forall\ n\ :\ \epsilon_1 - 2 * TTRT + \alpha \leq \tau(E, n+1) - \tau(E, n) \leq \epsilon_2 + 2 * TTRT - \alpha$$

Sur le site émetteur, si le système d'exploitation livre à l'interface réseau les messages synchrones en respectant l'équation ci-dessus, alors la QoS utilisateur sera respectée. Notons que le paramètre ϕ n'apparaît pas non plus dans cette équation, et ce pour les mêmes raisons que le paramètre β.

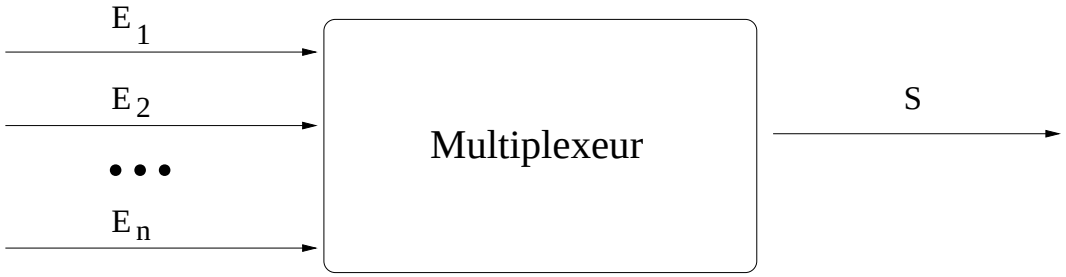


FIG. 5.4 – *Le composant multiplexeur*

5.3.2.3 Conclusion

Dans les parties précédentes, nous avons montré que le composant élémentaire pouvait être combiné à d'autres afin de modéliser de façon plus réaliste le comportement d'un périphérique réseau. La composition séquentielle utilisée pour modéliser un réseau FDDI nous a permis de spécifier des contraintes de QoS sur l'émission et/ou la réception de trames FDDI. Cette spécification est un excellent exemple de ce que nous espérons de notre

[2] Une solution plus réaliste du calcul de r devrait aussi tenir compte des messages déjà présents dans le tampon ainsi que des octets constituant les en-têtes des trames synchrones de FDDI [FEN 96].

modèle : une spécification modulaire des éléments du système afin d'offrir une réutilisabilité importante des spécifications. Ainsi, dans notre exemple, nous avons modélisé tous les composants du réseau FDDI avec des composants de notre bibliothèque.

Cette modélisation reste toutefois très simpliste et elle pourrait être affinée par les éléments suivants :

- Permettre le partage du média FDDI par plusieurs flots (plusieurs applications par exemple). En effet, notre spécification FDDI suggère l'utilisation d'un unique flot (par exemple, dans le calcul de r, nous n'avons pas intégré la présence de données déposées dans le tampon par d'autres flots). C'est généralement le cas dans bon nombre de travaux qui traitent de l'utilisation de FDDI pour des applications temps réel. Toutefois, cette contrainte est peu réaliste. Il existe néanmoins des solutions qui ont été proposées [FEN 96]. Celles-ci consistent à utiliser un serveur de bande passante synchrone. Le serveur reçoit en entrée plusieurs flots avec des contraintes éventuellement différentes. Lorsqu'un nouveau flot arrive, le serveur vérifie si, étant donné la bande synchrone allouée et les flots déjà présents, le nouveau flot peut être accepté. En dehors du contrôle d'admission, le serveur multiplexe tous les flots entrants en un seul flot sortant vers l'interface FDDI. En fait, le multiplexage consiste à ordonnancer les différentes trames afin de respecter leurs contraintes temporelles. La technique du serveur de bande synchrone peut être spécifiée sous la forme d'un composant multiplexeur (cf. figure 5.4) connecté au signal E du composant FDDI.
- Offrir des services mieux adaptés aux applications multimédias, et en particulier offrir aux puits l'impression que les temps de communication sont constants. Il suffit pour cela d'ajouter un compensateur de gigue au signal S du composant FDDI. La taille du tampon du compensateur est ajustée en fonction de la qualité de service offerte par le réseau (cf. section 4.4.3).

Il est clair que construire un modèle aussi complexe, en prenant tout ces paramètres en même temps, reste un travail difficile à réaliser. Là aussi la spécification modulaire est une approche intéressante puisqu'elle permet d'appréhender plus facilement ces difficultés en modélisant séparément les différents composants du système.

5.4 Modélisation de réseaux isochrones : le cas d'ATM

Nous étudions maintenant le cas d'un réseau isochrone. Nous illustrons les résultats par le réseau ATM.

5.4.1 Modèle pour un réseau isochrone

Notre étude des composants réseaux s'achève par ceux dont le temps de transit est défini comme suit :

Définition 5 (Temps de transit dans le réseau) *Le temps de transit d'une donnée dans le réseau respecte la propriété suivante :*

$$\forall\ n\ :\ \lambda_1 \leq \tau(S,n) - \tau(E,n) \leq \lambda_2$$

Ou en d'autres termes, nous regardons les réseaux qui garantissent le respect d'une gigue sur les temps de communication. Dans le cas de la définition (5), la gigue garantie est de $\lambda_2 - \lambda_1$ unités de temps.

Pour le cas des réseaux isochrones, les équations de la partie 5.3 restent valides si l'on considère la substitution suivante :

$$\begin{cases} \alpha = \lambda_1 \\ \lambda = \lambda_2 \end{cases}$$

Il existe toutefois une importante différence entre le cas synchrone et le cas isochrone : $(\lambda - \alpha)$ est généralement négligeable pour l'application dans le cas des réseaux isochrones alors qu'elle ne l'est pas dans le cas synchrone. En effet, il est courant, dans les réseaux isochrones, que la gigue fasse partie des paramètres de QoS négociés lors de la connexion de l'application, alors que la gigue subie par l'application dans le cas d'un réseau synchrone n'est pas négociable mais constitue plutôt une caractéristique temporelle du réseau.

5.4.2 Application au réseau ATM

Jusqu'à présent, nous avons toujours modélisé les périphériques réseaux avec une seule instance du composant élémentaire. Toutefois, cette solution simple n'est pas applicable dans certains cas, et en particulier pour les réseaux à mémoire et les réseaux maillés. Le cas d'ATM est intéressant car sa modélisation nous oblige à nous interroger sur le niveau de modélisation qu'il est souhaitable de choisir.

Une première solution consiste, comme pour FDDI, à modéliser les différents composants du réseau : les extrémités, les liens, les routeurs, etc. Il est alors nécessaire de modéliser la topologie du réseau et le comportement temporel de chacun de ses composants. Le composant de la figure 4.12 (page 52) peut ainsi modéliser un lien ATM où les composants m_1 et m_r sont des routeurs ATM et où les composants m_2 à m_{r-1} modélisent la propagation des cellules sur le lien. Cette solution est particulièrement adaptée à ATM. En effet, on considère généralement que la gigue dans un réseau ATM est créée par les extrémités et les nœuds du réseau alors que la gigue sur la propagation des cellules est négligeable. La gigue est essentiellement provoquée par l'ordonnancement des cellules et la gestion des tampons sur les routeurs [FOR 96]. Ceci se traduit aisément dans notre spécification en positionnant les paramètres des composants de la façon suivante :

- $\forall\ i\ \in \{2, r-1\}\ :\ \lambda_i = \alpha_i = p$ où p est le temps de propagation que le composant modélise.
- Les paramètres λ_1 et λ_r sont initialisés avec les temps de réponse des cellules sur les routeurs [ZHA 93].

Notons que l'utilisation d'un même composant de base pour modéliser tous les composants du réseau offre un avantage intéressant : le calcul des équations de QoS est homogène, et donc relativement peu coûteux. A partir de cet instant, il est possible d'envisager que le calcul dynamique des équations de QoS soit fait lors de la phase de connexion, par exemple (en fait, une fois que le chemin de la source vers le puits est déterminé). Le raisonnement précédent poussé à l'extrême est la solution adoptée par Anderson dans le Meta-scheduler [AND 93]. Dans cette plate-forme, tous les composants logiciels et matériels intervenants dans le support d'un flot de données continues sont modélisés par un composant unique dont les contraintes temporelles sont déduites du trafic utilisé (trafic LBAP, *Linear Bounded Arrival Process*).

Modéliser un réseau ATM par une séquence de composants de retard variable semble donc être une solution séduisante à notre problème. Toutefois, sa complexité est importante. En effet, pour qu'elle soit efficace, beaucoup d'éléments doivent être ajoutés à la spécification. Prenons l'exemple d'une extrémité : le modèle proposé ci-dessus ne décrit pas les mécanismes de lissage, le multiplexage des différents flots sur la couche physique et la gigue ainsi créée, les segmentations et assemblages effectués par les couches d'adaptation, etc. En fait, la construction d'une telle spécification nécessiterait un travail qui dépasse de loin le cadre de cette thèse. Une autre solution consiste à considérer une spécification de plus haut niveau, en modélisant le comportement des services offerts par ATM. C'est cette approche qui est décrite dans la suite où nous modélisons le service CBR (*Constant Bit Rate*).

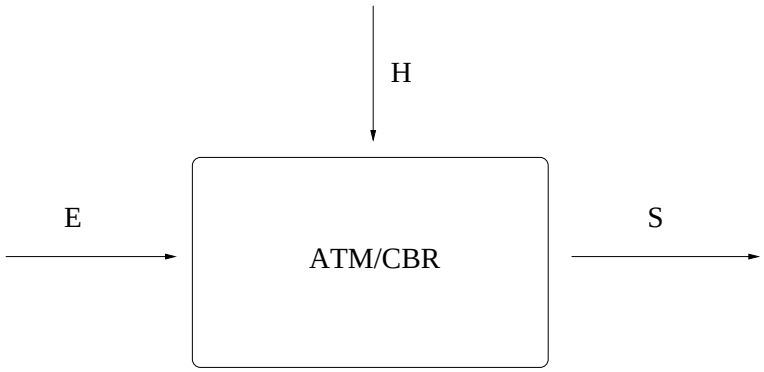


FIG. 5.5 – *ATM: le service CBR*

Le service CBR est destiné à la communication de données à débit fixe et comportant des contraintes temporelles fortes (tel que le transfert de données téléphoniques par exemple). Ce service est défini par les paramètres de description du trafic et de QoS suivants :

1. Le PCR (*Peak Cell Rate*) qui traduit le débit maximal de la source (en nombre de cellules par seconde).

2. Le ppCDV (*peak to peak Cell Delay Variation*) qui détermine la gigue maximale

acceptable par l'application (variation maximale du délai inter-arrivée des cellules chez le récepteur).

3. Le maxCTD (*maximum Cell Transfer Delay*) qui traduit le temps maximal de communication. Ce temps est composé d'une partie fixe induite par le temps de propagation sur le support physique entre autre, ainsi qu'une partie variable. Le temps de communication fixe est déterminé par $maxCTD - ppCDV$. Notons que la valeur du ppCDV est négociée pour une probabilité de perte des cellules données (paramètre p sur la figure 5.6).

4. Et enfin, le CLR (*Cell Loss Ratio*) qui définit le taux de perte des cellules. Nous n'utiliserons pas ce dernier paramètre puisque notre modèle ne peut pas spécifier des contraintes de fiabilité.

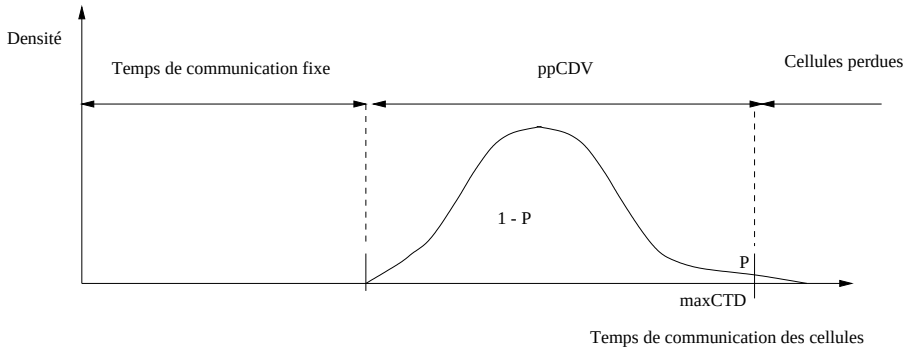


FIG. 5.6 – *Description de QoS et de trafic pour le service CBR d'ATM*

Nous modélisons donc une connexion CBR par un composant constitué d'un signal de sortie et d'un signal d'entrée (cf. figure 5.5). Chaque occurrence d'un signal modélise l'arrivée ou la sortie d'une cellule ATM du réseau. Nous supposons que $QE = QS = 48$ octets (quantité en octets de la charge utile d'une cellule).

Nous savons que le service CBR nous garantit la propriété suivante :

$$\forall\ n\ :\ maxCTD - ppCDV \leq \tau(S,n) - \tau(E,n) \leq maxCTD$$

Le rythme d'émission des cellules par la source est de PCR cellules par unité de temps, ce qui peut se modéliser par la QoS requise suivante :

$$\forall\ n\ :\ \tau(H'_{p'},n) = n * \frac{1}{PCR}$$

où $H'_{p'}$ est une horloge de période $p' = 1/PCR$. D'après les résultats sur les composants de retard variable de la partie 4.4.2.2, on en déduit que la QoS offerte par le réseau est :

$$\forall\ n\ :\ maxCTD - ppCDV \leq \tau(S,n) - \tau(H'_{p'},n) \leq maxCTD$$

5.5 Modélisation de réseaux asynchrones

Nous terminons le chapitre par le cas des réseaux asynchrones. Contrairement aux réseaux modélisés dans les parties précédentes, il n'est pas possible de connaître a priori le comportement temporel du composant.

L'exemple le plus connu de réseau asynchrone est Internet. Pour ce réseau, de nombreuses solutions proposent d'adapter le fonctionnement du système ou des applications à l'évolution du comportement temporel du réseau [BOL 94, FRY 96, BUS 96]. Pour ce faire, des outils de supervision sont utilisés. Le plus connu est celui proposée par Schulzerinne [BUS 96], et qui est aujourd'hui standardisé par l'IETF (*Internet Engineering Task Force*) au travers du protocole RTP (*Real Time Protocol*) [SCH 96]. C'est le protocole de contrôle de RTP, RTCP (*Real Time Control Protocol*), qui offre ces services de supervision. Grâce à ce dernier, il est possible à une application d'obtenir une estimation du délai de communication de bout en bout, de la gigue inter-arrivée des paquets sur la station réceptrice, du taux de pertes, etc. Nous verrons dans le chapitre 7, que la plate-forme d'exécution de POLKA utilise ce protocole pour les services de supervision qu'elle offre à ses applications.

Pour modéliser des réseaux asynchrones, nous proposons des mécanismes de supervision au travers des composants de rétroaction que nous avions définis dans la partie 4.2. Un réseau asynchrone peut donc être modélisé par un composant de retard fixe ou de retard variable. Le choix du composant dépend des informations qui sont collectées par le service de supervision. Si le système estime une borne sur les délais de communication, les équations décrites dans la partie 5.3 sont applicables pour déterminer les équations de QoS requise par la station émettrice. Si l'évaluation d'une gigue est maintenue, ce sont les équations de la partie 5.4 qui peuvent être exploitées. Enfin, une solution plus simple consiste à obtenir une approximation du délai de communication. Dans ce dernier cas, les équations utilisables sont celles que nous avons définies pour un réseau dont le temps de communication est constant (cf. partie 5.2).

5.6 Conclusion

Dans ce chapitre, nous avons appliqué notre modèle à quatre éléments réseaux ayant des caractéristiques temporelles différentes. Nous avons successivement proposé une spécification pour des réseaux à délai constant, puis pour des réseaux synchrones, isochrones et finalement asynchrones. Nous avons montré qu'il était possible de spécifier des services simples. Toutefois, de part le caractère déterministe de notre modèle de spécification, il existe des cas importants où notre modèle n'est pas applicable. C'est notamment le cas des services de communication à débits variables (exemple : service VBR-RT sur ATM) où des propriétés temporelles basées sur des espérances mathématiques (exemple : temps moyen de rotation du jeton dans une boucle FDDI). Comme nous le verrons dans les chapitres 9 et 10, cet aspect reste un problème non traité dans cette thèse.

Chapitre 6

Algorithmes d'ordonnancement et modèles de tâches

Dans le chapitre 4, nous avons proposé un modèle de spécification pour représenter le comportement temporel des différents composants d'un système. Dans ce chapitre, nous proposons des algorithmes d'ordonnancement qui exploitent ce modèle de façon automatique afin d'allouer les ressources processeurs conformément aux contraintes temporelles spécifiées par le développeur. Nous en profitons pour montrer comment le modèle du chapitre 4 peut être rapproché des concepts rencontrés dans la littérature sur l'ordonnancement d'applications temps réel ; ce rapprochement nous permet de comparer nos propositions aux résultats classiques de l'ordonnancement de systèmes temps réel [RIV 98, LEB 98].

Nous commençons tout d'abord par définir le modèle de tâches manipulé par les ordonnanceurs de POLKA. Puis, nous décrivons un algorithme capable d'ordonnancer un ensemble de tâches dans un environnement mono-processeur. Enfin, nous proposons une extension de celui-ci pour un environnement réparti.

6.1 Le modèle de tâches de POLKA

Nous définissons ici la notion de tâche utilisée dans POLKA de façon graduelle, en utilisant à chaque fois que cela sera possible, les termes, notations et abstractions généralement utilisés dans la littérature sur l'ordonnancement temps réel (cf. paragraphe 2.2). Dans un premier temps, une tâche POLKA peut être définie de la sorte :

Définition 6 (Tâche) *Une tâche* POLKA *est une suite d'instructions séquentielles exécutées sur un processeur. Une tâche peut être répétitive : dans ce cas, elle fait l'objet de plusieurs activations successives.*

Cette définition est insuffisante. Tout d'abord elle ne précise pas ce que constitue une tâche vis-à-vis du modèle de spécification manipulé par le développeur. En effet, dans POLKA, le développeur décrit une application en termes d'objets, de threads "à la Clouds" et de spécifications de QoS (cf. chapitres 3 et 4). L'abstraction de tâche n'est donc pas une

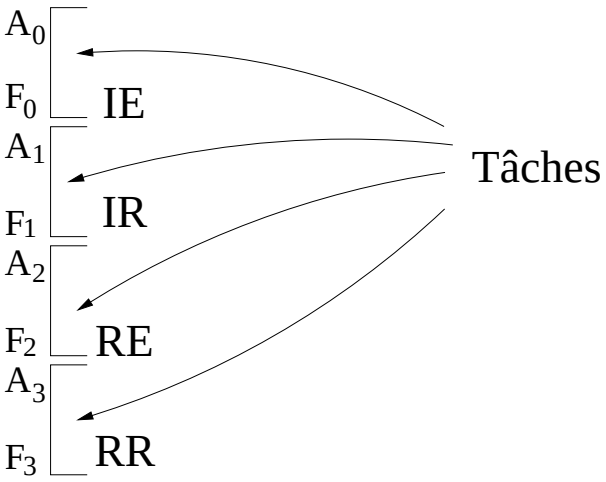


FIG. 6.1 – *Exemple de découpage d'une invocation d'objet en quatre tâches*

abstraction visible par le développeur d'applications ; elle est uniquement manipulée par l'ordonnanceur.

Il existe toutefois une correspondance directe entre le modèle utilisé par le développeur d'application et le modèle de tâches. Les règles définissant cette correspondance sont triviales : elles définissent ce que sont le début et la fin d'une tâche. Regardons la figure 6.1 qui représente une invocation de méthode synchrone. L'invocation de méthode est découpée en quatre tâches distinctes (T_0, T_1, T_2 et T_3). Pour obtenir ce découpage du code de l'application (décrite par un thread) en tâches, nous utilisons les événements observés lors des interactions entre objets (cf. partie 4.3). Les événements en question (qui sont IE pour l'émission de l'invocation, IR pour la réception de l'invocation chez le serveur, RE pour l'émission de la réponse chez le serveur et RR pour la réception de la réponse chez le client) isolent les tâches les unes des autres. Nous pouvons préciser un peu plus ce que contient la notion de tâche dans POLKA par cette nouvelle définition :

Définition 7 (Tâche) *Une tâche* POLKA *est une suite d'instructions séquentielles exécutées sur un processeur. Une tâche peut être répétitive : dans ce cas, elle fait l'objet de plusieurs activations successives. Une tâche T est délimitée par les deux événements suivants :*

- *A_T qui désigne l'événement modélisant l'arrivée de T dans le système.*
- *F_T qui désigne l'événement de fin d'exécution de T. Un événement F_T correspond généralement à l'un des événements IE, IR, RE et RR observés lors des interactions entre objets du modèle de spécification.*

Nous désignons par l'expression $\tau(A_T, n)$ (respectivement $\tau(F_T, n)$)[1] la date de la $n^{ème}$ activation de la tâche T (respectivement la date de terminaison de la $n^{ème}$ activation).

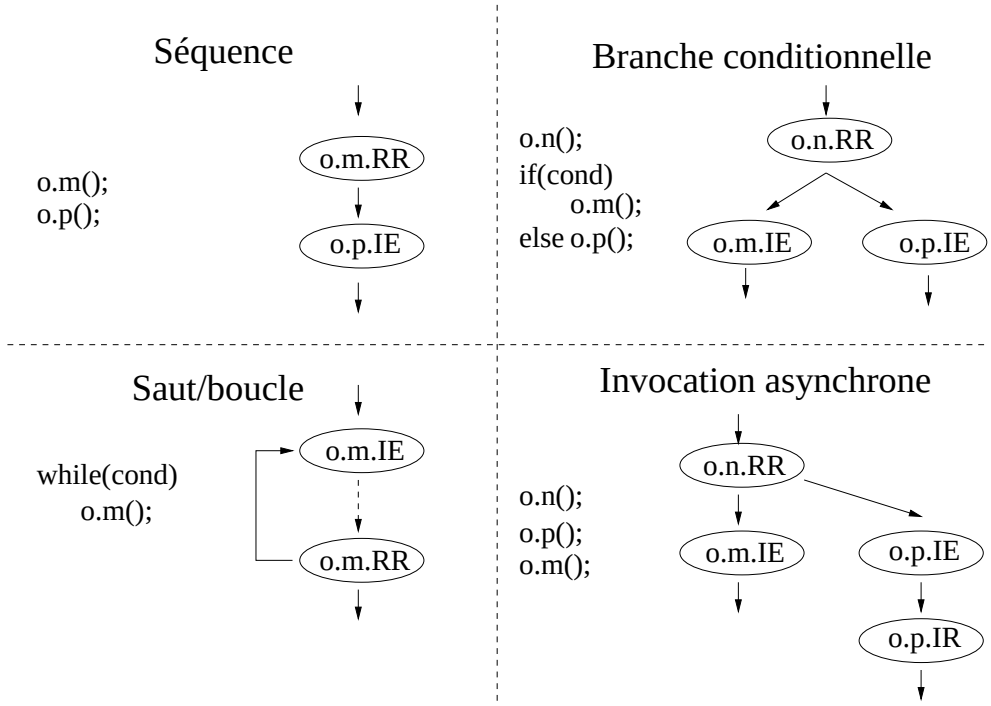


FIG. 6.2 – *Construction des graphes de tâches*

Bien sûr, des structures de threads plus complexes peuvent être observées dans une application. La figure 6.2 illustre les principaux cas de figure qui peuvent exister (rappelons que seules les invocations de méthodes sur des objets génèrent des événements observables dans POLKA). De gauche à droite, la figure 6.2 donne le graphe de tâches correspondant à une séquence d'invocations, à une branche conditionnelle, à un saut (la boucle est un cas particulier de saut) et enfin à une invocation asynchrone. Le cas de l'invocation synchrone est un cas particulier de la séquence. Les figures 6.1 et 6.2 nous permettent d'introduire une seconde abstraction manipulée par l'ordonnanceur : l'abstraction de graphe de tâches. Un graphe de tâches modélise les contraintes de précédence existant entre les tâches[2] :

Définition 8 (Graphe de tâches) *A chaque thread i du modèle de spécification est associé un graphe de tâches $\mathcal{G}_i$ défini par le couple $\mathcal{G}_i =< \mathcal{T}_i, \mathcal{D}_i >$, où $\mathcal{T}_i$ est l'ensemble des tâches et $\mathcal{D}_i$ est l'ensemble des arcs du graphe de tâches. Un graphe de tâches est orienté et connexe. Chaque arc de $\mathcal{D}_i$ modélise une contrainte de précédence entre deux tâches*

[1] Comme dans le chapitre 4, τ désigne l'opérateur de datation d'un événement.

[2] Il est remarquable que les graphes pour la branche conditionnelle et pour l'invocation asynchrone sont identiques. En effet, le graphe de tâches ne modélise pas les flots d'exécution parallèles mais uniquement les précédences entre les tâches.

(relation de précédence que nous noterons $\prec$). Ainsi l'expression $i \prec j$ stipule que la tâche j ne peut démarrer son exécution avant la fin de la tâche i ; en d'autres termes :

$$i \prec j \rightarrow \forall\ n\ :\ \tau(A_j, n) \geq \tau(F_i, n)$$

Dans la suite de ce chapitre, nous désignerons l'ensemble des graphes de tâches du système par

$$\gamma = \bigcup_{i=1}^{l} \mathcal{G}_i$$

où l constitue le nombre de threads du système.

Nous avons décrit ce qu'est une tâche dans le contexte de POLKA et à quelles contraintes de précédence celles-ci peuvent être soumises. Il nous reste maintenant à compléter notre définition en ajoutant les paramètres qui définissent les contraintes temporelles associées aux tâches :

Définition 9 (Tâche) *Une tâche* POLKA *T est une suite d'instructions séquentielles exécutées sur un processeur. Une tâche peut être répétitive : dans ce cas, elle fait l'objet de plusieurs activations successives. Chaque activation est délimitée par son arrivée (notée A_T) et sa fin (noté F_T). Les contraintes temporelles d'un tâche* POLKA *T sont caractérisées par une **conjonction** d'équations de QoS que nous noterons ϕ_T.*

Par la suite, nous désignerons l'ensemble des équations de QoS du système par :

$$\phi = \bigwedge_{\forall\ i\ \in\ \gamma} \phi_i$$

A partir de la définition ci-dessus, nous classerons les tâches en deux groupes : les tâches dites "pertinentes" et les tâches "non pertinentes" :

Définition 10 (Tâche pertinente) *Un tâche T est pertinente si $\phi_T \neq \emptyset$. Elle est non pertinente dans le cas contraire.*

Une tâche pertinente est donc une tâche à laquelle est associée une ou plusieurs contraintes temporelles. A titre d'exemple, une tâche périodique est une tâche pertinente alors que la tâche associée à une compilation ne l'est généralement pas (sauf si l'utilisateur spécifie, par exemple, une borne sur son temps de réponse). Finalement, les définitions données dans ce paragraphe induisent l'existence possible de deux catégories de dépendances entre des tâches T et T' :

- Des dépendances **logiques** matérialisées par l'existence d'un arc entre T et T' dans leur graphe de tâches.
- Des dépendances **temporelles** lorsqu'il existe dans ϕ_T et $\phi_{T'}$ au moins une équation portant sur les deux tâches.

Il est important de noter que le modèle de tâches ne comprend pas le temps d'exécution des tâches. En effet, les ordonnanceurs que nous proposons ne nécessitent pas une connaissance a priori de cette information. Cette particularité est à souligner compte tenu des applications ciblées dans cette thèse (applications dont les besoins en ressources processeurs sont difficiles à évaluer). Il est également remarquable que le modèle de tâches ne comprenne pas les notions de priorité, de périodicité, etc. En fait, le modèle de tâches ne définit pas de façon explicite les contraintes temporelles auxquelles une tâche est soumise : le modèle de tâches doit être capable d'exprimer le plus grand nombre des contraintes rencontrées dans un système multimédia. Pour des raisons de complexité, nous verrons qu'il est toutefois nécessaire d'énoncer quelques limitations sur l'expression des contraintes dans ϕ.

6.2 Expression de modèles de tâches grâce à des équations de QoS

Dans ce paragraphe, nous allons montrer que le modèle de tâches décrit ci-dessus est suffisamment général pour spécifier des modèles de tâches classiques dans les systèmes temps réel, mais aussi des modèles de tâches spécifiques aux applications multimédias. Nous invitons le lecteur à consulter le paragraphe 2.2 pour une description plus détaillée des modèles de tâches que nous allons aborder ici.

6.2.1 Modèles de tâches pour les applications temps réel

Revenons sur le modèle de la tâche périodique [LIU 73]. Une tâche périodique T est définie par le 4-uplet $(\tau(A_T, 1), D_T, C_T, P_T)$ où ${C_T}^3$ constitue le temps d'exécution d'une activation de la tâche, P_T sa période d'activation, D_T son échéance et $\tau(A_T, 1)$ la date de sa première activation. L'échéance D_T est relative aux dates d'activation. Enfin, si $\tau(A_T, k)$ est la date de la $k^{ème}$ activation de la tâche T, alors nous avons la relation de récurrence $\tau(A_T, k) = \tau(A_T, k-1) + P_T$. Définir une tâche périodique T dans notre modèle de tâche consiste à déterminer son graphe de tâches ainsi que l'ensemble ϕ_T. Dans le modèle de Liu et Layland, les tâches sont indépendantes : le graphe de tâches est donc réduit à un seul sommet par tâche. L'ensemble ϕ_T peut être défini de la façon suivante :

$$\forall\ n\ :\ \begin{cases} \tau(A_T, n+1) - \tau(A_T, n) = P_T \\ \tau(F_T, n) - \tau(A_T, n) \leq D_T \end{cases} \tag{6.1}$$

où la première équation définit la loi d'arrivée des différentes activations de la tâche T et la deuxième équation exprime la contrainte d'échéance de la tâche. Notons que cette formulation ne spécifie rien sur la relation existant entre D_T et P_T ; le jeu d'équations ne modélise donc pas uniquement des tâches périodiques à échéance sur activation [LIU 73]

[3] C_T est le plus souvent une borne sur le temps d'exécution de la tâche T.

mais peut éventuellement décrire les tâches isochrones ou périodiques avec gigue de G. Coulson et al. [COU 97].

Nous regardons maintenant le cas de la tâche sporadique [MOK 83]. Une tâche sporadique T, comme pour une tâche périodique, est définie par un 4-uplet $(\tau(A_T,1), D_T, C_T, P_T)$ où D_T est une échéance relative à l'activation, C_T le temps d'exécution pour une activation et $\tau(A_T,1)$ la date de sa première activation. Cette fois-ci, P_T ne constitue plus une période d'activation mais un délai minimal entre les arrivées de deux activations successives. Si $\tau(A_T,k)$ est la date de la $k^{ème}$ activation de la tâche sporadique T, alors nous avons cette fois-ci $\tau(A_T,k) \geq \tau(A_T,k-1) + P_T$. Comme pour la tâche périodique, il n'existe pas de dépendance entre les tâches et le graphe de tâches est réduit à sa plus simple expression : un sommet par tâche. L'ensemble ϕ_T d'une tâche sporadique T peut être constitué de cette façon :

$$\forall\ n\ :\ \begin{cases} \tau(A_T,n+1) - \tau(A_T,n) \geq P_T \\ \tau(F_T,n) - \tau(A_T,n) \leq D_T \end{cases} \tag{6.2}$$

Enfin, nous terminons ce paragraphe par les tâches apériodiques. Ces dernières ne sont pas répétitives. Elles sont définies par le triplet $(\tau(A_T,1), D_T, C_T)$ où $\tau(A_T,1)$ est la date de son activation, D_T son échéance relative à $\tau(A_T,1)$ et C_T son temps d'exécution. Encore une fois, le graphe de tâches comporte un seul sommet par tâche. Une tâche apériodique peut être modélisée par l'équation suivante :

$$\tau(F_T,1) - \tau(A_T,1) \leq D_T$$

6.2.2 Modèles de tâches pour les applications multimédias

Si les modèles de tâches présentés ci-dessus sont bien adaptés aux applications temps réel, ils ne le sont pas forcément aux applications multimédias. Nous avons vu dans le chapitre 2 que dans une application multimédia, on s'intéresse souvent à des contraintes que l'on appelle "contraintes de distance" [BUC 93, STE 95b, SAK 95, HAN 96]. Une contrainte de distance spécifie un délai minimal et/ou maximal entre deux fins de tâche. A titre d'exemple, un utilisateur peut vouloir spécifier un délai entre l'affichage de deux images successives qui soit compris entre un maximum de ϵ_2 unités de temps et un minimum de ϵ_1 unités de temps. Si T est une tâche qui affiche une image à chaque activation, la contrainte ci-dessus peut alors s'exprimer de la façon suivante :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(F_T,n+1) - \tau(F_T,n) \leq \epsilon_2 \tag{6.3}$$

Si une contrainte de distance est difficile à spécifier avec les modèles de tâches définis dans la partie 6.2.1, elle est aisée à écrire en QL (cf. équation (6.3)). Nous décrivons maintenant deux solutions qui ont été proposées dans la littérature pour la spécification

et l'ordonnancement de ce type de contrainte. Nous montrons comment chacune de ces solutions peuvent être exprimées dans notre modèle de QoS.

Si une tâche périodique T est définie par les paramètres $(\tau(A_T, 1), D_T, C_T, P_T)$ tel que $D_T = P_T$, le modèle de tâche périodique spécifie uniquement que l'occurrence $k-1$ de l'événement F_T doit intervenir avant le début de la $k^{ème}$ période. Ceci a pour conséquence de soumettre les occurrences des événements F_T à la contrainte suivante [COT 98] :

$$\forall\ n\ :\ C_T \leq \tau(F_T, n+1) - \tau(F_T, n) \leq 2 * P_T - C_T \tag{6.4}$$

Avec le modèle de tâche périodique, il est donc possible de spécifier implicitement une contrainte de distance maximale en fixant P_T par :

$$P_T = \frac{\epsilon_2 + C_T}{2}$$

où ϵ_2 constitue le délai maximal d'une équation de la forme (6.3). Cette première solution, si elle a pour avantage d'utiliser un modèle de tâches bien connu (tel que celui de Liu et Layland), manque de souplesse. De ce fait, Han et al. [HAN 96] proposent un modèle dont l'activation des tâches n'est plus déterminée par une période mais par la fin d'exécution de l'occurrence précédente, ce qui s'exprime en QL par :

$$\forall\ n\ :\ \begin{cases} \tau(A_T, n+1) - \tau(F_T, n) = 0 \\ \tau(F_T, n+1) - \tau(F_T, n) \leq D_T \end{cases} \tag{6.5}$$

Nous aurons l'occasion d'illustrer ce type de modèle avec une application multimédia dans la partie 8.2.

Regardons maintenant un modèle que nous avions décrit dans le paragraphe 2.2.2 : les cadences d'activation de Jeffay et al. [JEF 92a, JEF 95]. Une cadence d'activation T est définie par le 4-uplet $(\tau(A_T, 1), D_T, P_T, N_T)$ où N_T constitue le nombre d'activations de la tâche dans un intervalle de temps dont la durée est de P_T unités de temps, D_T l'échéance de la tâche relative à ses dates d'activation et $\tau(A_T, 1)$ l'arrivée de la tâche dans le système. Aucune hypothèse de distribution des N_T activations dans l'intervalle de temps n'est donnée. Cette fois-ci, nous utilisons une horloge logique H_{P_T} [4] dont la période est égale à P_T. Les contraintes temporelles d'une cadence d'activation peuvent alors être exprimées par :

$$\forall\ n\ :\ \begin{cases} \forall\ 0 \leq i \leq N_T\ :\ \tau(A_T, n+i) - \tau(H_{P_T}, n) \leq P_T \\ \tau(F_T, n) - \tau(A_T, n) \leq D_T \end{cases}$$

[4] Notons que l'on aurait pu, de la même façon, utiliser une horloge pour d'autres modèles (exemple : tâches périodiques).

6.3 Proposition d'un algorithme d'ordonnancement centralisé

Dans la partie précédente, nous avons décrit un modèle de tâches. Nous avons montré qu'il est suffisamment général pour modéliser les contraintes temporelles des modèles de tâches les plus couramment rencontrés.

Nous décrivons maintenant un ordonnanceur capable d'exploiter ce modèle de tâches dans un environnement mono-processeur. L'ordonnanceur proposé n'exige pas que le concepteur de l'application lui fournisse le temps d'exécution des tâches. Cette caractéristique est particulièrement importante compte tenu des environnements et des applications que nous ciblons et pour lesquelles cette information est difficilement accessible. Enfin, il garantit une progression équitable des tâches de même urgence (y compris les tâches sans contrainte temporelle).

Contrairement aux contraintes de QoS que nous avions utilisées dans le chapitre 4, nous limitons les équations de QoS de l'ensemble ϕ à des équations dont la forme est définie par la grammaire suivante (grammaire au format BNF) :

equation::= "$\forall$ n:" "τ" "(" e "," histo ")" "-" "τ" "(" f "," "n" ")" rel expr
histo::= "n" ope integer | "n"
ope::= "+" | "*"
rel::= "$\leq$" | "$\geq$"

où

- e et f sont des événements observés dans le système. Les événements en question peuvent être l'événement de fin d'exécution d'une tâche T (c'est-à-dire F_T) ou encore le top d'une horloge logique.

- *integer* est un entier.

- *expr* est une expression arithmétique quelconque.

Cette restriction sur la forme des équations de QoS est importante. En effet, elle constitue un compromis que nous estimons raisonnable et qui permet de simplifier sensiblement la complexité de notre algorithme d'ordonnancement (complexité qui est polynomiale, mais qui pourrait être exponentielle si cette restriction était levée).

La difficulté consiste à trouver une conjonction d'équations de la forme précédente qui soit logiquement équivalente à une équation exprimée dans le modèle de spécification du chapitre 4. Certaines équations peuvent être directement traduites. C'est notamment le cas des équations qui utilisent les opérateurs $-$ et / dans la règle *ope*. Malheureusement, d'autres opérateurs interdisent cette traduction (les opérateurs $\exists$ et $\vee$ par exemple) ; ils sont fort heureusement moins fréquents que les autres. Malgré tout, en l'absence d'équiva

lence logique, il est possible d'utiliser des équations de substitution grâce à la relation d'ordre proposée par Stefani et al. [BLA 98]. La relation d'ordre stipule que :

Définition 11 (Relation d'ordre) *Une contrainte Q_1 est dite plus faible qu'une contrainte Q_2 si et seulement si la formule logique*

$$Q_2 \rightarrow Q_1$$

est vraie ($\rightarrow$ est ici l'implication logique).

La solution consiste donc à utiliser une équation de substitution plus forte que l'équation à traduire, ce qui a pour conséquence de surcontraindre le système ainsi modélisé.

6.3.1 Description de l'algorithme

6.3.1.1 Présentation

L'algorithme d'ordonnancement de POLKA est un algorithme orienté échéances. C'est un ordonnanceur en ligne, préemptif et non oisif.

L'ordonnanceur utilise deux informations : l'ensemble ϕ des équations de QoS du système et l'ensemble γ des graphes de tâches.

L'ordonnanceur tient à jour un ensemble contenant toutes les tâches éligibles du système (l'ensemble *pretes*) et un singleton (que nous noterons *elue*) qui identifie la tâche a qui le processeur est alloué. Pour chaque tâche T, l'ordonnanceur va calculer deux informations :

- D_T. C'est un vecteur composé des échéances de chaque activation de la tâche T. Contrairement à d'autres modèles où l'échéance est relative à une information (exemple : relative à la période dans le modèle de Liu et Layland [LIU 73]), il s'agit ici d'échéances absolues, c'est-à-dire d'échéances exprimées dans l'horloge du système. Nous noterons $D_i(n)$ l'échéance de la n^{ème} activation de la tâche i. Initialement, on pose $\forall\ i, n\ :\ D_i(n) = +\infty$.

- E_T. C'est un vecteur composé des dates d'éligibilité pour chaque activation de la tâche T. Par date d'éligibilité de la tâche T, on entend la date à partir de laquelle l'ordonnanceur est autorisé à allouer le processeur à la tâche T. Comme pour le vecteur D_T, les éléments de E_T sont en fait des dates absolues. Nous noterons $E_i(n)$ la date d'éligibilité de la n^{ème} activation de la tâche i. La relation $\forall\ i, n\ :\ E_i(n) \geq \tau(A_i, n)$ est un invariant du système. Initialement, on pose $\forall\ i, n\ :\ E_i(n) = 0$.

De plus, l'ordonnanceur collecte les informations suivantes pour chaque activation des tâches :

- La date de fin d'exécution d'une activation (que nous noterons $\tau(F_T, n)$ pour désigner la date de fin d'exécution de la n^{ème} activation de la tâche T). Nous verrons que cette information est nécessaire pour effectuer les calculs de E et D ainsi que pour construire l'ensemble *pretes*. Initialement, on pose $\forall\ i, n\ :\ \tau(F_i, n) = +\infty$.

- Le temps d'exécution d'une activation. Cette information est inutile pour l'ordonnanceur mais est accessible aux utilisateurs du système qui peuvent l'utiliser à des fins de supervision.

Enfin, pour chaque tâche, l'ordonnanceur gère un compteur qui est incrémenté à chaque fin d'activation. Ce compteur mémorise l'activation pour laquelle la tâche attend ou détient la ressource processeur. Nous noterons act_T l'activation courante de la tâche T.

Schématiquement, le fonctionnement de l'ordonnanceur peut être décrit de cette façon : à chaque fois que la tâche $elue$ termine son activation, l'ordonnanceur effectue les opérations suivantes :

1. Il consigne la date de fin d'exécution de l'activation de la tâche $elue$.
2. Si $elue$ est une tâche pertinente, il détermine les dates d'éligibilité et les échéances qu'il est possible de calculer à partir du jeu d'équations ϕ_{elue}.
3. Il construit l'ensemble des tâches éligibles $pretes$. Pour être insérée dans l'ensemble $pretes$, une tâche T dont l'activation courante est act_T doit satisfaire deux conditions :
   - Si t est l'instant où l'ordonnanceur construit l'ensemble $pretes$, alors la propriété $E_T(act_T) \leq t$ doit être vraie (respect des dépendances temporelles).
   - Toutes les tâches précédentes de T dans le graphe de tâches auquel appartient T doivent avoir exécuté leur $act_T{}^{ème}$ activation (respect des dépendances logiques) ; ou encore $\forall j \in \gamma \mid j \prec T \ : \ \tau(F_j, act_T) < \tau(F_T, act_T)$.

   Finalement, l'ensemble $pretes$ est construit par :

$$pretes = \bigcup_{\forall j \in \gamma} j \mid [E_j(act_j) \leq t] \wedge [\forall \ i \ \in \ \gamma \mid i \prec j \ : \ \tau(F_i, act_j) < \tau(F_j, act_j)]$$

4. Enfin, la dernière étape consiste, à partir de l'ensemble $pretes$, à effectuer l'élection de la nouvelle tâche $elue$. L'élection s'effectue selon une politique EDF : est donc élue, la tâche dont l'échéance est la plus proche de t. Dans le cas où plusieurs tâches possèdent une échéance identique, une politique de tourniquet est appliquée à ces tâches.

6.3.1.2 Calcul des dates d'éligibilité et des échéances

Pour traduire les équations de QoS en termes d'échéances et de dates d'éligibilité, nous utilisons l'heuristique suivante : soient i et j deux tâches pertinentes telles que les équations suivantes existent dans le système :

$$\forall \ h \ : \begin{cases} \tau(j, h \ ope \ r) - \tau(i, h) \leq expr1 \\ \tau(j, h \ ope \ r) - \tau(i, h) \geq expr2 \end{cases} \tag{6.6}$$

Supposons que i vient de terminer sa n^{ème} activation ; il est alors possible de déterminer les échéances et dates d'éligibilité de l'activation k de la tâche j de la façon suivante :

$$\begin{cases} D_j(k) = min(D_j(k), \tau(F_i, n) + expr1) \\ E_j(k) = max(E_j(k), \tau(F_i, n) + expr2) \end{cases} \tag{6.7}$$

où $D_j(k)$ (respectivement $E_j(k)$) est l'échéance (respectivement la date d'éligibilité) de la k^{ème} activation de la tâche j (avec $k = n + r$ si *ope* est une addition et $k = n * r$ sinon). La présence des opérateurs min et max permet de tenir compte des valeurs de $E_j(k)$ et de $D_j(k)$ calculées antérieurement. Rappelons qu'au démarrage de l'application, nous avons $\forall\ i, j\ :\ D_j(i) = \tau(F_j, i) = +\infty$ et $\forall\ i, j\ :\ E_j(i) = 0$. Un cas particulier de l'équation (6.7) se produit lorsque un événement de l'équation de QoS est une horloge logique. Ainsi, si la tâche i termine sa n^{ème} activation et que le jeu d'équations suivant existe :

$$\forall\ h\ : \begin{cases} \tau(i, h) - \tau(H_p, h\ ope\ r) \leq expr1 \\ \tau(i, h) - \tau(H_p, h\ ope\ r) \geq expr2 \end{cases}$$

où H_p est une horloge logique de période p ; alors, nous évaluons les dates d'éligibilité et les échéances par :

$$\begin{cases} D_i(n+1) = min(D_i(n+1), \tau(H_p, k) + expr1) \\ E_i(n+1) = max(E_i(n+1), \tau(H_p, k) + expr2) \end{cases} \tag{6.8}$$

avec $\tau(H_p, k) = a + p.k$ où a est la date de début de l'horloge H_p et $k = n + r$ si *ope* est une addition ($k = n * r$ sinon).

Cette heuristique permet donc de calculer de façon automatique les échéances et dates d'éligibilité grâce aux équations de QoS et ce, sans disposer des temps d'exécution. L'absence des temps d'exécution est une des raisons pour laquelle nous n'utilisons pas les événements A_T. En effet, sans le temps d'exécution, il n'est pas possible de calculer une échéance avec A_T. Néanmoins, si cette heuristique est satisfaisante pour les tâches sans dépendance, elle ne l'est plus lorsqu'il existe des contraintes de précédence sur les tâches. Il est facile de trouver un exemple illustrant cette inadéquation : supposons qu'il existe deux tâches i et j à l'activation n tel que $(i \prec j)\ \wedge (D_j(n) < D_i(n))$. Dans ce cas de figure, tout en respectant la contrainte $\tau(F_i, n) \leq D_i(n)$, il est possible que $\tau(F_i, n) > D_j(n)$, interdisant ainsi le respect de l'échéance de j.

Le problème soulevé consiste à déterminer un algorithme d'ordonnancement pour un ensemble de tâches soumises à des contraintes d'échéances et de dépendances ayant des dates d'arrivée quelconques. Encore une fois, la solution ne doit pas nécessiter l'utilisation des temps d'exécution. Ce problème fut étudié très tôt par Blazewicz qui proposa un algorithme optimal [BLA 76]. L'ordonnanceur de Blazewicz est un ordonnanceur préemptif qui calcule l'échéance d'une tâche i par :

$$\forall\ i, n\ :\ D_i(n) = min(D_i(n), min(\forall\ j\ \mid\ i \prec j\ :\ D_j(n))) \tag{6.9}$$

En d'autres termes, cette solution force une tâche à acquérir la plus petite des échéances de ses tâches successeurs, s'il existe une tâche qui possède une échéance plus petite que la sienne. Dans la suite de cette thèse, nous parlerons d'**héritage** lorsque nous appliquerons l'équation (6.9).

Contrairement aux échéances, les dates d'éligibilité ne posent pas ce problème puisque l'on a naturellement $\forall\ i,j,n\ :\ i \prec j \rightarrow \tau(A_j,n) \geq \tau(F_i,n)$. Si l'algorithme d'ordonnancement conserve les dépendances entre les tâches, il n'est donc pas nécessaire d'effectuer un quelconque héritage des dates d'éligibilité.

Le calcul des échéances et des dates d'éligibilité est donc réalisé en deux phases : la première phase consiste à appliquer les équations (6.7) pour évaluer une échéance et/ou une date d'éligibilité. Enfin dans la deuxième phase, nous effectuons un héritage sur les graphes de tâches où une échéance a été modifiée (application de l'équation (6.9)).

6.3.1.3 Illustration de l'algorithme

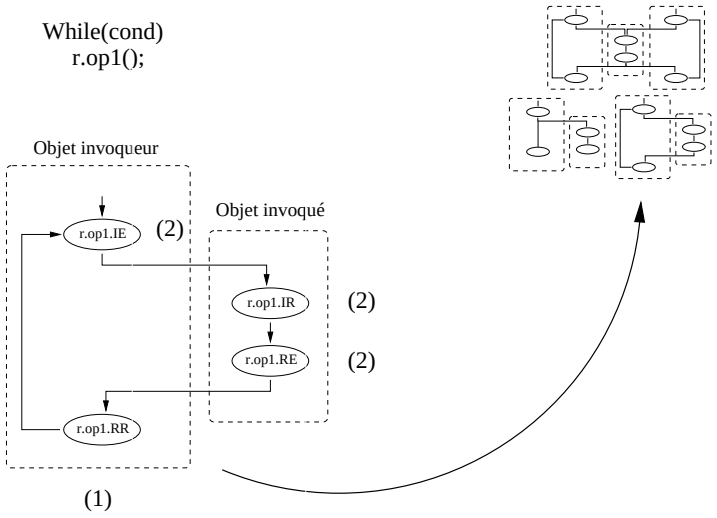


FIG. 6.3 – *Exemple de calculs effectués par l'ordonnanceur*

Illustrons les principes énoncés ci-dessus par un exemple simple qui met en œuvre un seul thread. Regardons le graphe de tâches de la figure 6.3 et supposons que ce graphe modélise un thread dont le code est constitué d'une boucle infinie sur l'invocation de la méthode $op1$ d'un objet r. Pour plus de commodité lorsque dans la suite de ce paragraphe,

nous parlerons d'une tâche t dans le graphe de la figure 6.3, t indiquera en fait la tâche dont la fin est délimitée par l'événement t. Supposons maintenant que le système d'équations suivant

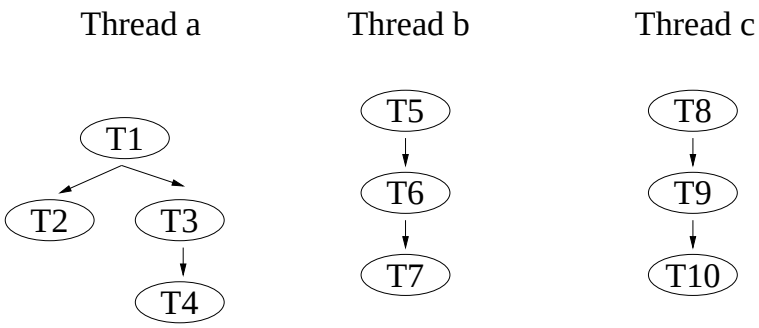
$$\forall\ n\ : \begin{cases} \tau(r.op1.RR, n+1) - \tau(r.op1.RR, n) \leq 10\ ms \\ \tau(r.op1.RR, n+1) - \tau(r.op1.RR, n) \geq 5\ ms \end{cases}$$

soit associé à ce thread et regardons comment l'ordonnanceur calcule les échéances et les dates d'éligibilité des différentes tâches du graphe.

Lors du démarrage du thread, l'ordonnanceur ne peut calculer aucune échéance des tâches $\{r.op1.IE, r.op1.IR, r.op1.RE, r.op1.RR\}$ et celles-ci sont alors initialisées à $+\infty$. Lorsque le thread termine la première activation de la tâche délimitée par l'événement $r.op1.RR$, l'ordonnanceur peut enfin évaluer les échéances des prochaines activations des tâches $\{r.op1.IE, r.op1.IR, r.op1.RE, r.op1.RR\}$. Le calcul se fait alors en deux étapes (cf. figure 6.3) :

1. L'ordonnanceur met d'abord à jour la prochaine échéance de la tâche $r.op1.RR$ en ajoutant 10 à la date de fin d'exécution de la première activation de $r.op1.RR$. Ainsi, l'échéance de la deuxième activation de $r.op1.RR$ vaut $D_{r.op1.RR}(2) = \tau(r.op1.RR, 1) + 10$. De la même façon, l'ordonnanceur évalue la date d'éligibilité par $E_{r.op1.RR}(2) = \tau(r.op1.RR, 1) + 5$.

2. Puis, il applique la formule de Blazewicz pour répercuter cette nouvelle échéance jusqu'à la tâche $r.op1.IE$. Cette deuxième étape consiste à parcourir le graphe de $r.op1.RR$ jusqu'à la racine (tâche $r.op1.IE$) et à effectuer pour chaque tâche i : $D_i(2) = min(D_i(2), D_{r.op1.RR}(2))$. La contrainte sur E n'est pas répercutée vers le haut du graphe, et on a donc $\forall\ i \neq r.op1.RR\ :\ E_i(2) = 0$.



$$\forall\ n\ : \begin{cases} \tau(T_6, n) - \tau(T_3, n) \leq 10\ ms \\ \tau(T_9, n) - \tau(T_3, n) \leq 5\ ms \end{cases}$$

FIG. 6.4 – *Autre exemple de calculs*

Bien sûr, des cas plus complexes peuvent intervenir ; en particulier lorsque ϕ_{elue} contient plusieurs équations dont les événements portent sur des tâches appartenant à des graphes

différents. Dans ce cas, la phase d'héritage sera réalisée sur chacun des graphes où une échéance aura été réévaluée. Par exemple, lorsqu'une occurrence de la tâche T_3 de la figure 6.4 se termine, l'ordonnanceur calcule une nouvelle échéance pour les tâches T_6 et T_9. Une opération d'héritage est alors appliquée aux graphes b et c.

Compte tenu des restrictions sur la forme des équations de QoS que nous avions énoncées, pour que le théorème de Blazewicz ne soit pas violé, il n'est pas nécessaire d'effectuer un héritage récursif[5]. Si ces restrictions étaient levées, la complexité de notre ordonnanceur en serait grandement augmentée (complexité exponentielle). Nous verrons dans la partie suivante qu'elle est polynomiale.

6.3.2 Pseudo-code et complexité

Nous donnons ici le pseudo-code de l'algorithme décrit ci-dessus et nous estimons sa complexité.

6.3.2.1 Pseudo-code de l'algorithme

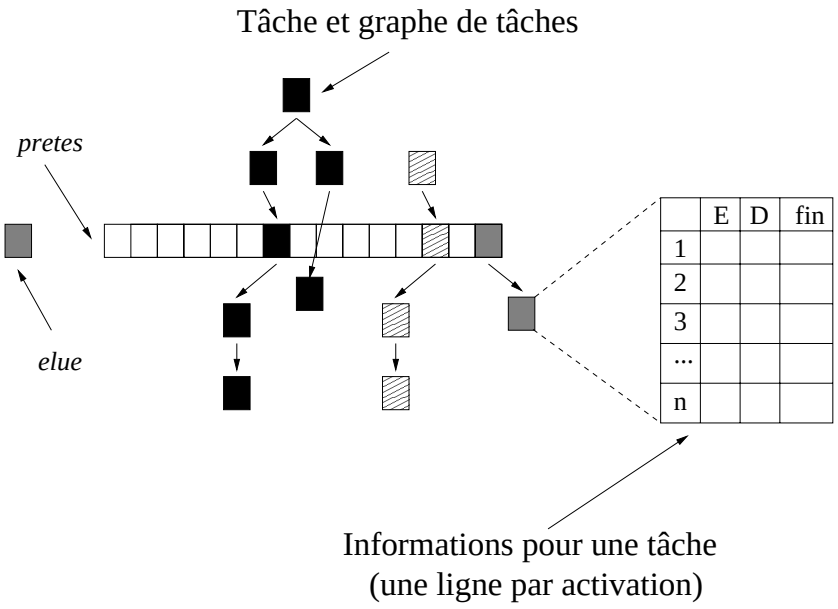


FIG. 6.5 – *Structure de données de l'algorithme*

[5]Par héritage récursif, nous entendons l'opération que consiste, lors de la phase d'héritage, à recalculer les échéances des tâches pertinentes et à éventuellement lancer une nouvelle phase d'héritage sur les graphes où des échéances ont été modifiées.

```
struct tache {
    long E[MAX_ACTIVATION];
    long D[MAX_ACTIVATION];
    long fin[MAX_ACTIVATION];
    long act;
    struct tache *gauche, *droite, *saut;
};
```

```
struct tache* pretes[MAX_THREAD];
struct tache* elue;
```

La syntaxe utilisée ici est proche du langage C. La structure de données de l'ordonnanceur est constituée des informations suivantes :

- A chaque tâche est associée une structure pour mémoriser les informations d'ordonnancement. La structure contient trois tableaux (champs E, D, et fin) qui stockent pour chaque activation la date d'éligibilité, l'échéance et la date de fin d'exécution. La structure mémorise aussi l'activation courante de la tâche (champ *act*) ainsi que trois pointeurs sur des tâches du système (champs *gauche*, *droite* et *saut*). Ces pointeurs représentent les dépendances logiques modélisées par le graphe de tâches. Le graphe de tâches est donc construit grâce à un chaînage de structures au travers de ces pointeurs. Dans la suite, nous parlons de descripteur de tâche pour désigner une de ces structures.

- Un tableau de pointeurs sur des descripteurs de tâche : le tableau *pretes*. Le tableau *pretes* contient autant d'éléments qu'il y a de graphes de tâches dans le système (et donc autant de threads dans le système). L'entrée i du tableau *pretes* contient la tâche courante du thread i.

- Un pointeur qui désigne la tâche en cours d'exécution. C'est le pointeur *elue*.

- Enfin les équations et les horloges logiques sont mémorisées dans des listes de structures (structures *equation* et *horloge*). On construit une liste pour chaque tâche. La structure *horloge* contient la date de début de l'horloge (champ a) ainsi que sa période (champ p). La structure *equation* mémorise des équations conformes à la grammaire BNF de la page 78 ; on y retrouve donc les mêmes informations. Par mesure de lisibilité, les champs de cette structure portent le même nom que ceux de la grammaire BNF.

```
struct equation {
    string e, f;
    string ope;
    string histo;
    string rel;
    long expr;
};
```

```
struct horloge {
    long a;
    long p;
};
```

L'algorithme d'ordonnancement est découpé en cinq parties. La partie principale, la fonction *ordonnance*(), est la fonction appelée par les applications. Elle consigne la terminaison de la tâche courante (adressée par le pointeur *elue*). Ensuite, elle affecte à l'entrée de la tâche *elue* dans la tableau *pretes* sa tâche successeur (successeur désigné par le descripteur de tâche *next* et qui peut être la tâche référencée par le pointeur *gauche*, *droite* ou *saut*). Puis, si la tâche *elue* est une tâche pertinente (cf. définition (10)), elle calcule les échéances et les dates d'éligibilité. Enfin, elle effectue l'élection de la nouvelle tâche grâce à la fonction *edf*(). La fonction *date*() retourne la date système. Dans le cas où la tâche nouvellement élue possède une date d'éligibilité postérieure à la valeur renvoyée par *date*(), l'ordonnanceur attend avant de réveiller la tâche.

Les autres parties de l'algorithme sont les suivantes :

- La fonction *edf*(). Elle parcourt le tableau *pretes* afin de déterminer quelle est la tâche à élire.

- La méthode *calculer*(). Elle effectue le calcul des échéances et des dates d'éligibilité. Selon que l'équation comporte ou non une horloge logique, la méthode *calculer*() appelle la fonction *calcul_horloge*() ou *calcul_evenement*(). Les fonctions *calcul_evenement*() et *calcul_horloge*() implantent respectivement les équations (6.7) et (6.8).

- La fonction *heritage*(). Elle réalise la phase d'héritage (équation (6.9)). Cette fonction est appelée par *calculer*().

```
tache* edf()
{
   tache *eligible, *nonEligible;

   Pour chaque tâche i de pretes
      Si (pretes[i]→E[pretes[i]→act]≤date()
            Et pretes[i]→D[pretes[i]→act] < eligible→D[eligible→act])
         Alors eligible=pretes[i];
      Fsi;

      Si (pretes[i]→E[pretes[i]→act]>date()
            Et pretes[i]→D[pretes[i]→act] < nonEligible→D[nonEligible→act])
         Alors nonEligible=pretes[i];
      Fsi;
   Fpour;

   Si (eligible contient une tâche de pretes)
      Alors Retourner eligible;
      Sinon Retourner nonEligible;
   Fsi;
}
```

```
void ordonnance(tache* next)
{
    elue→fin[elue→act]=date();

    pretes[elue]=next;

    Si (elue est une tâche pertinente)
        Alors calculer(elue);
    Fsi;

    elue=edf();
    Tant que elue→E[elue→act]>date()
        Attendre;
    Ftant;
    Réveiller la tâche elue;
}
```

```
void calculer(tache* finie)
{
    struct tache* herite[MAX_THREAD];
    long terminaison[MAX_THREAD];
    long activationCible;

    Pour chaque équation q
        Si (q utilise une horloge logique)
            Alors activationCible=calculer_horloge(q,finie);
            Sinon Si (q→f==finie)
                        Alors activationCible=calculer_evenement(q,finie);
                  Fsi;
        Fsi;
        Si (une échéance a été modifiée par calculer_evenement ou calculer_horloge)
            Alors Ajouter activationCible et le graphe de tâche modifié
                        dans les tableaux terminaison et herite;
        Fsi;
    Fpour;

    Pour chaque élément i du tableau herite
        Rechercher dans pretes la tâche courante h du thread i
        heritage(pretes[h], pretes[h]→act, herite[i], terminaison[i]);
    Fpour;
}
```

```
long calculer_evenement(equation* q, tache* finie)
{
   long cible;

   Si (q→ope=="*")
      Alors cible=finie→act*q→histo;
      Sinon cible=finie→act+q→histo;
   Fsi;
   Si (q→rel=="≥")
      Alors pretes[q→e]→E[cible]=
         max(pretes[q→e]→E[cible],q→expr+finie→fin[finie→act]);
      Sinon pretes[q→e]→D[cible]=
         min(pretes[q→e]→D[cible],q→expr+finie→fin[finie→act]);
   Fsi;
   Retourner cible;
}
```

```
long calculer_horloge(equation* q, tache* finie)
{
   long cible;

   Rechercher l'horloge h utilisée dans q;
   Si (q→ope=="*")
      Alors cible=(finie→act+1)*q→histo;
      Sinon cible=(finie→act+1)+q→histo;
   Fsi;
   cible=h→a+(h→p*cible);
   Si (rel=="≥")
      Alors pretes[finie]→E[finie→act+1]=
         max(pretes[finie]→E[finie→act+1],q→expr+cible);
      Sinon pretes[finie]→D[finie→act+1]=
         min(pretes[finie]→D[finie→act+1],q→expr+cible);
   Fsi;
   Retourner finie→act+1;
}
```

```
void heritage(tache* herite, long act, tache* termine, long actTerminaison)
{
   Si (herite==termine Et act==actTerminaison)
      Alors Retourner;
      Sinon
         Si (herite est une feuille)
            Alors heritage(herite→saut, act+1, termine, actTerminaison);
            Sinon heritage(herite→gauche, act, termine, actTerminaison);
                  heritage(herite→droite, act, termine, actTerminaison);
         Fsi;

         herite→D[act]=
            min(herite→D[act],
               herite→gauche→D[act],
               herite→droite→D[act]);
   Fsi;
}
```

6.3.2.2 Complexité de l'algorithme

Nous donnons ici une évaluation de la complexité de l'algorithme tel qu'il est décrit dans les paragraphes précédents (fonction *ordonnance*()). Nous détaillons pour ce faire les fonctions *edf*() et *calculer*() ; le reste de la fonction *ordonnance*() est de complexité négligeable. La fonction *edf*() est simple ; elle effectue un parcours séquentiel de la liste des tâches prêtes. La taille de cette liste est de l éléments. Sa complexité est donc en $O(l)$. La fonction *calculer*() est constituée de deux traitements : la phase qui calcule les échéances et les dates d'éligibilité ainsi que la phase d'héritage. Si m est le nombre maximum d'équations portant sur une tâche, alors la complexité de la première phase est en $O(m)$. La complexité de la deuxième phase (la phase d'héritage) est en $O(dgl)$ où d est le nombre d'activations que l'héritage va traverser et g le nombre de tâches dans le graphe. En effet, effectuer un héritage nécessite dg opérations puisque au total, l'algorithme devra traverser dg nœuds du graphe de tâches. Comme au pire cas, l'invocation de la fonction *calculer*() peut générer l héritages, la complexité de la phase d'héritage est en $O(dgl)$. Au total, la fonction *calculer*() est en $O(dgl + m)$.

Finalement, la complexité de l'ordonnanceur est donc polynomiale (puisque la fonction *ordonnance*() est en $O(dgl + m)+O(l)=O(l(dg+1)+m)=O(dgl + m)$).

Notons qu'en pratique d est généralement petit. En effet, lorsqu'une horloge logique est utilisée, nous avons systématiquement $d = 1$. De même, le cas le plus fréquemment rencontré dans les applications multimédias est celui où le paramètre r de l'équation (6.6) vaut $0 \leq r \leq 1$, ce qui induit une valeur de d faible (généralement comprise entre 1 et 2).

Enfin, il est bien sûr possible d 'améliorer sensiblement cette complexité. L'algorithme utilisé dans la plate-forme POLKA (cf. chapitre 7) est d'ailleurs légèrement différent de celui décrit dans ce chapitre. Les différences interviennent sur les moments où est effectué l'hé-

ritage. En effet, sans violer le théorème de Blazewicz, il est possible de retarder l'héritage, diminuant ainsi le nombre de nœuds concernés dans le graphe de tâches. A titre d'exemple, une complexité de $O(l(d+g))$ peut être obtenue avec la modification suivante : lorsqu'un héritage parcourt plusieurs activations, plutôt que de parcourir les dg nœuds du graphes de tâches, une solution consiste à mettre à jour les échéances de chaque feuille du graphe pour toutes les activations concernées par l'héritage, puis d'effectuer un héritage sur l'activation courante du graphe de tâches. Un héritage est effectué plus tard, lorsque toutes les tâches du graphe ont terminé une activation donnée. Dans ce cas de figure, la complexité de l'héritage est en $O(d+g)$ et la fonction *ordonnance*() est donc en $O(l(d+g)+m)$.

Cette optimisation s'appuie sur l'observation des applications que nous avons utilisées. La largeur de leurs graphes de tâches est faible (très souvent de taille 1). D'autres améliorations peuvent être apportées en caractérisant la forme du graphe de tâches ou l'ensemble des équations de QoS. Dans le cas d'un système où l'ensemble des applications est connu, ce type d'optimisation peut considérablement améliorer l'efficacité de l'ordonnanceur.

6.3.3 Optimalité et conditions d'ordonnançabilité

Nous terminons la description de notre algorithme par une discussion sur quelques propriétés classiquement étudiées pour les algorithmes d'ordonnancement temps réel : son optimalité et ses conditions d'ordonnançabilité. Ici, notre propos n'est pas d'offrir les moyens d'effectuer une réservation de ressources mais plutôt de donner, sous l'hypothèse où les temps d'exécution sont connus, des outils de comparaison de notre algorithme à des algorithmes classiques en théorie de l'ordonnancement [LEB 98].

Il n'est pas possible d'énoncer une quelconque propriété sur notre ordonnanceur sans avoir précisément défini pour chaque tâche T le contenu de ϕ_T ainsi que les dépendances qui existent entre les tâches du système (graphes de tâches). Notre modèle de tâches autorise la spécification d'une grande variété de modèles ; nous allons regarder deux d'entres eux : la tâche périodique et la contrainte de distance.

Commençons par le modèle de la tâche périodique de Liu et Layland. Pour ce faire, nous allons définir l'ensemble ϕ_T qui modélise les contraintes temporelles d'un tel modèle. Cette fois-ci, nous allons nous conformer au modèle initial de Liu et Layland (échéances sur activation). Nous notons P_T la période associée à la tâche périodique T. L'ensemble ϕ_T d'une tâche périodique T avec requête sur activation est alors :

$$\forall\ n\ :\ \tau(F_T, n) - \tau(H_{P_T}, n) \leq P_T$$

où H_{P_T} est une horloge logique de période P_T. Il est facile de se convaincre que l'application de notre algorithme à une telle équation génère une valeur de D_T et de E_T conforme au modèle de tâches de Liu et Layland. En effet, en appliquant les équations (6.8), on obtient $\forall\ n\ :\ E_T(n+1) = D_T(n) = (n+1)P_T$; ce qui correspond bien aux contraintes temporelles d'une tâche périodique. Avec ce modèle, le graphe de tâches est réduit à un seul nœud par tâche ; la phase d'héritage n'est donc pas nécessaire. Comme nous avons montré que le calcul des échéances et des dates d'éligibilité conduisait aux contraintes temporelles

du modèle périodique, que notre algorithme est préemptif et que l'élection des tâches est effectuée selon la politique EDF, si toutes les tâches du système sont définies comme la tâche T, les propriétés de notre algorithme sont celles de l'algorithme EDF proposé par Liu et Layland. Notre ordonnanceur est donc optimal **dans ce contexte** et l'équation (2.2) est une condition d'ordonnançabilité applicable dans ce cas.

Si les choses sont simples dans le cas précédent, elles deviennent plus complexes lorsque l'on traite des contraintes moins fréquemment étudiées telles que les contraintes de distance. Soit T une tâche sans dépendance dont l'ensemble ϕ_T est constitué comme suit :

$$\forall\ n\ :\ \begin{cases} \tau(A_T, n+1) - \tau(F_T, n) = 0 \\ \tau(F_T, n+1) - \tau(F_T, n) \leq Q_T \end{cases}$$

Tel qu'il est défini ci-dessus, le modèle de tâches n'est pas optimal s'il est ordonnancé par EDF. Il n'est donc pas non plus optimal avec notre ordonnanceur [LEB 98]. En effet, lorsqu'une tâche termine une activation, l'activation suivante est immédiatement éligible (c'est ce que nous avons appelé "l'effet de bourrage" dans l'annexe A.3.1). Pour les mêmes raisons, la condition d'ordonnançabilité (2.2) n'est plus applicable ; la notion de période n'existant pas dans ce modèle. Toutefois, une condition suffisante mais non nécessaire peut aisément être déterminée :

$$\forall\ T\ :\ \sum_{i=1}^{n} C_i \leq Q_T$$

Où C_T est le temps d'exécution de la tâche T (cf. annexe A.3.1).

Leboucher propose une solution pour construire un ordonnancement optimal pour un tel modèle de tâches avec un algorithme orienté échéance [LEB 98]. Sa proposition consiste à modifier l'ordonnanceur en retardant l'occurrence de l'événement F_T jusqu'à l'échéance de la tâche T : c'est l'algorithme de la dernière goutte ou LSD (*Last Single Drop*). Leboucher montre que, grâce à cet ordonnanceur, l'ordonnancement est optimal et que la condition d'ordonnançabilité (2.2) est une condition nécessaire et suffisante (avec Q_T comme période). Notons qu'en pratique, l'algorithme LSD est difficile à mettre en œuvre. En effet, il est parfois difficile de déterminer quelle est l'instruction d'une tâche qui constitue la dernière goutte. La goutte doit être suffisamment petite pour que son temps d'exécution soit négligeable (afin de ne pas violer l'échéance) mais doit constituer la ou les instructions "validant" les traitements effectués par la tâche (exemple : si une tâche réalise un traitement puis une présentation d'image, la dernière goutte doit être l'instruction déclenchant la présentation de l'image). Enfin, l'exécution en temps et en heure de la dernière goutte est difficile à effectuer de façon efficace.

Han et al. suggèrent une première solution identique à celle de l'algorithme LSD [HAN 96]. Ils proposent toutefois une alternative plus facile à implanter et qui consiste à modifier le modèle de tâches. La première contrainte du jeu précédent est alors remplacée par :

$$\forall\ n\ :\ \tau(A_T, n+1) - \tau(F_T, n) = \epsilon$$

où ϵ constitue une date d'exécution au plus tôt relative aux activations de T.

Nous n'irons pas plus loin sur l'optimalité et les conditions d'ordonnançabilité de notre ordonnanceur. Comme nous l'avons fait sur les deux exemples précédents, il est possible d'exploiter les très nombreux résultats qui ont été présentés dans la littérature sur l'ordonnancement, y compris ceux qui manipulent des modèles définis par un graphe de tâches [CHE 90].

6.4 Extension à une solution répartie

Dans cette partie, nous proposons une extension de l'heuristique présentée ci-dessus pour le support d'applications réparties. Les problèmes d'ordonnancement temps réel dans un contexte réparti ont bénéficié d'un intérêt moins important de la part de la communauté temps réel (cf. la synthèse de Stankovic et al. [STA 95]).

Les problèmes généralement traités dans ce domaine sont des problèmes d'ordonnancement global où des stratégies de placement dynamique des tâches sont proposées [CAR 94, KAI 98].

Le problème que nous étudions ici est plus simple puisque nous supposons un placement statique des tâches. En effet, les applications multimédias utilisent souvent des dispositifs physiques qui limitent les possibilités de migration ou de placement dynamique des tâches (exemples : carte audio, décodeurs matériels et bien sûr l'utilisateur).

D'autres travaux s'attachent à étudier des méthodes d'analyse temporelle des traitements répartis. C'est notamment le cas de l'analyse holistique initialement proposée par Tindell [TIN 94, SUN 94, LEB 95]. Dans une certaine mesure, les propositions ci-dessous s'inspirent de ces travaux.

6.4.1 Notre proposition

La technique que nous allons utiliser a déjà été appliquée dans des problèmes d'ordonnancement pour les applications temps réel réparties [KAO 94, JEF 95, SAM 97]. Le principe consiste à modéliser un traitement réparti sous la forme d'une succession de tâches [TIN 94]. Chacune de ces tâches modélise soit un traitement effectué sur une machine donnée, soit une opération de communication. Dans notre contexte, nous associons un traitement réparti à chaque flot de données multimédias. Le traitement réparti est alors composé de toutes les opérations effectuées sur le flot de la source jusqu'au puits.

Nous proposons, à partir du comportement temporel du réseau et des contraintes temporelles à respecter sur les puits des flots de données multimédias, de déterminer les

contraintes temporelles auxquelles doivent se conformer les sous-tâches successives d'un traitement réparti. Cette approche constitue le cheminement inverse de l'analyse holistique où, à partir des contraintes de chaque sous-tâche, les contraintes de bout en bout sont évaluées (exemple : temps de réponse).

Nous automatisons cette opération grâce à un algorithme de propagation des contraintes de QoS (cf. paragraphe 6.4.2) qui exploite le modèle de spécification décrit dans le chapitre 4. **Appliqué à notre ordonnanceur, déterminer les contraintes temporelles de toutes les sous-tâches d'une machine revient à construire pour cette machine un jeu d'équations de QoS qui sera fourni à l'ordonnanceur.** L'algorithme propage les contraintes de QoS des puits jusqu'aux sources, composants par composants. Lorsqu'un composant réseau est traversé, les équations de QoS requises par le composant réseau sont ajoutées dans le jeu d'équations donné à l'ordonnanceur du site émetteur.

L'ordonnancement global est donc effectué par la présence d'un ordonnanceur sur chacune des machines du système. Les ordonnanceurs exécutent l'algorithme décrit dans la partie 6.3 mais avec des jeux d'équations de QoS différents. Lorsqu'un thread s'exécute sur plusieurs machines, chacun des ordonnanceurs concernés maintient le graphe de tâches du thread.

Pour pouvoir calculer les échéances et les dates d'éligibilité, les ordonnanceurs partagent des informations d'ordonnancement. Ces informations sont constituées de la date de terminaison de chaque activation des tâches du système. Lorsqu'une tâche termine une activation, si celle-ci appartient à un thread qui s'exécute sur plusieurs machines, l'information est diffusée aux machines concernées.

Les opérations de calculs d'échéances et de dates d'éligibilité sont identiques dans le cas d'une application répartie ou centralisée. C'est sur les moments où sont déclenchés ces calculs qu'interviennent les différences. Dans le cas centralisé, les calculs sont effectués uniquement lors de la fin d'une tâche. Dans le cas réparti, ces calculs sont aussi effectués à la réception d'un message avertissant la terminaison d'une tâche sur une machine distante. Ainsi, la terminaison d'une activation de tâche appartenant à un thread s'exécutant sur n machines implique l'émission de $n-1$ messages sur le réseau et de n calculs d'échéances et de dates d'éligibilité.

Le partage des informations d'ordonnancement exige la présence d'une couche transport permettant de prendre en compte les contraintes temporelles existant sur ces informations. Les canaux de communication utilisés pour ces échanges ne doivent pas être nécessairement fiables. En effet, le déséquencement des informations d'ordonnancement d'un thread est sans conséquence puisque le graphe de tâches permet de les réordonner. Quand au déséquencement des informations entre deux threads, il est sans importance. La perte d'informations peut être résolue par un mécanisme de battements de cœur. Ce principe simple consiste à réémettre l'information à période régulière tant que celle-ci n'est pas caduque (exemple : terminaison de la tâche suivante). Un tel mécanisme nécessite au pire cas l'émission de l messages tous les battements de cœur ; où l est le nombre de threads dans le système. Si le système est composé de n machines, la complexité en terme de nombre de messages de cette technique est donc de $O(ln)$. Enfin, certaines applications peuvent sup-

porter la perte occasionnelle d'une information d'ordonnancement (information qui peut alors être remplacée par une estimation). Quels que soient les solutions utilisées (couche de transport fiable, "battements de cœur", estimation, etc), le débit nécessaire à la coopération des ordonnanceurs est peu élevé, compte tenu de la faible quantité d'informations nécessaires pour avertir un ordonnanceur distant de la terminaison d'une tâche.

6.4.2 Algorithme de propagation des contraintes de QoS

```
enum type_composant {reseau, local};

struct composant {
    type_composant type;
    int site, source, puits;
    qos* requise;
    qos* offerte;
};

qos* qosLocale[MAX_SITE];
```

Nous décrivons maintenant l'algorithme de propagation de la QoS qui détermine les jeux d'équations de QoS associées à chaque machine du système. L'algorithme a un fonctionnement proche de celui utilisé dans le chapitre 4 pour effectuer les compositions séquentielles de composants. Il manipule un graphe g orienté qui constitue le graphe de flots de données décrivant l'application (cf. chapitre 3). Chaque nœud du graphe est donc un composant et nous supposerons que les opérations de composition sont déjà effectuées (on est donc en présence d'un graphe où les composants ne possèdent plus de sous-composant). Les composants du graphe sont définis par la structure ci-dessus. Ils sont classés en deux familles : les composants qui modélisent un élément réseau et ceux qui modélisent un élément localisé sur une machine. Dans le premier cas, les champs *source* et *puits* mémorisent le nom des deux machines connectées par le composant réseau. Dans le cas d'un composant ne modélisant pas un élément réseau, le champ *site* contient le nom de la machine qui l'héberge. Comme à notre habitude, le comportement temporel de chaque composant est décrit par un contrat de QoS (champs $requise$ et $offerte$).

L'algorithme consiste à traiter successivement chaque nœud du graphe de flots de données g. Lorsqu'il n'y a plus de nœud dans le graphe g, l'algorithme est terminé et **le tableau *qosLocale* contient pour chaque machine un ensemble d'équations de QoS qui constitue des contraintes suffisantes mais non nécessaires permettant le respect de la QoS utilisateur** (à condition que suffisamment de ressources soient disponibles dans le système). Les nœuds du graphe sont inspectés par l'algorithme lorsque ceux-ci ne possèdent plus de nœuds successeurs. Lorsqu'un nœud i est traité, l'algorithme détermine si ce nœud est un élément réseau. Si c'est le cas, la variable *qosLocal* est mise à jour. Puis, l'algorithme reporte le comportement temporel du nœud i sur ses nœuds prédécesseurs. Finalement, le nœud i ainsi que ses arcs entrants sont supprimés du graphe et

l'algorithme passe au nœud suivant. Notons que la QoS offerte des nœuds appelés "nœuds feuilles" (les feuilles sont les nœuds ne possédant pas de successeur avant le déroulement de l'algorithme de propagation) est mémorisée dans le tableau *qosLocale* ; il s'agit bien souvent de la QoS spécifiée par l'utilisateur. L'algorithme est donné en figure 6.6.

```
void propagation(graph* g)
{
   Tant qu'il reste des composants dans g
      Choisir un composant c sans successeur;
      Si (c est une feuille de g)
         Alors qosLocale[c→site]=qosLocale[c→site] ⋃ c→offerte;
         Sinon Si (c→type==reseau)
                  Alors qosLocale[c→source]=qosLocale[c→source] ⋃ c→requise;
               Fsi;
      Fsi;

      Pour chaque composant p prédécesseur de c
         Reporter c→offerte sur p→offerte;
      Fpour;

      Supprimer le composant c du graphe g ainsi que
         tous les arcs arrivant sur c;
   Ftant;
}
```

FIG. 6.6 – *L'algorithme de propagation des contraintes de QoS*

La complexité d'un tel algorithme est polynomiale. La boucle *Tant que* de l'algorithme est en $O(l)$ si l est le nombre de composants du graphe. L'opération de report consiste à modifier les équations de la QoS offerte par le composant p en fonction des équations de QoS offerte du composant c. Si h est le nombre d'équations maximal dans un contrat de QoS, cette opération est donc en $O(h)$. Enfin, si i est le demi-degré sortant maximal de g, alors la complexité de l'algorithme est en $O(lhi)$. La complexité est donc polynomiale.

6.5 Conclusion

Ce chapitre propose des algorithmes d'ordonnancement qui permettent de déduire de façon automatique, à partir d'une spécification de haut niveau, les directives de gestion du processeur. Les algorithmes proposés dans ce chapitre sont destinés à des environnements banalisés et à des applications dont les besoins en ressources sont difficiles à évaluer. Le principe consiste à traduire, en termes d'échéances et de dates d'éligibilité, les équations de QoS associées aux tâches de l'application. Cette traduction ne nécessite pas la connaissance

a priori du temps d'exécution des tâches ; ce qui est important compte tenu des systèmes ciblés. Nous avons montré que les algorithmes proposés opéraient avec une complexité raisonnable (que ce soit dans le cas réparti ou centralisé). D'autre part, nous verrons dans le chapitre 8 que le surcoût engendré par ces algorithmes reste faible. Il faut toutefois préciser que ces résultats encourageants ont été obtenus grâce à des restrictions, que nous espérons raisonnables, sur la forme des équations de QoS manipulable par l'ordonnanceur.

Chapitre 7

Mise en œuvre : la plate-forme POLKA

Comme nous l'avons vu dans le chapitre 3, la plate-forme POLKA prend en entrée une spécification des applications multimédias en termes d'objets, de threads et de spécification de QoS décrivant le comportement temporel souhaité. Elle se charge alors d'ordonnancer automatiquement l'application de façon à respecter les contraintes de QoS, si suffisamment de ressources sont disponibles dans le système. Elle produit, par ailleurs, des informations de supervision qui permettent de suivre le comportement temporel de l'application et de déterminer celui des composants qui n'est pas connu **a priori**.

La plate-forme POLKA est construite autour d'un bus à objets CORBA, le bus à objets omniORB2 [LO 98] diffusé par le laboratoire de recherche commun à Olivetti et Oracle. Elle est principalement constituée d'un environnement d'exécution (sous la forme de démons et de bibliothèques) et d'un environnement de développement d'applications multimédias (sous la forme de compilateurs).

Dans un premier temps, nous décrivons l'architecture de l'environnement d'exécution. Puis, nous expliquons comment une application multimédia est réalisée avec l'environnement de développement. Nous terminons ce chapitre en détaillant l'interface des services offerts par POLKA.

7.1 Architecture de l'environnement d'exécution

Le but de l'environnement d'exécution (cf. figure 7.1) est d'offrir des services d'ordonnancement et de supervision pour les applications multimédias réparties avec les objectifs suivants (cf. chapitre 1) :

- Fournir des abstractions de threads et d'objets ayant des propriétés de transparence à la localisation.

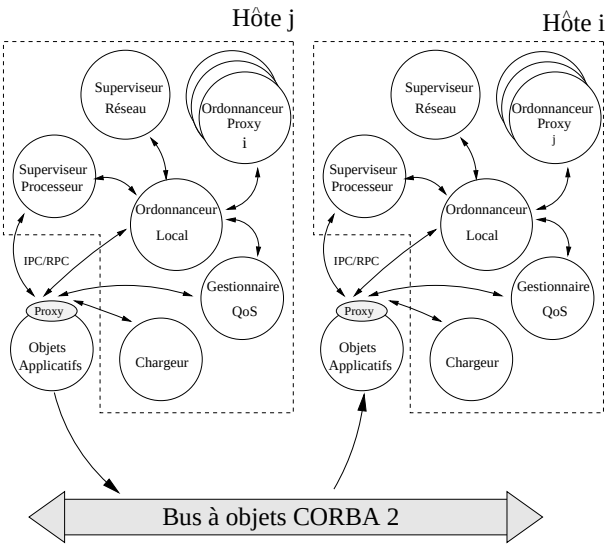


FIG. 7.1 – *Architecture du prototype*

- Offrir un niveau de portabilité qui soit le plus élevé possible. De ce fait, nous excluons l'emploi de systèmes dédiés aux applications temps réels et multimédias. Nous utilisons donc des systèmes d'exploitation non dédiés.

- Utiliser des technologies capables d'intégrer des logiciels existants, éventuellement écrits dans des langages différents (exemples : décodeurs/codeurs vidéo et audio, base de données, interface homme-machine, etc). En effet, les applications multimédias font souvent appel à des technologies complexes et très différentes les unes des autres. Une plate-forme multimédia doit donc être capable de les intégrer aisément.

CORBA répond à plusieurs de ces objectifs. En effet, CORBA est un système orienté objets offrant des services de transparence à la localisation. Une entité souhaitant invoquer un objet le fera d'une façon identique que l'objet soit local ou distant. De plus, CORBA offre des possibilités intéressantes d'intégration de systèmes ainsi qu'un certain niveau de portabilité et de support de l'hétérogénéité. Il est possible dans CORBA, de faire interopérer des applications écrites dans différents langages et exécutées dans des environnements différents. Enfin, ce qui n'est pas négligeable, CORBA permet de développer très rapidement et facilement des applications réparties.

CORBA n'est cependant pas particulièrement bien adapté au support des applications multimédias, ou plus généralement des applications ayant des contraintes temporelles [CNE 96, GIL 00]. Dans le cas des applications multimédias, on peut citer l'absence d'abstraction adaptée (telle que la notion de flot). Il est impossible de spécifier des contraintes temporelles et encore moins d'offrir un quelconque support pour ces contraintes. Le protocole de communication sous jacent, IIOP (*Internet Inter-ORB Protocol*), est peu adapté aux transferts de données ayant des contraintes temporelles. Enfin, de façon plus générale,

il n'existe pas de moyen pour gérer finement les ressources (exemple : choix d'un ordonnanceur à échéances pour ordonnancer les invocations de méthode). Le support de la QoS dans ce type d'environnement est d'ailleurs un problème difficile. Des travaux sur les bus à objets et les contraintes temps réel ont vu le jour. Ils portent essentiellement sur des aspects architecturaux ; c'est notamment le cas de Jonathan [DUM 98], DIMMA [DON 98] ou QuO [VAN 98]. Certaines solutions se sont aussi intéressées aux problèmes d'ordonnancement dans CORBA ; TAO (*The Ace ORB*) en est un exemple intéressant [GIL 00].

Bien que l'implantation actuelle de notre prototype soit réalisée à l'aide d'un bus à objets conforme au standard CORBA, il est important de préciser que le modèle de QoS ainsi que les techniques d'ordonnancement proposées dans cette thèse ne sont pas dédiées à CORBA. Ainsi, les systèmes à objets utilisant des protocoles de communication mieux adaptés aux applications multimédias que ne l'est IIOP, ou qui supportent la notion de flot, peuvent tout à fait bénéficier des techniques présentées ici [DUM 98, DON 98]. En fait, les techniques décrites dans cette thèse peuvent être appliquées à tous les systèmes à objets où des événements peuvent être observés durant leurs interactions.

7.1.1 Les objets du démon

Dans notre plate-forme, les objets POLKA sont donc des objets CORBA. Le démon est lui-même une application CORBA composée des objets suivants : le gestionnaire de QoS, le chargeur, l'ordonnanceur local, le superviseur processeur, les ordonnanceurs *proxies* et le superviseur réseau (cf. figure 7.1).

Le **gestionnaire de QoS** analyse les descriptions de QoS et fournit à l'ordonnanceur local les jeux d'équations de QoS.

Le **chargeur** initialise en mémoire les informations nécessaires à l'ordonnancement.

L'ordonnancement global de l'application est réalisé en faisant coopérer les ordonnanceurs locaux des différents sites. La plate-forme ordonnance les flots d'exécution sur chaque site du système, conformément aux équations de QoS. L'algorithme d'ordonnancement utilisé par les ordonnanceurs locaux est bien sûr celui décrit dans le chapitre 6. Néanmoins, la politique du tourniquet décrite page 80 n'a pas été implantée : aujourd'hui, lorsque plusieurs tâches possèdent une même échéance, l'ordonnanceur élit celle dont l'identifiant est le plus petit[1].

Le **superviseur processeur** offre à l'utilisateur la possibilité d'obtenir un retour d'informations sur l'ordonnancement des tâches du système ; en particulier lorsque celles-ci ne peuvent respecter leurs contraintes temporelles. Avec les services du superviseur processeur, une application peut choisir d'adapter en cours d'exécution ses besoins en fonction des ressources disponibles.

[1] En fait, l'absence de politique du tourniquet a surtout une incidence sur les tâches n'ayant pas de contrainte temporelle : celles-ci ont toutes une échéance initialisée à $+\infty$ et ce pour toutes leurs activations. Notons que l'absence de garantie de progression pour ces tâches peut conduire à des cas d'interblocage. Pour les tâches ayant des contraintes temporelles, les échéances sont croissantes d'une activation à une autre, l'absence de tourniquet a donc une influence nulle.

Sur chaque site i, il existe un objet **ordonnanceur *proxy*** j pour chaque site j du système (avec $j \neq i$). Ces *proxies* offrent à l'ordonnanceur local une vue des informations d'ordonnancement manipulées par les objets ordonnanceurs distants.

Enfin, le **superviseur réseau** collecte des informations concernant l'état du réseau telles que le taux de perte des paquets, les délais de communication de bout en bout et la gigue maximale. Ces informations sont utilisées par les ordonnanceurs locaux.

Le superviseur réseau et les ordonnanceurs *proxies* masquent aux autres objets de l'architecture POLKA le comportement temporel du réseau ainsi que les problèmes dus à la répartition. Les traitements effectués par ceux-ci et le type des informations de supervision obtenues dépendent des caractéristiques temporelles du réseau sous-jacent (exemple : on n'utilisera pas de supervision en présence d'un réseau offrant des services synchrones ou isochrones).

Pour ce faire, le superviseur réseau et les ordonnanceurs *proxies* utilisent le protocole RTP [SCH 96] sur UDP. Le protocole RTP a pour objet le transport de données ayant des propriétés temporelles (données multimédias par exemple). RTP ne garantit pas le respect des propriétés temporelles : c'est au réseau sous-jacent que revient cette tâche. Une session RTP est constituée d'un ensemble d'extrémités qui émettent ou reçoivent un flot de données. Une session peut être unicast ou multicast. Enfin, à chaque session RTP est associé un flot de contrôle géré par le protocole RTCP. Ce flot fournit, en particulier, une estimation du temps de communication des paquets RTP. Une session RTP est ouverte pour chaque couple de machines du système POLKA.

Le superviseur réseau exploite les informations produites par les paquets RTCP pour construire une évaluation des temps de communication des paquets RTP ainsi que des invocations de méthode d'objets applicatifs : si $T1$ désigne la date d'un événement IE observé sur le site i et si $T2$ correspond à la date de la première action correspondante qui peut être observée sur le site j, alors $T2 - T1$ estime le temps nécessaire pour acheminer l'invocation du site i vers le site j (cette estimation fait l'objet de traitement par des filtres).

Les superviseurs réseaux et les ordonnanceurs *proxies* utilisent les canaux de communication RTP. Les superviseurs réseaux se partagent les dates $T1$ et $T2$ par ce canal. Quant aux ordonnanceurs *proxies*, ils les utilisent pour véhiculer les informations d'ordonnancement.

Tous traduisent les informations qu'ils obtiennent des processeurs distants dans leur horloge. Cette conversion évite l'utilisation d'une horloge globale au système POLKA. La conversion utilise l'estimation des temps de communication des paquets RTP observée par le superviseur réseau.

En effet, chaque paquet RTP contient la date de son émission. Supposons que e soit l'événement associé à la fin de l'exécution d'une tâche sur le site i. Notons alors $\tau(i, e)$ la date de cet événement exprimée dans l'horloge de i. Cette information, qui constitue une information d'ordonnancement, est émise dans un paquet RTP à la date $\tau(i, s)$, où s est l'événement associé à l'émission du paquet. Le paquet RTP contient alors, à la fois $\tau(i, e)$ et $\tau(i, s)$. Ce dernier arrive sur le site j à la date $\tau(j, r)$ où r est l'événement associé à la réception du paquet RTP. Puisque le superviseur réseau fournit une estimation du temps

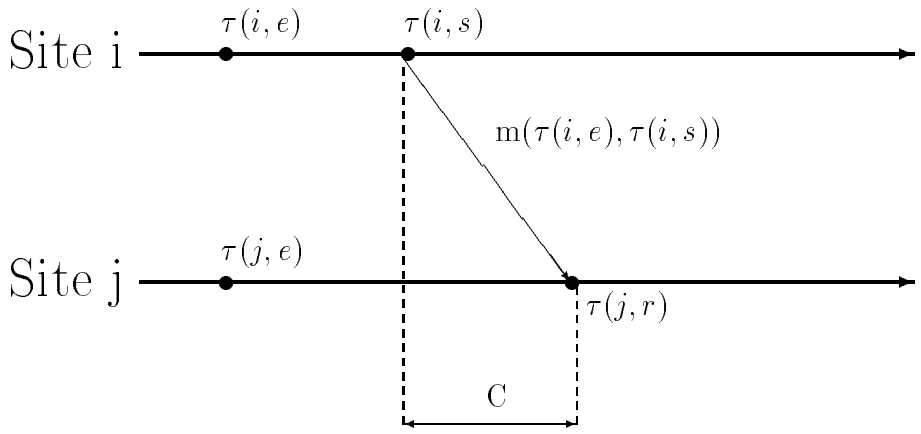


FIG. 7.2 – *Conversion des informations d'ordonnancement*

de transmission de ce paquet, le site récepteur peut "convertir" la date $\tau(i,e)$ en $\tau(j,e)$ par le calcul : $\tau(j,e) = \tau(j,r) - C - (\tau(i,s) - \tau(i,e))$ où C peut être, selon la façon dont est modélisé le réseau (composant de retard fixe ou de retard variable), soit une estimation du temps moyen de communication d'un paquet RTP, soit le temps de communication le plus long d'un paquet RTP. $\tau(i,s) - \tau(i,e)$ est le délai entre l'événement e et l'émission du paquet RTP correspondant (cf. figure 7.2).

7.1.2 La couche d'adaptation

L'utilisation de CORBA constitue un premier pas vers la portabilité des applications multimédias réparties. Elle est toutefois insuffisante. En effet, hormis les faiblesses de CORBA à cet égard, l'ordonnancement automatique des applications ciblées peut demander l'utilisation de services qui sont susceptibles de différer d'un système à un autre. Ainsi, pour permettre l'ordonnancement temps réel, certains systèmes fournissent une implantation des threads normalisés par POSIX. Force est de constater qu'il existe des différences parfois importantes entre les différentes implantations (que ce soit sur l'interface ou sur le comportement des services offerts).

Pour pallier à ces inconvénients, nous introduisons une "couche d'adaptation" dans la plate-forme. Elle a pour rôle d'offrir une vision homogène des services du système sous-jacent utilisés par POLKA et d'harmoniser leur comportement (exemple : elle configure le temporisateur de Linux avec une précision suffisante en utilisant les modifications proposées par le projet KURT [SRI 98]). En pratique, la couche d'adaptation est constituée d'un ensemble de classes C++ *inline* offrant principalement des abstractions de threads[2], de

[2]L'abstraction de thread définie dans la couche d'adaptation n'est pas directement manipulée par le développeur. Elle constitue une abstraction de flot d'exécution centralisée sur un seul processeur. En fait, les threads de la couche d'adaptation sont utilisés par la plate-forme POLKA pour construire la notion de

mécanismes de communication inter-processus (IPC et RPC locaux), d'outils de synchronisation intra et inter-processus et de temporisateurs. Nous en avons développé une version pour Solaris et une version pour Linux. Une implantation sur Linux-L4 [HAR 97] est en cours de réalisation.

7.2 Méthode de construction d'une application au-dessus de POLKA

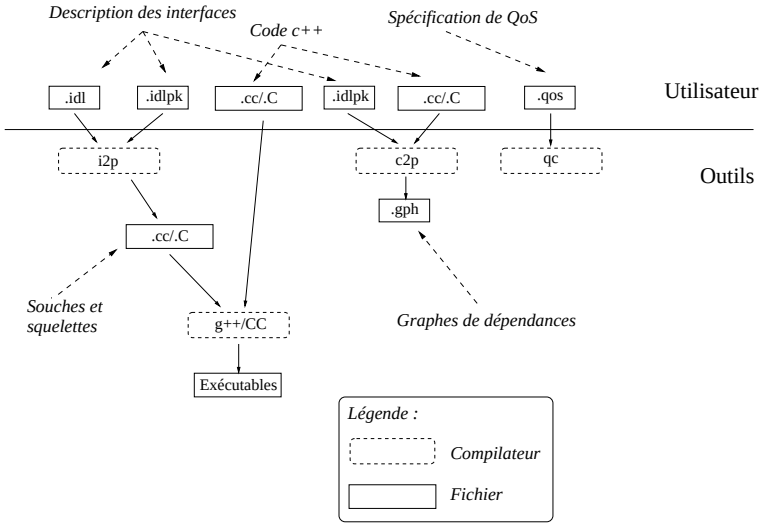


FIG. 7.3 – *Chaîne de production d'une application avec* POLKA

Nous présentons maintenant les différentes étapes qui interviennent lorsqu'un utilisateur souhaite construire une application avec POLKA.

La première étape consiste à spécifier l'application en termes d'objets et de threads ; ce qui se traduit, en particulier, par la description IDL de l'application. La spécification IDL est utilisée pour générer les souches et squelettes CORBA qui sont instrumentés par POLKA pour que l'ordonnanceur soit invoqué à chaque occurrence des événements IE, IR, RE et RR. Pour ce faire, le développeur utilise le compilateur IDL *i2p* de POLKA (cf. figure 7.3)[3]. Le compilateur *i2p* prend en entrée une interface IDL (fichier *.idl*) ainsi qu'un fichier énumérant les interfaces et méthodes à instrumenter (fichier *.idlpk*).

Par la suite, l'application est développée de la même façon que n'importe quelle autre application CORBA dans un environnement multi-thread. Toutefois, plutôt qu'utiliser la bibliothèque de thread offerte par le système, le développeur exploite la bibliothèque de

threads "à la Clouds" qui constitue l'abstraction finalement manipulée par l'utilisateur.

[3] L'utilisation du compilateur *i2p* rend dépendant la plate-forme POLKA à un ORB donné. Dans l'avenir, lorsque les ORB offriront les services d'intercepteurs (cf. CORBA 2.3 [OMG 99]), cette contrainte sera levée. La portabilité de notre environnement en sera accrue.

thread de POLKA. Bien entendu, aucune information d'ordonnancement (exemples : priorité, classe d'ordonnancement, quantum, etc) n'est précisée lors de la création des threads dans les programmes ; ces informations sont déduites pendant l'exécution de l'application par l'ordonnanceur POLKA grâce à la QoS spécifiée.

Une fois l'application écrite et compilée, il est nécessaire de spécifier le modèle de QoS qui permettra d'ordonnancer les threads de l'application. La phase de modélisation commence par l'identification des composants du système et de l'application. Chaque composant est alors spécifié en termes de jeux de contrats de QoS. Modéliser un composant consiste à déterminer le jeu d'équations de QoS requise pour chaque jeu d'équations de QoS offerte. Ce travail nécessite une connaissance précise du comportement temporel du composant. Néanmoins, POLKA offre une bibliothèque de composants génériques qui modélisent des comportements temporels usuels et qui peuvent être instanciés pour modéliser un composant spécifique (exemple : un composant qui applique à chaque événement d'entrée un retard compris entre une borne maximale et une borne minimale peut être instancié pour modéliser un réseau offrant des services de communication isochrones [STE 95b]).

Un utilisateur peut construire un composant donné par composition parallèle ou séquentielle de plusieurs composants de la bibliothèque. La bibliothèque peut être enrichie par l'utilisateur.

Une fois les composants modélisés, le développeur doit construire le graphe de flots de données qui représente le système dans sa globalité. Cette étape consiste à connecter les différents composants du système. Les événements IE, IR, RE et RR du modèle objet constituent les éléments permettant d'effectuer cette connexion.

Pour aider le développeur, un compilateur de QoS (le compilateur *qc* sur la figure 7.3) analyse et effectue la composition des descriptions de composant. Le compilateur calcule aussi, à partir du graphe de flots de données, les différents jeux d'équations de QoS qui doivent être fournis sur chacun des processeurs du système. Ce calcul est effectué en propageant les contraintes de QoS des puits jusqu'aux sources du graphe de flots de données. L'algorithme utilisé pour effectuer ce calcul est décrit dans la partie 6.4.2. Il est réutilisé par le gestionnaire de QoS de la plate-forme POLKA pour analyser les informations de QoS qui lui seront envoyées au début ou durant l'exécution de l'application.

Une fois le modèle de QoS réalisé, il ne reste plus qu'une étape à franchir : la construction des graphes de tâches associés à chaque thread de l'application. En théorie, ce traitement est automatisé grâce au compilateur *c2p* qui analyse les invocations de méthode CORBA et construit automatiquement les graphes de tâches (cf. figure 7.3). En pratique, le compilateur *c2p* n'est pas opérationnel à ce jour et les graphes de tâches doivent être saisis manuellement.

Il faut noter qu'une méthode différente peut être utilisée pour simplifier la phase de test (méthode que nous avons systématiquement employée pour la réalisation des applications du chapitre 8). En effet, les applications multimédias sont souvent complexes et valider les aspects fonctionnels et qualitatifs en même temps peut devenir une tâche difficile. La solution consiste à séparer en deux la phase de test. Tout d'abord, le concepteur

```
class polka_proxy {
   public:
      polka_proxy(void);
      ~polka_proxy();
      // Methodes de l'objet ordonnanceur
      //
      void schedule(const polka_tid tid, const char task);
      void endSchedule(void);
      void endThread(const polka_tid tid);
      polka_status suspendThread(const polka_tid tid);
      polka_status resumeThread(const polka_tid tid);
      polka_status getThreadStatus(const polka_tid tid, polka_threadStatus& s);
      polka_status getThreadSelf(polka_tid& tid);
      // Methodes de l'objet chargeur
      //
      polka_status subscribeObject(const char* name);
      polka_status subscribeThread(const char* name, polka_tid& tid);
      polka_status deleteObject(const char* name);
      polka_status deleteThread(const char name, const polka_tid tid);
      // Methodes de l'objet superviseur processeur
      //
      polka_status registerSchedulerFeedback(const polka_tid tid);
      polka_status waitSchedulerFeedback(struct schedulerFeedback& data);
      // Methodes de l'objet gestionnaire de QoS
      //
      polka_status addQosFromFile(const char* file);
      polka_status mergeQosFromFile(const char* file);
      polka_status deleteQosFromFile(const char* file);
      polka_status addQosFromBuffer(const char* buff, const int size);
      polka_status mergeQosFromBuffer(const char* buff, const int size);
      polka_status deleteQosFromBuffer(const char* buff, const int size);
      polka_status setClock(const char* id, const long period, const long arrivalTime);
      polka_status getSystemTime(double& time);
};
```

FIG. 7.4 – *Interface du proxy* POLKA

développe l'application en utilisant les threads du système d'exploitation. Il génère les souches et squelettes non instrumentés. Puis, il compile et teste les aspects fonctionnels de l'application. Dans un deuxième temps, il se concentre sur les aspects qualitatifs de l'application.

Il substitue les threads du système d'exploitation par des threads POLKA et génère les souches et squelettes avec *i2p*. Enfin, il teste les aspects temporels/qualitatifs de l'application. Il n'est pas toujours possible d'utiliser cette méthode, mais quand c'est le cas, le gain d'efficacité est important.

7.3 Interface de programmation offerte aux développeurs

Nous décrivons maintenant l'interface utilisée par les développeurs pour construire leurs applications. Nous regardons dans un premier temps comment interagir avec les objets du démon POLKA (cf. figure 7.1). Puis, nous énumérons ce qu'il est possible d'exprimer en terme de QoS dans la plate-forme. Enfin, nous décrivons les services disponibles qui permettent de faire varier la QoS durant l'exécution de l'application.

7.3.1 Des objets et des threads

La communication entre les applications et le démon POLKA s'effectue au travers d'un objet *proxy* (cf. figure 7.1). Une instance de cet objet est présente dans chaque processus applicatif. La communication entre le *proxy* et les objets du démon est réalisée grâce à des IPC et des RPC locaux (qui sont des abstractions offertes par la couche d'adaptation de POLKA). Un objet applicatif peut être amené à interagir avec quatre objets différents : son ordonnanceur local, le gestionnaire de QoS, l'objet chargeur et le superviseur processeur. L'interface du *proxy* est résumée dans la figure 7.4. Cette dernière contient tous les services des objets POLKA qui sont accessibles par une application.

En premier lieu, les applications commencent par s'adresser à l'objet chargeur. Les interactions avec l'objet chargeur permettent à l'applicatif de créer de nouveaux threads et de notifier au démon l'existence de nouveaux objets (méthodes *subscribeThread* et *subscribeObject*). A chaque nouveau thread, l'application reçoit un identifiant de thread POLKA dont le type c++ est *polka_tid*. Un *polka_tid* est un type opaque pour l'utilisateur. Il est constitué de la concaténation d'un numéro de machine et d'un numéro séquentiel de thread. L'unicité de ces identifiants dans le temps et dans l'espace est garantie. La création d'un nouveau thread a pour conséquence le chargement et l'initialisation du graphe de tâches dans les structures de données de l'ordonnanceur local. Inversement, les méthodes *deleteThread* et *deleteObject* permettent de libérer les ressources allouées dans le démon POLKA.

Plus importantes sont les méthodes exportées par l'ordonnanceur. Une tâche est délimitée par les méthodes *schedule* et *endSchedule*. La méthode *schedule* permet au thread *tid* d'avertir l'ordonnanceur qu'il souhaite obtenir le processeur pour l'exécution de la tâche *task*. Cette méthode est bloquante tant que le processeur n'a pas été alloué au thread qui l'invoque. Lorsque le processeur est alloué au thread *tid*, celui-ci exécute sa tâche puis,

lorsque la tâche est terminée, invoque la méthode *endSchedule* pour en informer l'ordonnanceur. Le calcul des échéances, des dates d'éligibilité et l'élection de la tâche à exécuter par la suite, sont effectués à cet instant (cf. chapitre 6). L'utilisateur ne se sert généralement pas directement de ces méthodes puisqu'elles sont invoquées dans les souches et squelettes générés par le compilateur *i2p* (cf. paragraphe 7.4). En revanche, l'utilisateur peut explicitement demander à suspendre, réveiller ou obtenir des informations sur des threads avec les méthodes *suspendThread*, *resumeThread*, *getThreadStatus* et *getThreadSelf*. Enfin la méthode *endThread* permet d'avertir l'ordonnanceur que le thread *tid* ne souhaite plus obtenir le processeur. Une fois qu'un thread a invoqué cette méthode, il ne sera plus jamais élu. *endThread* ne détruit pas pour autant le thread en question. En effet, il est parfois nécessaire de conserver, bien après sa fin, les informations d'ordonnancement d'un thread. C'est notamment le cas lorsque plusieurs threads sont synchronisés par des équations de QoS. Un thread et ses informations d'ordonnancement ne sont définitivement détruits que par la méthode *deleteThread* de l'objet chargeur. Notons que, contrairement à une interface de thread POSIX, il n'existe pas de méthode permettant de manipuler une quelconque abstraction de priorité ou de classe d'exécution puisque ces informations sont automatiquement calculées par l'ordonnanceur grâce à la spécification de QoS. L'objectif de la plate-forme POLKA est justement de libérer le concepteur du choix de ces informations pour chaque thread.

L'application peut obtenir un retour sur l'ordonnancement produit grâce au superviseur processeur. Ce retour se matérialise par l'envoi d'un message du superviseur vers l'application lorsque l'ordonnanceur n'a pas pu respecter les contraintes temporelles d'une tâche. Ce message est alors lu par l'application grâce à la méthode *waitSchedulerFeedback*. Chaque message contient le nom de la tâche victime, sa date de fin d'exécution, son échéance et sa date d'éligibilité. Il est alors de la responsabilité de l'application de diminuer ses besoins en ressources en cas de violation de contraintes temporelles. Pour recevoir ces messages, l'application doit préliminairement demander que le superviseur les lui envoie ; ceci s'effectue par l'invocation de la méthode *registerSchedulerFeedback* pour chaque thread sur lequel l'application souhaite obtenir ces informations. D'autres informations sur l'ordonnancement peuvent être obtenues de la part du démon. En effet, à la demande d'un développeur, l'ordonnancement peut être sauvegarder sur disque. Chaque entrée de cet historique comprend la tâche et son numéro d'activation pour qui le processeur a été alloué. L'échéance, la date d'éligibilité et la date de fin d'exécution sont aussi consignées. Cette trace post-mortem est particulièrement intéressante lors de la phase de validation de l'application, mais ne peut pas être exploitée lors de son exécution pour adapter ses besoins aux ressources disponibles.

Le dernier objet accessible par le *proxy* est le gestionnaire de QoS. Les méthodes qu'il exporte permettent de modifier les définitions de QoS manipulées par les objets du démon. Les descriptions de QoS peuvent être lues, soit à partir d'un fichier, soit à partir de l'espace mémoire de l'application. Il est possible d'ajouter, de modifier ou de supprimer des descriptions de QoS. L'opération d'ajout s'effectue grâce aux méthodes *addQosFromFile* et *addQosFromBuffer*. Les méthodes *mergeQosFromFile* et *mergeQosFromBuffer* permettent de substituer des descriptifs de QoS (le comportement de *merge* est identique

à *add* si de nouveaux éléments sont définis). La suppression est réalisée par les méthodes *deleteQosFromFile* et *deleteQosFromBuffer*[4]. Enfin, une application peut aussi, en invoquant les services du gestionnaire de QoS, modifier les caractéristiques d'une horloge logique (cf. paragraphe 4.3) grâce à la méthode *setClock*, ou consulter l'horloge[5] du système POLKA au moyen de la méthode *getSystemTime*.

7.3.2 Description du modèle de QoS

Dans le chapitre 4, nous avons défini un modèle pour spécifier le comportement temporel des éléments d'un système multimédia. Dans ce paragraphe, nous proposons un langage permettant de décrire ces spécifications. Ce langage définit les spécifications temporelles que sont capables d'analyser le gestionnaire de QoS du démon POLKA et le compilateur *qc*. Sa syntaxe est exprimée avec la notation BNF. Dans les règles ci-dessous, *ident* est un identificateur, *entier* est un entier et *reel* est un réel. Le point de départ de la grammaire est la règle *debut*.

```
debut::= composants {composants} {dfd}
```

Les spécifications de QoS sont constituées de deux parties : la définition des composants et la définition des graphes de flots de données. La définition d'un composant commence par la règle *composants* :

```
composants::= "component" ident ":"
              [ ident "/" ident {"," ident "/" ident} ]
              signaux
              {contrats}
              "end"

signaux::=    "signal" unsignal {"," unsignal} ";"

unsignal::= direction ident
          | direction ident "=" portee

direction::= "in"
           | "out"
```

[4] Notons que dans la plate-forme il existe une commande UNIX (la commande *ql*) qui permet de charger des fichiers de QoS de façon interactive.

[5] Cette horloge est relative à l'horloge du système d'exploitation. Elle est utilisée pour tous les calculs d'ordonnancement.

```
contrats::= "qoscontract" ident ":" [ portee {"," portee} ]
               "common" {qoscommune}
               "provided" {qos}
               "required" {qos}
               "end"

qoscommune::= declencheur
          | variable
          | horloge

qos::= equation
     | declencheur
     | variable
     | horloge

horloge::= "clock" ident "=" "(" [ expression ] ","
         [ expression ] ")" ";"

variable::= "var" ident "=" "{" ident {"," ident} "}" ";"

declencheur::= "trigger" ident affectation {"," affectation} ";"

affectation::= portee [ "[" entier "]" ] "=" expression

equation::= "eqos"
            (   (ident "T" "(" portee , historique ")"
                 - "T" "(" portee "," "n" ")" rel expression {expression})
            | "empty"
            )
            ";"

historique::= "n" operateur donnee
        | "n"


portee::= ident {"." ident}
expression::= [signe] donnee [unite]
donnee::= entier | ident | "infinity"
rel::= "<" | ">"
signe::= "+" | "-"
operateur::= "+" | "-" | "/" | "*"
unite::= "ms" | "s" | "us" | "mns"
```

Un composant comprend les informations suivantes :

- Une liste de signaux (mot clef *signal*) qui spécifie les flots de données continues qui entrent et sortent du composant. Les signaux sont orientés ; ils sont soit en entrée (mot clef *in*), soit en sortie (mot clef *out*).
- Une liste de contrats de QoS (mot clef *qoscontract*). Chaque contrat comprend deux jeux de déclarations de QoS (la QoS requise et la QoS offerte par le composant). Il est possible de définir des déclarations communes (règle *qoscommune*). Les déclarations de QoS communes sont visibles à la fois par la QoS offerte et par la QoS requise. Dans la QoS offerte ou requise, il est possible de déclarer des équations, mais aussi des horloges, des variables et des déclencheurs. Nous avons déjà présenté les notions d'équations et d'horloges ; nous n'y reviendrons pas. En revanche, les notions de variables et de déclencheurs sont nouvelles. Nous décrivons l'utilisation des déclencheurs dans le paragraphe 7.3.3 ; quant aux variables, elles permettent à un ensemble d'événements de partager le même historique (et donc de partager les mêmes informations d'ordonnancement). L'utilisation d'une variable permet, par exemple, de spécifier une même contrainte temporelle sur les deux branches d'un graphe de tâches issues d'un test conditionnel.

Nous donnons, dans l'annexe C, tout un ensemble de composants comprenant des sous-composants. Lors d'une opération de composition, l'utilisateur spécifie dans l'en-tête du composant la liste des sous-composants qui doivent le constituer (cf. règle *composants*). La liste de sous-composants (dont chaque élément est séparé des autres par une virgule) stipule pour chacun d'entre eux un nom de "renommage". Ce nom est utilisé pour atteindre les définitions du sous-composant grâce à l'opérateur de portée ".". En effet, tout composant accède à toutes les déclarations effectuées par ses sous-composants (déclarations qui sont préfixées de son nom de renommage). Toutefois, il est à la charge de l'utilisateur de spécifier quelles sont les déclarations de sous-composants qui doivent être intégrées dans le composant "composite". Ainsi, l'utilisateur doit énumérer les contrats de QoS des sous-composants qu'il souhaite utiliser lorsqu'il créée un nouveau contrat de QoS (cf. règle *contrats*). L'utilisateur n'est pas obligé de récupérer toutes les déclarations de ses sous-composants. Néanmoins, il doit au moins spécifier les connexions entre les signaux du composant composite et ceux de ses sous-composants (règle *signaux*).

```
component flot :
        signals out S;

        qoscontract intra :
                common
                provided
                        eqos f1 T(S,n+1) - T(S,n) < 5 ms;
                        eqos f2 T(S,n+1) - T(S,n) > 5 ms;
                required
```

```
        end
end

component flotsSynchronises : video/flot, audio/flot
        signals out a=audio.S, out v=video.S;

        qoscontract inter : video.intra, audio.intra
                common
                provided
                        eqos l1 T(a,n) - T(v,n) < beta ms;
                        eqos l2 T(v,n) - T(a,n) < alpha ms;
                required
        end
end
```

L'exemple ci-dessus montre comment réaliser une composition. Le composant *flotsSynchronises* est construit grâce à deux instances du composant *flot*. Les deux sous-composants sont renommés *video* et *audio*. Le composant *flotsSynchronises* exporte un contrat de QoS nommé *inter*. Ce contrat de QoS est constitué de deux équations déclarées au sein du composant *flotsSynchronises* qui modélisent une synchronisation inter-flots. Le composant contient en outre les équations déclarées dans les sous-composants *video* et *audio* et qui modélisent les synchronisations intra-flots. En effet, l'en-tête du contrat de QoS *inter* stipule qu'il doit contenir les contrats de QoS *video.intra* et *audio.intra*. Enfin les signaux a et v du composant *flotsSynchronises* sont connectés aux signaux S des sous-composants *audio* et *video*.

Les opérations de composition permettent de réutiliser les spécifications temporelles. Pour améliorer cette réutilisabilité, l'utilisateur peut exploiter des composants dits "génériques". Si nous regardons de plus près la forme des équations de QoS de notre exemple précédent, nous constatons que les membres de droite des équations du composants *flots-Synchronises* ne sont pas des constantes numériques comme c'est le cas pour ceux du composant *flot* : ce sont des variables (variables *alpha* et *beta*).

Le composant *flotsSynchronises* est ce que l'on appelle un composant générique. Un composant générique est un composant paramétrable grâce à des variables. Les variables peuvent être déclarées partout où la règle *donnee* est utilisée. En général, un composant générique est un composant qui modélise un comportement temporel courant dans un système multimédia. Il peut être instancié avec des valeurs quelconques pour modéliser un élément particulier du système. Les composants de l'annexe C.1 constituent une "bibliothèque" de composants génériques. Nous aurons l'occasion de les utiliser pour modéliser une application multimédia dans le chapitre 8.

Une fois les différents composants du système décrits, il est nécessaire de définir le graphe de flots de données qui modélise une application. La description du graphe de flots de données doit instancier les différents composants constituant le système multimédia et faire le lien avec le modèle objet de l'application.

La déclaration d'un graphe de flots de données est la suivante (règle *dfd*) :

```
dfd::= "system" ident ":" instances {instances} end

instances::= "instance" ident ":"
             types
             "component" ident ";"
             contrat
             parametres
             signauxInstance
             "end"

types::= "networkDevice" ident ident ";"
         | "localComponent" ident ";"

contrat::= "qoscontract" ident [ ident "/" portee
               {"," ident "/" portee} ] ";"

signauxInstance::= "signals" unSig {"," unSig} ";"

unSig::= ident "=" evenement

evenement::= ident "." ident "." ("IE" | "IR" | "RE" | "RR")

parametres::= "parameters" [ unParm {"," unParm} ] ";"

unParm::= portee "=" (supervision | [signe] entier [unite])

supervision::= services "." information "." filtre

services::= "network"

information::= "delay" | "jitter"

filtre::= "max" | "min" | "average"
          | ("weigthed" reel)
```

Chaque application définit donc un graphe de flot de données à partir de la règle *dfd*. Un graphe de flots de données comprend une liste d'instances. Chaque instance (définie par la règle *instances*) fournit les informations suivantes :

– Le nom du composant instancié (mot clef *component*).

- Le contrat de QoS choisi parmi ceux exportés par le composant instancié (mot clef *qoscontract*) accompagné d'éventuels renommages (lorsque plusieurs instances doivent partager des déclarations de QoS).

- Une liste optionnelle de paramètres (mot clef *parameters*). En effet, dans le cas où le composant instancié est un composant générique, il est nécessaire, à ce niveau, de préciser la valeur de toutes les informations paramétrées. La liste de paramètres consiste en une liste d'affectations (une affectation par paramètre). L'opérateur de portée peut être utilisé pour accéder aux paramètres encapsulés dans des sous-composants. Chaque paramètre peut être initialisé avec deux types d'informations : une constante numérique ou un service de supervision. L'utilisation de constantes numériques pour l'affectation des paramètres caractérise un composant dont le comportement est connu a priori ; à l'opposé, l'utilisation d'un service de supervision signifie que l'on ne dispose pas de cette information. Le choix d'un service de supervision s'effectue en tenant compte de deux critères (cf. règle *supervision*) :

  1. L'information que l'on souhaite obtenir (estimation de délai de bout en bout, de gigue, de taux de perte ?).
  2. Le filtre que l'on souhaite utiliser (cf. paragraphe 4.2 page 36). Les filtres disponibles à ce jour permettent de calculer une valeur moyenne, maximale, minimale ou pondérée (filtre pondéré proposé par Schulzrinne [BUS 96]). Il est clair que le choix d'un filtre parmi l'ensemble disponible dépend de l'application. L'implantation actuelle s'attache donc à faciliter l'ajout de nouveaux filtres plutôt qu'à offrir une collection de filtres, qui de toute façon ne serait jamais exhaustive.

- Le type de composant instancié (mot clef *type*). Il s'agit ici de préciser sur quelle machine le composant est situé. Dans le cas où le composant modélise un dispositif réseau, les noms du site émetteur et du site récepteur sont consignés. Cette information est exploitée par le gestionnaire de QoS pour construire les jeux d'équations de QoS sur chacune des machines du système (cf. algorithme du paragraphe 6.4.2).

- Enfin, il est nécessaire d'associer aux signaux du composant instancié les événements qui vont être observés lors des interactions entre objets (mot clef *signals*). Le lien entre le modèle objet de l'application et le graphe de flots de données est effectué à ce niveau.

Comme pour les composants, le lecteur trouvera dans l'annexe C plusieurs exemples de graphe de flots de données. Certains de ces graphes de flots de données utilisent les services de supervision. Les descriptions de QoS données dans cette annexe modélisent le comportement temporel d'applications multimédias qui sont décrites dans le chapitre 8.

7.3.3 Services d'adaptation de QoS disponibles dans POLKA

Les systèmes qui offrent des mécanismes d'adaptation de la QoS peuvent être classés en deux catégories : ceux qui prévoient un ensemble de spécifications de QoS dans lequel

l'application va évoluer et ceux pour qui cette évolution n'est pas prévue avant l'exécution de l'application.

Dans le premier cas, c'est le système qui décide de placer l'application dans l'un ou l'autre des niveaux de QoS selon un critère donné (exemples : maximisation de la satisfaction totale du système, minimisation du coût global, etc) [TOK 92, SIJ 96, FRY 96, BRA 98]. L'application est alors simplement avertie que le système vient de lui affecter un niveau de QoS donné et qu'elle doit modifier ses besoins en ressources.

Dans le deuxième cas, l'initiative revient à l'application qui, grâce à des informations de supervision, décide de revoir ses besoins en terme de QoS [CEN 95, DIO 95, BUS 96].

Le choix de l'un ou l'autre des mécanismes dépend essentiellement des caractéristiques de l'application. Aussi, POLKA n'impose pas de stratégie particulière quant à la méthode qui sera utilisée par une application pour modifier ses caractéristiques de QoS. Les mécanismes offerts par POLKA autorisent l'une ou l'autre des stratégies, et éventuellement une combinaison des deux.

7.3.3.1 Adaptation effectuée par le système POLKA : utilisation des déclencheurs

La première stratégie consiste, lors de la conception, à prévoir les différents niveaux de QoS associés à l'application. Pour ce faire, le développeur dispose dans POLKA d'événements "déclencheurs" (ou *trigger*). Un événement déclencheur est déclaré du sein d'un contrat de QoS (cf. règles *contrats* et *declencheur* du paragraphe 7.3.2). Un déclencheur propose pour une ou plusieurs équations une valeur du membre de droite à prendre en compte lors de l'occurrence d'un événement particulier. Cet événement peut être généré soit par le système soit par l'application (exemples : lancement d'un nouveau flot, invocation d'une méthode particulière, etc).

```
component vod :
        signals out video;
        qoscontract sousTitrage :
                common
                provided
                        eqos a T(video,n+1) - T(video,n)< 40 ms ;
                        eqos b T(video,n+1) - T(video,n)> 40 ms ;
                        trigger debutSousTitrage a=50, b=50;
                        trigger debutSousTitrageMalEntendant a=80, b=80;
                        trigger finSousTitrage a=40, b=40;
                        trigger finSousTitrageMalEntendant a=40, b=40;
                required
        end
end
```

La déclaration ci-dessus donne un exemple d'utilisation de déclencheurs (un second exemple est présenté dans le paragraphe 8.2.2). Imaginons un système de vidéo à la de-

mande qui autorise l'utilisation du sous-titrage. Comme sur les disques DVD (*Digital Versatil Disk*), il est possible de présenter deux types de sous-titrages : un sous-titrage "normal" et un second plus complet pour les personnes mal-entendantes. On souhaite adapter la QoS offerte sur le flot vidéo en fonction de la présence ou non des sous-titrages. Le composant *vod* comprend deux équations (a et b) qui spécifient les contraintes de présentation du flot vidéo. Parallèlement à la présentation des données, l'application va générer quatre événements lors du lancement ou de l'arrêt d'un sous-titrage (événements déclenchés par l'utilisateur). Les événements en question sont $debutSousTitrage$, $finSousTitrage$, $debutSousTitrageMalEntendant$ et $finSousTitrageMalEntendant$. Les déclencheurs du composant ci-dessus spécifient donc qu'à l'occurrence d'un de ces quatre événements, les valeurs des équations a et b doivent être modifiées. Ainsi, le système utilisera, selon les cas, les jeux d'équations suivants :

1. Absence de sous-titrage :

```
eqos a T(video,n+1) - T(video,n)< 40 ms ;
eqos b T(video,n+1) - T(video,n)> 40 ms ;
```

2. Sous-titrage "normal" :

```
eqos a T(video,n+1) - T(video,n)< 50 ms ;
eqos b T(video,n+1) - T(video,n)> 50 ms ;
```

3. Sous-titrage pour mal-entendants :

```
eqos a T(video,n+1) - T(video,n)< 80 ms ;
eqos b T(video,n+1) - T(video,n)> 80 ms ;
```

7.3.3.2 Adaptation par l'application

Les déclencheurs permettent de prévoir des scénarios d'adaptation de QoS efficaces puisque dans ce cas de figure, un changement de QoS se réduit à une opération en $O(1)$ (une affectation de variable et un test pour être précis).

Lorsque des solutions plus flexibles sont nécessaires, le développeur dispose des méthodes exportées par le gestionnaire de QoS. Ainsi, il est possible de construire ou modifier des descriptions de QoS en mémoire, puis, de les charger dans l'ordonnanceur (avec la méthode $addQosFromBuffer$ par exemple). Cette solution est plus onéreuse puisque, dans la majorité des cas, il est nécessaire de dérouler l'algorithme décrit dans le paragraphe 6.4.2. Avec cette solution, toutes les informations de QoS sont modifiables et en particulier :

- Modification de la définition d'un composant (en ajoutant, supprimant ou modifiant des équations).
- Changement de paramètres dans les instances de composants génériques (exemples : période d'horloge, membre de droite d'une équation, etc).

– Substitution de composant dans un graphe de flots de données (exemple : changer un composant où les paramètres sont supervisés par un composant dont les paramètres sont connus a priori).

– Changement de filtre sur des données supervisées. Notons que le seul fait d'utiliser les fonctions de supervision dans la description d'un composant constitue déjà une façon d'adapter la QoS d'une application par le choix des filtres associés aux informations supervisées.

7.4 Synchronisation et ordonnancement des threads POLKA

Nous terminons ce chapitre par la description des techniques qui sont utilisées dans la plate-forme pour ordonnancer et synchroniser les threads POLKA.

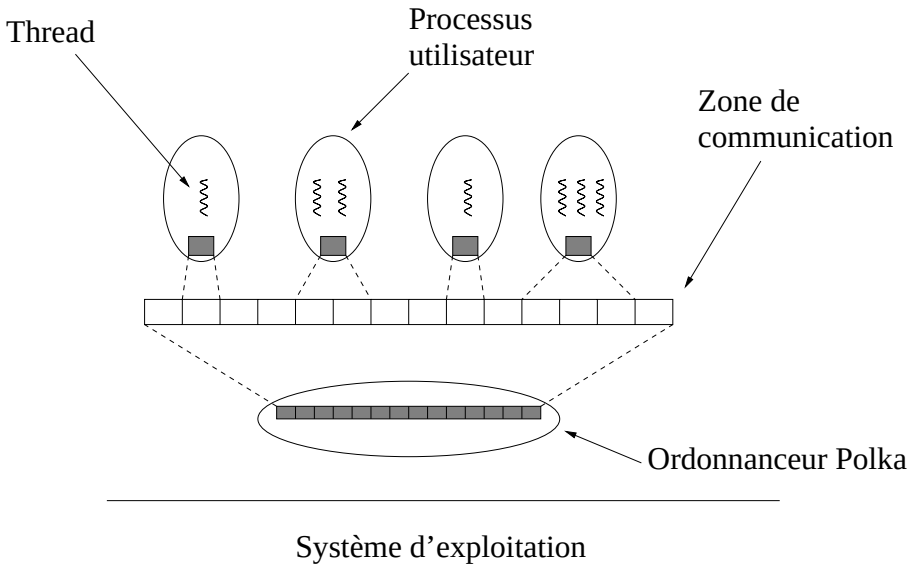


FIG. 7.5 – *Communication entre ordonnanceur et applicatifs*

Pour synchroniser les threads d'un système POLKA, nous utilisons le schéma des sémaphores privés [KRA 85]. Sur chaque site, il existe une zone de communication entre les différents processus utilisateurs et le démon POLKA. Outre le partage d'informations tel que l'état des threads, cette zone de communication comprend, pour chaque thread, un sémaphore initialisé à zéro. Ce sémaphore permet de bloquer le thread lorsque l'ordonnanceur de POLKA décide de ne pas lui attribuer le processeur. L'opération bloquante sur le sémaphore privé est effectuée dans la méthode *schedule* que le thread utilise lorsqu'il demande l'exécution d'une tâche à l'ordonnanceur POLKA.

Compte tenu de nos objectifs de portabilité, si ce schéma n'est pas forcément le plus efficace, il a l'avantage d'être un moyen très fiable pour bloquer et réveiller les threads du système. En effet, il existe des différences parfois notables sur l'ordonnancement des

threads d'un système à un autre. Notre implantation suppose néanmoins que le système d'exploitation, en plus d'offrir un ordonnanceur temps partagé, ait les caractéristiques suivantes :

1. Le système doit offrir un ordonnanceur préemptif à priorités fixes.
2. Une file d'attente doit être associée à chaque niveau de priorité. La politique de gestion de cette file d'attente doit permettre à tous les threads de même priorité de progresser ensemble.
3. Le système doit offrir une abstraction de thread. De plus, le système doit autoriser les threads à utiliser l'ordonnanceur temps réel.

Notons que la norme POSIX remplit ces trois caractéristiques.

7.4.1 Caractéristique (1) : choix des priorités pour les tâches du système

L'ordonnanceur système doit être préemptif à priorités fixes. La plate-forme POLKA associe une priorité aux tâches du système afin de garantir aux tâches POLKA les ressources processeurs (en particulier lorsque des applications temps partagé les côtoient). Si l'on suppose que l'ordonnanceur système considère le niveau de priorité 0 comme le niveau le plus prioritaire, alors :

- La priorité 0 est associée aux démons importants du système (exemples : nfsd, portmap, etc) ainsi qu'au serveur X11 et à un interpréteur de commandes (shell). Ce dernier permet de prendre la main sur le système lorsqu'une application POLKA hors service monopolise les ressources.
- La priorité 1 est allouée aux threads du démon POLKA (et donc à l'ordonnanceur POLKA) ainsi qu'aux threads des applications qui peuvent créer de nouveaux threads POLKA (exemple : les interfaces hommes-machines).
- Enfin la priorité 2 est donnée à tous les threads POLKA.

Dans les environnements non dédiés que nous ciblons, il est très courant qu'un utilisateur fasse cohabiter en "même temps" des applications sous contraintes temporelles (exemple : un navigateur web qui visionne un film au format MPEG) avec des applications sans contrainte temporelle (exemple : une compilation). Dans ce cas de figure, l'utilisateur souhaite que les contraintes temporelles de son application soient respectées tout en garantissant la progression de ses applications temps partagé (exemple : il souhaite que sa compilation progresse de façon satisfaisante pendant qu'il regarde son film).

Il est donc important d'offrir des solutions pour faire cohabiter efficacement des applications avec et sans contraintes temporelles [MER 94, NIE 97, JON 97]. L'utilisation des priorités que nous préconisons pour POLKA ne garantit pas de ressources processeurs aux applications temps partagé (applications non POLKA). Il est toutefois possible d'utiliser une technique proche des serveurs sporadiques [SPR 89] en créant une tâche POLKA fictive qui se chargera d'exécuter les applications temps partagé.

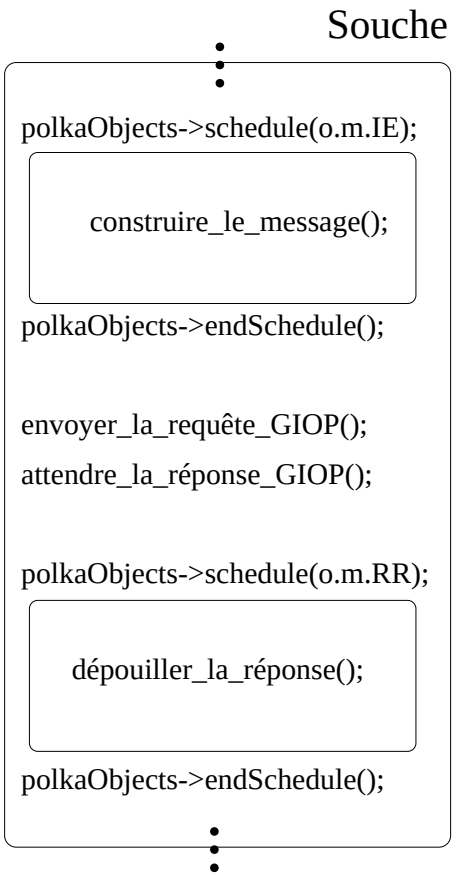


FIG. 7.6 – *Invocation distante d'une méthode*

7.4.2 Caractéristique (2) : gestion des files d'attente

Dans la grande majorité des cas, l'ordonnanceur POLKA maintient un seul thread éveillé, mais il existe des situations où plusieurs threads peuvent être éveillés simultanément. Pour cette raison, la plate-forme POLKA suppose l'existence dans le système d'une politique de gestion des files d'attente adaptée. Pour illustrer notre propos, nous avons représenté dans la figure 7.6 une souche CORBA instrumentée par *i2p*. La souche est découpée en deux tâches encadrées par les méthodes *schedule*() et *endSchedule*(). Sont exclues de ces tâches l'émission de la requête sur le réseau et la réception de la réponse. En effet, entre l'émission de la requête et la réception de la réponse, un temps important peut s'écouler ; temps pendant lequel la tâche va rester bloquée sur un appel système (une opération *read*() sur une socket TCP). Durant cette période, l'ordonnanceur système va donc allouer le processeur à une tâche non POLKA (donc à une tâche, qui, a priori, ne possède pas de contrainte temporelle). Pour éviter ce phénomène, l'ordonnanceur POLKA va permettre à deux threads POLKA de s'exécuter en parallèle pendant un court laps de temps. En effet, lors de l'invocation de la méthode *endSchedule* qui signale la fin de la

tâche $o.m.IE$, une seconde tâche va être réveillée (élue selon la politique EDF). Dans le cas général, l'invocation de la méthode *endSchedule* est directement suivie de l'invocation de méthode *schedule*. De ce fait, le thread qui vient de terminer une tâche se bloque sur son sémaphore privé. Ici, la phase de communication est intercalée entre la méthode *endSchedule* et *schedule*, permettant ainsi à deux threads POLKA de s'exécuter en parallèle. A la réception de la réponse GIOP, il est important que le système d'exploitation réveille la tâche bloquée sur la socket afin qu'elle puisse avertir l'ordonnanceur de son souhait d'exécuter la tâche $O.m.RR$ et de finalement se bloquer sur son sémaphore privé ; d'où l'importance du choix de la politique de gestion des files d'attente de l'ordonnanceur système **puisse qu'elle doit garantir la progression régulière de toutes les tâches** POLKA**.**

L'invocation de la méthode distante pose en fait un problème classiquement rencontré dans toutes les bibliothèques de threads utilisateurs : le support des appels systèmes bloquants [MUE 93]. Dans ces environnements, les appels systèmes bloquants sont remplacés par des appels asynchrones. On peut s'interroger sur la pertinence d'un tel mécanisme pour les appels systèmes bloquants qu'un thread POLKA peut être amené à invoquer. Compte tenu du surcoût important d'une interaction avec le démon POLKA, le temps de blocage doit être suffisamment long pour qu'une opération de réordonnancement soit rentable. Hormis les phases de communication, il existe toutefois des services où cette technique peut être exploitée ; c'est notamment le cas des sémaphores (sémaphores partagés par plusieurs threads POLKA pour gérer un tampon par exemple).

La norme POSIX propose une politique de gestion de file d'attente qui nous convient [GAL 95]. De ce fait, parmi les systèmes d'exploitation généralistes que nous ciblons, nombreux sont ceux qui offrent la caractéristique (2). Dans ces systèmes, il y a généralement plusieurs politiques de gestion de la file d'attente. Parfois, le système d'exploitation permet au développeur d'intégrer ses propres politiques ; c'est le cas de Chorus [ROZ 91]. POSIX propose deux politiques : SCHED_FIFO et SCHED_RR. SCHED_FIFO consiste en une gestion de type premier entré, premier sorti. La tâche en tête de file acquiert le processeur lorsque celui-ci est libéré. Elle le conserve jusqu'à sa terminaison ou jusqu'au moment où elle est endormie par le système (pendant une opération d'entrée/sortie par exemple). Lorsqu'une tâche est réveillée, elle est placée en fin de liste. La politique SCHED_FIFO ne garantit donc pas la progression régulière de toutes les tâches de même priorité.

La politique SCHED_RR offre une solution à ce problème. SCHED_RR offre des services qui sont proches de SCHED_FIFO, mais une notion de quantum est ajoutée pour la gestion de la liste. Les règles de gestion de la file pour la politique SCHED_FIFO s'appliquent à la politique SCHED_RR mais, cette fois, quels que soient les traitements effectués par une tâche, celle-ci ne peut conserver le processeur qu'un temps inférieur ou égal à la valeur du quantum. Lorsque une tâche a consommé le quantum, elle libère le processeur et est placée en fin de liste.

D'un système à un autre, il existe souvent de petites différences, qui au final, peuvent modifier considérablement l'ordonnancement généré. Ainsi, Solaris, Linux et Windows NT offrent une politique de type SCHED_RR. Cependant, sur Solaris, le quantum est paramétrable pour chaque thread noyau alors que sur Linux et Windows NT [SOL 98] le

quantum est le même pour tous les threads et est non modifiable. Malgré tout, pour que notre ordonnanceur puisse être porté sur un système d'exploitation, nous n'imposons pas de contrainte particulière sur la valeur du quantum. **Notre solution nécessite simplement la présence d'une politique de type SCHED_RR.**

7.4.3 Caractéristique (3) : utilisation des threads du système

Pour construire notre abstraction de thread "à la Clouds", POLKA a besoin que le système d'exploitation exporte une abstraction de thread. Dans les systèmes d'exploitation actuels, il existe trois classes de threads [DEM 94] :

- Les threads utilisateurs. Dans ce cas de figure, l'abstraction de thread n'existe pas dans le système d'exploitation. Les threads sont implantés sous la forme d'une bibliothèque. Cette solution est théoriquement très efficace puisqu'une commutation de contexte entre deux threads revient à un coût proche d'un appel de fonction. De plus, les systèmes de threads utilisateurs sont très flexibles : la création d'un nouvel algorithme d'ordonnancement nécessite la reconstruction de la bibliothèque mais pas une modification du système d'exploitation. Cette solution possède toutefois un inconvénient majeur : l'ordonnanceur du système ne possède aucune information sur les threads utilisateurs. Pour garantir l'obtention des ressources processeurs, il est nécessaire que notre système puisse hiérarchiser l'ensemble des tâches du système ; l'utilisation de threads utilisateurs est donc à exclure puisque la notion de priorité n'est valide qu'au sein d'un processus.

- Les threads noyaux. Cette fois, le thread est une abstraction connue par le système d'exploitation ; les threads sont donc ordonnancés par le système et il est possible de hiérarchiser l'ensemble des threads du système par des priorités. Toutefois, en principe, les threads noyaux sont moins efficaces que les threads utilisateurs. En effet, les commutations de threads noyaux et les opérations de synchronisations (sémaphores, barrières, etc) nécessitent une interaction entre l'espace utilisateur et l'espace noyau qui n'existe pas avec les threads utilisateurs. Enfin les threads noyaux sont moins flexibles puisque l'adjonction d'un nouvel ordonnanceur nécessite presque toujours la modification du noyau.

- La troisième solution est une solution hybride qui propose un schéma d'ordonnancement à deux niveaux [GOV 91, AND 92, COU 95]. Dans ces solutions, deux classes d'ordonnanceur sont présentes : l'ordonnanceur système et des ordonnanceurs utilisateurs. Cette solution nécessite la présence de mécanismes spécifiques pour la communication d'informations entre les ordonnanceurs utilisateurs et l'ordonnanceur système. Néanmoins, elle permet de bénéficier de l'efficacité et de la flexibilité des threads utilisateurs, tout en offrant la possibilité d'exporter des informations d'ordonnancement vers l'ordonnanceur du système, et ainsi de pouvoir hiérarchiser l'ensemble des threads utilisateurs du système. Malheureusement, les ordonnanceurs à deux niveaux sont peu répandus dans les systèmes d'exploitation que nous ciblons. Compte tenu de notre objectif de portabilité, nous écartons donc cette solution.

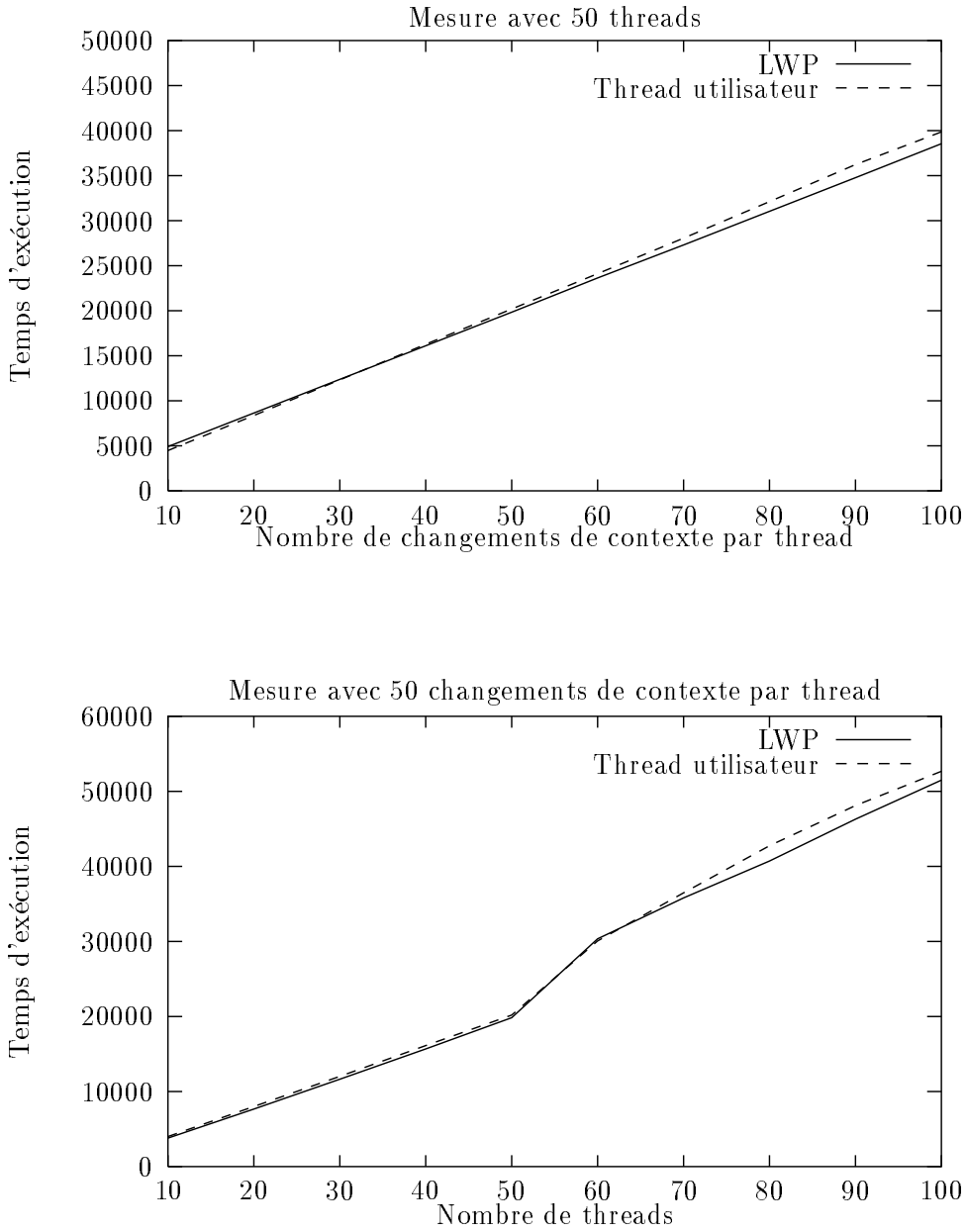


FIG. 7.7 – *Comparaison entre threads noyaux et threads utilisateurs sur Solaris 2.5*

Notre solution nécessite donc l'utilisation de threads noyaux ; même si celle-ci conduit à une efficacité généralement moindre que l'utilisation de threads utilisateurs [AND 92]. Notons que dans la réalité, il faut rester prudent quant à cette affirmation. A titre d'exemple, les courbes 7.7 montrent deux mesures effectuées sur une station Sparc1+ avec Solaris 2.5. Il s'agit du temps de réponse en micro-secondes d'un processus qui, dans un premier cas, exécute x LWP (le LWP est un thread noyau sur Solaris) et qui, dans le second cas, exécute x threads utilisateurs multiplexés sur un seul LWP. Le code de chaque thread est le suivant :

```
for(int i=0;i<nbChangementContexte;i++)
   {
      cout<<"thread id : "<<thr_self()<<endl;
      thr_yields();
   }
```

L'appel système *thr_yields*() avertit l'ordonnanceur que le thread actuellement en cours d'exécution souhaite libérer le processeur. Ce code provoque un nombre important de changements de contexte. Dans la courbe du haut de la figure 7.7, on fait varier le nombre de changements de contexte effectués par chaque thread. Dans la courbe du bas, on fait varier le nombre de threads x. On constatera que la différence entre le temps de réponse de x LWP et x threads sur un seul LWP est négligeable. Ce qui signifie que, dans Solaris, il n'existe pas une grande différence de coût entre un changement de contexte d'un thread noyau et un changement de contexte d'un thread utilisateur.

Enfin, il arrive que le choix entre threads noyaux et threads utilisateurs s'impose autrement que par des raisons de performances. Ainsi, les threads les plus diffusés avec Linux sont des threads noyaux basés sur l'appel système *_clone*().

7.5 Conclusion

Ce chapitre décrit de façon détaillée la mise en œuvre de notre ordonnanceur et des outils qui lui sont associés. Les solutions techniques choisies dans notre plate-forme ont pour but d'augmenter la portabilité des applications (choix d'un bus à objets CORBA, utilisation de systèmes d'exploitation banalisés, couche d'adaptation, etc).

Si pour l'environnement de développement, ces choix ont permis de faciliter la réalisation de la plate-forme, la mise en œuvre de notre ordonnanceur, dans ce cadre, implique des conséquences importantes en terme d'efficacité.

En effet, nous verrons dans le chapitre suivant, qu'une grande part du surcoût induit par notre ordonnanceur est constitué d'opérations de synchronisation. Il est clair que si les systèmes d'exploitation actuels étaient plus ouverts, (exemple : possibilité d'utiliser ou de construire des ordonnanceurs à double niveaux), le surcoût de nos propositions, et de toutes celles utilisant des services similaires, serait grandement diminué (voire négligeable). Néanmoins, nous verrons dans le chapitre 8, que les performances de notre plate-forme restent raisonnables compte tenu des application visées.

Chapitre 8

Exemples d'application et expérimentations

Dans ce chapitre, nous allons décrire trois applications que nous avons développées afin de valider certains aspects de la plate-forme POLKA.

La première application est une application centralisée qui manipule des images animées au format GIF. Nous décrirons, en particulier, comment cette application utilise la plate-forme POLKA pour adapter son comportement aux demandes de l'utilisateur.

La deuxième application manipule des flots MPEG. Nous montrerons comment construire une spécification de QoS de bout en bout, et quelles en sont les conséquences sur la gestion des ressources. Avec cette application, nous illustrerons l'utilisation de la seconde technique d'adaptation offerte par POLKA : les événements déclencheurs. Cette application est, elle aussi, centralisée.

Enfin, nous utiliserons une troisième application, qui simule la seconde, mais dans un environnement réparti. Nous évaluerons le surcoût engendré par POLKA et sa capacité à respecter les synchronisations spécifiées. Nous verrons, à cette occasion, que les seuls éléments qui différencient une application POLKA centralisée d'une application POLKA répartie se situent dans la spécification de QoS. Pour répartir une application POLKA il n'est donc pas nécessaire de la modifier ou de la recompiler.

8.1 Premier exemple : une application centralisée simple

Le format GIF [COM 89] est un format d'image très utilisé sur le web. Un fichier GIF peut contenir plusieurs images qui constituent alors une séquence d'animation. Dans ce cas de figure, le fichier contient la spécification d'une synchronisation intra-flot. La synchronisation intra-flot est exprimée par le biais d'une contrainte temporelle particulièrement simple puisqu'il s'agit d'une cadence d'affichage des images exprimée en $100^{èmes}$ de secondes.

L'application présentée ici permet de visionner ce type de fichier de façon synchronisée (cf. figure 8.1). Au lieu d'exploiter la synchronisation spécifiée dans le fichier, l'outil propose un ascenseur permettant de régler la cadence d'affichage grâce à une horloge logique.

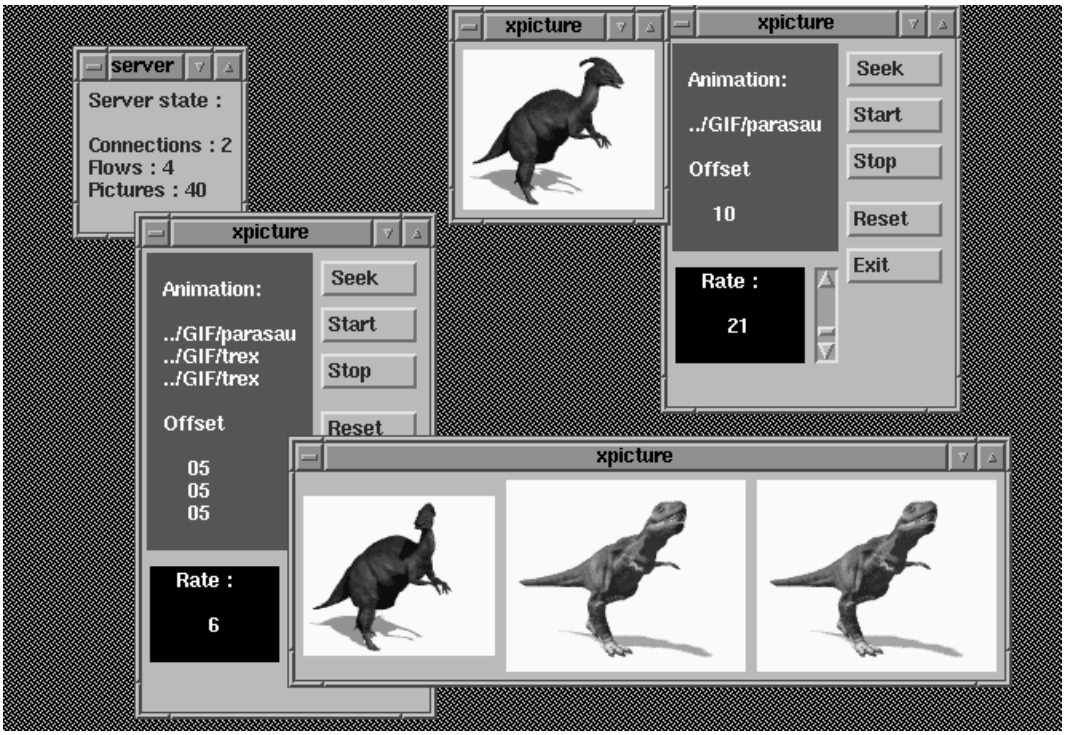


FIG. 8.1 – *Première application*

L'application est constituée d'un serveur d'animation et d'un ou plusieurs clients (cf. figure 8.2). Le client est un processus lancé par la commande UNIX *xpicture*. Chaque lancement d'un client instancie deux fenêtres : une fenêtre de contrôle qui permet de démarrer l'animation, de la suspendre, etc ; une fenêtre de présentation qui contient les animations. Un client peut demander la présentation de plusieurs animations. Les différentes animations d'une fenêtre sont synchronisées. La figure 8.1 montre deux clients demandant respectivement une et trois animations.

8.1.1 Spécification fonctionnelle de l'application

La spécification fonctionnelle de cette application est simple : pour chaque animation, le serveur crée un objet de type *pictureFlow* (cf. interface IDL de la figure 8.3). Lors de la création de cet objet, le serveur charge l'ensemble des images composant l'animation. Au démarrage d'un client, on va donc créer autant d'objets *pictureFlow* que d'animations demandées par le client. En terme de thread, chaque client contient autant de threads qu'il y a d'animations. Les threads bouclent sur l'invocation de la méthode *displayNextPicture* pour présenter la séquence d'images de l'objet correspondant. En effet, lorsque la méthode *displayNextPicture* est invoquée, le serveur d'animation demande au serveur X11 d'afficher l'image suivante de l'animation.

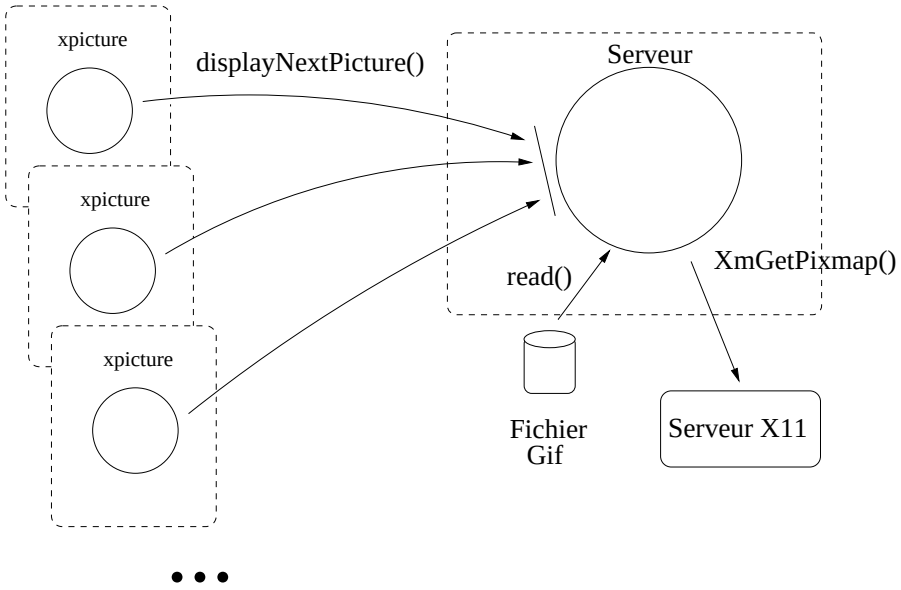


FIG. 8.2 – *Modèle objets de l'application*

8.1.2 Une spécification de QoS simple

Cette application est simple. Nous allons décrire comment nous construisons une spécification de QoS dans POLKA pour une application de ce type.

La première étape consiste à isoler les flots de données et les composants qui les manipulent. Les flots de données sont, bien sûr, les flots d'images de nos animations. Ces dernières transitent du serveur d'animation vers le serveur X11. Nous avons donc un flot de données par animation. Un graphe de flots de données possible pour cette application est donné dans la figure 8.4. Dans ce graphe, nous avons modélisé chaque client par un sous-composant du serveur d'animation ; ces sous-composants sont alors eux-même affinés

```
module gif {
    interface pictureFlow {
        void seek(in long o)
            raises (invalidOffset);
        void displayNextPicture();
        void reset();
    };
};
```

FIG. 8.3 – *Interface* IDL *de l'application*

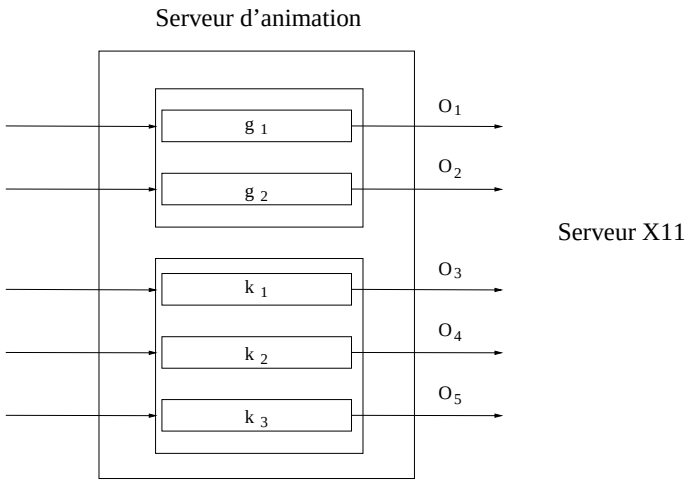


FIG. 8.4 – *Graphe de flots de données de l'application*

en sous-composants plus fins qui modélisent chacun un objet *pictureFlow* (ainsi, la figure 8.4 représente deux clients avec respectivement deux et trois animations).

Dans un premier temps, nous spécifions le composant de base qui modélise une animation et qui comprend une synchronisation intra-flot. Ce composant est constitué d'une horloge qui va cadencer l'affichage des images et de deux équations qui vont asservir les tops de l'horloge au signal de sortie (le signal de sortie modélise ici l'affichage des images d'une animation).

Le composant gif de l'annexe C.2 spécifie un composant de base. Le mot clef *clock* permet de déclarer l'horloge logique utilisée pour la synchronisation intra-flot (la période initiale de l'horloge est de 1000 ms). Les équations du contrat *intra* modélise la synchronisation intra-flot de l'animation. On s'autorise ici à une gigue maximale de 20 ms.

Une fois le composant de base spécifié, on peut l'utiliser pour modéliser la synchronisation des animations d'un même client (synchronisation inter-flots). Pour ce faire, on va combiner plusieurs composants gif, puis, ajouter les équations de synchronisation inter-flots. Si l'on effectue cette composition avec deux composants gif, on obtient un composant de la forme du composant $synchronizedGif$ de l'annexe C.2. Cette approche illustre bien les capacités de modularité (et donc de réutilisabilité) de notre modèle. Notons que l'on aurait pu pousser plus loin la réutilisabilité en spécifiant des composants génériques.

Finalement, la spécification de QoS se termine par la définition du graphe de flots de données qui, ici, est réduit à sa plus simple expression : un nœud par client. On instancie donc un composant $synchronizedGif$ pour chaque client présent dans le système. Le lien avec le modèle objet est alors trivial, puisqu'à chaque composant gif correspond un objet de type $pictureFlow$. Dans cette application, les clients utilisent la méthode $displayNextPicture$ pour demander au serveur d'animation l'affichage de l'image suivante d'un flot. Il est alors possible de lier le signal de chaque composant gif à l'événement $flow.displayNextPicture.RR$ (ou $flow$ est l'objet de type $pictureFlow$ associé

à un client). L'annexe C.2 donne le graphe de flots de données ainsi que l'ensemble des équations de QoS calculées par *qc* ou le gestionnaire de QoS. Le graphe de flots de données de l'annexe instancie un composant $synchronizedGif$, modélisant ainsi un client qui manipule deux animations.

Bien sûr, la spécification de QoS que nous venons de donner n'est pas unique. Si le concepteur souhaite d'autres synchronisations, il est libre de construire une spécification différente. Ceci s'effectue sans modifier le code de l'application. En effet, la seule information présente dans le code de l'application est l'existence d'une horloge logique pour chaque flot, mais rien n'oblige le concepteur à spécifier des contraintes temporelles portant sur celle-ci.

Notons que, dans cet exemple, nous nous sommes contentés de décrire notre système multimédia par deux composants : nous avons estimé, compte tenu du comportement temporel des éléments du système impliqués par notre application, que la spécification de la QoS utilisateur était suffisante. En effet, notre système ne possède pas de composant ayant un comportement temporel particulier. Nous allons voir dans la partie suivante, qu'il n'est pas toujours possible de construire des spécifications aussi simples et que parfois, il est nécessaire de décrire le système de bout en bout.

8.2 Deuxième exemple : modélisation de bout en bout d'une application

FIG. 8.5 – *Deuxième application*

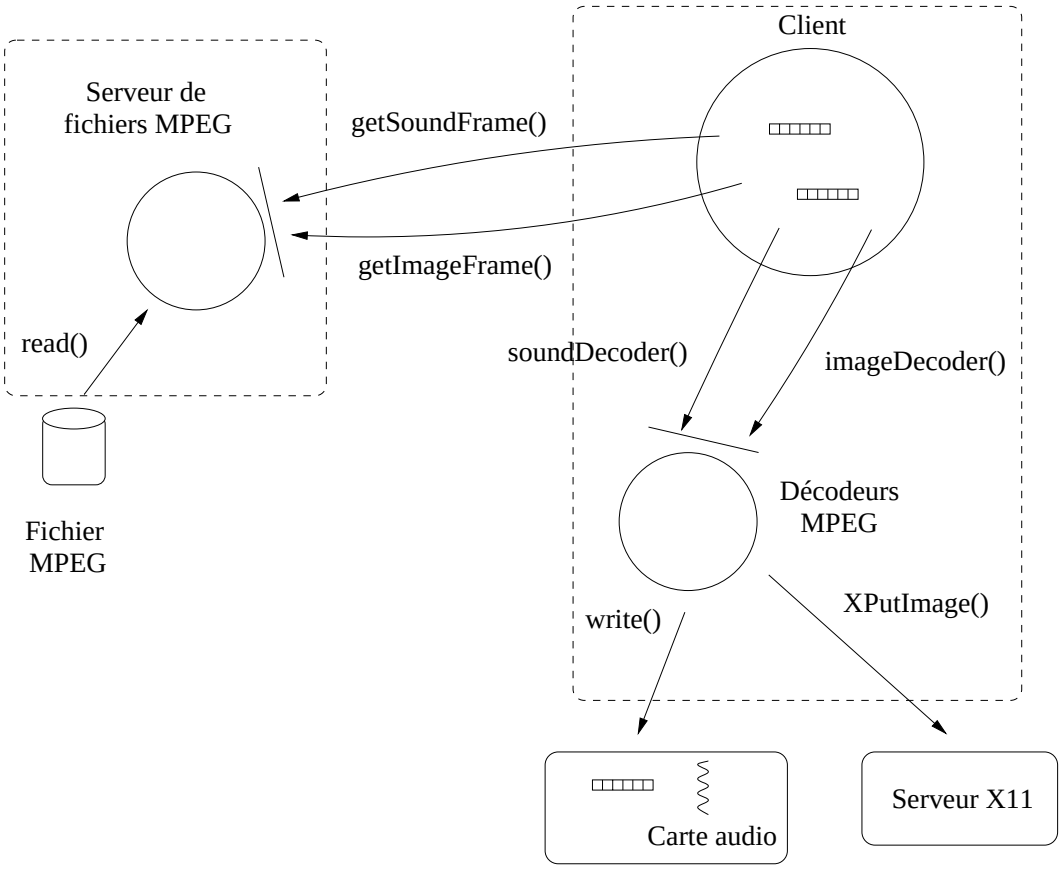


FIG. 8.6 – *Modèle objets de l'application*

8.2.1 Description de l'application étudiée

L'application utilisée ici est une simulation de la cérémonie des Césars (cf. figure 8.5). L'utilisateur visionne la cérémonie et a la possibilité de voter pour un film en compétition. Il peut aussi voir des extraits de films. Les films sont compressés au format MPEG. A un instant donné, il est donc possible que plusieurs flots MPEG soient en cours de lecture. Dans notre exemple, il y a quatre extraits disponibles : le nombre de flots vidéo est donc borné à cinq. Compte tenu de leurs besoins, le nombre de flots interdit une réservation au pire cas des ressources.

La figure 8.6 montre la découpe en objets de cette nouvelle application. On distingue trois objets : un client, un serveur de fichiers MPEG2 et un décodeur MPEG2 (audio et vidéo). La figure 8.7 décrit les interfaces IDL des deux objets.

Un objet client offre les services nécessaires à la restitution d'un film MPEG2[1] (audio et vidéo). Il est constitué de deux couples de threads.

[1] Dans la démonstration, les flots sont démultiplexés.

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

FIG. 8.7 – *Interface* IDL *de l'application*

A chaque flot est associé un tampon et deux threads. Les tampons permettent d'absorber la gigue du système de gestion de fichiers. Les threads produisent et consomment des éléments dans le tampon.

Ainsi pour la vidéo (respectivement pour l'audio), un thread remplit le tampon en bouclant sur l'invocation de la méthode $getImageFrame$ (respectivement $getSoundFrame$) à l'interface du serveur de fichiers MPEG2 (interface $mpegServer$). Chaque invocation rapatrie plusieurs trames afin d'amortir l'important surcoût fixe induit par CORBA. Un deuxième thread consomme les trames du tampon et boucle sur l'invocation de la méthode $imageDecoder$ (respectivement $soundDecoder$) de l'interface $mpegDecoder$. Les méthodes de cette interface prennent une trame, la décodent, puis la délivrent au périphérique de sortie.

Il est important de noter que toutes les méthodes CORBA sont synchrones. En d'autres termes, le thread qui consomme une trame vidéo ne peut en consommer une nouvelle tant que le traitement de la précédente n'est pas terminé. Dans le cas de la vidéo, il est difficile de prédire exactement quand l'image est réellement affichée dans la fenêtre puisque l'affichage final dépend du serveur X11 ; l'activité n'attend pas la confirmation de l'affichage du serveur X11. Pour le flot audio, la livraison de la trame décodée consiste à écrire les données dans le tampon du pilote de la carte audio. Les échantillons audio sont ensuite restitués sur le périphérique de manière autonome. La fin d'exécution de la méthode $soundDecoder$ ne signifie donc pas la fin de la restitution des échantillons audio, mais simplement que l'écriture de la trame dans le tampon est terminée.

Cette application est exécutée sur une machine Sun Ultra-sparc 1 cadencée à 167 MHz dotée d'une carte audio *dbri* et de 128 Mo de mémoire vive. Cette machine utilise le système d'exploitation Solaris 2.5. Certaines des informations données dans ce paragraphe sont spécifiques soit aux périphériques et logiciels utilisés, soit aux flots multimédias manipulés (ou en d'autres termes, aux caractéristiques des fichiers MPEG). Les mesures et les informations sur les flots multimédias (mesures quantitatives et temporelles) données

dans la suite de ce paragraphe sont produites à partir d'un fichier MPEG de test dont les caractéristiques sont les suivantes :

- Fichier vidéo : fichier MPEG 2 compressé en débit variable, constitué de 6650 images d'une taille de 352*288 pixels et générant un débit de 0,923 Mbits/s. La séquence vidéo est encodée à un rythme de 25 images par seconde. La taille du fichier est d'environ 40 méga octets. Le temps moyen pour décoder et afficher une image est de $55,24\ ms$.
- Fichier audio : fichier MPEG 2 layer 2. Données audio échantillonnées à 44,1 KHz (et donc générant un débit de 192 Kbits/s). La taille du fichier est d'environ 6,5 méga octets. Le temps moyen pour décoder une trame audio est de $2,11\ ms$.

Les décodeurs encapsulés dans notre application sont basés sur le décodeur audio *maplay* 1.2 de Tobias Bading (université de Berlin) et sur le décodeur vidéo *mpeg_play* 2.3 de l'université de Berkeley [ROW 93].

8.2.2 Un modèle de QoS simple : problèmes posés

Maintenant que nous avons décrit les aspects fonctionnels de notre application, nous pouvons aborder ses aspects qualitatifs. Avec cette application, nous validons deux fonctionnalités :

- L'utilisation des déclencheurs pour adapter la QoS d'une application.
- La capacité de POLKA à exprimer et traiter des synchronisations fines telles que la synchronisation voix-lèvres.

Pour ce faire, nous exécutons deux fois l'application. La première fois avec la spécification de l'annexe C.3.1.1 pour valider le fonctionnement des déclencheurs. La seconde fois avec la spécification de l'annexe C.3.1.2 pour valider la synchronisation voix-lèvres.

Comme pour l'application de GIF animé, l'application MPEG est définie comme un unique composant sur lequel nous allons placer des contraintes utilisateurs dans le contrat de QoS offerte, et ce sans nous soucier des autres composants du système.

La première spécification ne s'intéresse pas à la présentation du flot audio. En associant le mot clef $infinity$ au membre de droite de la seconde équation (cf. annexe C.3.1.1), on force l'ordonnanceur à calculer une échéance de valeur $+\infty$ pour les tâches effectuant la présentation des flots audio, ce qui a pour effet de ne jamais les ordonnancer.

Cette spécification fixe un délai maximum entre l'affichage des images des différents flots. Lors du démarrage de l'application, un seul flot vidéo est présent. Puis, à chaque fois que l'utilisateur le demande, un nouveau flot est lancé (dans la limite des quatre extraits proposés). Les déclencheurs sont utilisés pour faire basculer sur le dernier film demandé plus de ressources que sur les autres films. Les événements qui sont utilisés à cet effet sont ceux générés lors de la création de nouveaux flots vidéo (événements $newMovie1$, $newMovie2$, etc). A l'occurrence de tels événements, les déclencheurs spécifient alors une

cadence d'affichage élevée pour le nouveau flot et un ralentissement de la cadence des flots déjà existants.

La deuxième spécification (cf. annexe C.3.1.2) modélise les synchronisations intra-flots et inter-flots d'un film par trois équations. La première équation définit la contrainte intra-flot sur la vidéo, la seconde la contrainte intra-flot sur l'audio et la troisième la synchronisation voix-lèvres.

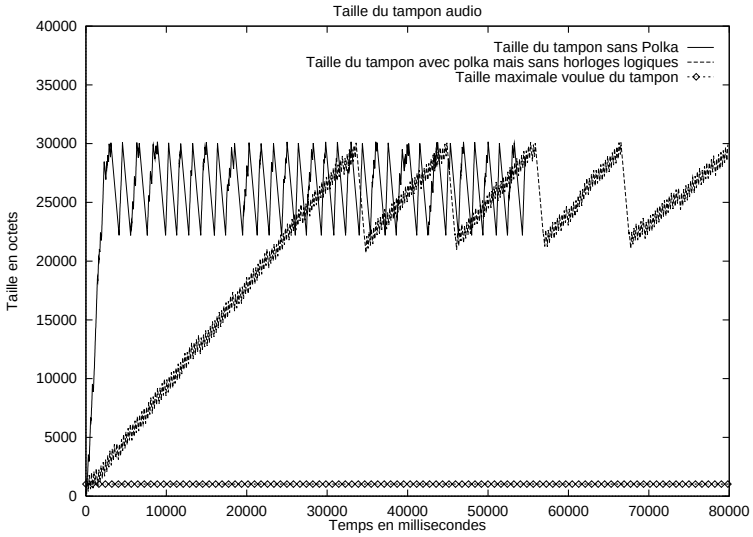


FIG. 8.8 – *Taille du tampon de la carte audio*

L'ordonnancement obtenu respecte les équations de QoS sous réserve de ressources suffisantes. Avec la première spécification de QoS, si plusieurs films sont exécutés simultanément, l'utilisateur peut, par le biais des déclencheurs, favoriser le dernier film lancé. Le premier test est donc concluant.

Toutefois, lorsque l'on utilise la deuxième spécification et que l'on évalue la synchronisation voix-lèvres, on peut s'interroger sur la pertinence de cette spécification à modéliser efficacement l'application ainsi que ses synchronisations. Si une telle spécification est utilisée, le comportement de l'application lors de son exécution n'est pas totalement satisfaisant :

- La synchronisation voix-lèvres est assurée au début de la restitution des flots mais elle dérive lentement pour être finalement perdue après quelques minutes d'exécution.

- Lorsque l'utilisateur suspend le film, une latence peut être observée : le flot vidéo est stoppé de façon quasi instantanée alors qu'il est nécessaire d'attendre 3 à 4 secondes pour que le flot audio cesse d'être restitué sur les hauts-parleurs. Cette latence est due à un surplus d'informations contenu dans le tampon de la carte audio. La figure 8.8 montre une estimation de la taille de ce tampon durant l'exécution de l'application. Sans POLKA le thread qui décompresse les trames audio remplit rapidement ce

tampon. En effet, par rapport au thread qui décompresse le flot vidéo, le thread qui traite les trames audio consomme peu de ressources processeurs, de ce fait il est plus souvent élu par l'ordonnanceur UNIX. Ceci a pour conséquence que la quantité de données audio produites par ce thread durant une période de temps dépasse de beaucoup la quantité consommée par la carte audio pendant cette même période. La taille maximale de ce tampon est alors rapidement atteinte[2]. Lorsque le jeu d'équations (C.3.1.2) est utilisé, l'ordonnanceur POLKA qui alloue le processeur, non plus sur des critères de consommation processeur mais sur des critères d'échéances, permet de ralentir l'engorgement du tampon. Néanmoins, après une trentaine de secondes, la taille du tampon atteint aussi son maximum (tout en oscillant de façon moins importante).

En fait, la deuxième spécification ne modélise pas les caractéristiques temporelles de certains éléments importants du système, qui influent sur le comportement de l'application. Ainsi, la spécification donnée ci-dessus ne dit rien sur la consommation des données par la carte audio. A quel rythme la carte audio va-t-elle consommer des éléments de son tampon et quelle QoS peut elle offrir à l'utilisateur ? Ces questions ne trouvent pas de réponse dans notre spécification. Ignorer le débit d'informations consommées par la carte audio explique le fait que son tampon soit engorgé. Il n'existe pas d'asservissement entre les opérations de dépôt dans le tampon et les opérations de retrait (en dehors de celui offert par le système quand le tampon est vide ou lorsqu'il a atteint sa taille maximale). Ceci explique aussi la perte de synchronisation voix-lèvres : en effet, la troisième équation du jeu (C.3.1.2) synchronise les dépôts de données audio dans le tampon avec la livraison des images sur l'écran et non la restitution des échantillons audio sur les périphériques de sortie avec l'affichage des images. Cette application montre qu'il existe donc des classes d'applications pour lesquelle la spécification que nous avons utilisée pour l'application de GIF animé est insuffisante.

Dans la suite de ce chapitre, nous proposons une spécification qui définit de bout en bout les différents composants de l'application et du système qui influent sur les flots multimédias. Nous proposons des jeux d'équations de QoS qui prennent en considération ces informations. Enfin, nous discutons des améliorations qu'apportent cette nouvelle spécification sur le comportement de notre application de test.

8.2.3 Modélisation de l'application de bout en bout

A partir de la figure 8.9 et des composants décrits dans la partie 4.4, nous décrivons une autre spécification de notre application. Cette fois-ci, nous modélisons tous les éléments de notre application multimédia, ainsi que ceux du système qui agissent sur les flots de données.

[2]Sur Solaris, lorsque ce tampon est plein (sa taille est d'approximativement 30000 octets sur les machines que nous avons utilisées), le producteur est bloqué jusqu'à la consommation d'éléments par la carte audio. Dans d'autres systèmes d'exploitation, cet engorgement peut conduire à une perte d'informations.

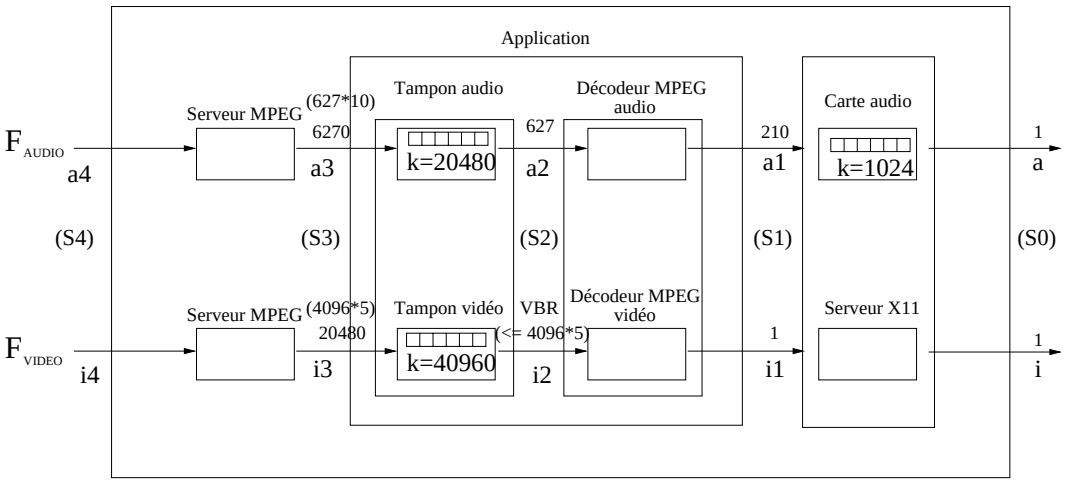


FIG. 8.9 – *Modélisation de bout en bout de l'application*

Nous retrouvons donc les deux flots multimédias audio et vidéo (F_{video} et F_{audio} sur la figure 8.9) ainsi qu'un ensemble de composants modélisant :

- La carte audio et le serveur X11.
- Les décodeurs MPEG-2 audio et vidéo.
- Les tampons alimentés par les threads qui invoquent les méthodes de l'interface IDL $mpegServer$.
- Les composants qui fournissent au client les paquets de trames MPEG mémorisés dans les tampons décrits ci-dessus.

Nous montrons comment, à partir de la QoS souhaitée par l'utilisateur et de notre modèle, il est possible de déterminer la QoS requise par la source, QoS qui constitue une condition nécessaire au respect de la QoS spécifiée par l'utilisateur.

Pour faciliter la lecture de cette partie, les informations suivantes ont été consignées sur la figure 8.9 :

- Si un composant contient un tampon, sa taille est spécifiée (paramètre k).
- Les signaux d'entrée et de sortie de chaque composant sont donnés en dessous des flèches modélisant les flots de données (ainsi, le signal d'entrée du décodeur vidéo MPEG est $i2$ alors que son signal de sortie est $i1$).
- Enfin, la taille des données véhiculées par chaque signal est précisée au dessus de chaque flèche. L'unité utilisée pour les signaux $a4, a3, a2, a1, a, i2, i3$ et $i4$ est l'octet. L'unité des signaux i et $i1$ est l'image. Tous les signaux transportent des éléments de données de taille fixe.

Dans la suite, nous décrivons tous les composants de la figure 8.9 et à partir d'une proposition de QoS utilisateur (jeu d'équations $(S0)$ sur la figure 8.9), nous donnons des solutions possibles pour les jeux d'équations $(S1), (S2), (S3)$ et $(S4)$.

8.2.3.1 Modélisation de la carte audio

Une carte audio compense la gigue sur le flot des données qui lui est fourni : elle possède un tampon que l'utilisateur alimente et elle restitue de façon autonome les données sur la ou les sorties audio. L'application délivre à la carte audio des données au format μ-LAW. Ce format utilise des échantillons d'un octet et nécessite une fréquence d'échantillonnage de 8000 Hz. En d'autres termes, la carte audio consomme un octet toutes les $0,125\ ms$.

Puisque la carte audio fonctionne comme un compensateur de gigue, il est possible de modéliser cet élément comme une instance du composant décrit dans la partie 4.4.3. La sortie audio correspond alors au signal S et la livraison d'un paquet de données à la carte audio, par un appel système $write()$, est modélisée par le signal E. Dans notre application, nous souhaitons une qualité satisfaisante de l'audio. L'utilisateur souhaite donc que tous les $0,125\ ms$ un octet soit livré, ce qui est spécifié par l'équation :

$$\forall\ n\ :\ \tau(a,n) = n * 0,125\ ms$$

Cette équation est identique à l'équation (4.13) du composant compensateur de gigue. Elle constitue une partie de la QoS utilisateur de l'application (jeu d'équations $(S0)$). Par la suite, nous noterons Ha_{pa} l'horloge logique associée à l'équation précédente et dont la période est $pa = 0,125\ ms$.

Avec le film utilisé, les trames en sortie du décodeur audio ont toutes la même taille : 210 octets[3]. La taille des blocs de données fournies au signal d'entrée $a1$ est donc de 210 octets. Le contrôle de la quantité de données contenues dans une carte audio est importante pour plusieurs raisons. Par exemple, lorsque l'on souhaite limiter les phénomènes de latence pour des flots qui peuvent être stoppés puis relancés par l'utilisateur. De même, la maîtrise de cette latence est particulièrement importante si l'on souhaite une synchronisation voix-lèvres satisfaisante. Supposons, à titre d'exemple, que nous souhaitions limiter la consommation mémoire du tampon de la carte audio à 1024 octets (ce qui représente une durée de $1024/0,125 = 128\ ms$). Dans ces conditions, la gigue maximale qui devra être autorisée sur les occurrences du signal $a1$ est de $\epsilon = \frac{1024*0,125}{2} = 64\ ms$. Ainsi, à partir de l'équation (4.14), l'équation

$$\forall\ n\ :\ -32\ ms \leq \tau(Hb_{pb},n) - \tau(a1,n) \leq 32\ ms$$

est une solution permettant le respect de la QoS demandée, où Hb_{pb} est une horloge logique de période $pb = 26,25\ ms$ (avec $26,25 = 0,125 * 210$). Cette équation constitue une partie du jeu d'équations $(S1)$ de la figure 8.9.

[3] En réalité, pour le flot audio MPEG utilisé, la taille oscille entre 209 et 210 octets. Notons que la taille de la trame compressée est plus grande que la taille de la trame décompressée ; ce qui est normal compte tenu du faible débit généré par le codage μ-LAW.

8.2.3.2 Le périphérique de sortie vidéo

Il est difficile de modéliser le comportement du serveur X11. Toutefois, une solution simple consiste à le spécifier par une instance du composant de retard variable (cf. partie 4.4.2). Si x est une borne sur le temps de réponse du serveur X11 lors de la livraison d'une image, alors le composant de retard variable peut être instancié avec $0 \leq l \leq x$.

Si l'utilisateur souhaite délivrer 25 images par seconde (soit une image toutes les 40 ms, c'est la cadence des films utilisés dans ce chapitre) en autorisant 40 ms de gigue entre deux images successives, la QoS qui doit être offerte par le composant "serveur X11" est de la forme :

$$\forall\ n\ : 20\ ms \leq \tau(i, n+1) - \tau(i, n) \leq 60\ ms$$

Cette équation constitue une partie du jeu $(S0)$. D'après les résultats énoncés dans la section 4.4.2.1, la QoS requise par notre composant vidéo est alors de

$$\forall\ n\ : 20 - x\ ms \leq \tau(i1, n+1) - \tau(i1, n) \leq 60 + x\ ms$$

8.2.3.3 De la QoS utilisateur vers la QoS demandée par l'application

$$\forall\ n\ : \begin{cases} \tau(a, n) = n * 0,125\ ms \\ 20\ ms \leq \tau(i, n+1) - \tau(i, n) \leq 60\ ms \\ \mid \tau(a, n*320) - \tau(i, n) \mid \leq 80\ ms \end{cases}$$

FIG. 8.10 – *Une proposition pour le jeu* $(S0)$

Les deux paragraphes précédents définissent les synchronisations intra-flots sur la vidéo et sur l'audio mais ne spécifient rien sur la synchronisation inter-flots. Celle-ci doit faire correspondre les occurrences des signaux i et a. Sachant qu'une image doit être affichée tous les 40 ms et qu'en 40 ms, $40/0,125 = 320$ échantillons audio sont délivrés par la carte audio, on en déduit qu'une solution pour spécifier la synchronisation inter-flots peut être énoncée par :

$$\forall\ n\ : \mid \tau(a, n*320) - \tau(i, n) \mid \leq 80\ ms$$

où 80 ms constitue une gigue généralement acceptable entre le flot audio et le flot vidéo[4] [STE 95b]. Cette équation trouve sa correspondance directe dans le jeu $(S1)$ par :

$$\forall\ n\ : \mid \tau(Ha_{pa}, n*320) - \tau(i1, n) \mid \leq 80\ ms$$

En effet, on sait que $\forall\ n\ : \tau(Ha_{pa}, n) = \tau(a, n)$.

[4] Bien que cela dépende non seulement de l'utilisateur qui regarde le film mais aussi des flots eux-mêmes.

La figure 8.10 résume l'ensemble des équations constituant le jeu $(S0)$ (QoS de l'utilisateur) et la figure 8.11 donne l'intégralité des équations qui, étant donné le jeu $(S0)$ et le comportement temporel du serveur X11 et de la carte audio, constitue la QoS que devra fournir l'application pour que la QoS utilisateur soit respectée.

$$\forall\ n\ :\ \begin{cases} -32\ ms \leq \tau(Hb_{pb}, n) - \tau(a1, n) \leq 32\ ms \\ 20 - x\ ms \leq \tau(i1, n+1) - \tau(i1, n) \leq 60 + x\ ms \\ \mid \tau(Ha_{pa}, n * 320) - \tau(i1, n) \mid \leq 80\ ms \end{cases}$$

FIG. 8.11 – *Une proposition pour le jeu* $(S1)$

8.2.3.4 L'application

Connaissant le jeu d'équations $(S1)$ (la QoS offerte par l'application), nous allons dans ce paragraphe déduire le jeu d'équations $(S3)$ qui constitue la QoS requise par l'application.

Comme nous le montre la figure 8.9, l'application peut être modélisée par un composant possédant deux entrées (les flots audio et vidéo provenant des serveurs de fichiers MPEG) et deux sorties (l'une vers la carte audio et l'autre vers la sortie vidéo). Le composant "application" peut ensuite être raffiné en quatre sous-composants : les décodeurs audio et vidéo ainsi que deux composants dont le rôle est d'absorber la gigue sur le service offert par les serveurs MPEG. Nous allons successivement étudier chacun d'entre eux.

8.2.3.4.1 Les décodeurs audio et vidéo

$$\forall\ n\ :\ \begin{cases} -29,89\ ms \leq \tau(Hb_{pb}, n) - \tau(a2, n) \leq 34,11\ ms \\ 20 - x\ ms \leq \tau(i2, n+4) - \tau(i2, n) \leq 60 + x\ ms \end{cases}$$

FIG. 8.12 – *Une proposition pour le jeu* $(S2)$

Nous commençons par considérer les deux décodeurs MPEG. Nous pourrions, comme pour le serveur X11, évaluer une borne maximale sur leur temps de réponse pour les modéliser par des composants à retard variable. Comme nous disposons des temps moyens de décompression pour le film utilisé ($2,11\ ms$ pour une trame audio et $55,24\ ms$ pour une image vidéo), nous optons ici pour des composants à retard fixe. On peut déduire grâce à la partie 4.4.1 que l'équation

$$\forall\ n\ :\ -32\ ms \leq \tau(Hb_{pb}, n) - \tau(a1, n) \leq 32\ ms$$

du jeu $(S1)$ devient dans le jeu $(S2)$:

$$\forall\ n\ :\ -29,89\ ms \leq \tau(Hb_{pb}, n) - \tau(a2, n) \leq 34,11\ ms$$

Le cas de la vidéo est plus complexe puisque, cette fois-ci, la notion d'image est perdue en entrée du décodeur. En effet, les signaux d'entrée du décodeur vidéo modélisent la consommation de données dans le tampon. Ces données sont retirées du tampon par paquets de 4096 octets (taille des informations véhiculées par le signal $i2$). Or, pour traiter une image, le décodeur utilise un nombre variable de signaux $i2$ (de 1 à 5). Si l'on considère le pire cas, 5 signaux $i2$ sont nécessaires pour décoder une image ; conformément aux résultats de la partie 4.4.1, l'équation

$$\forall\ n\ :\ 20 - x\ ms \leq \tau(i1, n+1) - \tau(i1, n) \leq 60 + x\ ms$$

du jeu $(S1)$ devient alors dans le jeu $(S2)$:

$$\forall\ n\ :\ 20 - x\ ms \leq \tau(i2, n+4) - \tau(i2, n) \leq 60 + x\ ms$$

Bien sûr, n'ayant plus la notion d'image et compte tenu du caractère variable du débit vidéo, il n'est plus possible à cet instant de définir une synchronisation inter-flots. Ainsi, le jeu $(S2)$ de la figure 8.12 ne contient plus que deux équations.

8.2.3.4.2 Les deux tampons

$$\forall\ n\ :\ \begin{cases} -198,35\ ms \leq \tau(Hc_{pc}, n) - \tau(a3, n) \leq 198,35\ ms \\ 5 * (20 - x)\ ms \leq \tau(i3, n+1) - \tau(i3, n) \leq 5 * (60 + x)\ ms \end{cases}$$

FIG. 8.13 – *Une proposition pour le jeu* $(S3)$

La suite traite des tampons. Nous regardons d'abord le cas du tampon audio, puis celui du tampon vidéo.

Le tampon audio peut être modélisé par une instance du composant de la partie 4.4.4. En effet, les sorties du tampon audio sont cadencées par l'horloge Hb_{pb} et soumises à une gigue de 64 ms, ce qui correspond exactement au type de contraintes étudiées dans la partie 4.4.4. Ainsi, l'équation

$$\forall\ n\ :\ -29,89\ ms \leq \tau(Hb_{pb}, n) - \tau(a2, n) \leq 34,11\ ms$$

du jeu $(S2)$ devient dans le jeu $(S3)$

$$\forall\ n\ :\ -198,35\ ms \leq \tau(Hc_{pc}, n) - \tau(a3, n) \leq 198,35\ ms$$

où Hc_{pc} est une horloge de période pc avec :

$$pc = \frac{26,25 * 6270}{627} = 262,5\ ms$$

La gigue sur le signal d'entrée $a3$ est déterminée par :

$$\epsilon' = \frac{\frac{20480*26,25}{627} - 64}{2} = 396,71\ ms$$

avec $198,35 * 2 = 396,71$. Nous renvoyons le lecteur à la partie 4.4.4.2 pour un supplément d'informations sur le calcul de pc et ϵ'.

Le cas du tampon vidéo est plus simple. En effet, ce dernier n'est pas cadencé par une horloge logique. Le tampon est constitué de deux parties. Pendant qu'une partie du tampon est remplie par les données du signal $i3$, le signal $i2$ prélève des informations dans la seconde partie. Lorsque la seconde partie est vide et que l'autre est pleine, les rôles sont échangés : le signal $i3$ produit des informations dans la seconde partie et le signal $i2$ consomme dans l'autre, et ainsi de suite. Pour garantir la QoS nécessaire à ce type de composant, une condition suffisante consiste à lui appliquer la même contrainte temporelle que celle exprimée par sa QoS offerte. Ainsi l'équation

$$\forall\ n\ :\ 20 - x\ ms \leq \tau(i2, n+4) - \tau(i2, n) \leq 60 + x\ ms$$

du jeu $(S2)$ devient dans le jeu $(S3)$:

$$\forall\ n\ :\ 5 * (20 - x)\ ms \leq \tau(i3, n+1) - \tau(i3, n) \leq 5 * (60 + x)\ ms$$

Finalement, la figure 8.13 résume le jeu d'équations de QoS $(S3)$.

8.2.3.5 QoS requise par les serveurs de fichiers MPEG

$$\forall\ n\ :\ \begin{cases} -198,35 + y\ ms \leq \tau(Hc_{pc}, n) - \tau(a4, n) \leq 198,35 + y\ ms \\ 5 * (20 - x) - z\ ms \leq \tau(i4, n+1) - \tau(i4, n) \leq 5 * (60 + x) + z\ ms \end{cases}$$

FIG. 8.14 – *Une proposition pour le jeu* $(S4)$

Nous terminons l'étude de notre application de test par la définition de la QoS requise par les serveurs de fichiers MPEG (ou, en d'autres termes, la QoS qui doit nous être offerte par le système de gestion de fichiers).

Nous modélisons ces deux composants par deux composants à retard variable. Soit y la borne sur le temps de réponse du serveur MPEG audio et z la borne sur le temps de réponse sur le serveur MPEG vidéo, le jeu d'équations de la figure 8.14 donne la QoS requise par les serveurs MPEG qu'il est possible de déduire à partir du jeu $(S3)$ et du composant défini dans la partie 4.4.2.

Finalement, le jeu $(S4)$ constitue une condition nécessaire pour le respect des contraintes de QoS de l'utilisateur. La condition devient suffisante si les ressources nécessaires sont disponibles lors de l'exécution de l'application.

8.2.4 Comportement induit par la nouvelle spécification

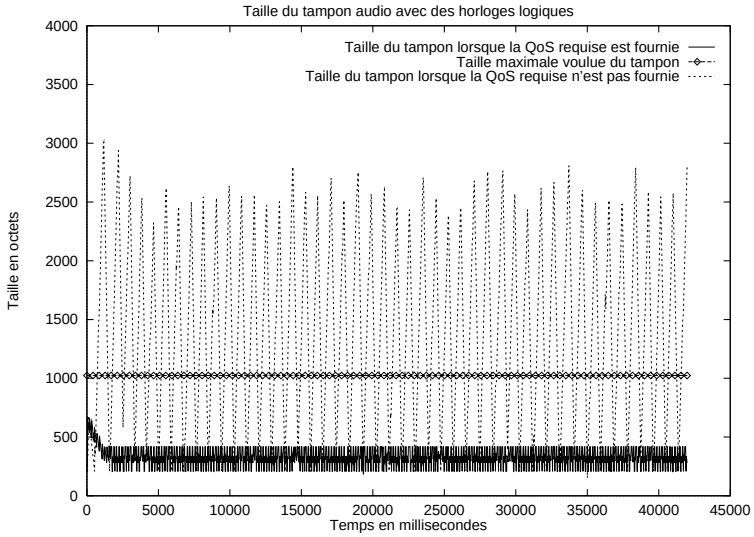


FIG. 8.15 – *Taille du tampon de la carte audio*

Nous avons modélisé notre application de bout en bout. Bien que ce soit un exercice intéressant, pour exécuter cette application sur la plate-forme POLKA, nous aurions pu nous arrêter à la construction du jeu d'équations $(S1)$. Si l'on exécute l'application avec ce jeu d'équations (cf. annexe C.3.2), les problèmes cités dans la partie 8.2.2 sont résolus :

- La synchronisation voix-lèvres est maintenue tout au long du film. Bien qu'aucune mesure n'ait été réalisée pour appuyer ce résultat, une nette amélioration est constatée par un observateur humain. Il faut toutefois préciser que la qualité de la synchronisation obtenue sur cette application reste inférieure à celle délivrée par des décodeurs effectuant cette synchronisation de façon "ad hoc". En effet, les décodeurs MPEG utilisent les estampilles temporelles stockées dans le flot de données. Celles-ci sont calculées lors du codage et du multiplexage des flots audio et vidéo. Elles précisent à quel moment doivent être décodées et/ou affichées les trames MPEG. Le caractère variable du débit du flot vidéo induit une désynchronisation transitoire si ces estampilles ne sont pas utilisées, ce qui est le cas dans notre application. De plus, il faut préciser que notre modèle de spécification n'est pas adapté à la spécification de telles contraintes. Enfin, notre décodeur vidéo utilise une heuristique peu efficace

pour maintenir un rythme d'affichage de 25 images par seconde. La synchronisation que nous obtenons reste toutefois raisonnable, voire suffisante, pour certains films [5].

- L'occupation du tampon audio est prédictible. Si l'on applique le jeu d'équations ($S1$), elle n'excède plus 1024 octets. La figure 8.15 montre que, si l'ordonnanceur respecte la QoS requise par le composant "carte audio", l'application utilise environ de 200 à 400 octets du tampon [6]. Cette légère variation de la taille du tampon peut s'expliquer par plusieurs raisons :
  - D'abord par l'ordonnancement du thread qui écrit les données dans le tampon. En effet, la gigue autorisée pour la livraison de chaque signal $a1$ se retrouve lors de chaque activation du thread.
  - Ensuite par le fait que les temps de réponse de l'application ne sont pas fortement déterministes.
  - Enfin, l'occupation du tampon des figures 8.8 et 8.15 est une estimation qui peut être légèrement différente de la réalité.

  La figure 8.15 montre également que, si la QoS requise n'est pas donnée au composant audio, l'occupation du tampon peut augmenter de façon considérable (jusqu'à 3000 octets sur la figure). Ce phénomène s'explique aisément : si l'ordonnanceur ne donne pas en temps et en heure la ressource processeur au thread qui décompresse les données audio, ce dernier prend un certain retard qu'il rattrape par la suite en décodant plusieurs trames audio successivement, ce qui implique une plus grande oscillation de l'occupation du tampon. Cette courbe illustre bien le caractère contractuel qui doit s'établir entre la QoS offerte et la QoS requise des différents composants du système pour que finalement, la QoS utilisateur puisse être respectée. Notons que 400 octets de données audio représentent seulement un temps de latence de 50 ms (soit moins que le temps moyen de traitement d'une image pour le film considéré dans cette application). On peut donc considérer le problème de la latence lors de la suspension du flot audio comme résolu avec cette proposition.

Dans cette application, le modèle proposé dans le chapitre 4 nous a permis de raisonner sur le comportement temporel de notre application et de son environnement, puis, de définir le jeu d'équations qui devait finalement être donné à l'ordonnanceur (jeu ($S1$)). Ce jeu est le résultat d'une combinaison de la QoS spécifiée par l'utilisateur (jeu ($S0$)) et du comportement des composants. Nous avons montré que cette application ne pouvait

[5] Dans ce chapitre, nous utilisons la valeur de 80 ms entre un élément audio et un élément vidéo pour spécifier la synchronisation inter-flots [STE 95b]. En fait, la véritable valeur dépend de la sensibilité de l'utilisateur et aussi du contenu du film ; ainsi, le besoin en synchronisation voix-lèvres pour une séquence d'images contenant l'annonce d'une speakerine en gros plan est plus forte que pour un film animalier contenant une voix "off".

[6] Pour être précis, cette valeur devrait être augmentée de 512 octets (soit $64/0,125$). En effet, dans cette application, nous n'avons pas effectué un préchargement du tampon avant la consommation de ses éléments [GAG 96]. Ce préchargement équivalent à une fois la gigue évite la famine du tampon (phénomène que nous n'avons pas constaté dans notre cas).

pas se suffire d'une spécification de QoS aussi simple que celle proposée pour l'application de GIF animé : en effet, cette application nécessite un support de bout en bout de la QoS [CAM 98]. Nous montrons dans l'application suivante comment cette modélisation de bout en bout permet le support des applications réparties.

8.3 Exécution d'une application répartie : évaluation de l'efficacité de POLKA

8.3.1 Description de l'application

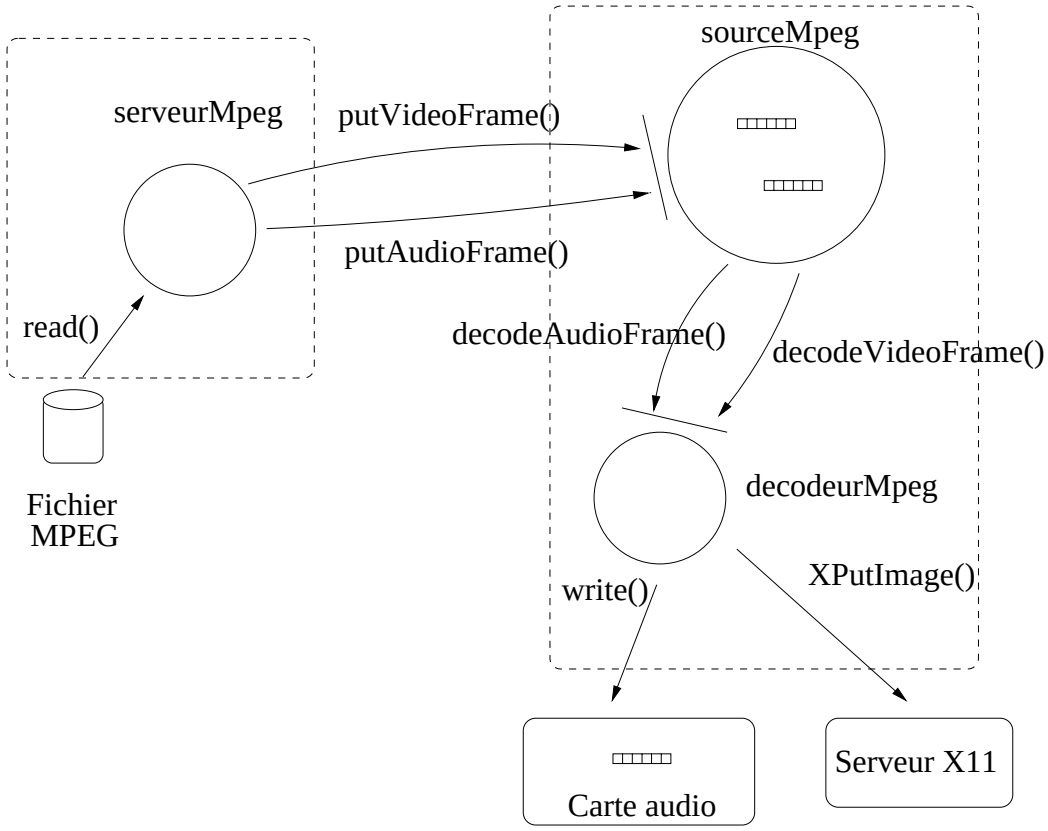


FIG. 8.16 – *Une application répartie sur* POLKA

Dans cette partie, nous modélisons une application répartie, puis, nous donnons quelques éléments sur l'efficacité de notre plate-forme. L'application utilisée est inspirée de l'application centralisée précédente. Elle est constituée de trois objets : un objet qui lit sur disque des trames MPEG et qui les dépose dans un tampon (objet *serveurMpeg* sur la figure 8.16), un objet client (objet *sourceMpeg*) qui consomme les trames et un objet qui décode les trames MPEG (objet *decodeurMpeg*). Cette fois, l'objet serveur MPEG est placé sur

une machine différente des objets client et décodeur. Enfin, contrairement à l'application centralisée, le transfert des trames par le réseau est à l'initiative du serveur de fichiers MPEG et non plus à l'initiative du client. En effet, si un bus à objets offrant une couche de communication dédiée aux flots multimédias[7] était utilisé avec cette application, c'est ce type de configuration qui serait utilisé.

```
module mpeg {
    interface mpegDecoder {
        void decodeAudioFrame(in audioFrame f);
        void decodeVideoFrame(in videoFrame f);
    };
    interface mpegServer {
        void putAudioFrame(in audioFrame f);
        void putVideoFrame(in videoFrame f);
        void getAudioFrame(out audioFrame f);
        void getVideoFrame(out videoFrame f);
    };
};
```

FIG. 8.17 – *Interface* IDL *de l'application répartie*

8.3.2 Spécification de QoS de l'application

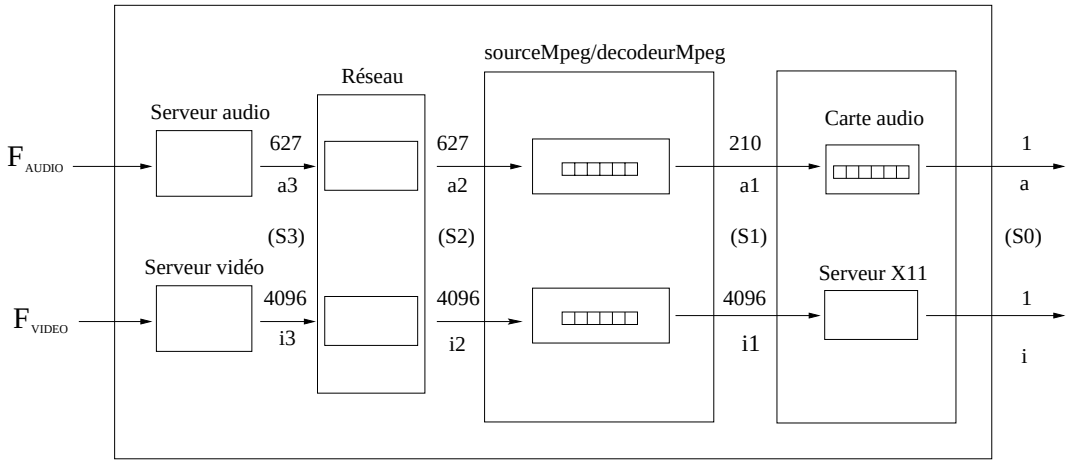


FIG. 8.18 – *Modélisation de l'application répartie*

[7] Service de communication éventuellement non fiable et ne générant pas de retransmission de données.

Modélisons le comportement temporel de l'application répartie. La figure 8.18 représente le graphe de flots de données de l'application. Un composant modélise la carte audio qui restitue le flot audio en sortie. Un autre, le serveur X11, est chargé d'afficher le flot vidéo. Les deux composants centraux représentent les décodeurs MPEG audio et vidéo d'une part, et le réseau d'autre part. Comme précédemment, on désigne par $(S0)$ le système d'équations correspondant à la QoS utilisateur attendue en sortie (respectée par la carte audio et le serveur X11).

Afin de faciliter l'interprétation de notre évaluation, nous simulons l'application précédente. En effet, les temps de décompression et de présentation des trames audio ou vidéo varient. Pour ne pas ajouter ces variations à celles produites par notre ordonnanceur, nous simulons ces opérations par des temporisations dont la durée est la durée moyenne observée lors de l'exécution de l'application sur Solaris 2.5 avec une machine Ultra-sparc 1 cadencée à 167 MHz et 128 Mo de mémoire vive (soit $2,11\ ms$ pour une trame audio et $55,24\ ms$ pour une trame vidéo).

Nous reprenons donc les caractéristiques temporelles et quantitatives de l'application précédente. Nous supposons que la carte audio a le même comportement temporel et que les trames audio ont une taille de 627 octets lorsqu'elles sont compressées et de 210 octets lorsqu'elles sont décompressées[8]. En revanche cette fois, nous simulons un débit constant pour la vidéo en supposant qu'une image est codée sur une trame dont la taille est de 4096 octets. Enfin, nous supposons que l'utilisateur souhaite obtenir un rythme d'affichage de 10 images par seconde. Toutes ces hypothèses, nous conduisent au jeu $(S0)$ suivant :

$$\forall\ n\ : \begin{cases} \tau(a,n)\ =\ n*0,125\ ms \\ 80\ ms \leq \tau(i,n+1) - \tau(i,n) \leq 120\ ms \\ -40\ ms \leq \tau(Ha_{pa}, n*800) - \tau(i,n)\ \leq 40\ ms \end{cases}$$

Il s'agit maintenant de déduire de $(S0)$ et du comportement des composants carte audio et serveur X11, le jeu d'équations $(S1)$ à fournir en entrée de ces composants. L'équation du jeu $(S1)$ pour le flot audio est :

$$\forall\ n\ :\ -32\ ms \leq \tau(Hb_{pb}, n) - \tau(a1, n) \leq 32\ ms$$

où Hb_{pb} est une horloge logique de période $pb = 26,25\ ms$ (puisque nous reprenons les mêmes paramètres que pour la deuxième application). En revanche, les équations intra-flots vidéo et inter-flots sont maintenant :

$$\forall\ n\ :\ 80\ ms \leq \tau(i1, n+1) - \tau(i1, n) \leq 120\ ms$$

et

$$\forall\ n\ :\ -40\ ms \leq \tau(Ha_{pa}, n*800) - \tau(i1, n)\ \leq 40\ ms$$

[8]Notons que la taille de la trame compressée est plus grande que la taille de la trame décompressée ; ce qui est normal compte tenu du faible débit généré par le codage μ-LAW.

Les équations ci-dessus sont différentes de celles que nous avions proposées dans la deuxième application. En effet, afin de simplifier la spécification, le serveur X11 est modélisé comme un composant de retard fixe. Finalement, le jeu $(S1)$ est donc :

$$\forall\ n\ : \begin{cases} -32\ ms \leq \tau(Hb_{pb}, n) - \tau(a1, n) \leq 32\ ms \\ 80\ ms \leq \tau(i1, n+1) - \tau(i1, n) \leq 120\ ms \\ -40\ ms \leq \tau(Ha_{pa}, n*800) - \tau(i1, n)\ \leq 40\ ms \end{cases}$$

Il nous faut maintenant examiner quelle QoS (jeu $(S2)$) il est nécessaire de fournir à l'entrée des objets $sourceMpeg/decodeurMpeg$ pour que la QoS offerte corresponde au jeu $(S1)$. Nous modélisons les décodeurs comme des composants de retard fixe. Nous reprenons les temps moyens de décompression qui sont de $2,11\ ms$ pour une trame audio et de $55,24\ ms$ pour une image vidéo, d'où les équations suivantes du jeu $(S2)$:

$$\forall\ n\ :\ -29,89\ ms \leq \tau(Hb_{pb}, n) - \tau(a2, n) \leq 34,11\ ms$$

et

$$\forall\ n\ :\ 15,24\ ms \leq \tau(Ha_{pa}, n*800) - \tau(i2, n)\ \leq 95,24\ ms$$

L'équation décrivant la synchronisation intra-flot sur la vidéo trouve sa correspondance directe dans l'équation :

$$\forall\ n\ :\ 80\ ms \leq \tau(i2, n+1) - \tau(i2, n) \leq 120\ ms$$

Et finalement, le jeu $(S2)$ est :

$$\forall\ n\ : \begin{cases} -29,89\ ms \leq \tau(Hb_{pb}, n) - \tau(a2, n) \leq 34,11\ ms \\ 80\ ms \leq \tau(i2, n+1) - \tau(i2, n) \leq 120\ ms \\ 15,24\ ms \leq \tau(Ha_{pa}, n*800) - \tau(i2, n)\ \leq 95,24\ ms \end{cases}$$

Jusqu'au jeu $(S2)$, la construction des différents jeux d'équations de QoS diffère peu de ce que nous avons présenté pour la deuxième application. Toutefois, dans la troisième application, nous introduisons un élément réseau. Le réseau est modélisé par un composant de retard variable ; $mina$ et $maxa$ désignent les temps minimaux et maximaux que met un échantillon audio pour traverser le réseau et $minv$ et $maxv$ désignent les temps minimaux et maximaux que met une image pour traverser le réseau. Comme l'application est exécutée sur un réseau Ethernet, il n'est pas possible de donner a priori des valeurs pour $maxa$, $maxv$, $mina$ et $minv$. Ces quatre informations constituent des variables dans les équations de QoS, dont les valeurs seront renseignées par les services de supervision de POLKA.

Ainsi, le jeu $(S3)$ est donc :

$$\forall\ n\ : \begin{cases} -29,89\ ms + maxa \leq \tau(Hb_{pb}, n) - \tau(a3, n) \leq 34,11\ ms + mina \\ 80\ ms - maxv + minv \leq \tau(i3, n+1) - \tau(i3, n) \leq 120\ ms + maxv - minv \\ 15,24\ ms + maxv \leq \tau(Ha_{pa}, n*800) - \tau(i3, n) \leq 95,24\ ms + minv \end{cases}$$

Ceci termine notre modélisation de l'application. **En pratique, l'utilisateur n'est pas obligé de fournir tout ce travail.** En effet, la plate-forme réalise automatiquement ce calcul, à condition bien sûr, que le concepteur fournisse :

- La QoS utilisateur.
- La description des composants.
- Le graphe de flots de données.

Dans notre exemple, la plate-forme génère automatiquement le jeu d'équations ($S3$) qui permettra d'ordonnancer le serveur afin que le client puisse respecter la QoS du jeu ($S0$). Le jeu d'équations ($S3$) constitue donc les contraintes de QoS fournies à l'ordonnanceur de la station émettrice. De même, le jeu d'équations ($S1$) constitue les contraintes de QoS fournies à l'ordonnanceur de la station réceptrice.

L'annexe C.4 contient la définition des composants et du graphe de flots de données décrivant cette application répartie. Les équations générées par *qc* à partir de cette définition sont énumérées dans la partie C.4.2. Nous renvoyons le lecteur à la partie 6.4.2 pour une description de l'algorithme de propagation des contraintes de QoS.

Notons que les seuls éléments qui différencient une application POLKA centralisée d'une application POLKA répartie se situent dans la spécification de QoS. Pour répartir une application POLKA il n'est donc pas nécessaire de la modifier ou de la recompiler.

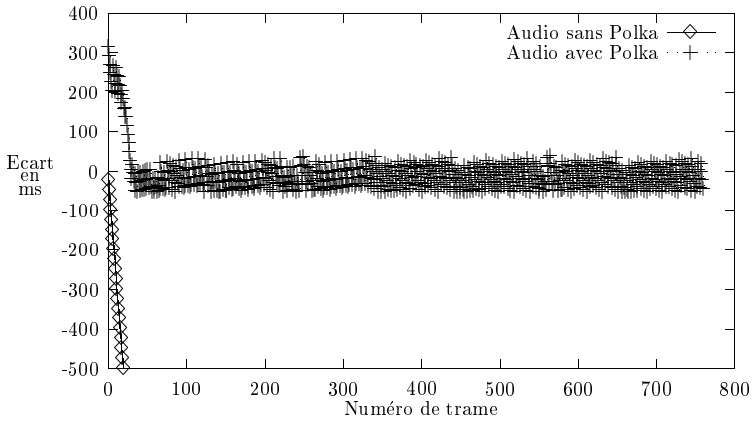


FIG. 8.19 – *Respect des synchronisations intra-flots*

8.3.3 Evaluation de l'efficacité

Nous terminons ce chapitre par une évaluation du surcoût induit par l'utilisation de la plate-forme POLKA, et son impact sur les synchronisations intra et inter-flots de notre

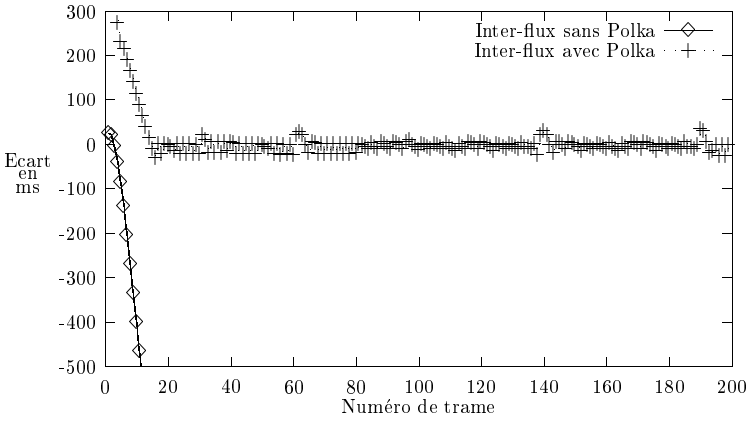


FIG. 8.20 – *Respect des synchronisations inter-flots*

application. Notre application répartie est exécutée sans POLKA, puis avec POLKA sur deux environnements :

1. Deux machines Linux comprenant un Pentium II cadencé à 350 MHz, avec 64 Mo de mémoire vive. Ces machines sont connectées par un réseau Ethernet à 100Mbits/s en étoile. Chaque machine Linux exécute un noyau 2.3.2.

2. Deux machines Ultra-Sparc 1 cadencées à 143 MHz avec 64 Mo de mémoire vive et connectées par un réseau Ethernet à 100Mbits/s en étoile. Ces machines utilisent un système d'exploitation Solaris 2.5.

La mesure des synchronisations intra et inter-flots est effectuée sur les machines Linux. La mesure du surcoût est réalisée sur Linux et Solaris. La courbe de la figure 8.19 positionne les dates de livraison des trames audio à la carte audio par rapport aux tops de l'horloge Ha_{pa}. Lorsque cette livraison est synchronisée avec un top d'horloge, la courbe croise la droite d'ordonnée zéro. Une variation de 64 *ms* autour de cette droite correspond à la tolérance sur la synchronisation intra-flot (système d'équations ($S1$)). On peut observer que, sans POLKA, les trames audio sont délivrées de façon complètement asynchrone aux tops d'horloge. Avec POLKA, comme le montre la courbe, la présentation des trames est asservie à l'horloge. Les contraintes de QoS intra-flots sont respectées.

La deuxième courbe (cf. figure 8.20) montre le délai entre l'affichage d'une image et la présentation des trames audio associées. Lorsque la courbe croise la droite d'ordonnée zéro, la synchronisation inter-flots est optimale. POLKA prouve, là encore, son efficacité.

Examinons maintenant le surcoût engendré par POLKA. Ce dernier comprend principalement le temps nécessaire pour calculer les échéances des tâches et, dans le cas où une application est répartie sur plusieurs machines, le coût pour transmettre les données locales d'ordonnancement vers les ordonnanceurs distants. Le surcoût comprend aussi le temps nécessaire pour synchroniser les différents threads POLKA ainsi que le temps de communication entre le démon POLKA et les applications au travers du *proxy* (IPC et RPC).

Surcoût	Linux	Solaris
Application centralisée	$135,6\ \mu s$	$364,94\ \mu s$
Application répartie	$541,3\ \mu s$	$744,29\ \mu s$

TAB. 8.1 – *Surcoût induit par* POLKA

Ce surcoût est consigné dans le tableau 8.1. Il détaille le surcoût moyen engendré pour une activation de notre ordonnanceur. Une activation de notre ordonnanceur est requise pour chaque exécution d'une tâche. Pour ramener ce surcoût au niveau d'une invocation de méthode, il faut donc multiplier ces chiffres par deux ou quatre selon que l'invocation soit asynchrone ou synchrone (cf. partie 6.1).

Si l'on compare ces résultats au temps nécessaire pour décoder et afficher une image ($55,24\ ms$), le surcoût généré par POLKA reste raisonnable (de l'ordre de $2,7$ % sur Linux et de $5,4$ % sur Solaris dans le cas réparti).

Notons que le temps consacré au calcul des échéances et des dates d'éligibilité ainsi que celui de l'élection EDF ne constituent qu'une petite partie du surcoût total de notre ordonnanceur. Dans le cas centralisé, dans la plate-forme sur Linux, ces opérations constituent seulement $24,712$ % du surcoût global. Il est de $14,178$ % sur Solaris. Le reste du surcoût est constitué d'opérations de synchronisation et de communication locale, plus ou moins efficace selon les services offerts par le système d'exploitation sous-jacent.

8.4 Conclusion

Dans ce chapitre, nous avons montré, au travers de plusieurs applications, comment notre modèle de spécification pouvait être appliqué dans le cadre d'une application centralisée, puis dans le cadre d'une application répartie. Nous avons montré que le modèle était suffisamment expressif pour des applications multimédias qui requièrent le support de la QoS de bout en bout. Enfin, nous avons donné quelques éléments qui montrent l'efficacité de notre plate-forme, et plus généralement de notre approche, et ce pour un surcoût raisonnable compte tenu des applications ciblées. Ce surcoût est pour trois quart constitué d'opérations de synchronisation et de communications locales ; c'est la raison pour laquelle nous avons initié un portage de POLKA sur une plate-forme Linux-L4 [HAR 97]. En effet, cette dernière offre des services de synchronisation particulièrement performants.

Malheureusement, le modèle proposé reste limité à des contraintes déterministes, ce qui parfois conduit à surcontraindre nos spécifications temporelles [BAI 96]. De même, nous avons pu constater, avec la deuxième application, que nos contraintes temporelles ne permettaient pas la spécification de flots de données continues générant un débit variable. Avant de conclure cette thèse, nous proposons, dans le chapitre suivant, une extension simple de notre modèle à des contraintes probabilistes. Nous montrons, avec un exemple d'application, comment il est possible d'améliorer la gestion des ressources sur un système multimédia par l'exploitation du comportement probabiliste de certains de ses composants.

Chapitre 9

Vers le support de contraintes probabilistes

Ce chapitre se propose de discuter de l'intérêt d'ajouter, dans notre modèle, des contraintes probabilistes. Le support des contraintes probabilistes est motivé par plusieurs raisons. La première motivation est une conséquence directe des objectifs de cette thèse. En effet, nous ciblons tout particulièrement des systèmes qui ne sont pas complètement déterministes. Or, il existe des éléments dans ces systèmes dont le comportement peut être décrit par des modèles stochastiques. Avec notre modèle actuel, ceux-ci sont approximés par une spécification déterministe, ce qui a pour conséquence de sur-contraindre inutilement le système. D'autre part, contrairement à une application temps réel, il arrive que l'utilisateur souhaite exprimer des contraintes probabilistes. Ainsi, un téléspectateur qui consulte une œuvre cinématographique, grâce à un serveur de vidéo à la demande, peut s'accommoder d'un faible pourcentage d'échec sur le respect des contraintes temporelles lors de la présentation. Enfin, pour exprimer des trafics à débit variable, les contraintes probabilistes sont couramment utilisées. C'est notamment le cas du service VBR-RT dans ATM [FOR 96] ou des trafics proposés dans Tenet [BAN 96].

Dans ce chapitre, nous introduisons une application répartie sur ATM. Nous modélisons son comportement temporel avec des contraintes probabilistes simples. Puis, nous montrons le gain obtenu vis-à-vis d'un modèle déterministe. L'objectif de ce chapitre n'est pas de proposer une solution satisfaisante à tout point de vue, mais plutôt d'illustrer de façon pratique les gains importants que peut induire l'utilisation de contraintes probabilistes dans notre plate-forme. Dans la conclusion de ce chapitre, nous décrivons les pistes qui nous semblent intéressantes à explorer dans ce cadre.

9.1 Présentation de l'application, contexte

L'application que nous utilisons est constituée d'un objet client et d'un objet serveur (cf. figure 9.1). Elle comprend deux threads (un premier dans l'objet client et un second dans l'objet serveur). Ces deux threads gèrent un flot d'images au format XPM. Le thread client

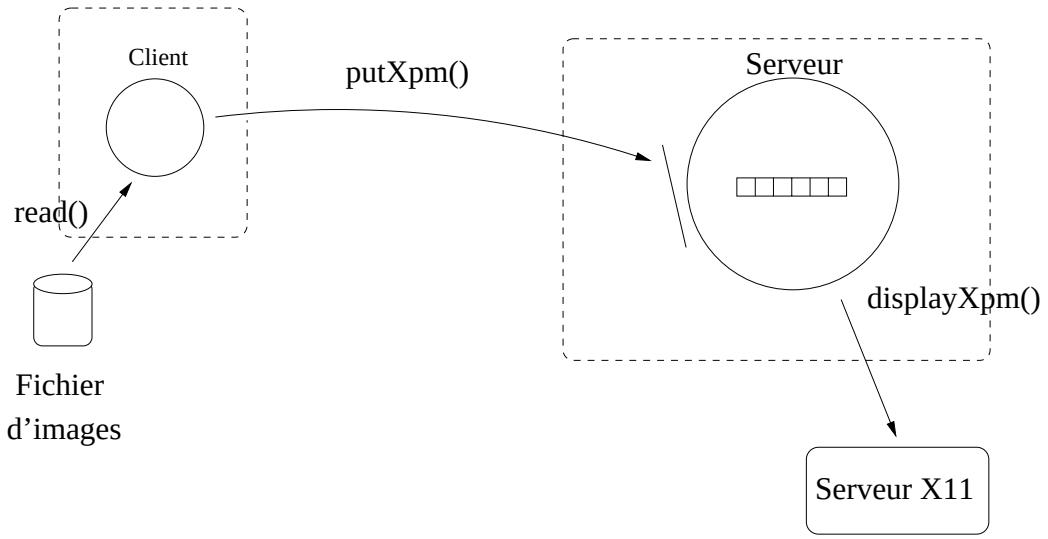


FIG. 9.1 – *Modèle objets de l'application*

```
module proba {
    const long xpmSize = 20000;
    typedef octet xpm[xpmSize];
    interface server {
        void putXpm(in xpm data)
        void displayXpm();
    };
};
```

FIG. 9.2 – *Interface* IDL *de l'application*

lit les images depuis le disque, puis, les transfère par le réseau au serveur. L'objet serveur contient un tampon qui stocke les images reçues du réseau. Son interface est décrite par la description IDL de la figure 9.2. Le thread client transmet les images grâce à la méthode *putXpm*(). La méthode *displayXpm*() est invoquée par le thread serveur pour provoquer l'affichage par le serveur X11.

Le transfert des images est effectué sur un canal ATM CBR. La plate-forme ATM que nous utilisons est un réseau en étoile utilisant un commutateur FORE LE155. Ce commutateur est capable, entre autres, de supporter des services de communication CBR et UBR (*Unspecified Bit Rate*). Nous avons déjà présenté le service CBR (cf. pages 19 et 67). Celui-ci offre à l'utilisateur une garantie de débit et de gigue maximal inter-cellules. Le service UBR est un service de type "best effort" : aucune spécification de QoS et de trafic n'est fournie par l'application et le réseau ne donne pas de garantie sur le comportement

Temps d'aller-retour		
Minimum	Maximum	Moyenne
23804 μs	56593 μs	24130,37 μs

TAB. 9.1 – *Temps d'aller-retour d'une invocation de méthode sur le canal CBR*

temporel et la bande passante de la connexion. L'application est exécutée sur des machines PC Pentium II cadencées à 350 MHz avec 64Mo de mémoire centrale. Elles possèdent une carte ATM PCI FORE PCA200E. La partie logicielle de la plate-forme est constituée du logiciel distribué par l'EPFL [ALM 96, ALM 97]. La plate-forme autorise la spécification d'un débit crête (PCR) et d'une gigue sur cellules (ppCDV) pour chaque connexion CBR. Les connexions CBR sont implantées au dessus d'une couche AAL5. Pour bénéficier du service CBR de la plate-forme ATM, nous utilisons une version particulière du bus à objets omniORB2. Cette version d'omniORB2, développée par B. Driss à l'ENST Paris [DRI 00], intègre les protocoles ATM en proposant une couche GIOP (*Global Inter-ORB Protocol*) au dessus de la couche AAL5. Le bus à objets offre ainsi la possibilité d'ouvrir des connexions GIOP soit sur TCP-IP (grâce à Classical-IP), soit sur AAL5 en utilisant un service UBR ou CBR.

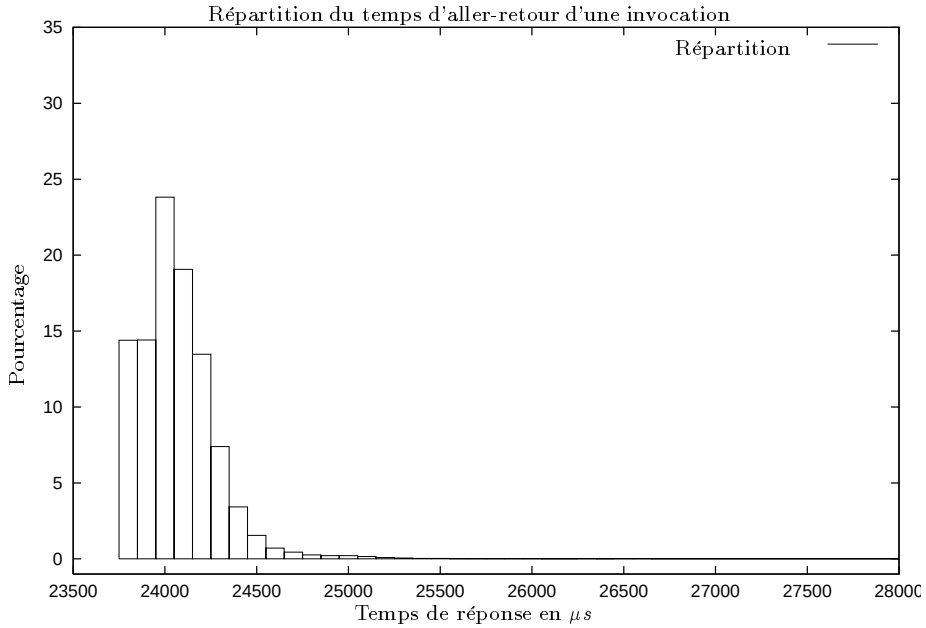


FIG. 9.3 – *Répartition du temps d'aller-retour d'une invocation sur le canal CBR*

Toutes les expérimentations qui sont décrites dans la suite sont réalisées sur un canal CBR bidirectionnel alloué avec un débit crête de 20000 cellules par seconde et une gigue

maximale de 100 ms. Toutefois, si nous observons le temps d'aller-retour des invocations de méthode qui transfèrent les images du client vers le serveur (méthode $putXpm$), nous constatons que la majeure partie des transmissions sont effectuées avec une gigue bien inférieure. En effet, la gigue maximale est de 32789 μs mais dans 98,692 % des cas, celle-ci est inférieure à 900 μs. La tableau 9.1 et la courbe 9.3 résument la répartition du temps de réponse de notre application. Le lecteur peut consulter les conditions de mesure de ces informations dans l'annexe A.4.1.

Pour motiver l'intégration des contraintes probabilistes dans notre modèle, nous nous proposons dans la suite de ce chapitre d'exploiter cette caractéristique. Nous modélisons l'application décrite ci-dessus par une spécification déterministe puis probabiliste et nous discutons de l'impact de ces deux modélisations sur les ressources processeurs et mémoires.

9.2 Modélisation temporelle de l'application

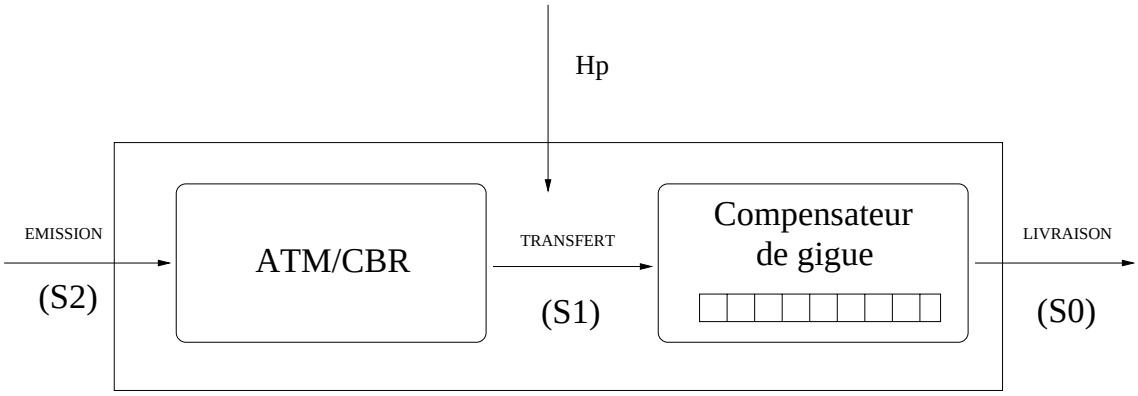


FIG. 9.4 – *Graphe de flots de données de l'application*

La spécification temporelle de l'application peut être définie comme suit (cf. figure 9.4). Nous modélisons le serveur (c'est-à-dire le puits) comme un compensateur de gigue et la connexion ATM par un composant de retard variable. L'application contient un seul flot de données. Le graphe est composé de trois signaux. Le signal $LIVRAISON$ présente les images au serveur X11. Le signal $TRANSFERT$ modélise l'arrivée d'une image chez le serveur. Enfin, le signal $EMISSION$ constitue l'émission des images par le client.

9.2.1 Impact sur la ressource processeur

Dans un premier temps, nous décrivons les jeux d'équations de QoS qui modélisent de façon déterministe notre application. La QoS utilisateur est constituée d'une seule équation qui spécifie que les images doivent être présentées à une cadence fixe :

$$\forall\ n\ :\ \tau(LIVRAISON, n) = n * p \tag{9.1}$$

Cette équation utilise une horloge logique H_p de période p pour cadencer les émissions des signaux $LIVRAISON$. Dans la suite de ce chapitre, on souhaite délivrer 25 images par seconde ; nous posons donc $p = 40\ ms$. Notons qu'en choisissant cette équation très contraignante comme QoS utilisateur, nous plaçons délibérément notre application dans une situation où elle aura beaucoup de difficultés à respecter ses contraintes (cf. méthode de calcul des échéances dans le chapitre 6). Ce choix va nous permettre de mesurer l'impact d'une spécification probabiliste sur le non respect de la QoS utilisateur. Nous établissons les jeux d'équations de QoS $(S1)$ et $(S2)$ grâce au jeu $(S0)$ et à la spécification temporelle des composants compensateur de gigue et de retard variable. On obtient ainsi l'équation suivante pour $(S1)$:

$$\forall\ n\ :\ -20\ ms \leq \tau(H_p, n) - \tau(TRANSFERT, n) \leq 20\ ms \tag{9.2}$$

en supposant que la taille du tampon du compensateur de gigue est de 2 entrées (cf. partie 4.4.3) ; puis le jeu $(S2)$:

$$\forall\ n\ :\ 36,6\ ms \leq \tau(H_p, n) - \tau(EMISSION, n) \leq 43,8\ ms \tag{9.3}$$

si l'on considère que le composant ATM est un composant de retard variable dont le temps de retard minimal est de $23,8\ ms$ et le temps de retard maximal est de $56,6\ ms$ (cf. partie 4.4.2.2).

Cette première modélisation garantit à l'utilisateur la QoS qu'il a spécifiée sous réserve de la disponibilité des ressources suffisantes sur l'émetteur, sur le récepteur et dans le réseau. Néanmoins, elle impose une contrainte forte sur le thread du site émetteur puisque celui-ci doit être ordonnancé de façon à ne jamais violer l'équation de QoS (9.3) et ce, en considérant les pires temps de communication. Notons que l'équation (9.3) autorise le thread émetteur à utiliser au pire cas $43,8 - 36,6 = 7,2\ ms$ de temps processeur pour lire et transmettre une image ; ce qui est impossible puisque le temps de transfert sur la connexion d'ATM est d'au moins $23,8\ ms$. **Notre spécification déterministe sur-contraint donc de façon inutile le thread émetteur**.

Une alternative à cette modélisation déterministe consiste à substituer à l'équation (9.3) une équation qui tient compte de la répartition des temps de communication dans le canal CBR. Ainsi, si l'on considère la connexion ATM comme un composant de retard variable dont le délai de bout en bout varie entre $23,8\ ms$ et $24,7\ ms$, il est possible de remplacer l'équation (9.3) par :

$$\forall\ n\ :\ 5,3\ ms \leq \tau(H_p, n) - \tau(EMISSION, n) \leq 43,8\ ms \tag{9.4}$$

Bien sûr, cette dernière équation implique que dans $1,31$ % des cas l'équation $(S1)$ soit violée (puisque le délai de bout en bout est compris entre $23,8\ ms$ et $24,7\ ms$ dans $98,69$ % des cas).

Critères	Déterministe	Probabiliste
Marge en ms	$-18,01\ ms$	$13,57\ ms$
Retard moyen en ms	$11,33\ ms$	$8,36\ ms$
Famine en pourcentage	0 %	0,18 %
Débordement en pourcentage	0 %	0,05 %

TAB. 9.2 – *Comparaison entre la spécification probabiliste et la spécification déterministe*

Evaluons ces deux spécifications. Nous exécutons successivement l'application, d'abord avec comme jeu d'équations $(S2)$ l'équation (9.3), puis avec l'équation (9.4). Nous regardons l'impact de ces deux équations à la fois sur les ressources consommées, mais aussi sur la QoS finalement délivrée à l'utilisateur. Pour comparer ces deux exécutions, nous utilisons plusieurs critères, dont celui de la marge. Ce dernier est défini comme suit :

Définition 12 (Marge) *Soit T une tâche et n l'une de ses activations. Soit $D_T(n)$ l'échéance de la $n^{ème}$ activation de la tâche T et $\tau(T,n)$ sa date de terminaison. On appelle marge de la $n^{ème}$ activation de la tâche T la durée évaluée par $D_T(n) - \tau(T,n)$.*

La marge d'une tâche représente le temps dont elle dispose pour respecter ses contraintes temporelles : plus la marge d'une tâche est grande et plus l'ordonnanceur POLKA dispose de temps pour allouer le processeur à la tâche. Nous constatons aisément comment le modèle déterministe peut sur-contraindre de façon inutile une application multimédia. En effet, dans le tableau 9.2 qui relate les mesures effectuées sur l'application, on remarque que cette marge a significativement augmenté dans le cas probabiliste (dans le cas déterministe, celle-ci est négative car le thread viole systématiquement l'équation (9.3)). Or, l'augmentation de cette marge ne s'est pas faite au détriment de la QoS offerte à l'utilisateur puisque, au contraire, le retard moyen sur la présentation des images a diminué. Ceci s'explique par le fait que le site récepteur est moins contraint par le thread exécutant la méthode $putXpm()$, laissant ainsi plus de ressources processeurs pour le thread qui exécute la méthode $displayXpm()$. De même, les phénomènes de famine ou de débordement du tampon, induits par le fait qu'il existe une probabilité que l'équation $(S1)$ ne soit plus respectée, sont faibles. En effet, le taux de débordement observé durant l'exécution de l'application est peu important ; il implique la perte d'une image toutes les 80 secondes. Le taux de famine est, quant à lui, négligeable (d'autant plus que les famines sont observées à l'initialisation de l'application, avant l'établissement d'un régime permanent).

9.2.2 Dimensionnement du tampon

Dans le paragraphe précédent, nous avons pu constater que, dans le cadre de POLKA, l'utilisation de contraintes probabilistes permettait d'effectuer des économies de ressources significatives tout en conservant une QoS utilisateur satisfaisante. Nous regardons maintenant le cas de la ressource mémoire. En effet, dans le cas déterministe, il est facile de

calculer la taille du tampon du compensateur de gigue de façon à garantir l'absence de famine et de débordement. Nous avons vu dans la partie 4.4.3 qu'un tampon égal à deux fois la gigue était suffisant dans ces conditions. Hélas, dans le cas probabiliste, il n'est plus possible d'utiliser cette méthode.

Dans la suite de ce chapitre, nous proposons un modèle stochastique pour dimensionner le tampon. Nous montrons ainsi qu'il est toujours possible d'évaluer la consommation mémoire avec notre modèle de spécification temporelle. Nous modélisons le comportement du compensateur de gigue par une chaîne de Markov, puis nous utilisons cette chaîne pour prédire l'occupation du tampon. Nous terminons finalement par une comparaison de la taille du tampon observé durant l'exécution de l'application avec les prévisions données par la chaîne de Markov.

9.2.2.1 Le composant compensateur de gigue

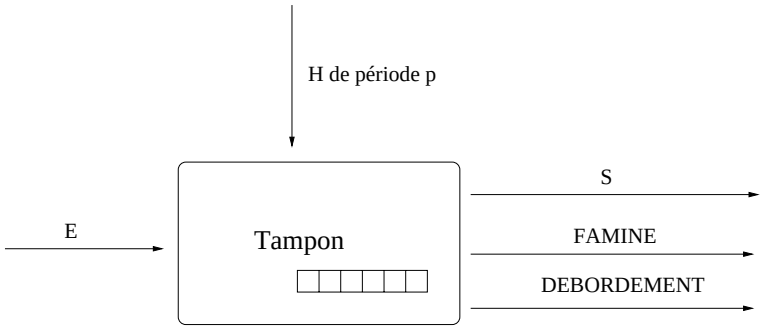


FIG. 9.5 – *Le composant compensateur de gigue*

Le composant compensateur de gigue (cf. figure 9.5) fait intervenir trois signaux : un signal d'horloge H_p de période p, un signal d'entrée E et un de sortie S. Le composant possède un tampon dont la taille est de 2 entrées (la chaîne de Markov proposée ici est toutefois aisément généralisable à une taille quelconque). Nous supposons que la taille des données véhiculées par les signaux E et S sont identiques (nous avons donc $QS = QE$). Nous considérons la QoS offerte suivante :

$$\begin{cases} P(\forall\ n\ :\ \tau(S,n) = n * p) = 1 - j - s \\ P(\forall\ n\ :\ \tau(S,n) < n * p) = j \\ P(\forall\ n\ :\ \tau(S,n) > n * p) = s \end{cases}$$

où $P(q)$ est la probabilité que la contrainte de QoS q soit respectée. Dans ce chapitre, nous utilisons des contraintes probabilistes particulièrement simples. A chaque jeu d'équations de QoS est associée une loi de probabilité sur un ensemble fini d'événements. Les événements correspondent à l'arrivée d'un signal E ou S dans le système à un instant donné.

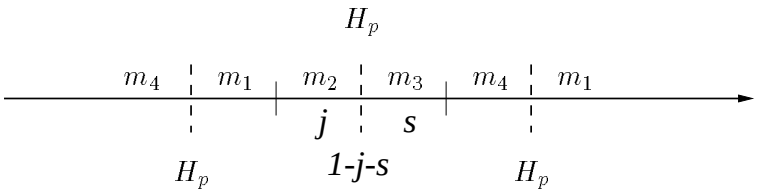


FIG. 9.6 – *Evénements observés dans la chaîne de Markov*

De ce fait, si c est un ensemble de contraintes de QoS modélisant une QoS offerte ou une QoS requise, et si p_i est la probabilité de la $i$ème équation de QoS de c, alors le concepteur doit assurer que :

$$\sum_{\forall\ i\ \in\ c} p_i = 1$$

L'utilisateur spécifie donc une probabilité sur le respect de chaque équation du contrat de QoS. Dans le cas de l'application de ce chapitre, l'utilisateur souhaite une livraison des images à une cadence donnée tout en acceptant qu'un pourcentage d'images ne respecte pas cette contrainte (ici, le pourcentage correspondant à la probabilité $s + j$).

$$\begin{cases} P(\forall\ n\ :\ \epsilon_1 \leq \tau(H_p, n) - \tau(E, n) \leq \epsilon_2) = m \\ P(\forall\ n\ :\ \epsilon_2 < \tau(H_p, n) - \tau(E, n) < \epsilon_1) = 1 - m \end{cases}$$

La QoS requise par le composant est donnée par le jeu d'équations de QoS ci-dessus. Le jeu définit une fenêtre temporelle dans laquelle les occurrences des événements E interviennent. m constitue la probabilité que les occurrences du signal E arrivent dans cette fenêtre. Lorsqu'un signal est délivré au composant, nous considérons plusieurs cas de figure selon l'instant où ce signal intervient entre deux tops de l'horloge. Ces différents instants sont représentés sur la figure 9.6 :

- La donnée arrive dans la fenêtre temporelle spécifiée par la QoS requise. Trois cas sont alors étudiés :
  1. La donnée arrive avec une probabilité m_2 avant le top de l'horloge.
  2. La donnée arrive avec une probabilité m_3 après le top de l'horloge.
  3. Enfin, on suppose négligeable la probabilité que la donnée arrive à l'occurrence du top de l'horloge.
- La donnée est livrée hors de la fenêtre temporelle autorisée. La donnée arrive en avance avec une probabilité m_1 et arrive en retard avec une probabilité m_4.

On pose donc $m = m_2 + m_3$ et $1 - m = m_1 + m_4$.
De plus, nous supposons que :

$$P(\forall\ n\ :\ |\ \tau(S, n) - n * p\ |\ > p) = 0$$

et que $\mid \epsilon_1 \mid \leq p$ et $\mid \epsilon_2 \mid \leq p$. Ces hypothèses sont importantes. En effet, le nombre d'états de la chaîne est proportionnel au nombre d'entrées dans le tampon mais aussi aux relations qui existent entre p, ϵ_1 et ϵ_2.

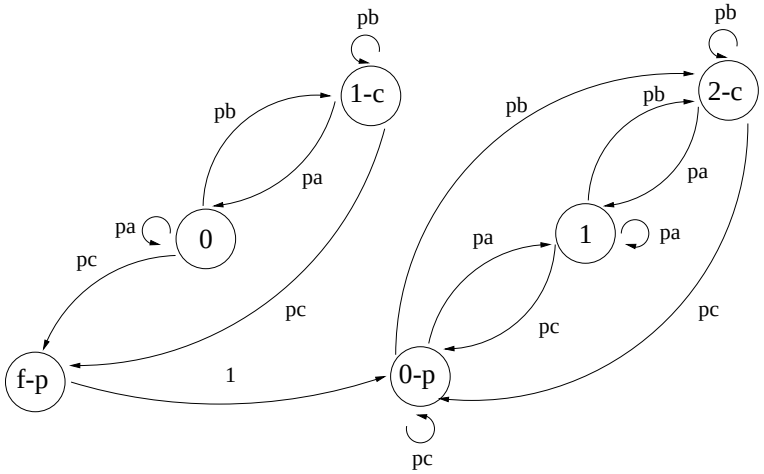


FIG. 9.7 – *Tampon avec deux entrées*

Enumérons maintenant les états du système. La chaîne de Markov relate l'état du tampon à chaque top de l'horloge. A chaque top, on évalue la modification de la taille du tampon. Avec les hypothèses ci-dessus, le tampon peut croître ou diminuer d'une unité, ou encore ne pas évoluer. Selon les cas de figure, il est possible qu'une opération de consommation ou de production associée au top i soit prise en compte dans notre modèle à la fin de la période d'horloge. Soit k la taille du tampon, au top i, la taille du tampon est donc représentée par trois sommets (cf figure 9.7) :

- L'état étiqueté k signifie que le tampon contient k éléments et que toutes les opérations de production et de consommation du top i sont effectuées.
- L'état étiqueté $k - c$ signifie que le tampon contient k éléments et qu'une opération de consommation reste à comptabiliser.
- L'état étiqueté $k - p$ signifie que le tampon contient k éléments et qu'une opération de production reste à comptabiliser.

Enfin, l'état $f - p$ signifie que le système est en situation de famine et qu'une opération de production interviendra d'ici la fin de la période d'horloge.

Nous énumérons maintenant les transitions du modèle. On observe la date d'arrivée de l'occurrence i des signaux E et S par rapport au $i$ème top de l'horloge H_p. On suppose que les occurrences des signaux sont indépendantes les unes des autres. Les événements modélisés par la chaîne de Markov sont les suivants :

a) S arrive sur le top et E intervient dans la zone m_1. La taille du tampon ne change pas. $(P(a) = m_1 * (1 - s - j))$.

b) S intervient avant le top et E dans la zone m_1. La taille du tampon n'évolue pas. $(P(b) = m_1 * j)$.

c) S intervient après le top et E dans la zone m_1. La taille du tampon augmente et une opération de consommation reste à effectuer. $(P(c) = m_1 * s)$.

d) S arrive sur le top et E intervient dans la zone m_2. La taille du tampon ne change pas. $(P(d) = m_2 * (1 - s - j))$.

e) S intervient avant le top et E dans la zone m_2. La taille du tampon ne change pas. $(P(e) = m_2 * j)$.

f) S intervient après le top et E dans la zone m_2. La taille du tampon augmente et une opération de consommation reste à effectuer. $(P(f) = m_2 * s)$.

g) S arrive sur le top et E intervient dans la zone m_3. La taille du tampon diminue et une opération de production reste à effectuer. $(P(g) = m_3 * (1 - s - j))$.

h) S intervient avant le top et E dans la zone m_3. La taille du tampon diminue et une opération de production reste à effectuer. $(P(h) = m_3 * j)$.

i) S intervient après le top et E dans la zone m_3. La taille du tampon ne change pas. $(P(i) = m_3 * s)$.

j) S arrive sur le top et E intervient dans la zone m_4. La taille du tampon diminue et une opération de consommation reste à effectuer. $(P(j) = m_4 * (1 - s - j))$.

k) S intervient avant le top et E dans la zone m_4. La taille du tampon diminue et une opération de consommation reste à effectuer. $(P(k) = m_4 * j)$.

l) S intervient après le top et E dans la zone m_4. La taille du tampon ne change pas. $(P(l) = m_4 * s)$.

D'où les probabilités de la chaîne de Markov :

- La taille du tampon ne change pas : $pa = P(a) + P(b) + P(d) + P(e) + P(i) + P(l) = s(m_3 + m_4) + (1 - s)(m_1 + m_2)$.
- La taille augmente et une opération de consommation doit avoir lieu : $pb = P(c) + P(f) = s(m_1 + m_2)$.
- La taille diminue et une opération de production doit avoir lieu : $pc = P(g) + P(h) + P(j) + P(k) = (1 - s)(m_3 + m_4)$.

Résolvons le modèle. Nous cherchons à évaluer les probabilités des différents états de celui-ci. Le graphe de la figure 9.7 contient deux composantes fortement connexes. Nous nous intéressons plus particulièrement au comportement de l'application en régime permanent. Nous allons donc ignorer la composante constituée des nœuds $1-c, 0$ et $f-p$. La seconde composante fortement connexe constitue une chaîne de Markov apériodique ; elle est donc régulière et un régime permanent existe. Soit $\pi_* = \pi_0 + \pi_1 + \pi_2$, la somme des probabilités des états de la chaîne en régime permanent ; si l'on applique successivement le

théorème des coupes de Lemaire [LEM 78] en isolant chacun des nœuds, on obtient alors le système d'équations :

$$\begin{cases} (pb+pa)\pi_0 &= pc(\pi_1+\pi_2) \\ (pb+pc)\pi_1 &= pa(\pi_0+\pi_2) \\ (pc+pa)\pi_2 &= pb(\pi_1+\pi_0) \\ 1 &= \pi_0+\pi_1+\pi_2 \end{cases}$$

Il est alors trivial d'en déduire les probabilités de chaque état :

$$\begin{cases} \pi_0 &= \left(1+\frac{pb+pa}{pc}\right)^{-1} \\ \pi_1 &= \left(1+\frac{pb+pc}{pa}\right)^{-1} \\ \pi_2 &= \left(1+\frac{pc+pa}{pb}\right)^{-1} \end{cases}$$

9.2.2.2 Application du modèle

Etat/Taille	Observé total	Observé permanent	Calculé
Famine	0,0018	0	-
0	0,0002	0	0,0070
1	0,2822	0,2797	0,1964
2	0,7153	0,7198	0,7966
Débordement	0,0005	0,0005	-

TAB. 9.3 – *Etat/Taille du tampon du compensateur*

Pour terminer ce chapitre, nous instancions le modèle proposé ci-dessus avec les informations obtenues lors de l'exécution de l'application. Les probabilités sont initialisées comme suit :

- L'émission des signaux $LIVRAISON$ est effectuée la plupart du temps en retard vis-à-vis de l'horloge H_p. Compte tenu de l'heuristique utilisée pour calculer les échéances et les dates d'éligibilité (cf. chapitre 6), les signaux $LIVRAISON$ ne peuvent a priori pas être livrés avant le top de l'horloge. Nous choisissons la valeur de 10^{-7} pour représenter la probabilité d'un événement rare (probabilité que nous estimons suffisamment négligeable). Après mesure, il est possible de poser $s = 0,8311$ et donc $j = 10^{-7}$.
- La connexion ATM fournit dans 98,7 % des cas une gigue inférieure à 900 μs. Nous posons donc $m = 0,98692$.
- Après mesure, on constate que 97,12 % des signaux $TRANSFERT$ arrivent dans la zone m_2 (soit $m_2 = 0,97124 * m$ et $m_3 = m - m_2$).

– Enfin, compte tenu de l'heuristique utilisée pour calculer les échéances et les dates d'éligibilité, si un signal $TRANSFERT$ arrive hors de la fenêtre autorisée (cf. équation (9.2)), il arrive très rarement en avance (soit $m_1 = 10^{-7} * (1 - m)$ et donc $m_4 = (1 - m) - m_1$).

Le tableau 9.3 compare l'état du tampon observé lors de l'exécution de l'application avec les résultats fournis par la chaîne de Markov. La première colonne donne la taille du tampon ainsi que la probabilité de famine et de débordement sur l'ensemble de l'application (durant laquelle nous avons traité 10000 images). Pour pouvoir uniquement comparer le fonctionnement en régime permanent, la deuxième colonne propose les statistiques concernant la livraison des images 201 à 10000. Hormis l'absence de famine et d'un tampon vide, on constate peu de différences sur les autres états. Enfin, la dernière colonne contient les probabilités estimées par la chaîne de Markov.

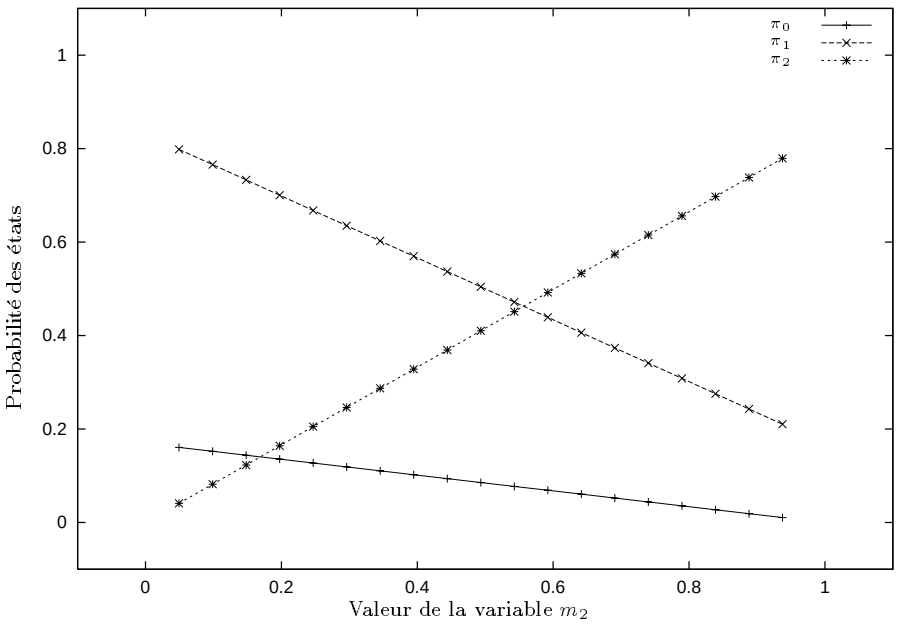


FIG. 9.8 – *Evolution de π_0, π_1 et π_2 en fonction de m_2*

Le modèle semble donc corroborer les données obtenues de façon expérimentale et, dans la plupart des cas, la taille du tampon est de deux entrées. Notons que cette dernière information dépend essentiellement du taux de charge du processeur sur le site émetteur. En effet, nous avons exécuté l'application sur deux machines peu chargées. De ce fait, une grande partie des entrées (signal $TRANSFERT$) interviennent dans la zone m_2 et très peu dans la zone m_3, puisque le thread émetteur dispose du processeur dès sa date d'éligibilité atteinte. Cette tendance s'inverse progressivement lorsque la charge augmente sur le site émetteur. En effet, le thread émetteur doit attendre de plus en plus souvent que le proce-

esseur lui soit alloué. De ce fait, le rapport entre m_2 et m_3 s'inverse et les probabilités d'avoir un ou deux éléments dans le tampon changent (cf. figure 9.8).

9.3 Conclusion

Dans ce chapitre, nous avons présenté un exemple d'application où l'utilisation d'un modèle déterministe avec POLKA n'était pas appropriée. Nous avons décrit cette application et nous avons proposé une spécification déterministe, puis, une spécification probabiliste. Nous avons montré qu'une spécification déterministe pouvait sur-contraindre le système et ainsi diminuer la qualité de service délivrée à l'utilisateur (cf. tableau 9.2). De même, nous avons constaté que l'utilisation d'un modèle stochastique permettait d'affiner la spécification temporelle de l'application, tout en autorisant la spécification de contraintes de fiabilité (exemples : taux de famine ou de débordement), mais surtout en diminuant les ressources processeurs consommées. Il semble clair que, pour certaines applications multimédias, le modèle déterministe de POLKA est insuffisant.

Toutefois, nous n'avons pas réellement cherché à intégrer les notions de contraintes probabiliste à notre modèle de spécification et une solution couvrant les différents besoins suscités par la présence de contraintes probabilistes (flots à débits variables, contraintes de fiabilité, QoS utilisateur probabiliste, etc.) reste à étudier. Le choix du modèle est donc un problème ouvert et il existe plusieurs formalismes candidats à ce jour (exemples : chaîne de Markov, réseaux de Petri stochastiques et temporels, réseaux de files d'attente, etc).

Chapitre 10

Conclusions et perspectives

Cette thèse traite du support des applications multimédias réparties. Elle se concentre sur la spécification de leurs contraintes temporelles et sur l'allocation de leurs ressources. Nous nous intéressons plus particulièrement aux applications dont les besoins en ressources peuvent évoluer pendant leur exécution (notamment du fait d'une demande de l'utilisateur) ou dont les besoins en ressources sont difficiles à évaluer (notamment du fait des données ou dispositifs manipulés).

Notre solution consiste à séparer les spécifications fonctionnelles et temporelles d'une application. Le développeur décrit le comportement fonctionnel de son application en termes d'objets et de threads "à la Clouds". Puis, il spécifie son comportement temporel en termes de composants, d'équations de QoS et de graphe de flots de données. La séparation des aspects fonctionnels et qualitatifs d'une application facilite son développement, sa maintenance et sa portabilité. La spécification temporelle est alors exploitée de façon automatique pour déduire les directives de gestion des ressources du système. La thèse traite principalement de la gestion des ressources processeurs en proposant des algorithmes d'ordonnancement. Il a toutefois été montré par d'autres travaux que le modèle que nous proposions permettait de générer les directives de ressources différentes (cf. l'étude de la ressource ATM [DRI 00]).

Nos algorithmes ont été intégré dans une plate-forme, la plate-forme POLKA. Ainsi, une fois modélisé les contraintes temporelles d'une application, la plate-forme se charge d'ordonnancer automatiquement les threads du système conformément à leurs contraintes temporelles.

Bien que les techniques présentées dans cette thèse puissent être utilisées dans tout système à objets où les interactions entre objets sont observables, notre implantation actuelle est basée sur un bus à objets CORBA 2 sur Solaris et Linux. Ce choix a simplifié sa mise en œuvre. La plate-forme est constituée d'un environnement d'exécution sous la forme d'une application CORBA, ainsi que d'un environnement de développement sous la forme d'un jeu de compilateurs. L'environnement d'exécution intercepte les invocations de méthode CORBA pour les ordonnancer en respectant les contraintes temporelles. L'envi-

ronnement de développement permet à l'utilisateur de construire ses applications CORBA et de spécifier ses contraintes de QoS.

Nous avons montré, par la réalisation de plusieurs applications, que ces environnements répondaient bien aux objectifs visés. En particulier, nous avons évalué la capacité de notre plate-forme à respecter les contraintes temporelles d'une application ainsi que le surcoût qu'elle génère.

Les contributions de cette thèse sont donc les suivantes :

- Sur le plan de la modélisation, nous avons proposé un modèle permettant de spécifier les contraintes temporelles de bout en bout de façon modulaire. Nous l'avons illustré à travers de nombreux exemples. Notre modèle utilise l'abstraction de graphe de flots de données pour décrire les traitements effectués par les applications multimédias. Le graphe de flots de données est un paradigme fréquemment utilisé dans les applications multimédias [BOU 95, POS 97, SIE 97]. Son utilisation est d'ailleurs courante dans les plates-formes multimédias [JEF 91, HOR 92, AND 93, JEF 95, MIC 97, MIT 99].

- Sur le plan algorithmique, nous avons proposé des algorithmes d'allocation du processeur. Nous avons montré qu'il était possible de construire des algorithmes raisonnablement complexes permettant d'exploiter de façon automatique une spécification de QoS. Nos algorithmes d'ordonnancement sont orientés EDF [LIU 73]. Contrairement à d'autres solutions, notre ordonnanceur ne nécessite pas une connaissance a priori du temps d'exécution des tâches du système [NIE 97, JON 97].

- Sur le plan architectural enfin, nous avons proposé et testé une architecture implantant nos propositions algorithmiques. Comme DIMMA ou TAO, notre plate-forme est basée sur un bus à objets au standard CORBA [DON 98, GIL 00]. Néanmoins, grâce à notre modèle de spécification, les contraintes temporelles exprimables dans notre plate-forme sont plus variées et mieux adaptées à celles rencontrées dans les applications multimédias.

Les perspectives de nos travaux sont nombreuses. Les plus importantes concernent la fourniture de garantie de service sur l'allocation des ressources et le support des contraintes probabilistes.

Dans le premier objectif visé, il s'agit de garantir à l'utilisateur le respect des contraintes temporelles qu'il a spécifiées dans ses applications. Pour ce faire, le modèle de spécification que nous proposons dans cette thèse doit intégrer un modèle de ressources permettant de vérifier que le système dispose des ressources nécessaires [LEB 98]. Cette extension est actuellement en cours d'étude dans le cadre d'une collaboration entre l'ENST et le CNET. Afin de disposer d'un environnement de validation, un portage de notre plate-forme dans un environnement ATM a été réalisé [DRI 00]. Il consiste principalement à intégrer dans le bus à objets omniORB2 une couche de communication utilisant le service CBR offert par ATM/AAL5. Notons qu'un travail similaire a été effectué sur Jonathan, le bus à objets Java du CNET qui, à terme, devrait être supporté par la plate-forme POLKA [DUM 98, SEI 99].

Le second point, le support des contraintes probabilistes, nous semble aussi une voie intéressante à explorer. En effet, le modèle de QoS proposé à ce jour reste limité à des contraintes **déterministes** pour lesquelles nous avons étudié le pire cas. De ce fait, nos spécifications peuvent surcontraindre le comportement temporel des applications [BAI 96]. Nous avons montré dans le chapitre 9 les gains importants, en termes de ressources processeurs, qu'un ordonnanceur tel que celui de POLKA pouvait espérer par l'utilisation de contraintes probabilistes. Le support de ces contraintes est donc important. Il l'est d'autant plus qu'elles permettent de spécifier des éléments, contraintes ou comportements du système que notre modèle actuel ne peut efficacement exprimer ; c'est le cas des contraintes de fiabilité et des trafics dont le débit est variable. Nombreux sont les travaux ou standards, y compris le projet POLKA, qui se soustraient à ces contraintes grâce à des hypothèses simplificatrices. Pour les contraintes de fiabilité, le standard MPEG 2 en est un excellent exemple puisque la norme stipule l'utilisation d'un réseau "idéal" pour le transport de trames MPEG (c'est-à-dire un réseau ne produisant pas d'erreur). Il est évident que, dans la pratique, ces contraintes ne peuvent être ignorées [GRI 98] et notre modèle devra à terme les supporter. Les solutions proposées dans le chapitre 9 restent très simples et de nombreuses voies sont à explorer : utilisation de réseaux de files d'attente, de modèles de trafics (tels que ceux proposés pour modéliser les trafics sur Internet [BOL 93, BOL 95, NIC 99]), etc.

D'autres points moins importants ne sont pas traités dans cette thèse.

C'est notamment le cas de nombreux problèmes de recherche opérationelle que soulèvent nos jeux d'équations, et qui pourraient donner lieu à l'ajout de fonctionnalités intéressantes dans nos outils de compilation. Dans cette thèse, nous avons toujours implicitement supposé que les jeux d'équations de QoS que nous utilisions étaient optimaux et faisables. Un jeu d'équations est dit infaisable si, quelles que soient les ressources disponibles dans le système, il existe au moins une équation du jeu qui soit violée. Un jeu d'équations est optimal s'il n'existe pas de jeu d'équations logiquement équivalent qui soit plus simple à exploiter durant l'exécution de l'application. Dans la réalité, si une application complexe est modélisée, il devient difficile de vérifier manuellement la véracité de ces propriétés. Il existe toutefois des outils de recherche opérationnelle qui permettraient probablement de répondre à ces questions. A titre d'exemple, l'optimisation d'un jeu d'équations de QoS peut être vu comme une instance du problème du sac à dos où à chaque équation est associée un coût d'exploitation par la plate-forme et une importance donnée par le développeur. De même, le problème de la faisabilité d'un jeu d'équations de QoS peut très certainement être traité par des outils de programmation linéaire.

Enfin, le support d'applications multimédias coopératives nécessitant des communications de groupe est un sujet d'une importance croissante. Ce type d'application est répandu dans les environnements que nous ciblons, notamment sur Internet où il est de plus en plus courant de rencontrer des applications de vidéo-conférence ou de réalité virtuelle. Nous envisageons d'offrir ce type de service au travers de mécanismes d'invocation sur groupes d'objets ; de tels mécanismes ont été proposés soit pour des applications réparties tolérantes aux pannes [MAF 95], soit pour des applications multimédias coopératives [DUM 98].

Chapitre 11

Références

[ACK 82] Ackermann. « Dataflow Languages ». *IEEE Computer*, 15(2):15–25, February 1982.

[AGR 92] G. Agrawal, B. Chen, W. Zhao, et S. Davari. « Guaranteeing synchronous message deadlines with the timed token protocol ». pages 468–475. in Proc. of the 12th IEEE International Conference of Distributed Computing Systems, June 1992.

[ALM 96] W. Almesberger. « Linux ATM API, Draft, Version 0.4 ». ftp://lrcftp.epfl/pub/linux/atm/api/atmapi0.4.tar.g z, July 1996.

[ALM 97] W. Almesberger. « ATM on Linux - The 4rd year ». in the proceedings of 4th International Linux Kongress, March 1997.

[ALU 92] R. Alur et T.A. Henzinger. Logics and models of real time: a survey. Dans J.W. de Bakker, K. Huizing, W.-P. de Roever, et G. Rozenberg, éditeurs, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 74–106. Springer-Verlag, 1992.

[AND 92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, et H. M. Levy. « Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism ». *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[AND 93] D. P. Anderson. « MetaScheduling for Distributed Continuous Media ». *ACM Trans. on Computer Systems*, 11(3):226–252, 1993.

[BAI 96] V. Baiceanu, C. Cowan, D. McNamee, C. Pu, et J. Walpole. « Multimedia Applications Require Adaptive CPU Scheduling ». Workshop on Resource Allocation Problems in Multimedia Systems, Washington DC, December 1996.

[BAK 78] H. G. Baker. « List Processing in Real-Time on a Serial Computer ». *Communications of the ACM*, 21(4):280–94, 1978.

[BAN 96] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, et H. Zhang. « The Tenet Real-Time Protocol Suite : Design, Implementation, and Experiences ». *IEEE/ACM Transactions on Networking*, 4(1):1–11, February 1996.

[BER 87] G. Berry, P. Couronné, et G. Gonthier. « Programmation synchrone des systèmes réactifs: le langage Esterel ». *Technique et Science Informatiques*, 6(4):305–315, 1987.

[BER 98] G. Berry. « The Foundations of Esterel ». in Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, 1998.

[BLA 76] 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.

[BLA 98] G. Blair et J. B. Stefani. *Open Distributed Processing and Multimedia.* Addison-Wesley, 1998.

[BOC 95] S. Bocking. « Communication Performance Models ». Rapport Technique, TR-95-013, International Computer Science Institute, Berkeley, March 1995.

[BOL 93] J. C. Bolot. « Characterizing End-to-End Packet Delay and Loss in the Internet ». *In Journal of High-Speed Networks*, 2(3):305–323, December 1993.

[BOL 94] J. C. Bolot, T. Turletti, et I. Wakeman. « Scalable feedback control for multicast video distribution in the Internet ». pages 58–67. In Proc. ACM SIGCOMM'94, London, UK, September 1994.

[BOL 95] J. C. Bolot, H. Crépin, et V. Garcia. « Analysis and Control of Audio Packet Loss over Packet-Switched Networks ». pages 163–174. Workshop on Network and Operating System Support for Digital Audio and Video, 1995.

[BOU 95] A. Boucher, Z. Yaar, E.J. Rubin, J.D. Palmer, et T.D.C Little. « Design and Performance of a Multi-Stream MPEG-I System Layer Encoder/Player Set ». pages 435–446. San Jose, Proc. IS&T/SPIE Symposium on Electronic Imaging Science and Technology (Multimedia Computing and Networking), February 1995.

[BRA 97a] T. Braun. « Internet Protocols for Multimedia Communications. Part I. IPng - The Foundation of Internet Protocols ». *IEEE Multimedia, July-September*, 4(3):85–90, 1997.

[BRA 97b] T. Braun. « Internet Protocols for Multimedia Communications. Part II. Resource Reservation, Transport and Applications Protocols ». *IEEE Multimedia, October-December*, 4(4):74–82, 1997.

[BRA 98] S. Brandt, G. Nutt, T. Berk, et M. Humphrey. « Soft Real-Time Application Execution with Dynamic Quality of Service Assurance ». International Workshop on Quality of Service (IWQOS'98), Napa - CA, May 1998.

[BUC 93] M.C. Buchanan et P.T. Zellweger. « Automatically Generating Consistent Schedule For Multimedia Applications ». *Multimedia Systems Journal*, 1(2):55–67, 1993.

[BUD 96] M. M. Buddhikot, G. M. Parulkar, et R. Gopalakrishnan. « Scalable Multimedia-On-Demand via World-Wide-Web (WWW) with QoS Guarantees ». pages 23–26. Sixth International Workshop on Network and Operating System Support for Digital Audio and Video, NOSSDAV'96, Zushi, Japon, April 1996.

[BUS 96] I. Busse, B. Deffner, et H. Schulzrinne. « Dynamic QoS Control of Multimedia Applications based on RTP ». *Computer Communications*, 19(1):49–58, January 1996.

[CAM 98] A. Campbell, C. Aurrecoechea, et L. Hauw. « A Survey of QoS Architectures ». *Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151, May 1998.

[CAN 94] S. Mc Canne. « Vic and RTPv2 ». Rapport Technique, Audio-Video Transport Working Group, IETF 31, San Jose, CA, December 1994.

[CAR 94] C. Cardeira et Z. Mammeri. « Ordonnancement de tâches dans les systèmes temps réel et répartis ». *APII*, 28(4):353–384, 1994.

[CCI 90] CCITT. « Recommendations H.261 : Video codec for audiovisual services at p*64 kb/s ». White book, 1990.

[CEN 95] S. Cen, C. Pu, R. Staehli, C. Cowan, et J. Walpole. « A Distributed Real-Time MPEG Video Audio Player ». Appeared in the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'95). Durham, New Hampshire, USA, April 1995.

[CEN 97] S. Cen. « *A software Feedback Toolkit and its Application in Adaptive Multimedia Systems* ». PhD Thesis, Oregon Graduate Institute of Sciences and Technology, October 1997.

[CHE 90] H. Chetto, M. Silly, et T. Bouchentouf. « Dynamic Scheduling of Real-time Tasks Under Precedence Constraints ». *Real Time Systems, The International Journal of Time-Critical Computing Systems*, 2(3):181–194, September 1990.

[CHO 96] H. Choi, J. S. Yoo, et O. B. Chang. « A New Control Service Model Based on CORBA for Distributed Multimedia Objects ». pages 467–474. In the proceedings of EUROMICRO-22, 1996.

[CLA 90] D. D. Clark et D. L. Tennenhouse. « Architectural Considerations for a New Generation of Protocols ». Dans *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990.

[CNE 96] CNET. « Requirements for a Real Time ORB ». Rapport Technique, RT/TR-96-8, ACTS Project, May 1996.

[COM 89] CompuServer. « Graphics Interchange Format Version 89a : Programming Reference ». Rapport Technique, Colombus, Ohio, July 1989.

[COT 98] F. Cottet, M. Courtes, et M. Holle. « Traitement de la gigue temporelle pour les ordonnancements temps réel par échéance ». pages 63–77. Real Time Systems, Paris, janvier 1998.

[COU 95] G. Coulson et G. Blair. « Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems ». *ACM Operating Systems Review*, 29(4):17–24, 1995.

[COU 97] G. Coulson et A. Mauthe. « Scheduling and Admission Testing for Jitter Constrained Periodic Threads ». *ACM Multimedia Systems Journal*, 5(5):337–346, 1997.

[DAS 90] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, et C. J. Wileknloh. « The Design and Implementation of the Clouds Distributed Operating System ». *Computing Systems Journal*, 3(1):11–46, Winter 1990.

[DEL 95] L. Delgrossi et L. Berger. « RFC1819 : Internet Stream Protocol Version 2 (ST2) - Protocol Specification ». Network Working Group, pages 1-109, August 1995.

[DEM 94] I. Demeure et J. Farhat. « Systèmes de processus légers : concepts et exemples ». *Technique et Science Informatiques*, 13(6):765–795, juin 1994.

[DIO 95] C. Diot. « Adaptive Applications and QoS Guaranties ». Invited paper in the MmNet'95 International Conference on Multimedia Networking. Aizu-Wakamatsu, Japan, September 1995.

[DON 98] D. I. Donaldson, M. C. Faupel, R. J. Hayton, A. J. Herbert, N. J. Howarth, A. Kramer, I. A. MacMillan, D. D. J. Orway, et S. W. Waterhouse. « DIMMA - A Multi-Media ORB ». pages 141–156. MIDDLEWARE'98. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 1998.

[DRI 00] B. Driss. « Support of Multimedia Applications by a CORBA and ATM Based Distributed System ». Master's thesis, ENST Paris, January 2000.

[DUM 98] B. Dumant, F. Horn, F. Dang-tran, et J. B. Stefani. « Jonathan : an Open Distributed Processing Environment in Java ». pages 173–190. MIDDLEWARE'98. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 1998.

[FAR 96] J. Farhat-Gissler. « *Ordonnancement automatique d'applications présentant des contraintes de QoS temporelles* ». Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications, septembre 1996.

[FEN 96] F. Feng, W. Zhao, et A. Kumar. « Bounding Application-to-Application Delays for Multimedia Traffic in FDDI-Based Communication Systems ». pages 174–185. in Proceedings of MULTIMEDIA COMPUTING AND NETWORKING, San Jose, January 1996.

[FER 90] D. Ferrari et D. C. Verma. « Buffer Space Allocation for Real-Time Channels in a Packet Switching Network ». Rapport Technique, numéro TR-90-022, International Computer Science Institute, Berkeley, June 1990.

[FOR 96] ATM Forum. « Traffic Management Specification, Version 4.0 ». Document References : af-tm-0056.000, April 1996.

[FRY 96] M. Fry, A. Seneviratne, A. Vogeland, et V. Witina. « Delivering QoS controlled Continuous Media on the World Wide Web ». pages 45–53. in Proceedings of the 4th international IFIP Workshop on Quality of Service, Paris, March 1996.

[GAG 96] M. Gagnaire et D. Kofman. *Réseaux Haut Débit : réseaux ATM, réseaux locaux, réseaux tout-optiques*. Masson-Inter Editions, Collection IIA, 1996.

[GAL 95] B. O. Gallmeister. *POSIX 4 : Programming for the Real World* . O'Reilly and Associates, January 1995.

[GAO 94] H. Gao et K. Nilsen. « Reliable General Purpose Dynamic Memory Management for Real-Time Systems ». Rapport Technique TR94–09, Iowa State University, July 1994.

[GEM 92] J. Gemmell et S. Christodoulakis. « Principles of Delay Sensitive Multimedia Data Storage and Retrieval ». *ACM Transactions on Information Systems*, 10(1):51–90, January 1992.

[GEM 97] D. J. Gemmell et C. G. Bell. « Noncollaborative Telepresentations Come of Age ». *Communications of the ACM*, 40(4):79–89, April 1997.

[GEO 96] L. George, N. Rivierre, et M. Spuri. « Preemptive and Non-Preemptive Real-time Uni-processor Scheduling ». INRIA Technical report number 2966, 1996.

[GIL 00] C. D. Gill, D. L. Levine, et D. C. Schmidt. « The Design and Performance of a Real-Time CORBA Scheduling Service ». *To appear in the International Journal of Time-critical Computing Systems*, April 2000.

[GOP 96] R. Gopalakrishnan et G. M. Parulkar. « Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing ». Proceedings of the ACM SIGMETRICS, Conference on Measurement and Modeling of Computer Systems, Philadelphia, May 1996.

[GOV 91] R. Govindan et D. P. Anderson. « Scheduling and IPC Mechanisms for Continuous Media ». pages 68–80. 13th ACM Symposium on Operating Systems Principles, October 1991.

[GRI 98] S. Gringeri, B. Khasnabish, A. Lewis, K. Shuaib, R. Egorov, et B. Basch. « Transmission of MPEG-2 video Streams over ATM ». *IEEE Multimedia*, 5(1):58–71, January 1998.

[GUE 91] P. Le Guernic, T. Gautier, M. Le Borgne, et C. Le Maire. « Programming Real Time Applications With SIGNAL ». INRIA-RENNES, Rapport numéro 1446, 1991.

[HAF 98] A. Hafid, , G. Bochmann, et R. Dssouli. « Distributed Multimedia Application and Quality of Service : a Review ». *Electronic Journal on Networks and Distributed Processing*, 2(6):1–50, 1998.

[HAL 91] N. Halbwachs, P. Caspi, P. Raymond, et D. Pilaud. « Programmation et vérification des systèmes réactifs : le langage Lustre ». *Technique et Science Informatiques*, 10(2):139–157, 1991.

[HAN 96] C. C. Han, K. J. Lin, et C. J. Hou. « Distance-constrained Scheduling and Its Applications to Real-time Systems ». *IEEE Transactions on computers*, 45(7):814–826, July 1996.

[HAR 85] D. Harel et A. Pnueli. « On the Developement of Reactive Systems ». pages 477–498. In Logic and Models of Concurrent Systems. Proc NATO Advanced Study Institute on Logics and Models for Verifications and Specification of Concurrent Systems. New York, 1985.

[HAR 87] D. Harel. « Statecharts: A visual formalism for complex systems ». *Science of Computer Programming*, 8(3):231–274, June 1987.

[HAR 97] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, et J. Wolter. « The Performance of micro-Kernel-Based Systems ». 16th ACM Symposium on Operating Systems Principles in Saint-Malo (SOSP'97) - France, October 1997.

[HEN 97] R. Henriksson. « Predictable Automatic Memory Management for Embedded Systems ». Dans Peter Dickman et Paul R. Wilson, éditeurs, *OOPSLA '97 Workshop on Garbage Collection and Memory Management*, October 1997.

[HER 97] J.F. Hermant, L. Leboucher, et N. Rivierre. « Real time fixed and dynamic priority driven scheduling algorithms : theory and experience ». Rapport de recherche de l'INRIA numéro 3081, 1997.

[HOR 92] F. Horn, L. Hazard, J. B. Stefani, G. Coulson, et G. Blair. « An Integrating Platform and Computational Model for Open Distributed Multimedia Applications ». Presented at the 3rd International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 92), San Diego, 1992.

[HUA 97] T. Y. Huang. « *Worst-Case Timing Analysis of Concurrently Executing DMA I/O and Programs* ». PhD Thesis, University of Illinois at Urbana-Champaign, April 1997.

[ISO 94] ISO/IEC. « JTC 1/SC 29/WG 11 number 702 Rev. Information Technology - Generic Coding of Moving Pictures and associated Audio, recommandation H 262." ». Draft International Standard, Paris 25 March, 1994.

[ISO 95] ISO. « Press Release, 29th Meeting of JTC 1/SC 29/WG 11 ». number 1110, March 1995.

[JAH 86] F. Jaharian et A. K. Mok. « Safety Analysis of Timing Properties in Real-Time Systems ». *IEEE Trans. on Software Engineering*, 12(9):890–904, September 1986.

[JEF 91] K. Jeffay, D.L. Stone, et D.E. Poirier. « YARTOS: Kernel support for efficient, predictable real-time systems ». *Proc. joint Eight 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, May 1991.

[JEF 92a] K. Jeffay. « On Kernel Support for Real-Time Multimedia Applications ». Dans *Proceedings Third IEEE Workshop on Workstation Operating Systems*, pages 39–46, April 1992.

[JEF 92b] K. Jeffay, D. L. Stone, et F. D. Smith. « Kernel Support for Live Digital Audio and Video ». *Computer Communications*, 15(6):388–395, August 1992.

[JEF 95] K. Jeffay et D. Bennett. « A Rate-Based Execution Abstraction For Multimedia Computing ». *In Lectures Notes in Computing Science, T. D. C. Little and R. Gusella, Springer-Verlag, Heidelberg*, 1018:64–75, April 1995.

[JOH 97] M. S. Johnstone. « *Non-Compacting Memory Allocation and Real-Time Garbage Collection* ». PhD Thesis, University of Texas at Austin, December 1997.

[JON 97] M. Jones, D. Rosu, et M. Rosu. « CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities ». 16th ACM Symposium on Operating Systems Principles in Saint-Malo (SOSP'97) - France, October 1997.

[KAI 98] C. Kaiser et C. Santellani. « Petrarque : plate-forme d'exécution pour l'ordonnancement adaptatif Temps réel strict d'application réparties ». *Technique et Science Informatiques*, 17(1):39–62, janvier 1998.

[KAO 94] B. Kao et H. Garcia-Molina. « Subtask Deadline for Complex Distributed soft Real-Time Tasks ». Proceedings of ICDCS'94 International Conference on Distributed Computing Systems, April 1994.

[KRA 85] S. Krakowiak. *Principe des systèmes d'exploitation des ordinateurs.* Dunod Informatique, 1985.

[LAU 98] M. Laubach et J. Alpern. « RFC2225 : Classical IP and ARP over ATM ». Network Working Group, pages 1-17, April 1998.

[LEB 95] L. LEBOUCHER et J.B. STEFANI. « Admission Control for end to end distributed bindings ». Centre National d'Etudes des Télécommunications (CNET), Lecture Notes in computer science, Teleservice and Multimedia Communications, vol 1052, November 1995.

[LEB 98] L. LEBOUCHER. « *Algorithmique et Modélisation pour la Qualité de Service des Systèmes Répartis Temps Réel* ». Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications de Paris, septembre 1998.

[LEH 90] J. P. LEHOCZKY. « Fixed priority scheduling of periodic task sets with arbitrary deadlines ». pages 201–209. in Proc. 11th IEEE Real Time Systems Symposium, Lake Buena Vista, December 1990.

[LEM 78] B. LEMAIRE. « *Contribution à l'étude des systèmes avec attente : les méthodes de conservation* ». Thèse de doctorat d'état, Université de Paris VI, 1978.

[LI 90] K. LI. « Real-Time Concurrent Collection in User Mode ». Dans Eric JUL et Niels-Christian JUUL, éditeurs, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[LIN 88] K. J. LIN et S. NATARAJAN. « Expressing and Maintaining Timing Constraints in Flex ». pages 96–105. In Real Time Systems Symposium, December 1988.

[LIU 73] C. L. LIU et 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.

[LO 98] S. L. Lo et S. POPE. « The implementation of a High Performance ORB over Multiple Network Transports ». pages 157–172. MIDDLEWARE'98. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 1998.

[MAF 95] S. MAFFEIS. « *Run Time Support for Object Oriented Distributed Programming* ». PhD Thesis, Université de Zurich, February 1995.

[MER 94] C. W. MERCER, S. SAVAGE, et H. TOKUDA. « Processor Capacity Reserves: Operating System Support for Multimedia Applications ». In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, May 1994.

[MIC 97] MICROSOFT. *PC 98 System Design Guide*. Microsoft Press, September 1997.

[MIT 99] S. MITCHELL, H. NAGUIB, G. COULOURIS, et T. KINDBERG. « A QoS Support Framework for Dynamically Reconfigurable Multimedia Applications. ». pages 17–30. in Proc. of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, Helsinki, Finland, June 1999.

[MOK 83] A.K. MOK. « *Fundamental Design Problems of Distributed Systems for the Hard-Real Time Environnement* ». PhD Thesis, Departement of electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, May 1983.

[MUE 93] F. MUELLER. « A Library Implementation of POSIX Threads under UNIX ». pages 29–42. Proc. of USENIX Technical Conference, San Diego, California, January 1993.

[NIC 99] N. NICLAUSSE. « *Modélisation, évaluation de performances et dimensionnement du World Wide Web* ». Thèse de doctorat, Université de Nice-Sophia Antipolis, juin 1999.

[NIE 93] J. Nieh, J. N. Northcutt, et J. G. Hanko. « SVR4 UNIX Scheduler are unacceptable for multimedia applications ». Dans *Lecture Notes in Computer Science, Springler-Verlag, Proceedings of the 4th International Workshop on NOSSDAV*, volume 846, pages 49–60, Lancaster, U.K., November 1993.

[NIE 97] J. Nieh et M. Lam. « The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications ». 16th ACM Symposium on Operating Systems Principles in Saint-Malo (SOSP'97) - France, October 1997.

[NIL 94] K. D. Nilsen. « Reliable Real-Time Garbage Collection of C++ ». *Computing Systems*, 7(4), 1994.

[OMG 99] OMG. « The Common Object Request Broker : Architecture and Specification. Revision 2.3 ». TC Document 99-10-07, October 1999.

[OST 92] J. S. Ostroff. « A Verifier for Real Time Properties ». *The Journal of Real Time systems*, 4:5–35, 1992.

[OWE 98] P. Owezarski et M. Diaz. « Conception et implémentation d'applications multimédias en Solaris 2 ». *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 10(3):311–331, juillet 1998.

[PHI 96] I. R. Philp, K. Nahrstedt, et J. W. S. Liu. « Scheduling and Buffer Management for Soft Real-Time VBR Traffic in Packet Switched Networks ». Dans *Proceedings, 21st Conference on Local Computer Networks*, pages 143–152, Minneapolis, Minnesota, October 1996.

[POS 97] E. J. Posnak, R. G. Lavender, et H. M. Vin. « An Adaptive Framework to Developing Multimedia Software Components ». *Communications of the ACM*, 40(10):43–47, October 1997.

[RAN 93] P. V. Rangan et H. M. Vin. « Efficient Storage Techniques for Digital Continuous Multimedia ». *IEEE trans. on Knowledge and Data Engineering*, August 1993.

[RIC 98] P. Richard. « Télévision numérique : comment choisir son "bouquet" ». *Science et Vie, 964:117-123*, janvier 1998.

[RIV 98] N. Rivierre. « *Ordonnancement temps réel centralisé, les cas préemptifs et non préemptifs* ». Thèse de doctorat, Université de Versailles Saint Quentin, février 1998.

[ROB 94] P. Robin, G. Coulson, A. Campbell, G. Blair, et M. Papathomas. « Implementing a QoS Controlled ATM Based Communications System in Chorus ». Proceedings of the 4th International Workshop on Protocols for High Performance Networks, Vancouver, Canada, 1994.

[ROD 89] M. A. Rodrigues et V. R. Saksena. « Support for Continuous Media in the DASH System ». Technical Report UCB/CSD 89/537, University of California, Berkeley, October 1989.

[ROW 93] L. A. Rowe, K. Patel, et B. C. Smith. « Performance of a Software MPEG Video Decoder ». Proc. ACM Multimedia'93, Anaheim, CA, August 1993.

[ROZ 91] M. Rozier, V. Abrassimov, et F. Armand. « Overview of the CHORUS Distributed Operating Systems ». Rapport Technique, Chorus Systèmes, CS/TR-90-25.1, February 1991.

[SAK 94] M. Saksena, J. Da Silva, et A. K. Agrawala. « Design and Implementation of Maruti-II ». pages 72–102. In Principles of Real-Time Systems, Sang son (ed.). Chapter 4, 1994.

[SAK 95] M. Saksena, R. Gerber, et W. Pugh. « Parametric Dispatching of Hard Real-time Tasks ». *IEEE Trans. on Computers*, 44(3):471–479, 1995.

[SAM 97] S. Samia-Bouzefrane et F. Cottet. « Un outil de validation temporelle des applications temps réel réparties ». pages 37–51. Real Time Systems, Paris , janvier 1997.

[SCH 96] H. Schulzrinne, S. Casner, R. Frederick, et V. Jacobson. « RFC1889 : RTP : A Transport Protocol for Real-Time Applications ». Network Working Group, pages 1-75, January 1996.

[SEI 99] L. Seinturier. « Intégration de liaisons ATM dans l'ORB CORBA Jonathan ». Rapport Technique, CNET NT/CNET/6125, janvier 1999.

[SEV 87] K. Sevcik et M. Johnson. « Cycle Time Properties of the FDDI Token Ring Protocol ». *IEEE Transactions on Software Engineering*, 13(3):376–385, March 1987.

[SIE 97] S. Siewert, G. Nutt, et M. Humphrey. « Real-Time Parametrically Controlled In-Kernel Pipelines ». Third IEEE Real Time Technology and Application Symposium (RTAS'97), Work-In-Progress, Montreal - Canada, June 1997.

[SIJ 96] P. Sijben et S. J. Mullender. « Quality of Service in Distributed Multimedia Systems ». in Trends in distributed systems, Springer Lectures Notes on computing systems, TREVS 1161, 1996.

[SOL 98] D. A. Solomon. *Inside Windows NT.* Microsoft Press, May 1998.

[SPR 89] B. Sprunt, L. Sha, et J.P. Lehoczky. « Aperiodic task scheduling for hard-real-time systems ». *The Journal of Real Time Systems*, 1:27–60, 1989.

[SRI 98] B. Srinivasan, S. Pather, R. Hill, F. Ansari, et D. Niehaus. « A Firm Real Time System Implementation using Commercial Off-The-Self Hardware and Free Software ». pages 112–119. 4th IEEE Symposium on Real-time Technology and Applications, Denver Colorado, June 1998.

[STA 95] J. Stankovic, M. Spuri, M. Di Natale, et G. Buttazzo. « Implications of Classical Scheduling Results For Real-Time Systems ». *IEEE Computer*, 28(6):16–25, June 1995.

[STE 93] 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.

[STE 95a] R. Steinmetz. « Analysing the Multimedia Operating System ». *IEEE Multimedia*, 2(1):67–84, 1995.

[STE 95b] R. Steinmetz et K. Nahrstedt. *Multimedia : Computing, communicating and applications.* Prentice Hall, innovative technology series, 1995.

[STE 96] J.B. Stefani, G.S. Blair, G. Coulson, M. Papathomas, P. Robin, F. Horn, et 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):249–263, January 1996.

[SUN 94] J. Sun, R. Bettati, et J. W. S. Liu. « An End-to-End Approach to Schedule Tasks with Shared Ressources in Multiprocessor Systems ». pages 18–22. 11th IEEE Workshop on Real Time Oprating Systems and Software - Seatle, May 1994.

[TIN 94] K. W. Tindell et J. Clark. « Holistic schedulability analysis for distributed hard real-time systems ». *Microprocessing and Microprogramming*, 40(2-3):117–134, April 1994.

[TOK 92] H. Tokuda, Y. Tobe, S. Chou, et J. Moura. « Continuous Media Communication with Dynamic QoS Control Using ARTS with an FDDI Network ». pages 88–98. ACM SIGCOMM'92, August 1992.

[VAN 98] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, et D. E. Bakken. « QuO's Runtime Support for Quality of Service in Distributed Objects ». pages 207–222. MIDDLEWARE'98. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 1998.

[VEG 97] L. Vega et J.P. Thomesse. « Vers une caracterisation temporelle des profils de communication ». pages 81–97. Real Time Systems, Paris , janvier 1997.

[VET 95] R. J. Vetter. « ATM Concepts, Architectures, and Protocols ». *Communications of the ACM*, 38(2):30–38, February 1995.

[VOG 95] A. Vogel, B. Kerhervé, G. Von Bochmann, et J. Gecsei. « Distributed Multimedia and QoS : A Survey ». *IEEE Multimedia*, 2(2):10–19, Summer 1995.

[WIC 97] I. J. Wickelgren. « The facts about FireWire ». *IEEE Spectrum*, 34(4):19–25, April 1997.

[WOL 91] V. Wolfe, S. Davidson, et I. Lee. « RTC : Language Support For Real-Time Concurrency ». pages 43–52. Proceedings of the 12th IEEE Real-time Systems Symposium. San Antonio, Texas, December 1991.

[WRA 94] S. Wray et T. Glauert. « Networked Multimedia : The Medusa Environment ». *IEEE Multimedia*, 1(4):54–63, 1994.

[ZHA 93] H. Zhang. « *Service Disciplines For Packet-Switching Integrating-Services Networks* ». PhD Thesis, University of California at Berkeley, November 1993.

[ZHE 95] Q. Zheng et K. G. Shin. « Synchronous Bandwidth Allocation in FDDI Networks ». *IEEE Transaction on Parallel and Distributed Systems*, 6(12):1332–1338, December 1995.

Annexe A

Démonstrations

A.1 Modélisation de systèmes multimédias

A.1.1 Expression de la QoS

-Montrons que :

l'équation (4.2) implique l'équation (4.3).

-Eléments de preuve :

Nous pouvons réécrire $\tau(e, n+1) - \tau(e, n)$ par

$$\begin{aligned} \tau(e, n+1) - \tau(e, n) &= \tau(e, n+1) - \tau(H_p, n+1) \\ &+ \tau(H_p, n+1) - \tau(H_p, n) \\ &+ \tau(H_p, n) - \tau(e, n) \end{aligned}$$

Or, nous avons par hypothèse :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(e, n+1) - \tau(H_p, n+1) \leq \epsilon_2$$

et

$$\tau(H_p, n+1) - \tau(H_p, n) = p$$

et enfin :

$$\forall\ n\ :\ -\epsilon_2 \leq \tau(H_p, n) - \tau(e, n) \leq -\epsilon_1$$

Ce qui nous permet d'obtenir, en sommant les trois équations ci-dessus que :

$$\forall\ n\ :\ p - (\epsilon_2 - \epsilon_1) \leq \tau(e, n+1) - \tau(e, n) \leq p + (\epsilon_2 - \epsilon_1)$$

L'équation (4.2) implique donc bien l'équation (4.3).

A.1.2 QoS requise du composant à retard fixe avec horloge logique

-Montrons que :

l'équation

$$\forall\ n\ :\ \epsilon_1 + l \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + l$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'}, n) - \tau(S, n) \leq \epsilon_2$$

-Eléments de preuve :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'}, n) - \tau(S, n) \leq \epsilon_2$$

devient par substitution :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'}, n) - \tau(E, n) - l \leq \epsilon_2$$

et donc la QoS requise :

$$\forall\ n\ :\ \epsilon_1 + l \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + l$$

A.1.3 QoS requise du composant à retard fixe sans horloge logique

-Montrons que :

l'équation

$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(E, n + r) - \tau(E, n) \leq \epsilon_2$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(S, n + r) - \tau(S, n) \leq \epsilon_2$$

-Eléments de preuve :

La preuve est triviale puisque la QoS offerte :

$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(S, n + r) - \tau(S, n) \leq \epsilon_2$$

devient par substitution :

$$\forall\ n, r\ :\ \epsilon_1 \leq (\tau(E, n + r) + l) - (\tau(E, n) + l) \leq \epsilon_2$$
$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(E, n + r) - \tau(E, n) \leq \epsilon_2$$

A.1.4 QoS requise du composant à retard variable sans horloge logique

-Montrons que :

l'équation

$$\forall\ n, r\ :\ \epsilon_1 - \beta + \alpha \leq \tau(E, n+r) - \tau(E, n) \leq \epsilon_2 + \beta - \alpha$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n, r\ :\ \epsilon_1 \leq \tau(S, n+r) - \tau(S, n) \leq \epsilon_2$$

-Eléments de preuve :

Soit $\tau(E, n+r) - \tau(E, n)$ que nous réécrivons par :

$$\begin{array}{rcl} \tau(E, n+r) - \tau(E, n) & = & \tau(E, n+r) - \tau(S, n+r) \\ & & +\tau(S, n+r) - \tau(S, n) \\ & & +\tau(S, n) - \tau(E, n) \\ & = & -(\tau(S, n+r) - \tau(E, n+r)) \\ & & +(\tau(S, n+r) - \tau(S, n)) \\ & & +(\tau(S, n) - \tau(E, n)) \end{array}$$

Or, nous avons :

$$\forall\ n, r\ :\ \alpha \leq \tau(S, n+r) - \tau(E, n+r) \leq \beta$$

et donc :

$$\forall\ n, r\ :\ -\beta \leq -(\tau(S, n+r) - \tau(E, n+r)) \leq -\alpha$$

Comme nous avons aussi :

$$\epsilon_1 \leq \tau(S, n+r) - \tau(S, n) \leq \epsilon_2$$

et

$$\forall\ n\ :\ \alpha \leq \tau(S, n) - \tau(E, n) \leq \beta$$

En sommant les trois équations précédentes, on obtient la QoS requise suivante :

$$\forall\ n, r\ :\ \epsilon_1 - \beta + \alpha \leq \tau(E, n+r) - \tau(E, n) \leq \epsilon_2 + \beta - \alpha$$

A.1.5 QoS requise du composant à retard variable avec horloge logique

-Montrons que :

l'équation

$$\forall\ n\ :\ \epsilon_1 + \beta \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + \alpha$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'}, n) - \tau(S, n) \leq \epsilon_2$$

-Eléments de preuve :

Soit l, le retard provoqué par le composant, nous procédons comme pour la preuve du composant de retard fixe :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'}, n) - \tau(S, n) \leq \epsilon_2$$

devient :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(H'_{p'}, n) - \tau(E, n) - l \leq \epsilon_2$$
$$\forall\ n\ :\ \epsilon_1 + l \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + l$$

Nous cherchons les conditions, qui, pour toute valeur de l (telle que $\alpha \leq l \leq \beta$), impliquent que l'équation précédente soit vraie. Il faut regarder le pire cas parmi :

$$\forall\ n\ :\ \epsilon_1 + \alpha \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + \alpha$$

et

$$\forall\ n\ :\ \epsilon_1 + \beta \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + \beta$$

Le pire cas intervient lorsque :

$$\forall\ n\ :\ max(\epsilon_1 + \alpha, \epsilon_1 + \beta) \leq \tau(H'_{p'}, n) - \tau(E, n) \leq min(\epsilon_2 + \alpha, \epsilon_2 + \beta)$$

d'où la QoS requise suivante :

$$\forall\ n\ :\ \epsilon_1 + \beta \leq \tau(H'_{p'}, n) - \tau(E, n) \leq \epsilon_2 + \alpha$$

A.1.6 Le composant compensateur de gigue

-Montrons que :

le compensateur de gigue constitue bien une instance du problème étudié dans [GAG 96] et que les équations utilisées spécifient les bonnes contraintes.

-Eléments de preuve :

Nous revenons sur le problème de l'émission de cellules ATM par une AAL1 et nous commençons par déterminer les écarts maximal et minimal séparant l'arrivée d'une cellule i et d'une cellule $i+1$ chez le récepteur. Nous adoptons les notations suivantes :

- $\triangle = M - m$ est la gigue maximale sur le temps de communication (avec m le temps minimal de communication et M le temps maximal).
- b_i est le temps de communication de la cellule i.
- τ_0 la date d'émission de la première cellule.
- e_i la date d'émission de la cellule i.
- r_i la date de réception de la cellule i.
- p est la cadence d'émission (et donc de réception).

Notons e_{max}, l'écart maximal, le pire cas intervient quand $b_i = m$ et $b_{i+1} = M$. Ainsi $r_i = e_i + m = \tau_0 + p.i + m$, de même que $r_{i+1} = e_{i+1} + M = \tau_0 + p.(i+1) + M$. On en déduit que $e_{max} = r_{i+1} - r_i = M - m + p$.

On opère de même pour e_{min}, l'écart minimal, dont le pire cas intervient lorsque $b_i = M$ et $b_{i+1} = m$. Ainsi $r_i = e_i + M = \tau_0 + p.i + M$, de même que $r_{i+1} = e_{i+1} + m = \tau_0 + p.(i+1) + m$. On en déduit que $e_{min} = r_{i+1} - r_i = m - M + p$. On peut alors conclure si $\tau(r,i)$ est la date de réception de la $i$ème cellule que dans le problème traité dans [GAG 96] :

$$\forall\ i\ :\ p - M + m \leq \tau(r, i+1) - \tau(r,i) \leq p + M - m$$

ou encore

$$\forall\ i\ :\ p - \triangle \leq \tau(r, i+1) - \tau(r,i) \leq p + \triangle$$

Or dans la partie 4.3, nous avons vu que

$$\forall\ n\ :\ \epsilon_1 \leq \tau(e,n) - \tau(H_p, n) \leq \epsilon_2$$

où e est un événement et H_p est une horloge de période p, implique que

$$\forall\ n\ :\ p - (\epsilon_2 - \epsilon_1) \leq \tau(e, n+1) - \tau(e,n) \leq p + (\epsilon_2 - \epsilon_1)$$

Si l'on pose $\triangle = \epsilon_2 - \epsilon_1$, on obtient la même contrainte temporelle induite par la transmission d'un flot CBR (*Constant Bit Rate*) sur une couche AAL1. Le composant compensateur de gigue est donc bien une instance du problème traité dans [GAG 96].

A.1.7 Généralisation du composant compensateur de gigue

-Montrons que :

les propriétés (1) et (2) sont vraies.

-Eléments de preuve :

La preuve ci-dessous est inspirée de celle donnée dans [GAG 96]. Nous utilisons les notations suivantes :

- τ_0 la date d'émission de la première cellule.
- a_i la distance en unités de temps entre τ_0 et l'émission de la cellule i.
- b_i le temps de communication de la cellule i (compris entre m et M).
- t_i^e la date d'émission de la cellule i.
- t_i^r la date de réception de la cellule i.
- t_i^d la date de livraison de la cellule i.

Dans un premier temps, nous cherchons à évaluer l, le temps de retard à appliquer chez le récepteur pour la livraison du flot afin d'éviter toute famine du tampon :

- On sait que $t_1^d = t_1^e + b_1 + l$.
- et que $\forall\ i \neq 1\ :\ t_i^d \in [t_1^d + a_i + \epsilon_1\ ;\ t_1^d + a_i + \epsilon_2]$.
- On sait aussi que $t_i^r = t_1^e + a_i + b_i$.
- Pour qu'il n'y ait pas de famine du tampon, il faut que $\forall\ i\ :\ t_i^d - t_i^r \geq 0$. Le pire cas intervient lorsque $t_i^d = t_1^d + a_i + \epsilon_1$, il faut donc rechercher le cas où :

$$\begin{array}{rl} t_i^d - t_i^r & \geq 0 \\ t_1^d + a_i + \epsilon_1 - t_i^r & \geq 0 \\ t_1^e + b_1 + l + a_i + \epsilon_1 - t_i^r & \geq 0 \\ t_1^e + b_1 + l + a_i + \epsilon_1 - t_1^e - a_i - b_i & \geq 0 \\ b_1 + l + \epsilon_1 - b_i & \geq 0 \\ l & \geq b_i - b_1 - \epsilon_1 \end{array}$$

Ainsi, si $l \geq b_i - b_1 - \epsilon_1$ est vrai, alors il n'y aura pas de famine. Cherchons maintenant le cas le plus défavorable (cas qui maximise la valeur de l). Ce dernier intervient lorsque $b_1 = m$ et $b_i = M$, ce qui nous conduit à $l \geq M - m - \epsilon_1$, ou finalement $l = \triangle - \epsilon_1$ est une condition nécessaire et suffisante pour éviter une famine dans le tampon.

Nous regardons maintenant la taille minimale du tampon qui garantit, étant donné les gigues sur la livraison et les communications, qu'aucun débordement du tampon n'interviendra. A cet effet, nous cherchons le temps de transit maximal d'une cellule dans le tampon.

- On cherche donc le temps de transit maximal d'une cellule i dans le tampon, ou encore $max(j)$ tel que $j = t_i^d - t_i^r$.
- On sait que $t_i^r = t_1^e + a_i + b_i$. et que $t_i^d \in [t_1^d + a_i + \epsilon_1 \ ; \ t_1^d + a_i + \epsilon_2]$.
- La cellule i arrive au plus tôt chez le récepteur lorsque $b_i = m$, c'est-à-dire à l'instant $t_i^r = t_1^e + a_i + b_i = t_1^e + a_i + m$.
- La cellule est livrée au plus tard à la date $t_i^d = t_1^d + a_i + \epsilon_2$. Or, comme le flot est retardé de $b_1 + l$ à partir de τ_0, la première cellule est délivrée au plus tard en $t_1^d = t_1^e + M + l$ (car $b_1 = M$), et donc la cellule i est livrée au plus tard en $t_i^d = t_i^d + a_i + \epsilon_2 = t_1^e + M + l + a_i + \epsilon_2$.
- De là, on déduit que :

$$\begin{array}{rcc} j & = & t_i^d - t_i^r \\ j & = & t_1^e + M + l + a_i + \epsilon_2 - t_i^r \\ j & = & t_1^e + M + l + a_i + \epsilon_2 - (t_1^e + a_i + m) \\ j & = & M + l + \epsilon_2 - m \end{array}$$

or, comme $l = \triangle - \epsilon_1$, on a finalement $j = 2 * \triangle - \epsilon_1 + \epsilon_2$ ou encore $j = 2 * \triangle + \epsilon$.

A.1.8 Composition parallèle avec des composants de retard fixe

-Montrons que :

l'équation

$$\forall \ n \ : \ \epsilon_1 - l_1 + l_2 \leq \tau(E_1, n) - \tau(E_2, n) \leq \epsilon_2 - l_1 + l_2$$

constitue la QoS requise pour la QoS offerte

$$\forall \ n \ : \ \epsilon_1 \leq \tau(S_1, n) - \tau(S_2, n) \leq \epsilon_2$$

-Eléments de preuve :

En partant de la QoS offerte et par substitution, on obtient :

$$\begin{array}{r} \forall \ n \ : \ \epsilon_1 \leq \tau(S_1, n) - \tau(S_2, n) \leq \epsilon_2 \\ \forall \ n \ : \ \epsilon_1 \leq \tau(E_1, n) + l_1 - \tau(E_2, n) - l_2 \leq \epsilon_2 \end{array}$$

et finalement :

$$\forall\ n\ :\ \epsilon_1 - l_1 + l_2 \leq \tau(E_1, n) - \tau(E_2, n) \leq \epsilon_2 - l_1 + l_2$$

qui constitue la QoS requise par le composant m pour satisfaire l'équation (4.17).

A.1.9 Composition parallèle avec des composants de retard variable

-Montrons que :

l'équation

$$\forall\ n\ :\ \epsilon_1 - R_1 + r_2 \leq \tau(E_1, n) - \tau(E_2, n) \leq \epsilon_2 - r_1 + R_2$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n\ :\ \epsilon_1 \leq \tau(S_1, n) - \tau(S_2, n) \leq \epsilon_2$$

-Eléments de preuve :

Nous réécrivons $\tau(E_1, n) - \tau(E_2, n)$ par :

$$\begin{aligned} \tau(E_1, n) - \tau(E_2, n) \quad &= \tau(E_1, n) + \tau(S_1, n) \\ &\quad -\tau(S_1, n) \\ &\quad -\tau(E_2, n) + \tau(S_2, n) \\ &\quad -\tau(S_2, n) \end{aligned}$$

Or, nous avons :

$$\forall\ n\ :\ \epsilon_1 \leq \tau(S_1, n) - \tau(S_2, n) \leq \epsilon_2$$

et

$$\forall\ n\ :\ -R_1 \leq \tau(E_1, n) - \tau(S_1, n) \leq -r_1$$

ainsi que

$$\forall\ n\ :\ r_2 \leq \tau(S_2, n) - \tau(E_2, n) \leq R_2$$

En sommant les trois équations précédentes, on obtient finalement la QoS requise suivante :

$$\forall\ n\ :\ \epsilon_1 - R_1 + r_2 \leq \tau(E_1, n) - \tau(E_2, n) \leq \epsilon_2 - r_1 + R_2$$

A.1.10 Composition séquentielle avec des composants de retard fixe

-Montrons que :

l'équation

$$\forall\ n,k\ :\ \epsilon_1 \leq \tau(E,n+k) - \tau(E,n) \leq \epsilon_2$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n,k\ :\ \epsilon_1 \leq \tau(S,n+k) - \tau(S,n) \leq \epsilon_2$$

-Eléments de preuve :

On sait que

$$\begin{aligned}
\tau(S,n+k) &= \tau(J_r,n+k)\\
&= \tau(J_{r-1},n+k) + l_r\\
&= \tau(J_{r-2},n+k) + l_r + l_{r-1}\\
&\ldots\\
&= \tau(J_{r-i},n+k) + \textstyle\sum_{j=0}^{i-1} l_{r-j}\\
&\ldots\\
&= \tau(J_1,n+k) + \textstyle\sum_{j=0}^{r-2} l_{r-j}\\
&= \tau(E,n+k) + \textstyle\sum_{j=0}^{r-1} l_{r-j}
\end{aligned}$$

de même :

$$\begin{aligned}
\tau(S,n) &= \tau(J_r,n)\\
&= \tau(J_{r-1},n) + l_r\\
&\ldots\\
&= \tau(E,n) + \textstyle\sum_{j=0}^{r-1} l_{r-j}
\end{aligned}$$

Ainsi par substitution, on obtient :

$$\begin{gathered}
\forall\ n,k\ :\ \epsilon_1 \leq \tau(S,n+k) - \tau(S,n) \leq \epsilon_2\\
\forall\ n,k\ :\ \epsilon_1 \leq \tau(E,n+k) + \textstyle\sum_{j=0}^{r-1} l_{r-j} - (\tau(E,n) + \sum_{j=0}^{r-1} l_{r-j}) \leq \epsilon_2\\
\forall\ n,k\ :\ \epsilon_1 \leq \tau(E,n+k) - \tau(E,n) \leq \epsilon_2
\end{gathered}$$

A.1.11 Composition séquentielle avec des composants de retard variable

-Montrons que :

l'équation

$$\forall\ n,k\ :\ \sum_{j=1}^{r}(-\beta_j+\alpha_j) + \epsilon_1 \leq \tau(E,n+k) - \tau(E,n) \leq \epsilon_2 + \sum_{j=1}^{r}(\beta_j - \alpha_j)$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n,k\ :\ \epsilon_1 \leq \tau(S,n+k) - \tau(S,n) \leq \epsilon_2$$

-Eléments de preuve :

En appliquant successivement le résultat énoncé dans le paragraphe 4.4.2.1 et en commençant du composant m_r jusqu'au composant m_1, on obtient :

$$\begin{gathered}
\forall\ n,k\ :\ \epsilon_1 \leq \tau(S,n+k) - \tau(S,n) \leq \epsilon_2 \\
\forall\ n,k\ :\ -\beta_r + \alpha_r + \epsilon_1 \leq \tau(J_{r-1},n+k) - \tau(J_{r-1},n) \leq \epsilon_2 + \beta_r - \alpha_r \\
\forall\ n,k\ :\ \textstyle\sum_{j=r-1}^{r}(-\beta_j+\alpha_j) + \epsilon_1 \leq \tau(J_{r-2},n+k) - \tau(J_{r-2},n) \leq \epsilon_2 + \sum_{j=r-1}^{r}(\beta_j-\alpha_j) \\
\dots \\
\forall\ n,k\ :\ \textstyle\sum_{j=r-i+1}^{r}(-\beta_j+\alpha_j) + \epsilon_1 \leq \tau(J_{r-i},n+k) - \tau(J_{r-i},n) \leq \epsilon_2 + \sum_{j=r-i+1}^{r}(\beta_j-\alpha_j) \\
\dots \\
\forall\ n,k\ :\ \textstyle\sum_{j=2}^{r}(-\beta_j+\alpha_j) + \epsilon_1 \leq \tau(J_1,n+k) - \tau(J_1,n) \leq \epsilon_2 + \sum_{j=2}^{r}(\beta_j-\alpha_j)
\end{gathered}$$

et finalement :

$$\forall\ n,k\ :\ \textstyle\sum_{j=1}^{r}(-\beta_j+\alpha_j) + \epsilon_1 \leq \tau(E,n+k) - \tau(E,n) \leq \epsilon_2 + \sum_{j=1}^{r}(\beta_j-\alpha_j)$$

A.2 Application du modèle : spécification d'éléments réseaux

A.2.1 Réseaux avec service de communication constant

-Montrons que :

l'équation

$$\forall\ n\ :\ \tau(E,n) = n * p' - \lambda$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n\ :\ \tau(S,n) = n * p'$$

-Eléments de preuve :

A partir de l'équation :

$$\forall\ n\ :\ \tau(S,n) = n * p'$$

par substitution avec la définition (3), on obtient finalement :

$$\forall\ n\ :\ \tau(E,n) + \lambda = n * p'$$

D'où la QoS requise :

$$\forall\ n\ :\ \tau(E,n) = n * p' - \lambda$$

A.2.2 Réseaux avec service de communication synchrone

-Montrons que :

l'équation

$$\forall\ n\ :\ n * p' - \lambda \leq \tau(E,n) \leq n * p' - \alpha$$

constitue la QoS requise pour la QoS offerte

$$\forall\ n\ :\ \tau(S,n) = n * p'$$

-Eléments de preuve :

On veut que

$$\forall\ n\ :\ \tau(S,n) = n * p'$$

or

$$\forall\ n\ :\ \alpha \leq \tau(S,n) - \tau(E,n) \leq \lambda$$

d'où

$$\begin{gathered}\forall\ n\ :\ \alpha \leq n * p' - \tau(E,n) \leq \lambda \\ \forall\ n\ :\ -\lambda \leq \tau(E,n) - n * p' \leq -\alpha \\ \forall\ n\ :\ n * p' - \lambda \leq \tau(E,n) \leq n * p' - \alpha\end{gathered}$$

A.3 Algorithmes d'ordonnancement et modèles de tâches

A.3.1 Condition d'ordonnançabilité pour des tâches avec une contrainte de distance

-Montrons que :

$$\forall\ j\ \in\ \gamma\ :\ \sum_{i=1}^{n} C_i \leq Q_j$$

est une condition d'ordonnançabilité suffisante et non nécessaire pour un ensemble (noté γ) de n tâches POLKA indépendantes où chaque tâche j doit respecter les contraintes temporelles suivantes :

$$\forall\ k\ :\ \begin{cases} \tau(A_j, k+1) - \tau(F_j, k) = 0 \\ \tau(F_j, k+1) - \tau(F_j, k) \leq Q_j \end{cases} \tag{A.1}$$

où $\tau(A_j, k)$ est la date d'arrivée dans le système de la k^{ème} activation de la tâche j est $\tau(F_j, k)$ sa date de terminaison. Nous notons C_j le temps d'exécution de la tâche j.

-Eléments de preuve :

Nous étudions successivement le cas d'un système constitué de deux tâches, puis, nous généralisons la preuve à n tâches.

A.3.1.1 Système avec deux tâches

Le système est composé des tâches T_1 et T_2. Nous admettons les hypothèses suivantes :

Hypothèses :

2.a Nous considérons que l'ordonnancement calculé par POLKA est cyclique (fait constaté expérimentalement).

2.b Les tâches sont synchrones [RIV 98]. Nous avons $\forall\ i\ \in\ \gamma\ :\ \tau(A_i, 1) = 0$.

2.c Les tâches T_1 et T_2 sont ordonnançables séparément, en d'autres termes :

$$\forall\ i\ \in\ \gamma\ :\ Q_i \geq C_i \tag{A.2}$$

2.d Enfin, sans perte de généralité, nous supposons que $Q_1 \geq Q_2$.

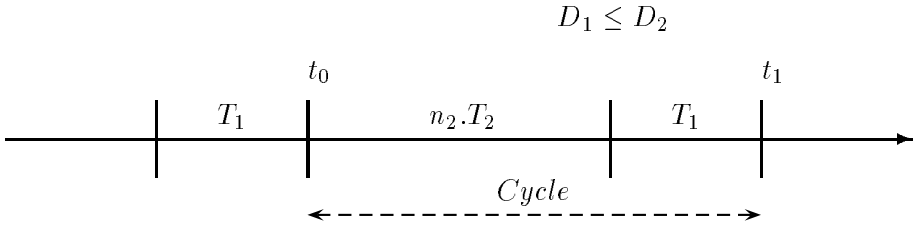


FIG. A.1 – *Système avec deux tâches*

La figure A.1 montre un cycle d'ordonnancement. Nous notons n_i le nombre d'activations de la tâche i dans un cycle et D_i l'échéance de la tâche i. Dans cette figure, un cycle est constitué de n_2 activations de la tâche T_2 ainsi que d'une activation de la tâche T_1. On considère que le début d'un cycle commence à la fin d'une activation de la tâche T_1 (tâche de plus grand D_i), soit en t_0 dans la figure A.1. Pour évaluer la valeur de n_2, on cherche à déterminer quand l'ordonnanceur choisit d'élire la tâche T_1. La condition $D_1 \leq D_2$ constitue le moment où T_1 est de nouveau élue (cf. chapitre 6). Il faut donc chercher quand :

$$D_2 \geq D_1$$

ou encore

$$t_0 \ + \ n_2.C_2 \ + \ Q_2 \geq t_0 \ + \ Q_1$$

et finalement

$$n_2 \geq \frac{Q_1 \ - \ Q_2}{C_2}$$

Le nombre d'activations de T_2 dans un cycle est donc égal à :

$$n_2 \ = \ \begin{cases} 1 & si \ Q_1 \ = \ Q_2 \\ \left\lceil \frac{Q_1 \ - \ Q_2}{C_2} \right\rceil & si \ Q_1 \ > \ Q_2 \end{cases} \tag{A.3}$$

On sait que la durée du cycle (que nous noterons T_c) est de :

$$T_c = n_2 \ . \ C_2 \ + \ C_1 \tag{A.4}$$

Nous allons maintenant étudier le respect des contraintes Q_1 et Q_2. Regardons tout d'abord le cas de la tâche T_1. Il faut que la valeur de T_c permette à cette tâche de respecter sa contrainte de QoS, soit : $T_c \leq Q_1$. Le cas de la tâche T_2 est différent selon ses activations. Dans un cycle donné, les $n_2 - 1$ activations adjacentes de T_2 respectent leur contrainte de QoS, puisque par hypothèse $Q_2 \geq C_2$. Le problème se pose différemment pour la première activation de T_2. En effet, celle-ci est séparée de l'activation précédente de T_2 par une activation de T_1. Autrement dit, cette activation contraint Q_2 de la façon suivante : $Q_2 \geq C_1 \ + \ C_2$. Les différentes tâches doivent donc respecter :

$$\begin{cases} Q_1 \geq T_c \\ Q_2 \geq C_1 \ + \ C_2 \end{cases} \tag{A.5}$$

Regardons sous quelles conditions (A.5) est vrai. Nous avons déjà discuté de la contrainte pour T_2 ($Q_2 \geq C_1 + C_2$). La contrainte pour T_1 ($Q_1 \geq T_c$) peut s'écrire $Q_1 \geq n_2 \ . \ C_2 + C_1$. Ce qui nous amène à considérer les deux cas traités par la formule (A.3) :

1. Le cas où $Q_1 = Q_2$. Ici $n_2 = 1$ et la contrainte devient $Q_1 \geq C_2 \ + \ C_1$.

2. Le cas où $Q_1 \neq Q_2$. Soit δ, un entier. Nous supposons que $(Q_1 - Q_2 + \delta)$ et C_2 sont multiples (δ permet de rendre $(Q_1 - Q_2)$ et C_2 multiples). On a alors $0 \leq \delta < C_2$. Cet artifice nous permet de supprimer l'arrondi de l'équation (A.3). La contrainte devient alors :

$$\begin{array}{lcl} Q_1 & \geq & \frac{Q_1 - Q_2 + \delta}{C_2}.\, C_2 \; + \; C_1 \\ Q_1 & \geq & Q_1 \; - \; Q_2 \; + \; \delta \; + \; C_1 \\ Q_2 & \geq & \delta \; + \; C_1 \end{array}$$

Ainsi, T_1 est garantie du respect de sa QoS si :

$$\left\{ \begin{array}{lcl} Q_1 & \geq & C_2 \; + \; C_1 \\ Q_2 & \geq & \delta \; + \; C_1 \; avec \; 0 \leq \delta < C_2 \end{array} \right.$$

Le système de contrainte (A.5) devient alors :

$$\left\{ \begin{array}{l} Q_1 \geq C_1 \; + \; C_2 \\ Q_2 \geq C_1 \; + \; C_2 \end{array} \right.$$

Finalement, une condition d'ordonnançabilité suffisante pour un système constitué de deux tâches respectant les hypothèses 2.a à 2.d est :

$$\forall \, j \; \in \; \gamma \; : \; Q_j \geq \sum_{i=1}^{n} C_i \qquad \text{(A.6)}$$

Nous montrons maintenant que l'inéquation (A.6) est aussi nécessaire. Supposons que (A.6) ne soit pas nécessaire. Dans ce cas, il existe un ensemble de tâches ordonnançables avec $Q_1 < C_1 + C_2$ ou $Q_2 < C_1 + C_2$. On a montré plus haut que $Q_2 \geq C_1 + C_2$ était nécessaire pour que T_2 respecte sa QoS. Regardons $Q_1 < C_1 + C_2$. Le cas le moins contraignant pour T_1 est celui où $n_2 = 1$. Or même dans ce cas, $Q_1 < C_1 + C_2$ ne permet pas à T_1 de respecter sa QoS. L'inéquation (A.6) est donc bien nécessaire car il n'existe pas de jeu de tâches ordonnançables qui n'ait pas besoin de la respecter. En conclusion, l'inéquation (A.6) est nécessaire et suffisante pour un jeu de deux tâches.

Nous terminons cette partie par la définition d'une notion que nous utiliserons par la suite : la notion de marge (cf. figure A.2). Celle-ci nous permettra de distinguer les activations qui sont nécessaires dans un cycle, pour que les tâches correspondantes respectent leur QoS, de celles qui sont ajoutées par l'ordonnanceur du fait que celui-ci soit non oisif. La marge d'une tâche j est notée M_j et elle se calcule par :

$$\forall j \; \in \; \gamma \; : \; M_j \; = \; Q_j \; - C_j \; - \; \sum_{i \neq j} n_i . C_i \qquad \text{(A.7)}$$

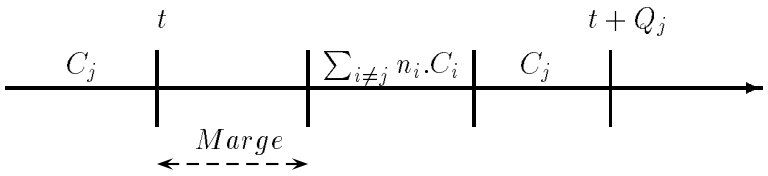


FIG. A.2 – *Marge d'une tâche j*

Tâches	C_i	Q_i
1	3	15
2	1	10

TAB. A.1 – *Exemple d'un système avec deux tâches*

A partir de l'équation (A.7), M_j s'interprète comme le temps entre deux activations de la tâche j, diminué de C_j et de toutes les activations des autres tâches qui doivent être exécutées durant cet intervalle de temps afin que leur QoS soit respectée. Dans la marge, on ne prend donc pas en compte "l'effet de bourrage" produit par l'ordonnanceur. En effet, avec des tâches définies par le jeu d'équations de QoS (A.1), lorsque l'activation d'une tâche se termine, l'activation suivante est automatiquement éligible. En d'autres termes, l'ordonnancement de ce type de contraintes ne peut engendrer de séquence d'ordonnancement où le processeur est inactif (cf. chapitre 6). Si l'on prend l'exemple du jeu de tâches donné dans le tableau A.1, pour une activation de T_1, l'ordonnanceur exécute 5 activations de T_2, alors qu'une seule est nécessaire pour que T_2 respecte sa QoS. Pour ce jeu de tâche, la marge est donc définie par :

$$\begin{cases} M_1 = Q_1 - C_1 - C_2 \\ M_2 = min(Q_2 - C_2,\ Q_2 - C_1 - C_2) \end{cases}$$

et finalement :

$$\forall\ j \in \gamma\ :\ M_j = Q_j - \sum_{i=1}^{n} C_i \tag{A.8}$$

A.3.1.2 Généralisation à n tâches

Nous nous plaçons dans le cas d'un jeu de n tâches et nous regardons les conditions qui permettent d'ajouter la tâche T_{n+1}. Nous supposons que les tâches respectent les hy-

pothèses suivantes :

Hypothèses :

3.a Les Q_i sont contraints par : $\forall\ i,j\ :\ (i<j) \rightarrow (Q_i \geq Q_j)$.

3.b L'inéquation (A.2) est vraie pour toutes les tâches du système.

3.c Le jeu de tâches est ordonnançable par l'inéquation (A.6). Soit $\forall j \in \{1,...,n\}\ :\ C_j \geq \sum_{i=1}^{n} C_i$.

3.d $Q_{n+1} \geq \sum_{i=1}^{n+1} C_i$.

3.e $\forall i \in \{1,...,n+1\}\ :\ \tau(A_i,1)=0$.

3.f On suppose que les marges des différentes tâches sont évaluées par la formule (A.8).

Nous explorons plusieurs possibilités suivant la valeur de Q_{n+1}.

Regardons si, avec ces hypothèses, la contrainte Q_{n+1} peut toujours être garantie. Le cas le plus défavorable est $\forall i,j \in \{1,...,n+1\}\ :\ Q_i = Q_j$. En effet, dans ce cas de figure, T_{n+1} est aussi urgente que les tâches 1 à n. Ainsi, ses activations interviendront aussi souvent que celles des autres tâches. Ici, l'hypothèse 3.d permet à Q_{n+1} de supporter une exécution de chaque tâche T_i (avec $i \in \{1,...,n\}$) entre deux de ses activations. Les tâches ayant une urgence égale, l'ordonnanceur calculera des cycles comprenant une activation de chaque tâche, et donc la contrainte Q_{n+1} sera respectée. En effet, si l'ordonnanceur avait calculé un ordonnancement tel que $T_{n+1}, T_k, T_k, T_{n+1}$ où $k \in \{1,...,n\}$, ceci aurait signifié que T_k était plus urgente que T_{n+1} ; ce qui n'est pas possible.

Considérons maintenant le cas où $\forall k \in \{1,...,n\}\ :\ Q_{n+1} < Q_k$ et $\forall i,j \in \{1,...,n\}\ :\ (i<j) \rightarrow (Q_i \geq Q_j)$. Ici, T_{n+1} est plus urgente que toutes les tâches du jeu de tâches. En d'autres termes, dans un cycle donné, l'ordonnanceur exécutera plusieurs activations de T_{n+1}. Ces différentes activations de T_{n+1} ne peuvent pas être séparées les unes des autres par plus d'une activation de T_k. En effet, une séquence d'ordonnancement $T_{n+1}, T_k, T_k, T_{n+1}$ signifierait que T_k est plus urgente que T_{n+1}. Or ce n'est pas le cas. L'hypothèse 3.d suffit donc pour garantir à T_{n+1} sa contrainte de QoS.

Regardons maintenant si l'adjonction des activations de T_{n+1}, peut entraîner une violation de la QoS d'une tâche du système (tâche 1 à n). Comme pour le paragraphe précédent, nous étudions deux cas extrêmes :

1. $\forall i,j \in \{1,...,n+1\}\ :\ Q_i = Q_j$. Ici, toutes les tâches ont la même urgence. Les tâches 1 à n doivent alors avoir une marge suffisante pour supporter une activation de T_{n+1}.

   En d'autres termes, il faut que $\forall j\ \in \{1,...,n\}$:

$$\begin{array}{rcl} M_j & \geq & C_{n+1} \\ Q_j - \sum_{j=1}^{n} C_j & \geq & C_{n+1} \\ Q_j & \geq & \sum_{i=1}^{n+1} C_i \end{array}$$

2. $\forall k \in \{1,...,n\}$: $Q_{n+1} < Q_k$ et $\forall i,j \in \{1,...,n\}$: $(i < j) \rightarrow (Q_i \geq Q_j)$. Dans ce deuxième cas, T_{n+1} est la tâche la plus urgente. Dans un même cycle, il y aura donc plusieurs activations de T_{n+1}. Par les hypothèses 3.a et 3.f, la tâche T_n est celle dont la marge est la plus faible du jeu de tâches. On peut déterminer le nombre maximal d'activations de T_{n+1}, que T_n peut supporter sans violer sa QoS. T_n peut supporter $k = \left\lfloor \frac{M_n}{C_{n+1}} \right\rfloor$ activations de T_{n+1}. Considérons maintenant un jeu de tâches constitué des tâches 1 à $n+1$. On sait que le nombre j d'activations de T_{n+1} pour une activation de T_n est alors de (cf. équation (A.3)) :

$$j = \frac{Q_n \ - \ Q_{n+1}}{C_{n+1}}$$

Si $j + 1 \leq k$[1], on est assuré que T_n supportera le nombre d'activation maximal de T_{n+1}. En effet, j représente le nombre maximal d'activation de T_{n+1} pour une activation de T_n, par rapport à leur urgence respective. Le nombre effectif est moindre, du fait de l'existence des contraintes ajoutées par les tâches 1 à $n-1$. Ainsi on a :

$$\begin{array}{rcl} j \ + \ 1 & \leq & k \\ \frac{Q_n \ - \ Q_{n+1}}{C_{n+1}} \ + \ 1 & \leq & \frac{M_n}{C_{n+1}} \\ Q_n \ - \ Q_{n+1} \ + \ C_{n+1} & \leq & M_n \\ Q_n \ - \ Q_{n+1} & \leq & Q_n \ - \ \sum_{i=1}^{n} C_i \ - \ C_{n+1} \\ Q_{n+1} & \geq & \sum_{i=1}^{n} C_i \ + \ C_{n+1} \end{array}$$

ce qui est vrai par l'hypothèse 3.d. On a donc démontré que la marge de T_n supportait le nombre maximal d'activations de T_{n+1}. Il reste à démontrer que ceci est vrai pour les tâches $\{T_1, ..., T_{n-1}\}$, ce qui est trivial puisque nous avons supposé que les marges étaient évaluées par la formule (A.8), ce qui implique, grâce à l'hypothèse 3.a, que :

$$\forall j \ : \ M_j \geq M_{j-1}$$

L'inéquation (A.6) reste donc suffisante pour un jeu de tâches défini par les tâches $T_1, ..., T_{n+1}$. Toutefois, nous avions posé comme hypothèse que les marges des tâches du système étaient évaluées par la formule (A.8). Nous montrons ici qu'un jeu de tâches de ce type implique que les marges soient effectivement évaluées de cette façon.

Soit le jeu de tâches $\{T_1, .., T_n\}$. Supposons que :

Hypothèses :

4.a Le jeu de tâches satisfait l'inéquation (A.2) ; soit $\forall j \in \{1, ..., n\}$: $Q_j \geq C_j$.

4.b Le jeu de tâches est ordonnançable et l'inéquation (A.6) est vraie ; soit $\forall j \in \{1, ..., n\}$: $Q_j \geq \sum_{i=1}^{i=n} C_i$.

[1] On ajoute 1 à j car l'équation (A.3) est arrondie à l'entier supérieur.

Rappelons que nous avions défini la marge d'une tâche j par l'équation (A.7). La marge permet de supprimer l'effet de bourrage produit par l'ordonnanceur. En effet, l'ordonnanceur exécute dans un cycle autant d'activations d'une tâche qu'il est possible, même si une seule est suffisante pour respecter sa QoS. Il est donc utile de connaître le nombre d'activations nécessaires et suffisantes de chaque tâche pour savoir si l'on peut encore en accepter de nouvelles. Nous cherchons ici, à déterminer le nombre minimal d'activations de chaque tâche dans un cycle qui permet le respect de toutes les contraintes de QoS. Considérons une séquence d'ordonnancement qui commence par une activation de T_n, la tâche la plus urgente du système, et regardons ce qui se passe jusqu'à l'activation suivante de T_n. Nous avons supposé que le jeu de tâches en question était ordonnançable et qu'il respectait l'inéquation (A.6). Il vient qu'entre deux activations de T_n, il est possible d'exécuter au moins une activation de chaque tâche T_k (avec $1 \leq k \leq n-1$) tout en respectant Q_n. Par les hypothèses 3.a et 3.b (c'est-à-dire $\forall j : (Q_j \geq Q_{j-1}) \wedge (Q_j \geq \sum_{i=1}^{i=n} C_i)$, un tel ordonnancement est déjà suffisant pour garantir à toutes les tâches T_k (avec $1 \leq k \leq n-1$) leur contrainte de QoS. Un tel raisonnement peut être conduit pour toutes les tâches T_k (avec $1 \leq k \leq n-1$) et on en conclut que pour un tel jeu de tâches, une seule activation de chaque tâche par cycle est suffisante pour respecter leurs différentes contraintes de QoS. Ce nombre d'activations est bien entendu nécessaire, puisque s'il était inférieur à 1, la tâche ne serait jamais exécutée. En conclusion, les marges des tâches $\{T_1, ..., T_n\}$ peuvent être définies par l'équation (A.8).

A.4 Vers le support de contraintes probabilistes

A.4.1 Evaluation de la gigue induite par un lien CBR

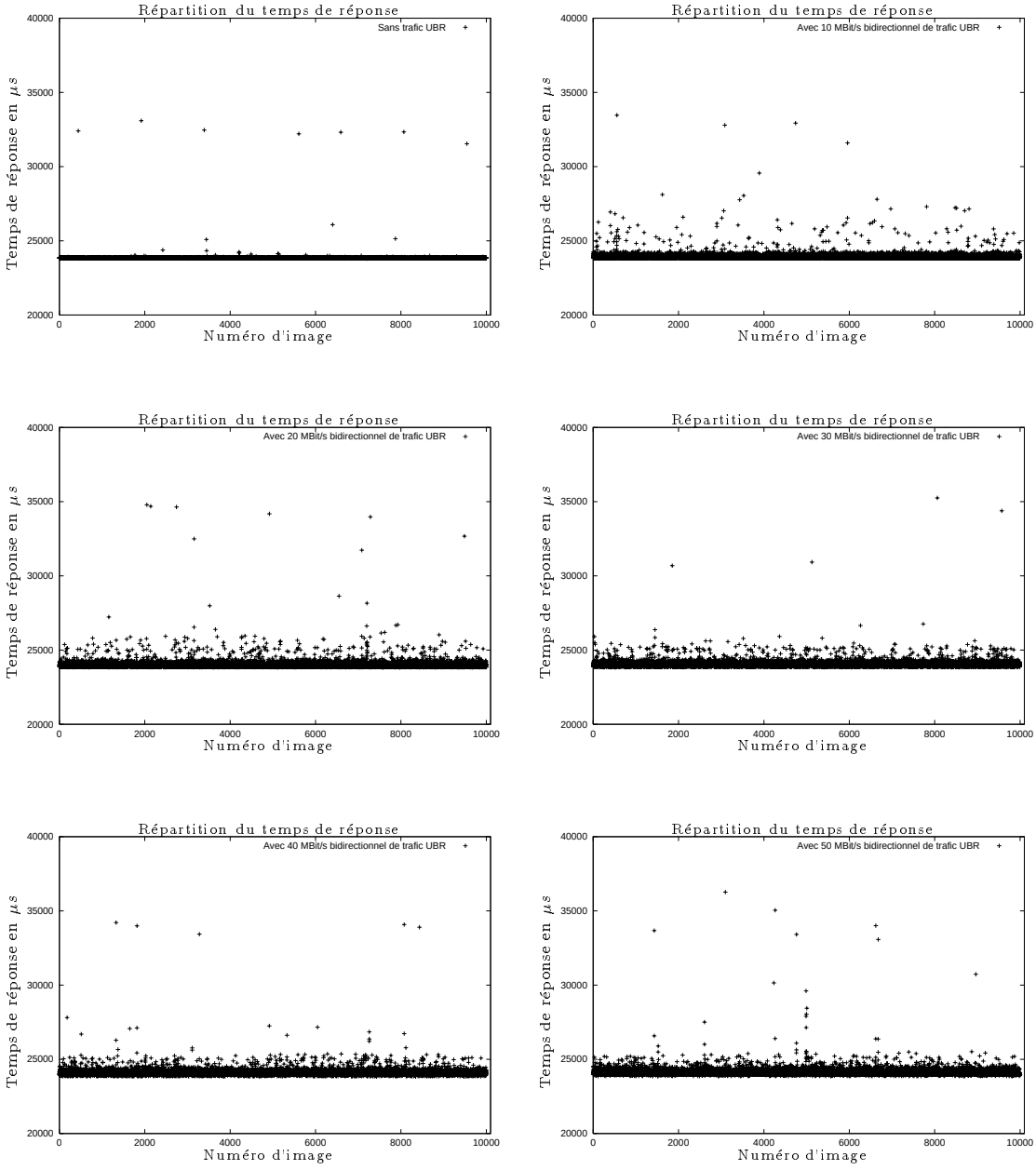
Les estimations sur la valeur maximale, minimale et la répartition de la gigue utilisée dans la partie 9.2 sont obtenues de la façon suivante. Nous exécutons l'application décrite dans le chapitre 9 sur un canal CBR bidirectionnel pour lequel nous avons réservé un PCR de 20000 cellules par seconde et un ppCDV de 100 ms. Le débit généré par l'application pour chaque invocation de méthode est de 20063 octets pour la requête et de 24 octets pour la réponse (une invocation de la méthode $putXpm$ véhicule 20000 octets d'information utilisateur). Enfin, une invocation est effectuée toutes les 40 ms. Afin d'estimer la gigue induite par le réseau, nous mesurons le temps d'aller-retour pour chaque invocation et ce sous divers niveaux de charge. Pour ce faire, nous générons un débit fictif sur deux connexions UBR (une du client vers le serveur et l'autre dans le sens inverse). Le trafic sur chaque connexion UBR est constitué de l'envoi périodique d'un datagramme UDP de 4096 octets. Afin d'éviter des phénomènes de synchronisation, la période varie de façon aléatoire. La période d'émission est calculée de la façon suivante :

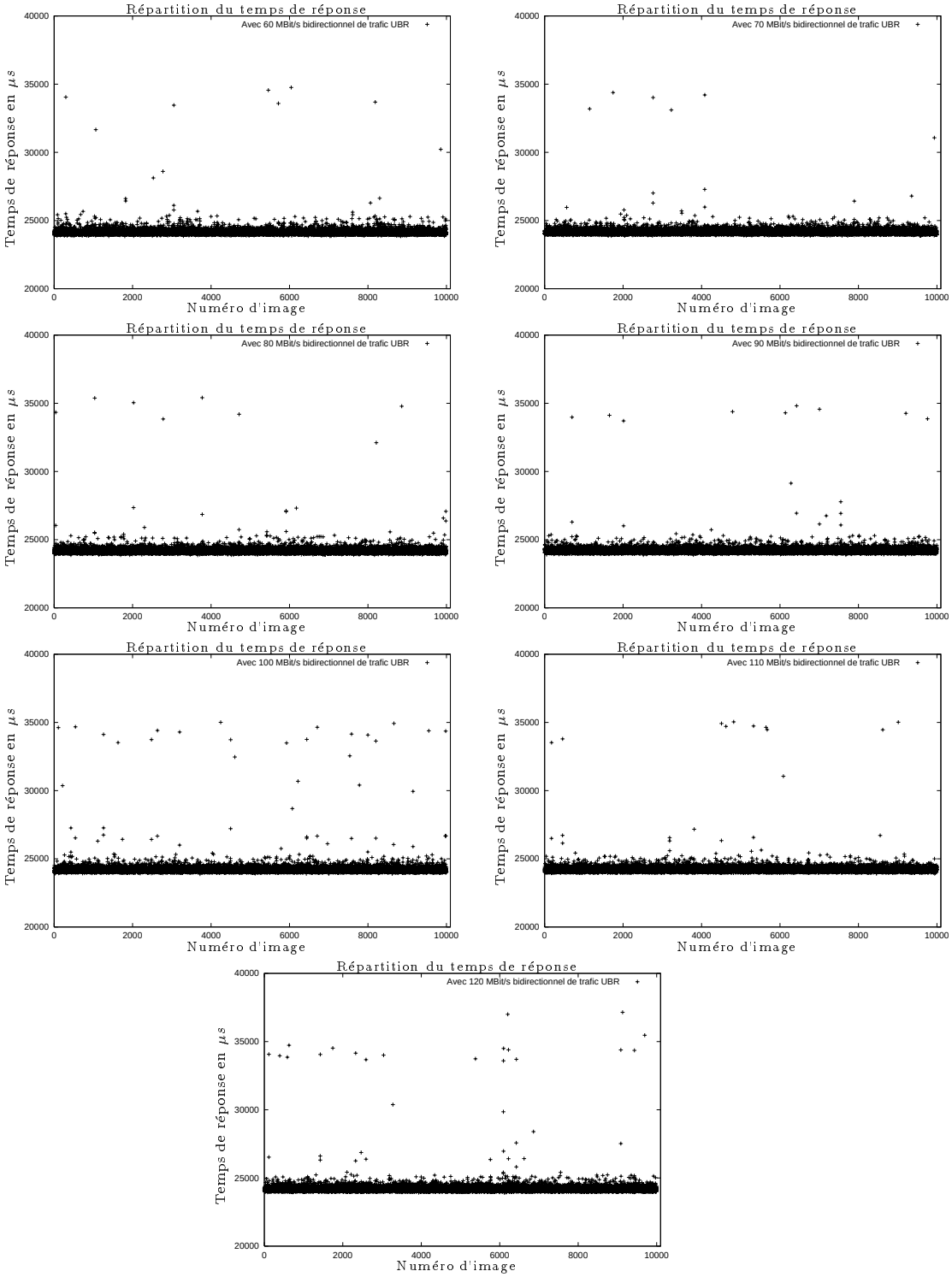
$$période = \frac{débit}{4096} * (drand48() + 0,5)$$

où *débit* est le débit à émettre en octets et $drand48()$ une fonction qui renvoie une valeur aléatoire uniformément répartie entre 0 à 1.

Les connexions UDP sont ouvertes sur un réseau Classical-IP utilisant le service UBR.

Les courbes ci-dessous montrent les temps d'aller-retour obtenus par l'application avec des niveaux de débits différents pour les connexions UBR. Les temps maximaux, minimaux et moyens de l'ensemble de ces mesures sont répertoriés dans le tableau 9.1. Leur répartition est donnée dans le graphe 9.3.




Répartition du temps de réponse
Avec 60 MBit/s bidirectionnel de trafic UBR
Avec 70 MBit/s bidirectionnel de trafic UBR
Avec 80 MBit/s bidirectionnel de trafic UBR
Avec 90 MBit/s bidirectionnel de trafic UBR
Avec 100 MBit/s bidirectionnel de trafic UBR
Avec 110 MBit/s bidirectionnel de trafic UBR
Avec 120 MBit/s bidirectionnel de trafic UBR
Temps de réponse en μs
Numéro d'image
20000
25000
30000
35000
40000
0
2000
4000
6000
8000
10000

Annexe B

Spécifications Esterel de composants

B.1 Module de retard fixe

```
module retardFixe :
    %
    % Parametres
    %
    constant l : integer;
    %
    % Types
    %
    type ELEMENT;
    %
    % Signaux
    %
    input H;
    input E(ELEMENT);
    output S(ELEMENT);
    %
    % Corps du module
    %
    every E do
        await l H;
        emit S(?E);
    end every
end module
```

B.2 Module de retard variable

```
module retardVariable :
    %
    % Fonctions et procedures
    %
```

```
    function lireRetard() : integer;
    %
    % Types
    %
    type ELEMENT;
    %
    % Signaux
    %
    input H;
    input E(ELEMENT);
    output S(ELEMENT);
    %
    % Corps du module
    %
    loop
        await H;
        present E then
            var attente : integer in
                attente := lireRetard();
                await attente H;
                emit S(?E);
            end var
        end present
    end loop
end module
```

B.3 Modélisation d'un tampon en Esterel

```
module tampon :
    %
    % Constantes
    %
    constant TAILLE_TAMPON : integer;
    constant TAILLE_ELEMENT_ENTREE : integer;
    constant TAILLE_ELEMENT_SORTIE : integer;
    %
    % Types
    %
    type ELEMENT_ENTREE;
    type ELEMENT_SORTIE;
    %
    % Signaux
    %
    output TAMPON_FAMINE;
    output TAMPON_DEBORDEMENT;
    input REQUETE_ECRITURE(ELEMENT_ENTREE);
    input REQUETE_LECTURE;
    output REPONSE_LECTURE(ELEMENT_SORTIE);
    %
    % Fonctions et procedures
```

```
    %
    function lireElement() : ELEMENT_SORTIE;
    procedure ecrireElement()(ELEMENT_ENTREE);
    function nombreElements() : integer;
    %
    % Corps du module
    %
    loop
        await
            case REQUETE_LECTURE do
              var donnee : ELEMENT_SORTIE in
                  if (nombreElements() > 0)
                    then donnee := lireElement();
                       emit REPONSE_LECTURE(donnee);
                    else emit TAMPON_FAMINE;
                  end if
              end var
            case REQUETE_ECRITURE do
              if ((nombreElements() + 1) * TAILLE_ELEMENT_ENTREE < TAILLE_TAMPON)
                then call ecrireElement()(?REQUETE_ECRITURE);
                else emit TAMPON_DEBORDEMENT;
              end if
        end await
    end loop
end module
```

B.4 Module compensateur de gigue

```
module compensateur :
    %
    % Parametres
    %
    constant TAILLE_TAMPON : integer;
    constant CADENCE : integer;
    %
    % Types
    %
    type OCTET;
    type ELEMENT_ENTREE;
    %
    % Signaux
    %
    input H;
    input E(ELEMENT_ENTREE);
    output S(OCTET);
    output TAMPON_DEBORDEMENT;
    output TAMPON_FAMINE;
    %
    % Corps du module
    %
```

```
    signal REQUETE_ECRITURE : ELEMENT_ENTREE, REQUETE_LECTURE,
        REPONSE_LECTURE : OCTET in
        %
        % Le tampon du composant
        %
        run tampon [ type OCTET/ELEMENT_SORTIE ]
        ||
        %
        % Le comportement du composant
        %
        signal PREMIER_ELEMENT in
            var send : OCTET, gigue := (TAILLE_TAMPON * CADENCE)/2 : integer in
                %
                % Attente pour eviter la famine sur le tampon
                %
                await PREMIER_ELEMENT;
                await gigue H;
                %
                % Livraison des donnees
                %
                every CADENCE H do
                    emit REQUETE_LECTURE;
                    await REPONSE_LECTURE;
                    emit S(?REPONSE_LECTURE);
                end every
            end var
            ||
            var premier := true : boolean in
                every E do
                    if (premier)
                      then emit PREMIER_ELEMENT;
                        premier := false;
                    end if;
                    emit REQUETE_ECRITURE(?E);
                end every
            end var
        end signal
    end signal
end module
```

B.5 Module du composant réseau élémentaire

```
module reseau :
    %
    % Fonctions et procedures
    %
    function lireTempsDeCommunication() : integer;
    %
    % Types
    %
```

```
    type ELEMENT;
    %
    % Signaux
    %
    input H;
    input E(ELEMENT);
    output S(ELEMENT);
    %
    % Corps du module
    %
    loop
        await H;
        present E then
            var temps : integer in
                temps := lireTempsDeCommunication();
                await temps H;
                emit S(?E);
            end var
        end present
    end loop
end module
```

B.6 Module du protocole à jeton temporisé

```
module BoucleTemporise :
    %
    % Constantes
    %
    constant TAILLE_ELEMENT : integer;
    constant TAILLE_TAMPON_SORTIE : integer;
    constant TAILLE_TAMPON_ENTREE : integer;
    constant BANDE_PASSANTE_SYNCHRONE : integer;
    constant TTRT : integer;
    constant DEBIT : integer;
    %
    % Types
    %
    type OCTET;
    type ELEMENT;
    %
    % Signaux
    %
    output SORTIE_MESSAGE(ELEMENT);
    output EMIS_OCTETS_SYNC(OCTET);
    output DEPART_JETON;
    input ENTREE_MESSAGE(ELEMENT);
    input RECEP_OCTETS_SYNC(OCTET);
    input ARRIVEE_JETON;
    input HORLOGE;
    %
```

```
% Fonctions et procedures
%
function finTrame(OCTET) : boolean;
function debutTrame(OCTET) : boolean;
function lireTamponEntree() : OCTET;
procedure ecrireTamponEntree()(ELEMENT);
function nombreTamponEntree() : integer;
function lireTamponSortie() : ELEMENT;
procedure ecrireTamponSortie()(OCTET);
function nombreTamponSortie() : integer;
%
% Corps du module
%
signal ENTREE_REQUETE_ECRITURE : ELEMENT,
    ENTREE_REQUETE_LECTURE,
    SORTIE_REQUETE_ECRITURE : OCTET,
    SORTIE_REQUETE_LECTURE,
    ENTREE_REPONSE_LECTURE : OCTET,
    SORTIE_REPONSE_LECTURE : ELEMENT,
    TAMPON_ENTREE_FAMINE,
    TAMPON_SORTIE_FAMINE,
    TAMPON_ENTREE_DEBORDEMENT,
    TAMPON_SORTIE_DEBORDEMENT in
    %
    % Le tampon ENTREE du composant
    %
    run tampon [ type OCTET/ELEMENT_SORTIE;
        type ELEMENT/ELEMENT_ENTREE;
        constant 1/TAILLE_ELEMENT_SORTIE;
        constant TAILLE_ELEMENT/TAILLE_ELEMENT_ENTREE;
        constant TAILLE_TAMPON_ENTREE/TAILLE_TAMPON;
        function lireTamponEntree/lireElement;
        procedure ecrireTamponEntree/ecrireElement;
        function nombreTamponEntree/nombreElements;
        signal TAMPON_ENTREE_DEBORDEMENT/TAMPON_DEBORDEMENT;
        signal TAMPON_ENTREE_FAMINE/TAMPON_FAMINE;
        signal ENTREE_REQUETE_ECRITURE/REQUETE_ECRITURE;
        signal ENTREE_REQUETE_LECTURE/REQUETE_LECTURE;
        signal ENTREE_REPONSE_LECTURE/REPONSE_LECTURE
    ]
    ||
    %
    % Le tampon de SORTIE du composant
    %
    run tampon [ type OCTET/ELEMENT_ENTREE;
        type ELEMENT/ELEMENT_SORTIE;
        constant 1/TAILLE_ELEMENT_ENTREE;
        constant TAILLE_ELEMENT/TAILLE_ELEMENT_SORTIE;
        constant TAILLE_TAMPON_SORTIE/TAILLE_TAMPON;
        function lireTamponSortie/lireElement;
        procedure ecrireTamponSortie/ecrireElement;
        function nombreTamponSortie/nombreElements;
        signal TAMPON_SORTIE_DEBORDEMENT/TAMPON_DEBORDEMENT;
```

```
      signal TAMPON_SORTIE_FAMINE/TAMPON_FAMINE;
      signal SORTIE_REQUETE_ECRITURE/REQUETE_ECRITURE;
      signal SORTIE_REQUETE_LECTURE/REQUETE_LECTURE;
      signal SORTIE_REPONSE_LECTURE/REPONSE_LECTURE
]
||
%
% L'ecriture d'un message synchrone dans le tampon ENTREE
%
every ENTREE_MESSAGE do
     emit ENTREE_REQUETE_ECRITURE(?ENTREE_MESSAGE);
end every;
||
%
% Livraison d'une trame FDDI
% a l'application
%
signal RECEPTION_TRAME in
     every RECEPTION_TRAME do
          emit SORTIE_REQUETE_LECTURE;
          await SORTIE_REPONSE_LECTURE;
          emit SORTIE_MESSAGE(?SORTIE_REPONSE_LECTURE);
     end every
     ||
     loop
          %
          % Reception des donnees circulant sur
          % la boucle
          %
          do
               var receiving := false : boolean in
                    every RECEP_OCTETS_SYNC do
                         %
                         % On verifie que c'est un debut ou une fin
                         % de trame
                         %
                         if (finTrame(?RECEP_OCTETS_SYNC))
                           then [
                                  emit SORTIE_REQUETE_ECRITURE(?RECEP_OCTETS_SYNC);
                                  emit RECEPTION_TRAME;
                                  receiving := false;
                             ]
                           else if (debutTrame(?RECEP_OCTETS_SYNC))
                                  then [
                                         receiving := true;
                                         emit SORTIE_REQUETE_ECRITURE
                                              (?RECEP_OCTETS_SYNC);
                                    ]
                                  else if (receiving)
                                         then emit SORTIE_REQUETE_ECRITURE
                                                   (?RECEP_OCTETS_SYNC);
                                    end if
                             end if
```

```
                        end if
                    end every
                end var
            watching ARRIVEE_JETON;
            %
            % On emet son trafic synchrone
            %
            do
                loop
                    emit ENTREE_REQUETE_LECTURE;
                    await ENTREE_REPONSE_LECTURE;
                    emit EMIS_OCTETS_SYNC(?ENTREE_REPONSE_LECTURE);
                each HORLOGE;
            watching TTRT/BANDE_PASSANTE_SYNCHRONE HORLOGE;
            %
            % On libere le jeton
            %
            emit DEPART_JETON;
        end loop
    end signal
end signal
end module
```

Annexe C

Exemples de spécification en QL d'applications multimédias

C.1 Description de composants standards

```
component retardfixe:
        signals in I, in O;

        qoscontract avechorloge:
                common
                        clock rfc=(period,);
                provided
                        eqos rf1 T(rfc,n) - T(O,n) < epsilon2;
                        eqos rf2 T(rfc,n) - T(O,n) > epsilon1;
                required
                        eqos rf3 T(rfc,n) - T(I,n) < epsilon2+l;
                        eqos rf4 T(rfc,n) - T(I,n) > epsilon1+l;
        end
        qoscontract sanshorloge:
                common
                provided
                        eqos rf1 T(O,n+k) - T(O,n) < epsilon2;
                        eqos rf2 T(O,n+k) - T(O,n) > epsilon1;
                required
                        eqos rf3 T(I,n+k) - T(I,n) < epsilon2;
                        eqos rf4 T(I,n+k) - T(I,n) > epsilon1;
        end
end

component retardvariable:
        signals in I, in O;

        qoscontract avechorloge:
                common
                        clock rvc=(period,);
                provided
```

```
                        eqos rv1 T(rvc,n) - T(O,n) < epsilon2;
                        eqos rv2 T(rvc,n) - T(O,n) > epsilon1;
                required
                        eqos rv3 T(rvc,n) - T(I,n) < epsilon2+alpha;
                        eqos rv4 T(rvc,n) - T(I,n) > epsilon1+beta;
        end
        qoscontract sanshorloge:
                common
                provided
                        eqos rv1 T(O,n+k) - T(O,n) < epsilon2;
                        eqos rv2 T(O,n+k) - T(O,n) > epsilon1;
                required
                        eqos rv3 T(I,n+k) - T(I,n) < epsilon2+beta-alpha;
                        eqos rv4 T(I,n+k) - T(I,n) > epsilon1-beta+alpha;
        end
end

component compensateur:
        signals in I, in O;

        qoscontract unique:
                common
                        clock cc=(period,);
                provided
                        eqos c1 T(cc,n) - T(O,n) < 0;
                        eqos c2 T(cc,n) - T(O,n) > 0;
                required
                        eqos c3 T(cc,n) - T(I,n) < epsilon2;
                        eqos c4 T(cc,n) - T(I,n) > epsilon1;
        end
end

component gigueur:
        signals in I, in O;

        qoscontract unique:
                common
                        clock go=(po,);
                        clock gi=(pi,);
                provided
                        eqos g1 T(go,n) - T(O,n) < epsilon2;
                        eqos g2 T(go,n) - T(O,n) > epsilon1;
                required
                        eqos g3 T(gi,n) - T(I,n) < epsilon3;
                        eqos g4 T(gi,n) - T(I,n) > epsilon4;
        end
end
```

C.2 Première application du chapitre 8

C.2.1 Spécification de QoS fournie par l'application

```
component gif:
        signals out 0;

        qoscontract intra :
                common
                        clock c=(1000,);
                provided
                        eqos rf1 T(c,n) - T(0,n) <  10 ms;
                        eqos rf2 T(c,n) - T(0,n) > -10 ms;
                required
        end
end


component synchronizedGif : g1/gif, g2/gif
        signals out o1=g1.0, out o2=g2.0;

        qoscontract synchro : g1.intra, g2.intra
                common
                provided
                        eqos l1 T(o2,n) - T(o1,n) < 5 ms;
                        eqos l2 T(o1,n) - T(o2,n) < 5 ms;
                required
        end
end

system xpicture:

        instance gif:
                type localComponent spoutnik;
                component synchronizedGif;
                qoscontract synchro   gc0/g1.c, gc0/g2.c;
                parameters ;
                signals o1=flow0.displayNextPicture.RR,
                        o2=flow1.displayNextPicture.RR;
        end


end
```

C.2.2 Spécification de QoS générée par le gestionnaire de QoS ou par le compilateur *qc*

```
clock gc0=(1000,);

eqos gif.g2.rf2 T(gc0,n) - T(flow1.displayNextPicture.RR,n) > -10 ms ;
eqos gif.g2.rf1 T(gc0,n) - T(flow1.displayNextPicture.RR,n) < +10 ms ;
eqos gif.g1.rf2 T(gc0,n) - T(flow0.displayNextPicture.RR,n) > -10 ms ;
eqos gif.g1.rf1 T(gc0,n) - T(flow0.displayNextPicture.RR,n) < +10 ms ;
```

```
eqos gif.l1 T(flow1.displayNextPicture.RR,n) - T(flow0.displayNextPicture.RR,n) < +5 ms ;
eqos gif.l2 T(flow0.displayNextPicture.RR,n) - T(flow1.displayNextPicture.RR,n) < +5 ms ;
```

C.3 Deuxième application du chapitre 8

C.3.1 Une spécification de QoS simple

C.3.1.1 Spécification de QoS pour tester l'adaptativité

```
component film:
        signals out son, out image;

        qoscontract qos:
                common
                provided
                        eqos ie T(image,n+1) - T(image,n) < 40  ms ;
                        eqos se T(son,n+1) - T(son,n) < infinity ;
                required
        end
end

component lesFilms: f1/film, f2/film, f3/film, f4/film, f5/film

        signals out s1=f1.son, out s2=f2.son, out s3=f3.son,
                out s4=f4.son, out s5=f5.son,
                out i1=f1.image, out i2=f2.image, out i3=f3.image,
                out i4=f4.image, out i5=f5.image;

        qoscontract qos: f1.qos, f2.qos, f3.qos, f4.qos, f5.qos
                common
                provided
                        trigger newMovie1 f2.ie=100 , f1.ie=3000,
                                          f3.ie=3000, f4.ie=3000,
                                          f5.ie=3000;
                        trigger newMovie2 f3.ie=100 , f1.ie=6000,
                                          f2.ie=6000, f4.ie=6000,
                                          f5.ie=6000;
                        trigger newMovie3 f4.ie=100 , f1.ie=12000,
                                          f2.ie=12000, f3.ie=12000,
                                          f5.ie=12000;
                        trigger newMovie4 f5.ie=100 , f1.ie=18000,
                                          f2.ie=18000, f3.ie=18000,
                                          f4.ie=18000;
                required
        end
end

system mpeg:

        instance cesard:
                type localComponent ubu;
                component lesFilms;
```

```
                qoscontract qos;
                parameters;
                signals i1=image1.imageDecoder.IE, i2=image2.imageDecoder.IE,
                        i3=image3.imageDecoder.IE, i4=image4.imageDecoder.IE,
                        i5=image5.imageDecoder.IE,
                        s1=son1.soundDecoder.IE, s2=son2.soundDecoder.IE,
                        s3=son3.soundDecoder.IE, s4=son4.soundDecoder.IE,
                        s5=son5.soundDecoder.IE;

        end

end
```

C.3.1.2 Spécification de QoS pour tester la synchronisation voix-lèvres

```
component film:
        signals out son, out image;

        qoscontract qos:
                common
                provided
                        eqos ie T(image,n+1) - T(image,n) < 40  ms ;
                        eqos se T(son,n+1) - T(son,n) < 40 ms ;
                        eqos inter T(son,n*5) - T(image,n) < 80 ms ;
                required
        end
end

system mpeg:

        instance cesard:
                type localComponent ubu;
                component film;
                qoscontract qos;
                parameters;
                signals image=image1.imageDecoder.IE,
                        son=son1.soundDecoder.IE;

        end

end
```

C.3.2 Spécification de QoS de bout en bout

```
component film:
        signals out son, out image;

        qoscontract qos:
                common
                        clock ha=(26.25,);
                provided
                        eqos ie1 T(image,n+1) - T(image,n) < 60  ms + x s;
                        eqos ie2 T(image,n+1) - T(image,n) > 20  ms - x s;
```

```
                        eqos se1 T(ha,n) - T(son,n) < 32 ms ;
                        eqos se2 T(ha,n) - T(son,n) > - 32 ms ;
                        eqos inter1 T(ha,n*320) - T(image,n) < 40 ms ;
                        eqos inter2 T(ha,n*320) - T(image,n) > - 40 ms ;
                required
        end
end

system mpeg:

        instance cesard:
                type localComponent ubu;
                component film;
                qoscontract qos;
                parameters x=10;
                signals image=image1.imageDecoder.IE,
                        son=son1.soundDecoder.IE;

        end

end
```

C.4 Troisième application du chapitre 8

C.4.1 Spécification de QoS fournie par l'application

```
component fluxfixe: faudio/retardfixe, fvideo/retardfixe
        signals in  a2=faudio.I, in  i2=fvideo.I,
                out a1=faudio.O, out i1=fvideo.O;

        qoscontract qos: faudio.avechorloge, fvideo.sanshorloge
                common
                        clock ha=(period,);
                provided
                        eqos a1 T(ha*k,n) - T(i1,n) < epsilon2 ms;
                        eqos a2 T(ha*k,n) - T(i1,n) > epsilon1 ms;
                required
                        eqos a3 T(ha*k,n) - T(i2,n) < epsilon2+l ms;
                        eqos a4 T(ha*k,n) - T(i2,n) > epsilon1+l ms;
        end
end

component fluxvariable: vaudio/retardvariable, vvideo/retardvariable
        signals in  a2=vaudio.I, in  i2=vvideo.I,
                out a1=vaudio.O, out i1=vvideo.O;

        qoscontract qos: vaudio.avechorloge, vvideo.sanshorloge
                common
                        clock ha=(period,);
                provided
```

```
                        eqos a1 T(ha*k,n) - T(i1,n) < epsilon2 ms;
                        eqos a2 T(ha*k,n) - T(i1,n) > epsilon1 ms;
                required
                        eqos a3 T(ha*k,n) - T(i2,n) < epsilon2+alpha ms;
                        eqos a4 T(ha*k,n) - T(i2,n) > epsilon1+beta  ms;
        end
end



system dis:

        instance application:
                type localComponent twist;
                component fluxfixe;
                qoscontract qos ha/ha, hb/faudio.rfc;
                parameters fvideo.k=1,
                           faudio.l=2.11,
                           l=55.24,
                           faudio.period=26.25,
                           k=800, period=0.125,
                           epsilon1=-40, epsilon2=40,
                           faudio.epsilon1=-32, faudio.epsilon2=32,
                           fvideo.epsilon1=80, fvideo.epsilon2=120;
                signals a1=md.decodeAudioFrame.RR,
                        a2=ms.putAudioFrame.RR,
                        i1=md.decodeVideoFrame.RR,
                        i2=ms.putVideoFrame.RR;
        end

        instance reseau:
                type networkDevice rock twist;
                component fluxvariable;
                qoscontract qos ha/ha, hb/vaudio.rvc;
                parameters vaudio.period=26.25,
                           vvideo.k=1,
                           k=800, period=0.125,
                           vaudio.alpha=monitoring.network.delay.min,
                           vaudio.beta=monitoring.network.delay.max,
                           vvideo.alpha=monitoring.network.delay.min,
                           vvideo.beta=monitoring.network.delay.max;
                signals a1=ms.putAudioFrame.RR,
                        a2=ms.putAudioFrame.IE,
                        i1=ms.putVideoFrame.RR,
                        i2=ms.putVideoFrame.IE;
        end
end
```

C.4.2 Spécification de QoS générée par le gestionnaire de QoS ou par le compilateur *qc*

C.4.2.1 Equations sur la machine où s'effectue la présentation des données

```
clock ha=(0.125,);
clock hb=(26.25,);


eqos application.fvideo.rf2 T(md.decodeVideoFrame.RR,n+1)
                - T(md.decodeVideoFrame.RR,n) > +80 ms ;
eqos application.fvideo.rf1 T(md.decodeVideoFrame.RR,n+1)
                - T(md.decodeVideoFrame.RR,n) < +120 ms ;
eqos application.faudio.rf2 T(hb,n) - T(md.decodeAudioFrame.RR,n) > -32 ms ;
eqos application.faudio.rf1 T(hb,n) - T(md.decodeAudioFrame.RR,n) < +32 ms ;
eqos application.a1 T(ha,n*800) - T(md.decodeVideoFrame.RR,n) < +40 ms ;
eqos application.a2 T(ha,n*800) - T(md.decodeVideoFrame.RR,n) > -40 ms ;
```

C.4.2.2 Equations sur la machine où s'effectue la lecture des données depuis le disque

```
clock ha=(0.125,);
clock hb=(26.25,);


eqos reseau.vvideo.rv4 T(ms.putVideoFrame.IE,n+1)
                 - T(ms.putVideoFrame.IE,n) > -vvideo.beta ms +vvideo.alpha ms +80 ms ;
eqos reseau.vvideo.rv3 T(ms.putVideoFrame.IE,n+1)
                 - T(ms.putVideoFrame.IE,n) < +vvideo.beta ms -vvideo.alpha ms +120 ms ;
eqos reseau.vaudio.rv4 T(hb,n) - T(ms.putAudioFrame.IE,n) > +vaudio.beta ms -29.89 ms ;
eqos reseau.vaudio.rv3 T(hb,n) - T(ms.putAudioFrame.IE,n) < +vaudio.alpha ms +34.11 ms ;
eqos reseau.a3 T(ha,n*800) - T(ms.putVideoFrame.IE,n) < +alpha ms +95.24 ms ;
eqos reseau.a4 T(ha,n*800) - T(ms.putVideoFrame.IE,n) > +beta ms +15.24 ms ;
```

Annexe D

Publications associées à cette thèse

D.1 Article dans revue avec comité de lecture

- *Modélisation et support d'applications multimédias réparties.* Revue Calculateurs parallèles, Réseaux et Systèmes répartis, Septembre 1999, 11(2):161-191. Isabelle Demeure, Laurent Leboucher, Nicolas Rivierre, Frank Singhoff.

D.2 Communications internationales avec comité de lecture

- *Support of Temporal QoS Constraints for Distributed Object-Oriented Multimedia Applications.* Article soumis à ACM Multimedia 2000. Frank Singhoff, Isabelle Demeure, Laurent Leboucher, Nicolas Rivierre.

- *Environnement d'exécution pour les applications réparties sous contraintes temporelles: une solution CORBA-RTP.* $10^{èmes}$ Rencontres Francophones du Parallélisme (RENPAR'10). Strasbourg, 9-12 Juin 1998, pages 53-57. Frank Singhoff, Isabelle Demeure.

- *Automatic Scheduling of a Dynamic Multimedia Applications with Polka: a Case Study.* Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98), Work in Progress session, Denver, Colorado, USA June 3-5, 1998, pages 15-19, Isabelle Demeure, Frank Singhoff, François Horn.

D.3 Communications nationales avec comité de lecture

- *Modèle et plate-forme pour le support d'applications multimédias réparties.* Conférence Française sur les systèmes d'exploitation (CFSE'1), pages 97-108, Rennes, 8-11 Juin 1999. Isabelle Demeure, Laurent Leboucher, Nicolas Rivierre, Frank Singhoff.

- *Spécification et ordonnancement dynamique d'applications multimédias: l'environnement POLKA.* Real time systems (RTS'98), pages 101-115, Paris la Défense Janvier

1998. Frank Singhoff, Isabelle Demeure.

Index