ARCHITECTURE DE SYSTÈMES DANS L'ENVIRONNEMENT MMS

THÈSE N[O] 1501 (1996)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Pierre CASTORI

Ingénieur diplômé de l'Ecole pour les Etudes et la Recherche en Informatique et Electronique, Nîmes, France
de nationalités française et canadienne

acceptée sur proposition du jury:

Prof. H. Nussbaumer, directeur de thèse
M. N. Chevassus, corapporteur
Prof. A. Strohmeier, corapporteur
Prof. J.-P. Thomesse, corapporteur

Lausanne, EPFL
1996

à mes parents
et à mes grands-parents

Remerciements

Tout d'abord, je tiens à remercier mon directeur de thèse, M. le Professeur Henri Nussbaumer, qui m'a donné la chance de pouvoir travailler dans son laboratoire et m'a incité et encouragé à faire cette thèse. M. Nussbaumer a toujours su trouver le temps pour lire, critiquer et corriger les différents chapitres de ce mémoire, malgré une charge de travail importante. Il a également toujours su trouver les idées justes qui m'ont motivé et permis d'avancer. Plus que le résultat final représenté par cette thèse, c'est une méthode de travail que j'ai appris à son contact. Je tiens à lui exprimer ici toute ma reconnaissance.

L'aide et les conseils de Patrick Pleinevaux qui a suivi de près mon travail depuis des années ont été plus qu'essentiels pour mener cette thèse à bien. Je lui suis redevable de la genèse de nombreuses idées présentes dans ce manuscrit. Je tiens à lui exprimer mes remerciements les plus sincères pour son dévouement et ses encouragements.

Merci également à M. le Professeur Jean-Dominique Decotignie qui a été d'un grand secours en m'aidant à construire et à structurer le chapitre sur l'ordonnancement dans MMS.

Je voudrais ensuite remercier les membres du jury qui ont accepté la pénible tâche de lire et de critiquer ce manuscrit :

- M. le Professeur André Schiper, EFPL, Directeur du Laboratoire de Systèmes d'Exploitation, Président du Jury
- M. Nicolas Chevassus, Aérospatiale, Chef de projet productique
- M. le Professeur Alfred Strohmeier, EPFL, Directeur du Laboratoire de Génie Logiciel
- M. le Professeur Jean-Pierre Thomesse, CRIN-CNRS, INPL, Directeur Adjoint du Centre de Recherche en Informatique de Nancy

Anne Schlageter a été d'une aide plus qu'appréciable dans la préparation des différents voyages pour les conférences et plus généralement en se chargeant (ou plutôt en me déchargeant) de nombreux problèmes administratifs. Je tiens à la remercier chaleureusement pour ses encouragements mais aussi pour sa constante disponibilité et sa bonne humeur, deux qualités récemment renforcées par Carinne Hornung que je voudrais remercier également.

Je tiens tout particulièrement et tout spécialement à remercier mes amis, Cris Fuhrman et Kateel Vijayananda, qui m'ont à plusieurs reprises apporté le soutien moral nécessaire pour poursuivre cette thèse pendant des périodes particulièrement difficiles, sans causes directes avec mon travail. Leur amitié est un bien précieux que je n'oublierai pas.

Les discussions souvent passionnées mais toujours des plus fructueuses avec Kateel Vijayananda, Guevara Noubir, Franck Vamparys et Ming Li ont fortement contribué à développer nombre d'idées présentes dans cette thèse. Je tiens à les en remercier profondément.

Je voudrais aussi remercier Hector Fabio Restrepo pour sa gentillesse, son amitié et sa disponibilité permanente et sans qui toute personne navigant dans le laboratoire MAP se trouve totalement perdue !

Je souhaiterais aussi remercier l'ensemble des collaborateurs du Laboratoire d'Informatique Technique qui entretiennent cette ambiance d'échanges chaleureux et amicaux indispensables, je le pense, à la bonne réalisation d'une thèse.

Enfin, je remercie mes parents et tous mes proches pour leurs encouragements et leur constant support tout au long de cette période de travail.

Résumé

"Manufacturing Message Specification" (MMS) est un protocole de la couche application de l'OSI dont le but est de permettre le contrôle par des applications distantes (les clients) d'équipements industriels divers et hétérogènes (les serveurs). Cette messagerie industrielle est en train de devenir un point de passage incontournable dans le monde des applications industrielles mais aussi dans d'autres domaines qui ne sont pas limités à la productique. Mais si MMS règle le problème de l'hétérogénéité c'est au prix d'une norme complexe, encore trop méconnue et souvent sous-exploitée. L'objectif de cette thèse est de proposer une architecture des systèmes basés sur la norme MMS qui permette à la fois d'en simplifier l'utilisation et d'en étendre les possibilités pour mieux satisfaire les besoins des utilisateurs.

Nous commençons par effectuer une étude détaillée des fonctionnalités MMS que sont les sémaphores et les événements, aspects de MMS encore sous-utilisés pour ne pas dire inconnus. La méthode suivie est de ramener les sémaphores et événements MMS à des concepts similaires et connus. Cette approche qui n'a semble-t-il pas encore été suivie par d'autres auteurs nous permet de clarifier les concepts de sémaphore et d'événement MMS en vue de faciliter leur compréhension et leur utilisation dans les applications MMS.

Il semble qu'il n'existe à ce jour aucun serveur MMS qui implante la totalité des services et fonctionnalités décrits dans la norme. Notre but est de proposer une architecture générale d'un tel serveur qui favorise une implantation facile et harmonieuse de tous les services et intègre plus particulièrement la gestion des événements. Une telle architecture générique a été élaborée et a fait l'objet d'une implantation. Elle est composée de sous unités dont chacune est responsable pour le traitement d'un groupe particulier de services MMS. Ces unités sont contrôlées par le Processeur de Transactions qui constitue le coeur même du serveur. L'avantage de cette architecture vient de sa clarté et du découplage des diverses unités. Elle permet une modification aisée du serveur ainsi que l'ajout de nouveaux services.

Les événements MMS font l'objet d'une étude approfondie. Nous définissons une extension à la détection des événements dont le but est de mieux satisfaire les besoins des applications industrielles basées sur MMS. Cette extension permet aux utilisateurs de définir eux-mêmes les conditions menant à la détection d'un événement, ce qui n'est pas possible dans la norme actuelle. Elle a pour avantage de rester complètement compatible avec les applications MMS existantes dans la mesure où le protocole MMS n'est pas modifié et le comportement des serveurs est seulement étendu.

Une autre extension à la norme est également proposée. Elle permet aux utilisateurs de fournir des priorités aux services MMS et aux serveurs d'exécuter les requêtes en fonction de ces priorités. Plus généralement, le problème de l'ordonnancement des requêtes et des activités concurrentes d'un serveur est un point essentiel pour les applications industrielles temps-réel mais qui n'a pas du tout été abordé par la norme MMS. Nous analysons donc les possibilités et limitations de MMS dans le domaine du temps-réel. Nous montrons comment associer des échéances aux requêtes de service MMS sans modifier la norme. Nous montrons également comment exploiter les modificateurs MMS pour satisfaire une exécution temps-réel des requêtes de service et proposons des solutions pour assurer une

détection en temps-réel des événements MMS de type scruté.

MMS ne semble pas avoir été conçu pour être intégré aisément à des systèmes répartis. Pourtant, la répartition sur différents sites d'applications coopérantes ou travaillant en parallèle est une constante toujours plus forte des scénarios réels. Nous étudions le comportement des systèmes MMS en présence de nombreux serveurs et clients entrant en compétition ou collaborant à la réalisation d'une tâche commune. Ce faisant, nous cherchons toujours à minimiser l'impact que ceci pourrait avoir sur le protocole MMS. Nous nous attachons à résoudre, avec MMS seulement, quelques uns des problèmes classiques comme les rendez-vous entre clients, lecteurs/rédacteurs et problème des philosophes. Par ailleurs, deux algorithmes gérant la concurrence d'accès aux données réparties sur des sites MMS sont proposés. Ces algorithmes sont basés sur des techniques de détection/résolution des interblocages et utilisent abondamment les services de sémaphores et d'événements MMS.

Abstract

"Manufacturing Message Specification" (MMS) is an OSI application layer protocol that allows remote applications (called clients) to control and supervise various heterogeneous industrial devices (called servers). MMS is becoming widely accepted as the main protocol for open communications between heterogeneous machines in many areas that are not limited to manufacturing. MMS facilitates the cooperation of heterogeneous devices. But, this is done at the cost of a rather complex and under-utilized standard. The aim of this thesis is to propose an architecture of MMS-based systems that simplifies the use of MMS and extends its capabilities to satisfy more closely the needs of user applications.

We start with a detailed analysis of MMS events and semaphores. These two aspects of MMS are still misunderstood and often even ignored. We compare MMS semaphores and events to well-known and similar concepts in fields not related to MMS. This approach allows us to clarify the concept of semaphore and event in MMS. It facilitates the understanding we have of these concepts and encourages their use in MMS applications.

We do not know of any MMS server that offers all of the services and functionalities described in the MMS standard. We propose a general architecture of such servers that eases the implementation of all MMS services and integrates more specifically the management of events. A server based on such a generic architecture has been implemented. It is composed of several units. Each unit is responsible for the management of a specific group of MMS services. These units are controlled by the Transaction Processor which constitutes the core of the server. The benefits of this architecture rely in the clarity it provides and the decoupling of the various units. This architecture facilitates server modifications as well as the addition of new services.

We provide a deeper analysis of MMS events by defining an extension to MMS event detection. This extension satisfies more closely the needs of industrial applications based on MMS. It allows user applications to provide predicate expressions that can lead to an event occurrence. This is something that is not possible in the current MMS standard. This extension is entirely compatible with existing MMS applications since the MMS protocol is not modified and the server behavior is only extended.

We also propose another extension to MMS. This last extension allows users to include priorities to MMS service requests and allows servers to execute these requests based on the priority values. More generally, the problem of requests and tasks scheduling in a server is a major issue for industrial applications. However, it is not addressed at all by MMS. Thus, we analyze the capabilities and limitations of MMS in the domain of real-time systems. We show how requests can be given deadlines without modifying the MMS protocol. We make use of MMS modifiers to satisfy a real-time execution of MMS requests and propose solutions to ensure a real-time detection of MMS events.

MMS does not seem to provide application developers with the adequate tools to deal with distributed systems requirements. We study the behavior of MMS systems when applications require the cooperation or competition of multiple clients in the presence of multiple servers. In doing so, we always try to reduce the influence our solutions could have on the MMS protocol. We propose MMS-based

solutions for classical problems such as rendez-vous between clients, readers/writers and the dining philosophers problem. We also define two algorithms for accessing data distributed to several MMS sites. These algorithms are based on deadlock detection/resolution techniques and make an abundant use of MMS semaphores and events.

Table des matières

1 Introduction **1**

1.1 Le protocole MMS 1

1.1.1 Disposition d'un site MMS 2

1.1.2 Machine Virtuelle de Fabrication 3

1.1.3 La représentation par objet dans MMS 3

1.1.4 Associations d'application 4

1.1.5 Adressage et désignation 4

1.1.6 Services confirmés 5

1.1.7 Services non confirmés 6

1.1.8 Modificateurs 6

1.2 Cadre de travail 6

1.2.1 Objectifs 6

1.2.2 Démarche suivie 7

1.2.3 Niveaux de modification de la norme MMS 8

1.2.4 Avantages et inconvénients 9

1.3 Organisation 9

2 Les sémaphores MMS **15**

2.1 Introduction 15

2.2 Aperçu des différences et similitudes 16

2.2.1 Equivalences simples entre sémaphores MMS et sémaphores classiques 16

2.2.2 Différences structurelles 17

2.3 Comparaison entre les sémaphores MMS et les sémaphores classiques 19

2.3.1 Les points communs 19

2.3.2 Que peut-on faire avec les sémaphores MMS que l'on ne peut pas faire avec les sémaphores classiques? 20

2.3.3 Que peut-on faire avec les sémaphores classiques que l'on ne peut pas faire avec les sémaphores MMS? 24

2.3.4 Qu'est ce qui est spécifique à MMS? . . . 27
2.3.5 L'invariant du sémaphore MMS . . . 30
2.4 Pourquoi deux classes de sémaphores . . . 35
2.5 Sémaphores MMS et interblocages . . . 36
2.6 Sémaphores MMS et CSP . . . 37
2.7 Conclusion sur les sémaphores MMS . . . 38

3 Les événements MMS **41**
3.1 Introduction . . . 41
3.2 Définitions d'un événement . . . 41
3.3 Conditions d'apparition d'un événement . . . 43
3.3.1 Apparition par déclenchement . . . 43
3.3.2 Apparition par scrutation . . . 45
3.4 Création d'une condition événementielle . . . 47
3.5 Visibilité d'une condition événementielle . . . 48
3.6 Notification d'un événement . . . 49
3.7 Souscription à un événement . . . 50
3.8 Exécution d'actions liées aux événements . . . 52
3.9 Filtrage des événements scrutés . . . 53
3.10 Conditionnement d'un service sur l'apparition d'un événement . . . 55
3.11 Limitations des événements MMS . . . 58
3.11.1 Détection basée sur un prédicat . . . 58
3.11.2 Ordonnancement . . . 60
3.11.3 Temps de validité d'une action . . . 62
3.11.4 Répartition . . . 62
3.11.5 Diffusion des notifications . . . 65
3.12 Pourquoi deux classes d'événements? . . . 66
3.13 Conclusion sur les événements . . . 66

4 Architecture d'un serveur MMS **69**
4.1 Introduction . . . 69
4.1.1 Exigences . . . 69
4.1.2 Hypothèses . . . 70
4.1.3 Notes sur les représentations "par objet" . . . 71
4.2 Un modèle pour l'exécution des services MMS . . . 73
4.2.1 Les objets Transaction . . . 73

4.2.2 Services bloquants et services immédiats 74
4.2.3 Services suspensibles 76
4.2.4 Etats d'un service MMS confirmé 77
4.2.5 Aperçu général de l'architecture d'un serveur 79
4.3 Les activités concurrentes d'un serveur 80
4.3.1 Identification des activités concurrentes 80
4.3.2 L'ordonnancement des activités 82
4.3.3 La communication entre activités 84
4.4 Le Processeur de Transactions 85
4.4.1 Opérations exportées 87
4.4.2 Composition 89
4.5 Les gestionnaires de services MMS 89
4.5.1 Structure générale 89
4.5.2 Traitement particulier des services bloquants 91
4.5.3 Gestionnaires particuliers 92
4.5.4 Relations de dépendance 93
4.5.5 Comportement des gestionnaires simples 95
4.5.6 Le gestionnaire d'événements 96
4.6 MMS et systèmes d'exploitation 99
4.7 Analyse et performances 102
4.7.1 L'implantation 102
4.7.2 Quelques mesures de performance 104
4.8 Conclusion 106

5 Principales extensions proposées 109

5.1 Introduction 109
5.2 Extension à la détection d'événements MMS 109
5.2.1 Introduction 109
5.2.2 XED vu depuis les clients 110
5.2.3 XED vu depuis les serveurs 112
5.2.4 Les problèmes introduits par XED 116
5.2.5 Impact de XED sur la norme MMS 119
5.2.6 Intégration de nouveaux types d'événement 121
5.2.7 Implantation de XED 124
5.2.8 Conclusion sur XED 126
5.3 Des priorités pour les services MMS 126

5.3.1 Introduction : le besoin d'avoir des priorités 126
5.3.2 L'affectation des priorités aux services MMS 128
5.3.3 L'ordonnancement des services MMS 129
5.3.4 L'inversion de priorité 130
5.3.5 Problèmes dus à l'héritage de priorité 133
5.3.6 Impact de XES sur la norme MMS 134
5.3.7 Ordonnancement avec des échéances 135
5.3.8 Conclusion sur XES 136
5.4 Conclusion 137

6 MMS et ordonnancement 139

6.1 Introduction 139
6.1.1 Bref état de l'art 140
6.1.2 Définitions 141
6.1.3 Modèle et exigences 142
6.2 Modèles de serveurs MMS 143
6.3 Quelles limitations pour MMS ? 146
6.3.1 Services déterministes 146
6.3.2 Associations d'application 148
6.3.3 Détection et traitement des événements 148
6.4 Systèmes à contraintes de temps strictes 148
6.4.1 Paramètres essentiels 149
6.4.2 Méthode de négociation des paramètres temporels 150
6.4.3 Extension de l'objet `Association` 151
6.4.4 Combinaison avec des associations non temps-réel 152
6.4.5 Tests d'ordonnancement 153
6.5 Systèmes à contraintes de temps lâches 157
6.5.1 Première approche : test de faisabilité à la réception 157
6.5.2 Seconde approche : utilisation du temps de contrôle d'un sémaphore MMS . . . 159
6.5.3 Troisième approche : utilisation du temps d'attente d'un sémaphore MMS . . . 160
6.5.4 Ordonnancement des requêtes 163
6.5.5 Politique d'allocation des jetons du sémaphore 167
6.5.6 Combinaison des `Acceptable Delay` et des `Control Timeout` 170
6.5.7 Implantation 172
6.6 Détection temps-réel des événements 173
6.6.1 Modèle pour la détection d'événements 175

6.6.2 Algorithmes utilisés . . . 176
6.6.3 Démarrage du nouvel ordonnancement . . . 178
6.6.4 Exécution des autres tâches du serveur . . . 181
6.6.5 Prise en compte des ressources partagées . . . 181
6.7 Conclusion . . . 183

7 MMS dans un environnement de systèmes répartis 185
7.1 Introduction . . . 185
7.2 Solutions MMS à des problèmes classiques . . . 186
7.2.1 Rendez-vous MMS . . . 186
7.2.2 Lecteurs/Rédacteurs . . . 190
7.2.3 Le problème des philosophes . . . 199
7.3 Gestion répartie des événements et services MMS . . . 205
7.3.1 Temps de validité d'une action événementielle . . . 205
7.3.2 Exécution d'une action événementielle distante . . . 207
7.3.3 Exécution d'une requête de service MMS distante . . . 211
7.4 Algorithmes de contrôle de la concurrence . . . 214
7.4.1 Hypothèses de base . . . 214
7.4.2 Modèle et notations . . . 215
7.4.3 Résolution des interblocages par temporisation . . . 216
7.4.4 Détection des interblocages : la méthode des sondes . . . 231
7.5 Conclusion . . . 237

8 Conclusion 239
8.1 Principales contributions . . . 239
8.2 Perspectives futures . . . 241

A Conventions 243

B Abbréviations utilisées 245

C Lexique des termes utilisés 247

D Tableau récapitulatif des services MMS 249

E Code PROMELA pour la vérification des algorithmes proposés 253

F Scénarios d'application 273
F.1 Installation d'embouteillage de produits chimiques . . . 273

F.1.1 Le scénario 273
F.1.2 Modélisation avec MMS 274
F.2 Scénario Renault : serveurs de terminaux 276
F.2.1 Le scénario 276
F.2.2 Exemple de solution 278
F.3 Serveur d'impression 280

Table des figures

1.1 Disposition générale d'un site MMS . . . 3
1.2 Le schéma client/serveur dans MMS . . . 5
1.3 Schéma de l'articulation entre les chapitres . . . 11
1.4 Structuration générale de la thèse . . . 13

2.1 Les objets MMS Semaphore et Semaphore Entry . . . 17
2.2 Le modèle des objets SE . . . 21
2.3 Rendez-vous avec les sémaphores MMS . . . 25
2.4 Solutions avec modificateurs et avec services `Take Control` et `Relinquish Control` . . . 29
2.5 Sémaphore MMS banalisé en CSP . . . 38

3.1 L'objet Condition Evénementielle . . . 43
3.2 Evénement MMS déclenché . . . 44
3.3 Evénement MMS scruté . . . 45
3.4 Exemples de transitions d'événements scrutés . . . 46
3.5 L'objet Enregistrement Evénementiel . . . 51
3.6 L'objet Action Evénementielle . . . 52
3.7 Filtrage des événements dans MMS . . . 54
3.8 Modification d'un service MMS . . . 55
3.9 Comparaison des solutions avec modificateur et avec action pour un service `Read` . . . 57
3.10 Mesure de R_{evt} en fonction du nombre d'octets dans la réponse `Read` selon la fréquence de l'événement . . . 58
3.11 Séquencement des étapes lors d'une action locale et répartie . . . 63
3.12 Séquencement des étapes lors d'une action à distance avec deux notifications . . . 64

4.1 Représentation schématique de l'application MMS et de la pile de communication . . . 70
4.2 L'objet Transaction défini par MMS . . . 74
4.3 Service bloquant et service immédiat . . . 75
4.4 Classification des services MMS selon leur caractère bloquant . . . 76
4.5 Machine d'états d'un service MMS confirmé . . . 78

4.6 Architecture générale d'un serveur MMS . . . 81
4.7 Activités concurrentes et architecture d'un serveur MMS . . . 83
4.8 Ordonnancement des activités concurrentes . . . 85
4.9 Structure du Processeur de Transactions . . . 90
4.10 Exemple d'exploitation du temps d'arrêt d'un service bloquant . . . 91
4.11 Relations de dépendance entre les objets MMS . . . 94
4.12 Gestionnaire d'événements . . . 97
4.13 L'objet `Event Notification` . . . 100
4.14 Structure simplifiée des systèmes d'exploitation et de MMS . . . 101
4.15 Distribution des temps de réponse des services MMS . . . 105
4.16 Exécutions synchrone et asynchrone des services MMS . . . 107

5.1 Un exemple d'utilisation classique des événements MMS . . . 111
5.2 Configuration des objets dans XED . . . 112
5.3 Un exemple d'utilisation de XED avec les événements MMS . . . 113
5.4 Séquence de services pour définir un événement avec XED . . . 115
5.5 Variation du temps d'évaluation d'une expression selon le nombre de variable qu'elle contient 125
5.6 Structure de l'`InvokeId` utilisé dans XES . . . 130
5.7 L'inversion de priorité avec les services MMS . . . 131
5.8 L'héritage de priorité rendu inutile . . . 134
5.9 L'objet transaction étendu pour XES . . . 136

6.1 Définition des temps utilisés dans ce chapitre . . . 142
6.2 Nombre de variables pouvant être lues en une seule requête `Read` . . . 147
6.3 Négociation des paramètres temps-réel . . . 152
6.4 Approche test de faisabilité à la réception de chaque requête . . . 158
6.5 Utilisation du temps d'attente d'un sémaphore . . . 162
6.6 Comportement des trois approches étudiées . . . 163
6.7 Utilisation du temps d'attente d'un sémaphore avec ordonnancement . . . 165
6.8 Utilisation simultanée des deux techniques . . . 171
6.9 Automate de Nutt pour le traitement d'une requête . . . 172
6.10 Temps d'attente du sémaphore et `Acceptable Delay` . . . 173

7.1 Première solution de rendez-vous proposée dans [AM95b] . . . 187
7.2 Solution simplifiée du rendez-vous proposée dans [AM95b] . . . 187
7.3 Rendez-vous avec les sémaphores MMS . . . 188
7.4 Seconde solution de rendez-vous proposée dans [AM95b] . . . 188

7.5 Un rendez-vous manqué par l'application A2 . . . 189
7.6 Solution de rendez-vous basée sur les événements MMS . . . 189
7.7 Solution de rendez-vous multiples basée sur les événements MMS . . . 190
7.8 Définition ASN.1 du service `Take Control` étendu . . . 196
7.9 Définition ASN.1 du service `ReportSemaphoreEntryStatus` étendu . . . 198
7.10 Le modèle étendu des objets SE . . . 199
7.11 Le problème des philosophes . . . 200
7.12 Le problème des philosophes avec et sans XED . . . 205
7.13 Définition ASN.1 du service `Define Event Action` étendu . . . 207
7.14 L'objet Action Evénementielle étendu avec l'intervalle de validité . . . 207
7.15 Définition ASN.1 du service `Report Semaphore Entry Status` étendu . . . 208
7.16 L'objet Action Evénementielle étendu avec l'adresse du serveur distant . . . 209
7.17 Séquencement des étapes lors d'une action distante avec modificateurs . . . 210
7.18 Définition ASN.1 du service `Get Event Action Attributes` étendu . . . 211
7.19 Automate de Nutt pour l'exécution distante des actions . . . 212
7.20 L'objet transaction étendu . . . 213
7.21 Automate de Nutt pour l'exécution distante des services . . . 213
7.22 Variations de V en fonction du nombre de clients . . . 222
7.23 Temps de réponse moyen du système . . . 223
7.24 Temps de réponse moyen par transaction . . . 224
7.25 Nombre moyen de redémarrages par transaction . . . 224
7.26 Variations de V en fonction du nombre de clients et pour 3 sémaphores . . . 225
7.27 Variations de V en fonction du nombre de clients et pour 5 sémaphores . . . 226
7.28 Variations de V en fonction du nombre de clients et pour 10 sémaphores . . . 226
7.29 Différences entre les valeurs de $\bar{R}$ obtenues en théorie et par simulation . . . 229
7.30 Circulation de la sonde entre clients et serveurs . . . 234

F.1 Architecture générale de l'installation . . . 274
F.2 Contrôle de l'embouteillage . . . 276
F.3 Architecture générale du système . . . 277
F.4 Choix initial de l'application à exécuter . . . 278
F.5 Exécution de l'application et reprise lors d'une panne . . . 279
F.6 Serveur d'impression . . . 280

Liste des tableaux

1.1 Compatibilité d'un client classique avec les serveurs modifiés 10
1.2 Compatibilité d'un serveur classique avec les clients modifiés 10

2.1 Correspondance entre sémaphores MMS et sémaphores classiques 16
2.2 Correspondance entre les attributs des sémaphores MMS et des sémaphores classiques 19
2.3 Comparaison entre le modificateur `Attach To Semaphore` et les services `Take Control / Relinquish Control` 30
2.4 Effets d'un `Take Control` 33
2.5 Effets d'un `Relinquish Control` 33
2.6 Effets d'un `Take Control` avec réquisition 33
2.7 Effets de l'expiration d'un temporisateur `Control Timeout` 34
2.8 Effets d'une perte d'association 34
2.9 Effets de l'expiration d'un temporisateur `Acceptable Delay` 34

3.1 Comparaison des notifications d'événements CMISE et MMS 50
3.2 Comparaison de l'utilisation de modificateurs et d'actions événementielles 56

4.1 Principales actions exécutées lors des transitions 79
4.2 Transitions d'état effectuées lors des activités concurrentes 86
4.3 Communications entre gestionnaires 96
4.4 Procédures de traitement des transitions d'événement 99
4.5 Taille de chaque objet principal constitutif du serveur 103
4.6 Temps de réponses pour certains services d'événements 105
4.7 Temps d'exécution interne de certains services 106

5.1 Transitions d'événement dans MMS classique et dans XED 114
5.2 Transitions dues à un mauvais usage de XED 119
5.3 Réutilisation des codes d'erreur MMS dans XED 121
5.4 Exemple de composition d'événements XED 124
5.5 Temps d'évaluation des différents lexèmes 125

5.6 Comparaison entre les trois façons d'affecter des priorités dans MMS 131

6.1 Nombre de variables pouvant être lues en une seule requête `Read` 147
6.2 Comparaison de différente types d'association 151
6.3 Comparaison entre les trois approches présentées 168
6.4 Conflits d'accès aux objets MMS (R - *accès en lecture* / W - *accès en écriture*) 182

7.1 Comparaison entre les différentes solutions au problème des philosophes 205
7.2 Nombre moyen d'accès aux ressources par transaction lorsque toutes les transactions y ont accédé au moins une fois. 225
7.3 Tableau récapitulatif des solutions et extensions proposées 238

D.1 Tableau récapitulatif 249
D.2 Tableau récapitulatif (suite) 250
D.3 Tableau récapitulatif (suite) 251

Chapitre 1

Introduction

1.1 Le protocole MMS

"Manufacturing Message Specification" (MMS) est une norme ISO ("International Standards Organization") conçue pour assurer l'interfonctionnement de dispositifs industriels hétérogènes [HB90], [BG91], [Nus91b]. Ces dispositifs peuvent être aussi variés que des calculateurs, des automates, des robots, des commandes numériques ou des ordinateurs conventionnels.

MMS est le composant essentiel du profil MAP ("Manufacturing Automation Protocol") développé par General Motors en 1988 [Gen88]. MAP est une suite de protocoles normalisant le comportement de l'ensemble des sept couches du modèle OSI ("Open System Interconnection"). Le but de MAP est de permettre l'intégration des différents équipements informatiques d'une usine en un système cohérent. MAP assure donc l'**interconnexion** entre ces divers équipements. L'ajout de l'élément MMS permet alors de résoudre en plus le problème important de l'**interfonctionnement** entre ces dispositifs [NC90].

Dans l'environnement OSI, MMS est un élément de service application ou ASE ("Application Service Element"). Il se situe donc au niveau de la couche 7 Application. MMS offre un ensemble complet de services génériques couvrant les besoins des diverses applications en milieu industriel.

La norme MMS se compose de plusieurs parties :

- La *définition des services* [ISO90a] : on trouve dans cette partie la description du comportement visible d'une application MMS. Les services fournis par MMS ainsi que les paramètres associés à ces services sont décrits en détail. Dans cette partie l'accent est donc mis sur le **comportement** d'un serveur MMS ainsi que sur la **perception** des fonctionnalités offertes par un serveur.
- La *spécification du protocole* [ISO90b] : on trouve dans cette partie la syntaxe abstraite du protocole MMS c'est-à-dire le format que doivent avoir les données échangées. Ces données sont aussi appelées *PDUs* ("Protocol Data Units"). La syntaxe abstraite de MMS est décrite en ASN.1 (Abstract Syntax Notation 1) [ISO87a]. Le document [ISO90b] établit aussi les règles permettant l'échange d'information entre deux applications MMS. Dans cette partie, l'accent est donc mis sur le **format** des données MMS.
- Les *normes d'accompagnement* (ou "Companion Standards") : MMS offre des services génériques applicables à tout équipement industriel. Parallèlement, il est nécessaire de personnaliser l'utilisation de MMS pour les diverses classes d'appareils informatiques. Ainsi, les normes d'accompagnement traitent des aspects propres à une classe d'équipement industriel donnée et viennent en complément des services MMS génériques. Il existe des normes d'accompagnement pour les

robots, les automates programmables, les commandes numériques, etc. Pour rester aussi général que possible, dans cette thèse nous ne considérons pas les normes d'accompagnement.

MMS fonctionne en mode client/serveur orienté-connexion. Les applications client requièrent la réalisation d'une opération sur des applications distantes (les applications serveur). Pour cela les clients effectuent des requêtes de service vers les serveurs. Les serveurs répondent en retour ou indiquent qu'une erreur s'est produite. Avant tout transfert de données il est nécessaire d'établir une connexion entre les applications communiquantes. L'ouverture de cette connexion implique une négociation des services utilisés et des paramètres mis en jeu pour la session MMS qui démarre.

MMS a initialement été développé pour être appliqué dans le domaine des applications manufacturières et du contrôle de procédés industriels. En fait son champs d'application est beaucoup plus vaste et MMS est souvent utilisé dès qu'il s'agit de régler un problème d'interopérabilité entre différents équipements informatiques hétérogènes. Au départ conçu pour les réseaux locaux industriels, MMS s'est progressivement adapté à une utilisation sur réseaux longue distance. Les divers projets visant à intégrer MMS et TCP/IP ainsi que la future norme TASE.2 (Telecontrol Application Service Element)[1] témoignent de l'intérêt actuel de ne pas limiter MMS aux seules applications fonctionnant sur un réseau local.

1.1.1 Disposition d'un site MMS

Un "site" MMS est divisé en trois ou quatre parties distinctes selon qu'il s'agisse d'un client ou d'un serveur. Ces parties sont superposées et font usage l'une de l'autre selon le modèle classique par couches :

- *Pile de communication* : la pile de communication comprend l'ensemble des protocoles nécessaires pour assurer une communication fiable entre deux sites MMS. La pile de communication comprend les couches 1 à 6 du modèle OSI.
- *Fournisseur de services MMS* (ou "MMS provider") : le fournisseur de services offre aux applications MMS une interface du type envoyer/recevoir. Il communique directement avec la couche Présentation et l'élément de service ACSE ("Association Control Service Element"). Il fournit aux applications qui l'appellent des primitives de communication et d'établissement de connexions. Le fournisseur de services contient la Machine de Protocole MMS (MMPM). La MMPM gère les règles permettant les échanges de messages entre deux applications MMS. Elle est symétrique et identique du coté client comme du coté serveur. Le fournisseur de services permet aussi de faire le codage/décodage des PDUs MMS.
- *Application MMS* (ou "MMS user") : l'application MMS en tant que telle est la partie "intelligente" du site. S'il agit du serveur, elle coordonne et exécute les différentes requêtes de service et plus généralement exécute toutes les fonctions associées à un serveur MMS. S'il s'agit du client, c'est l'application utilisateur proprement dite. Son comportement n'est pas normalisé par MMS.
- *Dispositif industriel* : le but de MMS étant de permettre la communication avec des dispositifs industriels, un serveur MMS masque généralement un tel dispositif. L'application MMS joue en quelque sorte le rôle de pilote pour le dispositif. Elle masque la complexité du dispositif et permet un accès simple et uniforme à l'information qu'il contient. Notons que l'interface entre l'application serveur et le dispostif sous-jacent n'est pas du ressort de MMS.

Cette disposition en plusieurs parties est représentée sur la figure 1.1.

[1]TASE.2 est un protocole basé sur MMS assurant la communication entre centres de contrôle pour la conduite de systèmes de puissance.

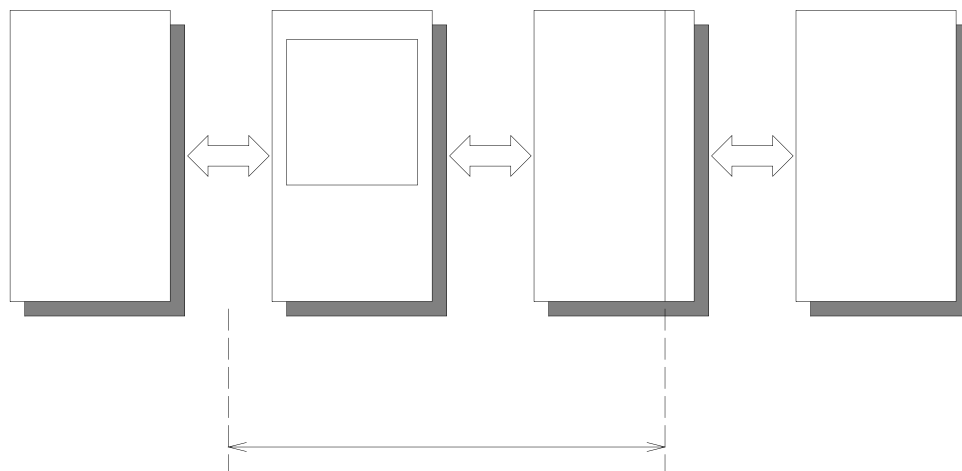


FIG. 1.1 - Disposition générale d'un site MMS

1.1.2 Machine Virtuelle de Fabrication

Une machine virtuelle de fabrication (ou VMD pour "Virtual Manufacturing Device") est une représentation abstraite d'un ensemble d'interfaces et de fonctionnalités appartenant à un système réel de production. Une VMD représente l'image qu'ont les applications client d'un équipement réel de production [Wac93]. Un serveur doit contenir une ou plusieurs VMD alors qu'un client lui n'en contient pas (à moins qu'il ne soit aussi serveur).

Une VMD comprend la fonction opératoire ("executive function") qui représente la capacité de traitement de la VMD. La fonction opératoire exécute les requêtes de service, effectue les opérations locales sur les objets, gère l'accès aux ressources, etc.

1.1.3 La représentation par objet dans MMS

L'information contenue dans une VMD est modélisée sous forme d'*objets abstraits*. Malgré l'utilisation du terme "objet", MMS ne se préoccupe pas du tout des notions orienté-objet classiques telles que l'encapsulation des données, l'héritage, etc. Le terme "objet" doit ici s'entendre dans son sens le plus restreint c'est-à-dire "structure de donnée". On peut toutefois retrouver certaines notions orienté-objet classiques dans la définition des objets MMS. A ce sujet, une excellente analyse du formalisme objet de MMS a été effectuée par Pleinevaux dans [Ple94a].

Un objet se définit par ses attributs et par les opérations qui peuvent lui être appliquées. Les attributs des objets MMS sont de types divers : booléens, entiers, chaînes de caractères, listes d'autres objets MMS, etc.

Chaque objet est une instance particulière d'une classe d'objets. Par abus de langage, on confond souvent un objet et sa classe. Un objet MMS est purement descriptif et ne préjuge pas de l'implantation qui en est faite. L'objet MMS est souvent la représentation visible de l'information contenue dans le dispositif physique sous-jacent à la VMD. A ce titre, un objet peut être pré-défini dans la VMD au moment de l'implantation. Mais il peut aussi être créé dynamiquement au moyen de certains services MMS.

Un objet est accessible par son nom (objet nommé) ou par une adresse (objet anonyme). Le nom

d'un objet doit être unique dans son environnement. MMS définit également la *visibilité* d'un objet. Celle-ci peut être de trois types :

- spécifique à la VMD (ou "VMD-specific") : l'objet est alors visible par tous les clients;
- spécifique à un domaine (ou "domain-specific") : l'objet est alors visible par les clients ayant accès au domaine concerné;
- spécifique à une connexion (ou "application-association-specific") : l'objet est alors visible sur une seule connexion donc pour un seul client.

1.1.4 Associations d'application

MMS est un protocole orienté connexion. Avant toute interaction entre un client et un serveur, il est nécessaire d'ouvrir une connexion entre ces deux applications. Dans la terminologie MMS cette connexion est appelée *association d'application* ou simplement *association*. Une association est un lien logique entre deux applications. Une fois l'association ouverte, les applications MMS peuvent communiquer. L'association est bi-directionelle, et donc les données peuvent s'échanger dans les deux sens sur une association.

L'ouverture d'une association doit se négocier. Elle se fait au moyen du service `Initiate`. L'application appelante (généralement le client) propose un certain nombre de paramètres et de fonctionnalités qu'elle désire utiliser. L'application appelée (généralement le serveur) accepte ou refuse la proposition. Elle peut aussi proposer en retour d'ouvrir l'association mais avec des paramètres moins contraignants.

L'association peut se terminer à tout moment par l'une ou l'autre des deux applications. On fait toutefois la distinction entre la terminaison brutale (service `Abort`) et la terminaison normale (service `Conclude`). La première détruit tous les services en cours alors que la seconde ne permet la rupture de l'association que si les services en cours sont terminés.

1.1.5 Adressage et désignation

Pour établir une association entre deux applications, il est nécessaire que l'appelant connaisse le "nom" de l'application destinataire. MMS utilise la dénomination et les fonctions OSI pour tout ce qui concerne la désignation et l'adressage des applications. Pour établir une association avec un serveur, un client MMS doit normalement connaître :

- l'*appellation de processus d'application* du serveur (ou "Application Process Title"). Un processus d'application représente l'ensemble des ressources d'un système ouvert qui permettent le traitement de l'information. Une exécution particulière d'un processus d'application est appelée *instance de processus d'application*;
- l'*appellation d'entité d'application* (ou "Application Entity Qualifier") utilisée par l'application serveur. Une entité d'application représente un ensemble de ressources de communication OSI dédiées à un processus d'application. Une activité donnée d'une entité d'application est appelée *instance d'entité d'application*;
- un identificateur d'instance de processus d'application.
- un identificateur d'instance d'entité d'association.

L'établissement des associations se fait en partie grâce à des fonctions de répertoire qui permettent de mettre en correspondance une adresse de la couche présentation avec l'appellation de l'entité d'application destinataire fournie par le demandeur. La création d'une association entraîne l'affectation d'un *identificateur d'association* propre à cette association. Tant que cette association est active, la communication entre les deux applications et la désignation de l'application distante se fait au moyen de cet identificateur. Les quatre paramètres décrits ci-dessus ne sont alors plus utilisés.

Nous ne décrirons pas plus en détail les techniques d'adressage et de désignation sous MMS car nous n'en faisons pas usage dans cette thèse. Nous renvoyons le lecteur à [Nus91a] pp. 329–378 pour une description détaillée de ces concepts.

1.1.6 Services confirmés

Le protocole MMS est basé sur le paradigme bien connu client/serveur. Une application client effectue une requête de service vers une application serveur. Après traitement de cette requête, le serveur envoie une réponse contenant les résultats (ou les erreurs) du traitement. Selon le formalisme OSI, un client envoie une *requête* qui est reçue côté serveur en tant qu'*indication*. Le serveur envoie la *réponse* à cette indication qui devient *confirmation* côté client (fig. 1.2).

Par abus de langage, on parlera souvent de requête pour signifier requête et/ou indication et de réponse pour signifier réponse et/ou confirmation. Selon la requête de service qui est effectuée, il peut y avoir ou non un accès vers le dispositif physique modélisé par la VMD.

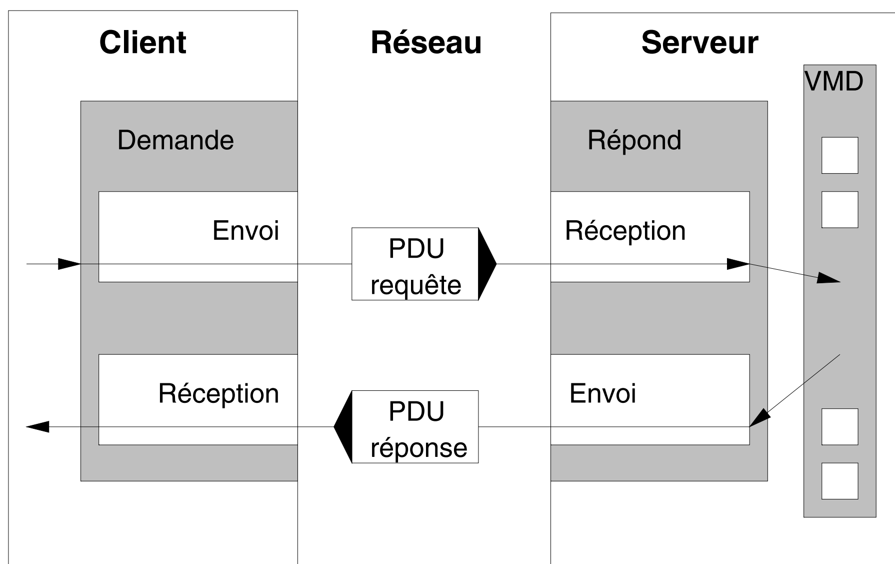


FIG. 1.2 - Le schéma client/serveur dans MMS

MMS propose en tout 78 services confirmés. Il arrive dans certains cas qu'une application serveur effectue une requête vers un client (pour le téléchargement de domaines par exemple). Dans ce cas le serveur se comporte comme un client vis-à-vis du client qui lui-même se comporte comme un serveur vis-à-vis du serveur.

Chaque service confirmé est identifié par un numéro (`InvokeId`) qui est unique sur l'association qui porte cette requête de service et ce pendant toute la durée de vie du service c'est-à-dire jusqu'à ce que la confirmation soit reçue par le client ou que l'association soit terminée. L'unicité est garantie par la MMPM qui de chaque côté rejette toute requête de service dont l'`InvokeId` est identique à celui d'une requête en cours.

Les services MMS `Initiate`, `Conclude` et `Cancel` fonctionnent aussi par requête/réponse mais sont utilisés pour la gestion d'associations et n'ont pas d'`InvokeId`. Dans la définition du protocole MMS ces trois derniers services n'apparaissent pas dans la rubrique des services confirmés. Dans cette thèse, l'expression "services confirmés" ne comprend donc pas ces trois services.

1.1.7 Services non confirmés

MMS autorise aussi l'utilisation de services non confirmés. Pour ceux-ci, seule une requête suivie d'une indication existe. Toutefois, un service non confirmé s'effectue dans la direction serveur-client. C'est toujours le serveur qui prend l'initiative d'envoyer une requête de service non confirmée. Les conditions de lancement d'un service non confirmé ne sont pas normalisées par MMS sauf pour la notification d'événements (`Event Notification`).

Les services non confirmés ne sont qu'au nombre de trois :

- `Unsollicited Status` qui renseigne sur l'état de la VMD;
- `Information Report` qui envoie la valeur d'une variable;
- `Event Notification` qui notifie de l'occurrence d'un événement.

1.1.8 Modificateurs

Dans MMS, il est possible de conditionner l'exécution d'une requête de service confirmé sur la réalisation d'une ou plusieurs conditions. Ces conditions sont appelées *modificateurs* et sont de deux types :

- Apparition d'un événement : la requête de service reste bloquée jusqu'à l'apparition de l'événement souhaité.
- Prise de contrôle d'un sémaphore : la requête de service reste bloquée jusqu'à ce que le sémaphore spécifié soit acquis.

Chaque requête de service MMS confirmé peut contenir une liste de modificateurs. Chaque modificateur doit alors être satisfait séquentiellement dans l'ordre donné par cette liste.

Les modificateurs sont peu utilisés actuellement car méconnus et peu compris. Nous ne connaissons aucun produit MMS qui supporte les modificateurs. Dans cette thèse, nous faisons un usage fréquent des modificateurs dans le but d'illustrer leur intérêt. Nous montrons que l'utilisation des modificateurs permet de résoudre de façon simple et élégante nombre de problèmes dans l'environnement MMS.

1.2 Cadre de travail

1.2.1 Objectifs

Comme l'indique le titre de cette thèse, notre objectif est de proposer une architecture des systèmes basés sur la norme MMS qui permette à la fois d'en simplifier l'utilisation et d'en étendre les possibilités. Notre but est donc de comprendre en détail la norme MMS et de proposer des extensions et améliorations à certains problèmes non encore résolus. Nous pensons en effet que les possibilités

de la norme MMS n'ont pas encore été complètement comprises et exploitées. Ceci provient de la relative complexité de la norme MMS qui ne facilite pas l'obtention d'une perception globale, claire et cohérente de ce qu'est un serveur MMS ni des possibilités offertes par MMS dans des domaines aussi divers que l'ordonnancement, la gestion de données réparties, la synchronisation entre applications client ou tout simplement l'utilisation d'outils tels que les sémaphores et les événements MMS.

Dans cette thèse, nous cherchons à exploiter la norme MMS dans ses moindres détails. Nous souhaitons déterminer les types de problèmes auxquels MMS est adapté et ceux qui nécessitent au contraire des modifications ou des améliorations de la norme. Quand de tels changements sont nécessaires, nous minimisons toujours les modifications apportées et cherchons à conserver l'esprit de la norme MMS.

Globalement, nos objectifs et contributions sont les suivants :

- Comprendre et analyser le fonctionnement de mécanismes peu connus tels que les sémaphores et événements MMS.
- Déterminer les avantages et limitations de ces mécanismes et proposer des solutions alternatives.
- Proposer une architecture générique des serveurs MMS.
- Proposer des extensions à MMS notamment en ce qui concerne la détection d'événements et l'ordre d'exécution des requêtes de service dans les serveurs.
- Analyser dans une certaine mesure le comportement des systèmes MMS dans un environnement temps-réel.
- Fournir une "boîte à outils" d'algorithmes MMS apportant des solutions à des problèmes classiques tels que lecteurs/rédacteurs, philosophes, contrôle de la concurrence, etc. Le but ici est double : d'une part comprendre comment utiliser MMS avec des paradigmes connus et d'autre part fournir des outils pour résoudre un certain nombre problèmes avec MMS.

Cette thèse est donc orientée à la fois vers une **exploration** et une **exploitation** de la norme MMS.

Nous adoptons un point de vue essentiellement applicatif et nous nous concentrons sur l'étude des systèmes MMS seulement. **Nous ne traitons pas** de problèmes tels que :

- la communication et le traitement des messages dans les couches 1 à 6 de la pile OSI et plus généralement dans les couches sous-jacentes à MMS;
- la tolérance aux fautes qu'il s'agisse de fautes dans la communication entre les sites MMS, au sein même des applications MMS en présence ou dans les dispositifs physiques modélisés par les serveurs;
- la collaboration entre MMS et d'autres ASEs de la couche application ainsi que des problèmes d'adressage et de désignation des applications utilisant ces ASEs;
- tout problème lié à la gestion de réseaux, de façon plus générale.

1.2.2 Démarche suivie

Les deux questions qui dirigent cette thèse sont les suivantes :

1. **Que permet la norme MMS dans son état actuel?**

2. **Quelles améliorations peut-on apporter à MMS?**

Dans cette thèse nous identifions des limitations et incomplétudes de la norme MMS et proposons des solutions à ces problèmes. Notre souci constant est de chercher à garantir la compatibilité de ces solutions avec la norme MMS actuelle et de modifier la norme le moins possible.

Nous adoptons généralement la démarche qui consiste à proposer d'abord des solutions basées sur MMS sans faire aucune modification à la norme et sans faire aucune supposition sur le comportement des serveurs MMS. Dans certains cas, cette démarche s'avère insuffisante et il est nécessaire d'étendre les fonctionnalités des serveurs ainsi que le protocole MMS.

Le but essentiel de MMS est d'assurer l'interopérabilité de systèmes et dispositifs hétérogènes. Nous avons toutefois constaté lors d'études résumées dans [BCP+95] et [CPVV95] que cette interopérabilité n'est pas toujours garantie même pour des systèmes totalement conformes aux normes MAP/MMS. Ceci provient de la grande complexité des protocoles utilisés, de leur incomplétude qui laisse parfois trop de choix aux concepteurs ainsi que des nombreuses options possibles qui ne sont pas toujours supportées (justement parce qu'elles ne sont qu'optionnelles) et amènent à faire des choix incompatibles [Mey90]. Nous ne saurions alors modifier la norme MMS sans définir précisément les effets de ces modifications et les incompatibilités éventuellement engendrées par ces changements.

Pour ce faire, il est important de définir plus précisément différents degrés de modification de la norme MMS pour pouvoir savoir jusqu'à quel point il est possible de faire des changements sans altérer la compatibilité des applications MMS classiques avec les applications modifiées. Ceci fait l'objet de la section suivante.

1.2.3 Niveaux de modification de la norme MMS

On peut distinguer différents types de modifications des applications serveur MMS par ordre croissant de changements infligés à la norme MMS :

1. **Aucune modification** : c'est le niveau de base où aucune modification, aucun ajout de nouvelles fonctionnalités ne sont faits à la norme MMS. Le protocole MMS est identique à celui décrit dans [ISO90b]. Le comportement des serveurs MMS est strictement celui décrit par la norme [ISO90a]. On ne peut faire aucune hypothèse quant à ce comportement. Il est cependant entendu que les applications clientes MMS peuvent se comporter de façon quelconque.

2. **Extension des serveurs**: nous appelons *extension d'un serveur MMS* un comportement du dit serveur qui n'est pas décrit par la norme [ISO90a], qui ajoute de nouvelles fonctionnalités au serveur mais qui ne modifie en rien le comportement initial décrit par [ISO90a]. Une extension doit en plus s'intégrer harmonieusement avec le comportement normalisé c'est-à-dire :

   - doit exploiter au mieux les possibilités offertes par celui-ci;
   - ne doit pas modifier le protocole MMS [ISO90b];
   - ne doit pas ajouter de nouveaux services MMS;
   - ne doit pas invalider des services MMS existants;
   - comme pour les normes d'accompagnement, doit respecter l'esprit et l'intention première de la norme MMS [ISO90a].

3. **Modification des serveurs**: ce n'est qu'à partir de ce troisième niveau que de réelles modifications de la norme sont considérées. On autorise ici non seulement les extensions définies au

niveau précédent mais aussi des changements du comportement des serveurs MMS. Cela signifie qu'à ce niveau la première partie de la norme MMS [ISO90a] n'est pas respectée. Cependant, la seconde partie [ISO90b] décrivant le protocole reste inchangée.

4. **Extension du protocole** : le protocole MMS est étendu en ce sens que si aucun service n'est ajouté et si aucun paramètre de service n'est modifié, on peut à la fois étendre la gamme de valeur d'un paramètre donné et rajouter des paramètres aux services dans la mesure où ceux-ci sont optionnels. Une extension du protocole implique une extension ou une modification des serveurs MMS sans quoi elle n'aurait pas de sens.

5. **Modification du protocole** : ce niveau représente le degré maximum de modification. Tout changement est autorisé. On peut toutefois distinguer entre les simples ajouts de nouveaux services sans altérer par ailleurs le protocole et les modifications des services décrits dans la norme [ISO90b].

On définit les mêmes niveaux de modification pour les clients MMS en précisant qu'un client est de niveau X si :

- il est conforme au protocole et au comportement d'un serveur de niveau X mais pas de niveau X+1 et $X < 5$;
- il est conforme au protocole et au comportement d'un serveur de niveau X et $X = 5$.

1.2.4 Avantages et inconvénients

Dans le cadre de cette thèse, nous privilégions l'étude de solutions basées sur le premier niveau. En effet, l'avantage du niveau 1 réside dans la portabilité des solutions qui sont proposées. Comme aucune hypothèse n'est faite quant aux applications MMS basées sur ce niveau, celles-ci peuvent fonctionner sur n'importe quel produit MMS. L'inconvénient réside dans la limitation des capacités offertes par MMS.

Pour cette raison, nous montons d'un niveau et proposons également des extensions aux serveurs MMS. Nous verrons qu'en se limitant à ces seuls deux premiers niveaux nous pouvons exploiter la norme MMS de façon suffisante pour résoudre un certain nombre de problèmes. Il est important de constater que toute application client utilisant le protocole MMS suivant la norme peut utiliser un serveur étendu sans qu'aucune modification ne soit nécessaire. Les extensions des serveurs MMS restent donc entièrement compatibles avec les applications MMS standards. Pour une application cliente "classique" (c'est-à-dire de niveau 1), les extensions de niveau 2 sont donc transparentes.

A partir du troisième niveau, il est évident que les applications client exploitant les modifications des serveurs MMS ou du protocole MMS ne sont ni portables sur d'autre produits MMS ni compatibles avec les produits respectant strictement MMS. De même une application client MMS classique n'est en général pas compatible avec des serveurs de niveau supérieur ou égal à 3. Ces différents degrés de compatibilité sont résumés sur les tables 1.1 et 1.2 avec une brève explication.

Dans tous les chapitres de cette thèse nous précisons toujours à quel niveau de modification appartiennent les extensions et modifications à la norme MMS que nous proposons.

1.3 Organisation

Cette thèse s'articule autour de six chapitres principaux. Dans un premier temps nous analysons les outils fondamentaux que sont les sémaphores et événements MMS. Ceci fait l'objet des chapitres 2

Niveau de modif. du serveur	**Compatibilité**		**Commentaires**
	Protocole	Comportement	
(1) Inchangé	oui	oui	-
(2) Serveur étendu	oui	oui	Le comportement classique est inchangé
(3) Serveur modifié	oui	oui/non	Les fonctionnalités modifiées provoqueront un comportement différent de celui attendu. Compatible si ces fonctionnalités ne sont pas utilisées par le client
(4) Protocole étendu	oui	oui/non	Idem
(5) Protocole modifié	oui/non	oui/non	Idem + les services modifiés seront rejetés par la MMPM. Protocole compatible si les services modifiés ne sont pas utilisés

TAB. 1.1 - Compatibilité d'un client classique avec les serveurs modifiés

Niveau de modif. du client	**Compatibilité**		**Commentaires**
	Protocole	Comportement	
(1) Inchangé	oui	oui	-
(2) Client étendu	oui	non	Le comportement attendu ne sera pas celui du serveur
(3) Client modifié	oui	non	Idem
(4) Protocole étendu	non	non	Idem + rejet probable des services étendus par la MMPM du serveur
(5) Protocole modifié	non	non	Idem + rejet certain des services modifiés par la MMPM du serveur

TAB. 1.2 - Compatibilité d'un serveur classique avec les clients modifiés

et 3 respectivement. Notre but est de clairement positionner ces outils par rapport à l'utilisation plus classique et connue des sémaphores et événements qui est faite dans d'autres systèmes. Il nous est apparu important de détailler avec précision les sémaphores et événements MMS pour plusieurs raisons : ce sont des outils très puissants mais relativement peu utilisés dans les applications MMS courantes, il sont souvent à tort considérés comme trop complexes, nous les utilisons abondamment dans cette thèse pour résoudre nombre de problèmes.

Dans le chapitre 4 nous proposons une architecture générale des serveurs MMS. Nous montrons comment concevoir un serveur MMS complet de façon modulaire en prenant en compte l'ensemble des activités concurrentes qui se déroulent dans un serveur. Nous formalisons l'exécution des requêtes de services confirmés sous la forme d'une machine d'états et montrons comment les transitions d'un état à un autre se produisent dans un serveur MMS.

Le chapitre 5 présente une extension à la détection d'événements MMS. Cette extension permet aux applications client de définir avec précision les conditions d'apparition d'un événement. Ces conditions sont représentées sous la forme de prédicats librement définis et modifiables par les clients. Cette extension est tout à fait compatible avec la norme MMS. Nous l'analysons et nous la décrivons en détail puis nous l'intégrons dans l'architecture générale des serveurs MMS décrite au chapitre précédent. Dans ce chapitre nous proposons également une extension qui permet d'associer des priorités aux requêtes de service MMS et d'exécuter ces requêtes en fonction de ces priorités dans les serveurs.

Le chapitre 6 est entièrement consacré à l'étude de techniques d'ordonnancement temps-réel dans MMS. Cette aspect essentiel des applications industrielles n'est pas du tout abordé par la norme

MMS. Nous montrons comment assurer (ou au moins vérifier) le respect de contraintes de temps dans l'exécution des requêtes de service MMS et dans la détection des événements. Nous proposons des solutions pour associer des échéances aux requêtes de service MMS en exploitant les outils offerts par MMS sans modifier la norme.

Dans le dernier chapitre, nous nous plaçons dans un environnement de systèmes répartis, c'est-à-dire que nous faisons interagir plusieurs clients et plusieurs serveurs. Nous proposons des solutions basées sur MMS à des problèmes classiques tels que lecteurs/rédacteurs, philosophes et rendez-vous. Nous montrons comment étendre facilement la norme MMS pour permettre une exécution distante des requêtes de service MMS sur des serveurs secondaires. Nous décrivons également deux algorithmes assurant le contrôle de la concurrence dans l'environnement MMS. Ces algorithmes sont exempts d'interblocages et permettent d'accéder à des ressources partagées situées sur différents serveurs.

L'articulation entre les différents chapitres de cette thèse se décomposent très schématiquement comme le montre la figure 1.3. Une flèche d'un chapitre à un autre signifie que le second exploite les résultats du premier. Les chapitres sur un même niveau horizontal peuvent dans une large mesure être lus indépendamment. Dans le déroulement de cette thèse, nous avons cherché à progresser par niveau d'abstraction et de complexité toujours plus grands en nous plaçant sous des points de vue différents au fur et à mesure de la progression. Tout d'abord nous effectuons une analyse la norme MMS (chapitre 2 et 3). Puis nous nous concentrons sur le fonctionnement interne des serveurs (chapitre 4). Avec les chapitre 5 puis 6 nous étudions plutôt les interactions entre un (ou plusieurs) client(s) et un seul serveur. Le chapitre 7 quant à lui traite directement des interactions entre plusieurs clients et plusieurs serveurs.

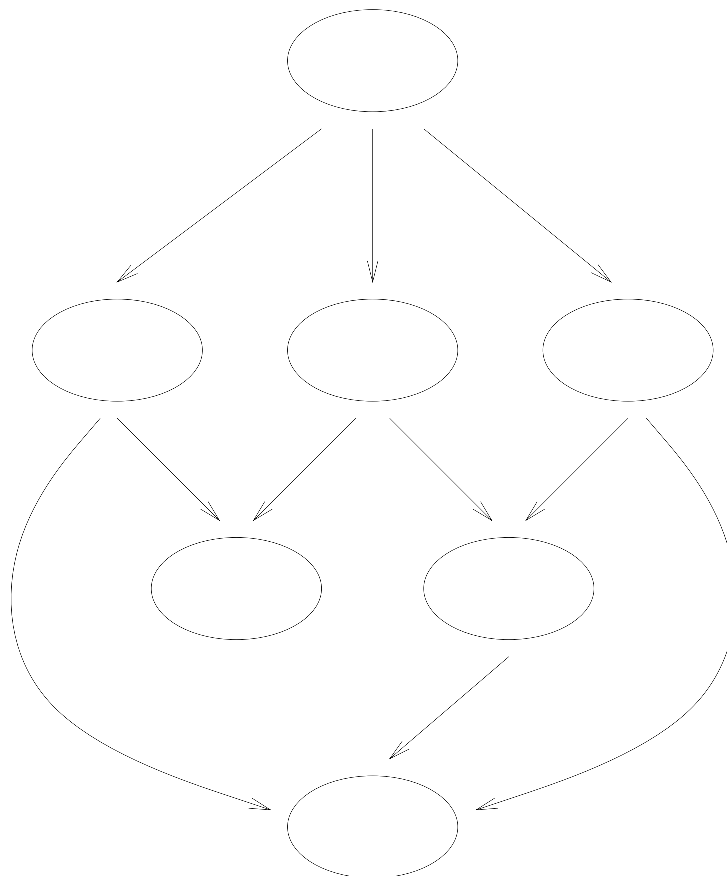


FIG. 1.3 - Schéma de l'articulation entre les chapitres

Dans le but d'avoir une vision globale et claire de l'étude effectuée, nous illustrons sur la figure 1.4 la structuration générale de cette thèse. Tout d'abord nous distinguons trois régions qui correspondent aux trois thèmes porteurs de cette thèse : exploration de la norme en vue d'une meilleure exploitation et, le cas échéant, d'une amélioration. Les différents chapitres sont représentés dans les petits cercles[2]. Chacun des chapitres est positionné dans la région correspondant à son thème dominant (ou sur deux régions quand les thèmes traités sont d'égale importance). Nous faisons apparaître à coté de chaque cercle les principaux sous-thèmes traités dans le chapitre correspondant.

Par ailleurs, ces chapitres gravitent autour d'un thème central : "Sémaphore et Evénements". En effet, de très nombreux résultats contenus dans cette thèse ont un rapport direct avec les sémaphores et les événements MMS, soit qu'ils proviennent d'une utilisation des sémaphores et événements, soit qu'ils représentent une amélioration de la gestion de sémaphores ou d'événements MMS.

Enfin, nous faisons apparaître au bas de la figure notre souci de toujours chercher à préserver l'existant normatif dans l'ensemble des solutions proposées, c'est-à-dire notre démarche consistant à exploiter au mieux les possibilités offertes par la norme MMS et à ne la modifier que le moins possible.

[2] Les chapitres 2 et 3 sont réunis dans le cercle "Analyse".

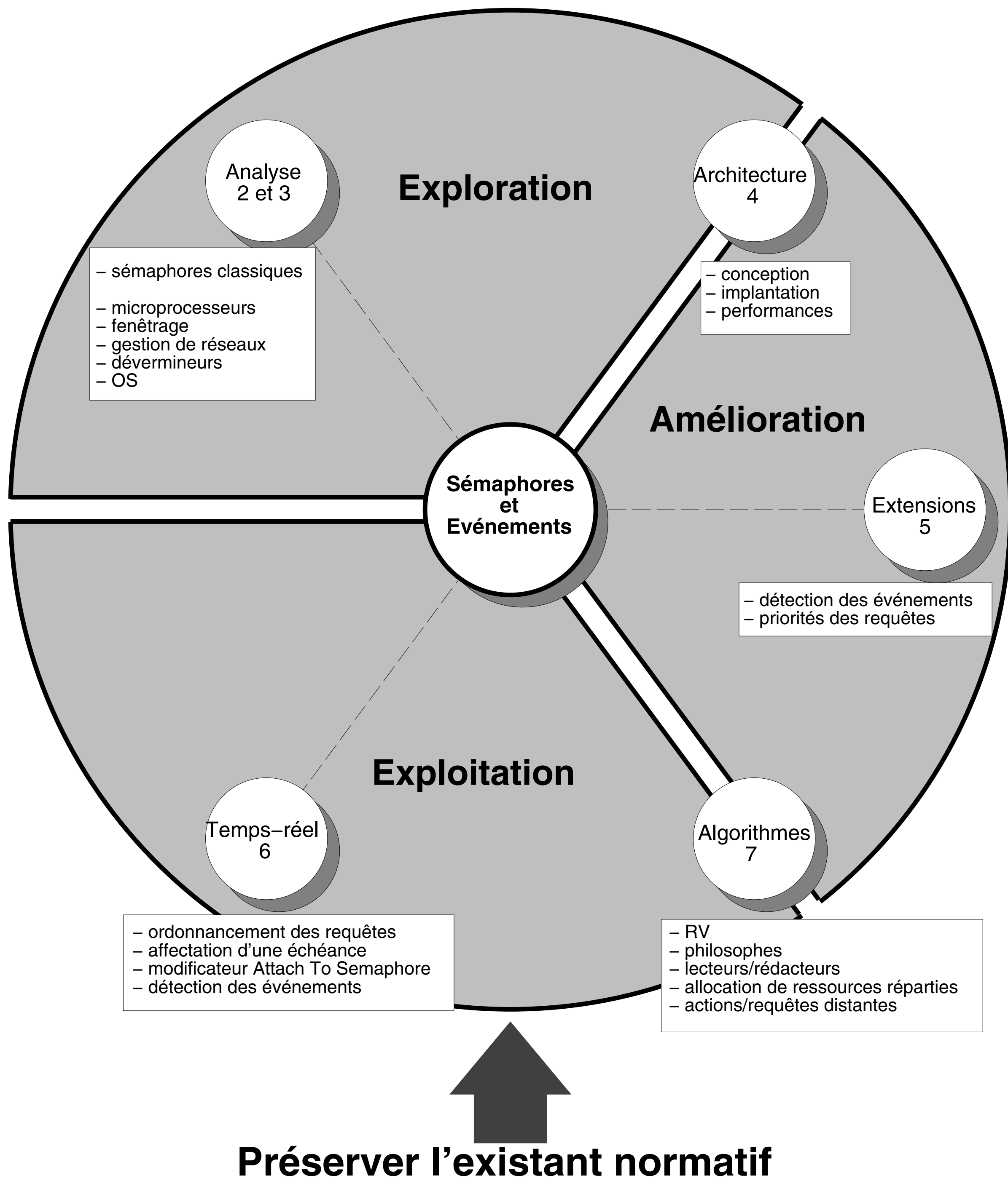


FIG. 1.4 - Structuration générale de la thèse

Chapitre 2

Les sémaphores MMS

2.1 Introduction

Ce chapitre comme le suivant est orienté vers l'analyse et la compréhension de parties fondamentales de la norme MMS. Cette étude doit nous permettre de faire apparaître les limitations de certains concepts de la norme. Nous analysons notamment les problèmes posés par la définition d'un sémaphore MMS. Nous proposons quelques solutions dans le cas où ces problèmes sont simples et ne demandent pas d'analyse détaillée. Le chapitre 7 traite de la résolution de problèmes et limitations plus compliqués qui nécessitent une étude plus approfondie.

La méthode suivie dans ce chapitre ainsi que dans le suivant est de ramener les sémaphores et événements MMS à quelque chose de connu. Cette approche qui n'a semble-t-il pas encore été suivie par d'autres auteurs doit nous permettre de clarifier le concept de sémaphore et d'événement MMS en vue de faciliter leur compréhension et leur utilisation dans les applications MMS. Notre but n'est donc pas d'expliquer de façon exhaustive comment se comportent sémaphores et événements MMS mais d'illustrer leur fonctionnement en les comparant à des concepts connus pour en faire ressortir les avantages et inconvénients et décider de la pertinence de leur définition actuelle.

Le fil directeur de ce chapitre se décompose donc en deux points :

1. compréhension et clarification du concept de sémaphore MMS en le comparant et ramenant à des mécanismes semblables et connus;
2. identification des limitations inhérentes à ce concept.

Le sémaphore dit "classique" est un outil bien connu introduit initialement par Dijkstra [Dij68a] pour permettre la protection de ressources, la synchronisation entre processus, le verrouillage de sections critiques de code, etc. Deux opérations sont applicables à un sémaphore. Sommairement, nous pouvons décrire ces opérations de la façon suivante. L'opération `P()` met en attente le processus qui l'exécute si le sémaphore est occupé. Dans le cas contraire, `P()` décrémente la valeur du sémaphore de un et termine. L'opération complémentaire `V()` incrémente le sémaphore de un et libère un des éventuels processus en attente sur le sémaphore. Nous nous intéressons au sémaphore car nous retrouvons dans MMS ce même mécanisme sous des traits similaires et différents à la fois.

Le sémaphore MMS est également un mécanisme qui autorise l'accès contrôlé à des ressources partagées par différentes applications MMS et, dans une moindre mesure, la synchronisation entre clients MMS. Nous proposons à l'annexe F quelques exemples possibles d'utilisation des sémaphores MMS dans des applications industrielles. L'objectif de ce chapitre est de réaliser une étude en profondeur

du concept de sémaphore MMS. Nous allons comparer les sémaphores MMS avec les sémaphores classiques introduits par Dijkstra [Dij68a] et montrer que de réelles différences existent. La plupart des résultats de cette étude ont également été exposés dans [Cas95e].

Dans la suite, le mot "processus" sera utilisé avec les sémaphores classiques tandis que les mots "application" ou "client" seront réservés pour les sémaphores MMS. Le symbole s identifie un sémaphore ainsi que sa valeur entière courante. Le symbole s_0 est sa valeur initiale.

Dans la littérature, on trouve généralement deux types de sémaphores : un pour lequel s est toujours positif (par exemple [Dij68a], [Hab72], [Hoa78], [Bac86], [Ray93]), un autre pour lequel s peut être négatif (par exemple [Dij68b], [BBF+70], [Sch86], [Dun91]). Entre ces deux types, les opérations connues `P()` et `V()` diffèrent légèrement. Nous ne considérons pas cette différence sauf dans des cas particuliers. Dans ces cas, le symbole "s+" (respectivement "s-") désignera une situation où les sémaphores sont positifs (respectivement négatifs).

2.2 Aperçu des différences et similitudes

2.2.1 Equivalences simples entre sémaphores MMS et sémaphores classiques

Pour manipuler les sémaphores, MMS introduit sept services et un modificateur. Ces services sont utilisés pour créer, détruire, modifier ou consulter un sémaphore. Un *modificateur* est une condition qui peut être ajoutée à tout service MMS confirmé et qui doit être satisfaite avant l'exécution proprement dite du service. Ces modificateurs sont fournis aux services sous forme d'une liste. Les modificateurs d'un service doivent être traités dans l'ordre de cette liste. Le modificateur `Attach To Semaphore` nécessite la prise de contrôle d'un sémaphore. Lorsque l'exécution du service est accomplie, le sémaphore est automatiquement libéré.

La table 2.1 établit une correspondance simple entre les services MMS et les sémaphores classiques. Le service MMS figurant dans la colonne de gauche du tableau est toujours plus complexe que son équivalent dans le domaine des sémaphores classiques situé dans la colonne de droite. Ce tableau vise uniquement à illustrer les principales fonctions des services de gestion de sémaphores MMS au moyen des caractéristiques bien connues des sémaphores classiques. Son but est de permettre une compréhension rapide et claire de la gestion des sémaphores MMS pour pouvoir aborder la suite de cette analyse.

Sémaphores MMS	Sémaphores classiques
Take Control	P()
Take Control avec réquisition	Pas d'équivalent
Relinquish Control	V()
Define Semaphore	Déclaration du sémaphore
Delete Semaphore	Le sémaphore est détruit (par exemple, la procédure où il est déclaré se termine)
Report Semaphore Status	Représenterait l'obtention de la valeur du sémaphore si ceci était possible avec un sémaphore classique
Report Pool Semaphore Status	
Report Semaphore Entry Status	Pas d'équivalent
Attach To Semaphore	Pas d'équivalent

TAB. 2.1 - Correspondance entre sémaphores MMS et sémaphores classiques

2.2.2 Différences structurelles

La différence la plus immédiate entre sémaphore classique et sémaphore MMS réside dans leur structure. Un sémaphore classique est un entier auquel on associe généralement une file d'attente [Sch86], [Nus86], [SP88]. Il peut donc être vu comme une structure de données à deux champs. Par commodité on dit que le sémaphore a la valeur x pour signifier que son champs entier a cette valeur x. Le sémaphore MMS est également une structure de données désignée sous le terme d'"objet" selon la terminologie MMS. Le sémaphore MMS est toutefois plus complexe que le sémaphore classique. Cet objet contient dix attributs de types différents (fig. 2.1).

```
Object: Semaphore

  Key attribute: Semaphore Name
  Attribute: MMS Deletable (TRUE, FALSE)
  Attribute: Class (TOKEN, POOL)
  Constraint: Class = TOKEN
        Attribute: Number Of Tokens
        Attribute: Number Of Owned Tokens
  Constraint: Class = POOL
        Attribute: List Of Named Tokens
        Attribute: List Of Named Token States
  Attribute: List Of Owners
  Attribute: List Of Requesters
  Attribute: Event Condition Reference
```

```
Object: Semaphore Entry

  Key attribute: Entry ID
  Attribute: Entry Class (SIMPLE, MODIFIER)
  Attribute: Semaphore Reference
  Attribute: Requester Application Reference
  Attribute: Application Association Local Tag
  Attribute: Invoke ID
  Attribute: Named Token
  Attribute: Priority
  Attribute: Remaining Acquisition Delay
  Attribute: Remaining Control Time Out
  Attribute: Abort On Time Out (TRUE, FALSE)
  Attribute: Relinquish If Connection Lost (TRUE, FALSE)
  Attribute: Entry State (QUEUED, OWNER, HUNG)
```

FIG. 2.1 - Les objets MMS Semaphore et Semaphore Entry

Les attributs `MMS Deletable` et `Event Condition Reference` sont spécifiques à MMS. Le premier précise si un sémaphore peut être détruit avec le service `Delete Semaphore`. Le second est expliqué à la section 2.3.4.3.

MMS définit deux classes de sémaphores : le *sémaphore banalisé* ("token semaphore") et le *sémaphore étiqueté* ("pool semaphore") selon la valeur de l'attribut `Class`. Ces deux classes de sémaphores ne sont pas fondamentalement différentes. La discussion qui suit est toujours valable pour ces deux classes. L'étude des différences entre ces classes et des motivations qui ont conduit à leur création fait l'objet de la section 2.4[1].

Les sémaphores MMS **contiennent des jetons** ("tokens"). La quantité de jetons libres représentent le nombre entier d'un sémaphore classique (s+). Un sémaphore MMS se voit affecté un nombre initial de jetons (`Number Of Tokens` ou `List Of Named Tokens`). Quand une application effectue une requête `Take Control` sur un sémaphore, un jeton libre de ce sémaphore lui est alloué. Le nombre d'applications pouvant prendre le contrôle du sémaphore est donc limité par le nombre initial de jeton de ce sémaphore. On dit qu'un sémaphore est *pris* quand tous ses jetons sont alloués. On dit qu'il est *libre* quand au moins un jeton est libre. Un sémaphore MMS ne contient pas d'attribut `Number Of Free Tokens` qui pourrait représenter la valeur courante s des sémaphores classiques (s+). La valeur courante d'un sémaphore MMS équivaut à "`Number Of Tokens - Number Of Owned Tokens`" (ou nombre d'états dans `List Of Named Token States` moins nombre d'états ayant la valeur `OWNED` dans `List Of Named Token States`).

[1] Hors de la section 2.4, chaque fois que nous parlons d'une requête `Take Control`, nous ne considérons pas le cas où cette requête contiendrait le paramètre `Named Token` applicable uniquement aux sémaphores étiquetés.

On retrouve dans le sémaphore MMS l'équivalent de la file d'attente associée au sémaphore classique. Dans MMS, cette file est explicitement définie et fait partie du sémaphore sous la forme de l'attribut `List Of Requesters`. Dans cette file d'attente, une application est identifiée par sa *rubrique de sémaphore* ("Semaphore Entry" ou SE). Un SE est un objet normalisé par MMS qui contient les informations nécessaires pour identifier (attributs 3, 4 et 5 fig. 2.1) et organiser les applications en attente. Avec les sémaphores classiques, ces objets existent implicitement et sont gérés par les composants du système qui sont responsables de la sélection des processus en attente en vue de leur exécution. On peut par exemple les comparer à des descripteurs de processus dans les systèmes d'exploitation. Dans MMS, l'ordonnancement des applications bloquées et la maintenance de l'information nécessaire pour cet ordonnancement font partie intégrante de la politique de gestion des sémaphores.

Quand un jeton est libéré et s'il y a des applications en attente, l'une d'entre elles est sélectionnée et acquiert le jeton. L'objet SE la représentant migre de la file `List Of Requesters` à la file `List Of Owners` et son attribut `Entry State` passe de la valeur `QUEUED` à la valeur `OWNER`.

L'attribut `List Of Owners` n'a pas d'équivalent dans les sémaphores classiques. Sa présence montre que MMS insiste sur la notion de *possession* d'un jeton par une application. Il est donc possible de distinguer les applications qui ont pris le jeton d'un sémaphore des autres. Un sémaphore classique, étant un nombre entier, ne présente pas cette notion de possession. Cet entier est simplement incrémenté ou décrémenté et ne peut fournir aucune information autre que le nombre de processus qui peuvent encore exécuter l'opération `P()` (s+) ou le nombre de processus qui sont bloqués dans l'exécution d'un `P()` (s-). Les processus qui viennent de compléter l'exécution d'une opération `P()` sont anonymes et n'ont pas obligatoirement besoin d'effectuer l'opération complémentaire `V()` plus tard. La notion de possession des jetons MMS imposent normalement aux applications MMS ayant effectué n `Take Control` sur un sémaphore l'exécution de n `Relinquish Control` sur ce même sémaphore.

L'attribut `Number Of Owned Tokens` (ou le nombre d'états de valeur `OWNED` dans `List Of Named Tokens`) est le nombre d'objets SE dans la file `List Of Owners`. Ces attributs n'ont pas d'équivalents avec les sémaphores classiques. Ils peuvent être déduits facilement à partir de la valeur du sémaphore classique comme le montre la table 2.2. C'est parce que les sémaphores MMS sont *acquis* ("owned") par opposition à *franchis* ("passed") qu'il n'est pas possible d'avoir `Number Of Owned Tokens` > `Number Of Tokens`. Il appartient aux mécanismes de gestion des sémaphores MMS d'éviter une telle situation. D'un autre coté, rien n'interdit d'incrémenter le sémaphore classique au delà de sa valeur initiale [Dij71]. Ce sont les processus utilisant le sémaphore qui doivent éviter d'avoir $s > s_0$ dans les situations ne permettant pas cet état (par exemple l'exclusion mutuelle).

Nous avons donc constaté que les sémaphores MMS et classiques présentent d'évidentes caractéristiques communes mais également des différences. La première différence réside dans leur structure plus complexe pour un sémaphore MMS que pour un sémaphore classique. Une autre différence vient de la notion d'appartenance du jeton et donc du sémaphore. On peut la retrouver dans les primitives de certains systèmes d'exploitation pour gérer l'exclusion mutuelle. Mais cette notion est absente des sémaphores classiques. La table 2.2 résume les résultats de cette section.

Mais ce qui est intéressant pour les programmeurs d'applications c'est la vision externe du comportement des sémaphores, c'est-à-dire comment ceux-ci sont manipulés. Les sections suivantes étudient les sémaphores sous ce point de vue.

Sémaphores MMS	Sémaphores classiques
Semaphore Name	Nom du sémaphore
MMS Deletable	Pas d'équivalent (spécifique à MMS)
Class	Pas d'équivalent
Number Of Tokens	Valeur initiale du sémaphore
Number Of Owned Tokens	Valeur initiale - Valeur courante (s+) Valeur initiale du sémaphore (s-)
List Of Named Tokens	Pas d'équivalent. Le nombre d'éléments dans la liste correspond à la valeur initiale du sémaphore
List Of Named Token States	Pas d'équivalent. Le nombre d'états ayant la valeur `FREE` correspond à la valeur courante du sémaphore
List Of Owners	Pas d'équivalent
List Of Requesters	Queue des processus en attente sur le sémaphore
Event Condition Reference	Pas d'équivalent (spécifique à MMS)
Number Of Tokens - Number Of Owned Tokens	Valeur courante du sémaphore (s+)

TAB. 2.2 - Correspondance entre les attributs des sémaphores MMS et des sémaphores classiques

2.3 Comparaison entre les sémaphores MMS et les sémaphores classiques

2.3.1 Les points communs

Quelques ressemblances immédiates entre sémaphores MMS et sémaphores classiques ont déjà été évoquées dans la section précédente. Les deux types de sémaphores sont utilisés pour la synchronisation et l'accès contrôlé à des ressources partagées. A première vue ils se comportent de la même façon. Ils ont tous deux une file d'attente qui leur est associée. Celle-ci liste les applications ou les processus en attente de la libération du sémaphore. Dans de nombreuses situations, un sémaphore MMS sera utilisé comme un sémaphore classique.

Les services `Take Control` et `Relinquish Control` sont en grande partie identiques aux opérations `P()` et `V()`. Du point de vue d'un client MMS, une requête `Take Control` dans sa forme la plus simple (c'est-à-dire sans paramètres optionnels) paraît procéder de la même façon que `P()`. Elle vérifie s'il reste un jeton libre et si oui, elle le prend. Sinon, elle crée un SE et le place en attente dans la liste `List Of Requesters`. La réponse du `Take Control` n'est pas envoyée au client tant qu'un jeton ne lui est pas alloué. Un client en attente ne peut donc poursuivre tant qu'il n'a pas de réponse tout comme un processus reste bloqué sur une opération `P()` tant que $s = 0$ (s+) ou $s < 0$ (s-).

En fait, ce qui se passe vraiment est légèrement différent. Le SE est toujours créé et ce dès l'exécution du `Take Control`. Il est mis dans la liste `List Of Requesters`. La VMD est alors responsable de la vérification régulière de la disponibilité de jetons libres du sémaphore et de la migration du SE dans la file `List Of Owners`. Une fois que le SE est mis dans la liste `List Of Owners`, le service `Take Control` peut se terminer et la réponse est envoyée au client correspondant. De même, une requête `Relinquish Control` se contente de retirer un SE de la liste `List Of Owners` et de le détruire. La VMD est responsable du réveil d'un des `Take Control` éventuellement en attente sur le sémaphore dont un jeton a été libéré. Dans les deux cas, l'attribut `Number Of Owned Tokens` (ou `List Of Named Token State`) doit aussi être mis à jour. Cependant, tout ce comportement est caché aux observateurs externes.

Avec les sémaphores généraux ($s_0 > 1$), le même processus ou des processus différents peuvent exécuter plusieurs fois sans être bloqués l'opération `P()`. De même, une ou plusieurs applications MMS peuvent prendre le contrôle de plusieurs jetons d'un sémaphore donc accomplir plusieurs requêtes de

service `Take Control`. Le bon fonctionnement des sémaphores classiques nécessite que `P()` et `V()` soient exécutées de façon atomique[2]. Cette atomicité doit aussi apparaître dans MMS pour les services `Take Control` et `Relinquish Control` et c'est à la VMD de l'assurer.

La relation connue et appelée *invariant du sémaphore* [Hab72], [Mar81] doit aussi s'appliquer aux sémaphores MMS. Notons #X le nombre de fois où l'opération X est appelée. On a pour les sémaphores classiques $s = s_0 + \#\mathtt{V}(s) - \#\mathtt{P}(s)$ que s soit positif ou négatif. Dans MMS, ceci est exprimé par les deux équations :

$$\mathtt{NumberOfOwnedTokens} + \#\mathtt{RelinquishControl} = \#\mathtt{TakeControl} \tag{2.1}$$

$$taille(\mathtt{ListOfRequesters}) + \mathtt{NumberOfTokens} + \#\mathtt{RelinquishControl} = \#\mathtt{TakeControl} \tag{2.2}$$

L'équation 2.1 s'applique dans le cas où il n'y a pas d'applications en attente d'un jeton. L'équation 2.2 s'applique dans le cas contraire. Les deux équations s'appliquent et se rejoignent dans le cas où aucune application n'est en attente d'un jeton et que tous les jetons sont pris. On a alors $taille(\mathtt{ListOfRequesters}) = 0$ et $\mathtt{NumberOfOwnedTokens} = \mathtt{NumberOfTokens}$.

Nous verrons ultérieurement que l'utilisation de certaines options MMS peuvent rendre invalide ces relations. Dans de tels cas, nous donnerons des relations plus adéquates pour MMS. A la section 2.3.5 nous proposons et démontrons l'invariant complet du sémaphore MMS.

Les similitudes entre sémaphores MMS et sémaphores classiques sont donc limitées aux cas d'une utilisation simple des services MMS. La figure 2.2 illustre les différents états que peut prendre un objet SE tel que décrit dans la norme [ISO90a]. Elle résume le comportement des sémaphores MMS et montre aussi en traits gras comment s'y intègre celui des sémaphores classiques. Les parties qui ne sont pas en gras sont étudiées dans les sections suivantes. En particulier, la définition de l'état `HUNG` est donnée à la section 2.3.2.3.

2.3.2 Que peut-on faire avec les sémaphores MMS que l'on ne peut pas faire avec les sémaphores classiques ?

Dans MMS, il existe de nombreuses options pour contrôler le comportement d'un sémaphore. Ces diverses options offertes aux applications client laissent entendre que le sémaphore MMS, plus qu'un simple sémaphore classique, est plutôt un mécanisme complexe de protection/synchronisation. Cette section décrit les caractéristiques des sémaphores MMS qui ne font pas partie de la définition d'un sémaphore classique mais que l'on peut souvent rencontrer dans des systèmes présentant des mécanismes de protection/synchronisation évolués. Toutes les caractéristiques propres à MMS seulement sont présentées dans la section 2.3.4.

2.3.2.1 L'ordonnancement des exécutions des requêtes `Take Control`

Quand les sémaphores furent définis pour la première fois, il n'était fait aucune allusion à la politique de réveil des processus bloqués sur un sémaphore. Une politique de base équitable était supposée et la solution FIFO fut introduite comme exemple sans pour autant être obligatoire [Dij68a], [Dij68b], [Dij71], [CHP71], [Pre75], [And79]. Plus tard d'autres mécanismes furent proposés (par exemple [Dun91]).

Dans MMS, la politique de réveil FIFO est requise si aucune priorité n'est fournie aux requêtes

[2] On entend par là que ces opérations constituent elles-mêmes une section critique.

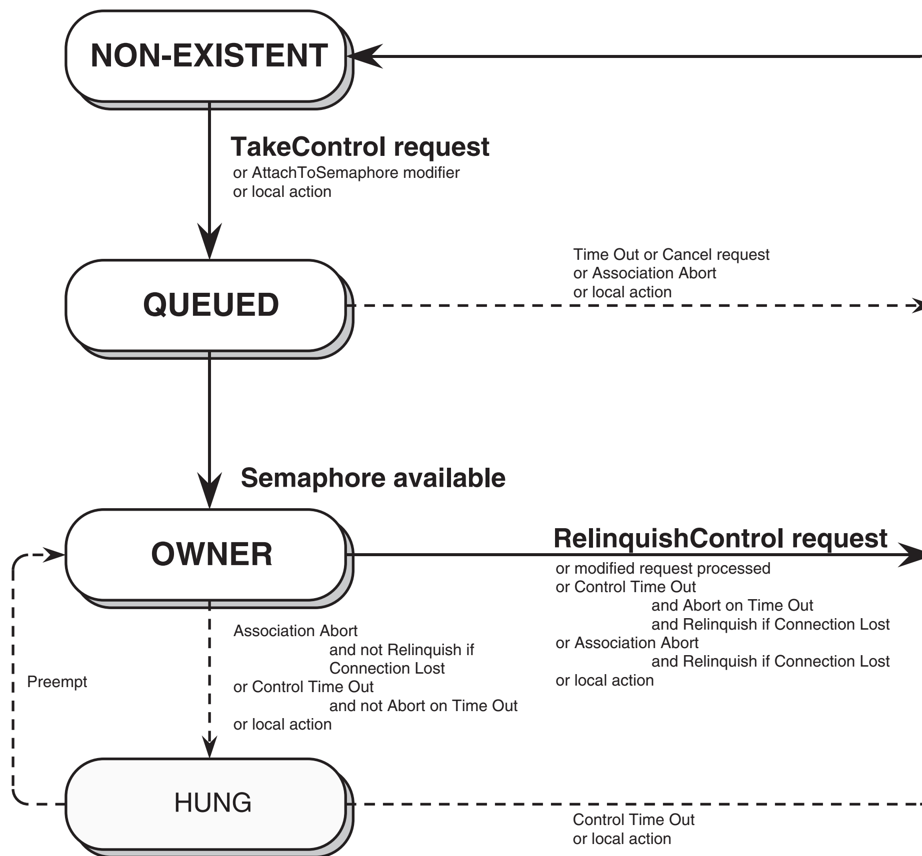


FIG. 2.2 - Le modèle des objets SE

`Take Control`. En ce sens, le sémaphore MMS est comparable au sémaphore dit *blocked-queue* de Stark [Sta82]. Il est ainsi toujours possible de savoir quelle sera l'application suivante qui pourra devenir propriétaire du premier jeton libéré. Ici, l'ordre FIFO signifie le même ordre que celui de l'exécution dans le serveur de la requête `Take Control` (et du modificateur `Attach To Semaphore`). Il doit être clair que cet ordre n'est pas l'ordre d'envoi des requêtes des différents clients et n'est pas non plus l'ordre de réception de ces requêtes par le serveur. Avec MMS, une requête de service 1 peut être exécutée après une requête de service 2 alors que 1 est arrivée au serveur avant 2. Ce comportement surprenant est parfaitement autorisé par la norme MMS. Il peut rendre difficile le développement d'applications utilisant MMS surtout dans un environnement réparti où l'ordre des messages est souvent d'une importance capitale.

Nous ne connaissons pas d'implantation réelle de serveurs MMS qui présente des cas où des requêtes de service sont exécutées dans un ordre différent de celui de leur réception. Mais le problème est que ce comportement est accepté par la norme. Par conséquent, aucune supposition ne peut être faite sur l'ordre d'exécution d'un service dans un serveur donné.

La seule garantie disponible est la suivante : les requêtes de service `Take Control`[3] effectuées sur un sémaphore occupé sont mises en attente dans l'ordre de leur exécution. Lorsqu'un `Relinquish Control` est exécuté, le jeton qui se libère est attribué à la requête `Take Control` la plus ancienne selon la politique FIFO. Dans ce cas, les possibilités de famine éventuelle d'applications bloquées sont évitées [Sta82].

Cependant, il est possible de fournir une priorité à une requête `Take Control`. Les priorités vont

[3] En fait, les SEs créés suite à ces requêtes.

de 0 (la plus haute) à 127. Le paramètre `Priority` du service est copié dans l'attribut de même nom du SE correspondant. Le SE est alors inséré dans la liste d'attente dans la position donnée par sa priorité. MMS affecte par défaut la priorité 64 aux requêtes qui n'utilisent pas le paramètre `Priority`. La gestion de priorité est un mécanisme connu que l'on retrouve dans la plupart des systèmes d'exploitation multi-tâches. Le système d'exploitation temps-réel VRTX fournit les appels `SC_PEND` et `SC_POST` sur les boîtes aux lettres du système [Rea86]. Si plusieurs processus sont en attente sur un `SC_PEND`, alors le premier message reçu par `SC_POST` est automatiquement attribué au processus le plus prioritaire. Mais en général, les priorités ne sont pas directement fournies dans les primitives comme c'est le cas pour `Take Control`. En effet, l'ordonnanceur connaît les priorités des processus et est responsable du réveil du processus prioritaire.

L'utilisation de priorités MMS peut entraîner la famine d'applications en attente sur un sémaphore avec une priorité basse. En effet, s'il existe une requête en attente sur un sémaphore avec une priorité p et si la file d'attente du sémaphore reçoit continuellement des requêtes dont la priorité est supérieure à p, alors la requête de priorité p ne sera jamais servie. Il n'y a aucune façon d'éviter la famine dans MMS lorsqu'on utilise les priorités dans les requêtes `Take Control` (ou dans le modificateur `Attach To Semaphore`).

2.3.2.2 Interrompre `P()`

Le paramètre `Acceptable Delay` d'une requête `Take Control` (`Remaining Acquisition Delay` du SE) informe la VMD que le client qui a effectué la requête ne désire pas attendre plus de `Acceptable Delay` millisecondes qu'un jeton soit libéré. A expiration de ce délai, le SE de l'application est retiré de la liste `List Of Requesters` du sémaphore et une réponse négative au `Take Control` est envoyée. L'opération `P()` ne présente pas une telle possibilité. Cependant, certains auteurs ont proposé d'associer un temps de garde à l'opération `P()`. Le noyau NUCLEUS décrit dans [Nus86] pp. 140-153 propose un tel mécanisme.

On retrouve aussi cette possibilité dans certains systèmes de communication entre processus. Le noyau Spring [SR87] par exemple fournit la primitive `RECVW` qui n'attend pas plus d'un temps donné la réception d'un message dans une boîte aux lettres. Avec cette primitive et sa duale `SEND` on implante facilement des sémaphores avec temporisation comme ceux de MMS. L'appel VRTX `SC_PEND` [Rea86] et les mécanismes d'exclusion mutuelle de pSOS$^+$ [Tho90] ont des capacités similaires.

D'autres systèmes proposent des primitives qui évitent à un processus d'attendre sur `P()` si le sémaphore est déjà pris (par exemple `semop` avec l'option `IPC_NO_WAIT` dans UNIX System V [Bac86] et `pthread_mutex_trylock` dans DCE [Fou93]). Ceci correspondrait à une requête `Take Control` dont l'`Acceptable Delay` est nul ou si petit qu'il n'attend jamais sur un sémaphore déjà pris et termine tout de suite. Dans MMS, on peut aussi savoir si un sémaphore est pris en utilisant le service `Report (Pool) Semaphore Status`. Mais ce dernier ne prend pas possession du sémaphore s'il le trouve libre.

On peut aussi interrompre un `Take Control` en cours au moyen du service `Cancel`. Encore une fois, il n'y a pas d'équivalent avec les sémaphores classiques car une opération `P()` ne peut être interrompue. L'une des raisons est que l'on ne peut pas identifier le processus qui doit être retiré de la file d'attente du sémaphore. On peut toujours choisir un processus de façon arbitraire mais ceci ne correspond pas à MMS. Si l'on suppose que l'identité n'a pas d'importance, certains auteurs préconisent d'utiliser l'opération `V()` pour interrompre un `P()`. Dans MMS, le service `Relinquish Control` ne peut pas être utilisé pour interrompre un `Take Control` de la même application en attente. Si cela est tenté, le `Relinquish Control` échoue et la requête `Take Control` reste en attente.

Ceci est une différence importante : dans MMS, une requête de prise de contrôle d'un sémaphore peut échouer ou peut être annulée. Avec les sémaphores classiques, `P()` est toujours exécutée.

Les échecs ou les interruptions des `Take Control` modifient les relations 2.1 et 2.2 de la façon suivante :

$$\texttt{NumberOfOwnedTokens} + \#\texttt{RelinquishControl} \leq \#\texttt{TakeControl} \quad (2.3)$$

$$taille(\texttt{ListOfRequesters}) + \texttt{NumberOfTokens} + \#\texttt{RelinquishControl} \leq \#\texttt{TakeControl} \quad (2.4)$$

En effet, lors d'une interruption de `Take Control`, il y a au moins une application en attente d'un jeton : celle dont le `Take Control` est interrompu. C'est donc la relation 2.2 qui s'applique. L'interruption d'un `Take Control` implique la diminution de `List Of Requesters` de 1, d'où la relation 2.4.

Quand la relation 2.1 redevient applicable, l'annulation du `Take Control` aura augmenté la quantité `#TakeControl` de 1, d'où la relation 2.2.

2.3.2.3 Contrôle du temps de verrouillage d'un sémaphore

Le temps de verrouillage d'un sémaphore peut être contrôlé dans la mesure où le paramètre optionnel `Control Timeout` est utilisé dans une requête `Take Control`. Ce paramètre est copié vers le champs `Remaining Control Timeout` du SE et décrémenté périodiquement à partir du moment où la requête `Take Control` prend effectivement possession d'un jeton. Ce paramètre correspond à un délai en millisecondes après lequel un jeton acquis est automatiquement libéré. Dans certains cas, le jeton est quand même conservé mais le SE passe à l'état `HUNG`[4]. La différence entre ces deux cas se fait en fonction des paramètres `Abort On Timeout` et `Relinquish If Connection Lost` (voir la section 2.3.4.1). Là non plus il n'y a pas d'équivalent dans les sémaphores classiques. L'option `Control Timeout` peut être utilisée pour résoudre les interblocages de manière évidente en cassant automatiquement le cycle de dépendance quand une application client attend depuis trop longtemps sur un sémaphore pris.

L'option `Control Timeout` peut également transformer les relations 2.1 et 2.2 en 2.3 et 2.4 respectivement.

La relation 2.1 devient 2.3 par diminution de `NumberOfOwnedTokens` lors de la fin de temporisation. S'il y a des applications en attente d'un jeton la relation 2.2 devient 2.4 car le jeton libéré est alloué à une de ces applications ce qui provoque une diminution de 1 de *taille*(`ListOfRequesters`).

En l'état actuel de la norme MMS, l'utilisation du paramètre `Control Timeout` pose un problème relatif au choix de l'objet SE à détruire lors d'un `Relinquish Control`. En effet, si une application client effectue plusieurs requêtes `Take Control` sur une même association et avec le même sémaphore, il est dit dans [ISO90a] que les SEs créés ne sont pas différenciables. Lors de l'exécution d'un `Relinquish Control` sur cette association, il est alors théoriquement possible de détruire n'importe lequel de ces SEs, ceux-ci étant identiques. En fait, comme le souligne Pirazzi dans [Pir95], ceci n'est vrai que dans le cas où aucun temporisateur `Remaining Control Timeout` n'est en cours. Sinon, les SEs sont bien différenciables et il est important de savoir lequel détruire si l'on veut garder la cohérence des temporisateurs. La norme MMS actuelle ne permet pas de faire cette distinction. Ainsi, un client peut être amené à supposer qu'il lui reste encore un certain temps de contrôle d'un sémaphore alors qu'en fait la VMD est sur le point de le libérer.

[4] Un jeton dans l'état `HUNG` a perdu son propriétaire sans toutefois être libéré.

2.3.2.4 Obtention d'informations sur l'état d'un sémaphore

Avec les sémaphores classiques les seules informations immédiatement disponibles sont fournies par la valeur entière s[5]. Elle représente le nombre de fois où `P()` peut être accomplie si la file d'attente du sémaphore est vide (s+). $|s|$ représente le nombre de processus bloqués si la file d'attente du sémaphore n'est pas vide (s-).

De même, avec MMS le nombre de jetons occupés d'un sémaphore peut être obtenu via le service `Report Semaphore Status`. Ce service fournit aussi le nombre initial de jetons c'est-à-dire le nombre total de jetons du sémaphore. Les autres paramètres retournés dans ce service sont spécifiques à MMS. Avec les sémaphores classiques, s_0 n'est pas disponible mais peut se calculer facilement à l'aide de l'invariant du sémaphore. Dans UNIX System V, les options `GETVAL`, `GETNCNT` de l'appel système `semctl` sont utilisées dans un but identique [Bac86].

Le service `Report Semaphore Entry Status` fournit toutes les informations concernant les applications en attente d'un jeton ou de celles qui détiennent un jeton (identité, priorité, temporisateurs, etc). Les sémaphores classiques n'ont pas d'équivalent mais de nombreux systèmes ont étendu le concept de sémaphore afin que les informations concernant les processus puissent être obtenues. UNIX System V avec l'option `GETPID` de l'appel système `semctl` en est un exemple [Bac86]. Cette option permet de retourner le numéro d'identification du dernier processus ayant effectué une opération sur le sémaphore considéré.

2.3.2.5 `Take Control` non bloquant

Un client MMS peut travailler en mode asynchrone (ou non bloquant). Ce mode défini par MMSI dans [Gen88] Vol. IV signifie qu'un client n'a pas besoin d'être bloqué en attente de la réponse d'une requête effectuée et qu'il peut continuer à exécuter un travail utile. Ceci s'applique aux requêtes `Take Control`. Donc un client peut toujours s'exécuter même s'il n'a pas reçu la réponse du `Take Control` c'est-à-dire qu'il n'est pas bloqué sur un sémaphore même si celui-ci est pris. Il n'a simplement pas accès au sémaphore. Il va de soi que dans l'exécution qui se poursuit, il ne doit pas y avoir de tentative d'accès aux éventuelles ressources protégées par le sémaphore. Ceci n'est pas garanti par MMS. Cette façon de procéder est identique aux appels système `$ENQW` et `$ENQ` respectivement bloquant et non bloquant de la gestion de sémaphores VAX/VMS [Dig82]. Le noyau temps-réel Sceptre contient des mécanismes d'exclusion mutuelle similaires qui autorisent les processus en attente de l'allocation de ressources partagées à être toujours actifs [BDD+84]. Par contre, un processus qui exécute `P()` est toujours bloqué jusqu'à la libération du sémaphore.

2.3.3 Que peut-on faire avec les sémaphores classiques que l'on ne peut pas faire avec les sémaphores MMS ?

2.3.3.1 Le problème de la possession du jeton

L'opération `V(`s`)` ne fait qu'incrémenter la valeur de s par un et réveille un éventuel processus en attente sur le sémaphore. Cette opération peut être exécutée par n'importe quel processus et quelque soit la valeur courante de s, même si c'est la valeur initiale s_0 [Mar81]. Il n'y a pas de trace dans le sémaphore des processus ayant exécuté une opération `P()` mais qui n'ont pas encore appelé `V()`. Ceux-ci ne sont d'ailleurs pas obligés d'appeler `V()`.

[5]Toutefois au sens strict de la définition d'un sémaphore, cette valeur n'est pas lisible par les processus. Seules les primitives `P()` et `V()` opèrent sur un sémaphore classique.

Ceci constitue une différence fondamentale par rapport aux sémaphores MMS. Seule une application A détenant un jeton d'un sémaphore s peut effectuer un `Relinquish Control` sur s pour libérer ce jeton. Il est précisé dans [ISO90a] que le service échoue si A ne détient pas de jeton de s. Il est donc obligatoire que A ait d'abord complété un `Take Control` (c'est-à-dire ait franchi le sémaphore) avant de pouvoir demander un `Relinquish Control`. `P()` et `V()` sont des opérations indépendantes c'est-à-dire que leur association n'est pas requise [Coo91]. Par contre, une requête `Take Control` n'est pas sensée être utilisée sans une requête `Relinquish Control`. De plus, le `Relinquish Control` doit être effectué sur la même association que le `Take Control` précédent.

Par ailleurs, le nombre initial de jetons d'un sémaphore MMS ne peut être augmenté en aucune façon. Les jetons sont affectés à un sémaphore MMS quand il est défini. Aucun nouveau jeton ne peut être créé pendant la durée de vie du sémaphore MMS. On peut par conséquent ajouter l'inégalité suivante à l'invariant du sémaphore : $\#\texttt{TakeControl} \geq \#\texttt{RelinquishControl}$.

Considérons le scénario suivant. Supposons que deux sémaphores classiques s_1 et s_2 soient initialisés à zéro. Il est bien connu que deux processus p_1 et p_2 peuvent effectuer un rendez-vous si p_1 exécute séquentiellement $\texttt{V}(s_1)$, $\texttt{P}(s_2)$ et p_2 $\texttt{V}(s_2)$, $\texttt{P}(s_1)$. Si on applique ceci à MMS, les deux applications recevront une erreur pour leur requête `Relinquish Control` car p_1 (resp. p_2) ne détient aucun jeton de s_1 (resp. s_2). Puis, si aucune action correctrice n'est entreprise, les deux applications attendront indéfiniment l'allocation de jetons non existant. Ceci n'est pas un interblocage – puisqu'aucune application n'est en attente d'un jeton détenu par l'autre – mais une utilisation naïve des services MMS (sans parler du fait que créer un sémaphore MMS avec un nombre initial de jetons nul n'a pas de sens).

La solution dans MMS est simple. D'abord les sémaphores MMS s_1 et s_2 doivent chacun avoir un jeton initial (pas zéro). Puis p_1 (resp. p_2) doit effectuer une requête `Take Control` sur le sémaphores s_1 (resp. s_2) **avant** le rendez-vous. Il s'agit là d'une phase d'initialisation indispensable qui doit être effectuée avant le lancement proprement dit des deux applications. A partir de là, on se retrouve dans la même configuration que précédemment et le rendez-vous peut se dérouler avec le même scénario. L'application p_1 effectue les requêtes $\texttt{Relinquish Control}(s_1)$ et $\texttt{Take Control}(s_2)$. L'application p_2 effectue la séquence complémentaire (fig. 2.3). Ceci illustre une méthode facile pour synchroniser deux applications client par rendez-vous dans l'environnement MMS. Notons toutefois que la phase d'initialisation peut ne pas être réalisable. Dans ce cas, cette solution ne peut pas marcher. Nous étudions d'autres solutions de rendez-vous au chapitre 7.

Application p_1	Application p_2	
$\texttt{TakeControl}(s_1)$	$\texttt{TakeControl}(s_2)$	*Phase d'initialisation*
.	.	
.	.	
$\texttt{RelinquishControl}(s_1)$	$\texttt{RelinquishControl}(s_2)$	
$\texttt{TakeControl}(s_2)$	$\texttt{TakeControl}(s_1)$	*Rendez-vous*
.	.	
.	.	

FIG. 2.3 - Rendez-vous avec les sémaphores MMS

On voit donc que dans MMS, la protection de ressources partagées est favorisée par rapport à la signalisation. L'association obligatoire `Take Control/Relinquish Control` annule les possibilités de synchronisation par signalisation. Ces deux services peuvent se comparer aux primitives connues

`lock/unlock` par opposition à `wait/signal`. Les premières sont utilisées pour l'exclusion mutuelle alors que les secondes le sont pour la signalisation. Les problèmes pour lesquels on utilise l'une ou l'autre de ces paires de primitives sont souvent différents [LE93]. Les opérations `P()` et `V()` implantent ces deux paires de primitives mais pas les services MMS.

Les solutions connues des problèmes classiques tels que les producteurs/consommateurs [Dij71], [Pre75], [Shr76], [And79], la coordination OU [Pre75] ou les lecteurs/rédacteurs [CHP71], [Sch86] ne peuvent donc pas s'appliquer sans modifications. En effet, tous sont basés sur l'exécution de `P()` et de sa complémentaire `V()` par deux processus différents. Des changements tels que celui présenté plus haut sont nécessaires pour appliquer ces solutions à MMS. Nous présentons dans [CK96] et au chapitre 7 deux solutions MMS au problème des lecteurs/rédacteurs.

Toutefois, on remarquera que le problème de possession du jeton peut présenter l'avantage d'éviter certaines erreurs qui peuvent facilement se produire avec les sémaphores classiques mais ne sont pas détectable par les compilateurs (par exemple l'inversion des opérations `P()` et `V()` [Shr76], [IBFW86]).

2.3.3.2 Acquisition en une seule fois de plusieurs jetons

Certains auteurs ont étendu l'opération `P()` afin de pouvoir décrémenter la valeur du sémaphore non pas de un mais d'un nombre quelconque passé en paramètre [VvL72], [Pre75]. P(s, x) sur un sémaphore s bloque si $s - x$ est négatif ou effectue $s := s - x$ et termine sinon. V(s, x) est l'opération complémentaire. Ceci est utilisé dans UNIX System V avec l'appel système `semop` par exemple [Bac86], [Roc86]. Dans la couche session du modèle de référence OSI un mécanisme de synchronisation similaire est utilisé au moyen duquel plusieurs jetons peuvent être acquis et relâchés par les primitives `TOKEN-PLEASE` et `TOKEN-GIVE` [ISO87b].

Dans MMS, il n'y a pas de façon d'acquérir plusieurs jetons en effectuant une seule requête `Take Control`. Ce service acquiert toujours un seul jeton libre et pas plus. Ceci peut sembler paradoxal pour un mécanisme aussi complexe que le sémaphore MMS. Il est par contre possible d'utiliser le modificateur `Attach To Semaphore` pour effectuer l'acquisition de plusieurs jetons dans un même service MMS. Considérons deux sémaphores MMS : s_1 a un seul jeton et s_2 a n jetons. Un client désire obtenir $k \leq n$ jetons de s_2 en utilisant un seul service MMS. Une solution est d'ajouter $k+1$ modificateurs `Attach To Semaphore` à n'importe quelle requête de service MMS confirmé, par exemple une requête de lecture `Read`[6]. Les k premiers modificateurs demandent chacun le contrôle d'un jeton de s_2. Le $(k+1)^{eme}$ modificateur spécifie le jeton de s_1. Si s_1 est déjà pris, la requête `Read` est bloquée après avoir acquis k jetons de s_2.

Le problème posé par le modificateur `Attach To Semaphore` provient de ce qu'il libère le jeton acquis immédiatement après exécution de la requête de service modifiée. Il est donc nécessaire de bloquer l'exécution de cette requête pendant la durée de la section critique c'est-à-dire pendant le temps où les k jetons de s_2 doivent être détenus. Pour cela on utilise le sémaphore s_1 dont on acquiert le jeton avant d'effectuer le service `Read` modifié. La séquence résultante est alors :

1. `Take Control`(s_1)
2. `Read` modifié /* acquiert k jetons de s_2 et se bloque sur s_1 */
3. /* Section critique */
4. `Relinquish Control`(s_1)
5. /* la requête Read est réellement exécutée ici */

[6] On choisit une requête qui ne fait aucune modification dans la VMD.

Mais cette façon de procéder n'est pas très propre car elle implique que le client qui veut acquérir les k jetons ne reçoit aucune réponse lui indiquant que ces jetons sont pris, comme c'est le cas pour un `Take Control`. Une solution est d'utiliser le service `Report Semaphore Entry Status` pour savoir quand les k jetons sont bien détenus par le client[7]. Mais alors la consultation périodique du sémaphore qui s'ensuit risque de ne pas rendre l'utilisation de modificateurs intéressante et la solution d'avoir k requêtes `Take Control` devient plus appropriée. Par ailleurs, cette solution ne permet pas la prise des k jetons de façon atomique contrairement à $\texttt{P}(s,k)$. L'acquisition en un service de plusieurs jetons d'un sémaphore ne peut donc pas se faire de façon simple dans MMS.

On peut aussi noter qu'il n'est pas possible dans MMS d'acquérir avec un seul service des jetons de différents sémaphores et de les conserver après exécution du service. Une telle opération est présentée dans [Pre75] et se retrouve encore dans le système d'exploitation UNIX. Dans [Dij71], Dijkstra a également étudié une opération `P()` fonctionnant en parallèle sur plusieurs sémaphores.

2.3.3.3 La destruction de sémaphores

Dans des programmes informatiques, un sémaphore classique cesse d'exister quand la visibilité de celui-ci se termine (par exemple le sémaphore est une variable locale à une procédure qui se termine). Ceci peut se produire n'importe quand. Un sémaphore MMS banalisé peut être détruit au moyen du service `Delete Semaphore` à condition qu'il soit effaçable par MMS (l'attribut `MMS Deletable` du sémaphore est VRAI) et qu'il n'y ait aucune application client détenant un jeton du sémaphore. Ceci introduit une restriction par rapport aux sémaphores classiques mais présente l'avantage d'éviter de laisser dans un état incohérent les applications détenant un jeton. Cette exigence se retrouve dans certains systèmes, par exemple la fonction `pthread_mutex_destroy` de DCE échoue si le mutex est déjà pris [Fou93].

Le noyau pSOS$^+$ permet aussi la création et la destruction dynamique de sémaphores [Tho90]. Les processus qui étaient en attente d'un sémaphore détruit sont mis dans l'état prêt. Ceci ne peut pas se produire dans MMS car si des applications sont en attente d'un sémaphore, alors ce sémaphore est pris et donc ne peut être détruit. Mais la norme MMS reste assez floue sur ce point et il n'est pas clair si un sémaphore pris de façon locale[8] peut être détruit. Si tel est le cas, il n'est pas décrit de procédure à suivre pour notifier les applications client en attente sur ce sémaphore. La solution la plus évidente est d'envoyer à chacune une réponse négative du `Take Control` en spécifiant comme erreur que l'objet sémaphore en question n'existe pas.

2.3.4 Qu'est ce qui est spécifique à MMS?

2.3.4.1 Gestion des associations

Toute requête de service MMS est exécutée sur une association. Un client peut avoir plusieurs associations avec le même serveur. Mais un client qui veut libérer un jeton d'un sémaphore doit exécuter la requête `Relinquish Control` sur la même association que celle où le `Take Control` a été fait. Par ailleurs, une application client MMS ne peut pas fermer l'association avec le service `Conclude` si elle détient un jeton d'un sémaphore sur cette association [ISO90a].

Les paramètres booléens optionnels `Relinquish Control If Connection Lost` et `Abort On Timeout` font partie de la requête `Take Control`. S'ils sont présents, ils sont copiés dans les attributs adéquats du SE correspondant (fig. 2.1). Le premier de ces paramètres indique si un jeton doit être libéré ou

[7] Le service `Report Semaphore Status` n'est pas suffisant car il ne permet pas de connaître les clients qui détiennent les jetons.

[8] Sémaphore pris sur décision de la VMD et non par un service `Take Control`.

si son SE doit aller dans l'état HUNG quand l'association entre le serveur et le client détenant ce jeton est perdue. Le second spécifie si l'association doit être rompue lorsque le temps de contrôle d'un jeton expire (Control Timeout).

Dans UNIX, les processus peuvent se terminer alors qu'ils détiennent un sémaphore. L'option SEM_UNDO de l'appel système semop est utilisée pour ordonner au noyau de relâcher ou non le sémaphore détenu par un processus qui se termine. L'option MMS Relinquish If Connection Lost peut être utilisée dans un but similaire. Elle est toutefois plus générale que dans UNIX car la perte d'une association n'implique pas nécessairement la mort de l'application client.

Les pertes d'associations peuvent modifier les relations 2.1 et 2.2 en 2.3 et 2.4 pour les mêmes raisons que l'option Control Timeout déjà citée. En effet, la perte d'une association peut entraîner la libération automatique d'un jeton donc la diminution de 1 des quantités Number Of Owned Tokens et *taille*(List Of Requesters).

2.3.4.2 Réquisition du contrôle d'un sémaphore

Un SE qui se trouve dans l'état HUNG peut être acquis par une application différente de celle qui était initialement propriétaire du jeton. Il suffit qu'un client envoie une requête de service Take Control contenant le paramètre Application To Preempt. Ce paramètre représente l'adresse de l'application client à laquelle on veut soustraire le jeton. Toute application client MMS a la possibilité d'acquérir le jeton détenu par toute autre application pour laquelle le temps de contrôle est expiré ou l'association avec le serveur perdue.

Le paramètre Application To Preempt ne permet pas d'acquérir le jeton d'une autre application dont le SE est dans l'état OWNER. La possibilité de prendre le contrôle d'un jeton d'un sémaphore est un moyen d'assurer la reprise sur panne dans MMS. En effet, un SE ne se trouve dans l'état HUNG qu'à la suite d'une perte d'association ou de l'expiration du temporisateur Control Timeout. Il peut être raisonnable de penser que ces situations se rencontrent essentiellement lors de la panne d'une application client. Il convient alors de récupérer le jeton anciennement détenu par cette application. Mais nous avons déjà mentionné que la perte d'une association n'implique pas nécessairement la mort du client. Dans ce cas, le client doit établir une nouvelle association avec le serveur. Comme le Relinquish Control ne peut se faire que sur la même association que le Take Control, le client ne peut qu'utiliser la réquisition citée pour récupérer le contrôle du jeton.

Cette possibilité propre à MMS augmente #Take Control sans changer NumberOfOwnedTokens ni nécessiter l'utilisation d'un Relinquish Control. Encore une fois les relations 2.1 et 2.2 deviennent donc 2.3 et 2.4.

2.3.4.3 Gestion des événements

Les événements MMS sont utilisés pour informer les clients de l'apparition d'événements sur les serveurs. L'objet “Event Condition” (EC) est utilisé pour maintenir l'information relative à un certain type d'événement. A chaque sémaphore MMS correspond un EC identifié par l'attribut Event Condition Reference (fig. 2.1). Il est créé automatiquement par le service Define Semaphore. Il sert à produire une notification d'événement qu'une application peut recevoir lorsque le délai Control Timeout expire. C'est une façon efficace d'informer les applications client que leur temps est expiré et qu'elles risquent de ne plus détenir le jeton. Les événements MMS fournissent une alternative à l'incapacité des sémaphores MMS de faire du signalement entre applications. La section 3 est dédiée à l'étude des événements MMS.

2.3.4.4 Le modificateur `Attach To Semaphore`

Nous avons introduit plus haut le modificateur `Attach To Semaphore` en disant qu'il s'agissait d'une condition optionnelle associée à n'importe quelle requête de service MMS confirmé. Ce modificateur conditionne l'exécution de la requête de service à la prise de contrôle d'un sémaphore. Le sémaphore est automatiquement libéré après l'exécution de la requête. Plusieurs modificateurs peuvent être associés à une requête de service. Ceux-ci sont alors traités par la VMD dans l'ordre de leur apparition dans la liste de modificateurs. L'acquisition des sémaphores désignés doit donc se faire séquentiellement.

Les paramètres utilisés dans un modificateur sont les mêmes que dans une requête `Take Control` (sauf `Application To Preempt` ce qui fait qu'un modificateur ne peut être utilisé pour acquérir le jeton détenu par une tierce application).

Un service MMS `xx_request` avec n modificateurs `Attach To Semaphore` est donc équivalent à une exécution de la séquence suivante :

```
TakeControl n° 1
      .
      .
TakeControl n° n
xx_request
RelinquishControl n° n
      .
      .
RelinquishControl n° 1
```

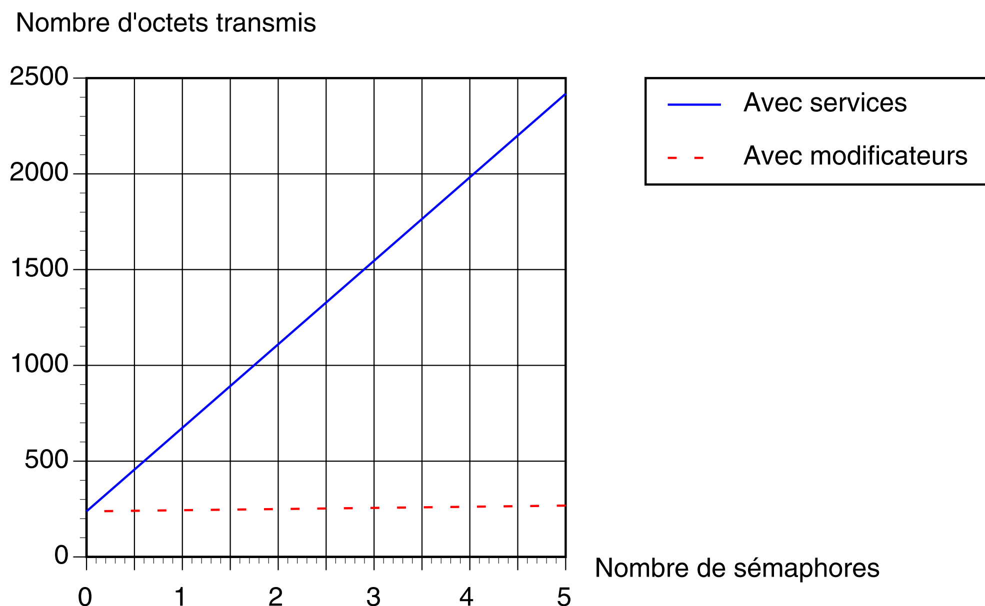


FIG. 2.4 - Solutions avec modificateurs et avec services `Take Control` et `Relinquish Control`

Cependant le nombre d'octets total nécessaire pour transmettre cette séquence est évidemment beaucoup plus grand que celui correspondant à l'utilisation de n modificateurs. Nous avons mesuré ces quantités pour un modificateur `Attach To Semaphore` sans aucune option dans un service de lecture

Read. L'intégration du modificateur au service Read n'ajoute que 6 octets[9]. Le rapport du nombre total d'octets transmis pour le cas avec modificateur sur le cas sans modificateur est alors déjà pour un seul modificateur de 0.36. Il est donc beaucoup plus intéressant d'utiliser un modificateur que les services Take Control et Relinquish Control quand un seul service MMS doit être protégé. La figure 2.4 montre comment le nombre d'octets transmis augmente en fonction du nombre de sémaphores utilisés. La solution avec modificateur est de loin la plus efficace. La taille des PDUs MMS varie en fonction du service, des paramètres et de la syntaxe de transfert utilisés. Mais contrairement aux mesures similaires que nous présentons pour le modificateur AttachToEventCondition dans la section 3, ici la différence entre les deux solutions est telle que l'influence de la taille des PDUs est négligeable.

Plus généralement, la table 2.3 compare l'intérêt des solutions avec modificateurs ou avec services Take Control / Relinquish Control en fonction de divers critères.

Le modificateur Attach To Semaphore ne peut protéger qu'une seule requête de service. Cette solution permet d'éviter l'exécution simultanée de services MMS qui devraient s'exclure. Toutefois ceci paraît bien limité d'autant que la VMD peut elle-même garantir une exécution sérialisée des services MMS manipulant des données communes. Par exemple deux services d'écriture sur une même variable MMS peuvent être sérialisés automatiquement par la VMD sans que ceci soit de la responsabilité des applications client, à la manière d'un moniteur [Hoa74]. L'exclusion mutuelle de services Write est d'ailleurs mentionnée dans la norme comme une possibilité de l'implantation des serveurs sans toutefois être rendue obligatoire.

Le modificateur Attach To Semaphore semble donc d'usage limité pour la protection des ressources. Il serait en fait beaucoup plus intéressant d'associer un modificateur Attach To Semaphore à un ensemble de services MMS.

Caractéristiques	**Modificateurs**	**Services**
Prise et libération automatique du sémaphore	oui	non
Nombre de services nécessaires avec n sémaphores	1	$2n + 1$
Plusieurs services dans la même section critique	non	oui
Attente sur plusieurs sémaphores avec le même service	oui, en utilisant plusieurs modificateurs	non
Combinaison avec attente d'un événement	oui, en utilisant le modif. Attach To Event Condition	non
Annulation du service par requête d'un client	oui, service Cancel seulement	oui, service Cancel puis Relinquish Control
Annulation automatique par expiration d'un temporisateur	oui	oui si bloqué sur Take Control

TAB. 2.3 - Comparaison entre le modificateur Attach To Semaphore et les services Take Control / Relinquish Control

2.3.5 L'invariant du sémaphore MMS

En nous appuyant sur les résultats obtenus dans les sections précédentes, nous pouvons maintenant donner un invariant plus adapté au sémaphore MMS. Pour tout sémaphore MMS, les relations suivantes sont vérifiées :

[9] Cette taille dépend bien sûr de la longueur du nom du sémaphore spécifié.

$$\texttt{NumberOfOwnedTokens} + \#RC + CT_r + AL_r + AD + \#TC_{preempt} = \#TC \quad (2.5)$$

$$taille(\texttt{ListOfRequesters}) + \texttt{NumberOfTokens} + \#RC + CT_r + AL_r + AD + \#TC_{preempt} = \#TC \quad (2.6)$$

avec les notations suivantes :

- #TC est le nombre total de requêtes `Take Control` effectuées;
- #RC est le nombre total de requêtes `Relinquish Control` effectuées;
- CT_r est le nombre d'expirations du temporisateur `Control Timeout` qui ont causé la libération d'un jeton;
- AL_r est le nombre de pertes d'associations qui ont entraîné la libération d'un jeton;
- AD est le nombre de requêtes `Take Control` qui ont échoué suite à l'expiration du temporisateur `Acceptable Delay`;
- $\#TC_{preempt}$ est le nombre de requêtes `Take Control` qui ont acquis un jeton dans l'état `HUNG`.

L'équation 2.5 s'applique dans le cas où il n'y a pas d'application en attente d'un jeton. L'équation 2.6 s'applique dans le cas contraire. Là encore les deux équations se rejoignent quand aucune application n'est en attente d'un jeton et que tous les jetons sont pris.

Dans ces relations l'opération #X ne prend pas en compte les requêtes de service X qui échouent comme par exemple un `Relinquish Control` effectué sur un sémaphore dont tous les jetons sont libres. Les requêtes qui échouent sont caractérisées par leur réponse négative (c'est-à-dire qu'elle ne contient pas les résultats du service mais l'erreur qui est survenue). La seule exception réside dans les réponses négatives des `Take Control` dont les délais `Acceptable Delay` sont expirés. Dans ce dernier cas ces requêtes `Take Control` sont comprises dans $\#TC$. Nous ne prenons pas non plus en compte le modificateur `Attach To Semaphore` car il acquiert et relâche toujours un jeton en un seul service et donc laisse les relations 2.5 et 2.6 inchangées. Seule l'option `Control Timeout` dans un modificateur peut changer ces relations en augmentant CT_r. Il suffit alors de considérer que $\#TC$ comprend les prises de contrôles de jetons par les modificateurs et que $\#RC$ comprend les libérations de jetons acquis par les modificateurs. Ainsi les relations 2.5 et 2.6 s'appliquent aussi pour le modificateur `Attach To Semaphore`.

Les inégalités suivantes s'appliquent également :

$$\#TC_{preempt} < \#TC \quad (2.7)$$

$$\#RC \leq \#TC \quad (2.8)$$

L'inégalité 2.7 est évidente puisque $\#TC$ prend en compte les requêtes `Take Control` avec réquisition et qu'il faut au moins qu'un jeton ait été pris pour effectuer une telle requête. L'inégalité 2.8 provient du fait qu'une requête `Relinquish Control` échoue quand tous les jetons d'un sémaphore sont libres. Il faut donc au moins autant de prises de contrôle de jetons qu'il y a de libérations.

Notons $nf(s)$ le nombre de fois que le sémaphore s a été franchi[10]. Alors l'exécution des requêtes `Take Control`, `Relinquish Control`, `Take Control` avec réquisition, les pertes d'associations et les expirations des temporisateurs laissent invariante la relation suivante :

[10] $nf(s)$ ne prend pas en compte les requêtes `Take Control` avec réquisition car lors d'une telle requête le sémaphore a déjà été franchi auparavant et n'a pas été libéré entre temps.

$$nf(s) = min(\#TC - \#TC_{preempt} - AD,\ \texttt{NumberOfTokens} + \#RC + CT_r + AL_r) \quad (2.9)$$

En nous appuyant sur la démonstration de [CRO75] pour le sémaphore classique, nous allons démontrer que les requêtes de service MMS `Take Control` et `Relinquish Control` laissent invariante la relation 2.9 et ce quelques soient les paramètres utilisés dans les requêtes `Take Control`.

Résumons d'abord les effets des différentes opérations applicables à un sémaphore MMS et pouvant modifier une des opérandes de la relation 2.9. Par commodité, notons `TC_EN_ATTENTE` un booléen qui est VRAI quand il existe des requêtes `Take Control` en attente de libération d'un jeton et FAUX dans le cas contraire. On a donc :

$$\texttt{TC_EN_ATTENTE} = (\#TC - AD - \#TC_{preempt}) > (\texttt{NumberOfTokens} + \#RC + CT_r + AL_r)$$

Les effets des opérations s'appliquant sur un sémaphore MMS sont les suivants :

<u>`Take Control`</u> :

$\#TC := \#TC + 1$
si `NumberOfTokens` > `NumberOfOwnedTokens`
  alors $nf(s) = nf(s) + 1$;   `NumberOfOwnedTokens` := `NumberOfOwnedTokens` + 1
  sinon *taille*(`ListOfRequesters`) := *taille*(`ListOfRequesters`) + 1

<u>`Relinquish Control`</u> :

$\#RC := \#RC + 1$
si `TC_EN_ATTENTE` = FAUX
  alors `NumberOfOwnedTokens` := `NumberOfOwnedTokens` - 1
  sinon $nf(s) = nf(s) + 1$;   *taille*(`ListOfRequesters`) := *taille*(`ListOfRequesters`) - 1

<u>`Take Control` avec réquisition</u> :

$\#TC := \#TC + 1$
$\#TC_{preempt} := \#TC_{preempt} + 1$

<u>Expiration d'un temporisateur `Control Timeout`</u> :

$\#CT_r := \#CT_r + 1$
si `TC_EN_ATTENTE` = FAUX
  alors `NumberOfOwnedTokens` := `NumberOfOwnedTokens` - 1
  sinon $nf(s) = nf(s) + 1$;   *taille*(`ListOfRequesters`) := *taille*(`ListOfRequesters`) - 1

<u>Perte d'association</u> :

$\#AL_r := \#AL_r + 1$
si `TC_EN_ATTENTE` = FAUX
  alors `NumberOfOwnedTokens` := `NumberOfOwnedTokens` - 1
  sinon $nf(s) = nf(s) + 1$;   *taille*(`ListOfRequesters`) := *taille*(`ListOfRequesters`) - 1

<u>Expiration d'un temporisateur `Acceptable Delay`</u> :

$\#AD := \#AD + 1$
si `TC_EN_ATTENTE` = VRAI
  alors *taille*(`ListOfRequesters`) := *taille*(`ListOfRequesters`) - 1

Forme initiale de la relation		Relation après exécution de $\#TC := \#TC + 1$	Effet sur $nf(s)$	Relations avec $nf(s)$ après exécution
$Dem < Aut$	$\begin{cases} nf(s) = Dem \\ nf(s) < Aut \end{cases}$	$Dem \leq Aut$	$nf(s) := nf(s) + 1$	$\begin{cases} nf(s) = Dem \\ nf(s) \leq Aut \end{cases}$
$Dem \geq Aut$	$\begin{cases} nf(s) = Aut \\ nf(s) \leq Dem \end{cases}$	$Dem > Aut$	-	$\begin{cases} nf(s) < Dem \\ nf(s) = Aut \end{cases}$

TAB. 2.4 - Effets d'un `Take Control`

Forme initiale de la relation		Relation après exécution de $\#RC := \#RC + 1$	Effet sur $nf(s)$	Relations avec $nf(s)$ après exécution
$Dem > Aut$	$\begin{cases} nf(s) = Aut \\ nf(s) < Dem \end{cases}$	$Dem \geq Aut$	$nf(s) := nf(s) + 1$	$\begin{cases} nf(s) \leq Dem \\ nf(s) = Aut \end{cases}$
$Dem \leq Aut$	$\begin{cases} nf(s) = Dem \\ nf(s) \leq Aut \end{cases}$	$Dem < Aut$	-	$\begin{cases} nf(s) = Dem \\ nf(s) < Aut \end{cases}$

TAB. 2.5 - Effets d'un `Relinquish Control`

Nous pouvons maintenant démontrer la relation 2.9. Remarquons d'abord que celle-ci est vérifiée avant que toute opération soit effectuée sur le sémaphore. En effet :

$$nf(s) = \#TC = \#TC_{preempt} = AD = CT_r = AL_r = \#RC = min(0, \texttt{NumberOfTokens}) = 0$$

Supposons la relation 2.9 vérifiée et examinons les effets des opérations précédentes sur cette relation. Pour alléger la notation notons Dem (pour “Demandes”) la quantité $\#TC - \#TC_{preempt} - AD$. Dem correspond au nombre total de demandes de prise de contrôle du sémaphore. Notons Aut (pour “Autorisations”) la quantité $\texttt{NumberOfTokens} + \#RC + CT_r + AL_r$. Aut correspond au nombre total d'autorisations de passage du sémaphore jusqu'au moment présent.

Les tables 2.4 à 2.9 donnent la valeur de $nf(s)$ après exécution de toutes les opérations pouvant affecter l'état d'un sémaphore MMS. On constate que la relation 2.9 reste toujours vérifiée. Nous avons donc montré que la relation 2.9 est vraie quelques soient les opérations appliquées à un sémaphore MMS.

Forme initiale de la relation		Relation après exécution de $\#TC := \#TC + 1$ $\#TC_{preempt} := \#TC_{preempt} + 1$	Effet sur $nf(s)$	Relations avec $nf(s)$ après exécution
$Dem < Aut$	$\begin{cases} nf(s) = Dem \\ nf(s) < Aut \end{cases}$	$Dem < Aut$	-	$\begin{cases} nf(s) = Dem \\ nf(s) < Aut \end{cases}$
$Dem \geq Aut$	$\begin{cases} nf(s) = Aut \\ nf(s) \leq Dem \end{cases}$	$Dem \geq Aut$	-	$\begin{cases} nf(s) \leq Dem \\ nf(s) = Aut \end{cases}$

TAB. 2.6 - Effets d'un `Take Control` avec réquisition

Forme initiale de la relation	Relation après exécution de $\#CT_r := \#CT_r + 1$	Effet sur $nf(s)$	Relations avec $nf(s)$ après exécution
$Dem > Aut \begin{cases} nf(s) < Dem \\ nf(s) = Aut \end{cases}$	$Dem \geq Aut$	$nf(s) := nf(s) + 1$	$\begin{cases} nf(s) \leq Dem \\ nf(s) = Aut \end{cases}$
$Dem \leq Aut \begin{cases} nf(s) = Dem \\ nf(s) \leq Dem \end{cases}$	$Dem < Aut$	-	$\begin{cases} nf(s) = Dem \\ nf(s) \leq Aut \end{cases}$

TAB. 2.7 - Effets de l'expiration d'un temporisateur `Control Timeout`

Forme initiale de la relation	Relation après exécution de $\#AL_r := \#AL_r + 1$	Effet sur $nf(s)$	Relations avec $nf(s)$ après exécution
$Dem > Aut \begin{cases} nf(s) < Dem \\ nf(s) = Aut \end{cases}$	$Dem \geq Aut$	$nf(s) := nf(s) + 1$	$\begin{cases} nf(s) \leq Dem \\ nf(s) = Aut \end{cases}$
$Dem \leq Aut \begin{cases} nf(s) = Dem \\ nf(s) \leq Dem \end{cases}$	$Dem < Aut$	-	$\begin{cases} nf(s) = Dem \\ nf(s) \leq Aut \end{cases}$

TAB. 2.8 - Effets d'une perte d'association

Forme initiale de la relation	Relation après exécution de $\#AD := \#AD + 1$	Effet sur $nf(s)$	Relations avec $nf(s)$ après exécution
$Dem > Aut \begin{cases} nf(s) < Dem \\ nf(s) = Aut \end{cases}$	$Dem \geq Aut$	-	$\begin{cases} nf(s) \leq Dem \\ nf(s) = Aut \end{cases}$
$Dem \leq Aut \begin{cases} nf(s) = Dem \\ nf(s) \leq Dem \end{cases}$	Ne peut pas se produire dans cet état		

TAB. 2.9 - Effets de l'expiration d'un temporisateur `Acceptable Delay`

2.4 Pourquoi deux classes de sémaphores

MMS définit deux classes de sémaphores : le sémaphore banalisé et le sémaphore étiqueté. La différence essentielle réside dans la politique d'allocation des jetons. Le sémaphore étiqueté contient des *jetons nommés*. Lors d'une requête de service `Take Control`, l'application client peut demander l'acquisition d'un jeton particulier au moyen du paramètre de service `Named Token`. Si ce jeton est déjà pris, alors l'application doit attendre même si d'autres jetons du sémaphore sont libres. Si le client ne demande pas de jeton particulier alors n'importe quel jeton libre lui est alloué. Dans les deux cas, le nom du jeton pris est retourné dans la réponse du `Take Control`. Ce nom doit ensuite être fourni à la requête `Relinquish Control` qui libère le jeton. Par contre, les jetons d'un sémaphore banalisé sont anonymes. N'importe quel jeton libre est alloué à une requête de prise de contrôle.

Une autre différence est que le jeton d'un sémaphore étiqueté est d'une certaine façon lié à une ressource physique modélisée par la VMD. Les sémaphores étiquetés sont donc toujours pré-définis et ne peuvent jamais être détruits par un service MMS. Le sémaphore banalisé peut être détruit par le service `Delete Semaphore` dans la mesure où son attribut `MMS Deletable` est à vrai. MMS rajoute un service qui ne s'applique qu'aux sémaphores étiquetés : `Report Pool Semaphore Status`. Ce service retourne la liste des jetons nommés libres ("free"), alloués ("owned") ou suspendus ("hung").

Le sémaphore étiqueté est utile pour regrouper des ressources apparentées. Par exemple, une VMD peut utiliser un sémaphore étiqueté pour protéger une série d'imprimantes contre des requêtes d'impression simultanées. Chaque imprimante est associée à un jeton nommé [Nus91b]. Il en résulte une façon plus claire de manipuler et gérer les ressources et facilite la programmation d'applications MMS. Le sémaphore banalisé peut aussi permettre un regroupement de ressources mais dans ce cas, ces dernières sont allouées de façon anonyme puisque les jetons ne sont pas nommés. Cependant, l'attribution de noms à des ressources peut aussi se faire avec les sémaphores banalisés en affectant un sémaphore à jeton unique à chaque ressource (par exemple à chaque imprimante).

En fait, c'est surtout une raison d'implantation qui pourrait justifier l'utilisation du sémaphore étiqueté pour le regroupement de ressources : il faut moins de place mémoire pour maintenir l'information représentant un sémaphore étiqueté avec n jetons que pour n sémaphores banalisés à jeton unique.

La protection de ressources dans MMS postule l'existence d'un lien entre un sémaphore étiqueté et une ressource physique réelle. Cela signifie que la visibilité de la prise de contrôle d'un sémaphore n'est pas limitée aux clients du serveur MMS où réside le sémaphore mais peut également s'étendre en dehors du domaine de discours de MMS. Ainsi, le sémaphore étiqueté devient un moyen de représenter un sémaphore "réel" tout comme une variable MMS est une abstraction permettant de représenter des variables réelles. Il n'appartient pas à MMS de normaliser la relation entre un sémaphore MMS et ce sémaphore réel. La nature de celle-ci est laissée libre à chaque implantation. Par exemple, on peut imaginer qu'une variable MMS soit associée au jeton d'un sémaphore étiqueté et que la VMD refuse tout accès à cette variable si le jeton n'est pas pris, et ce que l'accès soit externe (service `Write`) ou interne (par une action du dispositif physique modélisé par la VMD). C'est la raison pour laquelle un sémaphore étiqueté est pré-défini est non destructible par MMS. Mais il est tout à fait possible d'établir une relation identique avec un sémaphore banalisé. En effet, rien n'interdit un sémaphore banalisé d'être pré-défini, non destructible par service MMS et lié à une ressource physique.

Il est précisé dans la norme que le sémaphore étiqueté est utilisé pour la protection de ressources alors que le sémaphore banalisé est plutôt utilisé pour la synchronisation entre applications. En fait, dans une des premières versions de la norme MMS [ISO86], le sémaphore étiqueté était appelé *ressource* alors que le sémaphore banalisé était simplement appelé *sémaphore*. La synchronisation par signalement et l'exclusion mutuelle constituent bien deux problèmes différents [Dij68b], [Shr76]. Mais MMS ne semble pas avoir fait cette différence. Si le sémaphore banalisé doit être utilisé pour la synchroni-

sation, il est surprenant de trouver l'obligation d'associer les services `Take Control` et `Relinquish Control`. Ceci peut se comprendre pour le sémaphore étiqueté dont le but est de protéger l'accès à des ressources partagées [Coo91]. En effet, les ressources d'une VMD correspondent généralement à des ressources physiques réelles pour lesquelles des accès incontrôlés pourraient avoir des conséquences graves ou catastrophiques. Mais l'obligation d'associer `Take Control` et `Relinquish Control` pour les problèmes de synchronisation avec le sémaphore MMS banalisé n'est pas justifiée. On peut alors se demander si la différence entre les deux classes de sémaphore est vraiment utile puisque tous deux sont à certains détails près identiques :

1. Un sémaphore étiqueté avec n jetons nommés $j_1, ..., j_n$ est équivalent à n sémaphores banalisés non effaçables par MMS, avec chacun un jeton unique et nommés $j_1, ..., j_n$.

2. Un sémaphore banalisé avec n jetons et non effaçable par MMS est équivalent à un sémaphore étiqueté avec n jetons nommés dans la mesure où le service `Take Control` est le modificateur `Attach To Semaphore` sont utilisés sans spécifier de jeton nommé.

3. Un sémaphore banalisé peut, à l'image d'un sémaphore étiqueté, être pré-défini, non effaçable par MMS et lié à une ressource physique.

En résumé, seuls un "confort" de programmation et éventuellement des considérations d'implantation justifient de faire une différence entre sémaphore banalisé et étiqueté. S'il est certainement "confortable" d'utiliser un sémaphore étiqueté pour regrouper diverses ressources de caractéristiques identiques, ce n'est pas non plus obligatoire. Il semble plutôt que la définition de deux classes de sémaphore entretienne la confusion sur la compréhension et l'utilisation des sémaphores MMS. On pourrait suggérer que la norme MMS choisisse de normaliser uniquement les sémaphores banalisés ou alors seulement les sémaphores étiquetés. Dans ce dernier cas, la norme devrait aussi autoriser la création des sémaphores étiquetés avec le service `Define Semaphore`. Ceci devrait améliorer la compréhension de la gestion de sémaphores MMS et réduire le code nécessaire à leur implantation sans pour autant diminuer les possibilités des sémaphores MMS.

Enfin, compte tenu de l'apparente complexité des sémaphores MMS, il aurait été plus utile d'introduire des sémaphores (ou verrous) qui autorisent la différenciation entre accès en lecture et accès en écriture comme c'est le cas dans le domaine des bases de données ou avec le système d'exploitation VAX/VMS par exemple [Dig82]. Un attribut optionnel dans les paramètres de la requête de service `Take Control` pourrait indiquer si cette requête est utilisée pour modifier une ressource ou simplement pour la lire. Ceci pourrait être particulièrement intéressant pour l'implantation d'algorithmes de contrôle de la concurrence basés sur MMS [Cas95c]. Nous proposons une telle modification à la section 7.2.2.5.

2.5 Sémaphores MMS et interblocages

Le problème de l'interblocage se pose inévitablement dans tout programme utilisant des sémaphores. Dans MMS, les interblocages se produisent de la même façon qu'avec les sémaphores classiques. Mais MMS introduit aussi de nouvelles possibilités d'interblocage notamment avec les modificateurs.

Considérons le scénario suivant. Deux sémaphores banalisés s_1 et s_2 avec un seul jeton sont définis dans une VMD. Une requête `Take Control` est effectuée sur s_1 par un client donné. Ensuite, une requête de service quelconque notée `xx_request` est demandée par un autre client. Cette requête contient deux modificateurs `Attach To Semaphore` spécifiant s_2 et s_1 dans cet ordre. A ce stade, les deux sémaphores sont pris mais `xx_request` ne peut s'exécuter tant que s_1 n'est pas libéré. Maintenant,

le client qui détient s_1 effectue une requête `Take Control` sur s_2. Il en résulte un interblocage dont la cause est cachée par les modificateurs. Une autre situation d'interblocage se produit aussi lorsqu'un même service contient plus de modificateurs spécifiant le même sémaphore qu'il n'y a de jeton dans ce sémaphore. Là il suffit d'une seule requête de service pour provoquer l'interblocage. En fait, la requête s'auto-interbloque.

La façon de traiter les interblocages n'est pas spécifiée dans MMS. Si une VMD n'a pas la capacité de les prévenir ou de les détecter et de les résoudre, ce sont alors les clients qui sont responsables du bon déroulement de leurs requêtes de service vis-à-vis des interblocages. Dans le cas contraire, la seule allusion faite dans la norme concernant les interblocages est qu'une VMD peut utiliser tous les mécanismes appropriés pour éviter leur formation y compris le refus d'exécution d'une requête de service, comme c'est le cas dans VAX/VMS. La classe d'erreur `service-preempt` associée au code `deadlock` est d'ailleurs prévue à cet effet. Par ailleurs, nous avons vu que l'ordre d'exécution des requêtes ne correspond pas nécessairement à l'ordre de leur réception par le serveur. La modification de l'ordre de réception pourrait se justifier par la possibilité d'éviter la formation d'interblocages qui seraient autrement apparus sans ce ré-arrangement.

Toutefois, une VMD ne peut détecter des interblocages répartis c'est-à-dire des cycles de dépendance qui se forment entre plusieurs VMDs et plusieurs clients. Cela supposerait une autonomie de décision et une coopération entre serveurs à la manière des contrôleurs de transactions dans les bases de données [CMH83], ce qui n'a pas du tout été prévu par la norme. Nous proposons dans [Cas95c], [Cas95a] ainsi qu'au chapitre 7 deux algorithmes répartis situés au niveau des applications client MMS et dont le but est de détecter et résoudre les interblocages répartis dans l'environnement MMS.

2.6 Sémaphores MMS et CSP

Rappelons que CSP ("Communicating Sequential Processes") est un langage permettant de décrire formellement des programmes répartis et concurrents. CSP peut être utilisé pour prouver certaines propriétés du système modélisé (propriétés d'invariance, de sûreté, etc.).

Dans [Hoa78], Hoare donne l'implantation suivante d'un sémaphore classique en CSP :

```
S::val:integer; val:= 0;
     *[(i:1..100)X(i)?V() → val:= val + 1
     [](i:1..100)val > 0; X(i)?P() → val:= val - 1
     ]
```

Dans le seul but d'illustrer la différence avec les sémaphores MMS, nous allons donner ici une représentation CSP simplifiée d'un sémaphore MMS banalisé. Nous supposons qu'il n'y a pas plus de 100 clients MMS et que ceux-ci sont représentés par le tableau X(i:1..100). L'équivalent CSP du sémaphore MMS banalisé est donné sur la figure 2.5.

`ListOfRequesters`, `ListOfOwners`, `WaitTimer` et `LockTimer` sont des processus dont le comportement se comprend aisément et qui pour des raisons de brièveté ne sont pas décrits ici. Nous avons supposé que les paramètres optionnels fournissant les valeurs des temporisateurs d'une requête `Take Control` sont toujours présents. L'absence d'une telle valeur équivaut à avoir un temporisateur avec la valeur "pour toujours". Dans notre représentation CSP, démarrer ou arrêter un temporisateur ayant la valeur "pour toujours" n'a aucun effet.

```
S::tokens:integer; tokens:= NumberOfTokens;
      priority, acceptableDelay, controlTimeout:integer;
      isIn:boolean;

      *[(i:1..100)X(i)?relinquishControl(); ListOfOwners!has(i); ListOfOwners?isIn →
            [  isIn = false → X(i)!negativeResponse()
            [] isIn = true  → ListOfOwners!unqueue(i); LockTimer!Stop(i); X(i)!positiveResponse();
                  ListOfRequesters!status(); ListOfRequesters?empty;
                  [  empty = true  → tokens:= tokens + 1
                  [] empty = false → j:integer; ListOfRequesters!getTop();
                        ListOfRequesters?(j, controlTimeout); WaitTimer!Stop(j); ListOfOwners!queue(j);
                        LockTimer!Start(j, controlTimeout); X(j)!positiveResponse()
                  ]
            ]

      [](i:1..100)X(i)?takeControl(priority, acceptableDelay, controlTimeout); tokens > 0 →
            tokens:= tokens - 1; ListOfOwners!queue(i); X(i)!positiveResponse();
            LockTimer!Start(i, controlTimeout)

      [](i:1..100)X(i)?takeControl(priority, acceptableDelay, controlTimeout); tokens = 0 →
            ListOfRequesters!queue(i, priority, controlTimeout); WaitTimer!Start(i, acceptableDelay)

      [](i:1..100)LockTimer?Timeout(i); → ListOfOwners!unqueue(i)     # Normally goes to hung state

      [](i:1..100)WaitTimer?Timeout(i); → ListOfRequesters!unqueue(i); X(i)!negativeResponse()
      ]
```

FIG. 2.5 - Sémaphore MMS banalisé en CSP

Par ailleurs, nous n'avons pas considéré :

- Les services de création/destruction/consultation des sémaphores;
- Les options spécifiques à MMS `Relinquish If Connection Lost` et `Abort On Timeout` ainsi que l'état `HUNG`;
- Tous les cas possibles où les services MMS retournent une erreur.

Il est important de noter la différence au niveau des clients entre les deux exemples. Alors que pour les sémaphores classiques un processus client effectue simplement un `S!P()`, pour les sémaphores MMS un client doit effectuer `S!takeControl()` suivi de `[S?positiveResponse() [] S?negativeResponse()]`. Ceci permet au client de savoir si sa requête a réussi ou non. Ce point est sans objet pour les processus utilisant les sémaphores classiques puisqu'ils traversent simplement `P()`. Il y a certainement d'autres façons d'implanter un semaphore MMS en CSP. Celle-ci n'est qu'un exemple permettant d'illustrer les différences entre sémaphores classiques et sémaphores MMS.

2.7 Conclusion sur les sémaphores MMS

Les sémaphores MMS peuvent être perçus comme une sorte d'instance des sémaphores classiques auxquels on a apporté des améliorations et ajouté des caractéristiques propres à MMS. Le sémaphore MMS est un mécanisme puissant pour le contrôle d'accès aux ressources. Mais le sémaphores MMS

présente également quelques désavantages par rapport aux sémaphores classiques. Dans ce chapitre nous avons identifié des problèmes que nous résumons ici :

- Les sémaphores MMS sont différents des sémaphores classiques. Les primitives de gestion de sémaphores proposées dans certains systèmes d'exploitation ont souvent des caractéristiques proches de celles des services MMS. Mais aucune ne se comporte vraiment de la même façon ni regroupe toutes les options que l'on trouve dans MMS.
- Les sémaphores MMS sont essentiellement adaptés à la protection de ressources. Ils ne peuvent pas être utilisés de la façon habituelle pour résoudre les problèmes de synchronisation et de signalisation.
- L'invariant du sémaphore doit être adapté à MMS.
- Il semble qu'il y ait peu de justification à l'introduction de deux classes de sémaphore qui sont dans une large mesure identiques.
- Comme il a été souligné dans [Shr76], le sémaphore est un mécanisme de bas niveau. Il est surprenant de trouver un tel mécanisme dans un système haut niveau comme MMS. On pourrait s'attendre à disposer de services de niveau d'abstraction plus élevé qui cacheraient l'utilisation des sémaphores aux développeurs d'applications MMS[11].

  Dans une messagerie qui se veut aussi générale que MMS il peut toutefois être difficile de trouver de tels services qui ne soient pas dépendants d'un type d'application donnée.

[11] Par exemple en ce qui concerne la lecture/écriture de variables, MMS va déjà dans ce sens en proposant la possibilité d'accéder à plusieurs variables avec un seul service. Si des contraintes d'atomicité existent entre ces variables, il appartient alors au serveur de les satisfaire. Ceci est caché au client qui ne souhaite qu'obtenir les valeurs des variables ou être assuré de les avoir toutes modifiées. Il n'est donc pas nécessaire que le client protège l'ensemble des variables par sémaphore.

Chapitre 3

Les événements MMS

3.1 Introduction

Le but de ce chapitre est de clarifier le concept d'événement MMS en comparant la gestion d'événements MMS à d'autres systèmes connus comportant également une gestion d'événements tels que les systèmes d'exploitation, la gestion de réseaux ou le déverminage d'applications. Cet état de l'art sur la gestion d'événements doit nous permettre en particulier d'identifier les limitations des événements MMS. Nous consacrons des chapitres ultérieurs à l'étude de solutions pour surmonter certaines de ces limitations. Quelques exemples d'applications des événements MMS dans des applications industrielles sont proposés à l'annexe F.

La gestion d'événements MMS fournit les mécanismes et services permettant d'informer les clients MMS de l'occurrence d'événements dans les serveurs. Un événement peut apparaître à l'initiative d'un client auquel cas on le nomme *événement déclenché* ("NETWORK-TRIGGERED"). Les événements déclenchés permettent la synchronisation et le signalement entre clients MMS. Ils peuvent être comparés à des fonctions de systèmes d'exploitation répartis [Nus91b]. Mais un événement peut aussi survenir à l'initiative du dispositif physique modélisé par la VMD auquel cas on le nomme *événement scruté* ("MONITORED"). Il appartient à chaque serveur de s'assurer régulièrement que les conditions d'apparition d'un événement scruté sont ou non satisfaites. Les clients sont *notifiés* de l'apparition d'un événement et un événement peut donner lieu à plusieurs *notifications*. Les notifications d'événement peuvent être *acquittées* par les clients. Par ailleurs, un événement peut également entraîner l'exécution d'une ou plusieurs *actions événementielles*. Ces dernières ne sont autres que des services MMS automatiquement exécutés lors de l'apparition d'un événement. Enfin, MMS permet la définition de services dont l'exécution est différée jusqu'à l'occurrence d'un événement. L'exécution de ces services est donc conditionnée sur l'apparition d'un événement. On dit alors que ces services sont *modifiés* sur un événement.

Nous allons maintenant analyser plus en détail la gestion des événements MMS.

3.2 Définitions d'un événement

MMS ne donne pas une définition précise de ce qu'est un événement. Ceci provient sans doute du fait que la condition d'apparition d'un événement est hors du domaine de discours de MMS c'est-à-dire qu'il n'est pas du ressort de MMS de définir les conditions d'apparition d'un événement. Donc l'événement MMS n'est théoriquement qu'un moyen de représenter de façon homogène tous les événements des systèmes inscrits dans un environnement MMS.

Nous avons retenu un certain nombre de définitions d'événement dans la littérature :

1. *Un événement est un instant dans le temps* [DJJ89]
2. *Un événement est l'apparition de quelque chose de remarquable ou de significatif dans le déroulement normal des processus d'application* [Pim90]
3. *Un événement (simple) est le changement de la valeur d'un attribut d'un objet* [CPR86]
4. *Un événement dans un processus p est défini comme une entité atomique qui reflète un changement d'état de p ou d'un canal de communication adjacent à p* [LSM90], [CL85]
5. *L'événement est l'outil le plus primitif permettant de résoudre le problème de synchronisation* [Sch86]
6. *L'événement est l'objet de base permettant d'exprimer la synchronisation entre processus dans les protocoles* [BTE98]
7. *Un événement est un changement observable dans l'état d'un objet* [Ode94], [Gro94]
8. *Les événements sont les seuls objets observables dans un système et toute interaction avec le monde externe intervient sous forme d'événement* [DJJ89]

Nous avons cherché à organiser ces définitions par degré croissant d'abstraction et de précision.

Le choix d'une définition applicable à MMS est rendu difficile par le niveau d'abstraction auquel on se situe pour définir un événement. Comme nous le verrons plus loin, on peut identifier dans le contexte de MMS quatre niveaux d'événements. De ces quatre niveaux, nous pouvons pour l'instant en garder deux essentiels : l'événement réel et l'événement MMS.

Les définitions 1 et 2 (et dans une moindre mesure 3 et 4) traduisent ce que nous appelons un *événement réel* c'est-à-dire un fait significatif dans le déroulement d'un processus informatique. L'événement réel est en dehors du domaine de discours de MMS. En fait, l'*événement MMS* se situe à un niveau d'abstraction plus élevé car il incarne de façon homogène et unifiée tous les événements réels visibles et n'est qu'un moyen de représenter et de transmettre l'information afférente à l'événement réel. **L'événement MMS est donc la manifestation de l'événement réel**. Elle se traduit par la réception d'une notification[1]. Un événement réel qui ne produit pas de notification (donc d'événement MMS) ne peut pas être retenu comme significatif dans la déroulement d'une application MMS, simplement parce qu'il n'existe pour aucune application client MMS.

Les définitions 5 à 8 se rapprochent de l'événement MMS car elles font état de la manifestation de l'événement. L'événement MMS a deux fonctions principales : informer d'un changement d'état et synchroniser les applications MMS.

Le terme "objet" est fréquemment utilisé dans les définitions précédentes pour représenter un événement ou comme entité ayant provoqué un événement. Nous avons déjà mentionné que pour représenter l'information contenue dans une VMD, MMS fait appel au formalisme objet. Dans MMS, l'information commune à un type d'événement est donc représentée à l'aide de l'objet *Condition Evénementielle* (EC ou "Event Condition"). L'EC spécifie la condition de réalisation d'un événement [Nus91b]. La classe d'objet EC représente tous les événements possibles (visibles) dans le monde MMS. Une instance d'objet EC décrit un type d'événement possible (fig. 3.1).

[1] Qu'il s'agisse d'une notification MMS ou de la réponse d'un service bloqué en attente d'un événement (service modifié)

```
Object: Event Condition

  Key Attribute: Event Condition Name
  Attribute: MMS Deletable (TRUE, FALSE)
  Attribute: Event Condition Class (NETWORK-TRIGGERED, MONITORED)
  Attribute: State (DISABLED, IDLE, ACTIVE)
  Attribute: Priority
  Attribute: Severity
  Attribute: Additional Detail
  Attribute: List of Event Enrollment References
  Constraint: Event Condition Class = MONITORED
        Attribute: Enabled (TRUE, FALSE)
        Attribute: Alarm Summary Report (TRUE, FALSE)
        Attribute: Monitored Variable Reference
        Attribute: Evaluation Interval
        Attribute: Time Of Last Transition To Active
        Attribute: Time Of Last Transition To Idle
```

FIG. 3.1 - L'objet Condition Evénementielle

Les définitions 3 et 7 correspondent assez bien à la représentation d'un événement scruté puisque dans MMS ce type d'événement est perçu comme se produisant lors du changement de valeur de l'attribut `State` d'un objet EC. Les définitions 5 et 6 correspondent mieux à l'événement MMS déclenché car ce dernier permet d'effectuer des opérations de signalisation entre applications client MMS à la manière des primitives de synchronisation des systèmes d'exploitation. La définition 2 bien que plus floue s'adapte à ces deux classes d'événement. Nous étudions ces deux classes d'événement plus en détail dans la section suivante.

3.3 Conditions d'apparition d'un événement

3.3.1 Apparition par déclenchement

La façon la plus simple de faire apparaître un événement dans MMS est pour une application client de demander explicitement son déclenchement. Ceci se fait par l'intermédiaire du service `Trigger Event`. L'objet EC représentant l'événement considéré doit alors être de classe `NETWORK-TRIGGERED` (c'est-à-dire que l'attribut `Event Conditions Class` doit avoir la valeur `NETWORK-TRIGGERED`). L'événement réel qui donne naissance à l'événement MMS provient de la logique d'exécution d'une application client. Il est donc bien hors du domaine de discours de MMS. La figure 3.2 montre comment se produit l'événement déclenché.

Dans la section précédente, la définition 7 ne s'applique pas aux ECs de classe `NETWORK- TRIGGERED` car le déclenchement d'un événement au moyen du service `Trigger Event` n'entraîne pas de modification d'état de l'EC. En fait il n'y a modification d'aucun de ses attributs à part éventuellement sa priorité (attribut `Priority`). Mais la modification de la priorité est optionnelle et n'a aucun rapport avec le déclenchement de l'événement. L'état de l'EC reste donc inchangé. En particulier, son attribut `State` reste fixé à `DISABLED` durant toute sa durée de vie ce qui caractérise d'ailleurs les événements déclenchés par rapport à ceux scrutés.

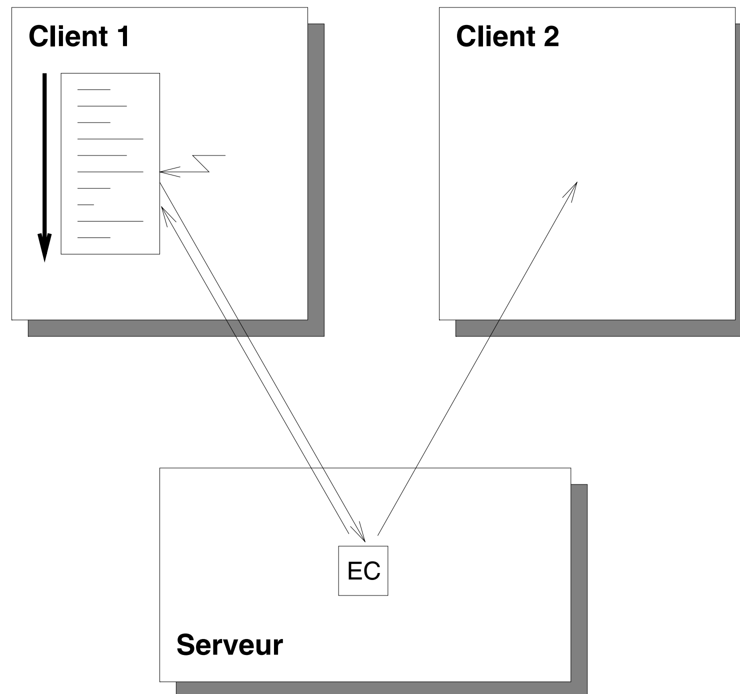


FIG. 3.2 - Evénement MMS déclenché

En fait l'événement déclenché par le service `Trigger Event` ne représente qu'un moyen de communication entre un client et un ou plusieurs autres clients via un serveur. Il s'apparente tout à fait aux événements et signaux entres processus d'un système d'exploitation. Par exemple, l'appel système `kill` de UNIX permet l'envoi d'un signal à un processus ou à un groupe de processus [Bac86]. L'opération `SIGNALER` du noyau temps-réel Sceptre réveille une tâche en attente d'un événement [BDD+84]. Ceci se retrouve également dans le langage temps-réel ORE où l'instruction `fire` permet d'envoyer la notification d'un événement à plusieurs processus en attente sur cet événement [DJJ89]. L'instruction ORE `trigger` permet le même résultat mais pour un seul processus. MMS regroupe dans le service `Trigger Event` ce que font ces deux instructions. MMS permet en plus de modifier la priorité de l'événement à déclencher lors de l'utilisation du `Trigger Event`. Dans le domaine des interruptions sur microprocesseurs, le service `Trigger Event` peut être comparé à une instruction de trappe ou exception (comme l'instruction `TRAP` des microprocesseurs Motorola 68000 [Vie91]). C'est en effet un processus utilisateur qui décide quand il va effectuer cette trappe. Dans la norme de gestion de réseau CMISE, il n'y a pas à proprement parler de primitive permettant le déclenchement d'un événement. Il est possible de déclencher un événement lors d'une opération de test ("test management function") avec la primitive `M-ACTION` [Sta93]. Mais la notification parvient à l'application qui demande le test et pas à d'autres applications. Toutefois la cause de déclenchement d'un événement dans CMISE n'étant pas normalisée, on peut parfaitement considérer qu'une application de gestion est elle même vue comme un objet géré et déclenche un événement. Dans ce cas, la notification parvient bien à une autre application de gestion.

Dans MMS, l'événement déclenché peut cependant apparaître de façon autonome c'est-à-dire sans l'utilisation du service `Trigger Event`. Dans ce cas, la raison de l'apparition de cet événement n'est théoriquement pas normalisée par MMS et les concepteurs de systèmes MMS sont totalement libres de décider des conditions qui font survenir un tel événement. On dit alors que l'événement est déclenché *de façon locale*. MMS définit toutefois un cas où l'événement est déclenché localement (sans `Trigger Event`) : lors de l'expiration d'un temporisateur signifiant la perte de contrôle d'un sémaphore par un client.

3.3.2 Apparition par scrutation

L'apparition d'un événement scruté procède d'un mécanisme un peu plus complexe que celui des événements déclenchés. Un EC scruté est lié à une variable MMS booléenne appelée *variable scrutée* ("monitored variable"). Ce sont en partie les transitions de VRAI à FAUX et de FAUX à VRAI de cette variable booléenne qui vont constituer les conditions d'apparition de l'événement MMS. L'événement réel correspondant implique la mise à jour de la valeur de cette variable. C'est le dispositif physique modélisé par la VMD qui est normalement responsable de cette mise à jour. On remarque ici que les dispositifs physiques doivent se résoudre à ce comportement c'est-à-dire qu'**ils doivent savoir qu'il faut mettre à jour une variable booléenne et quand la mettre à jour**. Là encore les conditions d'apparition de l'événement sont donc hors du domaine de discours de MMS.

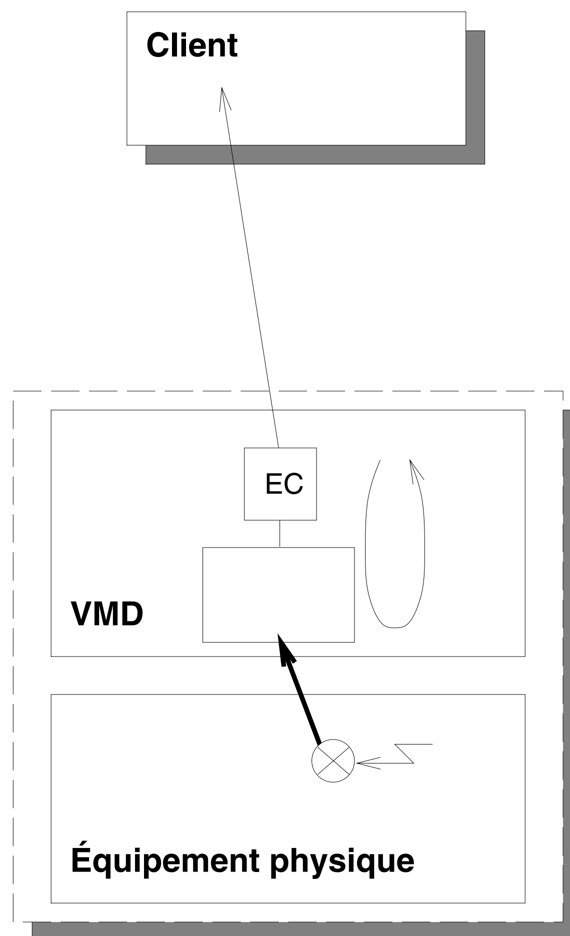


FIG. 3.3 - Evénement MMS scruté

A la valeur VRAI de la variable scrutée correspond la valeur `ACTIVE` de l'attribut `State` de l'EC. Réciproquement, à la valeur FAUX correspond la valeur `IDLE` pour l'EC. Le serveur doit scruter régulièrement la valeur de la variable booléenne et la comparer à l'état de l'EC. C'est en fait lorsque l'état est `ACTIVE` et la variable à FAUX ou que l'état est `IDLE` et la variable à VRAI que l'événement MMS est déclenché. L'attribut `State` de l'EC est alors modifié pour correspondre à la valeur courante de la variable scrutée. Un client MMS voit donc un événement comme étant le résultat d'une transition `ACTIVE-TO-IDLE` ou `IDLE-TO-ACTIVE` d'un objet EC. La figure 3.3 illustre ce comportement.

Toutefois MMS introduit cinq autres transitions pouvant mener à un événement :

DISABLED-TO-ACTIVE et DISABLED-TO-IDLE : surviennent lorsque la scrutation de l'EC considéré est activée par le service `Alter Event Condition Monitoring`. Son attribut `State` passe donc de l'état `DISABLED` à celui reflétant la valeur courante de la variable scrutée associée.

ACTIVE-TO-DISABLED et IDLE-TO-DISABLED : surviennent lorsque la scrutation de l'EC considéré est désactivée par le service `Alter Event Condition Monitoring`. Son attribut `State` passe donc de l'état reflétant la valeur courante de la variable scrutée associée à `DISABLED`.

ANY-TO-DELETED : survient lorsque l'EC ou la variable scrutée sont détruits alors que la scrutation est active et ce quelque soit la raison de cette destruction.

Il est difficile de décider si ces dernières transitions correspondent à des conditions de déclenchement des événements qui sont vraiment en dehors du domaine de discours de MMS. La décision d'utiliser le service `Alter Event Condition Monitoring` est bien propre à une application et n'est pas du ressort de MMS. Celle de détruire un EC avec le service `Delete Event Condition` aussi. Mais MMS a bel et bien normalisé les conséquences de l'exécution de ces services sur les événements.

La figure 3.4 tirée de [Nus91b] illustre quelques conditions d'apparition d'un événement MMS scruté.

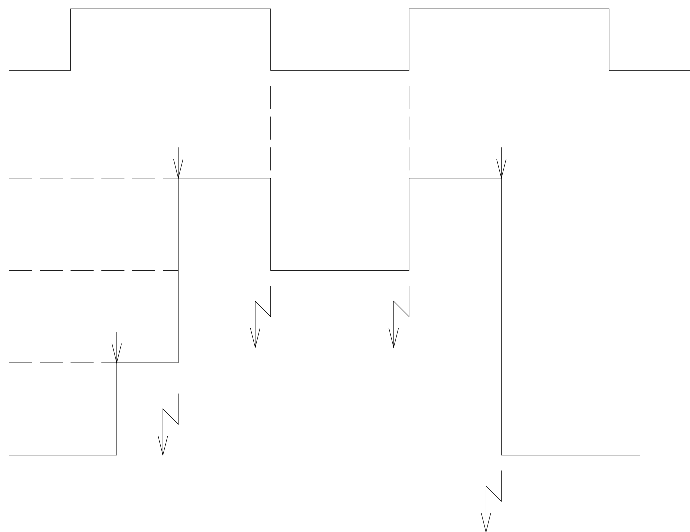


FIG. 3.4 - Exemples de transitions d'événements scrutés

Pour guider un serveur dans sa scrutation périodique des ECs, MMS permet aux clients de fournir explicitement des intervalles de scrutation pour chaque EC. Ils spécifient le temps maximum qui peut s'écouler entre deux consultations d'un EC et de sa variable associée (attribut `Evaluation Interval` fig. 3.1). Les clients peuvent aussi fournir des priorités aux ECs. Celles-ci sont normalement utilisées pour ordonner le traitement des événements qui se sont produits à des instants simultanés ou très proches. L'algorithme d'ordonnancement choisi n'est pas du ressort de MMS mais propre à chaque implantation.

Le nombre d'ECs qu'un serveur peut scruter peut être limité. Il dépend essentiellement des intervalles d'évaluation et des priorités fournis par les clients. Quand un client demande la scrutation d'un EC, un serveur peut être dans un des cas suivants :

1. il peut assurer la scrutation;

2. il est incapable d'assurer cette scrutation;
3. il pourrait assurer la scrutation mais risque de ne plus pouvoir garantir les temps de scrutation d'autres ECs.

Dans les cas 2 et 3, MMS prévoit de retourner une erreur de classe `time-resolution` au client qui demande la scrutation. Cette erreur spécifie que le serveur est incapable de supporter la scrutation. Le client peut alors essayer de fournir un intervalle d'évaluation plus grand et une priorité plus faible.

On retrouve un problème similaire dans le comportement du gestionnaire d'événement NEM ("Network Event Manager") dans le domaine de la gestion de réseaux [CPR86]. Un programme de gestion peut demander au NEM de surveiller un certain nombre d'événements susceptibles de l'intéresser. Ceux-ci sont insérés dans une table de requêtes. Chen *et al.* notent qu'il est important de contrôler la taille de cette table si l'on veut garantir le respect de contraintes de temps dans la détection des événements. Dans NEM, une approche possible consiste à détruire les requêtes dont les délais ne peuvent être garantis. Dans MMS, ceci s'apparente à une désactivation automatique de la scrutation des ECs dont les échéances ne peuvent être remplies. Si un serveur est capable d'évaluer ses capacités à scruter un nouvel EC, il peut refuser une scrutation et retourner une erreur au client qui l'a demandée. Ceci se fait alors dans la réponse du service `Alter Event Condition Monitoring`.

Si l'on suppose qu'un serveur peut calculer avant de débuter la scrutation d'un EC si celle-ci sera supportée alors on se ramène au cas que l'on a cité précédemment : le serveur refuse la scrutation de l'EC et retourne une erreur au client qui l'a demandée.

La scrutation des ECs peut se révéler un problème complexe faisant appel à des techniques d'ordonnancement temps-réel. Nous consacrons une partie du chapitre 6 à cet aspect des événements MMS. La scrutation des événements nécessite donc des capacités de calcul importantes. Ceci n'est pas toujours le cas. On comprend alors facilement pourquoi souvent seul un sous-ensemble des fonctionnalités d'un serveur est implanté dans les produits MMS.

3.4 Création d'une condition événementielle

MMS permet toute liberté dans la création des objets EC. Leur nombre n'est pas limité et les applications clients peuvent définir les conditions événementielles qu'elles désirent. Chaque implantation d'un serveur MMS peut par contre imposer des limitations en fonction de critères qui leur sont propres (comme la mémoire disponible). Ceci s'applique aussi bien aux événements déclenchés que scrutés. La création d'un EC scruté est cependant subordonnée à l'existence de la variable boolénne qui doit lui être associée. L'existence et l'utilité de cette variable dépendent de l'équipement physique sous-jacent à la VMD. Ainsi un EC scruté dépend aussi de cet équipement physique. La présence et la création d'un EC scruté dans la VMD se justifient par la capacité de l'équipement physique à modifier la valeur de la variable booléenne associée. Ceci explique que les ECs de type scruté soient souvent pré-définis lors de la programmation applicative du serveur associé car ils correspondent à des événements réels liés au dispositif physique. Par exemple, une VMD modélisant un robot peut fournir un EC lié à un événement détectant la fin de course d'un bras du robot. Il est possible de définir un EC sans qu'il n'existe d'événement réel correspondant. Mais dans ce cas, il appartient aux clients de provoquer l'événement et non au dispositif physique. On se ramène alors au cas d'un événement déclenché. Nous analysons à la section 3.12 dans quelle mesure ces deux types d'événements se ressemblent.

Quand un client désire créer un EC pour un événement déclenché ou scruté, il le fait au moyen du service MMS `Define Event Condition`. Certains services MMS impliquent la création automatique d'un EC pour des besoins qui leur sont propres. C'est le cas des services `Create Program Invocation` et `Define Semaphore` qui créent respectivement un EC de type scruté et un EC de type déclenché.

On peut comparer les ECs scrutés pré-définis aux interruptions d'un microprocesseur. Les ECs scrutés pré-définis dépendent du dispositif physique modélisé par la VMD. De la même manière, les interruptions peuvent être vues comme des événements pré-définis par la structure matérielle considérée. Comme pour les ECs scrutés pré-définis, leur nombre est limité. En outre la plupart ont une signification bien précise. Certaines interruptions sont toutefois réservées pour que l'utilisateur puisse installer ses propres vecteurs d'interruption.

On peut également comparer la création d'ECs à celle des signaux et événements des systèmes d'exploitation. Les signaux utilisables dans UNIX sont pré-établis et très restreints. On ne peut pas créer un événement au sens de MMS. Seuls deux signaux peuvent être définis par l'utilisateur (valeurs `SIGUSR1` et `SIGUSR2` de l'appel système `signal`) [Bac86]. VAX/VMS autorise une définition plus vaste et plus conviviale des événements [Dig82]. Le service `Associate Common Event Flag Cluster` (`$ASCEFC`) permet la définition d'un groupe de drapeaux ("flags") représentant des événements. Ce service est à rapprocher du service MMS `Define Event Condition`. Les événements VAX/VMS peuvent ensuite être détruits (service `Delete Common Event Flag Cluster` ou `$DLCEFC`) tout comme un client MMS peut détruire un EC avec le service `Delete Event Condition`.

Les événements disponibles dans un dévermineur sont définis par l'utilisateur. Il y en a généralement autant que ce dernier le désire mais seulement dans le cadre des classes d'événements supportés par le dévermineur (point d'arrêt, pression sur `<CTRL-C>`, etc). Les événements d'un système de fenêtrage sont également pré-définis et correspondent à des actions effectuées par l'utilisateur (pression sur une touche du clavier ou sur un bouton souris). Là encore, on peut rapprocher ces types d'événements aux ECs scrutés. Les événements d'un dévermineur correspondent aux ECs créés par l'utilisateur et liés à une variable boolénne pré-définie. Les événements des systèmes de fenêtrage correspondent plutôt à des ECs pré-définis.

3.5 Visibilité d'une condition événementielle

Rappelons que MMS définit la *visibilité* des objets contenus dans un serveur. Il y a trois types de visibilité :

- `vmd-specific` : l'objet est visible dans toute la VMD et donc accessible par tout client;
- `domain-specific` : l'objet n'est visible que dans un domaine et accessible aux seuls clients qui connaissent l'existence de ce domaine;
- `aa-specific` : l'objet n'est visible que sur l'association où il a été créé et donc accessible par le seul client correspondant à cette association.

Un objet EC peut avoir n'importe laquelle des trois visibilités définies par MMS. Dans le cas où la visibilité est `aa-specific`, seul l'EC scruté semble avoir un intérêt. En effet, une application client qui crée un EC déclenché de visibilité `aa-specific` est seule à connaître l'existence de cet EC. Elle est donc seule à pouvoir déclencher l'événement. Aucune autre application ne peut souscrire à cet événement.

Le langage ORE permet à un événement d'avoir une visibilité limitée au seul processus qui le déclare (comme MMS `aa-specific`), une visibilité étendue à un groupe de processus (MMS `domain-specific`) ou une visibilité globale (MMS `vmd-specific`) [DJJ89]. VAX/VMS n'a que deux niveaux de visibilité : les événements locaux à un processus et les événements communs à tous les processus [Dig82]. Comme pour l'événement MMS scruté `aa-specific`, l'événement VAX/VMS local ne s'utilise qu'en conjonction avec le système et non entre processus.

3.6 Notification d'un événement

Pour informer les clients MMS de l'apparition d'un événement, les serveurs utilisent le service non confirmé `Event Notification`. Cette opération de signalisation est donc asymétrique par nature [BDD+84]. Le service `Event Notification` transporte l'information relative à l'événement qui s'est produit. Il contient notamment le nom de l'EC associé, l'heure à laquelle s'est produit l'événement et éventuellement le résultat de l'action associée à cet événement (voir section 3.8). Le service `Event Notification` est commun aux événements scrutés et déclenchés. Selon le formalisme MMS, l'envoi d'une notification correspond à une requête même si pratiquement le message circule du serveur vers un client.

Il existe une forte similitude entre le service `Event Notification` de MMS et le service `M-EVENT-REPORT` de CMISE pour la gestion de réseaux. Ce dernier service est utilisé entre utilisateurs CMISE pour signaler l'apparition d'événements sur un site ou relativement à un objet géré. Le tableau 3.1 compare les caractéristiques de ces deux services. Les notifications d'événement MMS sont des services non confirmés. Le tableau cité montre en particulier comment combiner les acquittements MMS pour concilier l'aspect confirmé des notifications CMISE. Une requête d'acquittement MMS (service `Acknowledge Event Notification`) est comparée à l'indication CMISE `M-EVENT-REPORT`. On peut également faire le rapprochement entre un acquittement MMS et la confirmation d'une notification d'événement entre objets telle que définie dans la norme COSS ("Common Object Services Specification") d'OMG ("Object Management Group") [Gro94]. Là aussi, comme dans CMISE et MMS, cette dernière n'est pas obligatoire.

La façon de gérer les notifications d'événements dans COSS est beaucoup plus générale que dans MMS. L'objet qui envoie une notification est appelé *producteur* ("supplier") de l'événement. Celui qui reçoit *consommateur* ("consumer"). On distingue deux modèles : "push" et "pull". Dans le premier, l'initiative est au producteur car c'est lui qui décide quand envoyer la notification. Dans le second, l'initiative est au consommateur qui demande au producteur de lui fournir l'information relative aux événements survenus. MMS est du type "push" car les événements se produisent de façon inopinée et leurs notifications sont alors envoyées par le serveur vers les clients concernés. Les systèmes de fenêtrage sont généralement du type "pull". Par exemple dans le gestionnaire d'événements Macintosh c'est l'application qui doit consulter le gestionnaire pour connaître l'événement suivant à traiter (fonction `GetOSEvent`) [App85]. Dans la gestion d'événements de CNMA ("Communications Network for Manufacturing Applications"), les événements sont centralisés dans un tampon et ce sont aussi les applications qui doivent consulter cette liste d'événements au moyen de fonctions comme `GET-NEXT-EVENT` par exemple [Con92]. Ce modèle par consultation ne correspond pas directement à MMS mais une application client peut toutefois *percevoir* la réception des notifications d'événements de cette façon en utilisant une interface comme MMSI. Cette interface fournit la primitive `indication_receive` qui permet à une application client de consulter quand elle veut la liste des indications reçues et en particulier les notifications d'événements [Gen88] Vol. IV.

Dans CMISE, l'information transportée par une notification (paramètre `Event Information` dans la table 3.1) n'est pas normalisée et peut être aussi complexe que nécessaire. Il en va de même pour l'information transportée par les notifications des objets COSS [Gro94]. Le format des notifications MMS est par contre fixé par la norme. Mais il est possible de transporter de l'information supplémentaire en utilisant les actions associées aux événements avec par exemple un service `Read` complexe impliquant la lecture de plusieurs variables.

Caractéristiques CMISE	Equivalents MMS
La cause d'apparition de l'événement est hors du domaine de discours de CMISE	La cause d'apparition de l'événement est hors du domaine de discours de MMS
Requête/Indication `M-EVENT-REPORT`	Requête/Indication `Event Notification`
Réponse/Confirmation `M-EVENT-REPORT`	Requête/Indication `Acknowledge Event Notification`
-	Réponse/Confirmation `Acknowledge Event Notification`
`Invoke Identifier`	Non présent car la notification MMS n'est pas confirmée. C'est la combinaison des paramètres de la requête d'acquittement qui permet de faire correspondre de façon unique cet acquittement avec une notification précédente
`Mode` (permet de savoir si la notification est confirmée)	L'attribut `Alarm Acknowledgment Rule` permet de savoir si des acquittements sont requis mais uniquement pour les événements scrutés
`Managed Object Class`	Inutile, la notification provient toujours d'un objet EC
`Managed Objet Instance`	Nom de l'objet EC correspondant
`Event Type`	-
`Event Time` (temps d'apparition de l'événement)	L'attribut `Transition Time` de la notification a la même signification
`Event Information` (information non normalisée et propre à chaque application)	Pourrait s'apparenter aux résultats de l'action associée. Toutefois le format de ces résultats est normalisé dans MMS
`Current time` (heure de la génération de la réponse `M-EVENT-REPORT`)	-
`Event Reply`	-
`Errors`	-

TAB. 3.1 - Comparaison des notifications d'événements CMISE et MMS

3.7 Souscription à un événement

Une application client souscrit à un événement en créant dans le serveur un objet *Enregistrement Evénementiel* (EE ou “Event Enrollment”) au moyen du service `Define Event Enrollment`. La définition d'un EE est donnée sur la figure 3.5. On remarque en particulier que cet objet contient un lien vers l'objet EC correspondant à l'événement auquel le client souscrit (attribut `Event Condition Reference`) ainsi que l'adresse du client qui doit recevoir la notification (`Client Application`). Un EE créé de cette façon est de classe `NOTIFICATION` (son attribut `Enrollment Class` prend la valeur `NOTIFICATION`). Plusieurs clients peuvent souscrire au même événement donc il peut y avoir plusieurs EEs liés au même EC. L'apparition de l'événement provoque alors un envoi de notifications à chaque client représenté par un de ces EEs.

Un client peut ainsi se mettre en attente d'un événement en définissant un EE pour lui-même de la même façon que dans les systèmes UNIX avec l'appel `wait` [Bac86], VAX/VMS avec l'appel `Wait for Single Event Flag ($WAITFR)` [Dig82], ORE avec l'instruction `set watching` [DJJ89] ou encore COSS avec l'interface `ConsumerAdmin` [Gro94]. Quand un client MMS définit un EE pour lui-même, l'attribut `Client Application` de l'objet EE n'est pas utilisé. En effet, seul un identificateur de l'association entre le serveur et le client suffit pour envoyer la notification (attribut `Application Association Local Tag`).

Dans [Sch86] pp. 72-74, Schiper définit un événement comme une variable structurée contenant deux champs : un booléen qui permet de savoir si l'événement est survenu et une liste des processus en attente sur l'événement. Cette dernière liste est comparable à la liste des EEs identifiant les clients souscrivant à un événement dans MMS (attribut `List of Event Enrollment References` fig. 3.1).

Il existe toutefois une différence notable entre l'événement structuré défini ci-dessus et les événe-

```
Object: Event Enrollment

  Key Attribute: Event Enrollment Name
  Attribute: MMS Deletable (TRUE, FALSE)
  Attribute: Enrollment Class (MODIFIER, NOTIFICATION)
  Attribute: Event Condition Reference
  Attribute: Event Condition Transitions (DISABLED-TO-ACTIVE,
             DISABLED-TO-IDLE, IDLE-TO-ACTIVE, IDLE-TO-DISABLED,
             ACTIVE-TO-IDLE, ACTIVE-TO-DISABLED, ANY-TO-DELETED)
  Attribute: Application Association Local Tag
  Constraint: Enrollment Class = MODIFIER
        Attribute: InvokeId
        Attribute: Remaining Acceptable Delay
  Constraint: Enrollment Class = NOTIFICATION
        Attribute: Notification Lost (TRUE, FALSE)
        Attribute: Event Action Reference
        Attribute: Duration (CURRENT, PERMANENT)
        Attribute: Client Application
        Attribute: Additional Detail
        Attribute: Alarm Acknowledgment Rule (NONE, SIMPLE,
                   ACK-ACTIVE, ACK-ALL)
        Attribute: Time Active Acknowledged
        Attribute: Time Idle Acknowledged
        Attribute: State (DISABLED, IDLE, ACTIVE, IDLE-NO-ACK-I,
                   IDLE-NO-ACK-A, ACTIVE-NO-ACK-A, ACTIVE-ACKED,
                   IDLE, ACKED)
```

FIG. 3.5 - L'objet Enregistrement Evénementiel

ments MMS. Dans le premier cas, les processus qui sont mis en attente de l'événement sont suspendus sur l'événement. L'attente est donc bloquante et la souscription à un événement se confond avec l'attente de sa notification. Dans MMS, l'attente d'un événement n'est pas nécessairement bloquante. En fait, la façon dont une application client MMS est informée de la réception d'un service n'est pas normalisée par MMS. Ce sont les interfaces situées entre le fournisseur de services MMS et l'application client qui définissent la perception qu'à le client de l'envoi et de la réception des services MMS. L'interface MMSI propose des primitives bloquantes ou non bloquantes (comme `indication_receive` que nous avons déjà mentionné). Selon la primitive utilisée, les applications MMS peuvent ou non s'exécuter. Dans le cas non bloquant, elles peuvent continuer d'effectuer des requêtes de service alors même qu'elles sont en attente d'une notification. C'est d'ailleurs tout l'intérêt de **souscrire** à un événement par opposition à **attendre** un événement. Le cas se retrouve dans COSS pour la réception des notifications d'événements avec l'opération bloquante `pull` et l'opération non bloquante `try_pull` [Gro94]. Le système d'exploitation temps-réel pSOS$^+$ propose aussi aux processus en attente d'un événement deux alternatives : soit rester bloqués, soit faire une scrutation périodique.

Dans le domaine de la Programmation Orientée-Objet (POO) Bacon *et al.* ont défini dans [BBHM95] une extension pour un langage de définition d'interfaces (IDL ou “Interface Definition Language”). Cette extension permet la manipulation d'événements entre objets client et objets serveurs et se comporte de façon identique à la gestion d'événements MMS. En particulier des objets clients peuvent souscrire à des événements au moyen de la méthode `Register`. Seuls les clients ayant souscrit à une classe d'événement recevront les notifications de ces événements. Cette souscription se fait par créa-

tion d'un objet appelé "event template". Un objet client a la possibilité d'annuler sa souscription tout comme un client MMS peut détruire l'objet EE au moyen du service `Delete Event Enrollment`.

Enfin il faut noter que MMS permet la possibilité intéressante suivante : un client peut au besoin enregistrer un autre client à un événement dans la mesure où il connaît son adresse et la fournit explicitement lors du service `Define Event Enrollment`. Le serveur enverra alors les notifications à cet autre client et non à celui qui a créé l'EE.

3.8 Exécution d'actions liées aux événements

MMS donne la possibilité d'exécuter des actions lors de l'occurrence d'un événement. Ces actions sont locales au serveur et ne sont autres que des services MMS confirmés pré-définis pour cet événement et propres à chaque client. Une action événementielle (EA ou "Event Action") est représentée par un objet du même nom (fig. 3.6). La création de cet objet se fait au moyen du service `Define Event Action`. Le lien entre l'EA et l'EC représentant l'événement considéré se fait par un objet EE créé lors de la souscription à l'événement. L'occurrence de l'événement entraîne automatiquement l'exécution des actions associées. Les résultats d'une action sont retournés aux clients concernés dans la notification d'événement.

```
Object: Event Action

  Key Attribute: Event Action Name
  Attribute: MMS Deletable (TRUE, FALSE)
  Attribute: Confirmed Service Request
  Attribute: List Of Modifier
  Attribute: List Of Event Enrollment Reference
  Attribute: Additional Detail
```

FIG. 3.6 - L'objet Action Evénementielle

On retrouve là un comportement un peu identique aux interruptions d'un microprocesseur qui lorsqu'elles se produisent entraînent l'exécution de la routine dont l'adresse est spécifiée par le vecteur d'interruption. Le langage ORE [DJJ89] offre également une possibilité semblable : un "strip" représente une partie de code exécutée lors de l'apparition d'un événement. L'opération `bind` permet de lier un "strip" avec l'événement pour un processus donné comme le service `Define Event Enrollment` permet de lier un EC à un EA. Le modèle événement/action se retrouve aussi dans les systèmes de supervision ou de déverminage d'applications. L'utilisateur peut alors être vu comme le client MMS, le programme dévermineur comme le serveur MMS et l'application testée comme l'équipement modélisé par la VMD. Dans [LSM90] par exemple, l'utilisateur peut décider d'une action à effectuer lors de l'apparition d'un événement dans l'application supervisée (incrémenter un compteur quand la valeur d'un registre vaut 1 par exemple). Ces actions peuvent éventuellement affecter l'état de l'application supervisée et sont alors appelées *actions intrusives*. De façon similaire, dans MMS une action peut être intrusive dans le sens où elle peut induire un changement dans l'équipement modélisé par la VMD. L'extension définie par Bacon *et al.* dans [BBHM95] permet à un objet client de fournir le nom d'une méthode devant être appelée lors de l'apparition d'un événement. Toutefois cette méthode appartient au client et donc s'exécute dans le client contrairement à MMS où l'action s'exécute dans le serveur.

Le système d'exploitation temps-réel VRTX synchronise les processus par boîtes aux lettres [Rea86].

L'appel SC_PEND met un processus en attente sur une boîte aux lettres alors que l'appel SC_POST réveille ce processus. L'avantage des boîtes aux lettres est de pouvoir passer des données lors des signalements entre processus. On peut comparer la réception des données à celles des notifications MMS contenant le résultat d'actions. Mais l'action n'est pas exécutée dans le noyau et des données sont simplement passées d'un processus à un autre. Dans MMS, il n'est pas possible de passer des données par le service Trigger Event contrairement à l'appel SC_POST de VRTX. Le gestionnaire de réseaux NEM propose trois types d'actions sur apparition d'un événement : envoyer une lettre ("mail") à un utilisateur, envoyer un message à un processus ou exécuter une commande donnée [CPR86]. Les deux premiers types ne sont pas réalisables dans MMS car l'action événementielle MMS est toujours locale au serveur. Toutefois ces deux types d'action se comparent simplement à l'envoi d'une notification d'événement MMS. Le troisième type est plus proche de MMS puisqu'il s'agit de l'exécution d'une commande. Cependant lors de l'événement, certains paramètres peuvent être passés à la commande ce qui n'est pas le cas dans MMS.

L'exécution des actions événementielles se fait de la même façon que celle des services MMS. Il n'y a pas de différence entre les deux mis à part l'utilisation des résultats des exécutions. Pour les requêtes de service les résultats sont simplement retournés aux clients dans la réponse correspondante. Pour les actions, ceux-ci font l'objet d'un traitement ultérieur pour les associer à une notification d'événement. Le client reçoit donc ces résultats avec la notification correspondante. Cette similitude fait qu'une action comme une requête de service peut contenir des modificateurs. L'exécution de l'action et donc l'envoi de la notification associée se trouvent alors conditionnés par la satisfaction de tous les modificateurs. Ces modificateurs sont fournis lors de la définition de l'EA par le service Define Event Action.

3.9 Filtrage des événements scrutés

L'apparition d'un événement n'implique pas nécessairement une notification à tous les clients correspondant aux EEs de classe NOTIFICATION liés à l'EC qui représente cet événement. En fait, les notifications ne parviennent qu'aux clients qui ont souscrit à la transition qui a eu lieu. Les transitions auxquelles souscrivent les clients sont contenues dans l'attribut Event Condition Transitions des EEs. Ces transitions sont pré-définies (voir section 3.3.2). Lors du traitement d'un événement, chaque EE est examiné. Si la transition d'événement qui s'est produite fait partie des transitions apparaissant dans l'EE alors le traitement se poursuit et génère une notification. Dans le cas contraire aucune notification n'est envoyée. Comme le souligne Nussbaumer dans [Nus91b] pp. 309-311, il y a donc une action de *filtrage* des événements MMS. Ce filtrage se fait par comparaison aux sept transitions pré-définies. La VMD détecte des *événements potentiels* qui sont filtrés par les EEs pour donner des *événements effectifs* (fig. 3.7). Par exemple si une VMD modélise un capteur de température, pour détecter le franchissement d'un certain seuil de température on utilise la transition IDLE-TO-ACTIVE qui est la seule à donner naissance à l'événement effectif souhaité. Les autres transitions ne sont pas intéressantes dans ce cas.

Les événements effectifs autorisent la poursuite du traitement d'un service MMS si l'EE correspondant est de classe MODIFIER ou donnent lieu à une notification si l'EE correspondant est de classe NOTIFICATION. Il faut toutefois noter que pour un EE de classe NOTIFICATION, un événement effectif ne produit pas toujours une notification. Ceci peut être le cas lors d'une impossibilité d'établir une association entre le serveur et le client ou pour des raisons propres au serveur. MMS prévoit dans ces cas de mettre à VRAI l'attribut Notification Lost des EEs correspondant aux événements effectifs. Cet attribut est reporté dans le service Event Notification. Il permet donc aux clients concernés de savoir que des notifications ont été perdues quand les conditions autorisant l'envoi des notifications sont rétablies. L'utilisation du service Report Event Enrollment Status permet également d'obtenir

la valeur de l'attribut `Notification Lost`.

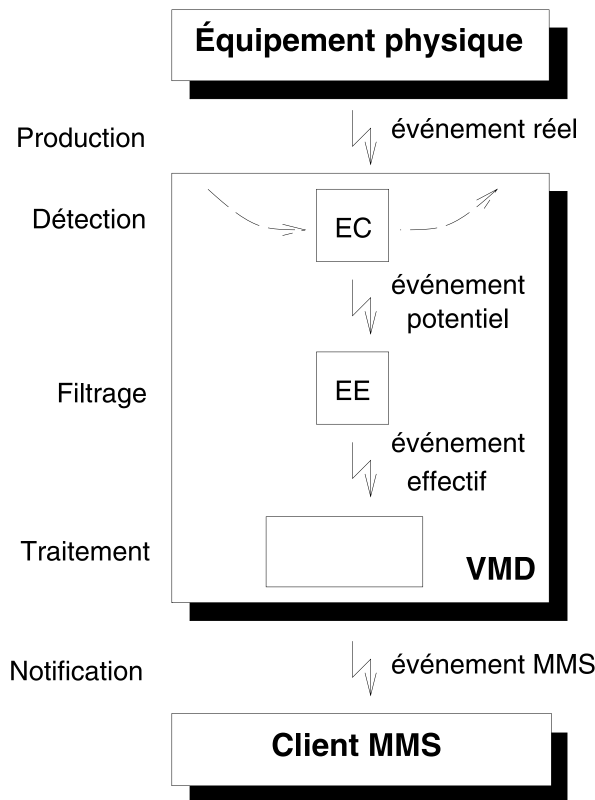


FIG. 3.7 - Filtrage des événements dans MMS

Le filtrage est une action que l'on retrouve dans la gestion de réseaux notamment dans CMISE [IC91]. Son but essentiel est d'éviter la saturation du réseau par des événements peu importants ou triviaux et de permettre de rapporter les événements jugés plus critiques [Bla92]. Dans CMISE, les *objets gérés* ("managed objects") envoient des notifications d'événements à des *processus agents* ("process agents"). Ceux-ci filtrent les événements au moyen d'un *discriminant* ("event forwarding discriminator"). Seuls les événements filtrés par le discriminant donnent lieu à une notification vers les *processus gestionnaires* ("managing processes"). On peut faire le rapprochement avec MMS en comparant l'équipement physique à l'objet géré, la VMD ou le serveur au processus agent et le client au processus gestionnaire. CNMA propose également une action de filtrage des événements et permet la création d'un filtre par la fonction `CREATE-EVENT-FILTER`. Un tel filtre peut être basé sur un intervalle de temps de sorte que seuls les événements survenus dans cet intervalle passent le filtre [Con92]. MMS ne permet pas d'avoir des filtres complexes comme CNMA et encore moins de conditionner l'apparition d'un événement sur une durée ou sur un instant précis.

Dans le modèle COSS, les objets peuvent s'envoyer des notifications d'événements qui sont filtrées par les canaux de communication entre objets [Gro94]. Dans certains systèmes d'exploitation comme celui des ordinateurs Macintosh par exemple on retrouve aussi la possibilité de masquer certains événements de sorte que le gestionnaire d'événements n'en prennent que certains en compte (fonction `SetEventMask`) [App85]. A un plus bas niveau, le masquage d'interruptions sur un microprocesseur procède également de la même idée. Dans CNMA, les fonctions `EVENT-GENERATOR-OFF` et `EVENT-GENERATOR-ONLINE` permettent respectivement la désactivation et ré-activation de la génération d'événements [Con92]. Ces trois derniers cas s'apparentent toutefois plus au masquage/démasquage de la détection d'un événement scruté par le service MMS `Alter Event Condition Monitoring`. Quand la scrutation d'un EC est masquée ("`DISABLED`"), il ne peut même pas surgir d'événement potentiel

lié à cet EC.

Dans le système d'exploitation VAX/VMS, il est possible de n'envoyer un signal à un processus qu'après l'apparition d'un ensemble d'événements décrit par une combinaison logique d'événements [Dig82]. L'appel `Wait for Logical OR of Event Flags ($WFLOR)` place un processus en état d'attente jusqu'à l'apparition d'un événement quelconque spécifié par cet appel. L'appel `Wait for Logical AND of Event Flags ($WFLAND)` place un processus en état d'attente jusqu'à l'apparition de tous les événements spécifiés dans cet appel. De même dans le domaine de la POO, l'extension proposée dans [BBHM95] permet aux objets de définir des événements composés. Un événement composé provoque une notification lorsqu'une combinaison d'événements plus simples se produit. Le filtrage MMS, très limité, ne permet pas de faire de telles combinaisons logiques d'événements réels.

Le filtrage MMS est encore plus limité par rapport à celui que l'on trouve dans CMISE. Dans MMS, le filtrage s'effectue en comparant la transition d'événement survenue avec les transitions autorisées (au maximum sept). Définir les conditions de déclenchement d'un événement MMS scruté ou définir le filtrage d'un événement potentiel revient au même. L'apparition d'un événement MMS scruté est essentiellement basée sur la valeur d'une variable booléenne. Dans CMISE, le filtrage est plus évolué en ce sens qu'il est basé sur une expression prédicat librement choisie par l'utilisateur. Nous discutons de cette limitation de MMS dans la section 3.11.1. Au chapitre 5, nous proposons une solution de niveau de modification 2 pour améliorer le filtrage dans MMS.

Enfin, il faut noter que dans MMS le filtrage par les EEs ne se produit que pour les événements scrutés. Le déclenchement d'un événement au moyen du service `Trigger Event` produit toujours un événement effectif. La norme MMS spécifie alors que l'attribut `Event Condition Transitions` des EEs liés à un EC de type `NETWORK-TRIGGERED` doit être vide.

3.10 Conditionnement d'un service sur l'apparition d'un événement

L'utilisation du modificateur `Attach To Event Condition` dans une requête de service permet de conditionner l'exécution de ce service sur l'apparition d'un événement. La figure 3.8 illustre le fonctionnement d'un service `Read` modifié. Comme une requête de service peut contenir plusieurs modificateurs, celle-ci peut se mettre en attente de plusieurs événements avant son exécution effective. L'événement sur lequel se fait l'attente est quelconque c'est-à-dire soit scruté soit déclenché. Le traitement d'un modificateur `Attach To Event Condition` entraîne la création automatique d'un EE dont l'attribut `Enrollment Class` prend la valeur `MODIFIER`. Ceci signifie que le traitement de cet EE n'implique pas l'envoi d'une notification mais doit signaler au serveur qu'il peut reprendre le traitement du service modifié.

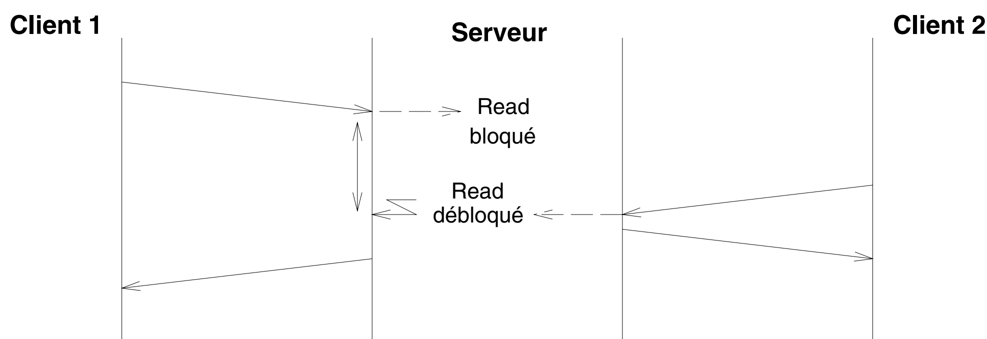


FIG. 3.8 - Modification d'un service MMS

Un service modifié par un modificateur `Attach To Event Condition` spécifiant un objet EC est presque équivalent à une action événementielle contenant le même service et liée au même EC. La table 3.2 effectue la comparaison de ces deux configurations. Une différence majeure est l'impossibilité d'exécuter plusieurs fois le service modifié. Quand l'événement attendu survient, l'EE est détruit. Le client qui désire exécuter ce même service si l'événement se reproduit doit de nouveau effectuer la requête de service modifiée. Ce n'est pas le cas pour les actions événementielles qui restent définies jusqu'à leur destruction explicite. Par ailleurs, l'information nécessaire à la gestion d'une requête de service modifiée reste stockée dans le serveur et dans la MMPM. En d'autres termes, une telle requête utilise les ressources disponibles et en particulier diminue le nombre de requêtes de service qui peuvent être envoyées sur la même association. Ce nombre est négocié lors de l'ouverture de l'association (paramètre `Negociated Max Ser Outstanding Called`). Si le client envoie autant de requêtes modifiées, il restera dans l'impossibilité de faire toute autre requête de service sur cette association tant qu'un événement débloquant un modificateur ne sera pas survenu. Cette situation n'apparaît pas avec l'utilisation des actions événementielles.

Caractéristiques	**Modificateurs**	**Actions événementielles**
Définition préalable des objets événementiels	EC et EE	EC, EE et EA
Définition de l'EC automatique	non	non
Définition de l'EE automatique	oui	non
Définition de l'EA automatique	pas d'EA	non
Nombre de services nécessaires	2	3 ou 4
Possibilité de réutilisation	non, destruction de l'EE	oui, tous les objets restent définis
Attente sur plusieurs événements avec le même service	oui, en utilisant plusieurs modificateurs	non
Combinaison avec attente de libération d'un sémaphore	oui, en utilisant le modif. `Attach To Semaphore`	non
Attentes de plusieurs services sur le même événement	oui	oui avec plusieurs EAs
Résultats du service	dans la réponse	dans la notification
Annulation du service par requête d'un client	`Cancel`, seulement par le client qui a effectué la requête modifiée	`Delete Event Enrollment` et `Delete Event Action` par n'importe quel client
Annulation par expiration d'un temporisateur	oui	sans objet

TAB. 3.2 - Comparaison de l'utilisation de modificateurs et d'actions événementielles

Les mesures faites avec notre implantation montrent que la quantité de données supplémentaires transmises lors de l'utilisation d'un modificateur dans une requête de service est minime par rapport au même service sans modificateur. Pour un service `Read` sur une variable entière le rapport du nombre d'octets transmis pour le cas sans modificateur sur le cas avec modificateur est de 0.85 (requêtes seules). L'addition du modificateur nécessite 23 octets supplémentaires. Il faut 150 octets pour envoyer une requête de service `Read` modifiée. Il en faut 501 pour créer l'EA contenant le même service `Read` et l'EE correspondant[2] (requêtes et réponses). Soit un rapport de 0.3. La réponse du `Read` modifié (identique à celle d'un `Read` sans modificateur) prend 111 octets. La notification contenant la réponse du service `Read` nécessite 164 octets. Le rapport entre les deux est de 0.68. En tout, le rapport (noté R_{evt}) du cas avec utilisation d'un modificateur sur le cas avec action événementielle est de 0.39. Si une application client n'est intéressée que par l'exécution d'un service lors de l'apparition unique d'un événement, il est donc plus intéressant en terme de données transmises sur le réseau, d'utiliser un modificateur qu'une

[2]Ces mesures ne prennent pas en compte les messages autres que MMS comme les acquittements de la couche transport.

action événementielle. Ce n'est qu'à partir de la sixième occurrence de l'événement que l'utilisation de l'action événementielle s'avère plus performante car la nécessité de renvoyer la requête de service modifiée après chaque événement pénalise la solution avec modificateur (fig. 3.9). Bien sûr, ces résultats dépendent de la taille des PDUs MMS donc des services effectués et de la syntaxe de transfert utilisée (ici BER (Basic Encoding Rules) [ISO88]).

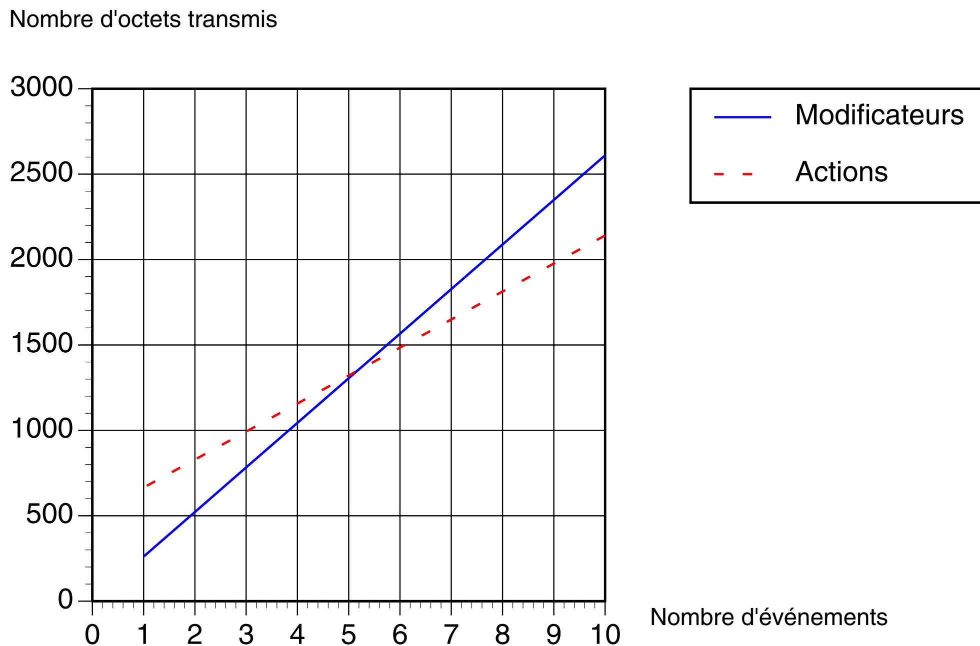


FIG. 3.9 - Comparaison des solutions avec modificateur et avec action pour un service `Read`

Pour vérifier l'influence de la taille des PDUs, nous avons effectué les mesures illustrées par la figure 3.10. Cette figure montre comment varie le rapport R_{evt} en fonction du nombre d'octets présents dans la réponse du service `Read`[3] selon la fréquence d'apparition de l'événement. Il est possible de faire varier le nombre d'octets de la réponse en fixant celui de la requête en supposant par exemple que le `Read` se fait sur une liste nommée de variables MMS `Named Variable List`. L'augmentation des octets dans la réponse correspond à une augmentation du nombre des variables présentes dans la liste. La liste reste identifiée par un seul nom d'où la taille fixée de la requête.

Sur cette figure, on retrouve le retournement de situation favorable aux actions événementielles quand le nombre d'occurrences de l'événement est supérieur à 5 et ce quelque soit le nombre d'octets dans la réponse. Pour 5 occurrences de l'événement, R_{evt} est stable et presque fixé à 1. Les deux solutions s'équivalent alors et ce toujours indépendamment du nombre d'octets. On remarque aussi que plus le nombre d'octets est grand, plus les courbes tendent vers 1 donc plus les deux solutions tendent à être identiques en terme du nombre total d'octets transmis. Enfin, on remarque que la fréquence d'apparition de l'événement n'influe que très peu sur R_{evt} dès que cette fréquence est supérieure à 5. En effet, les courbes pour 15 et 100 apparitions de l'événement sont très proches par rapport à celles pour 1 et 3 apparitions, beaucoup plus éloignées.

Il est possible de lier un modificateur à un temporisateur. Le décompte du temporisateur est démarré dès que l'EE est créé. Le temps restant est contenu dans l'attribut `Remaining Acceptable Delay` de l'EE (fig. 3.5). Si l'événement ne survient pas dans le laps de temps spécifié, l'EE est détruit et la requête de service modifiée est automatiquement annulée. Cette façon de faire n'est pas propre à MMS. Elle se retrouve par exemple dans les gestionnaires d'événements pour la gestion de réseaux.

[3] Dans la réponse proprement dite pour le cas avec modificateur, dans la notification pour le cas avec action.

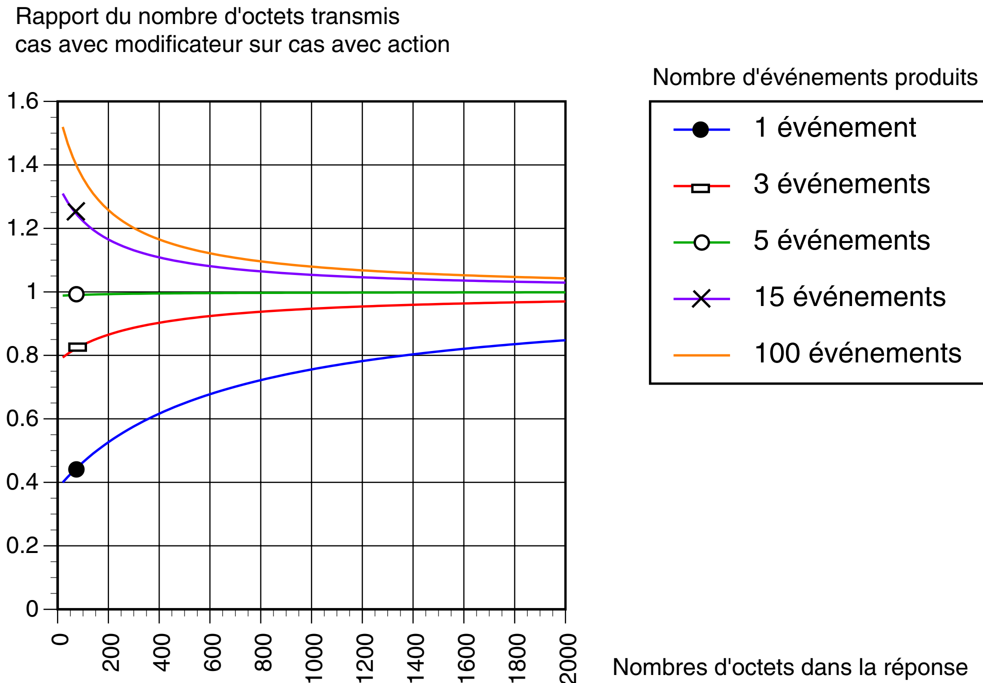


FIG. 3.10 - Mesure de R_{evt} en fonction du nombre d'octets dans la réponse Read selon la fréquence de l'événement

Dans NEM [CPR86], les applications de gestion peuvent se mettre en attente d'événements tels que "`nombre d'utilisateurs sur litsun.epfl.ch > 10`". Pour éviter une accumulation des requêtes d'attente, on peut associer un temporisateur à ces requêtes. Celles-ci sont automatiquement détruites quand le temporisateur associé arrive à expiration.

Les processus gérés par le noyau temps-réel pSOS$^+$ peuvent également associer une durée à l'attente d'un événement. Quand cette durée est écoulée, le noyau remet automatiquement le processus dans l'état prêt [Tho90]. Un processus du noyau Sceptre peut aussi associer une durée à l'attente d'un événement [BDD+84]. Mais cette opération se traduit en fait par la définition de deux événements : la fin de temporisation et l'événement lui même. L'avantage est de pouvoir traiter séparément les deux et d'annuler la temporisation avant sa fin par exemple. Dans un modificateur MMS avec temporisateur, on n'attend qu'un seul événement et il n'est pas possible d'annuler la temporisation sans annuler le service en attente sur le modificateur. Pour arrêter la temporisation il faut donc envoyer le même service avec modificateur sans l'option de temporisation puis annuler la première requête au moyen du service `Cancel`.

3.11 Limitations des événements MMS

3.11.1 Détection basée sur un prédicat

Si les clients MMS sont libres de définir autant d'événements que nécessaire, ils ne peuvent eux-mêmes décider de façon précise de la condition de déclenchement de ces événements. La condition de déclenchement d'un événement réel scruté dépend de chaque équipement modélisé sous-jacent à la VMD et s'y trouve pré-définie sans possibilité (via MMS) de la modifier. Cette condition est donc normalement hors du domaine de discours de MMS. Mais ceci limite considérablement l'intérêt des événements scrutés. En effet, les applications client n'ont pas la possibilité de déclencher des

événements basés sur des expressions prédicats du type “`Température` > 500” et plus complexes encore.

Pourtant, à part le filtrage de CMISE dont nous avons déjà parlé, on retrouve cette idée dans de nombreux systèmes tel NEM également pour la gestion de réseaux où un utilisateur peut demander à recevoir une notification pour un événement du type “`charge du serveur litsun.epfl.ch` > `3.0`” [CPR86]. Une telle syntaxe se retrouve dans le langage ORE avec l'instruction “`when <predicat> do <action>`” [DJJ89]. Dans [LSM90], les auteurs proposent un langage pour faciliter le déverminage et le suivi d'applications distribuées. Ce langage permet permet aux utilisateurs de définir sous forme de prédicat les événements intéressants. L'intérêt d'adopter une telle syntaxe pour les événements MMS a également été souligné dans [Ple90].

On peut remarquer qu'il est possible qu'un client MMS demande à recevoir une notification après qu'une combinaison logique d'événements se soit produite. L'utilisation de modificateurs `Attach To Event Condition` permet en effet à un client de définir un événement du type $E = (E_0$ et E_1 et ... et $E_N)$ où les E_i $(0 \leq i \leq N)$ sont des événements qui doivent apparaître dans l'ordre. Nous proposons ici deux solutions totalement supportées par MMS c'est-à-dire de niveau de modification 1.

1. **Modificateurs dans les actions événementielles** : soit un EA contenant N modificateurs. Le i^{eme} modificateur spécifie l'événement E_i $(1 \leq i \leq N)$. E_0 est l'événement dit principal. Son apparition déclenche l'exécution de l'action. Mais comme l'EA contient N modificateurs, l'apparition de E_0 ne suffit pas pour que l'action s'exécute. L'action et la notification restent bloquées jusqu'à l'apparition dans l'ordre des N événements E_i $(1 \leq i \leq N)$ spécifiés dans les modificateurs. Après occurrence de l'événement E_N, l'action est exécutée et la notification correspondant au premier événement E_0 est envoyée. La notification de l'événement $E = (E_0$ et E_1 et ... et $E_N)$ correspond donc à celle de E_0.

2. **Modificateurs dans une requête de service** : une solution à peine plus simple consiste à utiliser les modificateurs spécifiant les événement E_i $(1 \leq i \leq N)$ dans une requête de service MMS (et non dans une action). La réponse fait alors office de notification après que les N événements E_i se soient produits séquentiellement. Cependant, la réponse ne peut parvenir qu'au client qui a effectué la requête correspondante. Dans le cas précédent, la notification peut parvenir à n'importe quel client et même à plusieurs clients à la fois.

L'ordre imposé par les modificateurs sur la réalisation des événements MMS fait que les “ET” de l'événement $E = (E_0$ et E_1 et ... et $E_N)$ sont plutôt des “ET PUIS” non commutatifs au sens de l'instruction Ada “AND THEN” par exemple [Ins83].

Ces deux solutions sont à comparer avec l'appel système de VAX/VMS `Wait for Logical AND of Event Flags ($WFLAND)` dont nous avons déjà parlé. Le langage proposé dans [LSM90] permet la définition d'un événement à partir de l'occurrence dans un certain ordre d'autres événement. L'expression “e_1 ::= (e_2 => e_3 => e_4)” impose la réalisation des événements e_2 puis e_3 puis e_4 pour déclencher l'événement composé e_1. Ce langage propose ainsi des opérateurs permettant de définir des événements basés sur l'occurrence séquentielle, strictement séquentielle ou non ordonnée d'événements. Les occurrences strictement séquentielles interdisent l'apparition d'événements intermédiaires entre la réalisation de deux événements successifs apparaissant dans la définition d'un événement composé.

Dans MMS, il n'est toutefois pas possible de faire mieux qu'une série de “ET PUIS” dans la complexité de la définition d'un événement. De plus chaque événement E_i doit être un événement au sens MMS du terme c'est-à-dire qu'il ne peut surgir suite à la réalisation d'un prédicat défini par un client.

La définition des événements MMS introduit donc de réelles limitations sur les conditions d'appa-

rition d'un événement que l'on peut résumer ainsi :

- les événements scrutés dépendent de la valeur d'une simple variable booléenne ce qui est trop restrictif;
- les utilisateurs ne peuvent pas définir eux-mêmes la condition de déclenchement d'un événement;
- les dispositifs physiques doivent modifier eux-mêmes la valeur de la variable scrutée, c'est-à-dire qu'ils doivent connaître les conditions de déclenchement des événements;
- les types d'événements disponibles sont donc limités à ceux pré-définis par les dispositifs physiques;
- les dispositifs physiques doivent avoir été conçus pour une utilisation particulière et se conformer au comportement prescrit par MMS.

Ces problèmes seront revus et discutés en détail dans le chapitre 5. Nous proposerons dans ce chapitre une extension à MMS de niveau de modification 2 pour surmonter les limitations citées ci-dessus.

Il doit être clair que lorsque nous parlons de limitation dans la définition des événements MMS, nous entendons limitation dans l'utilisation des services MMS existant c'est-à-dire **limitation de la vision et des possibilités offertes aux applications client**. Les dispositifs physiques et informatiques modélisés par une VMD peuvent évidemment définir n'importe quelle condition de déclenchement d'un événement. **Cette condition n'est pas dans le domaine de discours de MMS et n'est donc pas accessible aux applications client.**

Cette remarque nous amène à nous poser légitimement la question de l'utilité de l'existence des événements scrutés puisque toute condition de déclenchement d'un événement peut être incluse et traitée dans le dispositif physique modélisé par la VMD. Les conditions de transition des événements de classe `MONITORED` et donc le filtrage d'un événement potentiel peuvent eux-mêmes être pris en charge hors de la VMD. L'événement résultant peut alors être mis en correspondance avec un événement MMS de classe `NETWORK-TRIGGERED`. En effet, MMS précise qu'un événement de classe `NETWORK-TRIGGERED` peut être déclenché de façon purement locale c'est-à-dire sans le service `Trigger Event` et qu'il peut lui correspondre n'importe quelle condition de déclenchement y compris donc les mêmes que celles d'un événement de classe `MONITORED`.

Pourquoi alors les conditions d'apparition des événements ne sont-elles pas totalement hors du domaine de discours de MMS comme cela est le cas dans CMISE ? Dans [EC86], Elloy *et al.* semblent d'ailleurs d'avis que l'évaluation des conditions événementielles ne devraient pas entrer dans le cadre de la normalisation des services MMS. Pourquoi introduire dans le domaine de discours une façon aussi limitée de définir un événement et pas un modèle de définition plus complet basé par exemple sur des prédicats proposés par les applications client ? Puisque MMS fait timidement entrer la définition d'apparition d'un événement dans le domaine de discours de MMS, le propos du chapitre 5 sera d'aller encore plus loin en autorisant un client à définir de façon complète et détaillée ces conditions d'apparition.

3.11.2 Ordonnancement

Il est dommage que la norme MMS n'offre aucune garantie quant à l'ordre de traitement des EEs liés à un EC lors de l'apparition de l'événement correspondant. La conséquence directe est qu'il n'est pas possible de donner un ordre d'exécution des actions événementielles associées à un événement.

L'impossibilité d'ordonner les EEs limite également l'intérêt d'utiliser les modificateurs. En effet, un client ne peut pas demander l'exécution ordonnée d'une série de requêtes modifiées sur l'apparition d'un événement, à la manière d'un petit programme ou script de commandes MMS téléchargé dans un serveur. L'intérêt d'un tel script par rapport à l'exécution séquentielle des mêmes services par un client est évidemment sa performance en terme de temps de réponse.

Certains auteurs affirment que l'ordre de traitement des EEs n'est pas du domaine de MMS mais de chaque implantation d'un serveur. Cependant, ceci nuit considérablement à la portabilité des applications MMS qui devrait pourtant être un des avantages cherchés dans l'adoption de la norme MMS (comme de toutes les normes en général). En effet, une application client qui compte sur un ordre d'exécution pour effectuer certaines actions dans un serveur donné ne peut plus faire cette supposition lors d'une utilisation avec un autre serveur. On peut toutefois trouver une justification à ce manque de précision en considérant le temps d'exécution des actions. La norme précise qu'en cas de présence d'actions pour un événement, le traitement des EEs liés à cet événement ne doit pas attendre la fin de l'exécution d'une action. En particulier si des EEs liés à des EAs coexistent avec des EEs sans EAs, il paraît normal de ne pas attendre la fin du traitement des actions pour envoyer les notifications des EEs sans EAs. Mais s'il était vraiment nécessaire de laisser aux concepteurs de serveurs la liberté de choisir l'ordre de traitement des EEs pour des raisons de performance, il aurait été souhaitable également d'introduire un mécanisme optionnel permettant aux clients de forcer cet ordre de traitement.

Notons au passage que l'impossibilité d'ordonner les actions est encore plus aggravée par le fait que l'exécution d'une action se déroule comme celle d'un service c'est-à-dire qu'elle implique la création et le traitement d'un objet transaction[4]. Et la norme MMS mentionne explicitement qu'il n'y a aucun ordre imposé pour l'exécution des objets transactions.

Dans [LSM90], Lumpp *et al.* illustrent l'intérêt de pouvoir redéfinir l'action associée à un événement lorsqu'un autre événement se produit. Dans MMS, le manque d'ordonnancement dans le traitement des EEs rend difficile la modification du lien existant entre un EC et un EA lors de l'occurrence d'un événement. En attachant quatre actions à un EC, ceci serait possible dans MMS si l'on pouvait garantir l'ordre d'exécution de ces actions. La séquence à exécuter serait alors :

```
Delete Event Enrollment
Delete Event Action
Define Event Action
Define Event Enrollment
```

MMS exige la destruction de l'EE liant un EC à un EA avant celle de l'EA. De même, un EA doit être créé avant la création de l'EE faisant le lien avec l'EC. S'il n'est pas possible de garantir cet ordre, on ne peut réaffecter une action événementielle sur l'occurrence d'un événement. Par ailleurs, il serait souhaitable que la séquence précédente s'exécute de façon atomique c'est-à-dire sans que d'autres requêtes de service MMS ne s'intercalent entre deux services de cette séquence. Mais là encore, cette garantie n'est pas donnée.

Notons quand même qu'il est parfaitement possible dans MMS de désactiver une action lors de l'occurrence d'un événement ou de produire un autre événement. Dans le premier cas il suffit d'avoir pour action le service `Delete Event Enrollment` seul; dans le second cas, `Trigger Event` ou `Write` selon qu'on ait un événement déclenché ou scruté.

Il appartient à chaque implantation serveur de décider de la politique d'ordonnancement des services, des actions et du traitement des EEs. Par exemple, certaines implantations peuvent favoriser

[4] Un objet transaction contient toute l'information nécessaire au traitement d'une requête de service confirmé ou d'une action événementielle. Nous traitons plus en détail des objets transactions à la section 4.2.1.

l'exécution des actions par rapport aux services car celles-ci sont associées à des événements (souvent des alarmes dont le temps de réponse peut être critique). Nous discutons plus en détail de ce problème dans la section 5.3.

3.11.3 Temps de validité d'une action

Un problème qui n'est pas du tout abordé dans MMS est le temps de validité d'une action. Dans la gestion d'événements du système NEM, l'action associée à un événement comporte un temps de validité [CPR86]. Celui-ci représente le temps T au bout duquel l'action n'a plus à être exécutée si l'événement est survenu et que pour une raison quelconque l'action n'a pas pu être prise en compte avant T. Par exemple si l'événement "`charge de litsun.epfl.ch < 0.2`" se produit à 12:00 alors démarrer l'action associée "`lancer une compilation Ada`" après 12:05 n'a plus d'intérêt. En effet, l'information attachée à l'événement risque alors de ne plus être valide.

Dans MMS, les actions n'ont pas de temps de validité qui leur sont associés. Une charge élevée du serveur MMS, le déclenchement rapproché de nombreux événements comportant des actions ou encore une action comportant un service bloquant sont pourtant des raisons suffisantes qui peuvent ralentir la prise en compte et l'exécution d'une action.

Nous proposons une solution simple mais qui a le désavantage d'être de niveau de modification 4 car elle implique une extension du protocole MMS. Il suffit de rajouter un attribut entier `Validity Interval` à l'objet EA (fig. 3.6). Cet attribut représente un temps en millisecondes. Il est associé à un temporisateur qui est démarré lors de l'apparition de l'événement. Quand le temporisateur atteint la valeur zéro, le temps de validité de l'action est expiré. L'utilisation de temporisateurs apparaît fréquemment dans MMS. Démarrer une temporisation associée à une action dont l'événement a été déclenché est relativement semblable à l'utilisation des temporisateurs tels qu'ils sont déjà définis et donc n'irait pas à l'encontre de l'esprit de MMS dans ce domaine. La seule difficulté tient au fait que le même événement pouvant apparaître plusieurs fois, il pourrait s'avérer nécessaire d'avoir plusieurs temporisateurs pour la même action.

Une telle extension de l'action événementielle permet également de guider les exécutions du traitement des événements et des actions dans le but de satisfaire des contraintes de temps au même titre que l'`Evaluation Interval` pour la détection des événements scrutés. En particulier, l'ordonnancement des actions mentionné dans la section précédente pourrait se faire en fonction des temps de validité. Cette seule remarque suffit à justifier l'intérêt que représente l'association d'une contrainte de temps à une action événementielle.

Nous analysons en détail cette proposition à la section 7.3.1.

3.11.4 Répartition

3.11.4.1 Exécution d'une action distante

Nous avons déjà mentionné que l'extension proposée dans [BBHM95] permet à un objet client de définir la méthode qui doit être appelée lors de l'apparition d'un événement sur un objet serveur. Cette méthode appartient au client et doit être appelée par le serveur quand l'événement se produit. Dans MMS, l'exécution d'une action événementielle se fait toujours de façon locale c'est-à-dire sur le serveur où s'est produit l'événement. Il n'est donc pas possible d'exécuter une action sur un autre serveur MMS en considérant le serveur où se produit un événement comme un client qui effectue une requête de service (l'action) vers un autre serveur.

Ceci ne serait pourtant pas incompatible avec MMS d'autant qu'il est justement prévu par la norme

qu'un serveur puisse effectuer lors de l'apparition d'un événement une requête de connexion `Initiate` s'il n'y a pas d'association entre le serveur et le client à notifier. Poursuivre en ce sens et effectuer une requête de service sur un autre serveur lors du traitement de l'événement survenu pourrait être une solution très intéressante.

Une telle facilité permettrait une meilleure gestion répartie des événements et réduirait le temps de réaction pour effectuer des actions sur des sites distants. Ceci se ferait toutefois au prix d'un temps de réponse accru de la notification car le serveur devrait attendre de recevoir la réponse de l'action avant d'envoyer cette notification.

Prenons l'exemple d'une VMD (notée VMD 1) modélisant un capteur de température pour un liquide dans une cuve et d'une autre VMD (notée VMD 2) modélisant un circuit de refroidissement de ce liquide. Un client contrôle le déroulement du processus de chauffage. Ces deux VMDs étant considérées comme distinctes, si on veut démarrer le refroidissement lors de l'événement "`Température > 500`" il faut d'abord que le client reçoive la notification correspondante depuis VMD 1 et ensuite qu'il effectue une requête de service MMS (par exemple `Start`) sur VMD 2 pour démarrer ce refroidissement.

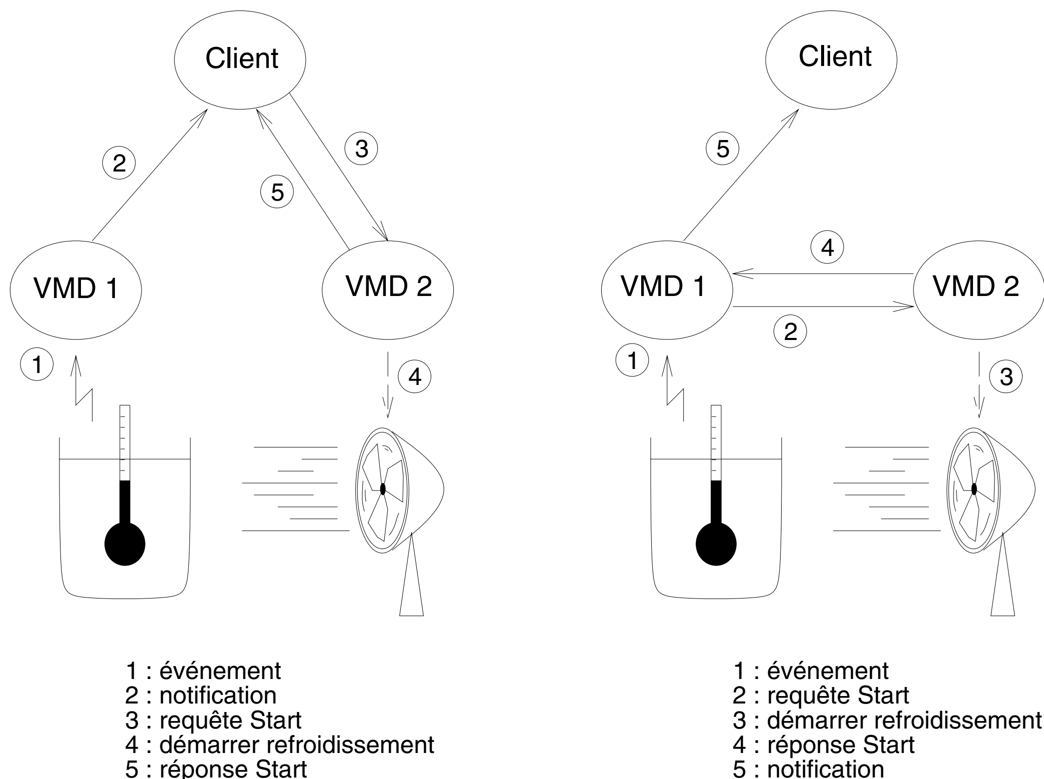


FIG. 3.11 - Séquencement des étapes lors d'une action locale et répartie

En utilisant une action à distance, VMD 1 effectuerait automatiquement le service `Start` sur VMD 2 puis enverrait la notification au client (fig. 3.11). D'après les résultats que nous avons obtenu dans [CV95] pour notre implantation d'un serveur MMS sur réseau MAP nous pouvons évaluer à environ 83 ms le temps nécessaire avant la prise en compte de l'action sur VMD 2 pour la première solution alors qu'il n'est que de 35 ms pour la seconde, soit 2.4 fois moins. Mais l'intérêt de l'une ou l'autre des solutions dépend aussi du caractère critique accordé à la notification par rapport à l'action. Dans le premier cas la notification prend 51 ms entre l'apparition de l'événement et sa réception par le client alors que dans le second nous l'avons estimée à 116 ms.

Il est relativement simple de conserver l'exécution d'une action à distance et recevoir une notification immédiate. Il suffit pour cela de créer un second EE lié à l'EC représentant l'événement. Le

client recevra alors deux notifications. La première est reçue en même temps que l'action répartie est effectuée, la seconde (la même que précédemment) est reçue après la réponse de l'action (fig. 3.12).

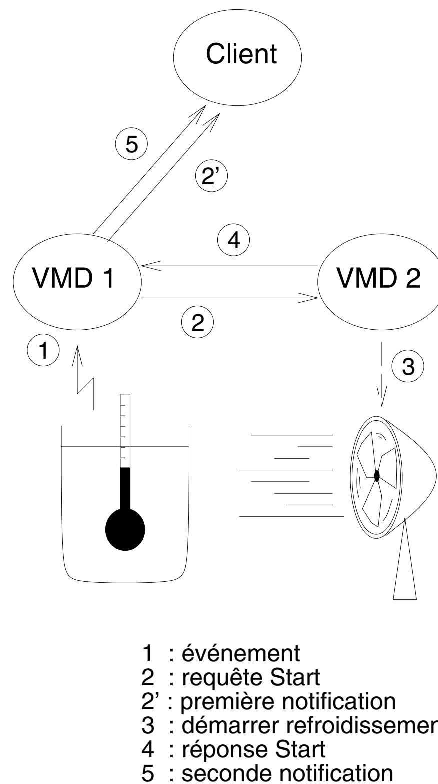


FIG. 3.12 - Séquencement des étapes lors d'une action à distance avec deux notifications

Nous décrivons à la section F.1 un autre exemple d'utilisation des événements MMS dans lequel la délocalisation de l'exécution de l'action événementielle serait souhaitable. Nous proposons donc d'ajouter à la gestion d'événements MMS la capacité d'effectuer des actions distantes. Nous étudierons en détail au chapitre 7 comment cette solution s'intègre à la norme MMS. D'ores-et-déjà il apparaît que cette solution est de niveau de modification 2 si les EAs sont pré-définis dans les serveurs[5]. Elle est de niveau de modification 4 sinon car il faut inclure dans le service `Define Event Action` un paramètre permettant l'identification de cette adresse et étendre l'objet EA avec un nouvel attribut.

De façon plus générale, les objets événementiels ne peuvent être répartis sur différents sites MMS avec par exemple l'EC sur le serveur où se produit l'événement et l'EA sur un autre serveur où devrait être exécutée l'action. Dans [Dak90], Dakroury a défini une méthode dite "multi-serveur" de répartition des objets MMS `Named Variable List` et `Program Invocation` sur différents serveurs. Il a été souligné que les objets événementiels pourraient également se prêter à cette répartition. Nous proposons au chapitre 7 une solution alternative.

3.11.4.2 Détection d'un événement défini de façon répartie

Nous étendons ici l'idée du filtrage des événements MMS potentiels. Nous avons montré qu'il n'est pas possible pour un utilisateur de définir une expression prédicat contrôlant le déclenchement d'un événement. Il lui est encore moins possible de définir un *événement réparti* c'est-à-dire un événement dont la réalisation dépend de l'état de variables situées sur des VMDs différentes [EC86].

[5] En fait si les adresses des sites où doivent être exécutées les actions distantes sont pré-définies.

Par exemple, dans [LSM90] le langage proposé permet le déverminage d'applications réparties. On peut donc définir des événements par des expressions du type "`(var == 1) on any node`". Celle-ci déclenche un événement lorsqu'une variable nommée `var` vaut 1 sur au moins un des sites du système considéré.

Une telle définition n'est pas possible dans MMS en l'état actuel de la norme. Elle nécessiterait la reconnaissance d'états globaux dans les systèmes répartis ce qui sort totalement du domaine de discours de MMS.

3.11.5 Diffusion des notifications

MMS est un protocole point-à-point. Ceci signifie en particulier qu'une association doit exister entre un serveur où se produit un événement et chaque client devant recevoir une notification de cet événement. Le traitement des transitions d'événements prévoit d'ailleurs l'ouverture d'une association entre le serveur et le client à notifier si celle-ci n'existait pas au préalable.

Le fait que MMS soit un protocole point-à-point annule la possibilité de diffusion des notifications d'événements. Si par exemple un EC de classe `NETWORK-TRIGGERED` est lié à n EEs alors le déclenchement d'un événement sur cet EC implique n traitements séquentiels des EEs et n envois successifs des notifications d'événements aux clients concernés.

Un traitement unique suivi d'une diffusion des notifications serait beaucoup plus efficace en terme de temps de traitement et de réception des notifications. Ceci a d'ailleurs été souligné par Rodd *et al.* dans [RZI90] qui proposent une modification de MMS pour satisfaire des contraintes temps-réel et mettent en évidence la nécessité de disposer de services de diffusion dans MMS. Nous avons montré dans [CV95] que pour notre implantation d'un serveur MMS, chaque client à notifier est pénalisé d'une durée à peu près proportionnelle à la position de l'EE correspondant dans la liste des EEs à traiter pour un même événement. Ainsi, pour le premier client, la notification parvient au bout de 69 ms après son déclenchement. Pour le second, elle parvient au bout de 87 ms. Pour le troisième au bout de 136 ms et pour le quatrième au bout de 158 ms. Il n'y a pourtant pas de critère qui permette de différencier un client par rapport à un autre.

La diffusion est une caractéristique importante des systèmes répartis mais qui n'est donc pas présente dans MMS en particulier pour les notifications d'événements. Mais on la retrouve dans le modèle objet de COSS où un objet peut même envoyer une notification à plusieurs autres objets sans connaître leur identité [Gro94]. Inversement, des objets peuvent recevoir des notifications sans connaître l'identité de l'émetteur. Ceci n'est évidemment pas possible dans MMS car une association doit toujours exister entre client et serveur pour envoyer une notification. Toutefois, nous avons déjà mentionné que dans le cas où la notification doit parvenir au même client qui a créé l'EE, le serveur n'utilise pas l'adresse du client mais uniquement un identificateur d'association local et propre à ce serveur.

Il faut cependant noter que le problème de la diffusion des notifications ne vient pas tant de MMS que du profil MAP initialement destiné à être le réseau pour MMS et qui est inadapté aux diffusions de messages. En effet, théoriquement MMS ne traite pas de la façon dont l'information est transmise et est indépendant de la technologie réseau sous-jacente [Gra92]. Toutefois, s'il est bien impossible de diffuser une notification en partie à cause du réseau sous-jacent à MMS, il n'est pas clair dans la norme [ISO90a] si le traitement des EEs lors d'une transition d'événement doit se faire séquentiellement pour chaque EE ou s'il est possible pour les EEs aux caractéristiques semblables de regrouper ces traitements en un seul. Ceci permettrait un gain de temps considérable. Ce problème est à rapprocher des exécutions d'actions événementielles. Si plusieurs EEs sont liés au même EA, faut-il exécuter l'action autant de fois qu'il y a d'EEs souscrivant à la transition ou une seule fois ? Encore une fois, si ce genre de décision est laissée libre à chaque concepteur de serveur MMS, il faudrait le

mentionner explicitement. Sans autre précision, deux implantations de serveur MMS risquent d'avoir des caractéristiques fort différentes en fonction de l'interprétation de la norme.

3.12 Pourquoi deux classes d'événements ?

Comme pour les sémaphores MMS, est-il possible de se restreindre à une seule classe d'événement ? Les deux classes d'événement introduites par la norme MMS peuvent paraître redondantes. En effet, un client MMS pourrait déclencher un événement scruté à distance au moyen d'un service `Write` de la même façon qu'il déclencherait un événement déclenché avec le service `Triggere Event`. Supposons qu'un EC de type scruté lié à une variable nommée v soit déclaré. L'écriture de la valeur VRAI (respectivement FAUX) dans v au moyen du service `Write` déclenche la transition `IDLE-TO-ACTIVE` (respectivement `ACTIVE-TO-IDLE`) si l'état de l'EC est `IDLE` (respectivement `ACTIVE`). C'est d'ailleurs de cette façon que se déclenchent les événements VAX/VMS. Le service `Set Event Flag` (`$SETEF`) et `Clear Event Flag` (`$CLREF`) mettent respectivement à 1 et 0 les bits représentant des événements [Dig82].

Pour déclencher un événement à distance, il semble donc possible de se passer du service `Trigger Event`. Mais un inconvénient de cette solution réside dans le fait que le client doit connaître la valeur booléenne de la variable scrutée de sorte qu'il puisse y écrire son inverse pour forcer la transition. Si le client est le seul à agir sur cette variable, il peut toujours connaître sa valeur ou l'obtenir au moyen du service `Read`. Par contre si cette variable peut être modifiée par plusieurs clients il est tout à fait possible que la valeur obtenue par le `Read` d'un client soit inversée par un autre client avant que le premier ait pu faire le `Write`. Dans ce cas, l'écriture dans cette variable ne déclenche aucune transition donc aucun événement.

Une solution un peu plus complexe doit alors être envisagée où les séquences `Read/Write` de clients différents sont toujours sérialisées et ne peuvent s'intercaler. L'utilisation d'un sémaphore banalisé MMS s à un seul jeton associé à la variable booléenne considérée v résout le problème. On aboutit alors à la séquence suivante que chaque client doit exécuter pour déclencher un événement scruté :

```
Take Control(s)
Read(v)
Write(not v)
Relinquish Control(s)
```

Cette séquence doit être exécutée deux fois pour déclencher deux événements alors que deux services `Trigger Event` suffisent et garantissent également ce double déclenchement. De plus, il faut noter que le service `Trigger Event` autorise la modification de la priorité de l'EC avant le déclenchement de l'événement. Dans la méthode précédente, ceci ne serait possible qu'au moyen du service supplémentaire `Alter Event Condition Monitoring`.

Dans un environnement distribué il n'est donc pas possible de garantir simplement le déclenchement à distance d'un événement MMS sans utiliser le service `Trigger Event`. L'utilisation du `Trigger Event` et la distinction faite entre les deux classes d'événement déclenché et scruté sont donc justifiées.

3.13 Conclusion sur les événements

Dans ce chapitre, nous avons décrit la gestion des événements MMS en effectuant un rapprochement avec d'autres systèmes comportant des événements. La table 3.3 résume ce qui a été dit précédemment et met en évidence les points communs entre les événements MMS et les événements tels qu'on les

trouve dans ces autres systèmes soient : les interruptions d'un microprocesseur, le déverminage de programmes informatiques, les systèmes d'exploitation, la gestion de réseaux et les systèmes de fenêtrage. MMS n'est pas directement comparable à un seul de ces systèmes de gestion d'événements mais on peut trouver dans tous des caractéristiques communes avec MMS et qui semblent avoir inspiré les auteurs de la norme [ISO90a].

Nous nous sommes également attachés à identifier quelques limitations des événements ainsi que quelques points obscurs de la norme MMS. Nous les résumons ici :

- Les conditions de déclenchement des événements scrutés sont très simples et ne permettent pas une personnalisation de la part des applications client en fournissant un prédicat lié à la réalisation de l'événement.
- De façon plus générale, les conditions de déclenchement d'un événement sont soit trop détaillées soit pas assez définies. Les concepteurs de la norme MMS semblent avoir eu du mal à distinguer avec précision ce qui devrait être dans le domaine de discours de MMS de ce qui ne devrait pas y être.
- La norme MMS n'offre de garantie ni sur l'ordre de traitement des EEs d'un même événement, ni sur l'ordre de traitement des actions événementielles et ni même sur celui des services MMS en général.
- Il n'est pas non plus prévu de moyen pour associer une contrainte de temps à l'exécution d'une action événementielle.
- Une action événementielle est locale au serveur où se produit l'événement et ne peut être exécutée sur un site distant ou dans une autre VMD.
- Les notifications MMS ne peuvent être diffusées à l'ensemble des clients devant recevoir la même notification. Les notifications doivent être envoyées une par une sur chaque association.

Certains de ces points seront repris et feront l'objet d'une étude plus approfondie dans les chapitres suivants de cette thèse de façon à proposer des solutions à certains problèmes ainsi que des extensions à la norme MMS pour d'autres problèmes.

MMS	Interruptions	Déverminage	Syst. d'exploitation	Ge
Client	Application utilisateur	Utilisateur	Processus	Applica
Serveur	Microprocesseur	Dévermineur	Noyau du système	Processu (ex: age
Équipement physique	Périphériques	Application déverminée	Architecture matérielle	Objets
Événement scruté	Interruption	Événement provoqué par le dévermineur (ex: point d'arrêt)	Signal ou événement originaire du noyau	N'impor provoqu (ex: cha
Événement déclenché	Trappe ou exception	Action utilisateur lors du déverminage (ex: CTRL-C)	Signal ou événement originaire d'un processus	Événem objet gé aussi un Rapport dans CM
Notification d'événement	Les résultats de l'exécution de la routine permettent de savoir qu'il y a eu interruption	L'utilisateur reçoit implicitement la notification en voyant comment se déroule le programme	Réception du signal et déblocage d'un processus	Notifica réseau (M-EVEN
Action événementielle	Exécution de la routine d'interruption	Action à exécuter sur un point d'arrêt (ex: affichage de variables, modification d'un registre, etc)	Actions exécutées par le noyau lors de l'occurrence de l'événement	Pas d'é Dans N comman (ex: dén
Filtrage par les EEs	Masquage des interruptions	Filtrage en appliquant un prédicat défini par l'utilisateur (ex: arrêt du programme quand "x=1")	Souvent limité à une combinaison logique d'événements simples	Filtrage appliqua
Définition d'un EC	Les interruptions et exceptions utilisables sont prédéfinies	Positionnement d'un point d'arrêt	Souvent prédéfinis. Sinon création par appel système (ex: $ASCEFC VAX/VMS)	Prédéfin
Définition d'un EA	Installation du vecteur d'interruption	Définition des actions à exécuter par le dévermineur sur un point d'arrêt	Généralement prédéfini	Dans N l'événen associée en une s l'utilisat
Définition d'un EE (fonction filtrage)	Positionnement des bits du registre d'état (ex: SR en 68000)	Définition du prédicat par l'utilisateur (requête à distance ou entrée au clavier)	Appel système permettant l'attente sur combinaison d'év. (ex: $WFLOR VAX/VMS)	Définitio CREATI CNMA)
Définition d'un EE (fonction adressage de la notification)	Implicite	Implicite	Implicite : attente de l'événement par appel système	Lors de discrimi
TriggerEvent	Instruction de trappe dans l'application utilisateur	Ex: modification au clavier de la valeur d'une variable pour induire un événement	Primitive d'envoi d'un signal (ex: kill UNIX)	Primitiv pour dé

Chapitre 4

Architecture d'un serveur MMS

4.1 Introduction

4.1.1 Exigences

A notre connaissance, il n'existe à ce jour aucun serveur MMS qui implante la totalité des services décrits dans la norme. Nous avons en particulier été les premiers à proposer une implantation quasi-complète et détaillée de la gestion d'événements MMS. Un certain nombre de sociétés proposent des serveurs MMS. Mais le nombre de services MMS supportés par ceux-ci est souvent restreint et les aspects les moins connus tels que les sémaphores et les événements sont rarement implantés [Ple94c]. Il semblerait qu'encore actuellement aucun serveur MMS commercial ne fournisse même un sous-ensemble minimal des services décrits dans la norme MMS en matière de gestion d'événements. Certes, il est encore rare qu'un serveur MMS doive fournir tous les services décrits dans la norme et en général seul un sous-ensemble s'avère nécessaire pour un produit donné, spécialisé sur un type d'application de productique particulière [VDC92]. Toutefois, il apparaît aussi que la complexité de la norme MMS, le manque de spécifications et une mauvaise vue d'ensemble des serveurs MMS sont autant de freins au développement de serveurs intégrant tous les services. Notre but est de proposer une architecture générale d'un tel serveur qui favorise une implantation facile et harmonieuse de tous les services et intègre tout particulièrement la gestion des événements, aspect de MMS encore sous-utilisé pour ne pas dire inconnu.

Nous commençons à la section 4.2 par une clarification des différentes étapes de l'exécution d'un service MMS confirmé en proposant un modèle pour leur exécution. Ce modèle doit permettre d'appréhender plus facilement et plus formellement le déroulement de ces exécutions et constitue le point de départ pour construire une architecture générique des serveurs MMS.

Dans ce chapitre, nous nous attachons donc à résoudre les problèmes généraux suivants :

- **Nouvelle classification** des services MMS;
- **Formalisation de l'exécution des requêtes** de service MMS confirmés;
- **Clarification du concept de serveur MMS** en terme de conception et d'implantation;
- **Définition d'une architecture simple et générique** des serveurs MMS. Nous entendons par *générique*, indépendante du langage de programmation, du système d'exploitation et surtout de l'application dédiée pour laquelle est normalement construit tout serveur MMS. L'architecture proposée doit donc s'adapter à tout scénario d'application serveur qui respecte la norme MMS;

- **Proposition d'une solution modulaire** autorisant l'ajout de nouveaux services et fonctionnalités à un serveur MMS;
- **Intégration de la gestion des événements MMS** au sein de cette architecture;
- **Identification de niveaux de priorités** dans l'exécution des différentes activités concurrentes d'un serveur.

4.1.2 Hypothèses

Pour poser les bases de la définition de l'architecture d'un serveur MMS, nous sommes amenés à faire les hypothèses suivantes :

1. L'application serveur MMS et la pile de communication sont deux processus différents. La MMPM et plus généralement le fournisseur de services MMS font partie du processus gérant la communication (fig. 4.1). L'application utilisateur est ainsi libérée des problèmes de communication et toute application MMS située sur le même site peut faire appel aux fonctions de communication offertes par le même fournisseur de services. Cette séparation permet également d'éviter de dupliquer la MMPM pour chaque applications MMS se trouvant sur un même site. Par ailleurs, les fournisseurs de services offrent souvent une interface normalisée (telle que MMSI). La séparation de l'application et de la MMPM accroît donc la portabilité des applications MMS.

2. L'architecture matérielle sur laquelle s'exécute l'application serveur est monoprocesseur et intégralement dédiée à cette application. En effet, bien souvent l'application MMS (serveur ou client) se trouve sur une carte séparée de celle de la pile de communication. Cette contrainte nous permet de ne pas considérer les problèmes de traitement dus aux protocoles de communication et de nous concentrer uniquement sur l'application serveur.

3. La pile de communication et l'application serveur MMS communiquent par deux files de messages (PDUs MMS), l'une pour les messages reçus par le serveur (`QueueReceptions`), l'autre pour les messages envoyés par le serveur (`QueueEnvois`). Ces files d'attente sont communes à toutes les associations existantes entre le serveur et les clients (fig. 4.1). Elles sont ordonnées par ordre d'arrivée (resp. d'envoi) des messages. Il est ainsi plus facile de conserver l'ordre d'arrivée des requêtes provenant de différents client. Par ailleurs, les requêtes n'ont pas de priorités et les associations non plus. Il n'est donc pas utile d'affecter une file d'attente par association. Enfin, le fait d'avoir une seule file d'attente au lieu d'une par association permet de traiter les requêtes de gestion d'association `Initiate` et `Conclude` de la même façon qu'une autre requête de service MMS.

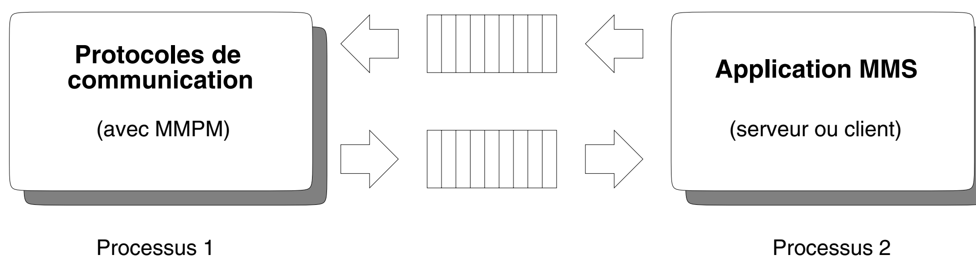


Fig. 4.1 - Représentation schématique de l'application MMS et de la pile de communication

4. `QueueReceptions` et `QueueEnvois` sont considérées de capacité infinie. En pratique la MMPM d'un client empêche les envois de requêtes si le nombre de réponses en attente devient égal au nombre de services en attente négociés durant l'ouverture de la connexion avec le serveur (en général, typiquement entre 1 et 10). Le nombre d'associations qu'un serveur peut accepter étant aussi généralement fixe, `QueueReceptions` ne peut jamais être saturée. Le caractère infini des files d'attente est surtout utilisé pour éviter qu'un serveur qui désire envoyer la réponse d'un service confirmé ou un service non confirmé ne soit bloqué parce que `QueueEnvois` est pleine.

5. Les messages en provenance de la pile de communication sont simplement déposés par ordre d'arrivée (FIFO) dans `QueueReceptions` sans aucune intervention du serveur. Celui-ci n'a qu'à consulter `QueueReceptions` pour savoir si des messages sont arrivés. La politique FIFO se justifie par le fait qu'il n'y a aucune raison de modifier l'ordre d'arrivée des messages à ce niveau. Par ailleurs, les messages dans `QueueReceptions` n'ont pas encore été consultés par le serveur. Si une autre politique d'ordonnancement devait être nécessaire, celle-ci interviendrait au niveau des autres files d'attentes du serveur (voir section 4.2).

6. Les messages envoyés par le serveur vers la pile de communication sont simplement déposés dans `QueueEnvois` sans autre action. Le processus gérant la pile de communication n'a qu'à consulter `QueueEnvois` pour savoir si des messages sont à envoyer.

7. Nous souhaitons pouvoir porter facilement notre serveur sur tout système. Notre définition de la généricité implique donc que l'on ne peut pas faire appel aux capacités de gestion de tâches d'un système d'exploitation ou d'un langage donné pour exécuter les entités d'un serveur susceptibles d'être parallélisées. Par exemple, on ne peut utiliser les "threads" DCE [Fou93], les processus UNIX [Bac86] ou les tâches Ada [Ins83]. Ceci implique que nous intégrions à notre architecture certaines fonctions de systèmes d'exploitation telles que l'ordonnancement des activités concurrentes. Le fait de gérer explicitement dans notre modèle de serveur les exécutions des activités concurrentes nous permet en outre un contrôle direct sur le déterminisme dans l'exécution de ces activités.

8. Bien qu'il soit possible suivant la norme MMS d'avoir plusieurs VMDs au sein d'un seul serveur, dans un but de simplification on peut identifier sans perte de généralité un serveur à une VMD. Les résultats exposés ici s'étendent facilement dans le cas d'un serveur avec plusieurs VMDs.

4.1.3 Notes sur les représentations "par objet"

4.1.3.1 Les objets MMS

MMS utilise le terme *d'objet* pour représenter l'information contenue dans une VMD. La représentation de ces objets se fait par une notation qui est propre à la norme MMS. Les objets sont identifiés de façon unique par un ou plusieurs attributs clés. Ces attributs clés sont représentés par les mots clés `Key Attribute`. Les attributs normaux sont représentés par le mot clé `Attribute`. Enfin, l'existence de certains attributs est conditionnée par la valeur d'autres attributs. Ceci est représenté par

les contraintes (mot clé `Constraint`). Une classe d'objet MMS est alors définie de la façon suivante :

```
Object: (name of class)

  Key Attribute: (name of attribute type (values))
  .
  .
  .
  Key Attribute: (name of attribute type (values))
  Attribute: (name of attribute type (values))
  .
  .
  .
  Attribute: (name of attribute type (values))
  Constraint: (constraint expression)
        Attribute: (name of attribute type (values))
        .
        .
        .
        Attribute: (name of attribute type (values))
```

Dans [Ple94a], Pleinevaux a étudié comment ce simple modèle objet dans l'environnement MMS intègre les notions connues des modèles orientés-objet classiques (adressage, encapsulation, héritage, etc). Notre but ici n'est que de ré-utiliser la notation objet MMS, dans un souci d'homogénéité, pour définir de nouveaux objets. Ces derniers représentent également une partie de l'information contenue dans les VMD. A la différence des objets MMS, ceux-ci ne sont évidemment pas visibles au travers des services MMS mais sont nécessaires au fonctionnement interne du serveur.

4.1.3.2 La méthode HOOD

La conception de l'architecture du serveur MMS s'est inspirée des principes de base de la méthode orientée-objet HOOD ("Hierarchical Object Oriented Design") [DHM93]. Seules les caractéristiques HOOD utilisées pour la conception de notre serveur MMS sont décrites ici et ce dans le but de fixer le cadre de notre exposé ainsi que de préciser la terminologie objet employée.

Dans [Ros92], Rosen a souligné qu'il fallait distinguer deux types de méthodes orientées-objet. Le type *orienté-objet par classification d'abord* est fondé sur une organisation des objets par classes. D'une classe peuvent être dérivées d'autres classes qui gardent le comportement de la première mais rajoutent également un comportement qui leur est propre. On aboutit ainsi à une organisation des objets par *héritage*. Ce type, largement influencé par les langages orientés-objet, est le plus répandu au point que pour beaucoup il n'y a pas de méthode orientée-objet sans héritage. Pourtant, un deuxième type d'organisation des objets tout aussi essentiel pour la conception de systèmes existe : le type *orienté-objet par composition d'abord*. Un objet est alors constitué de différentes parties et est donc construit au moyen d'objets plus élémentaires. On aboutit alors à une organisation des objets par *composition*. On retrouve cette composition dans la méthode Object-Oriented-Design (OOD) initialement définie par Booch [Boo86].

Les fondements essentiels de la méthode HOOD reposent sur deux types de hiérarchie entre objets :

- la hiérarchie "utilise" représente le fait qu'un objet nécessite les opérations exportées par d'autres objets et fait donc *appel* aux opérations de ces objets sous-jacents;

– la hiérarchie "inclusion" part de la constatation que les objets sont composés d'autres objets plus petits et permet donc la construction d'objets *parents* par briques plus élémentaires d'objets *fils*.

HOOD s'inspire donc du type orienté-objet par composition et ne fait que peu de cas des mécanismes d'héritage pour laisser la place essentielle à la composition. Nous n'utilisons donc pas l'héritage dans notre conception.

Dans HOOD un objet est représenté par une *interface* et une partie interne ("internals"). L'interface peut exporter des *opérations*, des *types* ou des *constantes*. La partie interne est l'implantation de ce que fournit l'interface. Bien sûr un objet n'est accessible que par son interface. La hiérarchie "utilise" est représentée par le fait qu'un objet peut faire appel à l'interface d'un autre objet. La partie interne d'un objet peut être constituée d'autres objets pour satisfaire la hiérarchie "inclusion". Un objet qui n'en contient pas d'autre est appelé objet *terminal*.

Les raisons essentielles qui ont conduit à l'utilisation de la méthode HOOD sont les suivantes :

– la méthode par composition est plus adaptée à l'architecture des serveurs MMS. Les objets MMS se prêtent mal aux mécanismes d'héritage et il semble naturel de voir un serveur MMS comme un assemblage d'objet-briques plus ou moins élémentaires;

– avantages connus et reconnus de la conception par objets : encapsulation, modularité, responsabilité propre à chaque objet, etc;

– représentation graphique claire et précise qui permet d'avoir immédiatement une vue d'ensemble d'un serveur MMS;

– modifications et extensions aisées du serveur;

– emploi d'une terminologie précise et codifiée;

– utilisation d'outils d'aide à la conception et autorisant dans une certaine mesure la vérification de la cohérence du système considéré.

4.2 Un modèle pour l'exécution des services MMS

Dans cette section nous ne traitons que des services confirmés. Les services non confirmés (en fait très peu nombreux dans MMS) ne sont pas considérés et le terme *service* désigne donc uniquement un service MMS confirmé.

4.2.1 Les objets Transaction

Pour pouvoir modéliser l'information nécessaire au traitement d'un service confirmé, MMS a introduit l'objet *transaction* ("Transaction Object" ou TO). Cet objet est initialisé lors de la réception d'une requête de service et existe jusqu'à ce que ce service soit traité, que celui-ci soit exécuté ou annulé. Dans un serveur, à une requête de service MMS correspond donc de façon unique un objet transaction. Cette unicité est réalisée grâce aux deux attributs clés `Application Association Identifier` (AAId) et `InvokeId` qui sont respectivement un identificateur de l'association sur laquelle a été faite la demande de service et un identificateur du service sur cette association. L'identificateur d'association est unique dans l'ensemble du serveur alors que l'identificateur de service `InvokeId` n'est unique que dans le contexte d'une association.

```
Object: Transaction

  Key Attribute: InvokeId
  Key Attribute: Application Association Identifier
  Attribute: List Of Pre-execution Modifiers
  Attribute: Current Modifier Reference
  Attribute: Confirmed Service Request
  Attribute: List Of Post-execution Modifiers
  Attribute: Cancelable (TRUE, FALSE)
```

Fig. 4.2 - L'objet Transaction défini par MMS

Au contraire des autres objets MMS, l'objet transaction n'est pas directement accessible par les services MMS. Par contre, celui-ci peut être détruit par suite d'un service `Abort` ou d'un service `Cancel`. Ceci revient à dire que le service en attente est annulé.

4.2.2 Services bloquants et services immédiats

Dans [Cas94a], nous avons défini une nouvelle classification des services MMS selon leur caractère *bloquant* ou *immédiat*. Un service est dit bloquant si les conditions pour son exécution ne sont pas réunies au moment où le service est pris en compte par le serveur. Ce qui caractérise le plus les services bloquants est que leur temps d'exécution ne peut pas être borné car il dépend de causes externes indépendantes du service et souvent de la VMD elle-même. Un service est dit immédiat quand le fil logique de son exécution ne rencontre jamais de blocage à cause de conditions externes non satisfaites. Ce type de service peut être exécuté complètement dès qu'il est reçu par le serveur et son temps d'exécution peut être borné (fig. 4.3).

La différenciation entre services bloquants et immédiats provient directement de la définition des services faite dans la norme MMS. Elle est importante car elle conditionne l'architecture des serveurs, la façon dont est traité un service bloquant étant différente de celle d'un service immédiat comme nous le verrons par la suite.

4.2.2.1 Services immédiats

Les services immédiats sont donc ceux pour lesquels on peut qualifier le temps d'exécution de borné si ce n'est de connu. En d'autres mots, une fois que l'objet transaction qui représente un service immédiat a été initialisé, le service peut s'exécuter de bout en bout sans être arrêté par des conditions externes telles que la non disponibilité de ressources. Les services `Identify`, `Get Variable Access Attributes` et `Report Event Condition Status` par exemple sont immédiats.

4.2.2.2 Services bloquants

Le service MMS `Take Control` est un exemple typique de service bloquant. En effet, ce service n'est pas exécuté jusqu'à son terme et sa réponse n'est pas renvoyée au client tant que le sémaphore sur lequel il s'applique n'est pas alloué à la SE correspondante. En conséquence, l'exécution de ce service dépend de conditions externes indépendantes du serveur, ici la libération d'un jeton du sémaphore par un client grâce au service `Relinquish Control`.

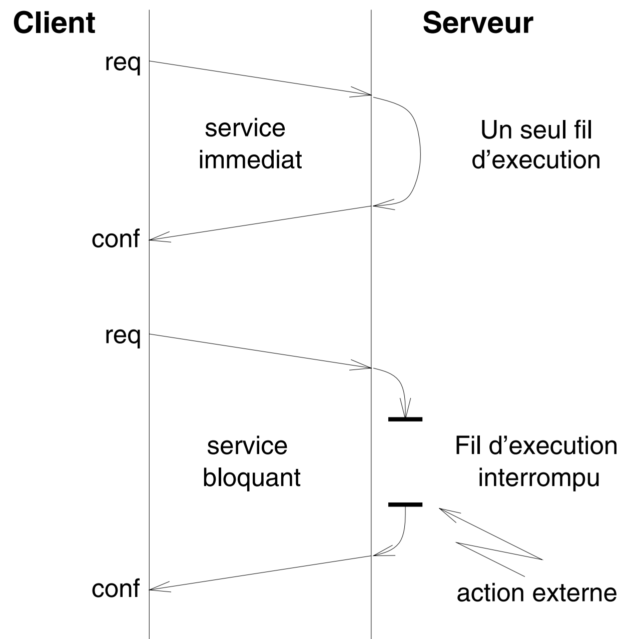


FIG. 4.3 - Service bloquant et service immédiat

Le service `Input` de gestion des stations opérateur est un autre service bloquant. Typiquement, `Input` attend qu'un opérateur humain entre une série de caractères sur un clavier. Ce service ne peut se terminer que lorsque l'opérateur a fini d'entrer sa chaîne de caractères (par exemple au moyen de la touche < `CR` >). Par conséquent, le temps d'exécution d'un service `Input` est inconnu par avance et peut même être "illimité" si aucun opérateur ne se trouve derrière la console. De plus, si l'on considère que claviers et écrans sont des ressources exclusives alors seul un service `Input` peut s'exécuter à un moment donné sur un objet station opérateur. Des services `Input` successifs doivent se mettre en attente du service `Input` courant. Ceci justifie doublement que le service `Input` soit bloquant.

Pour cette même raison, le service `Output` est bloquant lui aussi. En effet, si en soi `Output` peut toujours être exécuté (simple impression de caractères sur un écran par exemple), le fait qu'un `Input` soit en cours sur le même écran empêche l'exécution de l'`Output`. Parce que le service `Input` est bloquant, le service `Output` est "dégénéré" en service bloquant[1].

Donc nous dirons que si un service A doit attendre le terme de l'exécution d'un service B bloquant alors A est bloquant également. On peut donc définir deux niveaux de services bloquants : le premier pour lesquels l'exécution peut être arrêtée suite au déroulement normal du service et le second pour lesquels le service ne bloquerait pas s'il était exécuté seul mais qui peut bloquer suite à une exécution en cours d'un service de niveau 1.

De façon générale, tout service qui contient un modificateur est un service bloquant de niveau 1. Un modificateur qu'il soit de type `Attach To Semaphore` ou `Attach To Event Condition` bloque le service jusqu'à ce qu'une condition indépendante du serveur soit remplie : prise de contrôle d'un sémaphore ou occurrence d'un événement. Le temps d'exécution d'un service modifié est donc a priori inconnu et non borné.

On peut donc donner une classification des services MMS qui prend en compte le caractère bloquant ou immédiat. Celle-ci est résumée sur la figure 4.4.

---

[1] `Output` peut être implanté comme effectuant un simple dépot dans un tampon. Dans ce cas, `Output` n'est bloquant que quand le tampon est plein.

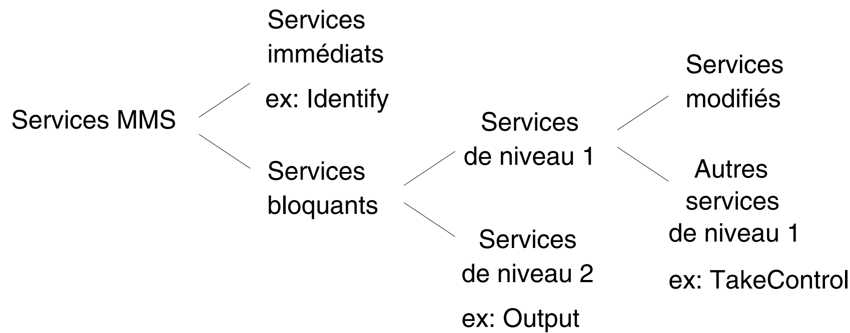


FIG. 4.4 - Classification des services MMS selon leur caractère bloquant

Dans certains cas, le fait qu'un service soit bloquant dépend du dispositif physique modélisé par la VMD. Bien que peu probable, il est tout à fait envisageable (c'est-à-dire non interdit par la norme) que le service `Read` doive attendre l'intervention d'un opérateur pour pouvoir s'exécuter, à la façon du service `Input`. Dans ce cas `Read` devient un service bloquant. Notre classification d'un service dans la catégorie bloquant ne peut prendre en compte de tels cas qui sont entièrement dépendants de chaque application serveur.

Il est important pour une application client de connaître le caractère bloquant ou non des services qu'elle demande à un serveur. Un client qui requiert un service immédiat peut s'attendre à recevoir une réponse dans les délais normaux (habituels) de transmission et d'exécution sur le serveur. Un client qui requiert un service bloquant ne peut pas présupposer du temps qui sera nécessaire pour recevoir la réponse. Ce client peut alors fonctionner en mode asynchrone et mettre à profit ce temps d'attente pour effectuer d'autres activités utiles.

4.2.3 Services suspensibles

Un service MMS est dit *suspensible* si son exécution est susceptible d'être retardée de façon significative sans pour autant que ce retard soit dû à des conditions externes non réalisées. **En ce sens, un service suspensible n'est pas (nécessairement) un service bloquant**. L'identification des services suspensibles peut se révéler utile pour implanter un serveur MMS dans la mesure où les temps d'attente introduits par ces services peuvent être exploités pour effectuer d'autres tâches en attente d'être servies. Par exemple, dans [VDC92], les auteurs ont mesuré pour le service MMS `Start` un temps de réponse de l'ordre de 84 ms alors que celui-ci est de l'ordre de 13 ms pour un service `Read`. Ceci s'explique par la nécessité de créer un nouveau processus pour le PI à démarrer. Le serveur attend passivement l'acquittement de création du processus de la part du système d'exploitation. Pendant ce temps au moins deux requêtes complètes de service `Read` auraient pu être effectuées. Mais le service `Start` n'est pas bloquant car même lorsque le serveur est en attente de la réponse du système d'exploitation, l'exécution du service se poursuit au niveau de l'appel système. Un service bloquant quant à lui risque toujours de voir son exécution complètement arrêtée.

La notion de service suspensible est fortement liée à chaque implantation particulière d'un serveur MMS et à la limite tout service MMS peut être considéré comme suspensible. Nous nous contenterons donc à donner une simple indication sur les services qui sont le plus susceptibles d'entrer dans cette catégorie. Notre classification d'un service dans la catégorie suspensible est donc guidée par la forte probabilité dans une implantation donnée d'avoir à effectuer pour ce service des actions traditionnellement qualifiées de lentes (lecture/écriture disque, impression sur un terminal, appel système, etc).

4.2.4 Etats d'un service MMS confirmé

Pour étudier de façon plus précise l'exécution des services MMS confirmés et dériver une architecture qui satisfasse ces exécutions, il est nécessaire de disposer d'un modèle un peu plus formel que la description textuelle faite par la norme [ISO90a]. Nous allons donc définir un modèle qui prenne en compte les caractères bloquant, immédiat et suspensible d'un service MMS. Nous ne faisons pas d'hypothèse quant à l'implantation pour rester aussi général que possible et que ce modèle s'applique à tous les serveurs MMS.

Les divers états possibles que peut prendre un service s'appliquent également à l'objet transaction représentant ce service [CP95b]. Durant son existence dans une VMD, un service confirmé MMS est donc dans l'un des états suivants :

- *REÇU*: dans cet état, le service n'existe pas en tant que tel c'est-à-dire que l'objet transaction correspondant n'a pas encore été créé mais la requête de service se trouve dans la file des requêtes en arrivée du serveur. La seule contrainte est que le serveur n'a pas encore commencé à traiter cette requête. Par conséquent, ce service n'existe pas pour la fonction exécutive de la VMD.
- *MODIFIÉ*: cet état rend compte des services qui se trouvent bloqués sur un modificateur non satisfait. Un service dans l'état MODIFIÉ n'a pas encore été exécuté, c'est-à-dire qu'il n'a jamais atteint l'état EN-COURS.
- *BLOQUÉ*: un service est bloqué quand son exécution est interrompue pour des raisons externes. Cet état est utilisé pour modéliser les services bloquants en dehors d'un blocage dû à un modificateur. Un service qui a atteint l'état BLOQUÉ a pu être dans l'état MODIFIÉ mais n'y retournera jamais.
- *PRÊT*: un service dans cet état est prêt à être exécuté dès que la fonction exécutive peut le servir. L'exécution d'un service s'entend au sens large, c'est-à-dire inclut celle des modificateurs associés. Un service qui est dans l'état PRÊT peut donc retourner dans l'état MODIFIÉ.
- *EN-COURS*: un service dans cet état est en cours d'exécution. Tous ses modificateurs ont été traités donc il ne retournera jamais dans l'état MODIFIÉ.
- *SUSPENDU*: cet état est introduit pour rendre compte des services suspensibles et faire la différence avec les services bloquants.
- *NON-EXISTANT*: dans cet état, un service n'existe plus c'est-à-dire que son traitement est terminé et l'objet transaction qui le représentait, s'il a jamais existé, a été détruit.

La machine d'états de la figure 4.5 décrit les transitions possibles entre les différents états d'un service MMS confirmé. Ces transitions sont étiquetées de la façon suivante :

1. Réception d'une requête de service MMS
2. Prise en compte de la requête de service par le serveur
3. Rupture de l'association (`Abort`)
4. Requête de service annulée (`Cancel`)
5. Un modificateur n'est pas satisfait
6. Un modificateur est satisfait (sémaphore libéré ou apparition d'un événement)
7. La requête de service est sélectionnée pour exécution
8. Une condition externe au serveur bloque l'exécution du service

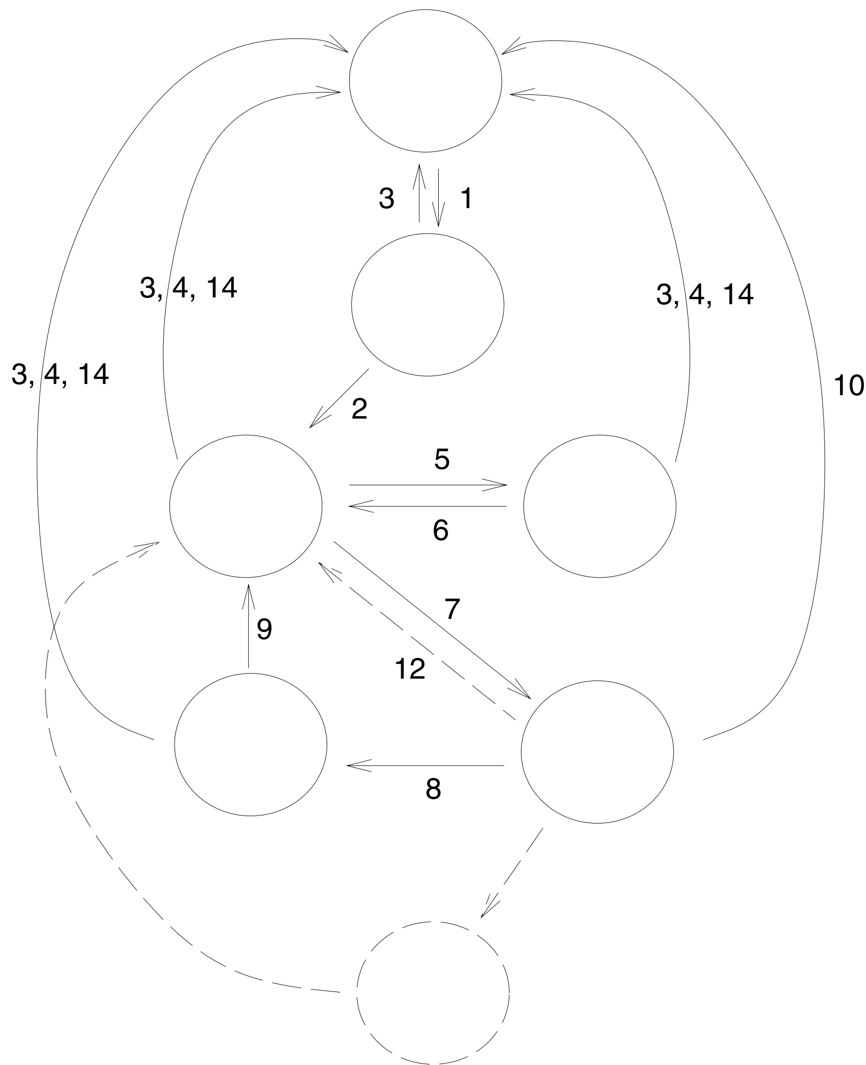


FIG. 4.5 - Machine d'états d'un service MMS confirmé

9. La condition de déblocage du service est satisfaite
10. L'exécution du service se termine normalement
11. L'exécution du service est suspendue
12. L'exécution du service est interrompue par le serveur au profit d'une requête de service plus prioritaire
13. L'exécution du service suspendu peut se poursuivre
14. Fin de temporisation

Contrairement aux autres états dont la justification découle du comportement des services décrits dans la norme MMS, il dépend d'une implantation donnée qu'un service se trouve dans l'état SUSPENDU. Cet état est donc introduit comme un guide d'implantation plutôt qu'un état à part entière du service. Il ne sera pas pris en compte dans le reste de cette étude pas plus que les transitions 11 et 13 le concernant. La transition 12 dépend de la politique d'ordonnancement des serveurs et donc aussi de l'implantation. Nous en discutons au chapitre 6.

Un service peut être bloquant parce qu'il contient des modificateurs. Nous faisons toutefois la distinction entre les états MODIFIÉ et BLOQUÉ car la façon de gérer un modificateur et un service bloquant diffère. La différence essentielle vient du fait qu'un service bloqué sur un modificateur n'a

pas encore commencé son exécution alors qu'un service qui se trouve dans l'état BLOQUÉ a été interrompu justement lors de son exécution.

Typiquement et dans la plupart des cas, un service MMS passe par les transitions 1, 2, 7, 10.

A chacune des transitions entre état correspond un ensemble d'actions à effectuer dans le serveur. Ces actions traduisent le comportement d'un serveur MMS tel qu'il est décrit dans [ISO90a]. Elles sont résumées sur la table 4.1.

Transitions d'état	**Actions concernées**
NON-EXISTANT → REÇU	Ne concerne pas le serveur
REÇU → NON-EXISTANT	Ne concerne pas le serveur
REÇU → PRÊT	Initialisation de l'objet transaction
PRÊT → MODIFIÉ	Initialisation EE traitant le modificateur ou initialisation SE bloqué sur un sémaphore
PRÊT → EN-COURS	Démarrage de l'exécution du service
PRÊT → NON-EXISTANT	Destruction de l'objet transaction Eventuellement envoi d'une erreur
EN-COURS → NON-EXISTANT	Destruction de l'objet transaction et envoi de la réponse ou d'une erreur
EN-COURS → BLOQUÉ	Sauvegarde de l'identification du service bloqué
EN-COURS → PRÊT	Préemption. Sauvegarde de l'identification du service
MODIFIÉ → NON-EXISTANT	Destruction objet transaction. Destruction EE ou SE Eventuellement envoi d'une erreur
MODIFIÉ → PRÊT	Destruction EE ou passage SE de `QUEUED` à `OWNER`
BLOQUÉ → PRÊT	Destruction info d'identification du service
BLOQUÉ → NON-EXISTANT	Destruction objet transaction et destruction info d'identification du service Eventuellement envoi d'une erreur

TAB. 4.1 - Principales actions exécutées lors des transitions

4.2.5 Aperçu général de l'architecture d'un serveur

Globalement, un serveur MMS se compose d'objets dont chacun est responsable du traitement d'un groupe particulier de services MMS et de la manipulation des classes d'objets MMS correspondantes [Ple92a], [RCP93], [CP93]. La responsabilité de la gestion d'un objet MMS appartient donc intégralement au gestionnaire concerné et à nul autre. L'accent est mis sur le découplage des diverses unités malgré les fortes relations de dépendance entre les objets MMS [Cas94a]. Cette décomposition est facile à appréhender dans la mesure où elle suit naturellement la classification des groupes de services de la norme et donne à chaque objet une responsabilité qui lui est propre. Parmi ces objets (notés *objets gestionnaires*), on identifie donc les gestionnaires de VMD, de domaines, de programmes, de variables, de sémaphores, de stations opérateur, d'événements, de journaux et de fichiers. Pour des raisons que nous expliquons plus loin, nous rajoutons également des gestionnaires d'associations et de temporisateurs. Cette décomposition favorise la modularité. Elle apporte ainsi une vision claire d'un serveur MMS et permet une maintenance plus facile.

Ces objets gestionnaires offrent une interface externe dont la plupart des opérations représentent des points d'accès à chaque service MMS. De plus, les relations de dépendance entre les objets MMS

impliquent une communication entre les gestionnaires. Un gestionnaire peut donc faire appel à un autre lorsque c'est nécessaire. Il existe donc des opérations dites internes qui ne sont pas liées à l'exécution d'un service particulier mais sont nécessaires au fonctionnement interne du serveur.

Les objets gestionnaires sont dirigés par le *Processeur de Transactions*. Le Processeur de Transactions peut lui-même être vu comme un gestionnaire d'objets transaction. Il reçoit les requêtes de services confirmés et renvoie les réponses. Il est responsable de l'identification du gestionnaire concerné par la requête de service à exécuter et appelle les opérations point d'accès aux services MMS qui correspondent aux services à exécuter. Le Processeur de Transaction règle donc les transitions d'état des services confirmés définies à la section 4.2.

A chaque état est associée une file d'attente des services dans cet état à un instant donné. La discipline de service de ces files d'attente est FIFO pour respecter l'ordre d'arrivée des requêtes MMS[2]. La file d'attente associée à l'état particulier REÇU est `QueueReceptions`. L'état NON-EXISTANT n'a pas de file d'attente associée. Le débordement des files d'attente associées aux états d'un service MMS est un problème qui ne se pose pas puisqu'à tout moment il ne peut y avoir plus d'objets transaction dans un serveur que la somme sur toutes les associations en cours du nombre maximum de services en attente négocié lors de l'établissement de chaque association. Cette négociation du nombre de requêtes en attente permet justement d'adapter les interactions entre des clients et un serveur MMS aux ressources disponibles dans le serveur (en l'occurrence la taille des files d'attente).

La figure 4.6 illustre l'architecture d'un serveur MMS.

4.3 Les activités concurrentes d'un serveur

4.3.1 Identification des activités concurrentes

Le fait que l'architecture ne repose pas sur un système d'exploitation particulier implique qu'il est nécessaire de gérer dans l'architecture même du serveur les exécutions parallèles (ou pseudo-parallèles) de tâches devant se dérouler en même temps (ou en apparence en même temps).

Pour pouvoir définir l'architecture d'un serveur, il est donc nécessaire d'identifier les différentes activités du serveur MMS qui sont susceptibles de s'exécuter de façon (pseudo)parallèle :

1. **La consultation des messages** dans `QueueReceptions` provenant de la pile de communication et l'initialisation des objets transaction qui représentent les services MMS transportés par ces messages. Cette activité n'est activée que s'il existe des messages dans `QueueReceptions`.

2. **L'exécution des services MMS** comprise au sens large c'est-à-dire avec le traitement des modificateurs. Cette activité n'est activée que s'il existe des objets transaction dans l'état PRÊT.

3. **La gestion des services bloquants** : un service bloqué est en attente de conditions externes au serveur (soit qu'elles soient liées au dispositif physique modélisé par la VMD, soit qu'elles soient dépendantes d'un client). Nous supposons que les actions permettant le déblocage d'un service se font de façon indépendante du processeur exécutant l'application serveur. Lorsque les conditions de déblocage sont remplies, une variable propre à chaque service bloqué est positionnée à une valeur indiquant que ce service peut se poursuivre. Il appartient à l'activité concurrente "gestion des services bloquants" décrite ici de consulter cette variable pour savoir quand un service peut être débloqué. Il existe donc un certain temps entre le moment où les conditions de déblocage

[2] Cette discipline est modifiée plus loin pour tenir compte de la priorité des requêtes.

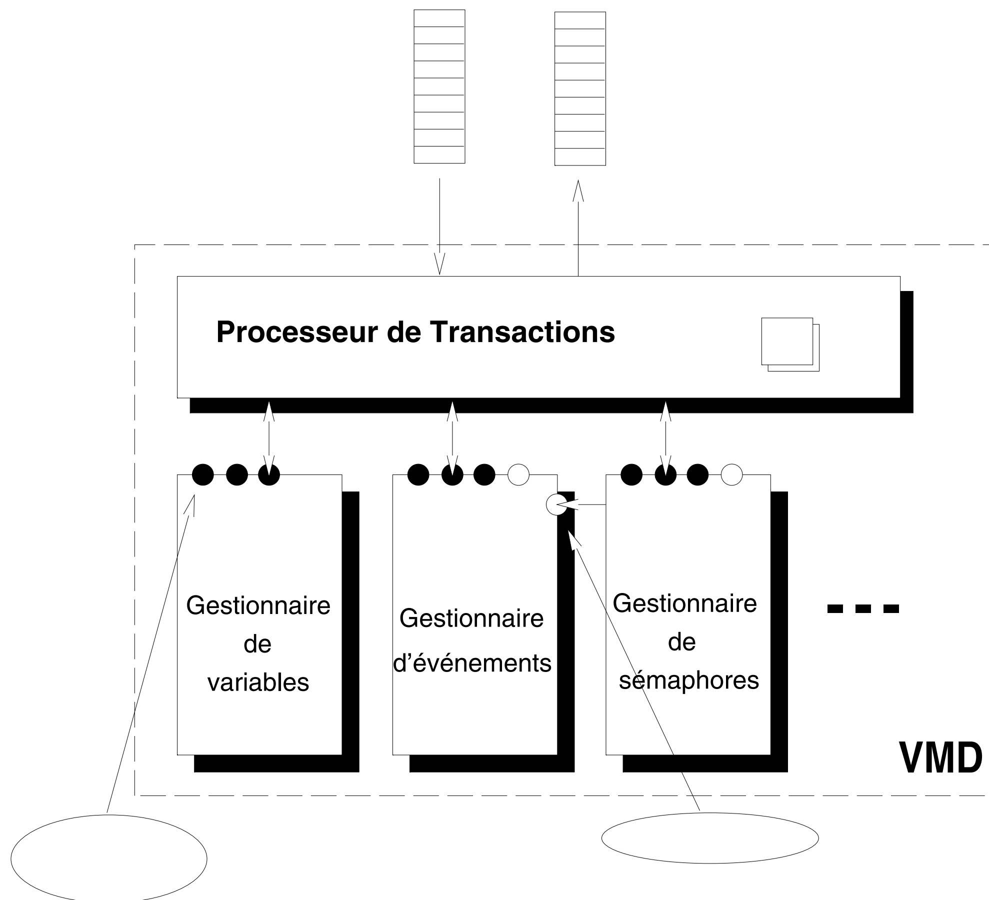


FIG. 4.6 - Architecture générale d'un serveur MMS

sont remplies et le redémarrage du service qui doit être effectué à ce moment. Cette activité n'est activée que s'il existe des objets transaction dans l'état BLOQUÉ.

4. **La gestion des temporisateurs** et des services qui les utilisent : nous supposons que le décompte des temporisateurs se fait de façon indépendante du processeur exécutant l'application serveur. L'expiration d'un temporisateur provoque le positionnement d'une variable à une valeur indiquant la fin de la temporisation. Il appartient à l'activité concurrente "gestion des temporisateurs" décrite ici de consulter cette variable pour savoir si le temps imparti est écoulé et dans l'affirmative d'effectuer l'action associée. Cette façon de faire implique qu'il existe un certain temps entre le moment où le temporisateur passe à zéro et le démarrage de l'action qui doit être effectuée à ce moment. Notons que la norme MMS ne requiert pas une granularité du temps plus faible que la milliseconde. Cette activité n'est activée que s'il existe des temporisateurs en cours de décompte.

5. **La détection des événements** par consultation périodique des ECs et des variables scrutées. Cette activité ne s'exécute que s'il existe des ECs dont la scrutation a été demandée.

6. **Le traitement des événements** lorsqu'ils ont été détectés ou déclenchés par un client. Cette activité n'est lancée que s'il existe des événements à traiter.

7. **La gestion des associations** : un serveur doit à tout moment pouvoir recevoir une nouvelle requête de connexion ou de rupture. En particulier, la terminaison brutale d'une association par

le service `Abort` entraîne une annulation de tous les services en cours sur cette association. Nous incluons dans cette activité le traitement du service `Cancel` car d'une part il s'applique à tous les services MMS confirmés et d'autre part il modifie effectivement l'état d'une association.

8. **La gestion et l'exécution des invocations de programmes MMS.** Nous supposons que l'exécution des invocations de programme MMS se fait sur une carte séparée ou sur le dispositif physique modélisé par la VMD et donc sur un processeur différent de celui du serveur. Nous considérons donc que l'application serveur n'est pas responsable de l'exécution des PIs et ne les incluons pas dans notre analyse. Seule une consultation périodique de l'état des programmes en cours doit être considérée (pour savoir si un programme s'est terminé par exemple). Ici, le serveur n'est responsable que de la gestion de l'objet PI qui est lui-même le reflet de l'état du programme correspondant.

9. **La gestion des services non confirmés**: ceci n'inclut que les services `Information Report` et `Unsolicited Status`. Le service `Event Notification` est pris en charge par le traitement des événements. L'utilisation des services non confirmés `Information Report` et `Unsolicited Status` implique une décision quant au moment de l'envoi d'un de ces services non confirmés à un client. Cette décision est propre à chaque implantation d'une application serveur et ne peut être prise en compte au niveau d'un serveur générique. Nous ne considérons donc pas cette activité concurrente.

4.3.2 L'ordonnancement des activités

Une exécution anarchique des différentes activités concurrentes d'un serveur donne lieu à des problèmes de protection des objets MMS contre des accès simultanés. Ces problèmes ont été étudiés dans [Cas94a] où la complexité des mécanismes de protection des objets MMS a été soulignée.

Pour éviter cette complexité, nous adoptons un ordonnancement simple non préemptif des activités concurrentes précédemment citées. A tout instant, une seule de ces activités est en cours. Celle-ci s'exécute toujours jusqu'à son terme sans être interrompue par une autre activité. Les accès aux objets MMS sont ainsi automatiquement sérialisés. Ceci signifie en particulier qu'une seule requête de service MMS peut être en cours d'exécution à un instant donné[3]. En d'autres termes, **la file d'attente associée à l'état EN-COURS contient au plus un élément** et ce uniquement pendant l'exécution d'un service c'est-à-dire pendant l'activité 2 “exécution des services”. A part la sérialisation des accès aux objets MMS, ceci garantit également l'exécution *atomique* de certains services (comme `Start`) tel que le préconise la norme MMS. Le mot “atomique” signifie ici qu'un service s'exécute complètement ou pas du tout, sans résultats intermédiaires observables et sans interactions avec un autre service MMS manipulant les mêmes données.

L'hypothèse que nous avons faite sur la gestion des temporisateurs peut impliquer une interruption de l'activité courante mais pour un temps très bref et en aucun cas n'introduit de problèmes d'accès conflictuels aux objets MMS. Au contraire, le fait d'attendre le tour de l'activité 4 “gestion des temporisateurs” pour effectuer l'action associée au passage à zéro d'un temporisateur permet d'éviter ces conflits puisque ce n'est qu'à ce moment là que d'éventuelles données communes avec l'activité en cours seront manipulées. L'ordonnanceur règle le passage d'une activité concurrente à une autre en fonction de critères qui peuvent être propres à chaque application serveur. La figure 4.7 illustre l'architecture d'un serveur en tenant compte des activités concurrentes. Les flèches en trait pointillés montrent quelques uns des accès directs d'une activité concurrente vers un gestionnaire de service.

[3] Par définition, le traitement des services bloquant nécessite l'interruption du service. Cette interruption ne se produit jamais au milieu d'une section critique ou lors d'un accès à un objet partagé et laisse donc toujours les données dans un état cohérent.

Par exemple l'activité 5 "détection des événements" utilise les services d'un moniteur d'événements qui se trouve dans le gestionnaire d'événements.

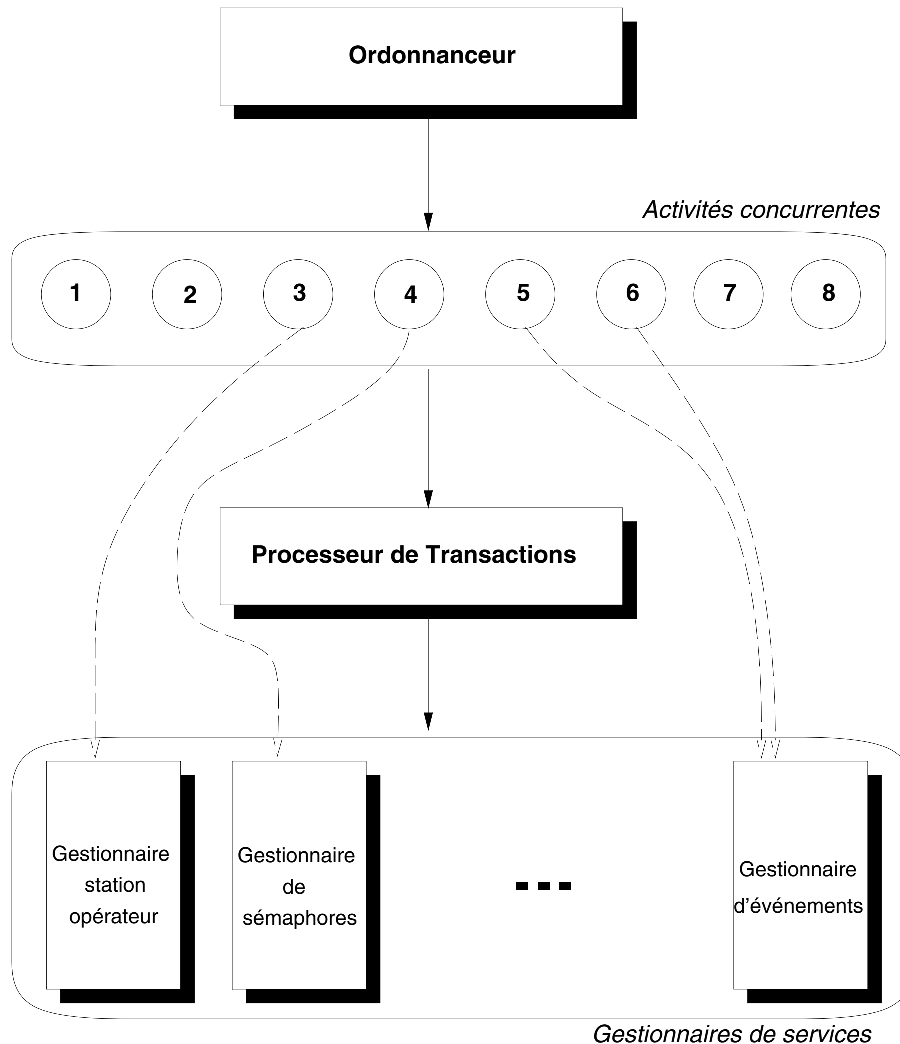


FIG. 4.7 - Activités concurrentes et architecture d'un serveur MMS

Comme nous l'avons vu, un client MMS peut fournir à un serveur des valeurs de durées au bout desquelles ou avant la fin desquelles une certaine action doit être exécutée. Ceci concerne les temporisateurs et la scrutation des événements. Il appartient à l'ordonnanceur de sélectionner les activités concurrentes de telles façon que ces temps soient respectés. Toutefois, le but de ce chapitre n'est pas de proposer une architecture satisfaisant des contraintes temps-réel associées aux différentes activités ci-dessus. Notre ordonnanceur reste général et simple : chaque activité concurrente est exécutée à tour de rôle et de façon itérative sans tenir compte de contraintes de temps associées, à la manière de *l'ordonnancement cyclique* [Coo91].

Pour éviter la complexité due à la gestion d'accès conflictuels aux objets, nous avons volontairement interdit la préemption des activités concurrentes et stipulé que chaque activité concurrente commencée se poursuit jusqu'à son terme. Il convient de préciser ce qu'est le terme d'une activité concurrente dans la mesure où celles-ci paraissent s'exécuter de façon continue. En fait, chacune de ces activités peut être vue indépendamment des autres comme un processus sans fin qui boucle sur lui-même et à chaque passe consulte une liste, retire un élément de cette liste et effectue un traitement sur cet élément. Par exemple, l'activité 2 "exécution des service", prend pour liste celle associée à l'état PRÊT et pour

éléments de cette liste les objets transactions. L'activité 4 "gestion des temporisateurs" travaille sur la liste des temporisateurs, etc. Si l'on fixe le nombre d'éléments sur lequel une activité peut travailler, elle arrive à son terme (dans une itération) lorsqu'elle a traité le quota d'éléments fixé même s'il reste d'autres éléments non traités dans la liste. Lorsqu'une activité arrive à son terme, on passe à la suivante.

Cet ordonnancement simpliste des activités concurrentes est réétudié au chapitre 6 pour satisfaire des contraintes temps-réel dans l'exécution des requêtes de service et de la détection d'événements. Mais on peut d'ores-et-déjà l'améliorer à peu de frais en identifiant des niveaux de priorité dans l'exécution des activités concurrentes. Par exemple, la détection des événements peut être considérée comme l'activité la plus prioritaire dans la mesure où la norme MMS fait explicitement mention de contraintes de temps qui peuvent être proposées par un client pour la scrutation des ECs. Le traitement des événements vient juste après, même si aucune contrainte de temps n'est imposée par la norme. Il paraît en effet peu raisonnable de chercher à garantir la détection d'un événement pour ensuite retarder l'envoi des notifications correspondantes d'autant que l'apparition d'un événement (alarme) peut correspondre à une situation critique qui demande une attention immédiate. A priorité égale, nous mettons ensuite la consultation des messages, l'exécution des services et la gestion des associations. Puis vient la gestion des services bloquants, des PIs, des services non-confirmés et des temporisateurs. Toutefois, on conviendra que même si cette classification par priorité paraît raisonnable, il serait souhaitable de laisser au concepteur d'un serveur MMS la liberté de choisir quelle activité concurrente est pour lui prioritaire compte tenu de l'application dédiée ou du dispositif physique qui est modélisé par la VMD.

On peut contrôler simplement la durée d'exécution de l'activité concurrente en cours en faisant varier le nombre d'éléments d'information ou d'objets sur lesquels celle-ci peut travailler. Les activités de priorité haute sont ainsi autorisées à travailler sur un nombre d'éléments important alors que celles de priorités plus basses travaillent sur un nombre plus réduit. Cette méthode revient à donner un certain poids à chacune des activités concurrentes. Ce poids peut être fixé à l'avance ou peut être utilisé de façon plus dynamique c'est-à-dire re-calculé à chaque itération. On aboutit alors à ordonnancer les activités concurrentes comme indiqué à la figure 4.8.

La fonction `ré-évaluer` re-calcule le poids de chaque activité lors de chaque itération. Cette façon de faire se retrouve dans le système d'exploitation UNIX où les priorités des processus sont revues régulièrement en fonction du temps CPU alloué à chaque processus [Bac86]. A partir de là, toutes les stratégies d'évaluation du quota d'éléments à traiter lors de chaque activité sont possibles. On doit évidemment tenir compte des priorités des activités concurrentes mais on peut aussi prendre en considération la charge du serveur par exemple. Celle-ci peut se mesurer par la taille des files d'attente associées à chaque état d'un service confirmé et particulièrement celle des états REÇU et PRÊT.

Il est même envisageable de représenter les priorités par des variables MMS modifiables par les clients au moyen du service MMS `Write`.

4.3.3 La communication entre activités

Il existe des contraintes de précédence entre activités se traduisant par la communication des résultats d'une activité vers une autre. Les activités concurrentes ne communiquent pas directement entre elles. Un élément d'information traité par une certaine activité mais qui demande un traitement ultérieur de la part d'une autre activité est mis dans une file d'attente. Le plus souvent cet élément d'information est un objet transaction et la file d'attente est une des files associées aux états d'un service MMS confirmé. Par exemple, un service débloqué par l'activité 3 "gestion des services bloquants" passe dans la file d'attente de l'état PRÊT pour traitement ultérieur par l'activité 2 "exécution des services".

```
déclarations
   combien : tableau[1 .. nb_activités] d'entiers
boucle
   ré-évaluer(combien)
   si queue(REÇU) ≠ ∅ alors
      CONSULTER_MESSAGES(combien[1])
   fin si
   si file(PRÊT) ≠ ∅ alors
      EXECUTER_SERVICE(combien[2])
   fin si
   si file(BLOQUÉ) ≠ ∅ alors
      GESTION_SERVICES_BLOQUANTS(combien[3])
   fin si
   si file(BLOQUÉ) ≠ ∅ alors
      GESTION_TEMPORISATEURS(combien[4])
   fin si
   si file(EC à scruter) ≠ ∅ alors
      SCRUTER_EC(combien[5])
   fin si
   si file(événements à traiter) ≠ ∅ alors
      TRAITER_EC(combien[6])
   fin si
   si file(REÇU) contient services d'associations ≠ ∅ alors
      GESTION_ASSOCIATIONS(combien[7])
   fin si
   si existent PI terminés ≠ ∅ alors
      GESTION_PI(combien[8])
   fin si
fin boucle
```

Fig. 4.8 - Ordonnancement des activités concurrentes

D'autres éléments d'information ne sont pas des objets transaction. Par exemple, la détection d'un événement par l'activité 5 "détection des événements" implique son traitement par l'activité 6 "traitement des événements". Là aussi les informations représentant l'événement à traiter sont mises dans une file d'attente mais n'ont rien à voir avec un objet transaction. Cette manipulation des événements est expliquée en détail dans la section 4.5.6.

4.4 Le Processeur de Transactions

Le Processeur de Transactions forme le coeur du serveur MMS car c'est lui qui récupère les requêtes de service MMS venant de `QueueReceptions`, appelle les gestionnaires en fonction des services demandés et renvoie les réponses une fois que les services ont été traités. Il unifie le traitement des services MMS confirmés pour ne confiner dans chaque gestionnaire de services que la partie de l'exécution qui est vraiment dépendante de la fonction MMS qui doit être effectuée.

La définition du Processeur de Transactions permet de traiter *tous* les services confirmés MMS

de façon identique et homogène. Le Processeur de Transactions manipule les files d'attente associées aux différents états d'un service confirmé et règle les transitions entre ces états. Il est activé par l'ordonnanceur suivant l'activité concurrente sélectionnée. Le tableau 4.2 montre les transitions d'état des services MMS confirmés qui peuvent se produire lors de la sélection d'une activité concurrente donnée.

Activité concurrente	Transitions concernées	Méthode associée
1 - Consultation des messages	REÇU → PRÊT	CreateTransactionObject
2 - Exécution des services	PRÊT → EN-COURS PRÊT → MODIFIÉ EN-COURS → NON-EXISTANT EN-COURS → BLOQUÉ * MODIFIÉ → PRÊT * BLOQUÉ → PRÊT	ProcessTransaction ProcessTransaction ProcessTransaction ProcessTransaction ProcessModifiedTransaction ProcessBlockedTransaction
3 - Gestion services bloquants	BLOQUÉ → PRÊT * BLOQUÉ → PRÊT * MODIFIÉ → PRÊT	ProcessBlockedTransaction ProcessBlockedTransaction ProcessModifiedTransaction
4 - Temporisateurs	MODIFIÉ → NON-EXISTANT BLOQUÉ → NON-EXISTANT * MODIFIÉ → PRÊT * BLOQUÉ → PRÊT	DeleteTransactionObject DeleteTransactionObject ProcessModifiedTransaction ProcessBlockedTransaction
5 - Détection des événements	Aucune	-
6 - Traitement des événements	Aucune * MODIFIÉ → PRÊT * NON-EXISTANT → REÇU	- ProcessModifiedTransaction -
7 - Exécution des PIs	Aucune	-
8 - Gestion des associations	BLOQUÉ → NON-EXISTANT PRÊT → NON-EXISTANT MODIFIÉ → NON-EXISTANT * BLOQUÉ → PRÊT * MODIFIÉ → PRÊT	DeleteTransactionObject DeleteTransactionObject DeleteTransactionObject ProcessBlockedTransaction ProcessModifiedTransaction
9 - Gestion services non confirmés	Aucune	-

TAB. 4.2 - Transitions d'état effectuées lors des activités concurrentes

L'objet Processeur de Transactions exporte donc des opérations qui sont appelées lors des activités concurrentes qui provoquent des transitions d'état. Ces opérations peuvent également être appelées par le Processeur de Transaction lui-même. Le nom de l'opération utilisée lors d'une activité concurrente donnée et pour une transition d'état donnée apparaît dans la table 4.2. Les transitions commençant par "*" représentent les transitions *induites*. Les transitions induites se produisent pour un autre objet transaction que celui sur lequel travaille l'activité concurrente (dit objet transaction principal). Mais ces transitions sont le résultat direct du traitement de l'objet transaction principal. Il peut s'agir par exemple du déblocage de services modifiés ou bloqués lors de la libération d'un sémaphore.

Chaque opération prend un ou plusieurs états de départ et un ou plusieurs états d'arrivée. Lors de l'appel d'une de ces opérations, le Processeur de Transactions sélectionne un objet transaction dans la file d'un des états de départ. Il effectue ensuite le passage de l'objet transaction vers la file d'attente d'un état d'arrivée et exécute les actions qui sont associées à cette transition.

Nous faisons en sorte que la sélection d'un objet transaction dans la file associée à l'état PRÊT

dépende de l'ordre d'arrivée du service au niveau du serveur (c'est-à-dire de l'ordre de la file de l'état REÇU). L'ordre d'exécution des services MMS n'est pas imposé par la norme. Toutefois, si l'on souhaite respecter au mieux l'ordre d'arrivée des services on maintient une structure FIFO pour la file d'attente associée à l'état PRÊT. Celle de l'état REÇU est automatiquement ordonnée par l'hypothèse 1 de la section 4.1.2. Quand un objet transaction passe dans l'état PRÊT, il est inséré dans la file par ordre d'arrivée. Le serveur maintient une variable compteur qui ordonne tous les services et est incrémenté à chaque service reçu. Ceci permet de ne pas perdre l'ordre de réception du service lorsque celui-ci retourne à l'état PRÊT alors qu'il provient des états BLOQUÉ ou MODIFIÉ. Tout objet transaction dans l'état PRÊT peut être traité sans condition. Lorsque l'opération associée à l'exécution des services MMS est activée, le Processeur de Transactions prend l'objet transaction situé en tête de liste dans la file d'attente de l'état PRÊT et démarre son traitement. De même, l'opération associée à la consultation des messages arrivés sélectionne le message en tête de file de l'état REÇU.

Par contre la structure des files d'attente des états MODIFIÉ et BLOQUÉ n'a que peu d'importance puisqu'on ne peut prévoir lequel des services dans ces états sera le prochain à être exécuté. En fait seuls les services modifiés sur un même événement ou bloqués en attente des mêmes conditions de libération doivent être ordonnés par ordre d'arrivée. Ceci permet alors lors du déblocage de ces services de favoriser l'exécution des premiers arrivés.

Les opérations qui ont pour état de départ MODIFIÉ ou BLOQUÉ reçoivent en paramètre l'identité de l'objet transaction à traiter (paramètres `InvokeId` et `AAId`). La sélection d'un objet transaction dans une file d'attente se fait alors au moyen de ces paramètres. Quant à l'état EN-COURS, nous avons déjà vu que sa file d'attente contient au plus un élément.

4.4.1 Opérations exportées

4.4.1.1 CreateTransactionObject

Appel :

- lorsque le Processeur de Transactions doit consulter les messages reçus dans `Queue Receptions`;
- lorsque le Processeur de Transactions doit traiter une action événementielle.

Etat de départ : REÇU

Etat d'arrivée : PRÊT

Description : crée et initialise l'objet transaction correspondant au message en tête de liste de l'état REÇU. Celui-ci est inséré à la fin de la file de l'état PRÊT. S'il s'agit d'une action événementielle celle-ci peut être insérée en tête de liste pour favoriser son traitement. Ce choix dépend des priorités accordées aux actions événementielles par rapport aux services MMS.

4.4.1.2 ProcessTransaction

Appel : lorsqu'il y a des services ou des actions événementielles prêts à être exécutés.

Etat de départ : PRÊT

Etat d'arrivée : NON-EXISTANT, BLOQUÉ ou MODIFIÉ

Description : sélectionne l'objet transaction en tête de file. Traite tous les modificateurs pouvant être satisfaits. Passe dans l'état MODIFIÉ dès qu'un modificateur ne peut être satisfait. Sinon,

identifie le service et appelle le gestionnaire correspondant. Le service passe donc dans l'état EN-COURS. S'il est exécuté jusqu'à son terme sa réponse est envoyée – les résultats sont retournés au gestionnaire d'événements s'il s'agit d'une action événementielle. Si l'objet transaction contenait des modificateurs de type `Attach To Semaphore`, les sémaphores correspondants sont libérés. L'opération `Delete Transaction Object` est ensuite automatiquement appelée. Si par contre l'exécution est bloquée, l'objet transaction est inséré dans la file de l'état BLOQUÉ.

Remarque : l'exécution du service `Relinquish Control` ou la libération des sémaphores pris par les modificateurs peuvent entraîner le déblocage de requêtes `Take Control` ou de modificateurs `Attach To Semaphore` en attente. Les opérations `Process Blocked Transaction` ou `Process Modified Transaction` respectivement sont alors automatiquement appelées pour faire passer les objets transaction correspondant dans l'état PRÊT. Ceci correspond aux transitions induites de la table 4.2.

4.4.1.3 ProcessBlockedTransaction

Appel : lorsqu'il y a des services ou des actions événementielles bloqués qui peuvent être exécutés.

Etat de départ : BLOQUÉ

Etat d'arrivée : PRÊT

Description : retire de la file de l'état BLOQUÉ l'objet transaction identifié par les paramètres d'entrée `InvokeId` et `AAId`. Le service passe donc dans l'état PRÊT, son exécution se terminera quand son tour viendra (opération `Process Transaction`).

4.4.1.4 ProcessModifiedTransaction

Appel : lorsque le modificateur qui bloque un objet transaction est satisfait.

Etat de départ : MODIFIÉ

Etat d'arrivée : PRÊT

Description : L'objet transaction identifié par les paramètres d'entrée `InvokeId` et `AAId` est inséré dans la file de l'état PRÊT. Son exécution se terminera quand son tour viendra (opération `Process Transaction`).

4.4.1.5 DeleteTransactionObject

Appel : dans tous les cas où un objet transaction doit être détruit : terminaison normale du service, fin de temporisation, rupture de l'association, annulation par `Cancel`.

Etat de départ : PRÊT, MODIFIÉ ou BLOQUÉ

Etat d'arrivée : NON-EXISTANT

Description : détruit l'objet transaction identifié par les paramètres d'entrée `InvokeId` et `AAId` de l'opération après l'avoir retiré de la file d'attente correspondant à son état. Les fins de temporisation ainsi que l'annulation du service par `Cancel` entraînent l'envoi d'une erreur au client concerné. Cette opération peut aussi détruire tous les objets transaction appartenant à une association spécifique.

Remarque : la terminaison d'un service bloqué ou la libération des sémaphores pris par les modificateurs peuvent entraîner le déblocage d'autres service bloqués, de requêtes `Take Control` ou de modificateurs `Attach To Semaphore` en attente. Les opérations `Process Blocked Transaction` ou `Process Modified Transaction` respectivement sont alors automatiquement appelées pour faire passer les objets transaction correspondant dans l'état PRÊT. Ceci correspond aux transitions induites de la table 4.2.

4.4.2 Composition

Le Processeur de Transactions est composé de quatre sous objets :

- l'objet classe `TransactionObjectClass` permet de manipuler les objets MMS transaction. Il renferme les attributs décrits par la norme pour l'objet transaction et exporte des opérations autorisant la lecture de ces attributs et leur mise-à-jour pour ceux qui sont modifiables.
- l'objet `TransactionObjectList` chargé de maintenir les listes de tous les objets transaction existant. Il fournit les opérations classiques de manipulation de listes.
- l'objet `ServiceDispatcher` est en fait l'interface vers les gestionnaires de services MMS. Il exporte des opérations qui représentent des portes d'accès vers ces gestionnaires de service. Leur but principal est l'identification du service a exécuté et l'appel du gestionnaire concerné.
- l'objet `TransactionObjectManager` effectue le traitement effectif des objets MMS en utilisant les trois objets précédemment cités. Les opérations exportées sont exactement celles du Processeur de Transaction.

Cette décomposition donne à chaque objet une responsabilité qui lui est propre. En particulier, l'ajout d'un nouveau gestionnaire de services MMS ou sa modification éventuelle n'entraînent pas de changements dans la logique même du Processeur de Transactions. Seule la partie interne de l'objet `ServiceDispatcher` est concernée par ces changements. L'interface elle reste inchangée. La figure 4.9 illustre sommairement cette décomposition.

4.5 Les gestionnaires de services MMS

4.5.1 Structure générale

Chaque objet gestionnaire est responsable du traitement d'un groupe donné de services MMS et manipule donc les objets MMS correspondant à ces services et uniquement ceux-ci. Les objets gestionnaires ont une structure comparable à celle du Processeur de Transactions. Chaque objet gestionnaire est composé au moins des sous-objets suivants :

- un *objet manipulateur* chargé de la manipulation de la classe d'objets MMS dont il a la responsabilité. Cet objet est générique en ce sens que chacune de ces instanciations représente un objet MMS de même classe mais distinct. Les attributs qu'il contient représentent les attributs définis dans la norme pour les objets MMS et éventuellement d'autres attributs nécessaires au fonctionnement interne du serveur. Chaque attribut peut être lu au moyen d'une opération qui lui est propre. A chaque attribut susceptible d'être modifié lors de la vie de l'objet correspond également une opération individuelle autorisant cette modification. Les opérations de création (d'instanciation) de l'objet et de destruction font aussi partie des opérations exportées. Cet objet de structure fort simple est toujours terminal.

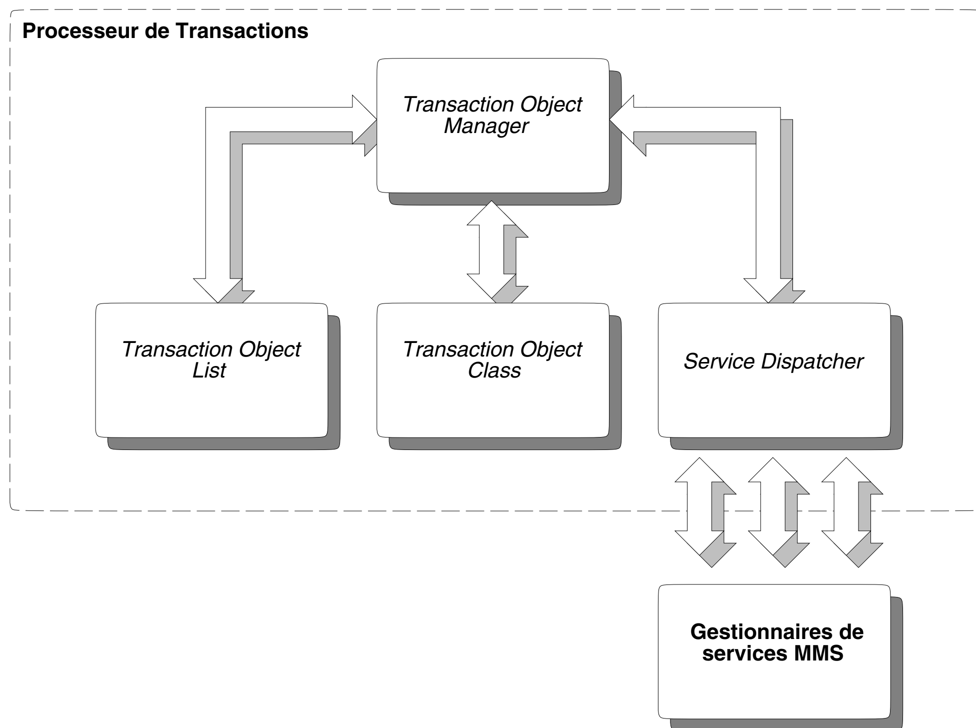


FIG. 4.9 - Structure du Processeur de Transactions

- un *objet liste* chargé d'offrir des fonctions de gestions de listes des instances d'objets manipulateurs. En général, ces objets exportent les opérations classiques du type insérer, retirer, vide, etc.

- un *objet contrôleur* qui, au moyen des opérations fournies par les objets manipulateur et liste, effectue le traitement de l'information en tant que tel et implante souvent l'exécution même des services MMS.

L'association de ces trois objets en forme un nouveau qui peut être vu comme une base de données d'objets MMS offrant des opérations permettant l'exécution des service MMS ainsi que des opérations internes de manipulation des objets MMS.

Cette décomposition est systématique à tous les gestionnaires de services. Leur structure respective est donc très proche et cette homogénéité facilite la compréhension de l'architecture globale du serveur.

De plus, tous les gestionnaires de services exportent des opérations point d'accès de deux types selon que le service traité soit bloquant ou non :

- `OpérationImmédiate (entrées, sorties)` : ce type de fonction traite les services immédiats. Les paramètres en entrée contiennent les arguments du service MMS fournis par le client. Les paramètres en sortie contiennent la réponse du service ou les cas échéant, une description de l'erreur qui a eu lieu. Tous les services immédiats sont traités de cette façon.

- `OpérationBloquante (InvokeId, AAId, entrées, sorties)` : ce type de fonction traite les services bloquants. Si l'exécution permet le passage de l'état EN-COURS à NON-EXISTANT (c'est-à-dire que le service n'est pas bloqué) alors les paramètres en entrée `InvokeId` et `AAId` ne sont pas utilisés et l'exécution est identique au cas précédent. Si par contre il y a passage de l'état EN-COURS à BLOQUÉ, alors les paramètres de sortie ne sont pas utilisés. `InvokeId`

et `AAId` servent alors à identifier de façon unique l'information relative au service bloqué dans l'objet gestionnaire.

4.5.2 Traitement particulier des services bloquants

La différenciation entre les deux types d'opérations (bloquante ou immédiate) d'un gestionnaire de services est nécessaire pour éviter qu'un serveur soit suspendu en attente des résultats d'un service bloqué dont la terminaison est imprévisible. Quand les conditions d'exécution d'un service bloquant ne sont pas réunies, le serveur doit pouvoir passer à une autre activité et exploiter ce temps d'attente comme le montre la figure 4.10.

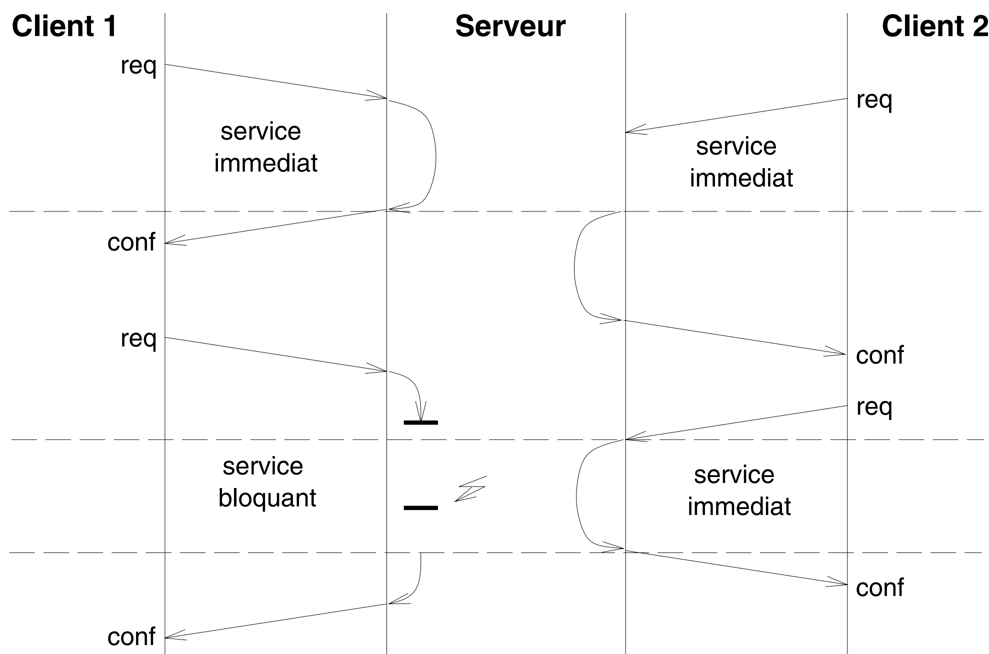


FIG. 4.10 - Exemple d'exploitation du temps d'arrêt d'un service bloquant

Le fait qu'un service soit bloquant ou non a une influence sur la conception de son gestionnaire. En particulier, les gestionnaires de services bloquants contiennent un objet liste qui permet de maintenir l'information relative à tous les services bloqués de ce gestionnaire à un instant donné. Les éléments de cette liste ont la forme suivante :

```
Object: Wait List Element

  Key Attribute: InvokeId
  Key Attribute: Application Association Identifier
  Attribute: Confirmed Service Response
```

Les deux premiers attributs permettent comme pour les objets transaction d'identifier le service bloqué de façon unique dans la liste ainsi que de retrouver l'objet transaction correspondant dans la file associée à l'état BLOQUÉ. L'attribut `Confirmed Service Response` reçoit les résultats de l'exécution du service lorsque celui-ci est débloqué.

Il faut noter que pour le gestionnaire de sémaphores cet élément et sa liste existe déjà. Ils sont en fait représentés par l'objet "Semaphore Entry" et la liste des entrées de sémaphore dans l'état `QUEUED`.

La gestion de cette liste est implicitement prise en compte par le comportement normal du serveur MMS lors des services `Take Control`, `Relinquish Control` et du modificateur `Attach To Semaphore`.

Les opérations bloquantes exportées par les gestionnaires vont donc créer un objet du type `Wait List Element` et l'initialiser au moyen des paramètres `InvokeId` et `AAId`. Cet objet est ensuite inséré dans sa liste en attente des conditions débloquant le service. A part pour le service `Take Control`, le déblocage d'un service n'est pas du ressort du serveur MMS. Nous ne les étudions donc pas. Il appartient à l'activité concurrente 3 "gestion des service bloquants" de détecter que les conditions de déblocage sont satisfaites. L'opération `Process Blocked Transaction` du Processeur de Transactions est ensuite appelée pour rétablir l'objet transaction identifié par les paramètres `InvokeId` et le `AAId` du service débloqué dans l'état PRÊT. Quand son tour arrive l'opération `Process Transaction` appelle le gestionnaire correspondant pour terminer le traitement du service et détruire l'objet `Wait List Element`. Ensuite, l'objet transaction est supprimé et la réponse du service débloqué retournée au client concerné. Si le service bloqué entraînait le blocage d'autres services, alors ceux-ci sont remis dans l'état PRÊT.

4.5.3 Gestionnaires particuliers

4.5.3.1 Le gestionnaire d'associations

Le but du gestionnaire d'associations est de maintenir l'information sur les associations actives, d'autoriser la création de nouvelles associations, de permettre leur terminaison et de négocier les paramètres d'établissement des associations.

Une association entre un client et un serveur est représentée par l'objet suivant :

```
Object: Association

  Key Attribute: Application Association Identifier
  Attribute: Last InvokeId
  Attribute: Client Parameters
  Attribute: Server Parameters
```

L'attribut `Application Association Identifier` permet d'identifier l'association. `Last InvokeId` donne la valeur de l'attribut `InvokeId` du dernier service effectué sur cette association. Les attributs `Client Parameter` et `Server Parameter` permettent de connaître les caractéristiques de l'association.

Le gestionnaire d'associations maintient la liste des objets `Association` correspondant aux associations actives. Les principales opérations exportées sont les suivantes :

- `NewAssociation` pour créer une nouvelle association
- `CloseAssociation` pour détruire une association
- `Negociate` pour effectuer la négociation des paramètres lors de la création d'une nouvelle association
- `Exist` pour savoir si l'association correspondant à l'`AAId` passé en paramètre existe toujours
- `GetAssociation` pour avoir l'`AAId` correspondant à l'adresse du client passée en paramètre

Le gestionnaire d'association peut faire appel à tous les gestionnaires de services manipulant des objets MMS de visibilité `application association specific` (variables, sémaphores, événements et journaux) pour détruire ces objets en cas de rupture d'une association.

4.5.3.2 Le gestionnaire de temporisateurs

Le gestionnaire de temporisateurs est accessible par tous les gestionnaires de services qui ont besoin de déclencher une action après l'écoulement d'un certain laps de temps.

Chaque temporisateur est représenté par un objet du type :

```
Object: Timer

  Key Attribute: InvokeId
  Key Attribute: Application Association Identifier
  Attribute: Remaining Delay
```

Pour manipuler les objets `Timer`, le gestionnaire de temporisateurs exporte les opérations suivantes :

- `NewTimer` crée un nouveau temporisateur, initialise sa valeur de départ et l'attache à un objet transaction ou "Semaphore Entry" au moyen des paramètres `InvokeId` et `AAId`.
- `GetRemainingDelay` retourne la valeur courante du temporisateur spécifié.
- `StartTimer` démarre le décompte du temporisateur spécifié.
- `StopTimer` arrête le décompte du temporisateur spécifié.
- `DeleteTimer` détruit le temporisateur spécifié.
- `ExistTimers` permet de savoir si des temporisateurs sont utilisés ou non.

Il appartient à l'activité concurrente 4 "gestion des temporisateurs" de consulter les régulièrement la valeur des temporisateurs pour savoir si ceux-ci sont passés à zéro.

4.5.4 Relations de dépendance

Les relations et donc les communications entre les gestionnaires de services découlent directement des relations entre objets MMS. Les objets MMS ont en effet pour caractéristique d'être fortement dépendant les uns des autres. Beaucoup d'objets MMS contiennent des références à d'autres objets MMS ou même des listes d'autres objets [Ple94a]. Ceci signifie qu'un service agissant sur une classe donnée d'objets peut fort bien avoir un impact sur d'autres objets MMS qui ne sont pas nécessairement contrôlés par le même gestionnaire que celui qui sert ce service. Il apparaît donc important de déterminer précisément et de façon exhaustive les relations entre les objets MMS pour pouvoir comprendre l'influence qu'ils ont les uns par rapport aux autres. Un exemple typique d'un tel cas est la destruction d'un domaine : tout objet référencé par le domaine à détruire doit aussi être effacé.

La figure 4.11 montre les relations de dépendance entre objets MMS. Une flèche d'un objet A vers un objet B implique que A contient une référence à B, donc qu'un service agissant sur A peut aussi agir sur B. Pour des raisons de clarté, une flèche d'un objet A vers un cadre signifie que A contient

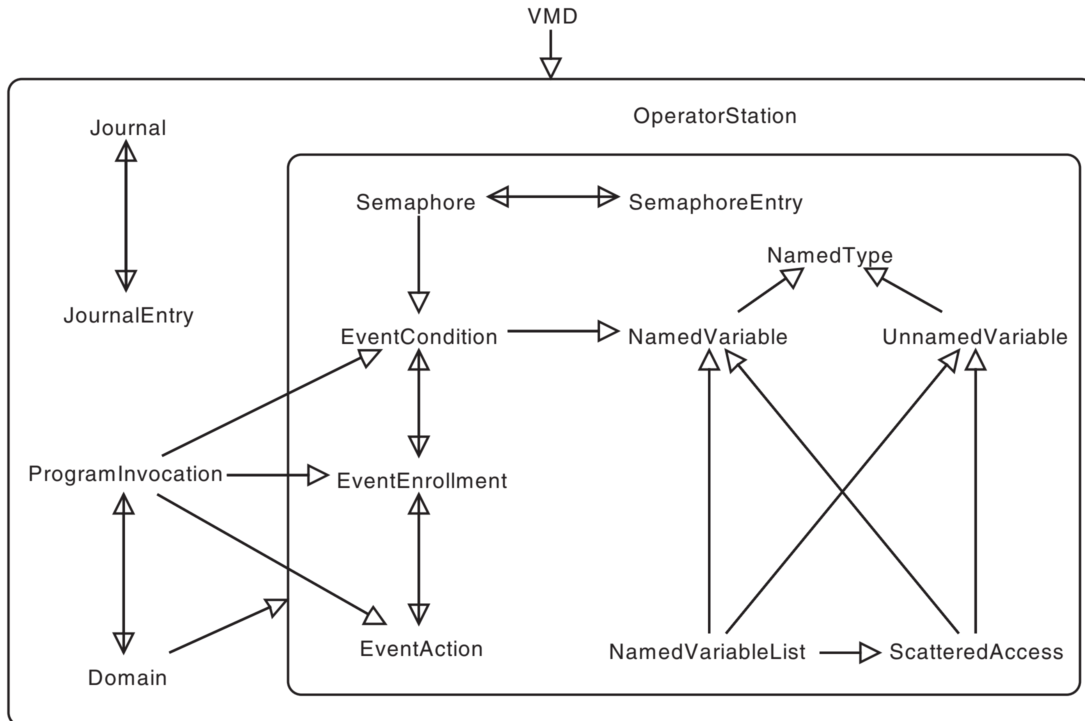


FIG. 4.11 - Relations de dépendance entre les objets MMS

une référence vers tous les objets inclus dans le cadre. Ceci s'applique aux objets VMD et Domain qui référencent presque tous les autres objets MMS. Le mot *référence* ne fait aucune supposition quant à l'implantation de la relation qu'il représente. Il doit être considéré comme une simple façon de faire un lien entre deux objets. Ce n'est donc pas nécessairement un pointeur direct vers l'objet référencé mais peut être le nom de ce dernier par exemple.

Notre architecture interdit volontairement les pointeurs vers les objets MMS car un gestionnaire donné ne doit être autorisé ni à avoir une visibilité directe sur les objets maintenus par un autre gestionnaire ni à instancier de lui même un tel objet. Au contraire, si le gestionnaire 1 désire accéder à un objet appartenant à un gestionnaire 2, il doit explicitement effectuer la requête au gestionnaire 2 c'est-à-dire appeler l'opération qui lui fournira le service requis. Ceci justifie entre autre que les opérations exportées par les objets gestionnaires ne se limitent pas aux points d'accès aux services MMS mais comprennent également des opérations internes utilisées pour la communication entre gestionnaires de services. Par exemple, d'après la norme [ISO90a], la définition d'un sémaphore MMS implique la création d'un objet EC ayant le même nom. Le gestionnaire de sémaphores ne doit pas être autorisé à instancier l'EC lui même mais doit demander au gestionnaire d'événements de le faire pour lui.

Des relations de dépendance entre gestionnaires apparaissent aussi de par la définition de la visibilité des objets MMS. Un gestionnaire de services gère tous les objets d'une classe donnée d'objets MMS quelque soit la visibilité respective de ces objets. En conséquence certains services de manipulation de domaines peuvent nécessiter un accès vers les gestionnaires d'objets MMS susceptibles d'avoir une visibilité `domain specific`. Réciproquement certains services de ces gestionnaires peuvent nécessiter un accès au gestionnaire de domaines. Quant à la visibilité `application association specific` elle implique un accès du gestionnaire d'associations aux gestionnaires pouvant accueillir des objets MMS ayant cette visibilité notamment pour le service `Abort` où il s'agit alors de détruire tous les objets MMS dont la visibilité est limitée à l'association interrompue.

4.5.5 Comportement des gestionnaires simples

Un des buts de notre architecture est de permettre un comportement homogène et simple de tous les gestionnaires de services. La plupart de ces gestionnaires se comportent donc de façon identique. La majorité des services MMS étant des services immédiats, les opérations des gestionnaires sont appelées, effectuent le service MMS considéré et retournent les résultats sans autres difficultés. Les services non confirmés s'effectuent de la même façon à cette différence près que ceux-ci ne correspondent pas à une requête de la part d'un client mais la décision de leur exécution est prise en interne par le serveur. Les caractéristiques principales de tous les gestionnaires et de leurs services sont résumées dans l'annexe D.

On soulignera toutefois les particularités suivantes propres à chaque gestionnaire :

- Le gestionnaire de VMD peut faire appel à n'importe quel gestionnaire de services pour les besoins des services `Get Name List` et `Rename`. Chaque gestionnaire est responsable de son groupe d'objets MMS et exporte donc des opérations internes `Get Name List` et `Rename` appelées par les opérations point d'accès de mêmes noms du gestionnaire de VMD.

- Le gestionnaire de domaines exporte une opération interne permettant la vérification de l'existence d'un domaine et appelée par les gestionnaires qui supportent la création d'objets MMS de visibilité `domain specific` (variables, sémaphores et événements) ou par le gestionnaire de programmes lors de la création d'un PI.

- Les gestionnaires de variables, sémaphores, journaux et événements exportent des opérations internes autorisant la destruction des objets MMS dont la visibilité est `application association specific`. Celles-ci sont appelées par le gestionnaire d'associations.

- Les gestionnaires de variables, sémaphores et événements exportent des opérations internes autorisant la destruction des objets MMS dont la visibilité est `domain specific`. Celles-ci sont appelées par le gestionnaire de domaines.

- Le gestionnaire de sémaphores exporte deux opérations internes appelées par le Processeur de Transactions. L'une permet la prise de contrôle d'un sémaphore par un modificateur, l'autre autorise la destruction d'un objet "Semaphore Entry" lié à un modificateur.

- Le gestionnaire d'événement exporte trois opérations internes appelées par le gestionnaire de sémaphores ou de programmes et qui autorisent la création/destruction d'un EC et le déclenchement d'un événement. Deux autres opérations internes appelées par le Processeur de Transactions permettent la création/destruction d'un EE traitant un modificateur

- Le gestionnaire de variables exporte un opération interne autorisant la lecture de variables booléennes et appelée par le gestionnaire d'événements lors de la scrutation des ECs.

Ces communications entre gestionnaires proviennent des relations de dépendance décrites précédemment. Nous les résumons dans la table 4.3 où nous précisons également si la communication entre deux gestionnaires est une simple consultation ou une modification. Cette table doit se lire de la façon suivante. Un "R" dans l'intersection d'une ligne et d'une colonne signifie que le gestionnaire figurant dans la ligne nécessite un accès en lecture seule à un objet maintenu par le gestionnaire figurant dans la colonne. Un "W" identifie de façon identique un accès en écriture. Une case vide signifie qu'aucun accès n'est requis.

La table 4.3 fait apparaître et souligne le besoin de protéger les objets MMS contre des accès conflictuels si ces accès n'étaient pas automatiquement sérialisés par la politique non-préemptive choisie pour l'ordonnancement des activités concurrentes et donc si les services MMS pouvaient être exécutés

	Association	VMD	Domaine	Prog.	Variable	Sémaphore	Opérateur	Evént.	Journal
Association					W	W		W	W
VMD		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Domaine			R/W		R/W	R/W		R/W	
Programme			R	R/W				W	
Variable			R		R/W				
Sémaphore			R			R/W		W	
Opérateur							R/W		
Evénement			R		R			R/W	
Journal									R/W

TAB. 4.3 - Communications entre gestionnaires

en parallèle (c'est-à-dire dans le cas où plusieurs services peuvent se trouver dans l'état EN-COURS à un instant donné) ou pseudo-parallèle (c'est-à-dire dans le cas où certains services sont interrompus pour permettre le traitement d'autres services).

Nous consacrons la section suivante au gestionnaire d'événements dont l'architecture est assez différente de celle des autres.

4.5.6 Le gestionnaire d'événements

Le gestionnaire d'événements est le plus complexe de tous les gestionnaires MMS. Il se compose des trois objets suivants :

- `Event Object Manager` (EOM) ou *Gestionnaire d'Objets Evénementiels* chargé de la manipulation des objets MMS d'événements et de l'exécution des services d'événements. Il exporte les opérations point d'accès aux services MMS.
- `Event Monitor` (EM) ou *Moniteur d'Evénements* chargé de la détection des événements de type `MONITORED` par consultation périodique des ECs et des variables associées.
- `Event Transition Processor` (ETP) ou *Processeur de Transitions d'Evénements* chargé du traitement des événements une fois que ceux-ci ont été détectés ou déclenchés.

Ces trois objets se partagent l'information représentée par les trois objets MMS EC, EE et EA. La figure 4.12 illustre les relations entre ces objets.

4.5.6.1 Le Gestionnaire d'Objets Evénementiels

L'EOM peut être considéré comme une base de données dont les éléments sont les trois objets événementiels MMS. Le rôle de l'EOM est de fournir les primitives nécessaires à la manipulation de ces objets. Les demandes de manipulation sont effectuées aussi bien de façon externe par le Processeur de Transactions qui appelle une méthode point d'accès à un service que de façon interne par un des deux autres objets composant le gestionnaire d'événements.

L'EOM a donc une structure identique aux objets base de données des autres gestionnaires. Il contient trois objets listes et trois objets manipulateurs correspondant à chacun des objets MMS EC, EE et EA. Un module de traitement permet l'exécution des services MMS.

Nous avons souligné dans [CP93] l'importance de l'EOM pour centraliser les accès aux objets MMS et pour protéger ces derniers contre des accès concurrents dans le cas où la préemption des activités

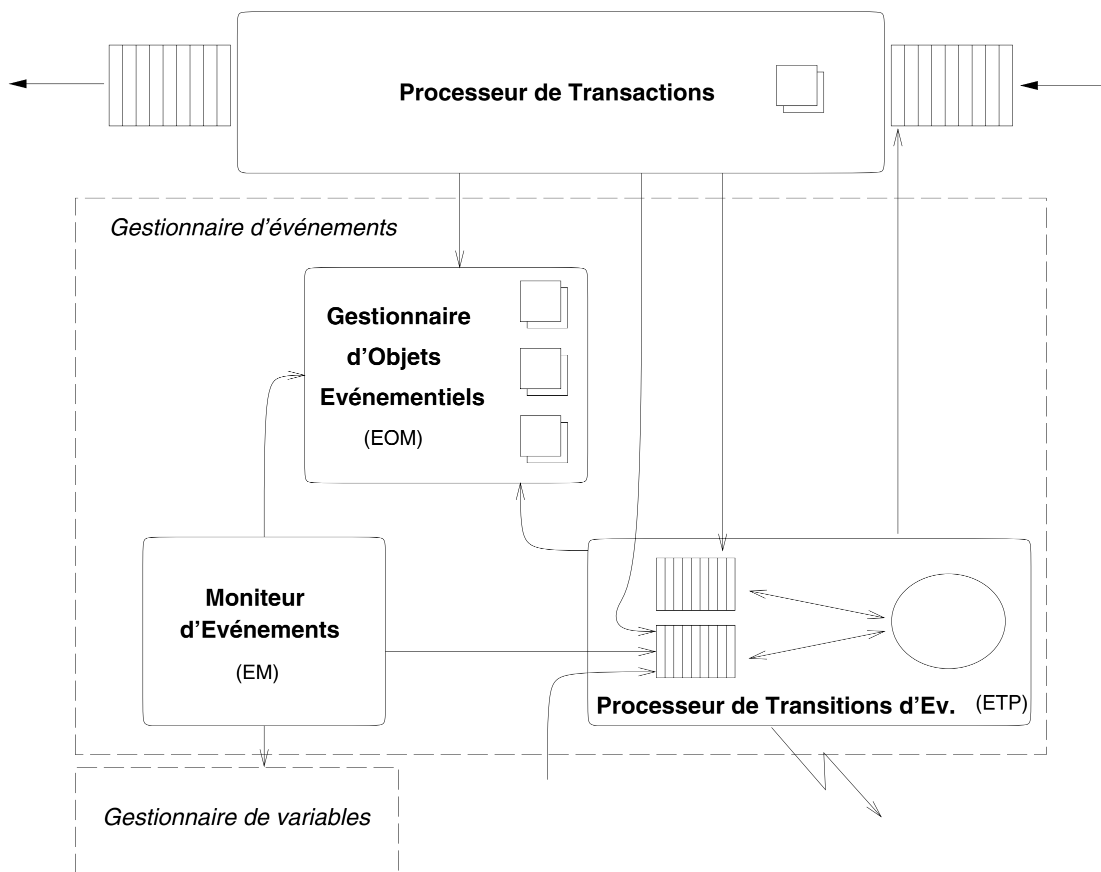


FIG. 4.12 - Gestionnaire d'événements

2 "exécution des services", 5 "détection des événements" et 6 "traitement des événements" était autorisée. Cette dernière caractéristique de l'EOM n'est pas nécessaire ici puisque nous sérialisons automatiquement les accès aux objets MMS de par l'ordonnancement choisi.

4.5.6.2 Le Moniteur d'Evénements

Le Moniteur d'Evénements est chargé de la détection des événements de type scruté. Cette détection se fait par évaluation cyclique des ECs et des variables associées à ces ECs. L'EM ne consulte que les ECs dont la scrutation a été activée par le service `Alter Event Condition Monitoring`. L'EOM maintient une liste des ECs dont la scrutation est active. Cette liste facilite le processus de scrutation car il n'est pas nécessaire de s'occuper des ECs qui ne sont pas scrutés. Il appartient à l'EOM de faire entrer et sortir les ECs de cette liste selon les requêtes `Alter Event Condition Monitoring` effectuées.

Pour aider le serveur à évaluer les ECs, les applications client peuvent fournir aux ECs un intervalle de scrutation (au moyen du paramètre `Evaluation Interval`) ainsi qu'une priorité (paramètre `Priority`). L'intervalle de scrutation représente la durée minimum pouvant s'écouler entre deux consultations d'un EC. Ces deux paramètres sont appelés *paramètres de scrutation*.

Chaque activation de l'EM par l'ordonnanceur implique la consultation d'un certain nombre d'ECs. Le choix des ECs à consulter se fait en fonction des paramètres de scrutation. Nous traitons de la politique d'ordonnancement des ECs et de l'utilisation de ces paramètres au chapitre 6. Pour chaque EC, l'EM consulte l'EOM pour recevoir l'état de l'EC et consulte également le gestionnaire de variables MMS pour obtenir la valeur de la variable booléenne associée à cet EC (figure 4.12). L'EM déclenche

l'événement selon les modalités dont nous avons déjà parlé à la section 3.3.2.

La détection d'un événement implique son traitement par l'ETP. La communication d'un événement entre l'EM et l'ETP se fait au moyen d'un objet file d'attente appelée `Event Queue` où l'information représentant l'événement est insérée sous forme d'un objet `Event Notification` décrit à la section suivante. L'ETP n'a plus alors qu'à retirer cet événement et à le traiter. La communication entre l'activité concurrente 5 "détection des événements" et 6 "traitement des événements" est donc assurée par `Event Queue`. Cette file d'attente reçoit aussi les événements déclenchés ce qui permet de les traiter comme les événements scrutés puisqu'il n'existe pas de différence fondamentale entre la façon de traiter un événement scruté ou déclenché. L'exécution du service `Trigger Event` se résume donc à créer puis insérer un objet `Event Notification` dans `Event Queue`[4]. Un événement déclenché peut également survenir de façon interne et indépendante d'un client MMS, c'est-à-dire sans l'usage du service `TriggerEvent`. Nous considérons dans ce cas que cet événement est automatiquement inséré dans `EventQueue`. Ce modèle s'inscrit dans la nécessité de proposer une architecture générique des serveurs MMS et s'adapte à toutes les façons dont un événement peut être déclenché localement.

Ce modèle de gestion des événements dans un serveur est à rapprocher du modèle CORBA combinant les styles "push" et "pull" avec l'EM étant le producteur, l'`Event Queue` étant le canal d'événements (event channel) et l'ETP le consommateur [Gro94]. Ainsi l'EM produit les événements et les dépose dans la file d'attente. L'ETP consulte cette file et consomme les événements qui y figurent quand elle n'est pas vide.

Plusieurs événements peuvent surgir à des moments rapprochés et donc demander à être traiter "en même temps". Cela revient à dire qu'il y a plusieurs objets `Event Notification` dans `Event Queue`. Nous avons déjà vu que certains événements peuvent être plus prioritaires que d'autres et que la différenciation se fait grâce à l'attribut `Priority` de l'EC représentant chaque événement. Nous organisons donc la file d'attente `Event Queue` par ordre de priorité. L'événement situé en tête de file est le plus prioritaire et c'est celui qui sera retiré par l'ETP lors de l'activation de l'activité concurrente 6 "traitement des événements".

4.5.6.3 Le Processeur de Transitions d'Evénements

L'ETP est directement lié à l'activité concurrente 6 "traitement des événements". Cet objet prend en charge les sept procédures menant à la notification d'un événement lorsque ce dernier est survenu. Ses rôles essentiels sont donc :

- d'ordonner les traitements d'événements selon la priorité des ECs qui leur ont donné lieu dans la mesure où plusieurs événements peuvent survenir plus ou moins au même moment;
- de demander au Processeur de Transactions d'exécuter les actions événementielles possiblement rattachées aux événements (modélisé en fait par un envoi de l'action dans `QueueReceptions`);
- de demander l'établissement d'une association avec le(s) client(s) à notifier dans le cas où celle n'existe pas au préalable;
- d'envoyer la notification de l'événement aux clients correspondant aux seuls EEs ayant souscrit à la transition.

La table 4.4 résume les caractéristiques des sept procédures définies par la norme et devant mener à la notification d'un événement. Nous mettons "oui" dans la colonne "Interruption" pour signifier que

[4] Le service `Trigger Event` peut aussi changer la priorité de l'EC sur lequel il s'applique. Dans ce cas, il faut aussi appeler l'EOM pour modifier cet EC.

l'exécution d'une procédure ne peut se faire par l'ETP et interrompt donc le traitement de l'événement qui reste en attente des résultats de cette procédure.

Procédure	Exécutions/év.	Interruption	Commentaire
1 - EC Update	1 fois	non	Mise-à-jour du temps d'occurence de l'événement
2 - EC Value Capture	1 fois	non	Capture d'attributs nécessaires à la notification
3 - EE Value Capture	Autant que d'EEs	non	Capture d'attributs nécessaires à la notification
4 - EE Update	Autant que d'EEs	non	Mise-à-jour du temps de transition de l'événement. Fin du traitement pour les EEs de type `MODIFIER`
5 - Action Execution	Autant que d'EEs avec un EA	oui	Exécution de l'action événementielle
6 - AA Establishment	Autant que d'EEs contenant les adresses de client sans AA	oui	Etablissement de l'association
7 - Notification Invok.	Autant que d'EEs de type `NOTIFICATION`	non	Construction et envoi de la notification

TAB. 4.4 - Procédures de traitement des transitions d'événement

Pour maintenir l'information nécessaire au traitement des sept procédures gérant une transition d'événement, nous avons introduit à la section précédente l'objet `Event Notification` ou EN (fig. 4.13). Celui-ci est créé lorsqu'un événement est détecté ou déclenché. A chaque événement qui s'est produit correspond donc un EN même si deux événements proviennent du même EC. L'objet EN est détruit lorsque toutes les notifications correspondant à son événement ont été envoyées et/ou que tous les EEs de type `MODIFIER` ont été traités.

La première série d'attributs correspond à ceux capturés aux objets EC et EE lors des procédures ETP 2 et 3. Ces attributs sont inclus dans la notification. Les attributs suivants sont utilisés pour le traitement de l'objet EN au cours des sept procédures ETP. Tous ces attributs correspondent à des attributs MMS déjà présents dans les objet événementiels ou alors à des paramètres du service `Event Notification`.

L'ETP contient un objet manipulateur des objets ENs et deux objets liste :

- `Event Queue` reçoit les caractéristiques des événements à traiter sous la forme d'objets ENs;
- `Waiting Event Queue` reçoit les objets ENs dont la notification correspondante est en attente des résultats d'une action événementielle (procédure ETP 5) ou de l'établissement d'une association (procédure ETP 6).

4.6 MMS et systèmes d'exploitation

Nussbaumer ([Nus91b] p. 298) a remarqué que le traitement des événements MMS est comparable à une fonction de système d'exploitation. En fait c'est l'ensemble de la norme MMS elle-même qui semble inspirée par les fonctions de systèmes d'exploitation centralisés ou répartis.

Comme nous l'avons déjà vu pour les événements MMS, il est possible de faire une analogie entre clients MMS et processus d'un système d'exploitation, entre un serveur et le noyau d'un tel système, entre l'équipement industriel sous-jacent à la VMD et l'architecture matérielle pour laquelle le système d'exploitation est conçu.

```
Object: Event Notification

  -- Attributs inclus dans la notification

  Key Attribute: Event Condition Name
  Attribute: Event Enrollment Name
  Attribute: Severity
  Attribute: State
  Attribute: Transition Time
  Attribute: Notification Lost
  Attribute: Alarm Acknowledgment Rule
  Attribute: Event Action Name
  Attribute: Action Results

  -- Attributs utilisés pour le traitement des ENs

  Attribute: Application Association Local Tag
  Attribute: Client Application
  Attribute: Confirmed Service Request
  Attribute: Duration
  Attribute: Event Condition Class
  Attribute: Event Condition State
```

FIG. 4.13 - L'objet Event Notification

A partir de là, on peut tracer un parallèle clair entre MMS et systèmes d'exploitation (OS)[5]. Le concept de VMD dans MMS fournit une abstraction de tous les équipements industriels utilisés dans l'environnement MMS. Un OS offre lui le concept de machine virtuelle qui permet de percevoir la machine à un niveau d'abstraction plus élevé et de l'exploiter de façon plus simple et homogène. MMS correspond d'ailleurs tout à fait à la définition suivante d'un OS donnée par Krakowiak dans [Kra87] :

> *Un système d'exploitation remplit une double fonction : présenter à ses utilisateurs une machine virtuelle facilitant l'écriture et l'exécution des applications, et fournissant un ensemble de services; gérer l'ensemble des ressources matérielles et logicielles, en assurant l'utilisation optimale et le partage équitable du matériel, la protection[6] et la conservation sûre de l'information.*

Les systèmes d'exploitation présentent très souvent une décomposition sous forme de couches superposées ou concentriques. Un serveur MMS peut aussi se visualiser sous forme de couches de niveaux d'abstraction différents [CP95a]. On peut en distinguer au moins quatre :

1. la couche machine (ordinateur);
2. la couche de base pour accéder à la machine (pilotes ou “drivers”);
3. la couche implantant les services MMS (corps du noyau);
4. l'interface aux services MMS (interface aux appels systèmes de l'OS).

[5] Abbréviation de “Operating System” pour ne pas confondre avec SE “Semaphore Entry”!
[6] La protection de l'information au sens sécurité est toutefois un aspect totalement absent de MMS.

Cette décomposition est illustrée sur la figure 4.14.

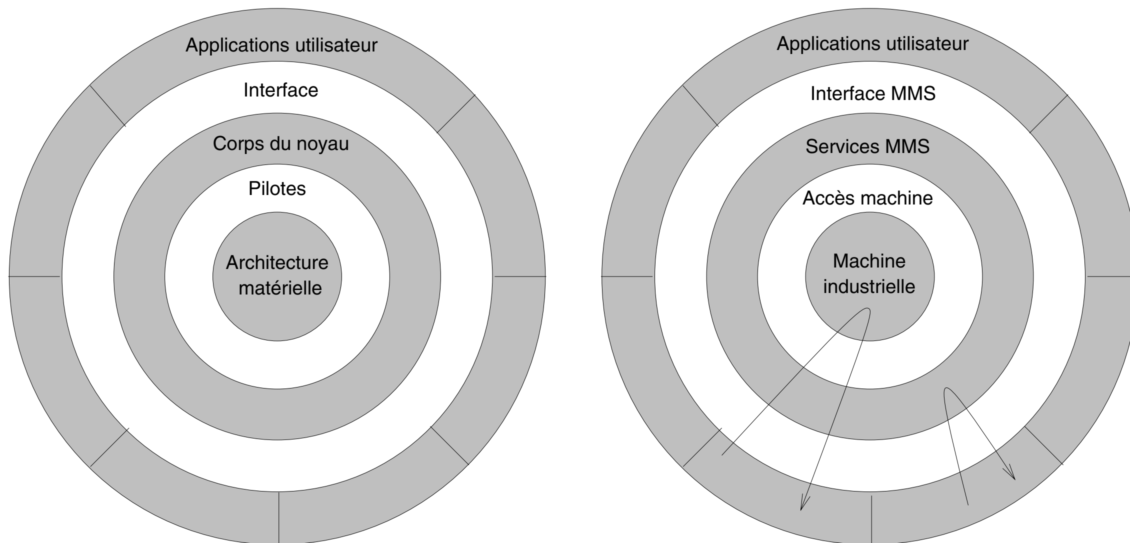


FIG. 4.14 - Structure simplifiée des systèmes d'exploitation et de MMS

A l'image d'un processus dans un OS, l'état dans lequel se trouve une application client MMS peut être décrit par la machine d'états de la figure 4.5. Cette dernière se compare à l'état d'un processus lors d'un appel système puisqu'il s'agit en fait de l'état d'une requête de service dans un serveur. En y remplaçant l'état NON-EXISTANT par l'état EN-COURS-UTILISATEUR on obtient une machine qui traduit à tout moment l'état du client. Cet état correspond à l'exécution propre à une application client qui n'est pas en attente d'une réponse d'un serveur MMS. Ceci se rapproche des états "USER-RUNNING" et "KERNEL-RUNNING" d'un processus UNIX [Bac86] qui traduisent qu'un processus est soit en cours d'exécution d'un appel du noyau (requête de service du serveur) soit en exécution hors du noyau.

On s'aperçoit également que les services que l'on trouve dans MMS présentent une forte analogie avec les appels systèmes et les fonctions propres aux systèmes d'exploitation :

- Les services de gestion de la VMD sont comparables à ceux permettant d'obtenir des informations sur le type d'OS (par exemple son numéro de version) mais aussi de connaître l'état d'une machine donnée (active, en panne, chargée, etc).
- La gestion de variables est le moyen le plus simple pour obtenir ou modifier les informations contenues dans un serveur tout comme un processus peut vouloir accéder à certaines informations ou zones mémoires d'une machine en passant par le système d'exploitation.
- La gestion de domaines peut se voir comme une manipulation des espaces mémoires de certains processus. Elle offre des possibilités de téléchargement de données ou de programmes d'une machine sur une autre. On imagine tout à fait son utilité dans les OS répartis pour gérer la migration de procesus et la répartition de la charge par exemple.
- La gestion des invocations de programmes MMS permet de contrôler l'exécution de programmes en offrant des services de création, démarage, arrêt, destruction, etc. Un OS dispose souvent d'appels système identiques pour contrôler les processus. Il y a toutefois une certaine différence avec MMS : les PIs ne représentent pas des clients MMS et sont locaux aux serveurs ou aux dispositifs physiques sous-jacents. Dans le monde OS, ceci signifierait que certains processus font partie du noyau et sont contrôlables de l'extérieur par d'autres processus. Un PI peut toutefois

être vu comme une partie d'un client qui s'exécute dans le serveur. Et en fait rien n'empêche un client MMS d'être un PI simplement parce que la définition d'un PI n'est pas dans le domaine de discours de MMS.

Une autre analogie avec les OSs vient de la façon dont certains PIs MMS volumineux exécutent leur code lorsque celui-ci est compris dans un domaine trop petit pour contenir l'ensemble du code représentant le PI. Ces PIs travaillent à la façon des “page faults” UNIX. Quand ils arrivent en bout de code, ils s'arrêtent et demandent un téléchargement du domaine correspondant au code suivant à exécuter.

- La gestion de journaux MMS autorise la supervision d'applications et l'enregistrement de données jugées importantes. La plupart des OSs proposent également ce type d'opération pour pouvoir par exemple garder une trace des connexions établies sur une machine donnée, ou des transferts de fichiers effectués.

- La gestion d'événements offre des capacités de signalisation et de synchronisation entre clients MMS de la même façon qu'entre processus d'un OS. De plus, un événement MMS peut surgir suite à une action du dispositif sous-jacent à la VMD comme un événement d'un OS peut arriver suite à une action de l'architecture matérielle (fin d'opération d'entrée/sortie sur un disque par exemple).

- La gestion de sémaphores MMS est aussi typique d'un OS. Les services autorisant la protection de ressources font partie intégrante de la plupart des OS. Dans un OS réparti, certaines ressources du système sont regroupées dans des “pools”. Les sémaphores MMS et particulièrement le sémaphore étiqueté sont tout à fait adaptés à la protection de ces ressources. Toutefois, comme nous l'avons signalé, MMS ne propose pas de méthode pour détecter ou prévenir les interblocages répartis, ce qu'un OS réparti intègre généralement.

- Les services de gestion de stations opérateur bien que très limités dans MMS sont comparables aux fonctions d'entrée/sortie sur terminaux, claviers ou autres périphériques gérés par un OS.

- Un serveur MMS peut agir comme serveur de fichiers ce qui est caractéristique d'un système d'exploitation réparti. Lecture, écriture, création, destruction de fichiers sont des opérations offertes par MMS qui se retrouvent évidemment dans les OSs. La consultation de répertoire est aussi possible avec MMS.

Finalement, le parallèle que nous effectuons ici n'est peut-être pas si surprenant. Il traduit en effet simplement que dans le cas de MMS comme dans celui d'un système d'exploitation, le but ultime est de pouvoir manipuler facilement et de façon homogène des dispositifs physiques différents au moyen d'un modèle abstrait représenté par une machine virtuelle.

On retrouve l'influence des OSs dans notre architecture où plus on descend les niveaux des hiérarchies “inclusion” et “utilise”, plus on trouve des objets élémentaires responsables de fonctions se rapprochant du dialogue avec le dispositif physique. L'architecture peut d'ailleurs être vue comme un ensemble de couches de niveaux d'abstraction croissants. Une étude détaillée de ces différents niveaux a été publiée dans [CP95a].

4.7 Analyse et performances

4.7.1 L'implantation

Un serveur MMS suivant l'architecture proposée dans ce chapitre a été implanté en Ada [Ins83]. Ada a également été choisi pour implanter la pile de protocoles MAP (nommée LITMAP) utilisée par

MMS [SVB93], [VP94], [VP95]. Ce choix est motivé par les avantages considérables qu'offre Ada en termes de facilité et clarté d'implantation, de portabilité du code et de migration sur systèmes embarqués. Par ailleurs, l'intégration harmonieuse du travail effectué par différentes personnes représentait également une exigence importante. L'intégralité de la pile de protocoles MAP/MMS a ainsi pu être réalisée en trois ans par une petite équipe au sein de notre laboratoire. Elle représente plus de 145000 lignes de code[7]. Le serveur MMS quant à lui totalise 16733 lignes de code source. Le code exécutable du serveur nécessite 2 449 408 octets (généré sur une Sun Sparc station ELC avec le compilateur Ada de VERDIX [VER90]).

Nous montrons sur la table 4.5 la taille des parties de code représentant les divers modules de base du serveur. Les gestionnaires de variables et d'événements sont complets et parmi les parties les plus complexes de MMS. Il n'est alors pas étonnant de constater que ces deux gestionnaires consituent à eux-seuls plus de la moitié du code du serveur. Il faut toutefois nuancer ces résultats dans la mesure où tout serveur MMS peut privilégier un gestionnaire particulier aux dépens des autres selon les besoins de l'application.

Nous faisons aussi apparaître sur la table 4.5 pour chaque gestionnaire les services MMS qui n'ont pas été implantés. Ces services sont relativement peu nombreux.

Module considéré	**Taille**	**Pourcentage**	**Services implantés**
Gestionnaire de VMD	141	0.84	Tous.
Gestionnaire de domaines	410	2.45	Uniquement `InitiateDomainSequence` `LoadSegment`, `TerminateDownloadSeq`.
Gestionnaire de programmes	593	3.54	Tous.
Gestionnaire de variables	5021	30.01	Tous. Certains types ne sont pas supportés.
Gestionnaire de sémaphores	1242	7.42	Tous sauf `ReportPoolSemaphoreStatus`.
Gestionnaire de station opérateur	358	2.14	Tous.
Gestionnaire d'événements	3643	21.77	Tous sauf `GetAlarmSummary` et `GetAlarmEnrollmentSummary`.
Gestionnaire de journaux	775	4.63	Tous.
Processeur de Transactions	1201	7.17	Y compris la gestion des modificateurs.
Utilitaires divers, types MMS	3349	20.01	-
Total	16733	100	-

TAB. 4.5 - Taille de chaque objet principal constitutif du serveur

La difficulté de l'implantation d'un serveur MMS provient essentiellement des nombreuses options possibles de chaque service MMS. La valeur prise par un paramètre dans une requête, la présence ou l'absence d'un paramètre sont autant de combinaisons qui rendent fastidieuse l'implantation de la procédure traitant cette requête et augmente la probabilité d'erreurs.

Il existe toutefois un comportement commun et systématique à la plupart des requêtes de services confirmés. Ce comportement nous a permis de définir une architecture modulaire où le Processeur de

[7] Comptées sur les ";".

Transactions représente le code commun au traitement de tous les services MMS et où le code propre à chaque service est confiné dans les gestionnaires de services.

4.7.2 Quelques mesures de performance

Nous avons effectué des mesures *externes* des temps de réponse de certains services MMS de gestion d'événements. Ces mesures sont détaillées dans [CV95]. Le temps de réponse se mesure au niveau des clients et représente la durée qui s'écoule entre l'envoi d'une requête et la réception de la réponse correspondante. Le serveur MMS est alors considéré comme une boîte noire qui reçoit et émet des messages. Ces temps incluent donc le temps de communication dans la pile MAP et sur le réseau physique – dans les deux directions – mais aussi le temps d'exécution dans le serveur. Nous avons également mené des mesures des temps d'exécution dans les serveurs (mesures *internes*).

Ces mesures ont été effectuées entre deux stations de travail Sun Sparc ELC sous SunOS 4.1.1 reliées par Ethernet (10 Mb/s). Pour toutes les mesures, au moins 500 répétitions ont été effectuées mais souvent plus de 2000 pour la plus grande majorité d'entre elles. Dans la suite nous donnons :

- le temps *minimum* réalisé;
- le temps *typique* : c'est la mesure la plus fiable car elle représente la tendance réelle. Lors de nos expériences, nous avons trouvé que la médiane était le temps le plus représentatif [Jai91];
- le temps *moyen* : c'est la moyenne de tous les temps mesurés. Cependant, ce temps n'est pas représentatif des valeurs réelles. Quelques échantillons seulement ont des valeurs proches de la moyenne. Les valeurs généralement élevées des temps moyens sont dues à des échantillons isolés qui éloignent la moyenne de la valeur typique. Il est souvent difficile de fournir une borne supérieure des temps de réponse car celle-ci dépend des autres processus s'exécutant dans le système, des erreurs de transmissions et d'événements aléatoires tels que fautes de page et les commutations entre processus UNIX [Cri89].

La figure 4.15 montre une distribution typique du temps de réponse des services MMS. La médiane correspond au pic. La répétition systématique des mesures a montré une bonne stabilité de la médiane mais de grandes variations de la moyenne.

4.7.2.1 Mesures externes

Le tableau 4.6 montre les temps de réponses obtenus pour la plupart des services de gestion d'événements MMS. Nous mentionnons aussi les mesures pour les services plus connus `Read` et `Write`.

Le temps de réponse moyen typique se situe autour de 62.6 ms. Le temps moyen minimum est autour de 54.9 ms. Les mesures sont à peu près identiques pour tous les services, essentiellement parce que les messages échangés ont des structures et des longueurs qui diffèrent peu. La plus grande partie de ces temps vient de la transmission dans la pile MAP alors que souvent moins d'une milliseconde est due au serveur soit 1.4 % du temps total. Ceci explique qu'il n'y ait pas de réelle différence entre les services de consultation/destruction d'objets et les services de création. Cette différence est masquée par les temps de communication. Les délais de communication sont surtout le fait du codage/décodage ASN.1 des messages (22.5 % du temps pour un service `Read`). Dans [SVB93], Sidou *et al.* exposent plus de détails concernant les temps d'exécution dans chacune des couches de LITMAP. Dans [LPV94], les mêmes temps sont analysés pour l'architecture réduite Mini-MAP avec un intérêt plus particulier pour le codage/décodage des PDUs.

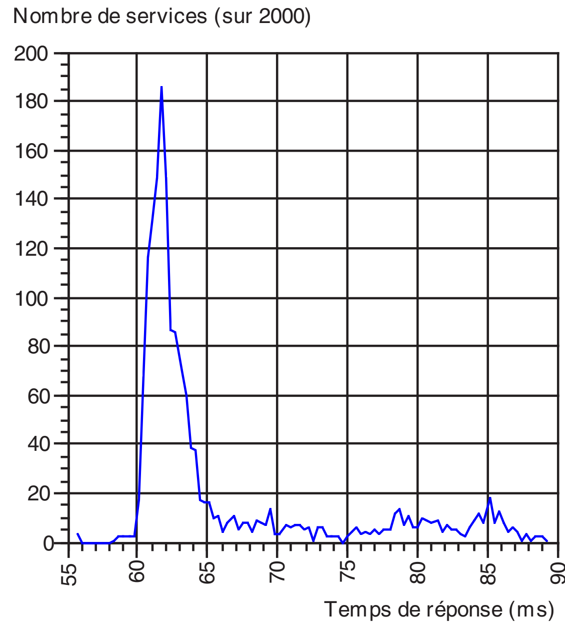


FIG. 4.15 - Distribution des temps de réponse des services MMS

Service MMS	Minimum (ms)	Typique (ms)	Moyen (ms)
DefineEventCondition	55.2	62.4	85.6
DeleteEventCondition	54.2	62.4	84.6
DefineEventAction	55.1	63.4	83.6
DeleteEventAction	54.6	62.2	83.2
DefineEventEnrollment	55.2	63.2	82.7
DeleteEventEnrollment	55.0	62.3	82.6
ReportEventConditionStatus	54.3	62.5	80.5
ReportEventActionStatus	53.7	62.1	81.5
ReportEventEnrollmentStatus	55.5	62.9	82.2
AlterEventConditionMonitoring	56.1	62.6	84.0
Read (1 variable entière)	55.7	65.1	84.1
Write (1 variable entière)	56.5	64.6	86.3
Read (10 variables entières)	76.7	94.1	106.3
Write (10 variables entières)	73.0	80.8	99.3

TAB. 4.6 - Temps de réponses pour certains services d'événements

4.7.2.2 Mesures internes au serveur

Le tableau 4.7 a été obtenu en mesurant le temps total d'exécution d'un service dans le serveur. C'est donc le temps qui s'écoule entre la réception d'une requête et l'envoi de la réponse correspondante. Nous constatons sans grande surprise que les services de création d'objets sont plus longs que ceux de consultation ou de simple modification.

En comparant les tableaux 4.6 et 4.7, il est clair que le temps passé dans le serveur est faible par rapport au temps de communication (en moyenne 878 μs soit 1.4 % du temps de réponse total). Pour

Service MMS	Minimum (μs)	Typique (μs)	Moyen (μs)
DefineEventCondition	989	1019	1109
DeleteEventCondition	808	820	833
DefineEventAction	1238	1277	1450
DeleteEventAction	805	818	829
DefineEventEnrollment	1250	1284	1426
DeleteEventEnrollment	825	837	850
ReportEventConditionStatus	723	736	744
ReportEventActionStatus	604	616	623
ReportEventEnrollmentStatus	715	726	736
AlterEventConditionMonitoring	642	655	662
Read (1 variable entière)	945	961	981
Write (1 variable entière)	1044	1060	1075
Read (10 variables entières)	2540	2578	2617
Write (10 variables entières)	3708	3754	3870

Tab. 4.7 - Temps d'exécution interne de certains services

les services usuels (définition, destruction, consultation d'objets) le temps interne d'exécution se situe entre 600 μs et 1500 μs. En l'absence de la pile de communication le débit potentiel du serveur est environ de 1138 services/s. Considérons un tel serveur lié à un seul client qui envoie continuellement des requêtes de service dès qu'il reçoit la réponse du service précédent. La lenteur de la communication fait qu'en réalité le serveur se trouve au repos pendant 79.2 % du temps en moyenne.

Une bonne solution pour augmenter le débit de services exécutés est de faire usage de l'asynchronisme de MMS. Le client MMS peut alors lancer plusieurs requêtes en même temps. Il n'a pas besoin d'attendre la réponse d'un service pour en envoyer une autre. La figure 4.16 montre le gain de temps obtenu pour le service `Report Event Condition Status` quand celui-ci est utilisé de façon asynchrone.

4.8 Conclusion

Dans ce chapitre nous avons proposé un modèle d'exécution des requêtes de service MMS confirmé. Ce modèle nous a fourni un point de départ pour construire une architecture générique des serveurs MMS. Cette architecture constitue la contribution principale de ce chapitre. Nous avons montré comment intégrer harmonieusement l'ensemble des fonctionnalités d'un serveur MMS au sein d'une même architecture. L'architecture proposée se veut extrêmement modulaire et attribue à chaque objet une responsabilité qui lui est propre. Nous avons étudié l'influence des services bloquants sur l'architecture et montré comment le Processeur de Transactions règle le passage des objets transaction d'un état à un autre. Par ailleurs, nous avons inclus dans cette architecture certains aspects méconnus de MMS tels que la gestion des modificateurs, des événements ou encore des sémaphores.

Nous avons également effectué une implantation d'un serveur MMS basé sur l'architecture proposée. L'expérience acquise lors de ce développement ainsi que la réalisation de tests d'interopérabilité avec des implantations commerciales de MMS nous ont permis de constater qu'il est très facile de rajouter des nouveaux services et qu'il est également relativement aisé d'identifier et de corriger les erreurs éventuelles du serveur.

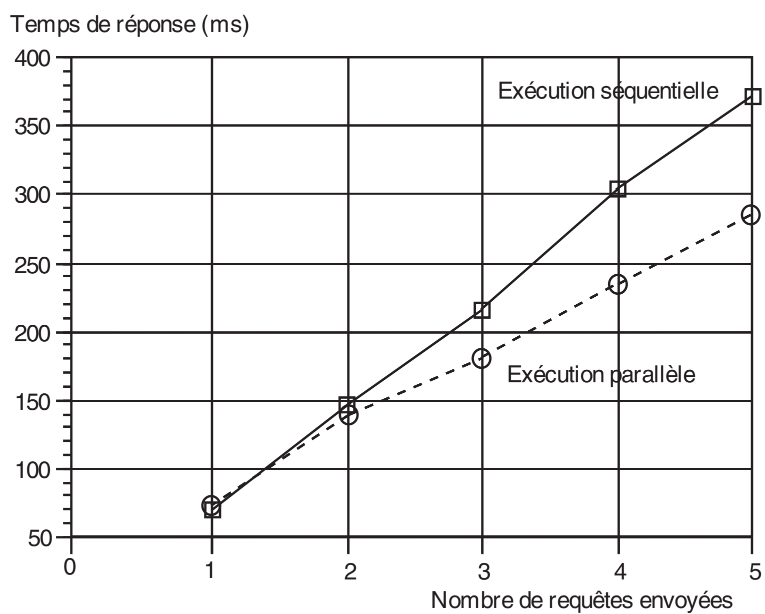


FIG. 4.16 - Exécutions synchrone et asynchrone des services MMS

Chapitre 5

Principales extensions proposées

5.1 Introduction

Les chapitres précédents étaient orientés vers la compréhension et l'analyse de la norme MMS avec notamment une proposition d'architecture des serveurs. Dans ce chapitre nous cherchons maintenant à améliorer certains aspects de la norme MMS. Nous proposons ainsi deux extensions de niveau de modification 2 :

- Une extension à la détection d'événements MMS qui permet aux applications utilisateurs de définir précisément sous forme d'un prédicat la condition devant mener à l'apparition d'un événement. Nous montrons comment s'intègre cette extension à l'architecture des serveurs définie au chapitre précédent.
- Une extension à l'exécution des requêtes de service confirmé qui permet aux utilisateurs de fournir des priorités à leurs requêtes et aux serveurs d'utiliser ces priorités pour ordonner l'exécution des requêtes en cours.

Nous analysons les avantages et inconvénients de ces extensions ainsi que leur influence sur le protocole et les serveurs MMS. Dans la définition de ces extensions nous cherchons toujours à préserver la compatibilité avec l'existant normatif.

5.2 Extension à la détection d'événements MMS

5.2.1 Introduction

Dans toutes les sous-sections de la section 5.2, nous ne considérons que les événements MMS de type scruté. Comme nous l'avons souligné dans le chapitre 2, la définition actuelle de l'occurrence d'un événement est limitée en ce sens qu'un événement MMS n'est déclenché que par la modification d'une variable booléenne. Ce choix limite singulièrement les services d'événements à disposition de l'utilisateur. En effet, on souhaiterait pouvoir définir plus précisément les circonstances de l'apparition d'un événement. Il serait souhaitable par exemple qu'un utilisateur puisse décider qu'une alarme est déclenchée lorsque la température d'un liquide dépasse un certain seuil. Il devient alors nécessaire de pouvoir définir les conditions de déclenchement des événements à partir de prédicats. La présente section a donc pour but de présenter une extension à MMS (*XED* pour "eXtended Event Detection") qui permet aux utilisateurs de définir eux-mêmes les conditions de détection d'un événement [Cas94b].

Pour fixer les idées prenons l'exemple suivant[1]. Considérons une VMD qui modélise simplement un dispositif de chauffage. Un liquide dans une cuve est chauffé jusqu'à une certaine température. Si pour quelque raison la température du liquide dépasse un certain seuil une alarme doit être déclenchée pour informer le client superviseur du problème rencontré. Pour modéliser cette alarme et les actions à effectuer lors de son apparition, nous utilisons les événements MMS. Un EC de classe `MONITORED` est défini pour identifier l'événement. Cet EC est lié a une variable booléenne (la variable scrutée). Si la valeur de cette variable est FAUX alors la température est dans l'intervalle autorisé. Si elle est VRAI, alors la température a dépassé un seuil critique que nous notons T_0. Ici, le dispositif physique modélisé est responsable du changement de la valeur de la variable scrutée. L'application client intéressée par la notification de l'événement considéré doit définir un EE lié à l'EC cité spécifiant qu'elle ne désire recevoir une notification d'événement que pour les transitions `IDLE-TO-ACTIVE`, ce qui représente le passage de la température d'une valeur autorisée à une valeur supérieure à T_0. Si une action doit être effectuée (couper le chauffage, vider la cuve, etc) alors un EA doit aussi être défini.

Quand tous les objets sont définis, on peut commencer la scrutation de l'EC au moyen du service `Alter Event Condition Monitoring`. T dénote la température courante. On démarre avec $T < T_0$ c'est-à-dire que la valeur de la variable scrutée est FAUX. Dès que $T \geq T_0$ le dispositif physique écrit VRAI dans la variable scrutée. La VMD détecte cette transition et lance le traitement de l'événement correspondant. Toutes les applications client qui ont souscrit à l'événement avec la transition `IDLE-TO-ACTIVE` reçoivent une notification (figure 5.1).

Cet exemple simple montre que la détection d'un événement dans MMS dépend de la capacité du dispositif physique modélisé par la VMD à connaître lui-même les conditions de déclenchement de l'événement. En d'autres mots, ces dispositifs doivent être “intelligents”. Cette solution n'est pas souple parce que les utilisateurs ne peuvent pas décider des conditions événementielles à définir et parce que l'événement MMS est complètement dépendant de l'événement réel pré-défini. Les dispositifs physiques doivent donc avoir été conçus pour un scénario d'utilisation particulier et doivent se plier aux exigences de la norme MMS.

5.2.2 XED vu depuis les clients

Supposons donc maintenant que nous disposions du même dispositif industriel que précédemment mais qu'il est cette fois incapable de décider si la température a dépassé le seuil T_0, c'est-à-dire qu'il ne peut mettre à jour la variable scrutée. Ce dispositif contient un capteur qui ne fait que retourner la valeur courante de la température T dans une variable MMS que nous appelons également T. Ce que nous voulons c'est que la VMD elle-même détecte la condition $T \geq T_0$ et démarre le traitement de transition d'événement quand cette condition devient vraie. C'est tout le but de l'extension proposée. XED permet l'évaluation d'expressions basées sur des prédicats et déclenche la notification d'événement en conséquence.

Du point de vue des clients MMS, XED se comporte de la façon suivante. Avant de définir l'objet EC qui sera utilisé pour détecter l'événement, l'application client doit définir une variable (ou utiliser une variable pré-définie) appelée *Variable Condition*. Cette variable remplace la variable scrutée. Sa valeur est de type `VisibleString` et doit contenir sous format texte la condition qui doit être évaluée (par exemple “$T \geq T_0$”). Quand l'objet EC est défini, le nom de la Variable Condition est fourni en lieu et place de celui de la variable scrutée. Le paramètre `MonitoredVariable` du service `Define Event Condition` est utilisé pour cela. Tous les autres paramètres de ce service sont inchangés par rapport à une définition normale d'une condition événementielle dans MMS.

L'expression prédicat contient les noms des variables MMS dont les valeurs sont nécessaires à

[1] Cet exemple est une version simplifiée du scénario d'application présenté à la section F.1.

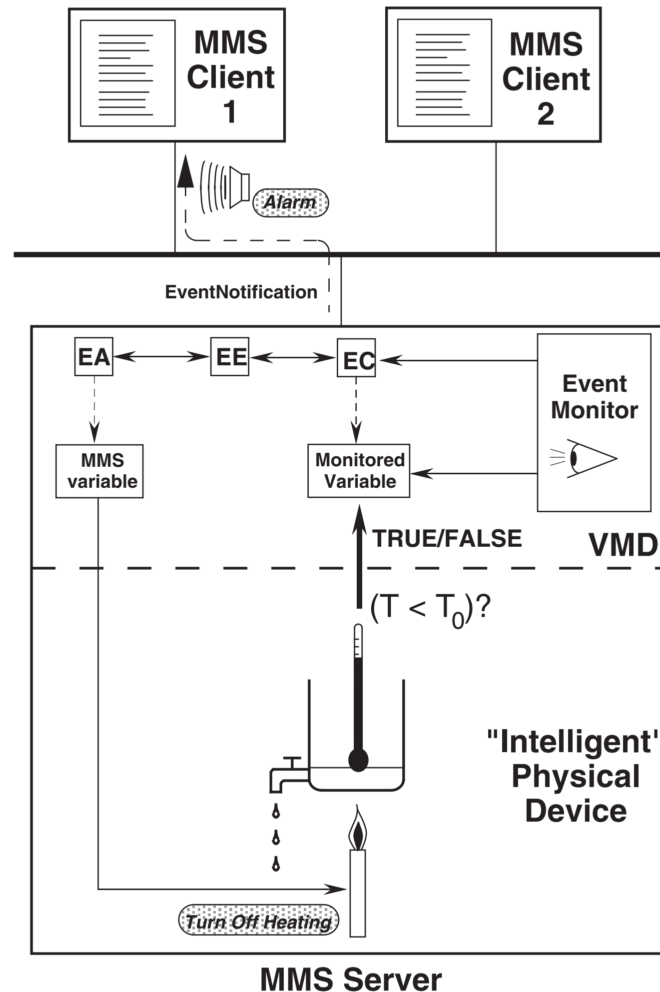


FIG. 5.1 - Un exemple d'utilisation classique des événements MMS

l'évaluation de la condition. Quand la Variable Condition est définie, l'application client doit s'assurer que ces variables existent vraiment. Dans notre exemple, il doit y avoir une variable nommée T ainsi qu'une variable nommée T_0 (voir figure 5.3). T_0 est une variable MMS qui joue le rôle de constante ou de paramètre de l'expression prédicat. Nous la désignons par l'expression *Variable Paramètre*. L'utilisation de la variable T_0 peut être évitée en mentionnant explicitement la température de seuil (par exemple "$T \geq 500$"). Cependant la méthode consistant à avoir une Variable Paramètre est plus souple car la valeur de seuil peut être changée facilement sans altérer la Variable Condition. La configuration des objets MMS est alors celle illustrée sur la figure 5.2.

Une fois que la Variable Condition et l'objet EC sont définis, les requêtes de service effectuées par l'application client sont identiques à celles d'une utilisation classique des événements MMS. On utilise donc le service `Alter Event Condition Monitoring` pour démarrer la scrutation de l'EC considéré. Quand ce service est traité, l'expression prédicat est évaluée. Si elle est vraie, alors l'attribut `State` de l'EC est mis à `ACTIVE`. Sinon, cet attribut est mis à `IDLE`. L'utilisateur souscrit toujours aux mêmes transitions d'événement. Ces transitions sont inchangées par rapport à MMS classique. Par exemple si un utilisateur souscrit à la transition `ACTIVE-TO-IDLE` alors une notification d'événement lui est envoyée quand la VMD trouve l'attribut `State` de l'EC à `ACTIVE` et que l'évaluation de l'expression prédicat donne pour résultat FAUX. Dans notre exemple, un utilisateur qui désire être informé du dépassement du seuil de température souscrit à la transition `IDLE-TO-ACTIVE`. La table 5.1 compare XED et MMS classique en fonction des circonstances qui déclenche les transitions d'événements.

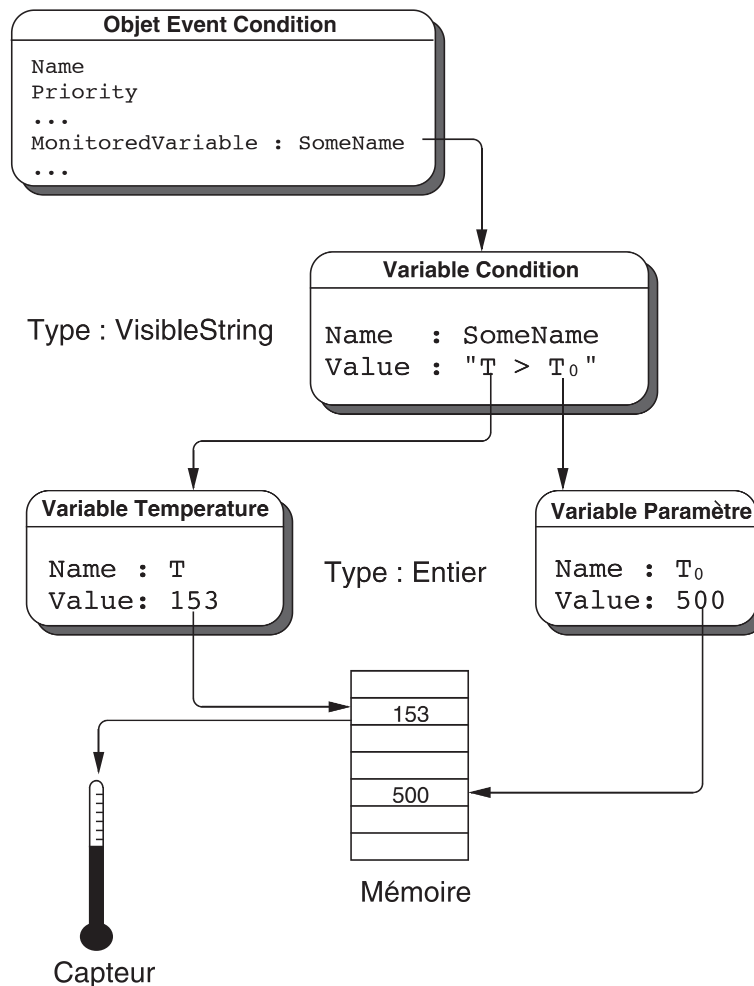


FIG. 5.2 - Configuration des objets dans XED

Les transitions classiques de MMS sont préservées. Aucune transition n'est enlevée ou ajoutée. La transition `ANY-TO-DELETED` qui survient généralement quand certains objets MMS sont effacés est étendue pour prendre en compte le cas de la destruction de variables identifiées dans l'expression prédicat de la Variable Condition.

La figure 5.4 résume les étapes permettant la définition d'un événement dans XED ainsi que la réception de la notification de cet événement.

5.2.3 XED vu depuis les serveurs

Nous allons maintenant décrire comment les serveurs MMS supportant XED manipulent l'expression prédicat contenue dans la Variable Condition. L'analyse grammaticale ainsi que l'évaluation de cette expression nécessite l'intégration d'un interpréteur d'expressions booléennes dans la VMD. Cet interpréteur (noté XEDI) est simple car il n'analyse que des expressions booléennes, une technique connue depuis des années dans le monde de la compilation et qui n'est par conséquent ni difficile à comprendre ni dure à implanter [ASU89]. XEDI supporte donc les mots clés classiques tels que `AND, OR, NOT` ainsi que les expressions arithmétiques et les opérateurs relationnels. La grammaire utilisée est décrite ici avec la notation classique BNF (ou Backus-Naur Form) suivant les conventions

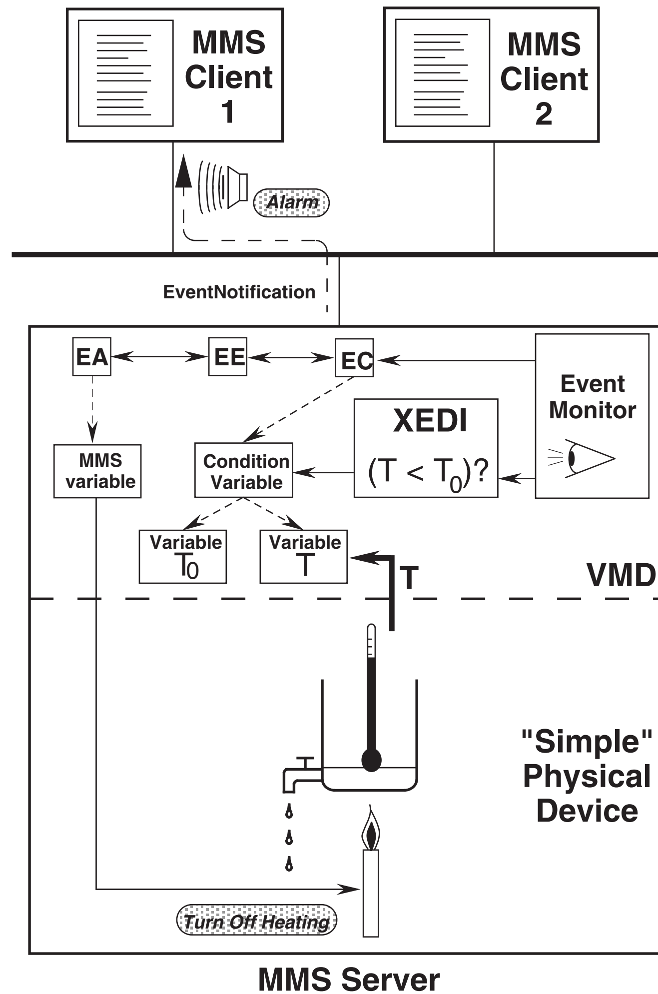


FIG. 5.3 - Un exemple d'utilisation de XED avec les événements MMS

suivantes :

- ::= sépare les deux membres d'une règle de production;
- [xx] dénote que xx peut ou non être présent;
- {xx} dénote que xx peut être répété autant de fois que nécessaire mais doit apparaître au moins une fois;
- | sépare les alternatives d'une production dans le membre de droite.

La grammaire est la suivante :

```
expression        ::= valeur | ( expression )
                      | operateur_unaire expression
                      | expression operateur_binaire expression

operateur_unaire  ::= + | - | NOT
```

Transition d'événement	MMS classique	MMS avec XED
DISABLED-TO-ACTIVE	Scrutation de l'EC activée et la variable scrutée est VRAIE	Scrutation de l'EC activée et l'évaluation de la Variable Condition donne VRAI
DISABLED-TO-IDLE	Scrutation de l'EC activée et la variable scrutée est FAUSSE	Scrutation de l'EC activée et l'évaluation de la Variable Condition donne FAUX
ACTIVE-TO-DISABLED	L'état de l'EC est ACTIVE et la scrutation de l'EC est désactivée	L'état de l'EC est ACTIVE et la scrutation de l'EC est désactivée
IDLE-TO-DISABLED	L'état de l'EC est IDLE et la scrutation de l'EC est désactivée	L'état de l'EC est IDLE et la scrutation de l'EC est désactivée
ACTIVE-TO-IDLE	L'état de l'EC est ACTIVE et la variable scrutée et FAUSSE	L'état de l'EC est ACTIVE et l'évaluation de la Variable Condition donne FAUX
IDLE-TO-ACTIVE	L'état de l'EC est IDLE et la variable scrutée et VRAIE	L'état de l'EC est IDLE et l'évaluation de la Variable Condition donne VRAI
ANY-TO-DELETED	L'EC est détruit et/ou la variable scrutée est effacée	L'EC est détruit et/ou la Variable Condition est effacée et/ou une variable de l'expression prédicat est détruite

TAB. 5.1 - Transitions d'événement dans MMS classique et dans XED

```
operateur_binaire ::= + | -  | *  | / | ^   | REM | MOD | & | < | <= | =
                        | /= | >= | > | AND | OR  | XOR

valeur            ::= integer | float | bitstring | boolean_array
                      | visiblestring | booleen   | MMSId

integer           ::= entier [E [+] entier]

float             ::= entier.entier [exposant]

bitstring         ::= '{[0 | 1]}'

boolean_array     ::= [{[t | T | f | F]}]

visiblestring     ::= "{caractere}"

booleen           ::= true | false

MMSId             ::= MMSId_simple | MMSId_compose

MMSId_simple      ::= mot

MMSId_compose     ::= mot.mot

mot               ::= lettre {[souligne] alphanumerique}

exposant          ::= E [+] entier | E [-] entier

entier            ::= chiffre {[souligne] chiffre}

caractere         ::= [char(1)-char(127)]
```

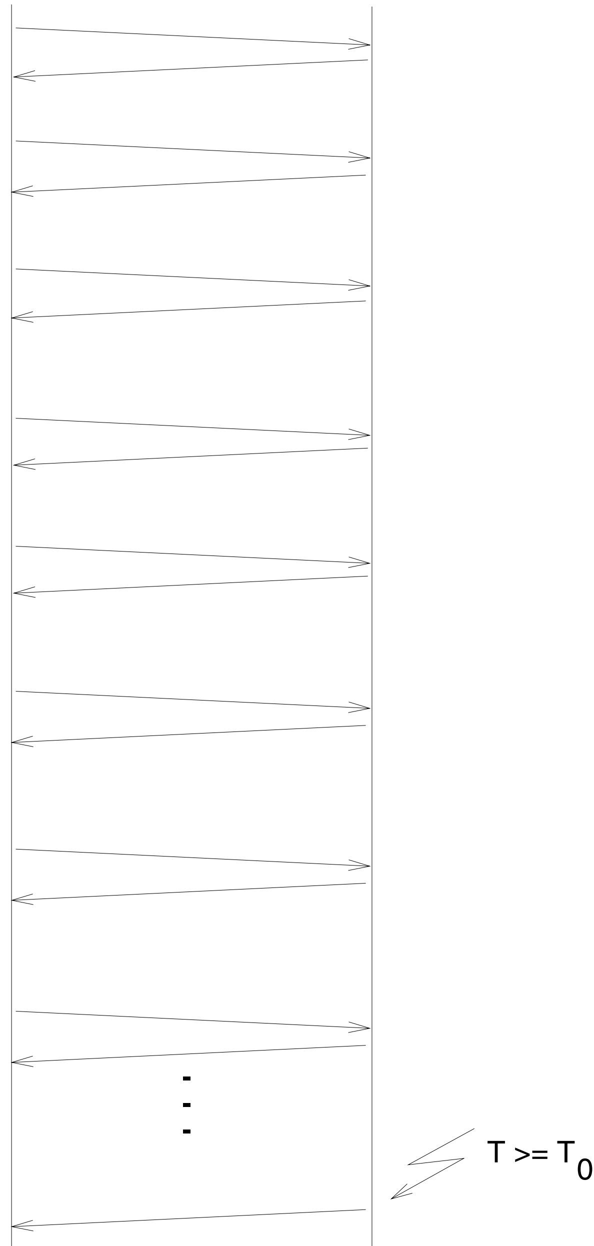


FIG. 5.4 - Séquence de services pour définir un événement avec XED

```
alphanumerique    ::= lettre | chiffre

lettre            ::= [A-Za-z]

chiffre           ::= [0-9]
```

Tous les identificateurs de cette grammaire (notés `MMSId`) font référence à des variables MMS locales au serveur où s'effectue la scrutation[2]. Les variables MMS anonymes (*Unnamed Variables*) ne sont pas

[2]Nous ne traitons pas des variables situées sur d'autres serveurs car ceci entraînent des problèmes de cohérence entre applications réparties que nous ne pouvons pas aborder ici.

supportées car leur représentation serait difficile et d'emploi inconfortable dans l'expression prédicat textuelle. Ceci n'introduit pas de limitation car il est toujours possible de définir une variable nommée basée sur la variable anonyme (service `Define Named Variable`) et d'utiliser cette nouvelle variable avec XED. Les variables référencées dans l'expression prédicat peuvent avoir n'importe laquelle des trois visibilités définies par MMS :

- `VMD-specific` : la variable est visible dans toute la VMD et accessible depuis n'importe quelle application client;
- `domain-specific` : la variable n'est visible que dans une région précise de la VMD. Toute application client peut néanmoins y accéder dans la mesure où elle connaît le nom du domaine en question. Le format utilisé dans l'expression prédicat est étendu pour inclure le nom du domaine de la façon suivante : `<nom-domaine>.<nom-variable>` (par exemple "`Capteur.TempCourante`");
- `aa-specific` : la variable n'est visible et accessible que par le client qui l'a créée. Le format de représentation de telles variables dans l'expression prédicat est le même que pour les variables de visibilité `VMD-specific`. XEDI recherche en premier les variables `VMD-specific` car celles de visibilité `aa-specific` sont d'intérêt plus restreint et d'usage peu probable avec XED.

Nous avons déjà vu précédemment que la scrutation des événements est effectuée par le moniteur d'événements (EM). Avec XED, cette scrutation se fait de façon identique avec pour seule différence qu'au lieu d'aller lire directement la valeur de la variable scrutée, l'EM appelle XEDI qui lui retourne le résultat de l'évaluation de l'expression prédicat (fig. 5.3). Pour rester compatible avec la méthode de détection classique des événements XEDI peut évidemment retourner la valeur boolénne de la variable scrutée. L'EM n'a alors pas besoin de différencier la variable scrutée de type `Boolean` ou la Variable Condition de type `VisibleString`. Dans les deux cas, le résultat retourné par XEDI est un booléen attendu par l'EM. La suite du traitement des événements et des transitions se poursuit exactement comme dans MMS classique.

5.2.4 Les problèmes introduits par XED

5.2.4.1 L'interprétation des expressions

XED introduit dans MMS des problèmes spécifiques au domaine de la compilation. Les plus immédiats sont ceux de vérification de syntaxe et de cohérence des types. La détection d'événements à partir d'expressions prédicat syntaxiquement incorrectes ne peut évidemment se faire. Les expressions prédicat des Variables Condition doivent suivre la grammaire définie précédemment. De même, une condition "$T > 500$" n'a pas de sens si T n'est pas une variable de type entier ou réel. On inclut également dans les problèmes de compilation le cas où l'expression prédicat contient des références à des variables qui ne sont pas définies correctement ou qui sont inexistantes.

Il est important que ces erreurs soient détectées avant le début de la scrutation de l'EC considéré si l'on ne veut pas donner lieu à des transitions d'événements imprévues comme nous le verrons plus loin. Les services `Define Event Condition` et `Alter Event Condition Monitoring` doivent donc être étendus pour permettre ces vérifications. Ces services doivent appeler XEDI pour effectuer :

- une vérification de syntaxe de l'expression prédicat;
- une vérification de la cohérence des types dans l'expression, basée notamment sur les opérateurs et les relations utilisés;
- un test d'existence des variables MMS identifiées dans l'expression prédicat.

En cas d'échec d'au moins une de ces vérifications, l'EC ne doit pas être créé ou la scrutation ne doit pas être enclenchée selon que le service considéré soit `Define Event Condition` ou `Alter Event Condition Monitoring` respectivement.

5.2.4.2 Les problèmes liés au temps

Le temps soulève deux problèmes par rapport à XED. Tout d'abord, il est clair qu'évaluer l'expression prédicat demande plus de calculs que pour les événements classiques uniquement basés sur un booléen. En particulier, XED nécessite des capacités de calcul arithmétique. Ceci implique que la détection étendue des événements prend plus de temps que la détection classique. Plus l'expression à évaluer est complexe et plus il y a de variables MMS référencées dans cette expression, plus le temps mis pour l'interpréter est long. De plus s'il était possible de fonctionner en interruption pour signaler le changement de la variable scrutée dans le cas MMS classique, ceci n'est plus possible avec XED qui nécessite une scrutation périodique. XED tend donc à ralentir la détection des événements, un des rares aspects de MMS où des contraintes de temps peuvent explicitement être fournies par les utilisateurs et doivent alors être respectées par les serveurs. Le chapitre 6 étudie une solution permettant la détection en temps réel des événements MMS.

XED demande donc de la discipline de la part des programmeurs d'applications MMS pour éviter les expressions prédicat trop longues et trop complexes. Cet état de fait n'est pas nouveau dans MMS et s'inscrit tout à fait dans la philosophie de la norme. En effet, [ISO90a] ne limite pas le nombre de EEs qui peuvent être liés à un EC par exemple. Il est entendu cependant que plus ce nombre est élevé, plus les clients considérés mettent de temps pour recevoir les notifications correspondantes. Le problème est du même ordre avec XED et les utilisateurs doivent en être conscients.

Le second problème lié au temps tient à la cohérence de la détection. L'évaluation d'une expression prédicat n'est pas instantanée. Il existe un certain temps pendant lequel un prédicat est vrai (ou faux) puis change de valeur. Ce temps est appelé *temps de maintien du prédicat* ("predicat holding time") dans [LSM90] où des prédicats semblables à XED sont définis pour le déverminage d'applications réparties. Des expressions très complexes ou longues augmentent les chances pour que le résultat d'une partie de l'expression déjà évaluée soit invalide à la fin de l'évaluation totale c'est-à-dire que le temps de maintien du prédicat soit dépassé. Par exemple, si l'expression est "(`PositionRobot` $\geq$ `10`) `and LongPrédicat`", il se peut que `PositionRobot` $<$ `10` après avoir évalué `LongPrédicat` alors que `PositionRobot` $\geq$ `10` était vraie avant. Une telle situation non désirée risque alors de provoquer des événements qui ne devraient pas apparaître ou au contraire risque de ne pas détecter des événements effectifs. De plus, si une action est liée à un événement, la pertinence de son exécution peut dépendre de la valeur du prédicat. Lumpp *et al.* [LSM90] notent que le temps de maintien du prédicat devrait être plus grand que celui d'exécution de l'action. Par ailleurs, la capacité d'une action intrusive[3] à produire des résultats valables dépend également de la complexité et de la longueur de l'action à exécuter [MLCS]. Ce dernier problème, qui apparaît aussi dans l'utilisation classique des événements MMS, n'est absolument pas considéré dans la norme MMS.

Encore une fois cette situation vient de la liberté laissée aux programmeurs d'applications client qui n'ont pas de contraintes quant à l'expression prédicat qu'ils veulent définir. De même la liberté d'imposer des contraintes plus strictes sur le comportement des serveurs MMS est laissée aux concepteurs d'applications serveur. En ce sens, XED respecte l'esprit de la norme MMS.

Mais de façon générale, la probabilité de rencontrer ce genre de problème repose fortement sur les caractéristiques propres à chaque implantation. Non seulement elle dépend des algorithmes d'ordonnancement et d'évaluation des conditions événementielles mais également du temps nécessaire pour

[3] Action ayant un effet sur l'équipement modélisé par la VMD.

interroger le dispositif physique modélisé par la VMD dans le cas où par exemple la consultation de certaines variables liées à des capteurs est nécessaire. En effet, le lien entre un dispositif industriel et d'une VMD étant purement local c'est-à-dire non traité par MMS, on peut très bien imaginer que la lecture d'une variable MMS implique plus que la simple consultation d'une zone mémoire et demande plusieurs dizaines ou centaines de millisecondes pour s'effectuer.

5.2.4.3 La reconfiguration

On appelle ici *reconfiguration* le fait de changer la condition de détection d'un événement dans XED. Ceci inclut la modification de la valeur de la Variable Condition au moyen du service MMS `Write` mais peut également être effectuée d'une façon plus sûre comme nous allons le voir. La reconfiguration est une méthode très souple car elle autorise le changement en ligne et à tout moment de la condition de détection d'un événement.

Les désavantages de la reconfiguration apparaissent quand on cherche à modifier directement la valeur de la Variable Condition. Ceci n'est pas conseillé pour les raisons suivantes :

- De nouvelles variables peuvent être référencées dans la condition modifiée. Il est alors possible que ces variables n'existent pas suite à une erreur de la part du client par exemple. Ceci provoque une transition d'événement `ANY-TO-DELETED` qui n'est pas ce à quoi l'utilisateur s'attendait. De plus, la scrutation de l'événement MMS est désactivée mais l'événement physique réel peut lui toujours se produire.
- Les types de l'expression prédicat modifiée peuvent être incohérents ou la syntaxe peut être incorrecte. Dans ce cas, XEDI n'est pas capable d'évaluer l'expression et désactive automatiquement la scrutation de l'EC correspondant. Ceci conduit à une transition `ACTIVE-TO-DISABLED` ou `IDLE-TO-DISABLED` ce qui encore une fois n'est sans doute pas le comportement attendu par l'utilisateur.

La table 5.2 montre quelles transitions peuvent être déclenchées à cause d'erreurs spécifiques à la reconfiguration dans XED. Une erreur de type ou de syntaxe ne permet pas à XEDI de mener à bien l'évaluation de l'expression prédicat. Nous choisissons alors de désactiver la scrutation de l'EC correspondant ce qui affecte son état à `DISABLED`. Par contre, quand XEDI ne peut trouver une des variables référencées dans l'expression prédicat, il en déduit que cette variable a été détruite. La scrutation de l'EC est là encore désactivée. L'état final de l'EC est donc le même que dans le cas d'une erreur de syntaxe ou de type mais la transition produite est `ANY-TO-DELETED`. Cette transition traduit en effet que l'un des objets MMS nécessaires au processus de scrutation a été détruit.

Ce genre d'erreur montre qu'il est toujours préférable et plus sûr d'effectuer une désactivation de la scrutation d'un EC avant de le reconfigurer. Pour ce faire, le service `Alter Event Condition Monitoring` devrait donc être utilisé avant tout service `Write` modifiant la valeur de la Variable Condition. Cependant, cette dernière étant une variable MMS normale il n'y a aucune raison pour forcer ce comportement. XED se veut aussi proche que possible de MMS et en ce sens respecte l'esprit de liberté laissé aux concepteurs d'applications MMS.

Si l'on désire changer complètement l'expression prédicat décidant du déclenchement d'un événement, il est clair que la modification de la valeur de la Variable Condition est nécessaire. Mais une reconfiguration plus sûre peut aussi être effectuée en ne modifiant seulement que les Variables Paramètre référencées dans l'expression prédicat. Ceci a pour avantage d'éviter les transitions imprévues décrites ci-dessus. Reprenons par exemple, l'expression "$T \geq T_0$" avec T et T_0 deux variables MMS de type réel. T est lié à un capteur de température et change dynamiquement. T_0 est une Variable Paramètre ayant pour valeur 500. Si l'application client décide de changer cette valeur de seuil à 300,

Transition d'événement	Causes du déclenchement
DISABLED-TO-ACTIVE	Ne se produit jamais à cause d'une erreur de configuration
DISABLED-TO-IDLE	Ne se produit jamais à cause d'une erreur de configuration
ACTIVE-TO-DISABLED	L'état de l'EC est ACTIVE et la Variable Condition est reconfigurée avec une erreur de type ou de syntaxe
IDLE-TO-DISABLED	L'état de l'EC est IDLE et la Variable Condition est reconfigurée avec une erreur de type ou de syntaxe
ACTIVE-TO-IDLE	Ne se produit jamais à cause d'une erreur de configuration
IDLE-TO-ACTIVE	Ne se produit jamais à cause d'une erreur de configuration
ANY-TO-DELETED	La Variable Condition est reconfigurée avec de nouveaux identificateurs de variables qui font référence à des variables MMS inexistantes

TAB. 5.2 - Transitions dues à un mauvais usage de XED

il suffit d'écrire cette nouvelle valeur dans T_0 au moyen du service MMS `Write`. Qu'il y ait ou non une erreur dans cette requête d'écriture, l'expression prédicat est inchangée et il ne peut donc y avoir aucune erreur de type ou de syntaxe. De même, puisqu'aucune nouvelle variable n'a été ajoutée il ne peut y avoir de référence a des variables inexistantes qui causeraient une transition `ANY-TO-DELETED`. T et T_0 sont les seules variables référencées et toujours existantes.

5.2.5 Impact de XED sur la norme MMS

5.2.5.1 Extensions des services MMS

XED fait partie des extensions de niveau de modification 2. Ceci signifie que la compatibilité ainsi que la portabilité des applications client MMS classiques sont garanties car aucun changement n'est fait au protocole. Les serveurs MMS supportant XED peuvent être utilisés en lieu et place des serveurs classiques sans qu'aucune modification des applications client ne soit nécessaire.

Nous allons examiner plus en détail les cinq services concernés par la scrutation des événements et présenter les extensions éventuelles qu'il faut leur apporter. Aucune de ces extensions n'implique de modification du protocole MMS [ISO90b] essentiellement parce que :

- les services existants sont ré-utilisés, aucun nouveau service n'est ajouté;
- les paramètres des services existants sont également ré-utilisés;
- les codes d'erreurs existants s'appliquent toujours à notre extension.

Cinq services sont susceptibles d'être modifiés par XED :

`Define Event Condition`: La norme [ISO90a] est peu précise quant au type de la variable scrutée référencée dans la requête de définition de l'EC (paramètre `MonitoredVariable`). S'il est précisé que cette variable doit être de type `Boolean`, il n'est dit ni qu'un autre type constitue une erreur, ni qu'une procédure de vérification de type doit être entamée lors de l'exécution de ce service, ni le type d'erreur qui doit être retournée au client s'il y a lieu. La solution la plus plausible est qu'un type non booléen invalide la création de l'EC et que l'erreur MMS `type-inconsistent` est renvoyée au client. Mais cette imprécision nous conforte dans l'idée qu'étendre ce service

pour accepter des variables de type `VisibleString` ne constitue pas une entorse à la norme. Ce sera donc la première extension de ce service. La seconde extension implique la vérification de syntaxe, de cohérence des types et d'existence des variables de l'expression prédicat. XEDI doit donc être appelé pour effectuer ces vérifications. Si une erreur se produit lors de ces vérifications, le service est interrompu. L'EC n'est pas défini et une erreur est retournée au client. Aucune autre action différente de celles exécutées par le standard ne doit être entreprise.

`Alter Event Condition Monitoring`: Ce service doit être étendu pour effectuer les mêmes vérifications sur la syntaxe, les types et l'existence des variables que pour le service `Define Event Condition` mais seulement lorsque la scrutation de l'EC est demandée, c'est-à-dire lorsque le paramètre booléen `Enabled` est vrai. Si une erreur surgit durant ces vérifications, non seulement la scrutation de l'EC n'est pas autorisé mais toute autre modification des attributs de l'EC spécifiées dans ce service est invalidée.

`Get Event Condition Attributes`: Ce service se comporte exactement comme dans MMS classique. Le paramètre de retour `MonitoredVariable` qui représente le nom de la variable scrutée s'applique toujours pour la Variable Condition de XED.

`Delete Event Condition`: Aucun changement par rapport à MMS classique. Avec XED aussi, ce service peut causer une transition `ANY-TO-DELETED` si l'EC est détruit durant sa scrutation.

`Report Event Condition Status`: Aucun changement par rapport à MMS classique.

5.2.5.2 Gestion des erreurs dans XED

Toute réponse d'un service confirmé MMS peut être le constat d'une erreur d'exécution dans le serveur. Dans ce cas cette réponse négative contient quatre arguments :

- `ErrorClass`: de type entier, ce code obligatoirement présent identifie le domaine général de l'erreur. Il contient un sous-paramètre `ErrorCode`. Ce dernier est de type entier et identifie plus précisément l'erreur dans la classe considérée.
- `AdditionalCode`: de type entier, la signification de ce code optionnel est laissée libre à chaque application.
- `AdditionalDescription`: de type `VisibleString`, ce paramètre permet de spécifier de façon lisible des informations sur l'erreur. Son contenu est laissé libre à chaque application.
- `ServiceSpecificInformation`: permet de préciser l'erreur. Il est optionnel et utilisé pour certains services MMS seulement.

La table 5.3 illustre les codes d'erreur `ErrorCode` ré-utilisés dans XED et la raison pour laquelle l'erreur est apparue. Tous font partie de la classe d'erreur MMS `definition` qui dénote un problème dans la façon dont est défini un objet.

Cependant, il n'est pas toujours suffisant de fournir ces codes d'erreurs simples. L'utilisateur est en droit d'attendre plus de précision quant à l'erreur qui a provoqué l'échec de la définition d'un EC par exemple. Ceci est particulièrement vrai quand il s'agit d'une erreur d'interprétation de l'expression prédicat qui n'est pas toujours évidente à repérer. Une explication plus détaillée ainsi que la position de l'erreur dans l'expression prédicat fournirait une aide appréciable.

Pour ce faire, il est encore une fois possible d'exploiter MMS sans modifier la partie du protocole définissant les erreurs et sans rajouter de nouveau codes d'erreur. Dans XED, nous avons choisi d'utiliser `AdditionalDescription` pour fournir sous forme claire et lisible à l'utilisateur une description de

Code d'erreur MMS	Raison de l'erreur
`type-inconsistent`	Le type de la Variable Condition n'est ni `Boolean` ni `VisibleString`
`object-undefined`	Au moins une des variables référencées dans l'expression prédicat n'existe pas
`type-inconsistent`	Certains types dans l'expression prédicat sont incohérents
`object-attribute-inconsistent`	Erreur de syntaxe dans l'expression prédicat

TAB. 5.3 - Réutilisation des codes d'erreur MMS dans XED

l'erreur qui s'est produite. Le code `AdditionalCode` contient quant à lui la position de l'erreur dans l'expression prédicat c'est-à-dire qu'il pointe sur le caractère couramment analysé par XEDI lorsque l'erreur s'est produite.

L'exploitation que nous faisons du format d'erreur défini par MMS est donc largement suffisante pour retourner un diagnostic complet des erreurs rencontrées par XEDI de la même façon que le ferait un compilateur.

Enfin, comme pour l'extension XES définie à la section 5.3 l'attribut `List Of Capabilities` des VMDs supportant XED doit contenir un élément indiquant la capacité à détecter les événements de façon étendue.

5.2.6 Intégration de nouveaux types d'événement

5.2.6.1 Séquences d'événements

Il est possible de définir un événement comme étant une séquence d'événements en utilisant les modificateurs MMS. Cette façon de faire a été illustrée à la section 3.11.1. Il faut noter toutefois que la définition d'un événement avec XED au moyen de l'opérateur "`AND`" ne correspond pas tout à fait à celle qui est faite en utilisant les modificateurs.

Par exemple, la condition événementielle XED "$(A > 3)$ `AND` $(B < 2)$" est **un seul** événement MMS. L'utilisation de modificateurs ferait apparaître **deux** événements MMS : "$(A > 3)$" et "$(B < 2)$". Par ailleurs, les modificateurs imposent un ordre sur la réalisation de ces conditions. Dans notre cas, on ne vérifie que $B < 2$ qu'après avoir obtenu $A > 3$. Avec le mot clé `AND` de XED, cet ordre n'a pas d'importance. On doit donc rajouter l'instruction "`AND THEN`" dans XED pour tenir compte de l'ordre de réalisation des conditions. Mais ceci impose d'une part à XEDI de disposer d'une mémoire pour conserver le fait que $A > 3$ a déjà été réalisée. D'autre part, il faut modifier XEDI de façon que l'évaluation de la partie gauche d'un "`AND THEN`" réalisée ne soit pas réévaluée mais uniquement la partie droite.

5.2.6.2 Variations de la valeur d'une variable

MMS ne permet pas de déclencher un événement quand la valeur d'une variable (autre que la variable scrutée) est modifiée à la manière de l'instruction `watch` du langage ORE [DJJ89]. Par exemple, l'expression "`watch(x) do S`" permet de lancer l'action `S` lorsque la variable `x` change de valeur. Dans [Ple90], Pleinevaux a déjà proposé d'étendre la définition des événements MMS avec les instruc-

tions `x CHANGES`, `x INCREASES` et `x DECREASES` qui définissent un événements respectivement lorsque la valeur de la variable `x` change, augmente ou diminue. Nous reprenons ces instructions pour les adapter à XED.

Il convient de définir plus précisément quand le prédicat `x CHANGES` est vrai et quand il prend la valeur FAUX. Sinon il faudrait ajouter une notion de temps pour savoir depuis quand la variable `x` est à évaluer (c'est-à-dire une instruction du type "`x CHANGED since 13:54`"). En effet, supposons que l'expression prédicat "`x CHANGES AND (Temperature < 40)`" soit évaluée. Lors d'une première évaluation on a bien "`x CHANGES`" qui est vraie mais pas "`(Temperature < 40)`". Lors d'une seconde évaluation de cette expression, on a "`(Temperature < 40)`". Doit-on alors considérer toute l'expression comme vraie parce que `x` a changée lors de l'évaluation précédente? Nous considérons que non et conservons la notion MMS d'évaluation périodique des événements scrutés. Nous définissons ainsi implicitement le résultat des trois instructions `CHANGES`, `INCREASES` et `DECREASES` par rapport à leur précédente évaluation :

- `x CHANGES` est vraie si la variable MMS `x` a une valeur différente de celle obtenue lors de la précédente évaluation de l'expression prédicat contenant cette instruction. Le type de la variable `x` peut être quelconque.
- `x INCREASES` est vraie si la variable MMS `x` a une valeur supérieure de celle obtenue lors de la précédente évaluation de l'expression prédicat contenant cette instruction. Le type de la variable `x` peut être entier ou réel.
- `x DECREASES` est vraie si la variable MMS `x` a une valeur inférieure de celle obtenue lors de la précédente évaluation de l'expression prédicat contenant cette instruction. Le type de la variable `x` peut être entier ou réel.

Pour la première évaluation, la comparaison se fait par rapport à la valeur de la variable `x` lorsque la scrutation de la condition événementielle associée est activée c'est-à-dire lors du service `Alter Event Condition Monitoring`.

Comme pour les séquences d'événements une modification de XEDI s'avère ici nécessaire. Il faut en effet pouvoir conserver la valeur précédente de la variable `x` pour effectuer la comparaison. On munit donc XEDI d'une table qui mémorise les valeurs des variables à scruter entre deux évaluations. Chaque entrée de cette table pointe vers une copie d'une variable apparaissant dans une instruction `CHANGES`, `INCREASES` ou `DECREASES`. Cette copie est mise à jour à chaque évaluation de l'expression prédicat contenant cette instruction.

5.2.6.3 Variations de la pente lors du changement de valeur d'une variable

Dans certains cas, il peut être intéressant de détecter la rapidité avec laquelle la valeur d'une variable change. En effet, pour certaines applications une variation brutale peut constituer un événement significatif alors qu'une variation lente ne l'est pas. Par exemple le processus d'échauffement d'une résine époxy dans un procédé pour les pièces en matériau composite ne doit pas être trop rapide pour éviter un début de polymérisation. Si tel est le cas, il est nécessaire de lever une alarme. Un échauffement lent n'est par contre pas significatif [Che96].

Nous proposons donc d'étendre les instructions `x CHANGES`, `x INCREASES` et `x DECREASES` par `x CHANGES MORE THAN y`, `x INCREASES MORE THAN y` et `x DECREASES MORE THAN y` respectivement. Le changement de valeur est relatif à la dernière scrutation de la variable `x`. Le client doit connaître la période d'évaluation de l'EC et donc de la variable `x`. Rappelons qu'il peut la fournir et la changer à tout moment au moyen du paramètre `Evaluation Interval` du service `Alter Event Condition`

`Monitoring`. Il peut alors fixer la valeur de seuil `y` au delà de laquelle la variation de la variable est considérée comme trop brutale. Plus précisément :

- `x CHANGES MORE THAN y` est vraie si la variable MMS `x` a une valeur différente de celle obtenue lors de la précédente évaluation de l'expression prédicat contenant cette instruction, que cette différence est positive et $y > 0$ (ou que cette différence est négative et $y < 0$) et que la valeur absolue de cette différence est supérieure à $|y|$. Le type de la variable `x` et du seuil `y` est entier ou réel.
- `x INCREASES MORE THAN y` est vraie si la variable MMS `x` a une valeur supérieure de celle obtenue lors de la précédente évaluation de l'expression prédicat contenant cette instruction et que cette différence est positive (avec $y > 0$) et supérieure à `y`. Le type de la variable `x` et du seuil `y` est entier ou réel.
- `x DECREASES MORE THAN y` est vraie si la variable MMS `x` a une valeur inférieure de celle obtenue lors de la précédente évaluation de l'expression prédicat contenant cette instruction, que cette différence est négative (avec $y > 0$) et que sa valeur absolue est supérieure à `y`. Le type de la variable `x` et du seuil `y` est entier ou réel.

On peut imaginer une extension identique avec détection d'un événement lorsqu'au contraire la variation de la valeur d'une variable est trop lente.

5.2.6.4 Composition d'événements XED

Il peut être intéressant de former un événement XED à partir d'autres événements. Supposons par exemple qu'il existe deux variables conditions nommées V_1 et V_2 attachées aux ECs EC_1 et EC_2 et ayant respectivement pour expression "`Temperature > 40`" et "`Pression < 10`". On veut définir un événement qui corresponde à la réalisation de ces deux prédicats. Cet événement est représenté par l'EC EC_3 lié à la variable V_3. Plutôt que d'écrire l'expression "`(Temperature > 40) AND (Pression < 10)`" dans V_3 nous proposons de rajouter à XEDI la capacité d'analyser les expressions "V_1 `AND` V_2" et "EC_1 `AND` EC_2" de sorte que l'on puisse écrire dans V_3 une de ces deux dernières expressions.

Pour ce faire on doit étendre XEDI de la façon suivante :

- quand le nom d'une variable V de type `VisibleString` apparaît dans une expression prédicat et précède ou suit un des mots clés `AND`, `OR` ou `XOR`, ou précède le mot clé `NOT`, XEDI remplace l'occurrence de cette variable par sa valeur. Le fait que V ne soit pas de type booléen ne constitue donc pas une erreur. Ce processus est itératif de sorte que si la valeur de V contient elle-même le nom d'une variable V' de type `VisibleString` utilisé avec les opérateurs logiques cités ci-dessus, alors XEDI remplace l'occurrence de V' par sa valeur;
- quand le nom d'un EC apparaît dans une expression prédicat précédent ou suivant un des mots clés `AND`, `OR` ou `XOR`, ou précédent le mot clé `NOT`, XEDI remplace l'occurrence de cet EC par :
  - la valeur booléenne de la variable scrutée s'il s'agit d'un EC classique donc lié à une variable de type booléen;
  - la valeur de la Variable Condition s'il s'agit d'un EC XED donc lié à une variable de type `VisibleString`.

  Le nom d'un tel EC ne doit pas aussi être le nom d'une variable MMS de type `VisibleString` sinon XEDI évalue la variable.

Il faut noter qu'il existe une ambiguïté concernant l'égalité de deux variables de type `VisibleString`. Pour l'illustrer, reprenons l'exemple précédent. Si l'on avait l'expression "V_1 = V_2" devrait-on l'interpréter comme une égalité entre la valeur de variables de type `VisibleString` (donc en effectuant une comparaison caractère par caractère) ou comme une égalité entre les booléens `(Temperature > 40)` et `(Pression < 10)` ?

Devant cette ambiguïté, nous avons décidé de ne pas inclure l'égalité dans la composition d'événements XED pour permettre la comparaison de valeurs de type `VisibleString`. Toutefois si le nom d'une variable de type `VisibleString` ou d'EC XED apparaît seul dans une expression prédicat alors on effectue l'égalité avec la valeur VRAI. Par exemple si l'expression prédicat d'un EC donné est simplement "V_1" alors son évaluation est le résultat du booléen `(Temperature > 40) = VRAI`.

Nous résumons l'emploi de la composition d'événements XED sur la table 5.4 en prenant pour exemple les valeurs suivantes :

- `Temperature = 50`
- `Pression = 10`
- `Tension = 1`
- V_1 `= (Temperature > 40)`
- V_2 `= (Pression < 10)`
- V_3 `= (`V_1 `AND Pression < 5)`

Expression prédicat	**Expression équivalente**	**Evaluation de l'expression**
V_1	`Temperature > 40`	VRAI
`NOT` V_2	`NOT Pression < 5`	VRAI
V_1 `OR` V_2	`Temperature > 40 OR Pression < 10`	VRAI
EC_1 `OR` EC_2	`Temperature > 40 OR Pression < 10`	VRAI
V_1 `OR` EC_2	`Temperature > 40 OR Pression < 10`	VRAI
V_1 `=` V_2	"`Temperature > 40`" = "`Pression < 10`"	FAUX
EC_1 `=` EC_2	"`Temperature > 40`" = "`Pression < 10`"	FAUX
V_1 `=` V_1	"`Temperature > 40`" = "`Temperature > 40`"	VRAI
V_3 `OR Tension < 3`	`(Temperature > 40 AND Pression < 5) OR Tension < 3`	VRAI

TAB. 5.4 - Exemple de composition d'événements XED

5.2.7 Implantation de XED

XED a été intégré au sein de notre serveur MMS en Ada. Les tests effectués ont montré que XED reste effectivement complètement compatible avec le protocole MMS [ISO90b] ainsi qu'avec le comportement des serveurs décrits dans la norme [ISO90a].

XED nécessite l'addition de 1785 lignes de code dans le serveur MMS. Près de la moitié ne représente que des librairies permettant de définir des opérations arithmétiques et de comparaison entre types Ada. La logique même de XEDI est regroupée dans 939 lignes de codes (53%) qui sont générées par les outils Ayacc [TTSC94] et Aflex [Sel90] sur des fichiers d'entrée totalisant 384 lignes de code. Sans

compter les librairies d'opérations sur les types Ada, XEDI représente une proportion de 26% sur la gestion des événements MMS et 5.6% sur le code total de notre serveur.

Nous avons mesuré les temps d'évaluation de certaines expressions prédicat. Les mesures effectuées confirment que XED ralentit la détection des événements MMS par rapport à MMS classique.

Dans notre implantation ce ralentissement est même considérable puisque l'évaluation d'une expression prédicat peut prendre en tout jusqu'à 30 ms alors que celle d'un EC classique dépasse rarement 2 ms. En fait, il semble que le code généré par Ayacc soit loin d'être optimal en ce qui concerne les temps d'exécution. Il est en effet apparu que XEDI perd plus de 10 ms en début et en fin d'évaluation d'une expression. Nous n'avons pas cherché à analyser en détail le code généré par Ayacc. Toutefois il est fort probable qu'un analyseur syntaxique plus efficace peut être implanté.

Mis à part les premiers et derniers lexèmes, la plus grande partie du temps d'évaluation de l'expression prédicat se passe dans la lecture des variables MMS qui y sont référencées. Il faut en moyenne 3 ms pour lire une variable MMS depuis XEDI. Le temps d'évaluation d'une expression varie linéairement en fonction du nombre de variables contenues dans cette expression (figure 5.5). Nous résumons sur le tableau 5.5 les temps d'évaluation de différents lexèmes.

Lexème	**Temps d'évaluation**
Premier et dernier	Plus de 10 ms
Variable référencée	3 ms
Autres lexèmes	moins de 500 μs

TAB. 5.5 - Temps d'évaluation des différents lexèmes

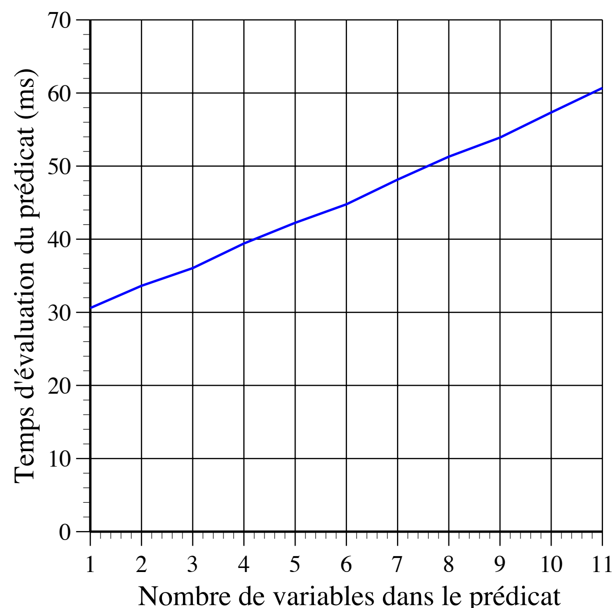


FIG. 5.5 - Variation du temps d'évaluation d'une expression selon le nombre de variable qu'elle contient

Nous revenons sur les performances de XED à la section 7.2.3 où nous proposons trois solutions MMS au problème des philosophes dont une basée sur XED.

5.2.8 Conclusion sur XED

Une version antécédente de MMS avait déjà intégré une proposition similaire à XED bien que présentée sous un angle très différent [ISO86]. Cette proposition a été discutée par Elloy dans [EC86, Ell87] et a fait l'objet d'une analyse pour les communications temps-réel dans [Ple90]. Elle n'est plus adaptée ici à cause d'une définition des événements MMS qui maintenant diffère complètement de ce qu'elle était alors. La complexité de la solution qui était proposée est sans doute la raison pour laquelle celle-ci n'a pas été retenue dans la norme MMS finale. L'intégrer maintenant telle quelle dans MMS impliquerait de sérieuses modifications du protocole et rendrait incompatibles ces nouvelles applications avec la norme actuelle. XED est simplifié par rapport à cette proposition et s'intègre naturellement dans le contexte d'un protocole maintenant normalisé, bien défini et stable.

Il faut noter l'importance que nous avons mis à réduire au maximum l'impact de XED sur la norme MMS. Non seulement XED ne modifie pas le protocole ou le comportement des serveurs mais en plus les actions qui doivent être effectuées par les clients restent extrêmement proches d'une utilisation classique des événements MMS. Du point de vue implantation des serveurs, les modifications à apporter restent simples et très limitées puisqu'en gros seul l'ajout de l'objet interpréteur XEDI est nécessaire. Cet objet lui-même se conçoit facilement avec l'aide d'outils tels que les très connus *lex* et *yacc* [LMB92].

L'utilisation de XED est particulièrement intéressante combinée avec les modificateurs MMS pour conditionner l'exécution d'un service. Ainsi, on peut par exemple effectuer une requête de service MMS en précisant que ce service ne doit être exécuté que lorsqu'un chariot arrive en fin de course et qu'un bouton donné est poussé.

Nous avons montré qu'il est possible d'étendre à peu de frais la détection des événements MMS pour mieux satisfaire les besoins des utilisateurs. Avec XED, la gestion des événements MMS est améliorée de façon significative. Nous résumons ici les avantages de XED sur la détection classique des événements MMS :

- adaptation aux besoins des clients, personnalisation et souplesse d'utilisation;
- effort d'implantation très réduit;
- compatibilité totale des serveurs avec les applications client MMS existantes.

Nous notons toutefois que XED peut ralentir considérablement la détection d'événements MMS. Une implantation efficace de XEDI est ici essentielle. Dans cette optique, on pourrait par exemple étudier une solution où l'expression prédicat est pré-compilée plutôt qu'interprétée.

5.3 Des priorités pour les services MMS

5.3.1 Introduction : le besoin d'avoir des priorités

Un serveur MMS peut potentiellement être connecté à de nombreux clients. Chaque client peut effectuer un nombre quelconque de requêtes et ce à n'importe quel moment. On peut donc s'attendre dans certaines circonstances à ce qu'un serveur MMS soit surchargé avec de nombreuses requêtes de service ou tout au moins qu'il en ait plus d'une prête à être exécutée à un instant donné. La norme MMS ne précise pas le comportement du serveur dans un tel cas et n'impose aucun ordre dans l'exécution des services MMS reçus. En fait, **la seule exigence** est que les objets transaction correspondant aux services reçus soient initialisés dans l'ordre de réception de ces services, c'est-à-dire dans notre cas dans l'ordre donné par `QueueReceptions`. La raison de cet impératif n'est pas donnée par la norme. Toutefois il semble qu'une raison suffisante vienne de la façon dont les annulations de services sont

traitées. En effet, le service `Cancel` annule une requête de service précédemment faite par le même client en détruisant l'objet transaction correspondant à cette requête. Si l'ordre d'initialisation des objets transaction ne respecte pas l'ordre d'arrivée des services il se pourrait que le `Cancel` ne trouve aucun service à annuler alors qu'il a bien été émis par le client après le service à interrompre. Mais si l'ordre d'initialisation des objets transaction est fixé, l'ordre d'exécution est lui laissé libre à chaque implantation d'un serveur.

Ceci peut paraître étrange mais s'inscrit en fait dans la philosophie de MMS qui est de laisser une grande liberté au niveau de l'implantation des applications MMS. Par ailleurs, des situations d'interblocage peuvent être évitées lorsque les exécutions de certaines requêtes de service sont inversées par rapport à leur ordre d'arrivée. Quelques soient les raisons qui ont conduit les concepteurs de MMS à laisser cette liberté, nous pouvons l'exploiter dans le but de contrôler l'ordre d'exécution des services MMS en favorisant le traitement d'un service par rapport à un autre s'ils sont tous deux prêts à être exécutés. Ceci est le propos de l'extension XES (eXtended Execution of Services) que nous allons exposer dans cette section. Nous proposons également une étude de cette extension dans [Cas95d].

Dans la plupart des cas, les serveurs exécutent les services un par un séquentiellement dans l'ordre de réception de ceux-ci quel que soit leur caractère critique. Les services qui devraient pourtant recevoir une attention immédiate ne sont alors exécutés que lorsque leur tour arrive. Pour éviter cela, il est nécessaire d'affecter des priorités aux requête de service MMS :

1. Une première solution est d'affecter statiquement une priorité à chaque classe de services. Par exemple, les services de gestion des événements sont plus critiques que ceux de manipulation de journaux, d'autant que ces derniers sont suspensibles puisqu'ils risquent de devoir accéder à des mémoires de masse. Un serveur donné peut donc affecter une haute priorité à un service `Trigger Event` qui correspond souvent au déclenchement d'une alarme. Si un service moins critique (tel que `Read Journal`) doit aussi être exécuté au même moment, il n'est pas sélectionné et doit attendre la fin d'exécution du `Trigger Event`. Cette affectation statique des priorités peut être satisfaisante dans certains cas limités. Mais en fait, dans la majorité des situations elle entraîne des applications très peu souples car les clients ne peuvent pas décider de l'importance à accorder à chaque requête de service qu'ils effectuent. Plus précisément, il n'y a :

   – aucune possibilité de changer les priorités d'une classe donnée de services donc aucune façon de modifier la priorité d'une classe de service par rapport à une autre;
   – aucune possibilité de différencier les priorités de requêtes de la même classe ou du même type (une alarme peut être plus critique qu'une autre).

2. Une seconde solution est de donner des priorités à chaque association. Tout service effectué sur une association donnée a alors la même priorité. Cette priorité peut être négociée lors de l'établissement de la connexion. Mais cette proposition revient à affecter une priorité fixe aux clients MMS. Les priorités des services effectués sur une même association ne peuvent être différenciées. Par ailleurs, certains services effectués sur une association de priorité basse peuvent être plus critiques que d'autres pourtant effectués sur une association de priorité haute. Enfin, la négociation de la priorité d'une association implique une modification du protocole MMS.

3. La meilleure solution est pour les applications client d'être libres de spécifier n'importe quelle priorité à n'importe quelle requête de service. Ceci implique que la priorité du service doit être contenue dans chaque requête individuelle et qu'il doit y avoir un moyen pour que les serveurs puissent prendre en compte ces priorités et exécuter les requêtes en conséquence. Cette solution inclut d'ailleurs la proposition d'affectation des priorités aux associations puisqu'il suffit pour un client de fixer la priorité de tous les services qu'il effectue à la même valeur.

La table 5.6 résume ces différentes possibilités.

Mais si la norme laisse une grande liberté quant à l'ordre d'exécution des requêtes, il n'en va pas de même pour l'ajout d'un paramètre de priorité dans les requêtes puisque ceci modifierait le protocole (niveau de modification au moins 4) et rendrait difficile sinon impossible l'interopérabilité avec le protocole MMS classique.

Nous avons donc choisi d'insérer la priorité d'une requête dans son numéro d'identification `InvokeId`. Cet entier non signé de 32 bits est présent dans chaque service confirmé et est utilisé pour identifier la requête de façon unique sur l'association correspondante.

5.3.2 L'affectation des priorités aux services MMS

C'est l'imprécision de la norme MMS (ou la liberté qu'elle laisse) qui nous permet d'utiliser l'attribut `InvokeId` pour transporter la priorité d'une requête de service. En effet, il n'est absolument pas précisé quelles doivent être les valeurs prises par l'`InvokeId`. Un client peut choisir ces valeurs de façon arbitraire dans la mesure où l'identification unique d'un service MMS sur une association est respectée. Des services successifs n'ont donc pas besoin d'avoir leurs `InvokeId` affectés à des valeurs monotones croissantes (ce qui est le cas le plus fréquent dans les applications actuelles).

Mais parce qu'il est nécessaire de conserver le caractère initial des `InvokeId` (identification unique des requêtes en cours) il ne peut pas non plus y avoir une correspondance directe entre les priorités et les `InvokeId`. Cependant deux solutions simples peuvent être adoptées :

- Diviser le domaine de définition des `InvokeId` en N intervalles. Chaque intervalle correspond à un niveau de priorité. Une requête avec un `InvokeId` tombant dans l'intervalle $0 \leq K \leq N$ se voit affectée la priorité K. Par simplicité, on peut affecter aux intervalles des longueurs identiques. Mais ceci n'est pas obligatoire. On peut faire varier la longueur de chaque intervalle en fonction de la fréquence d'utilisation du niveau de priorité qui lui correspond.

- Utiliser une partie de l'`InvokeId` pour coder la priorité. Par exemple on peut choisir les 8 bits les plus significatifs pour coder la priorité ce qui permet donc de coder 256 niveaux de priorité.

Dans la suite, nous adoptons la seconde solution, plus facile à mettre en oeuvre. Pour rester cohérent avec la définition MMS des priorités, notamment pour les sémaphores et les événements, on affecte la priorité la plus haute à la valeur la plus faible c'est-à-dire zéro. Il faut remarquer que l'utilisation de 8 bits pour coder la priorité limite le nombre des valeurs identifiant de façon unique une requête sur un même niveau de priorité à 2^{24}. Mais ceci ne constitue en rien une contrainte pour les applications MMS puisque le cas où il y aurait 2^{24} requêtes **en cours** sur une même association est peu susceptible de se produire, pour ne pas dire irréaliste. De toute façon, le nombre de valeurs différentes d'`InvokeId` nécessaires pour une session MMS n'est jamais plus grand que le nombre maximum de requêtes en attente possibles sur l'association (actuellement typiquement moins de 10). Ceci signifie qu'en fait une étendue des `InvokeId` de 0 à 10 pourrait être suffisante. On pourrait alors coder 2^{28} niveaux de priorité ce qui est tout-à-fait superflu. Et à la limite, comme la majorité des applications MMS sont exécutées en mode synchrone, l'`InvokeId` affecté à leurs requêtes de service peut se limiter à un seul numéro !

Que l'étendue de variation des valeurs identifiant de façon unique une requête sur une association et pour un niveau de priorité soit réduite à 2^{24} au lieu de 2^{32} ne constitue donc en aucun cas une limitation. C'est d'ailleurs la seule restriction introduite par XES. Il y a donc une sous-utilisation de l'`InvokeId` que nous exploitons ici.

Pour des raisons de compatibilité, l'extension proposée ici doit être de niveau de modification 2 tel que défini au chapitre 1. Ceci signifie en particulier qu'un client MMS classique connecté à un serveur supportant cette extension doit être libre de choisir n'importe quelle valeur pour l'attribut `InvokeId` puisque tel est le comportement décrit par la norme. L'identification unique d'un service sur une association se fait donc toujours à partir de l'`InvokeId` intégral et non sur les seuls 24 bits restants.

Qu'un client supporte ou non l'extension et qu'il soit connecté ou non à un serveur XES, il peut dans tous les cas effectuer sur une même association deux requêtes de service dont seules les parties priorités de l'`InvokeId` diffèrent.

5.3.3 L'ordonnancement des services MMS

5.3.3.1 Les services

L'ordonnanceur des services MMS (à ne pas confondre avec l'ordonnanceur global des activités concurrentes d'un serveur) est en fait le Processeur de Transactions. Comme nous l'avons vu le Processeur de Transactions se charge de faire passer les objets transaction d'un état à un autre suivant la machine d'état de la figure 4.5. Ordonnancer les requêtes de service MMS selon leur priorité revient à définir un ordre d'attente pour les files associées aux états des services MMS confirmés et surtout à l'état PRÊT. Les objets transaction des services sont ordonnés dans la file d'attente correspondant à leur état en fonction de leur priorité. Les objets transaction qui ont la même priorité sont ordonnés suivant l'ordre de réception des services correspondants.

Les services dans l'état PRÊT sont sélectionnés comme décrit en 4.4 : quand l'exécution d'un service se termine ou qu'un service passe à l'état BLOQUÉ, l'objet transaction en tête de liste de l'état PRÊT (le plus prioritaire) est sélectionné et passe à l'état EN-COURS. La requête qui lui correspond est donc exécutée.

5.3.3.2 Le cas particulier des actions événementielles

Nous avons déjà vu que les actions événementielles sont exécutées de la même façon que les requêtes de service : elles impliquent la création d'un objet transaction qui suit le graphe d'état de la figure 4.5. Il arrive souvent qu'une notification d'événement dénote un état anormal du système. Il est donc raisonnable de considérer que l'exécution des actions événementielles est plus critique que celle des services normaux. Les actions événementielles constituent ainsi un cas particulier. On peut leur affecter les plus hautes priorités de sorte qu'elles soient exécutées avant toute requête de service en attente.

Il est même possible d'ordonner les actions événementielles entre elles lorsqu'il y a plusieurs événements avec action simultanés. Ceci permet par exemple de souligner le caractère critique d'une alarme par rapport à une autre. Il suffit d'affecter les priorités de 0 à une valeur donnée X aux actions d'événements. Si la plus haute priorité des requêtes de service commence à X+1 alors aucun service ne peut être exécuté avant une action événementielle.

Diverses solutions sont envisageables pour effectuer l'affectation des priorités aux actions événementielles :

1. Les priorités des actions peuvent être pré-établies si les EAs sont des objets définis statiquement dans le serveur par le concepteur. Chaque EA existant a donc une priorité fixée qui lui est propre. Cependant l'application client n'a alors pas le contrôle de ces priorités ce qui comme pour les services est fort peu souple. De plus, ceci ne résout pas le problème de l'affectation de priorité pour les EAs créés dynamiquement par le client au moyen du service `Define Event Action`.

2. Les priorités peuvent être fournies au moyen d'une variable MMS associée à chaque action qui contient la valeur de cette priorité. Le client MMS peut alors écrire dans cette variable et changer la priorité d'une action à volonté et à tout moment.

3. Les priorités sont héritées directement de la priorité de l'événement associé à l'action, c'est-à-dire de l'objet EC.

La solution 2 est la plus souple dans la mesure où elle autorise un changement à tout moment de la priorité de l'action événementielle. D'autre part, celle-ci n'est pas liée à l'événement et peut prendre n'importe quelle valeur. Toutefois, la solution 3 est beaucoup plus élégante et de plus elle se situe dans l'esprit de la gestion d'événements MMS. Il est peu probable et pas logique qu'un événement ayant une priorité globale donnée se voit affecté une autre priorité durant la période d'exécution de l'action qui lui correspond. Par ailleurs, on peut aussi changer la priorité de l'action en changeant celle de l'événement. C'est donc cette troisième solution que nous adoptons. Pour rester compatible avec les priorités des événements MMS on donne aux priorités des actions événementielles la plage de valeurs de 0 à 127 et pour les services la plage de 128 à 255. On aboutit alors à la décomposition présentée sur la figure 5.6 de l'attribut `InvokeId` des objets transaction.

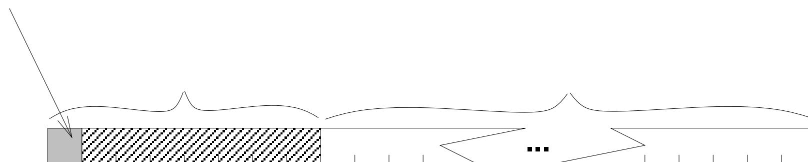


FIG. 5.6 - Structure de l'`InvokeId` utilisé dans XES

Le problème se pose de savoir s'il faut interdire d'affecter une priorité entre 0 et 127 à une requête de service normalement réservée aux actions événementielles. Cette situation est parfaitement envisageable car un client peut affecter n'importe quelle valeur à l'`InvokeId` d'un service. Nous ne souhaitons pas modifier la MMPM pour filtrer ce paramètre. L'extension que nous définissons doit être confinée au serveur. Par ailleurs, l'interdiction d'affecter une priorité entre 0 et 127 à un service revient à supprimer une plage de valeur pour les `InvokeId` ce qui est contraire à la norme MMS et risque d'introduire des incompatibilités entre les clients classiques et les serveurs XES.

Il est toujours possible de considérer que le bit de poids fort d'un `InvokeId` est à 1 systématiquement quelque soit sa valeur. Mais il devient alors inutile de coder la priorité d'un service sur huit bits de l'attribut `InvokeId` puisque sept suffisent. En fait il peut être intéressant de conserver la possibilité à des services dont l'urgence est extrême de faire passer leur exécution avant celle de certaines actions événementielles. Nous choisissons donc cette dernière solution. Il appartient alors aux clients XES de n'utiliser la plage de priorités 0-127 que pour les services qui requièrent vraiment un traitement prioritaire sur les actions événementielles.

5.3.4 L'inversion de priorité

Un des problèmes qui peut surgir suite à l'introduction de priorités dans l'exécution des services MMS est l'*inversion de priorité*. Ce problème connu dans le domaine de l'ordonnancement de tâches se traduit par le blocage d'une tâche par une ou plusieurs tâches moins prioritaires et ce pour une durée indéterminée [LR80], [SRL90].

	Priorité fixe	Priorité par association	Priorité par requête
Souplesse d'utilisation	faible	moyenne	grande
Utilisation de l'`InvokeId`	non	non	oui
Niveau de modification	2	4	2
Possibilité d'associer des échéances avec un niveau de modif. 2	non	non	oui
Combinaison avec les événements	non	non	oui
Difficulté relative d'implantation	facile	moyen	difficile

TAB. 5.6 - Comparaison entre les trois façons d'affecter des priorités dans MMS

Dans le contexte de MMS on peut considérer l'exécution d'une requête de service comme celle d'une tâche. Mais on peut étendre cette notion de tâche aux clients MMS en attribuant à chaque application client une priorité fixe. Dans ce cas tous les services issus d'un même client ont la même priorité. L'inversion de priorité se traduit alors de la façon suivante (fig. 5.7). Supposons qu'un client A_1 effectue une requête `Take Control` sur un sémaphore à un jeton s avec une priorité p_1[4] (étape 2 fig. 5.7). Cette requête passe à l'état BLOQUÉ parce que le jeton de s est déjà pris par une autre application client notée A_N (étape 1 fig. 5.7). Supposons qu'un ensemble de requêtes ayant des priorités p_i inférieures à p_1 soient dans l'état PRÊT. Si maintenant A_N envoie un service `Relinquish Control` sur s avec une priorité p_N inférieure à tous les p_i alors le client le plus prioritaire A_1 se retrouve bloqué par les requêtes de clients moins prioritaires (étapes 3 et 4 fig. 5.7). En effet, le `Relinquish Control` ne peut être exécuté et donc s ne peut être relâché puisque le serveur doit d'abord traiter les requêtes de priorité supérieure à p_N. Ceci s'aggrave encore plus si l'application A_N doit effectuer d'autres services avant le `Relinquish Control` avec des priorités également inférieures aux priorités p_i.

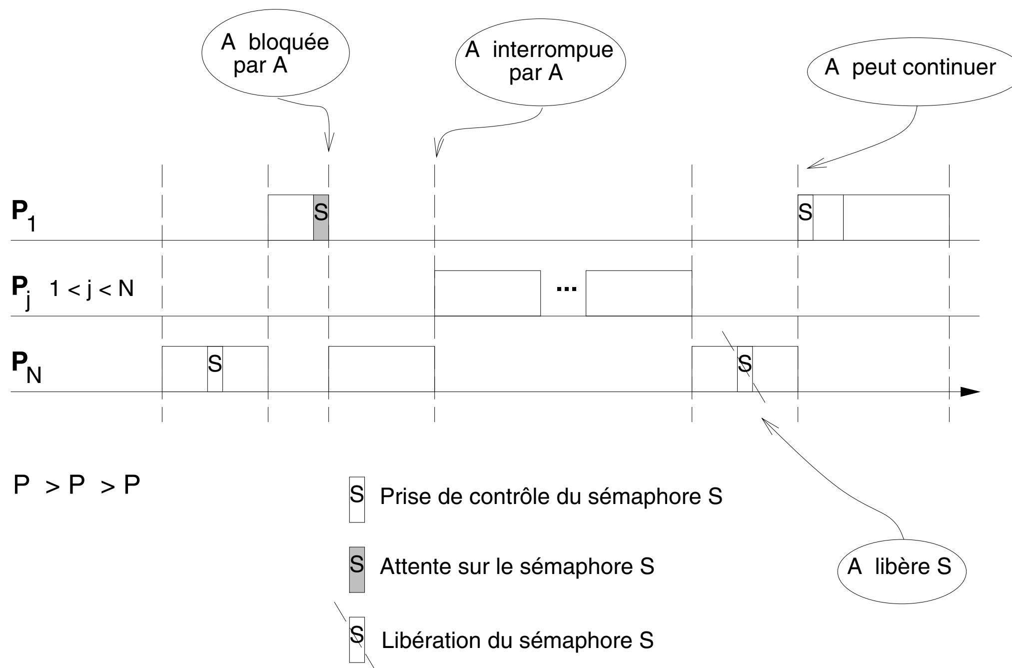


FIG. 5.7 - L'inversion de priorité avec les services MMS

[4] p_1 est la priorité donnée par l'`InvokeId`, ce n'est pas le paramètre MMS `Priority` de la requête `Take Control`.

Dans le domaine de l'ordonnancement de tâches, la première solution préconisée est d'empêcher qu'une tâche s'exécutant dans une section critique puisse être interrompue par une autre tâche et ce quelque soit sa priorité [SRL90], [BW90]. Mais ceci n'est satisfaisant que si les sections critiques des tâches sont de durée très courte ce qui n'est pas le cas dans MMS. En effet, ceci reviendrait à interdire l'exécution de tout service MMS dans un serveur autre que ceux provenant du dernier client ayant fait un `Take Control`.

Une autre solution au problème de l'inversion de priorité est la technique dite d'*héritage de priorité* [SRL90]. Dans le cadre de MMS, ceci consiste à augmenter la priorité de toutes les requêtes d'un client détenant un sémaphore à la plus haute priorité de tous les `Take Control` en attente transitive[5] sur ce sémaphore. Dans notre exemple, tous les services effectués par A_N y compris le `Relinquish Control` se font avec une priorité p_1. Ils s'exécutent donc avant les services de priorité p_i et le sémaphore s est relâché, libérant ainsi l'application la plus prioritaire A_1.

L'héritage de priorité doit être fait automatiquement par la VMD, c'est-à-dire sans demander à l'application client de changer la priorité transportée par l'`InvokeId`. Il est clair cependant que l'attribut `InvokeId` de l'objet transaction ne doit pas lui-même être modifié puisque c'est lui qui permet de garantir l'unicité d'une requête de service sur une association et permet au client d'identifier de façon univoque la réponse du service. Les serveurs MMS supportant XES doivent donc :

- lors du passage des états REÇU ou MODIFIÉ à l'état PRÊT d'une requête de service provenant d'un client détenant un sémaphore et sur lequel au moins un `Take Control` est en attente :
  - ne pas considérer la priorité transportée par l'`InvokeId` de cette requête;
  - affecter sa priorité au maximum entre sa priorité actuelle et la plus haute priorité de tous les `Take Control` en attente transitive sur ce sémaphore sans pour autant modifier l'attribut `InvokeId`;
  - insérer cette requête dans la file d'attente de l'état PRÊT selon sa nouvelle priorité;
- lors du passage à l'état BLOQUÉ d'une requête `Take Control` sur un sémaphore donné :
  - déterminer le sémaphore s sur lequel le `Take Control` est en attente transitive;
  - pour chaque service dans l'état PRÊT et appartenant au client détenant s, affecter sa priorité au maximum entre la priorité du `Take Control` et la priorité actuelle du service sans pour autant modifier l'attribut `InvokeId`;
  - réordonnancer la file d'attente de l'état PRÊT en fonction des nouvelles priorités.

La nécessité de différencier l'`InvokeId` et la priorité d'un objet transaction lorsqu'il y a inversion de priorité implique une extension de cet objet transaction. Celle-ci est décrite au paragraphe suivant.

L'inversion de priorité apparaît lorsqu'il y a compétition pour l'acquisition de ressources exclusives. Le cas s'applique donc particulièrement aux sémaphores MMS à jeton unique. Mais plus généralement, tout service bloquant pouvant exclure un autre service durant son exécution dès lors qu'il se trouve dans l'état BLOQUÉ peut entraîner une inversion de priorité. C'est le cas par exemple des services `Input` et `Output` qui peuvent nécessiter la prise de contrôle exclusive d'un clavier ou d'un écran [Cas94a]. L'inversion de priorité n'apparaît toutefois que si un service bloqué doit attendre la fin de l'exécution d'autres services plus prioritaires **alors que les conditions de son déblocage sont satisfaites**.

L'ordre d'attente des services `Take Control` dans l'état BLOQUÉ n'a pas d'importance car c'est l'attribut `Priority` de chaque requête `Take Control` qui détermine la prochaine de ces requêtes qui

[5] Un `Take Control` d'un client A est en attente transitive sur un sémaphore s s'il est bloqué sur s ou sur un sémaphore différent de s détenu par un client B en attente transitive sur s.

obtiendra le sémaphore. Ceci pose toutefois un problème que nous étudions plus loin. Par contre cet ordre est important pour les autres services bloquants. Ceux-ci doivent être débloqués dans l'ordre de leur priorité.

L'inversion de priorité décrite jusqu'ici est un problème d'ordonnancement des applications client MMS et non pas des requêtes. En effet, dans l'exemple de la figure 5.7 il n'y a dans le serveur aucune requête de priorité haute prête à être exécutée car l'application A_1 n'envoie pas de requête de service tant qu'elle n'a pas reçu la réponse du `Take Control`. Mais ce comportement dépend de A_1. Si A_1 envoyait des requêtes de priorité haute alors que le sémaphore ne lui est pas alloué, ces requêtes seraient malgré tout exécutées par le serveur. L'inversion de priorité se situe donc à un niveau d'abstraction plus élevé car c'est l'application client qui est bloquée. On retrouve ici l'analogie faite à la section 4.6 entre un serveur MMS et le noyau d'un système d'exploitation. De la même façon qu'un OS ordonne les processus dans une machine, le serveur MMS ordonne partiellement l'exécution des applications client. L'ordonnancement des requêtes dans un serveur et celui des applications client utilisant ce serveur sont deux problèmes liés mais différents.

Il peut également se produire une inversion de priorité entre requêtes MMS lorsque celles-ci accèdent de façon mutuellement exclusive à des objets MMS. Le problème est alors fortement dépendant de la politique et des algorithmes d'ordonnancement utilisés pour implanter le serveur MMS. Ce type d'inversion de priorité est uniquement un problème d'implantation des serveurs et non pas d'interaction client/serveur.

5.3.5 Problèmes dus à l'héritage de priorité

Le bon fonctionnement de l'héritage de priorité suppose que lorsqu'un sémaphore est libéré, la tâche de plus haute priorité en attente sur ce sémaphore en prend le contrôle. Cette hypothèse est nécessaire pour éviter les blocages inutiles de tâches ayant des priorités élevées [BW90]. Mais il faut remarquer que cette hypothèse n'est pas nécessairement remplie dans le contexte de MMS. Pour une requête `Take Control` i en attente sur un sémaphore s, notons p_i^{XES} sa priorité transportée par l'`Invoked` et p_i^{MMS} la priorité éventuellement transportée par l'attribut `Priority` du `Take Control`. Lors de la libération de s, l'allocation du jeton libéré se fait sur la base des priorités p_i^{MMS} et non des p_i^{XES} (fig. 5.8).

Il suit que l'hypothèse précédente n'est remplie et donc que la méthode par héritage de priorité ne fonctionne que si les priorités p_i^{MMS} et p_i^{XES} de N `Take Control` sur un sémaphore s sont ordonnées de la même façon c'est-à-dire :

$$\forall i, j \; 1 \leq i, j \leq N,$$

$$(p_i^{MMS} \prec p_j^{MMS}) \Leftrightarrow (p_i^{XES} \prec p_j^{XES})$$

avec la relation $\prec$ définie de la façon suivante $x \prec y$ si :

- $x < y$ ou
- $x = y$ et la requête correspondant à x est arrivée au serveur avant celle correspondant à y.

Lorsque cet ordre n'est pas respecté, une solution consiste à modifier les priorités MMS des `Take Control` bloqués sur un sémaphore de façon à agencer la liste des “Semaphore Entry” dans l'état `QUEUED` dans le même ordre que celui donné par les priorités XES. Toutefois, cette solution modifie le comportement du serveur et rend l'extension XES de niveau 3 ce que nous ne souhaitons pas.

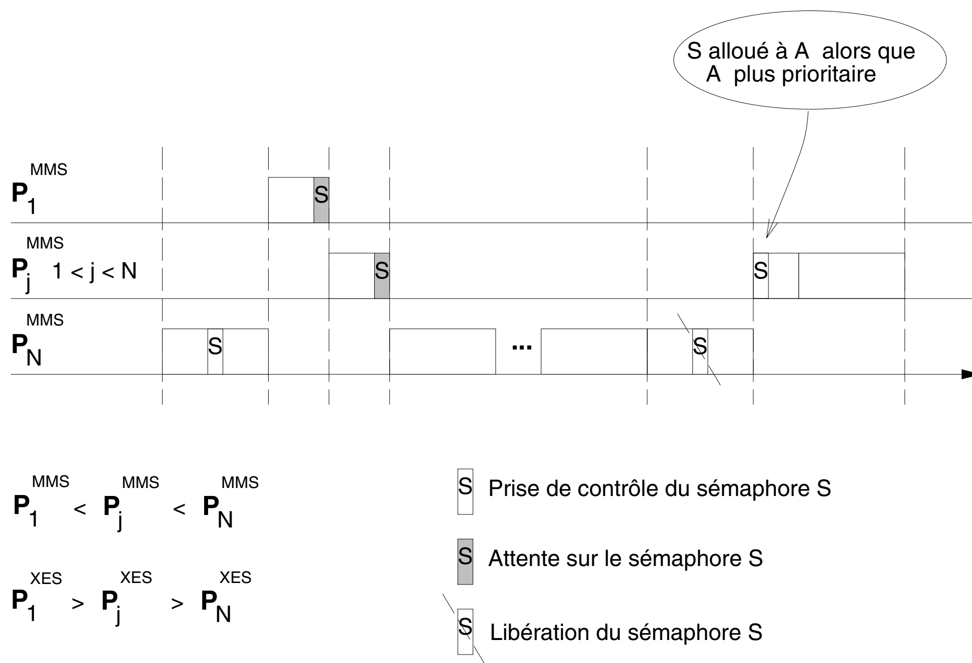


FIG. 5.8 - L'héritage de priorité rendu inutile

Dans le domaine de l'ordonnacement de tâches, cette situation revient à changer dynamiquement sur décision seule des tâches la priorité qui leur est allouée. L'ordonnanceur n'a aucune influence sur ce changement de priorité. Nous n'avons pas connaissance d'algorithmes permettant d'éviter l'inversion de priorité dans de telles situations.

Un des défauts majeurs de l'héritage de priorité réside dans son incapacité à éviter la formation d'interblocages. Dans [SRL90], [SRSC90] Sha *et al.* ont proposé une seconde méthode pour éviter l'inversion de priorité. Cette méthode connue sous le nom de *Priority Ceiling Protocol* (PCP) présente par rapport à l'héritage de priorité l'avantage d'éviter la formation de tout interblocage. Par contre, sa mise en oeuvre est plus complexe que pour l'héritage de priorité.

L'adaptation de PCP à MMS ne semble pas utile dans la mesure où le serveur MMS se charge normalement de la détection des interblocages indépendamment de XES. En effet, nous avons montré à la section 2.5 que le concepteur de serveur MMS avait une entière liberté pour implanter une VMD qui évite ou résout les interblocages. En présence d'une telle VMD, l'adaptation de PCP à MMS est redondant avec le mécanisme existant de détection des interblocages. Toutefois, PCP peut justement être le mécanisme choisi pour éviter les interblocages. Dans ce cas, l'intégration de XES à un tel serveur se fait naturellement.

5.3.6 Impact de XES sur la norme MMS

XES se veut une extension de niveau 2 telle que définie au chapitre 1. Ceci signifie que la compatibilité entre les clients classiques et les serveurs XES est assurée. On notera toutefois que les clients classiques peuvent voir leurs requêtes de service retardées ou exécutées avant d'autres selon la valeur des `InvokeId` qu'ils utilisent. En particulier, les services de la majorité des clients classiques sont exécutés en priorité par rapport aux autres puisque les `InvokeId` sont typiquement affectés de façon monotone croissante en partant de zéro. Dans un tel cas, la priorité d'un service ne passe dans la plage 128-255 qu'à partir du $2^{31^{eme}}$ service sur l'association considérée !

Pour minimiser cet effet sur les clients qui eux font usage de XES, on peut introduire au moment de l'établissement de la connexion avec le serveur une négociation sur l'utilisation ou non des priorités. L'aptitude lors d'une association client-serveur à supporter ou utiliser certaines fonctionnalités apparaît dans le paramètre `Parameter Support Options` de type `Bitstring`. Chaque fonctionnalité supportée correspond à un bit positioné à 1. L'aptitude à utiliser XES peut de même être représentée par un bit qui selon qu'il soit à 0 ou 1 indique si cette extension doit être considérée ou non pour l'association courante. Nous ne pouvons envisager de modifier ce paramètre pour y rajouter le support de XES sans obtenir un serveur modifié de niveau 4. Par contre, il est possible d'utiliser le bit 9 dont la signification et l'utilité ont été annulées par un rapport ISO corrigeant les défauts de la norme MMS [ISO92b]. Les clients classiques positionnent déjà ce bit à 0. Il suffit aux clients XES de le positionner à 1 pour informer le serveur de leur intention de faire usage des priorités dans les requêtes de service qu'ils effectueront.

Pour minimiser l'influence des services des clients classiques sur les services de clients utilisant les priorités ainsi que sur les actions événementielles, il suffit alors que les serveurs affectent une priorité fixe de valeur moyenne (donc ici égale à 192) à tout service effectué sur une association où le support de l'extension XES n'a pas été négocié. Le protocole MMS n'est par conséquent pas modifié et le serveur reste de niveau de modification 2.

Pour mieux appréhender l'extension XES proposée, il devient utile d'étendre la définition MMS des objets transaction de façon à inclure les trois attributs suivants :

- `Priority Assignment Scheme` qui prend les valeurs `ACTIVE` ou `UNUSED` et spécifie si l'objet transaction est défini sur une association qui supporte ou non XES;
- `State` pour spécifier dans quel état l'objet transaction se trouve;
- `Priority` pour sauvegarder la priorité de l'objet transaction. Cet attribut prend la priorité portée par l'`InvokeId` lors de la création de l'objet transaction s'il représente une requête de service. Il prend la priorité d'un EC s'il représente une action événementielle. Il peut être modifié par la VMD à tout moment essentiellement pour éviter l'inversion de priorité et permet donc de préserver l'attribut `InvokeId`.

Les deux derniers attributs sont définis uniquement si l'attribut `Priority Assignment Scheme` a la valeur `ACTIVE`. La définition complète de l'objet transaction étendu est alors représentée sur la figure 5.9 suivant le format MMS.

Notons également que l'attribut `List Of Capabilities` de la VMD doit comprendre une valeur indiquant l'aptitude du serveur à supporter XES. Chaque élément de cette liste identifie la capacité d'une VMD à effectuer une certaine tâche. La signification exacte de chacun de ces éléments et donc des tâches associées est propre à chaque implantation. Nous pouvons donc librement y inclure XES.

5.3.7 Ordonnancement avec des échéances

Il est possible d'utiliser XES en fournissant directement la valeur d'un délai dans le paramètre `InvokeId` en lieu et place de la priorité du service. Ce délai représente l'échéance avant laquelle la requête doit être exécutée à partir du moment où elle est reçue par le serveur. On peut alors utiliser un algorithme tel que *Earliest Deadline First* (EDF) [LL73] pour traiter les requêtes de services. En fait il suffit de réorganiser la file d'attente de l'état PRÊT selon l'ordre croissant des délais. Le mécanisme d'ordonnancement des requêtes est ensuite le même.

Pour coder l'échéance dans une requête de service MMS, il faut toutefois augmenter la taille de la partie de l'`InvokeId` utilisée pour le codage par rapport à celle que l'on utilisait pour traiter les

```
Object: Transaction

  Key Attribute: InvokeId
  Key Attribute: Application Association Identifier
  Attribute: List Of Pre-execution Modifiers
  Attribute: Current Modifier Reference
  Attribute: Confirmed Service Request
  Attribute: List Of Post-execution Modifiers
  Attribute: Cancelable (TRUE, FALSE)
  Attribute: Priority Assignment Scheme (ACTIVE, UNUSED)
  Constraint: Priority Assignment Scheme = ACTIVE
        Attribute: State
        Attribute: Priority
```

FIG. 5.9 - L'objet transaction étendu pour XES

priorités. Si notre unité de temps est la milliseconde et que l'on n'utilise que 8 bits de l'`InvokeId` on ne peut pas donner de délai plus grand que 256 ms. Avec 16 bits, on peut coder des délais à peine plus grands que la minute. Avec 24 bits nous arrivons à plus de 4 heures et demi ce qui semble être largement suffisant. Le nombre de requêtes en cours effectuées sur une association donnée et qui ont le même délai est alors limité à 256 ce qui n'est toujours pas une limitation très importante pour les mêmes raisons que celles exprimées plus haut. Nous consacrons une partie du chapitre 6 à l'étude de l'exécution temps-réel des requêtes de service MMS. Nous étudions alors plus en détail une solution similaire mais qui n'utilise pas les `InvokeId`s.

5.3.8 Conclusion sur XES

Nous avons utilisé le modèle d'exécution des services MMS défini en 4.2.4 pour illustrer comment les requêtes de service MMS peuvent recevoir des priorités et comment les exécutions de ces services sont ordonnancées dans un serveur MMS. L'affectation de priorités aux services MMS n'est pas considérée dans la norme. Notre solution est une extension de niveau 2 et présente l'avantage de ne pas modifier le protocole MMS ainsi que de rester compatible avec les applications MMS existantes. Par ailleurs, la solution proposée ne fait pas appel à des capacités de MMS rarement supportées par les produits actuels (sémaphores, événements ou modificateurs). Seul l'attribut `InvokeId` est utilisé et celui-ci et toujours présent dans toute implantation MMS même la plus simple. Notre extension est donc applicable à des serveurs MMS très simples supportant un nombre très faible de services (typiquement `Read` et `Write` seulement).

Deux problèmes ont toutefois été introduits par cette extension :

- l'inversion de priorité qui est partiellement résolue avec la technique d'héritage de priorité appliquée dans le contexte des serveurs MMS;
- l'indétermination quant à l'ordre d'exécution des services MMS de clients classiques dialoguant avec des serveurs XES dont nous n'avons pu que minimiser l'impact en proposant une négociation de l'aptitude à supporter XES au moment de l'établissement de l'association.

5.4 Conclusion

Ce chapitre était orienté vers une **amélioration** de la norme MMS. Nous avons présenté deux extensions à la norme MMS qui permettent de mieux satisfaire les besoins des utilisateurs. L'extension XED permet aux utilisateurs de définir avec précision les conditions de déclenchement d'un événement dans un serveur MMS ce qui n'est pas possible en l'état actuel de la norme. Ces conditions sont définies par un prédicat incorporé dans une variable MMS elle-même associée à l'événement à scruter. Nous avons effectué une implantation de cette extension et nous l'avons intégrée à notre serveur MMS. Quelques mesures de performances ont été effectuées. Elles permettent de chiffrer le surcoût en terme de code et de temps de traitement de notre extension pour une implantation donnée.

L'extension XES quant à elle facilite l'ordonnancement des requêtes de service dans les serveurs MMS en permettant à tout client d'associer une priorité à chacune de ses requêtes. Pour ce faire, nous exploitons la liberté laissée aux concepteurs d'applications par la norme MMS et utilisons le paramètre `InvokeId` pour inclure une priorité dans une requête de service. Ces deux extensions ne modifient pas le protocole MMS et sont de niveau de modification 2. Elles préservent donc la compatibilité des applications MMS.

L'extension XES nous permet d'effectuer une transition vers le chapitre suivant plus particulièrement dédié à l'étude de l'ordonnancement dans un serveur MMS.

Chapitre 6

MMS et ordonnancement

6.1 Introduction

L'objet de ce chapitre est d'étudier les problèmes d'ordonnancement qui peuvent se poser dans les serveurs MMS et d'analyser les capacités des systèmes MMS à garantir le respect de contraintes de temps.

Il est généralement admis que le modèle OSI n'est pas adapté aux besoins des applications temps-réel. Mais comme le souligne Grant dans [Gra92], le fait qu'OSI soit inapproprié ne signifie pas que MMS le soit également. L'intégration de MMS dans un environnement temps-réel n'a jusqu'ici pas été traité de façon très approfondie. Récemment, certains travaux ont été proposés pour étendre MMS avec des fonctionnalités temps-réel [RZI90], [AM95a], [AM95c], [EMR95], [MA95]. Toutefois, la plupart de ces études impliquent une modification du protocole MMS, un ajout de nouveaux services et objets, une addition de paramètres aux services et/ou ne traitent pas de la façon d'ordonnancer les requêtes dans les serveurs. De tels changements au protocole MMS se situent au niveau de modification 4 ou 5. Dans ce chapitre, pour des raisons de compatibilité nous faisons l'hypothèse que le protocole MMS ne peut pas être modifié. Nous cherchons donc à limiter notre niveau de modification à 2 ou au maximum à 3. Pour certaines des solutions exposées dans ce chapitre, nous pouvons même nous contenter du niveau de modification 1.

Nous sommes intéressés par l'*exécution temps-réel* des requêtes de service c'est-à-dire, du point de vue des clients, par la capacité de borner le temps de réponse d'un service MMS et donc par la capacité d'un serveur à ordonnancer les requêtes de services MMS. Nous ne prétendons pas résoudre tous les problèmes temps-réel qui peuvent se poser dans un système d'applications MMS. Ainsi, les problèmes soulevés dans [RZI90] tels que :

- l'obtention d'un temps global de référence;
- la synchronisation des horloges entre les sites MMS;
- l'acquisition et l'utilisation de la date de production de la valeur d'une variable;

ne sont pas considérés ici.

Dans ce contexte, le but et les contributions de ce chapitre sont les suivants :

- **Analyse des limitations de MMS** dans le domaine de l'ordonnancement temps-réel. Que faut-il améliorer pour pouvoir satisfaire une exécution temps-réel des requêtes de service et des autres activités d'un serveur ? Est-il possible de faire du temps-réel strict dans MMS ?

- **Etude des possibilités déjà offertes par la norme MMS** dans le domaine du temps-réel. Nous montrons à ce propos qu'il est possible sans modifier la norme d'associer une échéance aux requêtes de service MMS et d'assurer une exécution temps-réel de ces requêtes.
- Proposition d'un mécanisme possible pour **assurer une détection temps-réel des événements** MMS.
- **Intégration de XED** dans la détection temps-réel des événements.
- Proposition d'une **méthode de négociation** des paramètres permettant l'établissement d'une association temps-réel.

Ici encore, nous cherchons à explorer et à exploiter les possibilités offertes par la norme avant d'effectuer toute modification. Nous montrons que dans certains cas MMS offre plus de possibilités que ce que l'on pourrait croire à première vue. Nous restons donc fidèle à cet esprit même s'il est probable qu'une modification importante de la norme MMS dans le but de satisfaire des problèmes temps-réel pourrait se révéler plus satisfaisante.

De très nombreux travaux ont été réalisés dans le domaine du temps-réel et de l'ordonnancement. Le but de ce chapitre n'est pas de définir de nouveaux algorithmes d'ordonnancement mais de montrer comment exploiter certains résultats connus dans le contexte de MMS. Nous ne prétendons pas pouvoir couvrir l'intégralité des problèmes qui se posent dans le domaine du temps-réel et de MMS. Ce sujet est suffisamment complexe et important pour nécessiter une thèse à lui seul. **Ce chapitre n'est donc destiné qu'à ouvrir des perspectives sur MMS dans un environnement temps réel** et nous n'offrons qu'une vue partielle de l'ensemble des problèmes qui se posent.

Nous adoptons la démarche suivante. Nous commençons par préciser les fonctionnalités d'un serveur MMS auxquelles on peut associer des contraintes de temps. Puis, nous déterminons les limitations qu'il faut apporter à tout système MMS qui doit satisfaire des contraintes de temps. Nous traitons ensuite des systèmes à contraintes de temps strictes puis lâches. Dans ce dernier cas, nous proposons des solutions basées sur les modificateurs MMS pour contrôler l'exécution des requêtes de service. Nous terminons par une étude sur la détection des événements MMS.

6.1.1 Bref état de l'art

Il existe assez peu de contributions sur MMS et le temps-réel.

Dans [RZI90] et [ZR91], Rodd *et al.* identifient les caractéristiques essentielles des systèmes temps-réel répartis et les comparent à celles offertes par MAP/MMS. Les auteurs suggèrent que la norme MMS actuelle présente beaucoup de défauts quand elle est utilisée avec de tels systèmes. L'absence d'un temps global de référence et l'utilisation de communications en mode orienté-connexion par opposition au mode datagramme sont interprétés comme étant les limitations principales de MAP/MMS. Une proposition est faite pour étendre MMS avec des capacités temps-réel. De nouveaux attributs sont rajoutés à certains objets et de nouveaux objets et services sont créés.

Une partie du travail de [RZI90] est reprise et étendue dans [AM95a] et [AM95c]. Les auteurs proposent des solutions pour borner le temps de réponse des services MMS. Ils modifient également la norme MMS pour pouvoir prendre en compte les caractéristiques temps-réel. De nouveaux paramètres sont ajoutés à chaque requête de service et certains objets MMS sont étendus avec de nouveaux attributs. Une attention particulière est accordée à l'exécution des invocations de programmes MMS.

Dans [AM95b], Akazan *et al.* analysent également la norme MMS du point de vue des systèmes critiques en temps. Les auteurs suggèrent que le temps de réponse des services MMS souffre d'indé-

terminisme car de nombreux paramètres tels que la charge du serveur, l'ordre dans lequel les requêtes arrivent au serveur ainsi que les délais de transmission des messages sont indéterminés.

Une autre analyse du temps-réel avec MAP/MMS est effectuée par Grant dans [Gra92]. L'auteur insiste sur la nécessité d'avoir des temps de réponse connus. Cette étude montre que l'absence d'un estampillage des PDUs avec une priorité pouvant être prise en compte à chaque couche du modèle OSI invalide la possibilité d'avoir un traitement favorisant la transmission des messages urgents.

Toutes les références citées précédemment fournissent une description des problèmes et des limitations des systèmes MAP/MMS en ce qui concerne le temps-réel. Il apparaît cependant que la majorité de ces problèmes provient de la pile de communication MAP et pas directement de MMS. Il est d'ailleurs généralement admis que le modèle de référence OSI ne répond pas de façon satisfaisante aux exigences des systèmes temps-réel [Bur91], [Gra92].

Une séparation nette entre les problèmes OSI et MMS devrait donc être faite. Il paraît certain que MMS n'a pas été conçu sur des bases temps-réel. Cependant, nous pensons aussi que les capacités de MMS n'ont pas été complètement comprises et exploitées. Par ailleurs, il est également important de faire une séparation claire entre le respect des spécifications MMS et l'implantation qui en est faite. En terme d'interopérabilité et de compatibilité l'important reste que les applications industrielles "parlent" MMS. L'implantation de ces applications n'est pas du ressort de MMS et tout peut être fait dans la mesure où la norme est respectée. La norme MMS laisse donc une marge de créativité dans la conception d'un serveur. C'est cette marge que nous exploitons.

6.1.2 Définitions

Nous commençons par quelques définitions des termes qui sont utilisés dans ce chapitre. Nous utilisons les états d'un service MMS définis à la section 4.2.4. Les temps définis ci-dessous sont aussi représentés sur la figure 6.1 pour plus de clarté.

Temps de transfert: c'est le temps entre le moment où le client (resp. serveur) insère une requête (resp. réponse) dans `QueueEnvois`[1] et le moment où cette requête (resp. réponse) arrive dans `QueueReception` du côté serveur (resp. client). Il s'agit donc uniquement du temps passé dans les protocoles de communication.

Temps d'attente: c'est le temps que passe une requête MMS dans `QueueReceptions`, c'est-à-dire le temps passé dans l'état REÇU dans le serveur.

Temps d'exécution: c'est le temps mis par un serveur pour exécuter intégralement une requête de service entre le moment où elle est retirée de `QueueReceptions` et le moment où la réponse est insérée dans `QueueEnvois`. En d'autres termes, c'est le temps s'écoulant entre le moment où la requête est retirée de l'état REÇU et passe dans l'état NON-EXISTANT.

Temps de traitement: c'est le temps mis par un serveur pour exécuter intégralement une requête de service en dehors de toute autre tâche à exécuter. C'est également le temps s'écoulant entre le moment où la requête est retirée de l'état PRÊT et passe dans l'état NON-EXISTANT lorsque la requête ne suit que les transitions 1, 2, 7, 10 de la machine d'états de la figure 4.5 et lorsque le processeur n'est jamais affecté à l'exécution d'une autre tâche du serveur.

Temps de service: c'est le temps total passé par une requête dans le serveur. En d'autres termes, c'est le temps s'écoulant entre le moment où la requête est insérée dans la file d'attente de l'état REÇU et passe dans l'état NON-EXISTANT.

[1] `QueueEnvois` et `QueueReceptions` ont été définies à la section 4.1.2.

Temps de réponse: c'est le temps observé par un client MMS entre le moment où il envoie une requête et le moment où il reçoit la réponse correspondante. C'est donc le temps s'écoulant entre le moment où le client insère une requête dans `QueueEnvois` et que la réponse correspondante est insérée dans `QueueReception`.

Association temps-réel: association MMS sur laquelle les temps de réponse de tous les services MMS sont garantis.

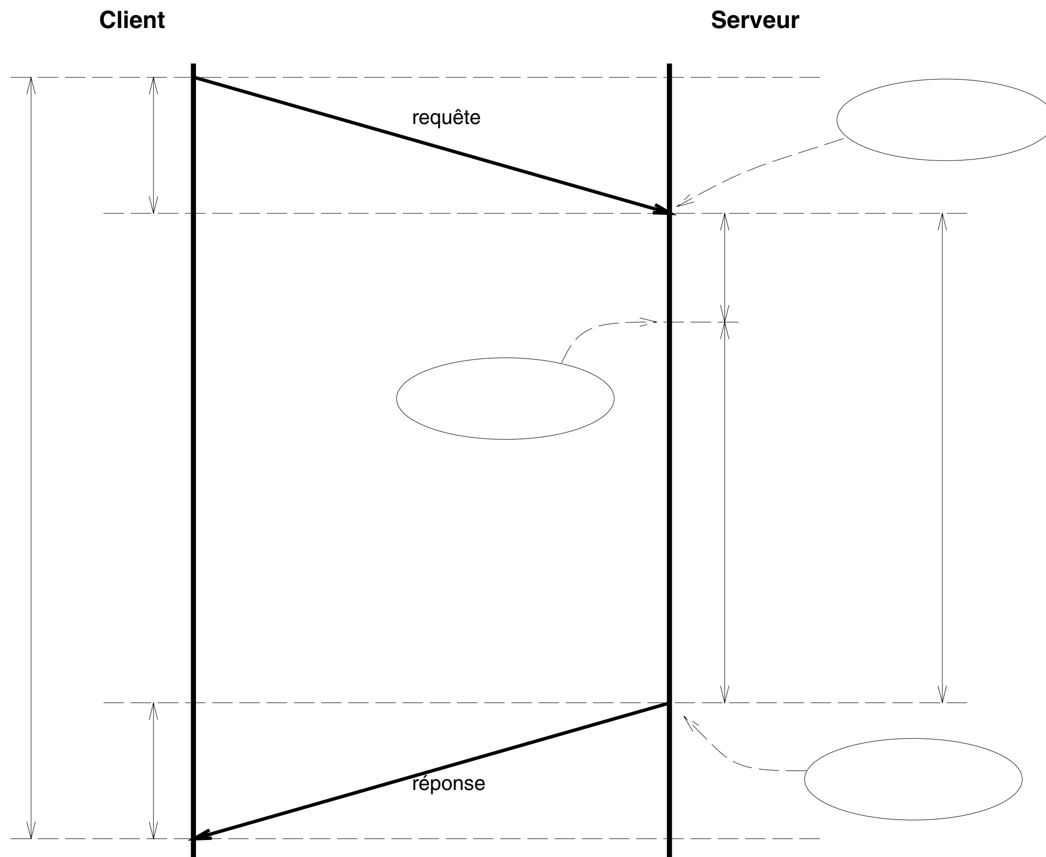


FIG. 6.1 - Définition des temps utilisés dans ce chapitre

6.1.3 Modèle et exigences

Du point de vue d'un client MMS, l'important est que le temps de réponse des services qu'il effectue soit borné. La première approche à suivre est alors d'implanter au niveau du client un test pour vérifier que les réponses des requêtes envoyées arrivent dans les temps impartis. Cette solution a été adoptée dans [AM95a] et définie formellement dans [AM95c] et [EMR95] au moyen d'automates de Nutt. Sans entrer dans les détails, le principe de base est d'ajouter une couche logicielle supplémentaire entre l'application client et le fournisseur de services MMS. Cette couche s'occupe des aspects temporels des requêtes MMS. Quand une requête est effectuée, un temporisateur est démarré. Ce temporisateur est initialisé avec une valeur représentant le temps maximum pendant lequel le client désire attendre la réponse de sa requête. Si la réponse est reçue avant la fin de temporisation alors elle est retransmise au client et le temporisateur est détruit. Si par contre la réponse n'est pas reçue avant la fin de temporisation alors le client reçoit un message d'erreur et sait qu'il doit entreprendre des actions correctives. Ainsi la vérification du respect des contraintes de temps se fait du côté client et ne

nécessite aucune modification à MMS. En fait, tout peut être fait du côté client puisqu'il s'agit de l'application utilisateur et que celle-ci n'est pas normalisée par MMS.

En pratique, une simple vérification du respect des contraintes de temps par le client ne suffit pas. Un client doit aussi être capable de demander à un serveur de traiter une requête dans un temps imparti. Le temps qu'une requête passe dans un serveur doit donc être borné si l'on désire borner le temps de réponse total.

Dans ce chapitre nous considérons le serveur MMS indépendamment de la pile de communication sous-jacente. Nous limitons notre étude aux applications MMS seules et nous considérons que les protocoles de communication et le serveur sont exécutés sur deux processeurs différents. Nous faisons l'hypothèse que **l'architecture de communication utilisée est toujours capable de borner les temps de transfert des PDUs MMS**. Cette hypothèse nous permet de nous concentrer uniquement sur les temps de service des requêtes MMS au niveau des serveurs; ceci dans le but de borner le temps de réponse total d'un service.

Un serveur MMS est vu comme un ensemble de tâches dont certaines sont périodiques et d'autres déclenchées par un stimulus extérieur au système. Le stimulus le plus typique est l'arrivée d'une requête de service. L'arrivée de chaque requête déclenche donc l'apparition d'une tâche dans le système. Cette tâche est alors dans l'état REÇU. Elle se termine quand la réponse correspondante est envoyée. La tâche passe alors dans l'état NON-EXISTANT. La requête correspondante est représentée de façon unique par un objet transaction qui peut donc être considéré comme un descripteur de tâche.

Dans le but de faciliter l'analyse, nous allons faire un certain nombre de simplifications :

- Dans un premier temps, les seules activités concurrentes considérées sont les requêtes de service MMS. Chaque requête de service MMS est exécutée par une tâche. A la section 6.6, nous traitons des activités concurrentes nécessaires au traitement des événements.

- Le temps d'exécution des routines implantant l'ordonnanceur est négligeable. Cela signifie que toutes les transitions entre les différents états d'un service confirmé (section 4.2.4) sont considérées comme nécessitant un temps nul.

- Le temps d'attente est considéré comme nul. Une requête reçue passe donc immédiatement dans l'état PRÊT. Les temps de transfert des PDUs MMS étant bornés, on peut alors connaître le temps de réponse maximum d'une requête de service MMS si l'on arrive à borner son temps d'exécution dans le serveur MMS.

Nous montrerons dans certains cas comment lever ces deux dernières hypothèses.

6.2 Modèles de serveurs MMS

Un serveur MMS est un système compliqué si l'on considère l'intégralité des fonctions proposées par la norme. Nous avons proposé au chapitre 4 une architecture supportant toutes ces fonctionnalités. L'addition de contraintes temps-réel rend la conception d'un serveur encore plus complexe. Le plus souvent, seul un sous-ensemble de fonctionnalités s'avère nécessaire pour satisfaire une application donnée. Par exemple, certains serveurs ne fournissent que les requêtes de lecture et d'écriture et ne s'en servent que pour transférer de l'information. L'idée retenue est alors simplement l'aptitude de MMS à résoudre les problèmes d'hétérogénéité, mais les capacités réelles de MMS ne sont pas exploitées.

L'intégration de caractéristiques temps-réel à un serveur MMS nécessite donc de bien cerner les différentes fonctionnalités que le serveur doit offrir. Ensuite, il est essentiel de savoir lesquelles de ces fonctionnalités sont soumises à des contraintes de temps.

La distinction entre activités concurrentes d'un serveur que nous avons proposé à la section 4.3 constitue une première approche au problème. Les questions qu'il faut se poser sont alors :

- Le serveur doit-il ou non supporter telle activité concurrente?
- Telle activité concurrente est-elle soumise à des contraintes de temps?

Mais ceci ne suffit pas. L'exécution des services MMS ne se fait pas de façon identique pour tous les services. Certains services sont simples, d'autres nécessitent l'intervention du dispositif physique (services intrusifs), d'autres encore engendrent une séquence de services et l'important est alors de tenir compte du temps pris pour effectuer toute la séquence. Nous réutilisons la classification des services proposée en 4.2 et l'étendons pour tenir compte de tous les cas. Dans chacun de ces cas, nous déterminons les problèmes temps-réel qui peuvent survenir :

1. *Services immédiats* non intrusifs : c'est le cas le plus simple d'exécution d'un service. La tâche qui exécute un tel service est sporadique[2] et ne se bloque pas. Elle n'encourt pas de délai d'attente dû au transfert de son exécution sur le dispositif physique sous-jacent.
2. *Services nécessitant l'interaction entre plusieurs gestionnaires*: ce cas est relativement semblable au précédent. Toutefois le fait qu'un service fasse appel aux opérations internes de plusieurs gestionnaires augmente la possibilité de rencontrer des **problèmes de blocages dus aux ressources partagées**.
3. *Services intrusifs*: les services intrusifs font intervenir la capacité de traitement du dispositif physique. Un service intrusif est le plus souvent un service suspensible. Un serveur supportant les services intrusifs doit donc prendre en compte le **problème des tâches qui suspendent volontairement leur exécution**. Il est possible de considérer le temps passé dans l'exécution d'une routine du dispositif comme faisant partie du temps d'exécution de la tâche. Toutefois ceci peut impliquer une sous-utilisation du processeur du serveur.
4. *Services bloquants*: les services bloquants[3] peuvent s'interrompre en attente de la satisfaction de conditions externes au serveur et indépendantes de ce dernier. Il n'est donc pas possible de borner le temps d'exécution d'un service bloquant si l'on ne connaît pas les caractéristiques temporelles de l'application qui doit débloquer le service. La prise en compte des services bloquants fait apparaître des **problèmes de synchronisation temps-réel entre tâches dans un système réparti**.
5. *Services avec modificateurs*: les services avec modificateur sont des services bloquants. La différence provient du fait que les services bloquant ne se bloquent qu'une seule fois alors que les services avec modificateurs peuvent se bloquer autant de fois qu'il y a de modificateurs. Les problèmes à prendre en compte avec les modificateurs sont les mêmes qu'avec les service bloquants bien qu'encore plus complexes.

[2] Rappelons qu'une tâche est dite *sporadique* si elle n'est pas périodique mais qu'il existe un temps minimum d'inter-arrivées entre deux instances de la tâche. Une tâche pour laquelle un tel temps n'existe pas est dite *apériodique* et un nombre quelconque d'instances de la tâche peut survenir à tout moment dans le système ce qui rend l'ordonnancement des tâches beaucoup plus difficile. Nous verrons plus loin pourquoi nous considérons les requêtes de service MMS comme des tâches sporadiques.

[3] Rappelons que tout service MMS peut être bloquant ou suspensible car MMS ne traite pas de la façon d'implanter une classe de service donnée. Pour chaque serveur donné et pour chaque service supporté par ce serveur, il convient donc aussi de décider de la classe du dit service.

6. *Services asynchrones*: rappelons qu'un client fonctionne en mode asynchrone s'il peut envoyer plusieurs requêtes de service sans attendre la réponse des précédentes. Par extension on dit que le service est asynchrone. Il est possible que des clients envoient les requêtes en mode asynchrone mais exigent que celles-ci soient exécutées dans l'ordre par le serveur[4]. Les services asynchrones font alors apparaître le **problème de contraintes de précédence entre tâches** sur une même association.

7. *Séquences de services*: certains services déclenchent un échange de plusieurs services entre le client et le serveur. Il s'agit essentiellement des services `Initiate Download Sequence` et `Initiate Upload Sequence` pour le téléchargement de domaines[5]. Pour d'autres services, il n'est pas possible d'envoyer tous les résultats voulus en une seule réponse et plusieurs séries de requêtes/réponses sont nécessaires. Le problème est alors de borner le temps mis pour exécuter toute la séquence. Comme ces séquences font intervenir un traitement sur le site client, il n'est pas possible de limiter l'étude temps-réel au seul serveur MMS. Nous nous trouvons ici encore devant un **problème de synchronisation temps-réel entre deux tâches communiquant par échange de messages**.

Enfin, la façon de traiter et d'ordonnancer les requêtes de service MMS dépend également de caractéristiques propres du système telles que la présence ou non de ressources partagées, la nécessité ou non de tenir compte de contraintes de précédence, etc.

Pour déterminer précisément le modèle d'un serveur MMS temps-réel, il faut donc travailler selon trois axes :

1. Activités concurrentes supportées (et contraintes de temps associées).
2. Types de services supportés (et contraintes de temps associées).
3. Caractéristiques propres du système (ressources partagées, contraintes de précédence, etc.).

La définition des types de services supportés, des activités concurrentes utilisées et des contraintes de temps associées à chaque type de service et activité doit nous permettre de formuler une *fiche descriptive* du serveur. Le but ultime est de pouvoir pour chaque fiche descriptive proposer une architecture temps-réel du serveur qui soit la plus simple, la plus efficace et qui satisfasse les besoins du concepteur. Nous ne saurions effectuer ce travail sans y consacrer au moins une thèse entière. Aussi nous restreignons notre analyse d'un serveur temps-réel aux seuls cas suivants :

- Nous ne prenons en compte que certaines des activités concurrentes définies à la section 4.3: l'exécution des services MMS puis dans un second temps la détection des événements. Toutes les autres activités concurrentes n'ont pas de contraintes de temps. En particulier, l'activité de consultation des messages s'exécute en un temps négligeable dès la réception d'une PDU. Ceci provient de notre simplification selon laquelle le temps d'attente des requêtes est nul.
- Nous imposons des contraintes de temps aux services immédiats simples, nécessitant éventuellement l'intervention de plusieurs gestionnaires. Ces services peuvent être intrusifs dans la mesure où leur temps d'exécution sur le dispositif physique est pris en compte dans le temps d'exécution sur le serveur. Tous les autres types de services sont traités sans contraintes de temps.

[4] L'ordre d'exécution n'est pas imposé par la norme MMS mais un serveur donné peut garantir que les requêtes reçues sont exécutées dans l'ordre de réception.

[5] Nous connaissons certaines applications pratiques de MMS qui nécessitent que le téléchargement complet d'un domaine se fasse en temps-réel.

– Dans certaines conditions que nous précisons plus loin, les services bloquants, suspensibles et avec modificateurs peuvent avoir des contraintes de temps.

– Il peut exister des données partagées. Ces données sont des objets MMS ou de l'information propre à chaque implantation d'un serveur.

6.3 Quelles limitations pour MMS ?

6.3.1 Services déterministes

Un service est dit *non déterministe* si la définition qu'en donne la norme MMS autorise dans l'exécution de ce service une série d'actions dont la durée (le temps de traitement) est indéfinie. Le service `Read` par exemple est non déterministe car un client peut toujours décider arbitrairement du nombre de variables à lire, ce nombre n'étant pas limité par la norme[6]. Le service `Get Name List` est également non déterministe car en général on ne connaît pas le nombre d'objets présents dans la VMD et donc le temps total pris pour traiter ce service. Par opposition, le service `Status` par exemple est *déterministe* car les actions qu'il a à effectuer sont fixées. La durée de traitement d'un service déterministe dépend donc uniquement de chaque implantation d'un serveur. La durée de traitement d'un service non déterministe dépend de chaque implantation d'un serveur mais également des applications client MMS.

Pour pouvoir garantir une exécution temps-réel des requêtes de service MMS, il faut pouvoir borner leur temps de traitement et donc connaître le temps de traitement maximum de chaque requête susceptible d'être effectuée par un client. La première contrainte que nous impose un serveur MMS temps-réel est donc de ramener les services non déterministes à des services déterministes. Pour un service comme `Read`, ceci impose de limiter le nombre de variables pouvant être lues en une seule requête. Le format de certaines réponses de services MMS prévoit déjà l'argument `MoreFollows` qui indique au client que tous les résultats de sa requête ne peuvent être inclus dans la PDU réponse et que d'autres réponses suivront si le client le désire [ISO90b]. Ceci n'est toutefois pas systématique et le service `Read` ne dispose pas de cet argument. Par ailleurs, nous avons exclu les séquences de services de notre analyse.

La solution que nous adoptons est de limiter la taille des PDUs MMS sur une association donnée. Une ancienne version de la norme MMS prévoyait de négocier cette taille lors de l'établissement des associations. Le paramètre `negotiated Max Segment Size` était utilisé à cet effet. Dans la norme finale, cette possibilité a été supprimée et le paramètre remplacé par un autre du même type `local Detail Called`. Ce nouveau paramètre est optionnel et sa signification est laissée libre à chaque implantation. Pratiquement, ce nouveau paramètre est toujours utilisé pour négocier la taille des PDUs MMS. Il s'agit d'ailleurs d'une recommandation d'un groupe de travail du NIST ("National Institute of Standards and Technology") qui effectue des proposition pour l'implantation des protocoles OSI [NIS93]. Nous utilisons ici ce paramètre dans la même optique et il y a donc toujours une limite sur la taille de ces PDUs de sorte que tous les services MMS sont déterministes.

Toutefois cette limitation est relativement peu précise car elle ne prend pas en compte la sémantique de la requête. Ainsi une requête de lecture n'est pas limitée par le nombre de variables à lire mais bien par la taille de la PDU correspondante. Le nombre de variables à lire est donc dépendant des paramètres supplémentaires insérés dans la requête et en particulier de la taille des noms de chacune des variables. On peut par exemple lire une seule variable dont le nom est long (la seule variable nommée `Temperature` prend 11 octets) mais plusieurs variables dont les noms sont plus courts (les cinq variables `V1`, `V2`, ..., `V5` prennent 10 octets). Par exemple une taille de 64 octets pour les les PDUs

[6] Pour un nombre fixé de variables, le service `Read` est déterministe.

MMS permet de lire sept variables dans le cas le plus simple c'est-à-dire sans utiliser les paramètres optionnels du service `Read` et en limitant le nom des variables à deux caractères. Ce calcul est fait après encodage ASN.1 BER de la PDU MMS par le compilateur ASNADAC [Ber93].

Nombre de caractères	Taille des PDUs (octets)				
	64	128	256	512	1024
2	7	16	35	71	144
5	5	11	24	50	101
10	3	7	16	33	67
20	2	4	9	20	40

TAB. 6.1 - Nombre de variables pouvant être lues en une seule requête `Read`

Le tableau 6.1 montre dans ce cas simple le nombre de variables qui peuvent être lues en une seule requête de lecture en fonction de la taille des PDUs et de la taille du nom des variables. Pour un nombre constant de caractères, le doublement de la taille des PDUs correspond en gros à un doublement du nombre de variables qui peut être lu. Par contre pour une taille de PDU fixée, la division par 2 du nombre de caractères dans un nom ne double pas le nombre de variables à lire. Ces résultats sont également reportés sur la figure 6.2.

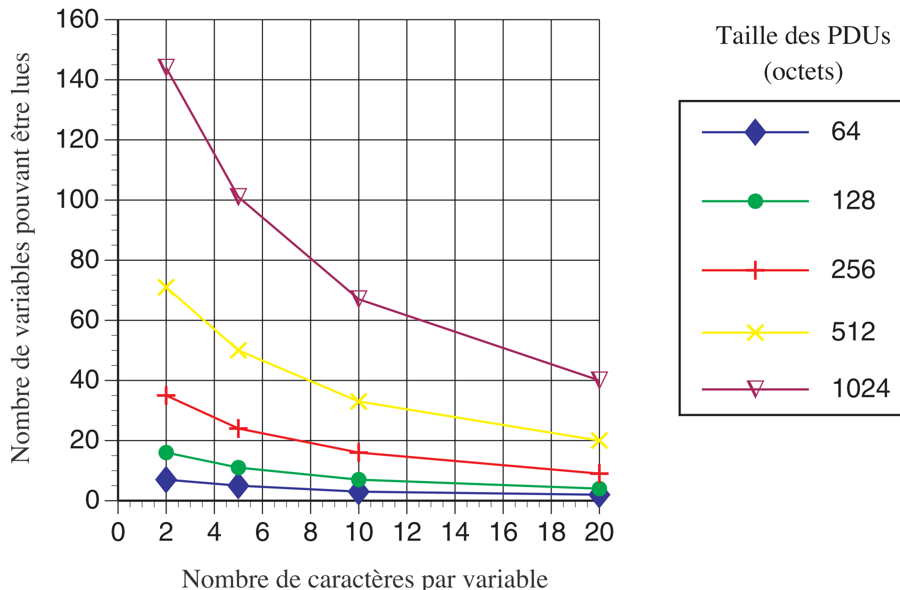


FIG. 6.2 - Nombre de variables pouvant être lues en une seule requête `Read`

La taille maximum des PDUs est une limitation qui s'applique à tous les services MMS sans considérer la sémantique propre à chacun de ces services. Une implantation serveur temps-réel devrait idéalement fournir des limitations propres à chaque classe de requête. Par souci de simplicité, nous n'adoptons pas une telle solution. Notre but essentiel étant d'établir une borne sur le temps d'exécution des requêtes MMS, nous nous contentons d'une limite sur la taille des PDUs MMS.

Cette limitation oblige les applications client à s'intéresser à un problème d'implantation qui normalement n'est pas de leur ressort. Mais si l'on veut pouvoir satisfaire des contraintes de temps, il semble inévitable que les applications soient concernées par de telles contraintes [ISO92c].

6.3.2 Associations d'application

Il n'est pas suffisant de limiter la durée de traitement d'un service MMS. Il faut également pouvoir établir une limite sur le nombre de requêtes de service qu'un serveur doit exécuter à un instant donné.

Cette limitation est effectuée de deux façons complémentaires :

- Limitation du nombre de requêtes simultanées effectuées sur une association : ce nombre est négocié lors de l'établissement de l'association. Il appartient au serveur de fournir une valeur dépendant des caractéristiques de l'implantation et éventuellement d'autres paramètres comme la charge du serveur.
- Limitation du nombre d'associations supportées par un serveur : ce nombre dépend également des caractéristiques du serveur mais aussi de l'architecture de communication sous-jacente.

Ces deux paramètres imposent une limite sur le nombre total de requêtes simultanées qui peuvent exister dans un serveur. Dans la suite, nous notons $max(\mathcal{R})$ cette limite.

En terme d'implantation, ces limitations nous permettent de pré-allouer dans le serveur la totalité des objets transaction nécessaires pour le traitement des requêtes de service MMS. Ceci nous apporte un gain de temps par rapport à une allocation dynamique de la mémoire toujours plus coûteuse.

6.3.3 Détection et traitement des événements

Le nombre d'événements à scruter n'est pas limité par la norme MMS. Il est clair que si l'on veut garantir une détection temps-réel des événements scrutés tout en assurant le fonctionnement des autres activités du serveur, il convient de limiter le nombre d'objets ECs en cours de scrutation à un instant donné.

De même, l'apparition d'un seul événement peut potentiellement donner lieu à un nombre quelconque d'actions événementielles et un nombre quelconque de notifications d'événement. Le nombre d'actions événementielles qui peuvent être exécutées dans un serveur est limité par le nombre maximum d'objets transaction autorisés. Mais ceci ne limite pas le nombre de notifications. Nous imposons donc une limite sur le nombre d'objets EEs pouvant être liés simultanément à un objet EC. Encore une fois, les valeurs de la limite sur le nombre d'ECs scrutés et celle sur le nombre d'EEs par EC sont dépendantes des caractéristiques du système. Ces valeurs peuvent d'ailleurs être dynamiques et changer en fonction de critères comme la charge du serveur.

L'incorporation de XED dans le mécanisme de détection temps-réel des événements MMS nécessite une limitation du nombre d'actions à effectuer par XEDI. Nous avons d'ailleurs vu au chapitre 5 qu'une telle limitation est souhaitable pour assurer la cohérence de la détection dans le temps. L'adoption d'une taille maximum des PDUs MMS limite automatiquement le domaine des valeurs qui peuvent être affectées à une Variable Condition par le service `Write`, et donc limite également la durée du processus de détection effectué par XEDI. Notons toutefois qu'il est beaucoup plus probable que cette limitation provienne de la définition même de la variable. En effet, la Variable Condition ne pourra prendre des valeurs plus étendues que la taille maximum de la chaîne de caractères qu'elle peut contenir.

6.4 Systèmes à contraintes de temps strictes

Dans un système à contraintes de temps strictes, aucune échéance ne peut être manquée sous peine de conséquences graves voire catastrophiques. Dans le cas d'un serveur MMS, il s'agit de garantir que toutes les requêtes envoyées sur une association sont toujours exécutées dans les temps impartis.

Pour obtenir une telle garantie, les ressources nécessaires au traitement des requêtes (mémoire, processeur, etc) doivent normalement être réservées au moment de l'établissement de l'association entre un client et le serveur. Si ces ressources sont disponibles, l'association peut être établie et l'application client est assurée que toutes les requêtes envoyées sont exécutées dans les temps. Quand le serveur ne peut offrir une telle garantie, il doit refuser l'établissement de l'association.

6.4.1 Paramètres essentiels

Pour garantir que toute requête envoyée sur une association soit exécutée dans les temps impartis, le serveur doit connaître lors de la phase d'établissement d'une association les trois paramètres suivants :

1. Le **temps de traitement maximum** : c'est le temps maximum requis par toute requête de service issue sur cette association pour être exécutée par le serveur en l'absence de toute autre tâche. Ce temps (noté C_i^{max} pour l'association i) est obtenu lors de la demande de connexion quand la VMD décide des requêtes de service que le client peut envoyer sur l'association. Le serveur peut alors connaître le service qui nécessite potentiellement le plus long traitement. Un serveur est libre de ne pas autoriser l'utilisation d'un type donné de service s'il s'estime incapable d'assurer les contraintes de temps autrement. En particulier, il n'est pas possible d'autoriser les services bloquants dans la mesure où leur temps d'exécution est inconnu. Pour les mêmes raisons, l'utilisation de modificateurs dans les requêtes de service devrait être proscrite. Pour tous les autres services, le pire temps de traitement peut et doit être connu du serveur[7].

2. Le **temps minimum d'inter-arrivées** : c'est le temps minimum qui doit s'écouler entre l'arrivée au serveur de deux requêtes successives sur l'association considérée. Ce temps dépend de l'application client qui connaît la fréquence avec laquelle elle envoie ses requêtes de service. Il n'est cependant pas possible d'insérer ce temps dans la PDU `Initiate` sans modifier le protocole MMS et donc sans introduire une modification de niveau 4 au moins.

   La solution est d'adopter le temps d'inter-arrivées le plus défavorable au serveur c'est-à-dire le meilleur temps de transfert d'une PDU MMS du client vers le serveur[8]. Ce temps doit être connu au préalable par le serveur et dépend des caractéristiques de l'architecture de communication. Nous notons T_i^{min} le temps minimum d'inter-arrivées sur l'association i. Une méthode très simple pour maximiser T_i^{min} est d'imposer un fonctionnement synchrone en n'autorisant qu'une seule requête en cours sur l'association i. Dans ce cas T_i^{min} devient au moins égal à deux fois le meilleur temps de transfert d'une PDU MMS entre les deux sites.

   Plus généralement, le serveur peut moduler le nombre de requêtes simultanées envoyées par le client au moyen du paramètre `negociated Max Serv Outstanding Called` (noté Req_i^{max} pour l'association i). Quand une association i est établie, le client dispose d'une fenêtre de Req_i^{max} requêtes de service. Il ne peut envoyer plus de Req_i^{max} requêtes non confirmées. Le serveur peut donc connaître la charge maximum Γ_i^{max} que peut générer l'association i :

$$\Gamma_i^{max} = Req_i^{max} \frac{C_i^{max}}{T_i^{min}} \tag{6.1}$$

3. L'**échéance minimum** : c'est la plus petite échéance susceptible d'être associée à une requête de service sur l'association considérée (notée D_i^{min}). Là encore, il n'existe aucune façon de connaître

[7] Nous ne considérons pas le cas de pannes du serveur ou du dispositif physique sous-jacent qui engendreraient des délais ou blocages anormaux.

[8] On fait ici l'hypothèse qu'un client ne peut envoyer des requêtes de service de façon groupée mais seulement l'une après l'autre.

ou négocier l'échéance minimum lors de l'établissement de l'association sans modifier les PDUs requête et réponse du service `Initiate`. Les clients et serveurs auraient alors un niveau de modification de 4 ou 5. Dans les sections suivantes, nous faisons en sorte qu'une échéance puisse être dynamiquement associée à toute requête de service MMS. Même s'il était possible d'associer à la connexion client/serveur une échéance minimum, les tests de vérification du respect de cette échéance nécessiteraient de modifier la MMPM. Pour rester à un niveau de modification égal à 2, les deux seules solutions acceptables sont :

- Fixer statiquement pour un serveur la valeur minimum des échéances. Cette valeur est connue a priori de tous les clients. Si un client envoie une requête avec une échéance plus faible, celle-ci est ramenée automatiquement à la valeur statique.
- Supposer que l'échéance des requêtes de service correspond au temps minimum d'inter-arrivées.

Dans tous les cas, la valeur de D_i^{min} doit être connue au préalable pour établir la connexion.

La connaissance de ces trois paramètres et des caractéristiques des associations déjà établies permet au serveur d'effectuer un test d'ordonnancement qui détermine si l'association pourra garantir les contraintes de temps annoncées. Nous en discutons à la section 6.4.5. Toutefois, nous avons vu qu'il n'est pas possible de communiquer à un serveur les valeurs T_i^{min} et D_i^{min} lors de la phase d'établissement de l'association sans modifier les PDUs requête et réponse du service `Initiate`. Nous proposons donc une méthode alternative à la section suivante.

6.4.2 Méthode de négociation des paramètres temporels

Le principe de cette méthode est d'associer deux variables MMS à toute connexion. L'établissement de l'association temps-réel se fait alors en deux phases :

Première phase : Le client établit une connexion avec un serveur au moyen du service `Initiate`. Le serveur crée alors automatiquement deux variables de visibilité `aa-specific` (donc accessibles seulement par ce client sur cette association). Ces variables sont de type entier `Unsigned32` et sont initialisées à zéro. Elles représentent les temps T_i^{min} et D_i^{min}. Conformément à la norme MMS, ces temps sont donnés en millisecondes. La plage de variation est alors de 1 ms à environ 50 jours. Les noms de chacune des variables sont identiques pour toutes les associations et doivent être connus des clients désirant établir une association temps-réel.

Deuxième phase : Une fois la connexion établie, il appartient au client de signaler les valeurs T_i^{min} et D_i^{min} qu'il compte utiliser sur cette association. Ceci est effectué au moyen d'une requête de service `Write`. Une tentative d'accès à ces variables entraîne le lancement d'un test d'ordonnancement qui permet de savoir si les paramètres proposés par le client sont supportés. Le serveur ne répond au `Write` qu'après l'exécution du test d'ordonnancement. Si la réponse est positive, les variables sont affectées aux valeurs proposées et le client est assuré que toutes les requêtes effectuées sur cette association verront leurs contraintes de temps satisfaites. Par contre, si le serveur envoie une réponse négative, le client sait que le serveur ne peut pas garantir le respect des contraintes temporelles avec les valeurs T_i^{min} et D_i^{min} proposées. Dans ce cas, les variables ne sont pas mises à jour et le client est libre de proposer de nouvelles valeurs.

Nous utilisons les codes d'erreurs MMS déjà définis pour construire la réponse négative du `Write`. Les erreurs suivantes peuvent être utilisées selon le cas rencontré : `memory-unavailable`, `processor-resource-unavailable` ou encore `capability-unavailable`. Ces erreurs sont toutes de classe `resource`.

Pour des raisons de compatibilité, on ne peut pas interdire à un client de demander l'exécution d'autres services avant que les variables T_i^{min} et D_i^{min} ne soient mises à jour. Aussi, toutes les requêtes de service effectuées avant cette affectation (y compris le service d'écriture de ces deux variables) se font sans contraintes de temps. Le serveur n'a donc aucune contrainte de temps à respecter sur cette association tant qu'il n'a pas envoyé de réponse positive au service d'écriture des deux variables citées.

Il n'y a pas de moyen de forcer un client à respecter les temps proposés sans changer la machine de protocole MMS. Si un client ne respecte pas T_i^{min} et D_i^{min}, alors le serveur n'est pas tenu de chercher à garantir l'exécution dans les temps des requêtes fautives.

Cette extension est de niveau de modification 2 puisqu'elle n'invalide aucune spécification MMS et ne modifie aucunement le protocole. Elle présente le double avantage de permettre de négocier les paramètres temps-réel mais aussi de permettre de les changer au cours d'une même session MMS sur la même association. Grant souligne dans [Gra92] que la possibilité de renégocier les paramètres temporels est une caractéristique souhaitable des systèmes temps-réel. Un client peut même commencer une session en mode normal et continuer plus tard en mode temps-réel. Ceci ne serait pas possible si on ajoutait simplement les paramètres T_i^{min} et D_i^{min} à la PDU requête du service `Initiate`. Dans ce cas, ces paramètres seraient associés à la durée de vie de l'association.

Un des problèmes introduits par cette méthode de négociation est que le serveur ne peut pas interdire l'utilisation des services bloquants et des modificateurs. La négociation des services supportés se fait avec le service `Initiate` donc avant celle des paramètres temps-réel. Toutefois, les applications pratiques sont toujours basées sur la coopération des entités qui les utilisent. Un client qui désire exploiter les capacités temps-réel d'un serveur n'a donc aucun intérêt à utiliser les services bloquants ou les modificateurs.

	MMS classique	`Initiate` **modifié**	**Méthode alternative**
Niveau de modification	1	4	2
Nombre de requêtes nécessaires	1	1	au moins 2
Utilisation de variables MMS	non	non	oui
Garanties temporelles	aucune	dès l'établissement de la connexion	après la seconde phase de l'établissement
Renégociation des valeurs temporelles	-	non	oui
Utilisation des services bloquants et modificateurs	oui	non	oui : première phase non : seconde phase
Nécessite une vérification de la part de la MMPM	-	oui	non
Temps d'établissement de l'association	rapide	moyen	lent

TAB. 6.2 - Comparaison de différente types d'association

L'intégralité de cette phase d'établissement est représentée sur la figure 6.3. Nous résumons ses caractéristiques principales sur la table 6.2. Nous y faisons figurer une comparaison avec une association MMS classique et la solution nécessitant la modification de la PDU `Initiate`.

6.4.3 Extension de l'objet `Association`

Nous avons défini à la section 4.5.3.1 l'objet `Association` qui permet au serveur de maintenir l'information relative à une connexion en cours. Nous étendons ici cette définition avec les paramètres temps-réel nécessaires.

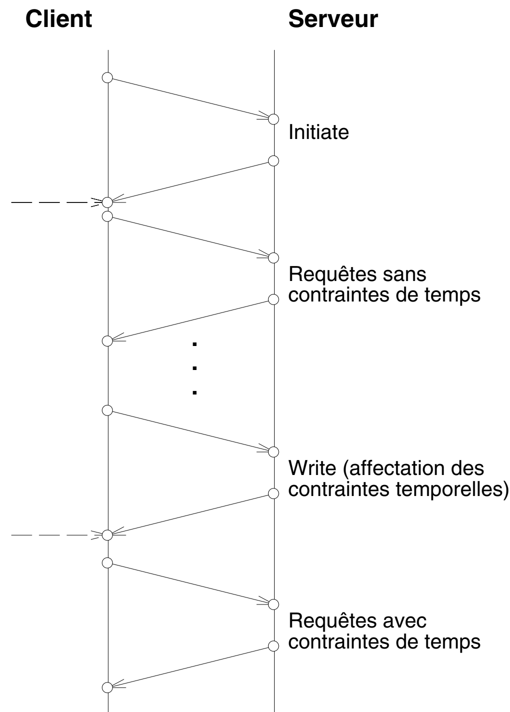


FIG. 6.3 - Négociation des paramètres temps-réel

L'objet `Association` est alors représenté par :

```
Object: Association

  Key Attribute: Application Association Identifier
  Attribute: Last InvokeId
  Attribute: Client Parameters
  Attribute: Server Parameters
  Attribute: Real Time Association
  Constraint: Real Time Association = TRUE
        Attribute: Minimum Inter Arrival Time
        Attribute: Maximum Computation Time
        Attribute: Minimum Deadline
```

Le paramètre `Real Time Association` prend la valeur VRAI pour les associations temps-réel, FAUX sinon. Lorsque la méthode de négociation décrite à la section précédente est utilisée, `Real Time Association` devient vrai dès que les deux variables T_i^{min} et D_i^{min} sont affectées et que la VMD passe les tests d'ordonnancement avec succès. Les trois paramètres suivants représentent respectivement T_i^{min}, C_i^{max} et D_i^{min}. Il sont utilisés par la VMD pour effectuer les tests d'ordonnancement.

6.4.4 Combinaison avec des associations non temps-réel

Bien souvent, des requêtes ayant des contraintes de temps strictes doivent cohabiter avec des requêtes qui n'ont pas de telles contraintes [FV90]. Dans ce cas, un client peut établir deux connexions avec un serveur. La première est dite temps-réel. Elle est réservée pour les requêtes critiques en temps. La seconde est utilisée pour les requêtes qui n'ont pas de contraintes de temps. Le serveur traite

toujours en priorité toutes les requêtes faites sur les connexions temps-réel. Notons que l'établissement d'une association (temps-réel ou non) est considéré comme un service non temps-réel. Un nouveau problème se pose toutefois si le serveur reçoit une requête critique en temps alors qu'il exécute une requête non temps-réel. Deux cas se présentent :

- l'ordonnanceur est préemptif : dans ce cas, la requête non temps-réel en cours peut être interrompue et la requête prioritaire exécutée dès son arrivée au serveur. Dans la phase d'établissement d'une association temps-réel, le traitement des requêtes non temps-réel n'a pas besoin d'être pris en compte. Les associations non temps-réel n'influent pas sur les échéances des requêtes temps-réel et peuvent être établies à tout moment.
- l'ordonnanceur est non-préemptif : dans ce cas, la requête non temps-réel en cours ne peut pas être interrompue. Dans la phase d'établissement d'une association temps-réel, le traitement des requêtes non temps-réel doit être pris en compte. Les tests d'ordonnancement doivent considérer également le traitement de la requête non temps-réel ayant le pire temps de traitement sur toutes les associations. Comme ce temps peut augmenter lorsqu'une nouvelle demande d'association non temps-réel est effectuée, il est possible que le serveur refuse également les demandes d'associations non temps-réel.

La préemption pose le problème de l'interruption des zones critiques notamment lorsque des objets MMS sont partagés entre différentes tâches. Le serveur doit alors connaître la durée d'exécution de la plus longue section critique de code et en tenir compte pour les tests d'établissement des associations de la même façon que dans le cas non-préemptif.

6.4.5 Tests d'ordonnancement

Les tests d'ordonnancement effectués par le serveur lors de l'établissement d'une connexion ou lors de la méthode de négociation définie au paragraphe précédent sont destinés à assurer que toute requête issue sur une association s'exécute toujours dans les temps impartis. Ce sont en fait des tests de faisabilité. La seule façon de garantir que les requêtes ne manquent jamais leurs échéances est d'effectuer les tests d'ordonnancement avec les valeurs les plus défavorables. Il faut donc considérer ces requêtes comme des tâches sporadiques de période T_i^{min}, de temps de traitement C_i^{max} et d'échéance D_i^{min}.

6.4.5.1 Choix de l'algorithme

Les tests d'ordonnancement dépendent de l'algorithme utilisé pour exécuter les requêtes de service MMS. La donnée de l'échéance D_i^{min} et l'arrivée dynamique et imprévisible des tâches suggère l'utilisation de l'algorithme *Earliest Deadline First* (EDF) [LL73]. L'algorithme EDF affecte les priorités des tâches en fonction de leur échéance. La priorité la plus importante est affectée à la tâche ayant l'échéance la plus faible. Quand l'algorithme est préemptif, à tout moment la tâche de plus haute priorité occupe le processeur.

Quand les priorités sont affectées de façon dynamique, l'algorithme EDF est la méthode de choix. Il est particulièrement adapté à notre cas puisque l'on connaît l'échéance de chacune des requêtes MMS et que ces requêtes surviennent à des moments imprévisibles bien que séparés au pire par les temps T_i^{min}. De plus EDF est un algorithme optimal dans le sens où tout ensemble de tâches ordonnançable peut être ordonnancé par EDF [LL73].

L'algorithme EDF est généralement utilisé dans les problèmes d'établissement de connexions temps-réel [FV90], [ZS90]. En particulier, Zheng et Shin proposent des tests d'ordonnancement pour assurer

la transmission de paquets de données sur des connexions temps-réel impliquant plusieurs sites de routage [ZS90]. Les auteurs soulignent que les équations proposées sont suffisamment générales pour être appliquées au cas de l'ordonnancement de tâches. Elles sont en plus nécessaires et suffisantes dans certains cas. Nous résumons ces résultats en les appliquant au cas des associations MMS temps-réel.

6.4.5.2 Equation de base

Il nous paraît inutile d'énumérer et de démontrer tous les tests d'ordonnancement proposés dans la littérature sur les variations de EDF ou d'autres algorithmes. Nous allons simplement montrer quelles sont les équations principales que l'on peut utiliser dans le contexte de MMS pour assurer l'établissement d'une association temps-réel.

Nous supposons disposer de $n-1$ associations temps-réel. Une nouvelle demande d'association temps-réel arrive au serveur. Le problème est d'accepter ou refuser cette nouvelle association.

D'après l'équation donnée dans [ZS90], la nouvelle association peut être établie si et seulement si :

$$\forall t \geq 0, \quad \sum_{i=1}^{n} \lceil \frac{t - D_i^{min}}{T_i^{min}} \rceil_0 \, C_i^{max} \leq t \tag{6.2}$$

La fonction $\lceil\rceil_0$ est définie par $\lceil x \rceil_0 = n$ si $n-1 \leq x < n,\ n = 1, 2, ...$ et $\lceil x \rceil_0 = 0$ si $x < 0$. Le terme somme représente le temps nécessaire pour exécuter toutes les requêtes arrivées dans l'intervalle $[0, t]$ et dont l'échéance ne dépasse pas t.

Etant donné qu'il est impossible de tester tous les instants t pour vérifier cette équation, Zheng et Shin proposent un autre test plus pratique.

Le serveur peut assurer le respect des contraintes de temps de l'ensemble des n associations si et seulement si :

$$\sum_{i=1}^{n} \frac{C_i^{max}}{T_i^{min}} \leq 1 \tag{6.3}$$

et

$$\forall t \in \Omega, \quad \sum_{i=1}^{n} \lceil \frac{t - D_i^{min}}{T_i^{min}} \rceil_0 \, C_i^{max} \leq t \tag{6.4}$$

avec :

- $$\Omega = \bigcup_{i=1}^{n} \Omega_i \tag{6.5}$$

- $$\Omega_i = \{D_i^{min} + kT_i^{min} : k = 0, 1, ..., \lfloor \frac{t_{max} - D_i^{min}}{T_i^{min}} \rfloor\} \tag{6.6}$$

- $$t_{max} = max\{D_1^{min}, ..., D_n^{min}, \frac{\sum_{i=1}^{n} 1 - \frac{D_i^{min}}{T_i^{min}} C_i^{max}}{1 - \sum_{i=1}^{n} \frac{C_i^{max}}{T_i^{min}}}\} \tag{6.7}$$

La complexité de ce test est de l'ordre de la cardinalité de l'ensemble Ω. Zheng et Shin proposent un autre test dans le cas où Ω devient trop grand. Ce nouveau test n'est toutefois que suffisant et n'est pas nécessaire (voir [ZS90]).

6.4.5.3 Prise en compte des ressources partagées

Nous avons vu au chapitre 4 que les objets MMS présents dans un serveur sont susceptibles d'être accédés simultanément par plusieurs tâches (ici plusieurs requêtes de service en cours d'exécution). Nous avions alors choisi d'effectuer un ordonnancement non préemptif des activités concurrentes du serveur pour résoudre le problème des ressources partagées.

Ne pas autoriser la préemption des requêtes est en effet la solution la plus simple pour prendre en compte les ressources partagées. Le problème qui se pose alors est d'assurer le respect des échéances même quand une requête prioritaire surgit alors qu'une autre est en cours d'exécution et ne peut donc être interrompue. Les tests d'ordonnancement nécessitent de prendre en compte le pire des cas c'est-à-dire le cas où le temps de traitement de la requête en cours d'exécution est le plus long parmi les temps de traitement de toutes les requêtes pouvant survenir. Nous notons C^{max} ce temps.

Zheng et Shin proposent également un test dans ce cas :

$$\forall t \geq D, \quad \sum_{i=1}^{n} \lceil \frac{t - D_i^{min}}{T_i^{min}} \rceil_0 \, C_i^{max} + C^{max} \leq t \tag{6.8}$$

où $D = min\{D_i^{min} : 1 \leq i \leq n\}$.

Là encore il est impossible de tester tous les instants t pour vérifier cette équation. Zheng et Shin proposent alors une solution similaire à celle de l'équation 6.4 où seul un ensemble fini de points doit être testé :

$$\sum_{i=1}^{n} \frac{C_i^{max}}{T_i^{min}} \leq 1 \tag{6.9}$$

et

$$\forall t \in \Omega, \quad \sum_{i=1}^{n} \lceil \frac{t - D_i^{min}}{T_i^{min}} \rceil_0 \, C_i^{max} + C^{max} \leq t \tag{6.10}$$

avec :

- $$\Omega = \bigcup_{i=1}^{n} \Omega_i \tag{6.11}$$
- $$\Omega_i = \{D_i^{min} + kT_i^{min} : k = 0, 1, ..., \lfloor \frac{t_{max} - D_i^{min}}{T_i^{min}} \rfloor\} \tag{6.12}$$
- $$t_{max} = max\{D_1^{min}, ..., D_n^{min}, C^{max} + \frac{\sum_{i=1}^{n} 1 - \frac{D_i^{min}}{T_i^{min}} C_i^{max}}{1 - \sum_{i=1}^{n} \frac{C_i^{max}}{T_i^{min}}}\} \tag{6.13}$$

Notons que C^{max} peut aussi être le temps de traitement d'une requête effectuée sur une association non temps-réel. On peut ainsi combiner l'existence d'associations ayant ou non des contraintes de temps comme nous l'avons indiqué à la section 6.4.4.

6.4.5.4 Calcul de l'échéance minimum

L'intérêt d'utiliser EDF ainsi que les résultats de [ZS90] réside aussi dans la possibilité de résoudre le problème suivant : étant donné $n-1$ associations temps-réel déjà établies avec un serveur MMS, calculer lors de la demande d'établissement d'une n^{eme} association la valeur minimale acceptable de D_n^{min} connaissant T_n^{min} et C_n^{max}.

Nous renvoyons le lecteur à [ZS90] pour prendre connaissance des équations résolvant ce problème. L'intérêt dans le cas de MMS est qu'un serveur peut facilement et rapidement rejeter une demande de connexion. En effet, un serveur MMS connaît à l'avance le pire temps de traitement maximum de tous les services qu'il supporte. Quand un client fonctionne en mode synchrone, le serveur peut aussi connaître le temps d'inter-arrivées minimum qui est donné par les caractéristiques de la pile de communication. Dans ce cas, la valeur de l'échéance minimum supportée peut aussi être pré-calculée. Le test d'acceptation de la demande de connexion se résume alors à une simple comparaison de deux entiers.

6.4.5.5 Les associations temps-réel strict, trop pessimistes

Nous ne nous étendrons pas plus sur le cas de serveurs MMS à contraintes de temps strictes. A moins de se situer dans des scénarios très particuliers où peu de services MMS sont utilisés et où toutes les requêtes effectuées ont à peu près les mêmes caractéristiques[9], l'utilisation d'un serveur MMS respectant des contraintes de temps strictes implique une sous-utilisation très importante des ressources et notamment du processeur à cause de la prise en compte des paramètres les moins favorables dans les tests.

Le nombre d'associations établies est alors peu élevé par rapport aux capacités réelles du serveur qui risque d'être au repos la plupart du temps. Dans la section 6.5, nous levons l'hypothèse du temps-réel strict et acceptons que certaines requêtes puissent manquer leur échéance.

Exemple :

> Supposons que le serveur ne reçoive que des requêtes `Write` sur une association donnée $i = 1$. Ces requêtes peuvent arriver avec une période de de 20 ms et l'échéance minimum fournie par le client est 5 ms.
>
> D'après nos mesures [CV95], le temps de traitement d'une requête `Write` prend environ 1 ms pour une variable de type entier alors qu'il faut jusqu'à 4 ms pour dix variables de type identique[10].
>
> On a donc : $C^{max} = C_1^{max} = 4$, $T_1^{min} = 20$ et $D_1^{min} = 5$.
>
> Si 75% des requêtes envoyées ne concernent qu'une seule variable et que seuls 25% concernent dix variables, alors la charge réelle de cette association est $\Gamma_1 = 0.75\frac{1}{20} + 0.25\frac{4}{20} = 8.75\%$. La charge utilisée pour les tests d'ordonnancement est $\Gamma_1^{max} = \frac{C_1^{max}}{T_1^{max}} = \frac{4}{20} = 20\%$. Il y a donc une perte de 11.25% pour cette seule association.
>
> D'après les équations 6.9 et 6.10, on peut établir jusqu'à cinq associations aux caractéristiques identiques.
>
> Toutefois si seules des requêtes de service `Write` sur une seule variable sont utilisées alors on pourrait ouvrir jusqu'à dix associations. De même, si les clients n'ont besoin de mettre à jour les valeurs des variables que toutes les 40 ms, alors on peut encore ouvrir jusqu'à dix associations au lieu de cinq.

---

[9] Les temps les plus défavorables communiqués lors de l'établissement des associations doivent être proches des temps réellement observés.

[10] Notons toutefois que ces mesures n'ont pas été effectuées sur un serveur optimisé dans le but d'assurer une exécution temps-réel des requêtes.

6.5 Systèmes à contraintes de temps lâches

Dans les systèmes à contrainte de temps lâches, il est acceptable que certaines requêtes manquent leur échéance. Nous incluons dans cette catégorie les requêtes dont l'exécution est refusée car elle entraînerait un non respect des contraintes de temps. La notion d'association temps-réel disparaît et le serveur ne se préoccupe pas de la provenance d'une requête. Toutes les requêtes sont traitées de façon identique. Le serveur ne connaît les caractéristiques d'une requête (d'une tâche) qu'au moment de son arrivée c'est-à-dire quand elle est retirée de la file d'attente de l'état REÇU. A toute requête i est associé un temps maximum de traitement C_i et une échéance D_i.

Lorsqu'une échéance est dépassée ou quand le serveur prévoit qu'une échéance va ou pourrait être dépassée, plusieurs solutions sont possibles. Nous les classons par ordre de sévérité :

1. **Rejeter la requête avant son exécution.** Il faut donc que le serveur effectue un test de faisabilité lors de la réception de chaque requête. Ce cas est le plus proche du temps-réel strict puisque toutes les requêtes dont l'exécution est commencée sont assurées de voir leur échéance respectée.

2. **Annuler la requête en cours d'exécution.** Ce cas se produit lorsque la requête a été démarrée et n'a toujours pas fini son traitement lorsque son échéance arrive.

3. **Retarder l'exécution de la suite de la requête** pour ne continuer que lorsqu'une plage de temps libre existe.

4. **Continuer l'exécution de la requête** malgré tout et accepter cette situation de surcharge.

Les conséquences de l'incapacité à respecter les contraintes de temps dictent le choix de la solution appropriée. Cette solution dépend donc des caractéristiques et de l'utilisation de l'application serveur considérée. Pour les deux dernières solutions, il peut aussi être intéressant d'informer le client de l'incapacité à satisfaire l'échéance de la requête. Toutes ces solutions sont examinées dans les sections qui suivent.

Jusqu'à maintenant, nous n'avons fait que **rajouter** à MMS des mécanismes garantissant l'exécution temps-réel des requêtes. Dans cette section, nous allons surtout **exploiter** les mécanismes proposés par MMS pour assurer une exécution en temps-réel lâche des requêtes de service MMS. Nous montrons qu'il est possible de faire en sorte que :

- toute requête de service confirmé transporte un paramètre temporel représentant l'échéance de la requête;
- le serveur MMS utilise ce paramètre temporel pour ordonnancer l'exécution des requêtes;
- un client sache si sa requête n'a pu être exécutée dans les temps.

6.5.1 Première approche : test de faisabilité à la réception

6.5.1.1 Principe

La façon la plus immédiate et la plus stricte pour contrôler l'exécution des requêtes de service MMS est d'effectuer un test de faisabilité à la réception de chaque requête. Si le test échoue, la requête est rejetée et passe directement de l'état REÇU à l'état NON-EXISTANT. Dans le cas contraire, la

requête est acceptée et est assurée d'être exécutée dans les temps (figure 6.4). Elle passe alors dans l'état PRÊT.

En cas de rejet, un message d'erreur doit être retourné au client qui a effectué la requête. Il se trouve qu'il est possible de réutiliser certains des messages d'erreur définis par la norme MMS :

- erreur de classe `resource` et de code `processor-resource-unavailable` : cette erreur spécifie que le processeur ne peut être alloué à la requête.
- erreur de classe `service-preempt` et de code `cancel` : cette erreur spécifie que la requête a été annulée pour des raisons propres au serveur.

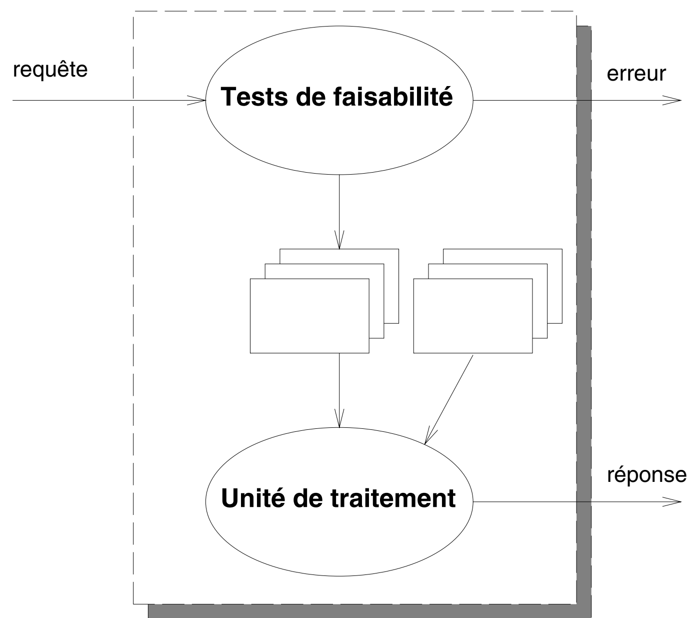


FIG. 6.4 - Approche test de faisabilité à la réception de chaque requête

Pour utiliser cette approche, il faut que le serveur connaisse les valeurs C_i et D_i associées à chaque requête i. Nous supposons que le serveur est capable de déterminer C_i en connaissant la requête i et les paramètres qu'elle transporte. Pour ce qui est de la valeur D_i, nous proposons deux solutions :

1. La première solution a déjà été évoquée à la section 5.3.8. Il s'agit d'utiliser le paramètre `InvokeId` de chaque requête de service pour représenter l'échéance associée. Nous avons vu qu'en utilisant 24 bits de l'`InvokeId` nous pouvions coder des échéances allant jusqu'à plus de 4 heures et demi tout en maintenant à 256 le nombre de requêtes effectuées simultanément sur la même association et ayant une échéance identique. Le serveur peut ainsi connaître les valeurs D_i de chaque requête i qu'il reçoit.
2. La deuxième solution repose sur l'utilisation du modificateur `Attach To Semaphore`. Nous supposons que le serveur contient un sémaphore banalisé S avec n jetons. La valeur de n est telle que $n \geq max(\mathcal{R})$. Toute requête ayant une contrainte de temps est envoyée au serveur avec le modificateur `Attach To Semaphore` référençant le sémaphore S et utilisant le paramètre `Acceptable Delay`. Ce paramètre prend la valeur D_i de la requête. Dès l'arrivée de la requête, le serveur lit la valeur `Acceptable Delay` associée et peut ainsi effectuer le test de faisabilité. Si la requête est acceptée, comme $n \geq max(\mathcal{R})$, il se trouve toujours un jeton libre pour cette requête qui ne reste donc jamais bloquée sur le sémaphore S.

Le paramètre `Acceptable Delay` ne sert donc ici que de moyen de transport de l'échéance. Il n'est fait aucun usage des fonctions associées à ce paramètre.

Ces deux solutions sont de niveau de modification 2. La première solution est toutefois plus élégante que la seconde. Nous avons volontairement proposé la deuxième solution et utilisé un sémaphore MMS pour introduire les deux approches des sections 6.5.2 et 6.5.3 qui sont entièrement basées sur l'utilisation du modificateur `Attach To Semaphore`.

Les modificateurs MMS offrent effectivement beaucoup plus qu'un simple moyen de transport de l'échéance d'une requête. Ils peuvent être utilisés pour ordonnancer l'exécution des requêtes, pour annuler ou rejeter les requêtes qui ne peuvent être exécutées dans les temps ou encore pour effectuer des actions correctrices dans le cas d'échéances non satisfaites. Nous pouvons ainsi exploiter les mécanismes de MMS pour proposer des approches alternatives. Les deux approches suivantes (sections 6.5.2 et 6.5.3) sont optimistes dans le sens où le serveur accepte toujours une requête. Si l'échéance de cette requête ne peut être respectée alors diverses actions définies par MMS sont effectuées.

L'algorithme utilisé pour effectuer ces tests de faisabilité dépend des caractéristiques du système. Il semble toutefois qu'un algorithme EDF soit une solution satisfaisante pour les mêmes raisons que celles qui nous ont conduit à ce choix pour les associations temps-réel : les requêtes surviennent à des moments imprévisibles, l'échéance et le temps de traitement ne sont connus qu'à la réception de la requête, l'affectation des priorités doit se faire de façon dynamique.

6.5.2 Seconde approche : utilisation du temps de contrôle d'un sémaphore MMS

Le temps d'exécution d'une requête dans le serveur peut être contrôlé au moyen du paramètre `Control Timeout` d'un modificateur appliqué au sémaphore S. La valeur fournie à ce paramètre par le client est l'échéance D_i de la requête. Nous supposons que $n \geq max(\mathcal{R})$. Le nombre de jetons du sémaphore est donc au moins égal au nombre maximum de requêtes que le serveur peut traiter "en même temps". Dans ce cas, chaque nouvelle requête dispose toujours d'un jeton libre. Quand la requête arrive au serveur, le modificateur est traité et le temporisateur associé au paramètre `Control Timeout` est démarré. Si la requête est exécutée avant son échéance, le jeton du sémaphore est automatiquement libéré et le temporisateur est désactivé.

Par contre, si le temporisateur expire, alors l'échéance de la requête est dépassée. Le comportement exact du serveur dans ce cas dépend des caractéristiques de l'application, de l'importance de la requête et de la politique choisie pour traiter les échéances dépassées. Le serveur peut :

- Annuler la requête en cours d'exécution et envoyer une erreur de classe `service-preempt` et de code `timeout`.
  - **Avantage** : Il n'y a aucune influence sur les autres requêtes du serveur.
  - **Inconvénient** : Il surgit un problème de cohérence dû à l'annulation de la requête. En particulier, cette solution semble difficilement applicable dans le cas de requêtes intrusives puisque le service MMS peut alors provoquer une action physique dans l'équipement industriel.
- Laisser la requête s'exécuter normalement.
  - **Avantage** : Il n'y pas d'annulation de la requête donc pas de problème de cohérence.
  - **Inconvénient** : Ceci peut provoquer un dépassement d'échéances pour d'autres requêtes.

– Retarder l'exécution de la requête à un moment où le serveur dispose de temps libre.

  – **Avantage** : Pas de problème de cohérence. Pas d'influence sur l'exécution des autres requêtes.
  – **Inconvénient** : Le temps de terminaison de la requête (qui initialement avait une échéance) est augmenté et devient même inconnu.

Les actions entreprises pour corriger le dépassement d'une échéance peuvent être aussi radicales que la rupture de l'association (service `Abort`). Ceci est déclenché automatiquement par le serveur MMS si l'attribut `Abort On Timeout` du modificateur est VRAI. Un tel comportement peut être nécessaire quand l'incapacité d'une requête à s'exécuter avant son échéance invalide l'exécution des requêtes successives effectuées sur la même association. La rupture de l'association annule toutes les requêtes en attente ainsi que la requête en cours d'exécution.

Parfois, l'incapacité à satisfaire l'échéance n'invalide pas la suite du déroulement de l'application. On attend alors un comportement moins radical de la part du serveur. Si le paramètre `Abort On Timeout` est FAUX alors il n'y a pas de rupture de l'association. Par contre la rubrique de sémaphore concernée passe dans l'état `HUNG` (suspendu). La norme MMS laisse une grande liberté aux concepteurs pour décider de ce qu'il faut faire des rubriques de sémaphore suspendues. Dans notre cas, si la requête est annulée, le jeton détenu est automatiquement remis à disposition des autres requêtes de service et la rubrique de sémaphore associée est détruite.

Une autre caractéristique intéressante de MMS provient de la possibilité de déclencher automatiquement un événement quand une fin de temporisation `Control Timeout` se produit. Cet événement peut ainsi être utilisé pour informer le client que sa requête n'a pas été exécutée dans les temps. Cette solution devrait être adoptée dans le cas où le serveur laisse la requête poursuivre son exécution ou au contraire la retarde. La notification d'événement informe alors le client que la réponse qu'il attend n'arrivera pas dans les temps. Le client peut alors effectuer des actions de correction telles qu'annuler la requête au moyen du service `Cancel` ou tout autre action en rapport avec l'application en cours. Notons qu'il est aussi possible d'informer simultanément d'autres clients du dépassement de l'échéance.

Grant souligne dans [Gra92] que les utilisateurs doivent être informés du succès ou de l'échec d'une requête à s'exécuter avant son échéance. La solution basée sur la temporisation `Control Timeout` proposée ici satisfait donc totalement cette exigence : une réponse positive identifie un succès alors qu'une réponse négative ou une notification d'événement identifie un échec. C'est une solution efficace pour associer une échéance à une requête de service MMS. Il n'est pas besoin d'ajouter des paramètres temporels supplémentaires à la requête. De plus MMS offre déjà des solutions de rétablissement en cas d'échec et celles-ci peuvent être choisies par les clients requête par requête.

Notons que nous n'utilisons pas le sémaphore S de façon classique puisqu'il n'y a jamais aucune requête qui reste en attente de la libération de S. Ce comportement est toutefois totalement conforme à MMS. Nous proposons dans la section suivante une méthode différente où les requêtes peuvent rester bloquées sur S.

6.5.3 Troisième approche : utilisation du temps d'attente d'un sémaphore MMS

Nous supposons maintenant que $n < max(\mathcal{R})$. Le nombre de jetons du sémaphore représente donc le nombre de requêtes "simultanément en cours d'exécution". Nous précisons ce que nous entendons par "simultanément en cours d'exécution" plus loin. Comme précédemment chaque requête de service qui désire utiliser les capacités temps-réel du serveur doit contenir un modificateur `Attach To Semphore` référençant le sémaphore S. Quand tous les jetons de S sont pris, les requêtes en arrivée restent

bloquées et attendent d'être exécutées. Ainsi, une requête arrivant au serveur est :

- mise en attente si tous les n jetons de S sont occupés (la requête passe dans l'état MODIFIÉ);
- exécutée dans le cas contraire (la requête passe dans l'état PRÊT puis EN-COURS).

Le modificateur de chaque requête de service doit utiliser le paramètre `Acceptable Delay`. Rappelons que ce paramètre représente le temps maximum que la requête peut attendre avant la libération d'un jeton du sémaphore. C'est donc aussi le temps d'attente avant que la requête soit sélectionnée pour exécution. Si tous les jetons sont pris alors un temporisateur est démarré avec la valeur transportée par le paramètre `Acceptable Delay`. Si la requête ne peut être exécutée avant l'expiration du temporisateur, alors cette requête est automatiquement annulée. Dans ce cas, le serveur envoie au client correspondant une réponse négative qui indique que la requête de service n'a pas été exécutée à cause d'une fin de temporisation. L'erreur de classe `service-preempt` associée au code `timeout` est utilisée à cet effet. Le client est ainsi informé que sa requête n'a pu être exécutée à temps.

Si par contre un jeton du sémaphore est alloué à cette requête, alors le temporisateur est désactivé et la requête exécutée. Ce jeton est automatiquement libéré à la fin de l'exécution de la requête et une autre requête en attente peut ainsi poursuivre. Toute requête reste dans l'état MODIFIÉ jusqu'à ce qu'elle soit sélectionnée pour exécution. La file d'attente de l'état MODIFIÉ représente donc exactement la liste des requêtes bloquées sur le sémaphore S.

Cette approche basée sur le paramètre `Acceptable Delay` est illustrée sur la figure 6.5. **Elle assure que si une requête de service MMS ne démarre pas son exécution au bout d'un intervalle de temps donné alors elle ne sera jamais exécutée**. Il doit être clair que refuser l'exécution d'une requête quand un modificateur ne peut être satisfait fait partie du comportement de MMS. Il n'y a pas besoin de rajouter de nouveaux paramètres ou de nouvelles fonctionnalités au serveur. Cette approche est donc de niveau de modification 1.

Notre but initial était d'associer une échéance à une requête de service MMS pour chercher à borner le temps qu'une requête passe dans le serveur. Si l'on suppose que le client connaît le plus petit temps de traitement possible d'une requête i dans le serveur (noté C_i^{min}), il peut affecter la valeur $D_i - C_i^{min}$ au paramètre `Acceptable Delay`. Ainsi, si la requête n'a pas débuté son exécution avant $D_i - C_i^{min}$, il n'y aucune chance pour qu'elle se termine d'ici D_i. $D_i - C_i^{min}$ est appelé le *point de non retour*. Toutefois l'adoption de la valeur C_i^{min} ne permet pas de garantir qu'une fois démarrée, la requête terminera son exécution avant D_i puisqu'elle peut s'exécuter pendant un temps plus long que C_i^{min}. Il faut alors se baser sur le temps de traitement le plus défavorable de la requête i (noté C_i^{max}) et non pas sur son meilleur temps. Le client affecte donc la valeur $D_i - C_i^{max}$ au paramètre `Acceptable Delay`. Par extension nous appelons toujours ce temps le point de non retour.

Cette méthode d'affectation assure que, dans le cas de traitement non préemptif des requêtes, si la requête i débute son exécution au pire à son point de non retour alors elle sera exécutée avant son échéance. Sinon, la requête est annulée et ne démarre jamais. Cette méthode présente toutefois quelques désavantages :

- Elle nécessite un traitement non préemptif des requêtes sinon il n'est pas possible de garantir qu'une requête qui démarre avant son point de non retour est exécutée avant son échéance puisqu'elle peut être interrompue par une autre requête.
- Elle fait une hypothèse pessimiste puisqu'elle invalide les requêtes qui pourraient démarrer leur exécution entre $D_i - C_i^{max}$ et $D_i - C_i^{min}$ et terminer avant D_i. Ceci peut être minimisé si la différence $C_i^{max} - C_i^{min}$ est faible.

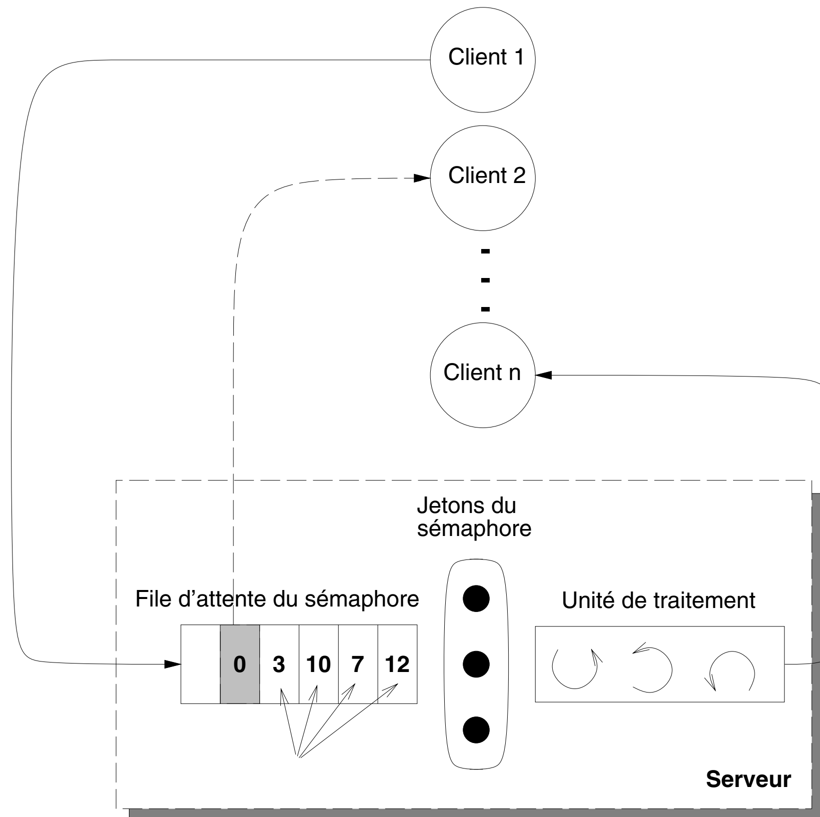


FIG. 6.5 - Utilisation du temps d'attente d'un sémaphore

- Elle impose au client de connaître le pire temps de traitement des requêtes dans le serveur. Ceci risque d'être difficilement réalisable puisque que ce temps change d'un serveur à un autre. La connaissance de C_i^{max} va ainsi à l'encontre du principe d'interopérabilité de MMS.

En général les clients ne connaissent pas le temps que prend le serveur pour exécuter une requête donnée. Toutefois, le rapport [ISO92c] souligne qu'une totale transparence n'est pas possible dans les systèmes temps-réel et qu'il semble que la connaissance des caractéristiques de base de l'implantation d'un système soit nécessaire pour assurer au mieux le respect de contraintes de temps. Il faut par exemple qu'un client connaisse les temps de transferts maximum assurés par la pile de communication pour pouvoir borner le temps de réponse d'une requête de service. Les clients doivent aussi connaître la taille maximum que peuvent avoir les PDUs qu'ils envoient. Il n'est alors pas totalement surprenant qu'un client doive aussi connaître certaines caractéristiques des serveurs avec lesquels il travaille (telles que C_i^{max}).

Pour faciliter cette connaissance des caractéristiques d'implantation, les serveurs peuvent inclure ces caractéristiques dans la liste de leurs capacités ou dans des variables MMS. Au moment de l'établissement de la connexion, le serveur envoie cette information au moyen du service `Information Report`. Mais le client peut également obtenir cette même information en utilisant le service `Read` ou `Get Capability List`. En fait, il existe certains produits commerciaux où le client MMS envoie des dizaines de requêtes juste après l'établissement de la connexion dans le but d'obtenir une image locale précise de la VMD [Sie93]. Ce n'est qu'après que l'application en tant que telle peut démarrer. La norme MMS est suffisamment souple pour permettre d'implanter de tels scénarios ayant pour but d'obtenir les données temps-réel nécessaires au fonctionnement du système.

Enfin, notons qu'il pourrait être possible de se passer de la connaissance de C_i^{max} en acceptant que le client n'affecte que la valeur D_i au paramètre `Acceptable Delay` et que ce soit le serveur qui change cette valeur de D_i à $D_i - C_i^{max}$ lors de la réception de la requête. Le client n'a donc pas besoin de connaître C_i^{max} et de plus le serveur peut remplacer C_i^{max} par la valeur exacte nécessaire au traitement de la requête et ainsi optimiser l'exécution des requêtes. Cependant cette solution modifie le comportement normalisé des serveurs et est donc de niveau de modification 3.

La figure 6.6 montre le comportement des trois solutions décrites jusqu'ici : tests de faisabilité, paramètre `Control Timeout` et paramètre `Acceptable Delay`. Une flèche en traits pleins identifie une réponse MMS positive et donc un succès. Une flèche en traits pointillés identifie au contraire une réponse MMS négative ou une notification d'événement et donc un échec à satisfaire l'échéance.

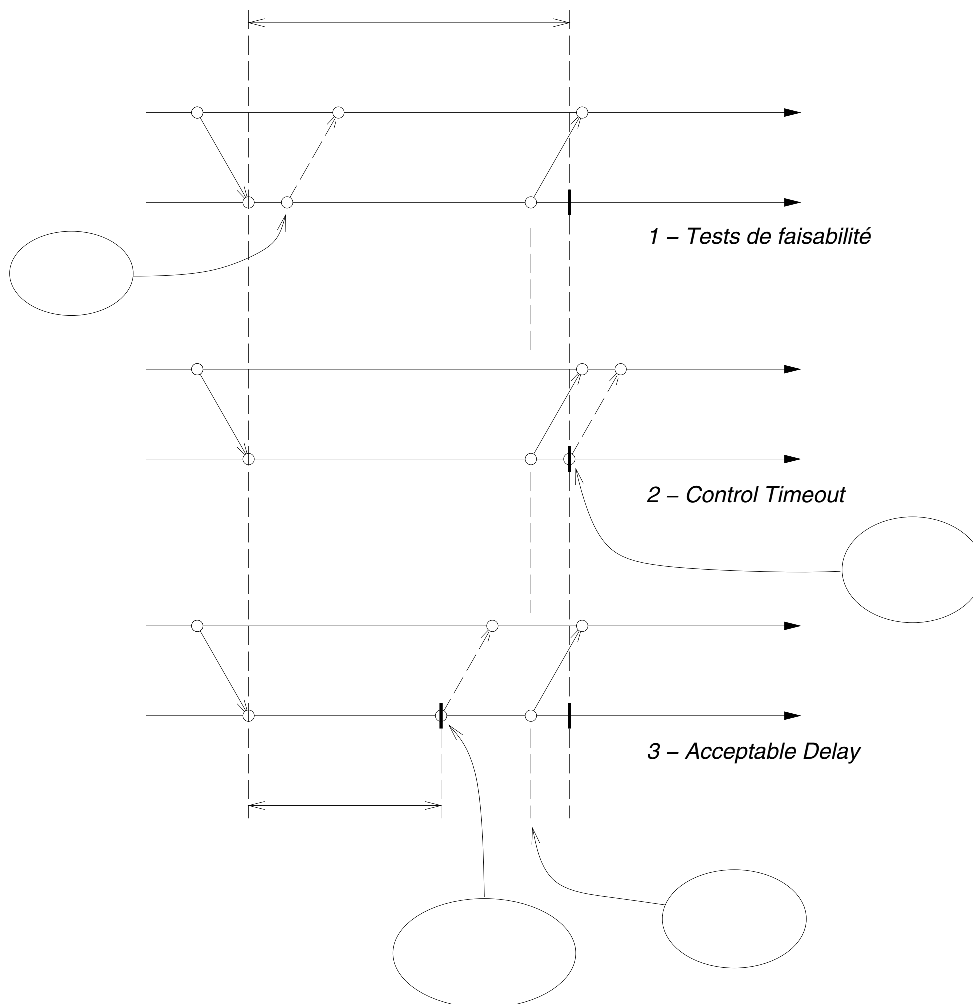


FIG. 6.6 - Comportement des trois approches étudiées

6.5.4 Ordonnancement des requêtes

Jusqu'à présent l'intérêt d'utiliser les paramètres `Acceptable Delay` et `Control Timeout` résidait dans la possibilité d'annuler automatiquement une requête dont l'échéance ne pouvait être satisfaite ou dont l'exécution dépassait un certain temps, ou bien d'informer le client du non respect de l'échéance.

Mais nous souhaitons aussi pouvoir ordonnancer l'exécution des requêtes de service MMS en utili-

sant ces mêmes mécanismes. La technique d'ordonnancement est différente selon que l'on fonctionne en mode préemptif ou non préemptif. Dans le premier cas, la requête en cours d'exécution peut être interrompue par une requête plus prioritaire qui arrive au serveur. Dans le second, la requête en cours d'exécution se termine toujours. Ainsi, à tout moment la requête ayant la priorité la plus haute doit s'exécuter (cas préemptif) ou la prochaine requête qui va s'exécuter doit être celle ayant la priorité la plus haute (cas non préemptif). La priorité de chaque requête est affectée en fonction des échéances transportées par les paramètres `Acceptable Delay` et `Control Timeout`. Nous voyons dans les sections suivantes comment effectuer cette affectation.

6.5.4.1 Ordonnancement basé sur le paramètre `Acceptable Delay`

Cas non préemptif

Nous considérons d'abord le mécanisme MMS associé au paramètre `Acceptable Delay` et examinons le cas non préemptif. En l'absence d'affectation de priorité au modificateur `Attach To Semaphore`, MMS agence les requêtes en attente de libération du sémaphore S suivant un ordre FIFO. Ce n'est donc pas nécessairement la requête dont le temps `Acceptable Delay` restant est le plus faible qui sera la prochaine à être sélectionnée (figure 6.5).

Nous proposons alors une solution naturelle qui est d'ordonnancer les requêtes de service MMS selon un ordre croissant des paramètres `Acceptable Delay` c'est-à-dire à la manière de la politique EDF connue. En fait, du point de vue du client, cet ordonnancement suit l'algorithme *Least Laxity First* (LLF) [MD78] puisque le paramètre `Acceptable Delay` de chaque requête i reçoit la valeur $D_i - C_i^{max}$.

Quand le sémaphore S est libre, le serveur sélectionne et exécute la requête en attente dont la valeur `Acceptable Delay` restante est la plus faible et ce même si d'autres requêtes sont arrivées au serveur avant la requête sélectionnée. Il faut donc que le serveur MMS ordonne la file d'attente du sémaphore S selon l'ordre croissant des paramètres `Acceptable Delay` (voir fig. 6.7). Tout se passe alors comme si chaque modificateur `Attach To Semaphore` contenait une priorité MMS qui dépend directement de son `Acceptable Delay`. Mais il appartient au serveur de déterminer cette priorité. Notons que la valeur réelle de ces priorités n'a pas d'importance. Il importe simplement que l'ordre de la file d'attente du sémaphore respecte l'ordre donné par les valeurs des `Acceptable Delay`. Ceci évite d'avoir à recalculer les priorités MMS de chaque rubrique de sémaphore quand une nouvelle requête arrive et doit être insérée au milieu de la file d'attente.

Le pseudo-code 6.A.1 résume schématiquement les actions effectuées par un serveur MMS utilisant l'ordonnancement basé sur le paramètre `Acceptable Delay`.

Pour déterminer le niveau de modification de cette solution, nous nous heurtons à un problème de clarté de la norme MMS. La norme semble normaliser la politique d'ordonnancement des files d'attente des sémaphores aussi bien dans le cas où les clients ne spécifient pas de priorité (ordre FIFO) que dans le cas où des priorités sont données (ordre suivant celui des priorités). Dans ce cas notre solution présente les désavantages suivants :

- un client ne doit pas spécifier de priorités dans les modificateurs `Attach To Semaphore` appliqués au sémaphores S;
- l'ordre FIFO n'est pas respecté, ce qui modifie le comportement du serveur et rend cette solution de niveau de modification 3.

Mais la norme MMS spécifie aussi que le traitement des priorités par un serveur est un problème local à la VMD et n'est donc pas normalisé ! Dans ce cas, il est possible d'affecter une priorité à la

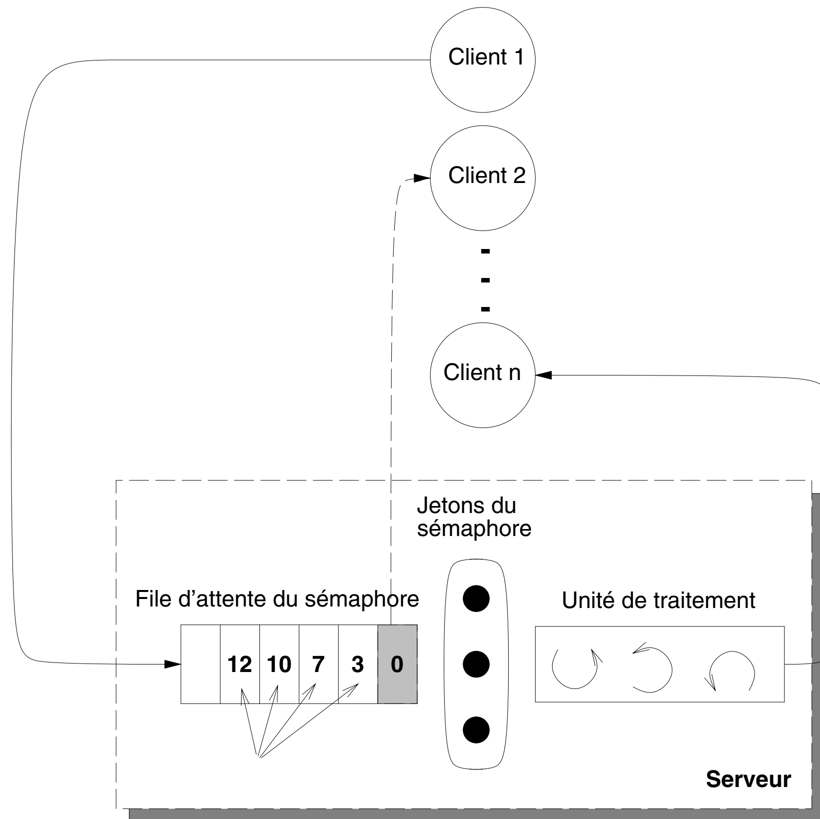


FIG. 6.7 - Utilisation du temps d'attente d'un sémaphore avec ordonnancement

rubrique de sémaphore en attente en fonction de son `Acceptable Delay` sans modifier le comportement des serveurs et le niveau de modification est égal à 2.

Même dans le cas d'un niveau de modification égal à 3, cette modification du comportement des serveurs reste mineure surtout au regard des avantages qu'apporte notre solution. Notons qu'elle se limite au seul sémaphore S utilisé pour notre exécution temps-réel. Par ailleurs, les clients utilisant S le font dans le but d'utiliser les facilités temps-réel du serveur et n'ont donc aucun intérêt à ajouter des priorités à leurs requêtes.

Cas préemptif

Il n'est pas possible d'utiliser un ordonnancement préemptif des requêtes avec une solution basée sur le paramètre `Acceptable Delay` sans devoir rajouter un temporisateur supplémentaire. En effet, quand une requête est sélectionnée pour exécution, un jeton du sémaphore S est pris et le temporisateur `Acceptable Delay` est désactivé. Il ne reste alors plus de mécanisme permettant de vérifier que la requête va ou non manquer son échéance. Dans le cas non préemptif, quand une requête n'est pas annulée avant la prise de contrôle du jeton, elle s'exécute toujours dans les temps puisque que son exécution débute toujours avant $D_i - C_i^{max}$. Ainsi l'exécution de cette requête se termine toujours avant D_i. Dans le cas préemptif il n'y a pas de telle garantie puisque la requête peut être interrompue à tout moment.

Notons que la technique d'ordonnancement basée sur le paramètre `Acceptable Delay` que nous venons de présenter est relativement proche du fonctionnement de l'algorithme D^{over} [KS95]. Cet algorithme ordonnance les tâches selon une politique EDF tant qu'il n'y a pas de surcharge c'est-à-dire tant qu'aucune tâche ne va manquer son échéance. Quand une tâche ayant pour échéance D et

```
 1:  Réception d'une requête req_i

 2:    si un jeton de S est libre alors
 3:      saisir un jeton de S;
 4:      exécuter req_i;
 5:    sinon
 6:      démarrer le temporisateur Acceptable Delay de req_i;
 7:      insérer req_i dans file(état MODIFIÉ) par ordre de priorité;
 8:    fin si;

 9:  Terminaison d'une requête req_i

10:    arrêter le temporisateur Acceptable Delay de req_i;
11:    envoyer la réponse de req_i;
12:    libérer le jeton pris;
13:    si file(état MODIFIÉ) ≠ ∅ alors
14:      retirer la requête req_j en tête de file(état MODIFIÉ);
15:      saisir un jeton de S;
16:      exécuter req_j;
17:    fin si;

18:  Fin de temporisation Acceptable Delay de req_k

19:    retirer req_k de file(état MODIFIÉ);
20:    envoyer une réponse négative pour req_k;
```

Algorithme 6.A.1: Ordonnancement basé sur le paramètre `Acceptable Delay`

temps d'exécution C n'a pas débuté son exécution avant $D - C$, cette tâche peut être annulée. D^{over} n'annule toutefois pas nécessairement cette tâche mais peut choisir d'en annuler une autre selon un critère basé sur le calcul d'un poids pour chacune des tâches.

6.5.4.2 Ordonnancement basé sur le paramètre `Control Timeout`

Notre solution d'ordonnancement des requêtes basé sur le paramètre `Control Timeout` se fait de façon identique à celui décrit à la section précédente. Mais cette fois-ci les requêtes sont toutes dans l'état PRÊT. L'ordonnancement est toujours similaire à l'algorithme EDF et se fait sur les valeurs `Control Timeout`. Du point vue des clients, l'algorithme est également EDF puisque pour chaque requête i la valeur transportée par le paramètre `Control Timeout` est D_i. Le serveur MMS utilise les priorités de la même façon que précédemment.

Contrairement à la solution basée sur le paramètre `Acceptable Delay`, il est maintenant possible de travailler en mode préemptif et d'interrompre la requête en cours si une requête plus prioritaire arrive au serveur[11]. Une requête plus prioritaire est définie comme une requête dont le temps `Control Timeout` initial est plus petit que le temps `Control Timeout` restant de la requête en cours. Les requêtes interrompues retournent dans l'état PRÊT (transition 12 sur la figure 4.5).

Nous illustrons avec le pseudo-code 6.A.2 la technique d'ordonnancement basée sur le paramètre

[11] Nous ne traitons pas du problème de protection des ressources ici. Nous sommes intéressés par décrire ce qu'il est possible de faire avec MMS. S'il existe des ressources partagées on préférera la méthode précédente.

```
 1:  Réception d'une requête req_i

 2:    saisir un jeton de S;
 3:    démarrer le temporisateur Control Timeout de req_i;
 4:    si CT_i < CT_courant alors
 5:      interrompre req_courant;
 6:      insérer req_courant dans file(état PRÊT) par ordre de priorité;
 7:      exécuter req_i;
 8:    sinon
 9:      insérer req_i dans file(état PRÊT) par ordre de priorité;
10:    fin si;

11:  Terminaison d'une requête req_i

12:    arrêter le temporisateur Control Timeout de req_i;
13:    envoyer la réponse;
14:    libérer le jeton pris;
15:    si file(état PRÊT) ≠ ∅ alors
16:      retirer la requête req_j en tête de file(état PRÊT);
17:      exécuter req_j;
18:    fin si;

19:  Fin de temporisation Control Timeout de req_k

20:    retirer req_k de file(état PRÊT);
21:    envoyer une réponse négative pour req_k, une notification d'événement ou
22:     rompre l'association correspondant à req_k;
23:    libérer le jeton pris;
```

Algorithme 6.A.2: Ordonnancement basé sur le paramètre `Control Timeout`

`Control Timeout`. CT_i représente la valeur transportée par le paramètre `Control Timeout` de la requête req_i. $CT_{courant}$ le temps de contrôle restant pour la requête $req_{courant}$ en cours d'exécution.

La table 6.3 résume les résultats et caractéristiques des trois approches étudiées : tests de faisabilité, utilisation du paramètre `Control Timeout` et utilisation du paramètre `Acceptable Delay`.

Notons que l'utilisation du modificateur `Attach To Semaphore` permet à des requêtes temps-réel et non temps-réel de co-exister. En effet, puisque les paramètres `Acceptable Delay` et `Control Timeout` sont optionnels, leur absence (ou simplement l'absence de modificateur) signifie que la requête correspondante n'a pas de contraintes de temps. Le serveur peut l'exécuter lorsqu'il existe des plages de temps libre du processeur.

6.5.5 Politique d'allocation des jetons du sémaphore

Nous avons proposé des solutions basées sur les sémaphores MMS pour contrôler l'exécution des requêtes MMS. L'important est que le mécanisme de sémaphore MMS soit utilisé et se comporte tel que décrit dans la norme MMS et surtout qu'il soit perçu par les clients tel qu'un sémaphore MMS. L'implantation réelle des procédures manipulant le sémaphore S n'est pas du ressort de MMS. En fait, ces procédures doivent être adaptées à nos besoins temps-réel et leur implantation peut être

Critères	Solutions		
	Tests de faisabilité	Acceptable Delays	Control Timeouts
Possibilité d'interrompre la requête en cours	oui	non	oui
Les clients doivent connaître C_i^{max}	non	oui/non	non
Les serveurs doivent connaître C_i^{max}	oui	non/oui	non
Temps nécessaire avant de savoir si l'échéance est manquée (cas non-préemptif)	durée du test	$D_i - C_i^{max}$	D_i
Temps nécessaire avant de savoir si l'échéance est manquée (cas préemptif)	durée du test	-	D_i
Conséquences du non respect d'une échéance	requête rejetée	requête annulée	rupture d'association notification d'événement
Requête possiblement annulée pendant son exécution	non	non	oui
Actions correctrices effectuées par	le client	le client	le serveur et/ou le client
Surcharge temporaire du serveur	non	possible	possible
Nombre initial de jetons du sémaphore	$n \geq max(\mathcal{R})$	$n < max(\mathcal{R})$	$n \geq max(\mathcal{R})$
Le serveur acquière/libère des jetons du sémaphore	non	oui	non
Ordonnancement	quelconque	LLF	EDF
Niveau de modification (sans ordonner les requêtes)	2	1	1
Niveau de modification (avec ordonnancement des requêtes)	2	2 ou 3	2 ou 3

TAB. 6.3 - Comparaison entre les trois approches présentées

complètement différente de celle des procédures manipulant les autres sémaphores du serveur (c'est-à-dire ceux qui ne sont pas utilisés dans le but de contrôler l'exécution des requêtes de service MMS). Ceci est rendu possible par le fait que la norme MMS ne spécifie que ce que les clients doivent percevoir et non comment réaliser les procédures d'un serveur.

Mais il est important pour les concepteurs d'applications serveur de comprendre comment les jetons du sémaphore S sont alloués. Si l'on utilise la solution basée sur les `Control Timeout` , alors il n'y a aucun problème puisque chaque requête acquière toujours un jeton. En effet, le sémaphore contient initialement autant ou plus de jetons libres que le nombre de requêtes simultanées que peut recevoir le serveur ($n \geq max(\mathcal{R})$).

Par contre, dans le cas des `Acceptable Delay`, la méthode d'allocation des jetons est un peu plus complexe. Il est nécessaire de déterminer plus précisément le nombre initial n de jetons du sémaphore S. Nous avons dit que n représente le nombre de requêtes pouvant s'exécuter simultanément sur le serveur.

Si l'on reste strictement dans le cadre de nos deux hypothèses initiales suivantes :

- l'architecture matérielle est mono-processeur;
- les services bloquants et suspensibles ne sont pas pris en compte dans notre analyse temps-réel;

alors n est le nombre de requêtes dans l'état EN-COURS et $n = 1$.

Notons d'abord que si l'on lève l'hypothèse d'une architecture mono-processeur, n est toujours le nombre de requêtes dans l'état EN-COURS mais ne vaut plus 1. Le nombre de jetons de S est alors égal au nombre de processeurs (ceci dans l'hypothèse où chaque requête de service est exécutée sur un processeur).

Nous levons maintenant la deuxième hypothèse ci-dessus et acceptons de traiter les services bloquants et suspensibles avec la méthode `Acceptable Delay`. Pour obtenir une utilisation du processeur acceptable il est nécessaire d'autoriser l'exécution de requêtes de service pendant que d'autres sont bloquées et/ou suspendues. Il faut donc que S dispose de plus d'un jeton puisque les requêtes bloquées et suspendues ne libèrent leur jeton que lorsqu'elles se terminent.

Ainsi, nous pouvons affirmer plus précisément qu'à tout moment le nombre de jetons pris par des requêtes de service doit représenter le nombre de requêtes qui sont sorties de l'état MODIFIÉ sans avoir encore atteint l'état NON-EXISTANT. Le nombre n n'est donc plus simplement le nombre de requêtes dans l'état EN-COURS et doit être supérieur à 1 pour permettre à une requête en attente de prendre un jeton quand d'autres requêtes se trouvent bloquées ou suspendues.

Nous sommes alors confrontés à deux problèmes :

1. Comment allouer et désallouer les jetons ?
2. Quelles valeurs de temps les clients doivent-ils fournir aux requêtes bloquantes et suspensibles ?

Allocation des jetons

La norme MMS précise qu'un serveur peut localement prendre le contrôle d'un sémaphore ou peut localement libérer un sémaphore. Le terme "localement" signifie que le serveur prend la décision d'acquérir ou de libérer un sémaphore de façon indépendante (c'est-à-dire sans intervention d'un client). Un serveur peut donc modifier dynamiquement le nombre de jetons occupés du sémaphore S en fonction de la charge du serveur ou d'autres critères. Nous allons utiliser cette possibilité pour prendre en compte les requêtes bloquées et suspendues. Dans ce cas, les serveurs MMS doivent :

- Acquérir initialement $n - 1$ jetons de S.
- Libérer un jeton quand une requête passe dans l'état BLOQUÉ ou SUSPENDU.
- Acquérir un jeton quand une requête passée dans l'état BLOQUÉ ou SUSPENDU se termine.

Cette façon de faire permet aux requêtes en attente (dans l'état MODIFIÉ) de ne pas rester bloquées jusqu'à ce qu'une requête bloquée ou suspendue se termine.

Affectation des `Acceptable Delay`

Le blocage ou la suspension d'une requête s'apparente un peu à une préemption. Comme nous l'avons vu, l'utilisation du paramètre `Acceptable Delay` se prête mal à la préemption et il est plus souhaitable d'utiliser le paramètre `Control Timeout`. Mais nous nous plaçons ici dans le cas de la technique basée sur l'`Acceptable Delay`.

Pour simplifier le problème et borner le temps d'exécution d'une requête bloquante ou suspensible, nous imposons que les requêtes dans l'état BLOQUÉ ou SUSPENDU soient prioritaires par rapport aux requêtes dans l'état MODIFIÉ. Cela signifie qu'une requête dont la condition de suspension ou de déblocage est satisfaite s'exécute avant toute requête située dans l'état MODIFIÉ. Toujours pour

simplifier, nous imposons un ordre FIFO dans l'exécution des requêtes sortant des états SUSPENDU ou BLOQUÉ[12].

Le temps de traitement d'une requête suspensible ou bloquante se divise en deux phases : avant et après la suspension ou le blocage. Pour garantir qu'une requête i suspensible ou bloquante s'exécute dans les temps impartis ou pas du tout, il faut affecter au paramètre `Acceptable Delay` l'échéance D_i à laquelle on soustrait les temps suivants :

- la durée maximale de la suspension ou du blocage pour tenir compte du temps d'arrêt de la requête;

- le temps de traitement complet de la requête (phase de traitement avant et après blocage ou suspension) comme pour une requête immédiate;

- le pire temps de traitement possible parmi toutes les requêtes MMS acceptées par le serveur. Ceci permet de tenir compte du fait que lors de sa reprise la requête suspendue ou bloquée ne peut interrompre l'exécution de la requête en cours et doit attendre sa terminaison;

- $(max(\mathcal{R})-1)$ fois le pire temps de traitement après blocage ou suspension de toutes les requête bloquantes et suspensibles. Ceci permet de tenir compte du fait que plusieurs requêtes peuvent être bloquées ou suspendues en même temps et que la prochaine à s'exécuter parmi ces requêtes n'est pas nécessairement la requête i. Et il peut potentiellement y avoir $max(\mathcal{R})-1$ autres requêtes bloquées ou suspendues qui s'exécutent avant la requête i.

La connaissance précise de tous ces temps par un client est relativement peu probable et rend cette méthode impraticable dans le cas général et de toutes façons très pessimiste. On peut toutefois se restreindre à des cas plus pratiques dans la mesure où l'on connaît plus de détails sur le système en présence. On peut par exemple n'autoriser que les requêtes suspensibles, limiter le nombre de requêtes suspensibles simultanément acceptées par le serveur, etc.

Remarquons par contre qu'il est relativement peu important d'affecter des échéances aux services bloquants. Nous avons déjà mentionné que ces services ont généralement un temps d'exécution inconnu dans la mesure où les conditions permettant la continuation de cette exécution sont externes au serveur MMS. Il en va de même des requêtes modifiées par un modificateur différent de `Attach To Semaphore` sur S. De plus la plupart des services bloquants contiennent de par leur définition un paramètre permettant au serveur de contrôler le temps d'attente du service dans l'état BLOQUÉ. Ceci fait partie intégrante de l'exécution du service et est indépendant du modificateur `Attach To Semaphore`. Ainsi, nous pouvons raisonnablement considérer que les requêtes allant dans les états BLOQUÉ ou MODIFIÉ (pour une autre raison que le sémaphore S) perdent leur priorités, c'est-à-dire que le serveur n'est plus tenu de satisfaire leur échéance. Ceci justifie notre hypothèse de départ.

6.5.6 Combinaison des `Acceptable Delay` et des `Control Timeout`

L'utilisation simultanée des deux paramètres `Acceptable Delay` et `Control Timeout` permet un contrôle encore plus fin sur l'exécution des requêtes MMS. On peut alors garantir simultanément les deux points suivants :

[12] Ceci n'est pas toujours satisfait en particulier pour les requêtes `Take Control` utilisant des priorités.

- Si une requête ne **démarre** pas son exécution avant un temps donné alors celle-ci est annulée et ne s'exécute jamais. Le client reçoit un message d'erreur spécifiant que sa requête n'a pas pu être exécutée.
- Si une requête ne **termine** pas son exécution avant un temps donné alors le serveur envoie une notification d'événement au client concerné et/ou termine l'association correspondante. La requête peut aussi être annulée si tel est le choix d'implantation du serveur considéré. Ceci ne fait toutefois pas parti des spécifications MMS.

Les clients n'ont plus nécessairement besoin de fournir les valeurs D_i au paramètre `Control Timeout` et $D_i - C_i^{max}$ au paramètre `Acceptable Delay`. En fait, un client peut décider d'affecter n'importe quels temps qui lui semblent utiles. Il suffit qu'il précise le temps maximum d'attente avant l'exécution effective de la requête ainsi que le temps maximum d'exécution après la prise en charge de la requête. Le serveur se charge ensuite du traitement de la requête.

La politique d'allocation des jetons du sémaphore S est ici légèrement différente de celle étudiée jusqu'à présent. Il faut avoir $n < max(\mathcal{R})$ pour que l'utilisation des `Acceptable Delay` ait un sens. Mais il faut aussi que $n > 1$ sans quoi les `Control Timeout` ne présentent que peu d'intérêt. Il appartient alors au serveur de contrôler à tout moment le nombre de jetons mis à disposition des requêtes, un peu à la façon du traitement des requêtes bloquantes et suspensibles présenté à la section 6.5.5. Rappelons que toute requête est dans l'état MODIFIÉ jusqu'à l'obtention d'un jeton. Une requête dans l'état EN-COURS ou PRÊT dispose déjà d'un jeton. La combinaison des deux techniques permet d'obtenir un contrôle sur le temps que passent les requêtes dans l'état MODIFIÉ et sur le temps qu'elles passent dans les états EN-COURS et PRÊT. Il faut en fait décider du critère menant à la préemption de la requête en cours d'exécution et donc à la libération d'un jeton du sémaphore par le serveur. Les requêtes en attente de libération du sémaphore S ne doivent pas avoir une priorité suffisamment grande pour interrompre la requête en cours. Par contre le serveur doit vérifier en fonction de ce critère toute requête en arrivée pour savoir si la requête en cours d'exécution doit être interrompue ou non.

La figure 6.8 montre comment l'utilisation simultanée des deux techniques se traduit au niveau des temps de calcul dans un serveur MMS. Le temps avant la prise en charge de la requête est contrôlé par le paramètre `Acceptable Delay`. Le temps d'exécution effective est contrôlé par le paramètre `Control Timeout`.

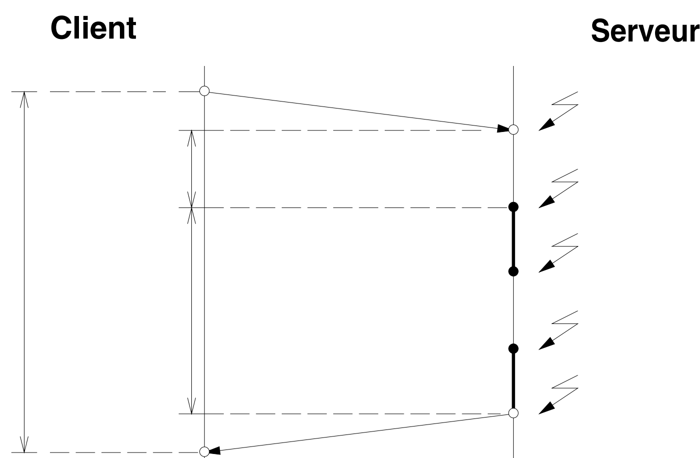


FIG. 6.8 - Utilisation simultanée des deux techniques

La figure 6.9 résume le fonctionnement des techniques combinées `Acceptable Delay` et `Control Timeout` sous forme d'un automate de Nutt [Nut72].

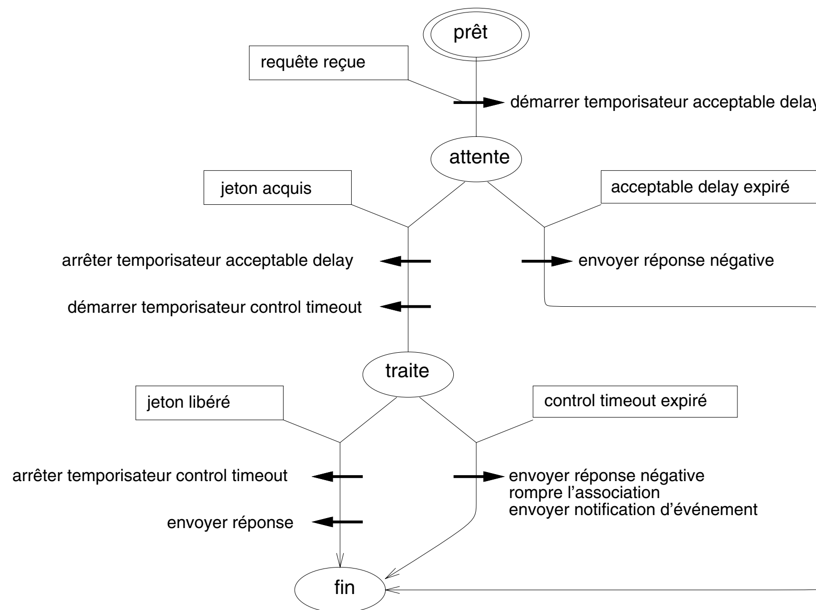


FIG. 6.9 - Automate de Nutt pour le traitement d'une requête

6.5.7 Implantation

L'implantation des algorithmes permettant l'exécution temps-réel des requêtes de service MMS est du ressort des concepteurs de serveurs. Notre but ici était de montrer que MMS fournit certains mécanismes qui autorisent les clients à demander une exécution temps-réel de leurs requêtes de service. Nous allons maintenant donner quelques suggestions quant à l'implantation des serveurs.

Quand un client utilise le paramètre `Acceptable Delay` (resp. `Control Timeout`), il souhaite que le décompte du temporisateur démarre au moment où la requête est insérée dans la file d'attente de l'état REÇU. Toutefois le décompte effectif ne commence que quand la requête est dans l'état MODIFIÉ (resp. PRÊT) et que le sémaphore a été consulté. Ceci est illustré sur la figure 6.10 pour le paramètre `Acceptable Delay` où l'on voit aussi que dans le cas général, le temps `Acceptable Delay` ne représente pas le temps qu'un client souhaite attendre avant que le sémaphore soit libre contrairement à ce qui est souvent affirmé. Par ailleurs, les transitions MODIFIÉ à NON-EXISTANT, PRÊT ou EN-COURS à NON-EXISTANT survenant lors de l'expiration d'un temporisateur impliquent un traitement menant à la construction d'une PDU réponse négative ou à une PDU notification d'événement. Ce traitement n'est bien entendu pas instantané et l'envoi de la PDU au client concerné n'intervient qu'après.

Pour régler ces problèmes, nous avons initialement fait l'hypothèse que les temps de transition entre les différents états d'une requête de service dans un serveur sont négligeables. Nous avons également supposé que le temps d'attente d'une requête dans l'état REÇU est nul. Mais il est clair que pratiquement ces temps ne sont pas nuls et il est donc nécessaire de les minimiser lors de l'implantation du serveur.

Nous proposons deux solutions simples pour les prendre en compte et les minimiser :

1. Lors de l'insertion d'une requête i dans l'état REÇU, la requête est estampillée avec l'heure courante H_i. Quand le serveur traite le modificateur `Attach To Semaphore` associé à la requête i

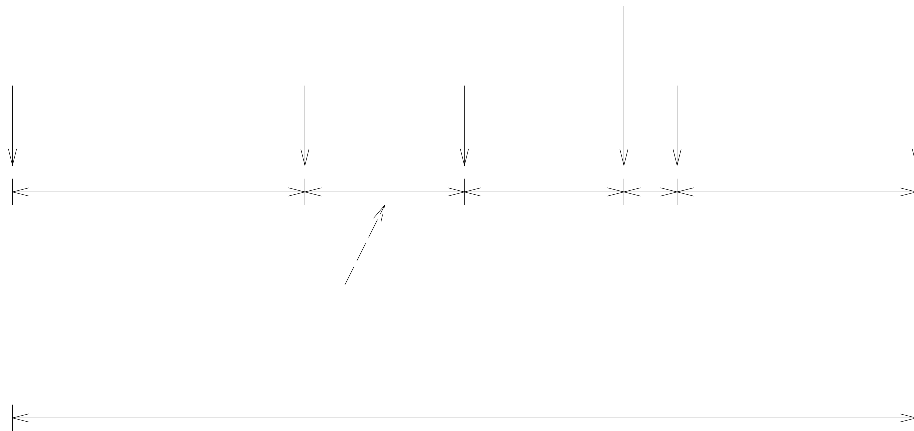


FIG. 6.10 - Temps d'attente du sémaphore et `Acceptable Delay`

et doit démarrer un temporisateur `Acceptable Delay` ou `Control Timeout`, il effectue les actions suivantes :

- Capture du temps courant H;
- Si la valeur V affectée par le client au temporisateur considéré est inférieure à $H - H_i$ alors la requête a déjà manqué son échéance;
- Si la valeur V affectée par le client au temporisateur considéré est supérieure à $H - H_i$ alors le temporisateur est démarré avec la valeur $V - (H - H_i)$.

Il est difficile de dire si cette solution modifie le comportement normalisé des serveurs ou au contraire s'en rapproche avec une plus grande précision. Dans le premier cas elle est de niveau de modification 3, dans le second de niveau de modification 1.

2. Il faut remarquer que les réponses négatives envoyées lors de l'expiration d'un temporisateur `Acceptable Delay` ou les notifications d'événement envoyées lors de l'expiration d'un temporisateur `Control Timeout` sont relativement identiques. Dans le premier cas, seul l'`InvokeId` de la requête change. Dans le second, plusieurs paramètres peuvent changer mais pratiquement il s'agit essentiellement du nom de l'EE associé au client. On peut donc en grande partie **préparer à l'avance** ces PDUs de sorte qu'il suffise de les déposer dans `QueueEnvois`.

6.6 Détection temps-réel des événements

La détection d'événements MMS est un des rares points où la norme mentionne explicitement que des contraintes de temps doivent être respectées. Ceci provient du fait que les événements MMS représentent souvent des alarmes qui nécessitent donc un traitement privilégié par rapport aux autres tâches du serveur. Dans [Pim90], Pimentel suggère que les événements MMS doivent respecter des contraintes temps-réel mais ne propose pas de solution.

Rappelons que la détection d'un événement se fait par scrutation périodique de l'EC qui représente cet événement et de la variable scrutée correspondante. L'événement est détecté quand l'état de l'EC (attribut `State`) change de valeur. Pour aider le serveur à scruter les événements, les applications

client peuvent fournir un intervalle minimum de scrutation ainsi qu'une priorité aux ECs. Ces deux paramètres (appelés *paramètres de scrutation*) se retrouvent dans les requêtes de service suivantes :

- `Define Event Condition` : le temps de scrutation `Evaluation Interval` et la priorité `Priority` sont ainsi affectés à l'EC lors de sa création.
- `Alter Event Condition Monitoring` : le temps de scrutation `Evaluation Interval` et la priorité `Priority` sont utilisés pour remplacer les valeurs courantes de ces mêmes attributs dans l'objet EC.

La définition d'un EC n'implique pas sa scrutation. Celle-ci n'est activée que lorsque le service `Alter Event Condition Monitoring` est utilisé.

On peut identifier différentes approches à suivre lorsqu'un client demande la scrutation d'un nouvel EC ou qu'il modifie les paramètres de scrutation d'un EC existant :

1. Refuser la scrutation de l'EC ou la modification des paramètres quand ceux-ci conduiraient à un ordonnancement impossible.
   - **Avantages** : les contraintes de temps sont toujours respectées (la scrutation se fait selon des contraintes de temps strictes).
   - **Inconvénients** : quand la scrutation d'un nouvel EC n'est pas acceptée, les clients doivent intervenir pour désactiver la scrutation d'ECs jugés moins importants ou leur affecter des périodes de scrutation plus grandes.
2. Accepter la scrutation de l'EC ou des nouveaux paramètres même s'ils conduisent à un ordonnancement impossible et désactiver automatiquement la scrutation de l'EC (ou des ECs) ayant la plus basse priorité de façon à permettre l'ordonnancement à nouveau.
   - **Avantages** : les contraintes de temps sont toujours respectées (la scrutation se fait selon des contraintes de temps strictes), les ECs les plus importants sont toujours scrutés, MMS permet d'informer les clients concernés par la désactivation des ECs de faible priorité.
   - **Inconvénients** : rejet de certains ECs, les clients doivent souscrire à la transition `ACTIVE-TO-DISABLED` ou `IDLE-TO-DISABLED` pour être informés que la scrutation d'un EC est désactivée.
3. Toujours autoriser la scrutation de l'EC ou le changement donné par les nouveaux paramètres et utiliser les priorités pour ordonner la scrutation lors des éventuelles échéances manquées.
   - **Avantages** : les priorités peuvent fournir une mesure de la gravité de l'échec à détecter un événement donné. Le serveur peut alors chercher à assurer la scrutation des ECs les plus prioritaires.
   - **Inconvénients** : il n'y a aucune garantie que les contraintes de temps soient tenues sauf pour les ECs les plus prioritaires.
4. Toujours autoriser la scrutation de l'EC ou le changement donné par les nouveaux paramètres sans se préoccuper des éventuelles échéances ratées.
   - **Avantages** : la simplicité de la scrutation.
   - **Inconvénients** : il n'y a aucune garantie que les contraintes de temps soient tenues.

Dans la suite, nous supposons que la détection doit toujours se faire avec des contraintes de temps strictes. Nous allons adopter la première approche : lors de la demande d'activation de scrutation d'un EC ou leur du changement de valeurs des paramètres de scrutation (service `Alter Event Condition Monitoring`), le serveur calcule si cette nouvelle scrutation peut être acceptée. Dans l'affirmative, un nouvel ordonnancement est déterminé et démarré à un instant donné. La façon de déterminer cet instant de démarrage est précisée plus loin. Si le serveur est incapable de satisfaire la scrutation avec ces nouvelles données alors le service `Alter Event Condition Monitoring` échoue et le serveur retourne l'erreur de classe `resource` et de code `processor-resource-unavailable` ou une erreur de classe `time-resolution`.

6.6.1 Modèle pour la détection d'événements

Dans cette section, nous n'associons plus de contraintes de temps aux requêtes de service MMS. Nous nous concentrons uniquement sur la détection des événements. Nous traitons donc des ECs de classe `MONITORED` et non `NETWORK-TRIGGERED`.

Il existe principalement deux approches pour détecter un événement :

- *Détection par interruption* : on peut en effet imaginer que tout changement dans la valeur de la variable booléenne associée à un EC entraîne une interruption.
  - **Avantages** : le processeur n'est pas occupé par l'activité de scrutation et peut être entièrement utilisé pour les autres tâches du serveur (sauf au moment des interruptions).
  - **Inconvénients** : cette solution ne prend pas en compte les transitions `ACTIVE-TO -DISABLED`, `IDLE-TO-DISABLED` et `ANY-TO-DELETED`. Elle ne permet pas de tenir compte des contraintes de temps et risque de conduire à une surcharge si plusieurs interruptions se produisent en même temps. Elle rend le serveur MMS dépendant du dispositif physique sous-jacent puisqu'il appartient à ce dernier de déclencher l'interruption. En outre, cette solution ne permet pas d'intégrer XED au processus de détection des événements.
- *Détection par scrutation* : cette approche est préconisée par la norme MMS. Elle consiste simplement à vérifier régulièrement que l'état d'un EC est en accord avec la valeur de la variable qui lui est associée.
  - **Avantages** : toutes les transitions sont prises en compte. On peut ordonner facilement les consultations des variables et ECs au moyen de techniques d'ordonnancement connues. On peut de surcroît intégrer facilement XED au processus de détection des événements.
  - **Inconvénients** : le processeur est occupé par l'activité de scrutation ce qui diminue sa disponibilité pour les autres tâches du serveur.

Compte tenu des inconvénients de l'approche par interruptions, nous adoptons l'approche par scrutation préconisée par la norme MMS. Le modèle retenu est le suivant. Nous associons à chaque EC une tâche chargée de la scrutation (appelée *tâche de scrutation*). Cette tâche est périodique de période l'intervalle de scrutation de l'EC. Les temps de traitement de ces tâches ne sont pas identiques. La tâche est plus ou moins longue selon qu'il y ait ou non un événement qui se produit. L'intégration de XED est une autre cause de disparité des temps de traitement entre tâches. En effet, comme nous l'avons vu à la section 5.2.7, le temps d'évaluation d'une expression prédicat par XEDI varie en fonction de cette expression et du nombre de variables qui y sont référencées. Chaque tâche i chargée de la scrutation d'un EC dispose donc d'une période T_i et d'un temps de traitement (maximum) C_i.

Les actions que doivent exécuter chacune de ces tâches sont les suivantes :

1. Consultation du Gestionnaire d'Objets Evénementiels pour obtenir l'identité de la variable à scruter et l'état de l'EC;
2. Consultation du Gestionnaire de Variables pour obtenir la valeur de la variable à scruter (ou appel à XEDI s'il s'agit d'un EC XED);
3. Comparaison entre l'état de l'EC et la valeur de la variable;
4. Si l'EC n'existe plus ou la variable n'existe plus ou
   l'état de l'EC est `ACTIVE` et la variable (ou l'évaluation de XEDI) est FALSE ou
   l'état de l'EC est `IDLE` et la variable (ou l'évaluation de XEDI) est TRUE

   alors

   (a) Créer un objet Event Notification (EN) (voir section 4.5.6);
   (b) Capturer le temps courant (temps d'apparition de l'événement) et le copier dans l'EN;
   (c) Insérer l'objet EN dans la file d'attente `Event Queue`;

   sinon terminer la tâche de scrutation de cet EC.

Nous avons souligné à la section 5.2.4.2 qu'il est important d'effectuer l'évaluation d'une expression prédicat par XEDI le plus rapidement possible pour éviter une incohérence dans la détection d'un événement. Il n'est cependant pas souhaitable de réduire la période de scrutation T_i d'un EC pour assurer une évaluation plus rapide. Ceci entraînerait une utilisation inutile du processeur. La solution la plus simple est d'associer une échéance $D_i \leq T_i$ à chaque tâche i responsable de la scrutation d'un EC XED. Il n'est pas nécessaire que cette échéance soit donnée par les clients. Il appartient à chaque serveur de déterminer une valeur d'échéance en fonction de l'expression prédicat à analyser. Rappelons qu'un client peut effectuer une reconfiguration de la Variable Condition (voir section 5.2.4.3). Dans ce cas, le serveur peut recalculer l'échéance D_i de la tâche associée. Dans le cas où XED n'est pas utilisé, nous avons simplement $D_i = T_i$.

6.6.2 Algorithmes utilisés

6.6.2.1 Choix de l'algorithme

La donnée **a priori** de l'échéance D_i ainsi que des périodes T_i de chaque tâche de scrutation suggère l'utilisation de l'algorithme *Deadline Monotonic* (DM) [LW82]. L'algorithme DM affecte statiquement les priorités des tâches en fonction de leur échéance. La priorité la plus élevée est affectée à la tâche ayant l'échéance la plus proche.

L'algorithme EDF pourrait également être utilisé. La différence entre DM et EDF se fait au niveau de l'affectation des priorités qui est statique pour DM et dynamique pour EDF. Cela signifie que pour DM il faut connaître à l'avance les caractéristiques des tâches alors que pour EDF celles-ci peuvent être connues uniquement au moment où la tâche arrive. EDF se ramène donc à DM quand on connaît par avance l'échéance, le temps de traitement et la période de toutes les tâches ce qui est le cas ici.

L'algorithme *Rate Monotonic* (RM) de Liu et Layland [LL73] est également très fréquemment utilisé dans les systèmes temps-réel. RM pourrait être adopté pour effectuer la scrutation des ECs de

type classique dans la mesure où $T_i = D_i$. Toutefois, RM introduit un certain nombre de limitations et notamment impose justement que $T_i = D_i$ ce que nous ne souhaitons pas pour les ECs de type XED. Cette limitation n'apparaît pas avec DM où il suffit que $T_i \geq D_i$. Audsley souligne d'ailleurs dans [Aud90] que RM n'est qu'un cas particulier de DM quand la période de chaque tâche est égale à son échéance.

Nous adoptons donc l'algorithme DM pour la scrutation des ECs dans le cas le plus général. Plus précisément, ce choix est motivé par les raisons suivantes :

- L'échéance et la période des tâches sont connues à l'avance. Il est donc possible d'établir à l'avance (avant le démarrage de la scrutation) un ordonnancement des requêtes.
- DM est un algorithme connu pour lequel des tests d'ordonnancement précis existent.
- DM est optimal dans le sens où s'il existe un algorithme statique capable d'ordonnancer les tâches alors DM le peut également. L'optimalité de DM a été établie par Leung *et al.* dans [LW82].
- DM est un algorithme hors-ligne c'est-à-dire qu'il permet ici de trouver un ordonnancement avant de démarrer la scrutation. Il se peut que les tests d'ordonnancement de l'algorithme DM soient longs et retardent le démarrage d'une nouvelle scrutation. Toutefois, dans [McE94] McElhone propose une optimisation des tests d'ordonnancement de l'algorithme DM pour les accélérer et permettre une utilisation en-ligne de DM.
- Il est possible de prendre en compte dans les tests d'ordonnancement le temps de blocage d'une tâche quand celle-ci doit accéder à une ressource partagée protégée par sémaphore. Nous pouvons donc adopter un algorithme préemptif tout en assurant l'intégrité des ressources partagées notamment celle des objets EC et variables.
- Nous avons initialement négligé les temps passés dans l'exécution des routines de l'ordonnanceur. Toutefois, il est également possible de prendre en compte ces temps dans les tests d'ordonnancement de l'algorithme DM. Ceci permet alors d'obtenir une analyse plus proche du système réel. Nous renvoyons le lecteur au travail de Burns *et al.* dans [Bur] et [BTW95].

6.6.2.2 Equation de base

Là encore nous n'allons ni énumérer ni démontrer tous les tests d'ordonnancement existant sur les variations de l'algorithme DM. Nous allons simplement montrer quelles sont les équations principales que l'on peut utiliser dans le contexte de MMS pour assurer une détection temps-réel des événements.

Nous disposons de n ECs à scruter définis dans le serveur et numérotés par ordre de priorité. Ainsi l'EC_1 a la priorité la plus élevée donc l'échéance D_1 la plus faible. Le problème est de savoir si l'ensemble des tâches de scrutation associées à ces ECs est ordonnançable. D'après la relation donnée par Audsley *et al.* dans [ABRW91], pour garantir qu'il existe un ordonnancement des tâches de scrutation qui respecte les contraintes de temps il suffit que :

$$\forall i \ : \ 1 \leq i \leq n, \quad C_i + \sum_{j=1}^{i-1} \lceil \frac{D_i}{T_j} \rceil C_j \ \leq \ D_i \tag{6.14}$$

La relation 6.14 s'obtient facilement en remarquant que pour toute tâche i le terme somme représente l'interférence sur i des tâches plus prioritaires que i. Cette relation se détermine en considérant un *instant critique* où toutes les tâches de scrutation demandent initialement à être exécutées au même moment. Un instant critique constitue donc un moment où la demande d'utilisation du processeur est

la plus forte. Liu et Layland ont montré dans [LL73] que si un ensemble de tâches périodiques est ordonnançable à un instant critique alors il est ordonnançable à tout moment de son exécution.

La relation de base 6.14 s'applique dans le cas le plus simple où l'algorithme est préemptif, les échéances sont inférieures aux périodes et aucune ressource n'est partagée.

La relation 6.14 définit une condition suffisante qui est en général pessimiste. Audsley propose d'autres tests d'ordonnancement dont un plus précis et un autre à la fois nécessaire et suffisant. Nous ne détaillons pas ces tests ici et renvoyons le lecteur à [ABRW91].

6.6.3 Démarrage du nouvel ordonnancement

Nous venons de voir qu'il était possible de déterminer au moyen de l'algorithme DM si la scrutation d'un ensemble donné d'ECs est possible. Ce test doit être effectué chaque fois que le service `Alter Event Condition Monitoring` est exécuté dans le serveur et que le client fournit de nouvelles valeurs aux paramètres de scrutation[13] ou qu'il demande la scrutation d'un nouvel EC. **Le test d'ordonnancement donné par l'équation 6.14 fait donc partie intégrante de l'exécution du service `Alter Event Condition Monitoring`.**

Dans le cas où le test s'exécute avec succès, le problème est de savoir quand débuter la scrutation avec le nouvel ordonnancement calculé. L'instant de démarrage de la nouvelle scrutation n'est pas normalisé par MMS. Nous pouvons donc choisir l'instant le plus approprié. Dans la suite, nous appelons *changement de mode* la modification des paramètres de scrutation ou l'ajout d'une nouvelle tâche de scrutation pour reprendre les termes de [TBW92].

La solution la plus simple et la plus couramment employée dans ces circonstances consiste à attendre la fin du PPCM (Plus Petit Commun Multiple) des périodes des tâches en cours (noté T_{PPCM}) et à ne commencer la nouvelle scrutation qu'à cet instant précis (noté δ_k). Si l'ordonnancement courant a commencé à un instant t, alors $\delta_k = t + kT_{PPCM}$ avec $k = 1, 2, \ldots$. A tout instant δ_k, on est effectivement assuré que les échéances de toutes les tâches sont satisfaites. On peut alors considérer δ_k comme le point zéro de départ d'un nouvel ordonnancement. Comme le nouvel ordonnancement calculé après le changement de mode satisfait aussi les échéances des tâches, les échéances sont globalement satisfaites avant et après δ_k.

Cette solution présente plusieurs désavantages:

- La durée qui s'écoule entre l'acceptation du changement de mode et l'instant δ_k suivant peut être relativement longue. Ceci retarde donc d'autant la prise en compte des nouveaux paramètres ou la scrutation du nouvel EC.
- L'exécution, pendant cet intervalle de temps, d'une autre requête `Alter Event Condition Monitoring` provoquant un changement de mode invalide l'ordonnancement calculé précédemment.
- La norme MMS semble imposer que la réponse du service `Alter Event Condition Monitoring` ne doit pas attendre le démarrage de la scrutation du nouvel EC. **La réception d'une réponse positive par une application client n'assure donc pas que la scrutation de l'EC est déjà active.**

On a donc tout intérêt à prendre en compte le changement de mode le plus tôt possible. Tindell *et al.* proposent dans [TBW92] une solution pour accepter le changement de mode avant le point

[13] Pratiquement le test n'est pas nécessaire quand les paramètres de scrutation fournis sont moins sévères.

δ_k suivant. Les auteurs proposent en fait une analyse pour déterminer si un ensemble de tâches est ordonnançable si l'on effectue un changement de mode à tout moment. Nous n'allons pas détailler les équations assez fastidieuses décrites dans cet article mais simplement extraire et adapter celles qui sont utiles pour notre analyse.

Dans le cas de MMS, deux situations peuvent entraîner un changement de mode. La reconfiguration d'une Variable Condition provoque un changement du temps de traitement et éventuellement de l'échéance associés à la tâche de scrutation. Un service `Alter Event Condition Monitoring` peut provoquer soit l'arrivée d'une seule nouvelle tâche, soit la modification de la période d'une seule tâche mais pas les deux à la fois. En ce sens, la détection d'événements MMS constitue un cas particulier du problème traité dans [TBW92] où un changement de mode peut impliquer l'apparition/disparition simultanée de plusieurs tâches ainsi que la modification des paramètres d'autres tâches.

Le problème est de savoir si toutes les tâches de scrutation satisfont toujours leur échéance quand un changement de mode intervient à un instant donné. Tout d'abord, notons qu'il y a trois types de tâches :

Type 1 : celles qui ne subissent aucun changement. Pour ces tâches, il faut alors assurer que le changement de mode n'entraîne pas le non respect d'une échéance.

Type 2 : celles dont la période, l'échéance et/ou le temps de traitement sont modifiés[14]. On est alors en présence de deux *versions* de telles tâches : une ancienne et une nouvelle. Quelque soit l'instant où le changement de mode est requis, on laisse toujours la tâche de scrutation en cours se terminer. La version modifiée de la tâche ne peut débuter au plus tôt qu'après la fin de la période de l'ancienne version de la tâche en cours. Pour ces tâches, il faut alors assurer que l'échéance des anciennes et nouvelles versions sont respectées.

Type 3 : les nouvelles tâches. Il faut également assurer que leur échéance est respectée.

Le principe de l'analyse est de trouver le temps de réponse[15] le plus défavorable de chacune des tâches en présence et de le comparer à leur échéance. Soit une tâche i de type 1 ou une ancienne version d'une tâche de type 2. Nous notons W_i la fenêtre correspondant au temps de réponse de la tâche i et x le moment relatif au début de cette fenêtre où se produit le changement de mode.

D'après les résultats de Tindell *et al.*, les équations suivantes, ici adaptées à notre cas, permettent de trouver par itération le temps de réponse de la tâche i :

Cas où seule une nouvelle tâche k est introduite

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{W_i^n}{T_j} \rceil C_j + \lceil \frac{W_i^n - x}{T_k} \rceil^+ C_k^{\uparrow i} \tag{6.15}$$

où $hp(i)$ est l'ensemble de toutes les tâches (sauf k) plus prioritaires que i; $C_k^{\uparrow i}$ désigne C_k si la tâche k est plus prioritaire que i et 0 sinon. La fonction $\lceil x \rceil^+$ est identique à $\lceil x \rceil$ mais rend 0 quand $x < 0$.

Le terme somme correspond à l'influence sur la tâche i de toutes les anciennes tâches plus prioritaires que i. Le troisième terme représente quant à lui l'influence sur la tâche i de la tâche k nouvellement ajoutée par le changement de mode dans le cas où cette nouvelle tâche est plus prioritaire que i. Dans le cas contraire, la tâche k n'a pas d'influence sur i.

[14] Une modification de la priorité MMS d'un EC est possible avec le service `Alter Event Condition Monitoring` mais nous n'adoptons pas la priorité MMS pour la scrutation des ECs. Ce changement n'a donc aucun effet sur la scrutation.

[15] Le temps de réponse d'une tâche est défini comme le temps s'écoulant entre le moment où la tâche demande à être exécutée et le moment où son exécution se termine.

Cas où les paramètres d'une tâche k sont modifiés

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{W_i^n}{T_j} \rceil C_j + max \left\{ \begin{array}{l} \lfloor \frac{x}{T_k} \rfloor C_k^{\uparrow i} + \lceil \frac{W_i^n - x}{T_{k'}} \rceil^+ C_{k'}^{\uparrow i} , \\ K_k C_j^{\uparrow i} + \lceil \frac{W_i^n - K_k T_j}{T_{k'}} \rceil^+ C_{k'}^{\uparrow i} \end{array} \right\} \quad (6.16)$$

où $K_k = \lceil \frac{x}{T_k} \rceil$ et k' représente la tâche k avec les nouveaux paramètres.

Le troisième terme représente l'influence sur la tâche i de la tâche k modifiée par le changement de mode dans le cas où cette tâche est plus prioritaire que i que ce soit son ancienne ou sa nouvelle version. Dans le cas contraire, la tâche k n'a pas d'influence sur i.

On peut montrer que les deux équations 6.15 et 6.16 convergent vers une valeur $W_i(x)$ (c'est-à-dire $\exists\ n \mid W_i^{n+1} = W_i^n = W_i(x)$) ou alors qu'il existe une valeur de n telle que $W_i^n > D_i$. Notons r_i le pire temps de réponse de la tâche i parmi toutes les valeurs de x. Pour être assuré que le système est ordonnançable à tout instant x où se produit le changement de mode, il faut encore satisfaire l'équation suivante :

$$D_i \geq r_i = max_{\forall x}[W_i(x)] \quad (6.17)$$

Tindell *et al.* proposent alors une solution pour trouver et ne tester que des valeurs significatives de x. Il reste ensuite le problème qui consiste à calculer le temps de réponse d'une nouvelle version d'une tâche affectée par un changement de mode. Une analyse similaire à celle que nous venons de voir est décrite dans [TBW92].

La solution proposée ici ne s'applique que dans le cas où l'algorithme d'ordonnancement utilisé est préemptif. Tindell *et al.* montrent néanmoins comment prendre en compte les ressources partagées en utilisant PCP à la façon de l'équation 6.20.

Nous convenons que toutes ces techniques peuvent sembler trop "poussées" pour aboutir à une simple détection d'événement. Pour des implantations réduites et performantes qui ne supportent pas XED et où la consultation de l'état d'un EC et d'une variable se chiffre en microsecondes, il est clair que la mise en oeuvre des techniques présentées ici est inutile et trop lourde. Toutefois, il faut rappeler et souligner que de nombreux facteurs tendent à augmenter les temps de traitement des tâches :

- Les temps de lecture des objets ECs et variables varient en fonction d'une application donnée. S'il est nécessaire d'accéder au dispositif physique sous-jacent, ces temps peuvent être longs.
- L'interprétation avec XED d'une expression prédicat augmente considérablement les temps de scrutation d'un EC et peut de plus nécessiter de lire de nombreuses variables MMS lors d'une seule scrutation.
- Dans certaines conditions, il pourrait être possible d'intégrer le traitement des événements à la tâche de scrutation ce qui facilite la gestion des événements et assure en plus qu'il existe une valeur de temps au plus tard après laquelle toutes les notifications d'événements ont été envoyées.

Aussi les solutions basées sur DM que nous exposons ici ont essentiellement pour but d'ouvrir des perspectives et ne sont que des suggestions susceptibles de s'appliquer à certains types de systèmes.

6.6.4 Exécution des autres tâches du serveur

La capacité à assurer la scrutation d'un ensemble d'ECs implique de façon évidente que :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{6.18}$$

Dans le cas contraire, la demande imposée au processeur pour la seule scrutation des événements est supérieure à ce qu'il est possible de faire. Pour permettre l'exécution des autres tâches du serveur, il faut réserver une partie du temps du processeur à cet effet. Il ne faut donc pas appliquer l'équation 6.18 à la lettre puisqu'une charge de 1 implique que le processeur sur lequel tourne l'application serveur ne s'occupe que de la détection des événements. Ceci est pour le moins inutile puisque qu'aucune requête de service ne peut alors être exécutée par le serveur MMS.

Comme nous l'avons souligné à la section 6.3.3, il convient donc d'introduire une limitation sur le nombre d'ECs à scruter ou plutôt sur le temps CPU dédié à la détection des événements[16].

Nous notons L_{ev} cette limitation ($0 < L_{ev} < 1$). Il suffit alors de transformer l'équation 6.18 en :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq L_{ev} \tag{6.19}$$

pour assurer qu'une fraction de temps au plus $1 - L_{ev}$ est réservée à l'exécution des tâches autres que la détection des événements. Lors de l'exécution du service `Alter Event Condition Monitoring`, il faut donc également assurer le respect de l'équation 6.19.

Il existe dans la littérature de très nombreux travaux sur le problème de l'ordonnancement d'un ensemble de tâches périodiques à contraintes de temps strictes en présence de tâches sporadiques généralement à contraintes de temps lâches. On trouvera un bon résumé de ces travaux dans [SSL89] et [GB95]. Nous n'allons pas détailler ces méthodes ici. Nous proposons simplement d'exécuter l'ensemble des autres tâches d'un serveur pendant le temps libre laissé par la scrutation des événements. Selon la méthode utilisée et les priorités affectées à chacune des activités concurrentes et des requêtes de service, on peut ou non garantir le respect d'éventuelles contraintes de temps associées à ces autres tâches.

On peut par exemple choisir d'établir des niveaux de priorités entre les activités concurrentes tel que nous l'avons proposé à la section 4.3.2. On favorise ainsi le traitement des événements puis l'exécution des requêtes de service. Ainsi chaque fois que le processeur n'est pas affecté à une tâche de scrutation, il traite d'abord l'ensemble des événements survenus, puis l'ensemble des requêtes MMS arrivées, etc. Ceci se fait dans la limite du temps disponible pour ces tâches.

6.6.5 Prise en compte des ressources partagées

L'équation 6.14 suppose que les tâches de scrutation peuvent être interrompues par l'ordonnanceur à tout moment ce qui met en péril les objets MMS partagés. Par ailleurs, la prise en compte des autres tâches du serveur augmente d'autant plus les possibilités d'accès aux objets MMS partagés. Il est donc nécessaire de protéger les accès à ces objets tout en garantissant le respect des contraintes de temps des tâches de scrutation. Un accès atomique aux objets MMS est d'ailleurs expressément requis par la norme MMS lors du traitement d'une transition d'événement.

[16] Le nombre d'ECs à scruter ne constitue pas une limitation précise dans la mesure où le temps pris pour la scrutation peut varier d'un EC à l'autre surtout lorsque XED est utilisé.

L'information partagée utilisée par les tâches de scrutation est composée :

- des objets ECs scrutés;
- des variables booléennes associées à chaque EC classique;
- des Variables Condition associées à chaque EC XED;
- des variables nommées MMS référencées dans les Variables Condition.

Chaque EC n'est accédé que par la tâche de scrutation qui lui est associée. Toutefois d'autres activités concurrentes ainsi que les requêtes de service MMS peuvent requérir un accès à un EC en cours de scrutation. Les variables sont non seulement accédées par les autres activités concurrentes et les requêtes de service mais potentiellement par d'autres tâches de scrutation. De plus, le traitement des événements survenus après détection ou déclenchement implique un accès de l'ETP[17] aux objets EA et EE.

Ceci est résumé sur la table 6.4.

Objets MMS	Requêtes de service (ou TP)	Traitement des événements (ou ETP)	Tâches de scrutation (ou EM)
Event Condition	R/W	R	R/W
Event Action	R/W	R	
Event Enrollment	R/W	R/W	
Variable booléenne	R/W		R
Variable Condition (XED)	R/W		R
Variables référencées (XED)	R/W		R

TAB. 6.4 - Conflits d'accès aux objets MMS (R - *accès en lecture* / W - *accès en écriture*)

Nous choisissons ici de protéger les objets MMS ainsi que toutes les données partagées d'une VMD par sémaphores. Mais ceci nécessite alors de connaître le pire temps de blocage que peut expérimenter une tâche de scrutation en attente d'un sémaphore occupé. Le protocole PCP que nous avons déjà mentionné est particulièrement adapté puisqu'il permet d'obtenir une borne sur le temps pendant lequel une tâche risque d'être bloquée sur un sémaphore [SRL90], [SRSC90]. En utilisant PCP, Sha *et al.* ont montré qu'une tâche ne pouvait être bloquée plus d'une fois par une tâche de priorité plus faible. Par ailleurs, PCP annule toute possibilité d'interblocage et résout le problème de l'inversion de priorité. Le protocole PCP est souvent considéré comme un progrès majeur dans le domaine de l'ordonnancement temps-réel des tâches à priorité fixe [Kop95]. Nous proposons donc d'utiliser PCP pour implanter un serveur MMS temps-réel et protéger l'accès aux objets partagés.

Nous notons B_i le pire temps de blocage de la tâche de scrutation i par une tâche de priorité inférieure à i. La prise en compte de B_i dans l'équation 6.14 se fait de façon très simple comme l'ont montré Audsley *et al.* dans [ABRW91] :

$$\forall i \ : \ 1 \leq i \leq n, \quad B_i + C_i + \sum_{j=1}^{i-1} \lceil \frac{D_i}{T_j} \rceil C_j \ \leq \ D_i \tag{6.20}$$

Il est donc particulièrement facile d'effectuer un test d'ordonnancement qui prenne en compte la protection des informations partagées lors de la scrutation d'un EC. Il faut cependant que le serveur

[17] Event Transition Processor défini au chapitre 4.

connaisse les temps B_i. Il doit donc connaître parfaitement le temps maximum que toute requête de service MMS ou tout autre activité concurrente du serveur peut passer dans une section critique.

Notons qu'il existe une solution plus simple que PCP applicable dans le cas de serveurs supportant un nombre de services restreint. Cette solution proposée par Mok dans [Mok83] consiste à refuser l'interruption de la requête en cours si elle s'exécute en section critique et ce quelque soit sa priorité. Cette méthode connue sous le nom de *kernelized monitor* a l'avantage de pouvoir être implantée très facilement en comparaison de PCP. Elle est également une solution contre l'inversion de priorité et les interblocages. Toutefois elle présente certains inconvénients :

- Les tests de faisabilité doivent s'effectuer en considérant la section critique la plus longue possible parmi toutes les exécutions possibles de toutes les requêtes MMS et de toutes les activités concurrentes. Ceci rend les tests particulièrement pessimistes surtout dans le cas de MMS où le code s'exécutant dans une zone critique peut nécessiter l'accès au dispositif physique sous-jacent à la VMD. Cet accès à l'équipement physique peut se révéler relativement lent.
- Les sections critiques doivent justement être courtes. Des sections critiques longues diminuent d'autant les possibilités d'ordonnancer les tâches de scrutation. La facilité d'implantation de la méthode *kernelized monitor* ne doit pas cacher la nécessité d'implanter avec précaution les sections critiques.
- Les requêtes ou activités concurrentes n'ayant aucun rapport avec la ressource critique en cours d'utilisation se retrouvent bloquées malgré tout.

Il ne faut donc choisir la solution *kernelized monitor* que dans certains cas particuliers.

6.7 Conclusion

Dans ce chapitre, nous avons analysé les serveurs MMS dans un contexte temps-réel strict et lâche. Nous avons montré qu'il était possible d'exploiter certaines fonctions offertes par MMS pour associer des contraintes de temps aux requêtes de service, ordonnancer leur exécution et assurer dans certains cas une exécution temps-réel ou permettre dans d'autres cas d'entreprendre des actions correctrices dès lors que les contraintes de temps ne peuvent être satisfaites. Ainsi, une des contributions principales de ce chapitre est de montrer que l'on peut utiliser le modificateur MMS `Attach To Semaphore` à cet effet. L'usage que nous faisons du sémaphore n'est pas classique. Toutefois, il est conforme à la norme MMS et permet de faciliter l'ordonnancement des requêtes comme nous l'avons mentionné. Le modificateur permet ainsi à lui seul :

- d'associer une échéance aux requêtes de service MMS confirmé;
- de vérifier que cette échéance est respectée;
- d'effectuer certains types d'actions correctrices déjà définies par MMS;
- d'ordonnancer les requêtes de service dans les serveurs.

Par ailleurs, nous avons étudié comment adapter certains algorithmes connus au cas des serveurs MMS. Nous avons appliqué ceci dans le cas de l'établissement d'associations temps-réel, notamment avec EDF. Nous avons aussi montré comment assurer la détection temps-réel des événements MMS. L'algorithme DM a été proposé à cette fin car il semble le plus adapté aux besoins exprimés par la détection des événements en particulier quand l'extension XED est utilisée.

Le temps-réel est un aspect essentiel des applications industrielles sur lequel les systèmes MMS futurs ne sauront faire l'impasse. Il ne fait aucun doute que beaucoup reste à faire dans ce domaine. Nous ne prétendons pas avoir effectué une analyse exhaustive des systèmes MMS dans le contexte des applications critiques en temps. Mais nous pensons avoir ouvert quelques voies d'exploration intéressantes qui se détachent des approches traditionnellement suivies dans ce domaine.

Chapitre 7

MMS dans un environnement de systèmes répartis

7.1 Introduction

Jusqu'à présent nous avons surtout centré notre étude sur l'architecture des serveurs MMS ou sur les interactions entre un (ou plusieurs) client(s) et un seul serveur. Ce chapitre est plus particulièrement dédié à l'étude des problèmes surgissant quand plusieurs clients communiquent avec un ou plusieurs serveurs, c'est-à-dire quand des applications client entrent en compétition ou concourent à la réalisation d'une tâche commune. Nous nous plaçons maintenant essentiellement du point de vue des clients.

Dans ce chapitre, nous analysons et proposons des solutions basées sur MMS aux problèmes suivants :

- Les rendez-vous entre applications client MMS.
- Le problème des lecteurs/rédacteurs.
- Le problème des philosophes.
- Gestion répartie des événements et services MMS comprenant :
  - Une proposition de modification des événements MMS permettant la définition d'un temps de validité pour les actions événementielles.
  - Une proposition de modification des événements MMS permettant l'exécution d'une action événementielle sur un serveur secondaire.
  - Une extension à la proposition précédente permettant l'exécution de toute requête de service confirmé sur un serveur secondaire.
- Contrôle de la concurrence entre applications client MMS.

Sauf mention contraire, tous les algorithmes présentés sont de niveau de modification 1. La plupart des algorithmes proposés dans ce chapitre ont été écrits avec le langage de spécification PROMELA puis validés avec l'outil SPIN [Hol93]. SPIN est un outil permettant l'analyse de la cohérence logique des systèmes concurrents et est plus particulièrement destiné aux protocoles de communication. SPIN peut exécuter des simulations aléatoires mais son intérêt réside surtout dans la possibilité de valider ou d'infirmer les algorithmes écrits en PROMELA. Cette validation se fait par une **analyse exhaustive**

de tous les états possibles du système considéré. SPIN permet ainsi de garantir l'absence d'interblocages, de messages non reçus ou de parties de code non atteintes. Il peut aussi assurer la correction d'un algorithme ou déceler des cycles dans l'exécution du code [Hol]. Les codes sources PROMELA des algorithmes les moins triviaux sont fournis dans l'annexe E.

7.2 Solutions MMS à des problèmes classiques

7.2.1 Rendez-vous MMS

7.2.1.1 Rendez-vous basés sur les sémaphores MMS

Dans [AM95b], Akazan *et al.* proposent deux solutions pour effectuer un rendez-vous entre deux applications MMS (A_1 et A_2). L'étude de ces solutions parfois inexactes va nous permettre de réaliser que l'utilisation des sémaphores et événements MMS n'est pas toujours triviale.

La première solution est basée sur l'emploi de deux sémaphores MMS s_1 et s_2 à un seul jeton. Cette solution nécessite l'utilisation d'une troisième application client (A_{rdv}) qui permet à A_1 et A_2 de faire le rendez-vous. Les auteurs utilisent deux variables MMS V_1 et V_2 qui lorsqu'elles ont ensemble la valeur 1 permettent la réalisation du rendez-vous. A_{rdv} attend de façon active que ces deux variables prennent la valeur 1. Cette solution illustrée sur la figure 7.1 présente les inconvénients suivants :

- il est nécessaire d'utiliser l'application supplémentaire A_{rdv} pour effectuer le rendez-vous;
- l'attente de A_{rdv} est active. Ce problème est d'ailleurs souligné par les auteurs;
- A_{rdv} doit démarrer les applications A_1 et A_2;
- les auteurs font la supposition que la VMD exécute les services `Read` et `Write` de façon mutuellement exclusive. Normalement, une VMD assure cette exclusion mutuelle. Mais rappelons que ceci n'est que fortement conseillé par la norme. Le rendez-vous risque de ne pas pouvoir s'appliquer à tous les types de serveurs MMS.

Notons d'abord que la séquence de services `Take Control`, `Relinquish Control` qu'effectuent les applications A_1 et A_2 peut être avantageusement remplacée par un seul service avec le modificateur `Attach To Semaphore`. Ceci rend le rendez-vous beaucoup plus rapide puisque le service `Relinquish Control` n'a pas à être effectué[1].

Par ailleurs, l'usage des variables V_1 et V_2 peut être remplacé par le service `Report Semaphore Entry Status` qui permet de savoir si les applications A_1 et A_2 ont atteint le point de rendez-vous. En fait, si aucune autre application ne manipule les sémaphores s_1 et s_2, il suffit de vérifier que la liste de SEs retournée dans la réponse du service `Report Semaphore Entry Status` est non vide pour être sûr que A_1 et A_2 sont prêtes pour le rendez-vous. On aboutit alors à la solution simplifiée de la figure 7.2.

Même avec ces améliorations, la solution que nous avons proposé à la section 2.3.3.1 est plus simple et ne présente pas les inconvénients cités ci-dessus. Nous la reproduisons sur la figure 7.3 pour

[1] L'application A_{rdv} peut également acquérir les sémaphores s_1 et s_2 au moyen de deux modificateurs `Attach To Semaphore` inclus à une même requête `Take Control` appliquée à un troisième sémaphore s. Ceci rend plus équitable la poursuite de l'exécution des applications A_1 et A_2 qui se retrouvent débloquées "quasi simultanément". Dans la solution des figures 7.1 et 7.2 A_2 est toujours pénalisée. En effet, A_2 n'est débloquée qu'après exécution par le serveur de `RelinquishControl(`s_2`)` de A_{rdv}. Il faut donc au moins compter en plus le temps que la requête `RelinquishControl(`s_2`)` parvienne au serveur. Si on travaille en mode synchrone il faut aussi compter le temps que la réponse `RelinquishControl(`s_1`)` parvienne à A_{rdv}. Dans notre implantation MAP/MMS, cette solution pénalise toujours A_2 d'au moins 60 ms. L'utilisation de modificateurs la pénalise d'une valeur de l'ordre d'une ms.

```
 Application A_rdv                          Application A1

begin                                      begin
 MMS_TakeControl(s1)                        % traitement avant le rendez-vous %
 MMS_TakeControl(s2)                        MMS_Write(V1, 1)
 MMS_Start(A1)                              MMS_TakeControl(s1)
 MMS_Start(A2)                              MMS_RelinquishControl(s1)
 boolean_variable:= true                    % traitement après le rendez-vous %
 while boolean_variable                    end A1
 begin
  if not(W1 = 1) then W1 = MMS_Read(V1)
  if not(W2 = 1) then W2 = MMS_Read(V2)     Application A2
  if (W1 = 1) and (W2 = 1) then            begin
  begin                                     % traitement avant le rendez-vous %
   boolean_variable = false                 MMS_Write(V2, 1)
   MMS_RelinquishControl(s1)                MMS_TakeControl(s2)
   MMS_RelinquishControl(s2)                MMS_RelinquishControl(s2)
  end                                       % traitement après le rendez-vous %
 end                                       end A2
end A_rdv
```

FIG. 7.1 - Première solution de rendez-vous proposée dans [AM95b]

```
 Application A_rdv                                       Application A1

begin                                                   begin
 MMS_TakeControl(s1)                                     % traitement avant le rendez-vous %
 MMS_TakeControl(s2)                                     MMS_Status(avec AttachToSemaphore(s1))
 MMS_Start(A1)                                           % traitement après le rendez-vous %
 MMS_Start(A2)                                          end A1
 repeat
  liste_SEs = MMS_ReportSemEntryStatus(s1, QUEUED)       Application A2
 until liste_SEs ≠ ∅
 repeat                                                 begin
  liste_SEs = MMS_ReportSemEntryStatus(s2, QUEUED)       % traitement avant le rendez-vous %
 until liste_SEs ≠ ∅                                     MMS_Status(avec AttachToSemaphore(s2))
 MMS_RelinquishControl(s1)                               % traitement après le rendez-vous %
 MMS_RelinquishControl(s2)                              end A2
end A_rdv
```

FIG. 7.2 - Solution simplifiée du rendez-vous proposée dans [AM95b]

mémoire. Seules les applications A_1 et A_2 participent au rendez-vous. Il n'y a donc pas d'attente active d'une tierce application ni d'utilisation de variables MMS donc pas de problèmes d'accès en exclusion mutuelle à ces variables.

Toutefois, l'inconvénient majeur de cette solution réside dans la phase d'initialisation. Nous supposons en effet que les clients intervenant dans le rendez-vous exécutent une phase d'initialisation lors de laquelle les jetons sont acquis et qu'aucune des deux applications client ne s'exécute tant que la phase d'initialisation de l'autre n'est pas accomplie. Si cette supposition n'est pas réalisable, il est nécessaire

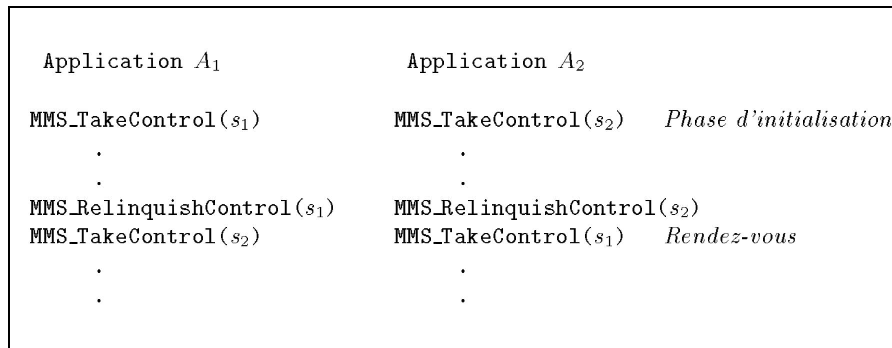


FIG. 7.3 - Rendez-vous avec les sémaphores MMS

de lancer les deux applications à partir d'une troisième et on se ramène donc à une solution du type de celle proposée par Akazan *et al.*.

7.2.1.2 Rendez-vous basés sur les événements MMS

La deuxième solution proposée dans [AM95b] est basée sur le modificateur `Attach To Event Condition` (fig. 7.4). Là encore, deux variables MMS V_1 et V_2 sont utilisées. Dès que ces deux variables valent 1 en même temps un événement lié à un EC scruté (noté EC_{rdv}) est déclenché. Chaque application attend sur une requête `Read` modifiée sur l'EC EC_{rdv}. Le rendez-vous devrait se produire à l'apparition de l'événement qui débloque les requêtes `Read` modifiées.

En fait cette solution risque de ne pas marcher dans la plupart des cas. Supposons que A_1 ait atteint le point de rendez-vous et ait exécuté la requête `Read`. Elle est donc bloquée en attente de la réponse du `Read`. A_2 atteint le point de rendez-vous et effectue la requête `Write(`V_2`)`. L'événement est alors déclenché. Il est fort probable que A_2 n'aura pas encore reçu la réponse du `Write` et n'aura donc pas encore pu faire le `Read`. **Il n'y a donc pas d'EE attaché à EC_{rdv} pour A_2 quand l'événement survient.** Le `Read` de A_2 s'effectue après apparition de l'événement et A_2 reste bloqué indéfiniment. Nous reproduisons sur la figure 7.5 un tel scénario. Pour les mêmes raisons, il est possible que les deux applications restent indéfiniment bloquées si elles atteignent le point de rendez-vous ensemble et effectuent la requête `Write` en même temps.

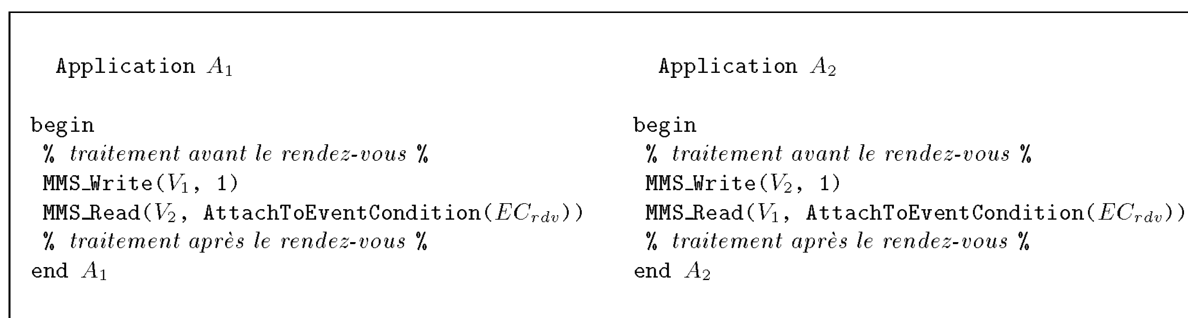


FIG. 7.4 - Seconde solution de rendez-vous proposée dans [AM95b]

La seconde solution proposée dans [AM95b] ne peut marcher que si les requêtes de lecture des deux applications arrivent au serveur et sont traitées **avant** le déclenchement de l'événement. En l'état

actuel de l'algorithme de la figure 7.4, ceci a peu de chances de se produire.

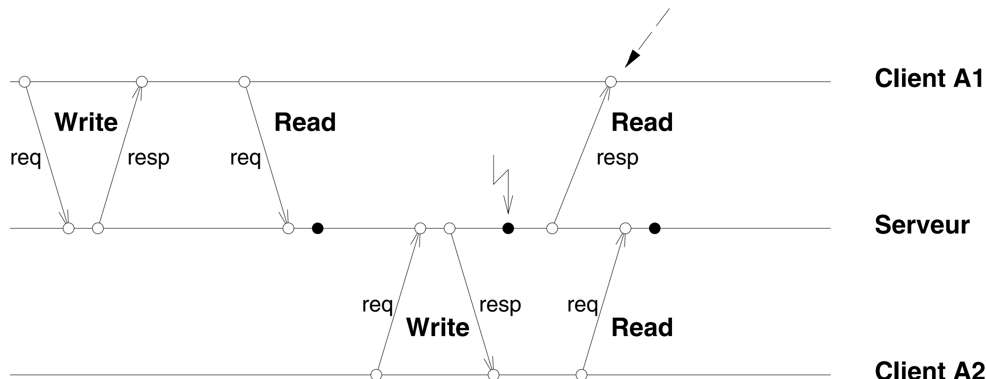
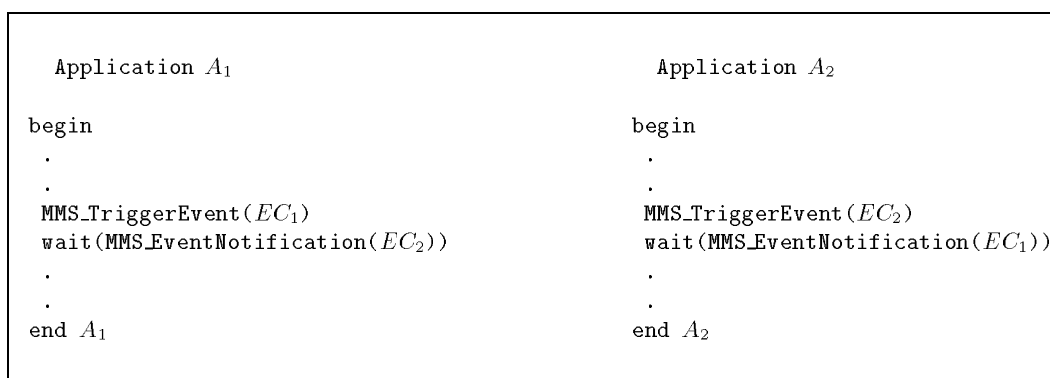


FIG. 7.5 - Un rendez-vous manqué par l'application A2

Enfin on remarquera que la solution ci-dessus suppose l'existence d'un EC scruté dont la transition `IDLE-TO-ACTIVE` est provoquée par la mise à 1 de deux variables MMS. La norme MMS n'impose en aucune façon la pré-définition d'un tel EC dans les serveurs. Il faut donc utiliser un serveur MMS qui définisse spécialement cet événement et soit conçu en ce sens.

Nous proposons sur la figure 7.6 une solution pour le rendez-vous entre client MMS basée sur les événements. L'application A_1 (respectivement A_2) dispose dans un serveur MMS d'un EC noté EC_1 (respectivement EC_2). Nous supposons que chaque application dispose d'un EE lié à l'EC de l'application avec laquelle elle désire effectuer un rendez-vous. Quand l'application A_1 atteint son point de rendez-vous elle déclenche EC_1 avec le service `Trigger Event`. Puis A_1 se met en attente d'une notification venant de EC_2. L'application A_2 fait symétriquement de même. Le rendez-vous se produit quand les deux applications reçoivent chacune la notification.



```
 Application A1                       Application A2

begin                                begin
 .                                    .
 .                                    .
 MMS_TriggerEvent(EC1)                MMS_TriggerEvent(EC2)
 wait(MMS_EventNotification(EC2))     wait(MMS_EventNotification(EC1))
 .                                    .
 .                                    .
end A1                               end A2
```

FIG. 7.6 - Solution de rendez-vous basée sur les événements MMS

Il est important de préciser que les applications client peuvent consulter la liste des messages MMS reçus et ne sont pas simplement bloquées tant qu'un message donné n'est pas arrivé. Supposons que A_2 ait depuis longtemps atteint son point de rendez-vous et ait déclenché l'événement EC_2. La notification est parvenue dans la file des messages reçus sur le site de A_1 mais A_1 n'est pas prête pour le rendez-vous et donc n'attend pas cette notification. Quand l'application client A_1 rejoint son point

de rendez-vous, si elle se met en attente de la notification sans consulter la liste des messages reçus, elle peut rester bloquée indéfiniment en attente d'une notification déjà parvenue.

Pour cette raison l'opération `wait(MMS_EventNotification(`EC_x`))` effectue en fait les étapes suivantes :

```
si MMS_EventNotification(ECx) déjà reçue alors
  continuer
sinon
  attendre MMS_EventNotification(ECx) dans la liste de réception
fin si
```

Remarquons qu'inverser les instructions `Trigger Event` et `wait(MMS_EventNotification)` dans **une seule** des deux applications n'invalide pas le rendez-vous.

7.2.1.3 Rendez-vous multiples

On étend facilement la solution de la section précédente pour effectuer un rendez-vous entre N applications A_i. Un serveur MMS contient N ECs (chacun noté E_i). Chaque EC EC_i est lié à un EE spécifiant que les notifications d'événements déclenchés sur EC_i doivent parvenir à un client particulier noté A_0. Le serveur contient en plus l'EC EC_0 lié à N EEs (chacun noté EE_i). Chaque EE_i identifie la souscription du client A_i à l'événement EC_0.

Le rendez-vous entre les applications client A_i s'effectue par l'intermédiaire du client A_0 comme indiqué sur la figure 7.7.

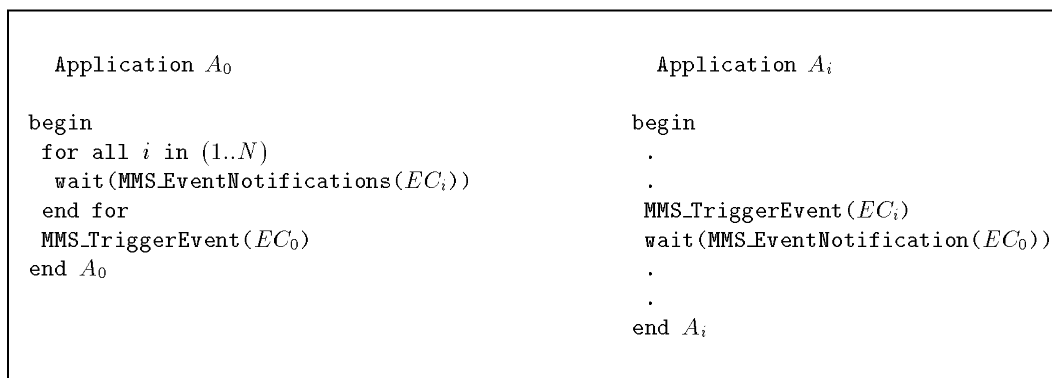


FIG. 7.7 - Solution de rendez-vous multiples basée sur les événements MMS

7.2.2 Lecteurs/Rédacteurs

7.2.2.1 Le problème

Le problème des lecteurs/rédacteurs est le problème bien connu où une ressource est partagée entre différents processus. Certains de ces processus appelés *lecteurs* ne font que consulter la ressource. Il peut donc y avoir simultanément plusieurs lecteurs en train d'accéder à la ressource. D'autres processus appelées *rédacteurs* peuvent modifier la ressource. Les rédacteurs s'excluent mutuellement et excluent les lecteurs également.

Dans [CHP71], une solution simple au problème des lecteurs/rédacteurs est présentée (algorithme 7.A.1). Elle est basée sur l'utilisation de sémaphores classiques. Un sémaphore binaire w protège la ressource contre les accès simultanés de différents rédacteurs et contre les accès simultanés entre lecteurs et rédacteurs. Une variable globale `readCount` est utilisée pour compter le nombre de lecteurs en cours d'accès à la ressource (ligne 1). Le premier lecteur (`readCount` = 1) est responsable de l'exclusion des rédacteurs en effectuant un `P(`w`)` (ligne 7). Le dernier lecteur (`readCount` = 0) doit ré-autoriser les accès de la part des rédacteurs en effectuant l'opération `V(`w`)` (ligne 17). Les rédacteurs ont un comportement plus simple que les lecteurs puisqu'un rédacteur entré en section critique exclut tout autre processus. Les rédacteurs ne font donc qu'appeler l'opération `P(`w`)` avant d'entrer en section critique et `V(`w`)` en en sortant. On notera que cette solution donne la priorité aux lecteurs quand un lecteur occupe déjà la section critique. Cette solution ne prévient donc pas la famine des rédacteurs.

```
 1:  Global variable readCount: integer = 0

 2:  procedure LECTEUR
 3:  begin
 4:    P(mutex)
 5:    readCount:= readCount + 1
 6:    if readCount = 1 then
 7:      P(w)
 8:    end if
 9:    V(mutex)
10:    ...
12:    la lecture se fait ici
13:    ...
14:    P(mutex)
15:    readCount:= readCount - 1
16:    if readCount = 0 then
17:      V(w)
18:    end if
19:    V(mutex)
20:  end LECTEUR
```

Algorithme 7.A.1: Solution classique au problème des lecteurs/rédacteurs

On ne peut appliquer une solution identique quand les opérations `P()` et `V()` sont associées. C'est le cas dans MMS avec les services `Take Control` et `Relinquish Control`. Le problème qui surgit ici vient du fait que le dernier lecteur n'est pas nécessairement le premier. Si les opérations `P()` et `V()` sont associées seul le lecteur qui a effectué `P(`w`)` peut appeler `V(`w`)`. Nous avons vu au chapitre 2 que les sémaphores MMS ne sont pas adaptés pour résoudre les problèmes de synchronisation de la même façon que le font les sémaphores classiques. Le problème des lecteurs/rédacteurs est un exemple typique où une solution différente des solutions habituelles doit être trouvée.

Il existe deux solutions possibles pour résoudre ce problème :

1. soit le premier lecteur autorise le dernier lecteur à prendre le contrôle du sémaphore w **sans le libérer**;

2. soit le premier lecteur est informé du moment où il doit libérer le sémaphore w.

Nous allons présenter deux solutions MMS au problème des lecteurs/rédacteurs. Chacune de ces solutions s'appuie sur une des deux situations citées ci-dessus. Ces deux solutions sont de niveau de

modification 1 puisque nous ne modifions et n'étendons ni les serveurs MMS, ni le protocole MMS. Nous avons publié les algorithmes suivants dans [CK96].

7.2.2.2 Solution basée sur les sémaphores MMS

Notre première solution (notée (1)) utilise la possibilité de réquisition des sémaphores MMS (algorithme 7.A.2). On utilise un sémaphore MMS banalisé w à un seul jeton. La différence principale avec la solution de [CHP71] réside dans les lignes 12 et 23-26. Le premier lecteur prend le contrôle de w avec une requête de service `Take Control` dont le paramètre `Control Timeout` est nul (ligne 12). Ceci provoque un passage immédiat dans l'état `HUNG` du SE correspondant au `Take Control`. Le sémaphore w est donc pris tout en étant dans un état permettant à un autre lecteur de le subtiliser au lecteur qui l'a acquis. A la ligne 24, le dernier lecteur obtient la référence (l'adresse réseau) du premier lecteur au moyen du service `Report Semaphore Entry Status`. Cette référence est reportée dans la requête de service `Take Control` préemptive pour acquérir le sémaphore w (ligne 26). Le jeton unique du sémaphore w qui appartenait au préalable au premier lecteur est alors en possession du dernier lecteur. Le premier lecteur n'a pas besoin de participer à cette opération et le sémaphore w n'est jamais libéré pendant ce transfert. Le dernier lecteur peut alors libérer le sémaphore puisque w lui appartient (ligne 27).

L'exécution de la ligne 24 est inutile si le premier lecteur est aussi le dernier. Nous introduisons donc la variable booléenne `was_first` qui prend la valeur VRAI quand l'algorithme est exécuté par le premier lecteur. Si le dernier lecteur se trouve être aussi le premier, il doit quand même effectuer l'opération de réquisition car MMS stipule qu'un SE dans l'état `HUNG` ne peut en sortir que par un `Take Control` avec réquisition. La requête de service `Relinquish Control(`w`)` peut ensuite être appelée.

Un des avantages de cet algorithme est que le premier lecteur n'a pas besoin de se préoccuper de la terminaison de l'algorithme quand d'autres lectures sont en cours. Quand le premier lecteur a accédé à la section critique, il termine simplement son exécution et ne prend plus part à l'algorithme. Par contre le dernier lecteur doit s'enquérir de l'identité du premier lecteur. Chaque lecteur doit donc pouvoir accéder à l'identité des autres lecteurs.

Contrairement à la solution classique (algorithme 7.A.1), il est possible de garantir une discipline FIFO pour l'attente des rédacteurs puisque les requêtes MMS `Take Control` bloquées sont mises en attente suivant l'ordre FIFO. On peut aussi donner des priorités aux rédacteurs de sorte que ce soit le rédacteur avec la plus haute priorité qui soit le suivant à acquérir w et donc à accéder à la ressource. Il suffit que chaque `Take Control(`w`)` des rédacteurs soit effectué avec le paramètre `Priority` qui donne la priorité de la requête.

Toutefois, ceci ne résout pas le problème de la famine des rédacteurs puisque les lecteurs restent prioritaires tant qu'il y a une lecture en cours. On peut facilement rendre la solution (1) équitable pour les rédacteurs comme pour les lecteurs en utilisant un sémaphore supplémentaire rw. Il suffit d'encapsuler le code d'entrée de la section critique des lecteurs par `Take Control(`rw`)` et `Relinquish Control(`rw`)`. Quant aux rédacteurs, leur première opération doit être `TakeControl(`rw`)` et leur dernière `Relinquish Control(`rw`)`. Comme MMS garantit une politique d'attente FIFO, si un rédacteur arrive alors qu'une lecture est en cours, il passera devant tout lecteur arrivant après lui. Les lecteurs sont en effet bloqués sur rw. C'est la solution classique qui est illustrée notamment dans [CRO75] pp. 317-318 et [Sch86] pp. 65-67. Elle reste adaptée à MMS puisque c'est toujours l'application qui a fait `Take Control(`rw`)` qui effectue `Relinquish Control(`rw`)`. Nous notons cette solution (2).

La solution (2) nécessite l'utilisation d'un sémaphore supplémentaire et ne permet pas de favoriser les rédacteurs par rapport aux lecteurs. Dans [CHP71], une solution (notée (3)) permettant de donner la priorité aux rédacteurs est présentée. Elle requiert aussi l'utilisation de sémaphores supplémentaires

```
 1: procedure LECTEUR
 2:   adr : MMS Application Reference;
 3:   readCount_local : integer := 0;
 4:   was_first : boolean := FALSE;
 5: begin
 6:   MMS_TakeControl(mutex);
 7:   readCount_local := MMS_Read(readCount);
 8:   readCount_local := readCount_local + 1;
 9:   MMS_Write(readCount, readCount_local);
10:   if readCount_local = 1 then
11:     was_first := TRUE;
12:     MMS_TakeControl(w, controlTimeout = 0);
13:   end if;
14:   MMS_RelinquishControl(mutex);
15:   ...
16:   la lecture se fait ici
17:   ...
18:   MMS_TakeControl(mutex);
19:   readCount_local := MMS_Read(readCount);
20:   readCount_local := readCount_local - 1;
21:   MMS_Write(readCount, readCount_local);
22:   if readCount = 0 then
23:     if not was_first then
24:       MMS_ReportSemaphoreEntryStatus(w, adr);
25:     end if;
26:     MMS_TakeControl(w, adr); /* réquisition du premier lecteur */
27:     MMS_RelinquishControl(w);
28:   end if
29:   MMS_RelinquishControl(mutex)
30: end LECTEUR
```

Algorithme 7.A.2: Solution MMS au problème des lecteurs/rédacteurs utilisant la réquisition des sémaphores

(en tout cinq sémaphores sont nécessaires). Cette fois les rédacteurs exécutent eux aussi un algorithme identique à celui des lecteurs dans la solution (1). La solution (3) s'adapte à MMS de façon similaire à la solution (1) en exploitant la possibilité de réquisition d'un sémaphore. Nous ne l'étudions donc pas. En fait on peut exploiter les priorités MMS sur le sémaphore rw de la solution (2) pour déterminer les priorités entre lecteurs et rédacteurs. En donnant des priorités hautes aux rédacteurs ceux-ci passeront avant les lecteurs et vice-versa.

7.2.2.3 Solution utilisant les événements MMS

Dans cette solution que nous notons (4), nous utilisons les événements MMS pour informer le premier lecteur du moment où le sémaphore w doit être libéré (algorithme 7.A.3). Il est spécifié dans la norme [ISO90a] qu'à chaque sémaphore MMS est lié un objet EC (ici noté EC_w). Cet EC est normalement utilisé pour déclencher un événement quand le temps de contrôle d'un sémaphore expire. Nous l'utilisons ici pour permettre au dernier lecteur d'envoyer une notification au premier lecteur. Pour pouvoir recevoir cette notification, le premier lecteur doit d'abord définir un EE lié à EC_w (ligne 12). Pour ce premier lecteur, l'algorithme se poursuit normalement jusqu'à la fin où il doit alors se mettre en attente de la notification d'événement déclenchée par le dernier lecteur (ligne 32).

Quand cette notification arrive, le premier lecteur détruit l'EE créé précédemment (ligne 33) et libère le sémaphore w (ligne 34).

```
 1: procedure LECTEUR
 2:   readCount_local: integer;
 3:   was_first: boolean:= FALSE;
 4: begin
 5:   MMS_TakeControl(mutex);
 6:   readCount_local:= MMS_Read(readCount);
 7:   readCount_local:= readCount_local + 1;
 8:   MMS_Write(readCount, readCount_local);
 9:   if readCount_local = 1 then
10:     was_first:= TRUE;
11:     MMS_TakeControl(w);
12:     MMS_DefineEventEnrollment attached to EC_w;
13:   end if;
14:   MMS_RelinquishControl(mutex);
15:   ...
16:   la lecture se fait ici
17:   ...
18:   MMS_TakeControl(mutex);
19:   readCount_local:= MMS_Read(readCount);
20:   readCount_local:= readCount_local - 1;
21:   MMS_Write(readCount, readCount_local);
22:   if readCount_local = 0 then
23:     if was_first then
24:       MMS_DeleteEventEnrollment attached to EC_w;
25:       MMS_RelinquishControl(w);
26:     else
27:       MMS_Trigger(EC_w);
28:     end if;
29:   end if;
30:   MMS_RelinquishControl(mutex);

31:   if was_first and readCount_local ≠ 0 then
32:     wait(MMS_EventNotification(EC_w)); /* Attente du dernier lecteur */
33:     MMS_DeleteEventEnrollment attached to EC_w;
34:     MMS_RelinquishControl(w);
35:   end if;
36: end LECTEUR
```

Algorithme 7.A.3: Solution MMS au problème des lecteurs/rédacteurs utilisant les événements

Le dernier lecteur est responsable du déclenchement de l'événement au moyen du service `Trigger Event` (ligne 27). Si le dernier lecteur est aussi le premier alors il a seulement besoin de libérer w (ligne 25) après avoir détruit l'EE (ligne 24). Dans ce cas, ce premier/dernier lecteur n'attend aucune notification (ligne 31).

Contrairement à la solution (1), le premier lecteur est responsable de la terminaison correcte de l'algorithme. Toutefois, quand le premier lecteur attend la notification, il n'a pas besoin d'être bloqué. Il peut effectuer un travail utile dans la mesure où ce travail n'implique pas une manipulation du sémaphore w ou des données en section critique.

Cette solution (4) peut paraître compliquée mais il faut se rendre compte que les lignes 10 à 12, 23 à 28 et 31 à 35 ne sont exécutées que pour les premiers et derniers lecteurs. Les autres lecteurs

manipulent simplement `readCount` et entrent en section critique.

7.2.2.4 Remarque sur les performances

Ces algorithmes sont surtout intéressants quand le temps d'accès à la ressource (surtout en lecture) est long par rapport au temps d'exécution du code d'entrée et de sortie de la section critique. Plus ce temps est long, plus il y a de chances d'avoir de lecteurs simultanément en section critique. Il est évident qu'exécuter ces algorithmes sur un réseau avec des processus situés sur différents sites est fort différent d'une exécution sur une même machine où les processus se partagent une mémoire commune.

Si l'accès à la ressource critique ne nécessite l'emploi que d'un service MMS, il est certainement beaucoup plus intéressant d'implanter les algorithmes présentés ici dans le serveur MMS plutôt que de donner aux clients la responsabilité de protéger la ressource. Par exemple pour la lecture et l'écriture d'une variable MMS, les clients lecteurs se contentent de faire une requête `Read` et les rédacteurs `Write`. Il appartient alors à la VMD d'autoriser des requêtes `Read` concurrentes et d'exclure toute requête quand un `Write` est en cours. Il est de la responsabilité de chaque implantation serveur d'assurer ce mécanisme d'exclusion.

Quand l'accès à la section critique se fait avec plusieurs requêtes de services MMS, il ne suffit plus que la VMD offre une protection indépendante à chacune des requêtes. C'est en effet l'intégralité de la section critique qui doit être protégée pour garantir un accès atomique. Dans la section suivante nous proposons d'étendre la requête du service `Take Control` pour résoudre ce problème. Cette extension nous permet de donner une solution extrêmement simple au problème des lecteurs/rédacteurs.

Dans [Ray92], Raynal décrit la solution classique distribuée au problème des lecteurs/rédacteurs. L'algorithme proposé se base sur celui de Chandy et Misra [CM84] et est fondé sur le principe des permissions. Un processus désirant entrer en section critique doit obtenir la permission des autres sites. Entre deux processus, il y a un type de permission pour l'écriture et deux types de permissions pour la lecture. Pour n processus il y donc en tout $\frac{3n(n-1)}{2}$ permissions.

Il est possible de transposer cet algorithme dans l'environnement MMS en utilisant les événements pour envoyer les demandes/obtentions de permissions. Un client désirant une permission déclenche un événement avec le service `Trigger Event`. La notification parvient au client qui détient la permission. Ce client doit à son tour déclencher un événement pour donner la permission. Toutefois ceci nécessite un serveur contenant $3n(n-1)$ objets ECs[2] et autant d'objets EEs. Face au nombre d'objets événementiels nécessaires et au nombre de messages MMS que cet algorithme génère, nous n'avons pas considéré son adaptation à MMS.

La solution décrite par Raynal ne tient pas compte de l'asymétrie résultant de l'existence de sites clients et de sites serveurs comme c'est le cas dans MMS. Tous les sites sont identiques du point de vue de l'algorithme effectué. En fait les solutions (1) à (4) décrites précédemment sont aussi basées sur le principe des permissions (obtention du sémaphore w). Mais l'utilisation d'un site serveur permet de centraliser ces permissions et de réduire leur nombre à 1.

7.2.2.5 Extension des sémaphores MMS

Comme nous l'avons déjà souligné dans la section 2.4, la gestion de sémaphores MMS aurait grandement bénéficié de la définition de primitives qui différencient les accès en mode partagé ou exclusif. Nous proposons donc de rajouter cette capacité à MMS sous la forme d'une option du service `Take Control`. Cette extension va alors nous permettre de résoudre très simplement le problème des lec-

[2] Il y a $\frac{3n(n-1)}{2}$ ECs pour la demande de permissions et autant pour l'envoi des permissions.

teurs/rédacteurs. Le nouveau paramètre optionnel que nous ajoutons (noté `Control Mode`) est de type entier et prend les valeurs `shared` (0) ou `exclusive` (1). Quand `Control Mode` n'est pas présent dans une requête `Take Control`, on dit que la requête demande le contrôle d'un jeton en mode `normal` (3). Cette solution est de niveau de modification 4 car elle étend le protocole MMS. Toutefois elle ne modifie pas le comportement des serveurs mais ne fait qu'étendre ceux-ci avec de nouvelles fonctionnalités. Nous avons proposé cette extension dans [CK96].

La nouvelle définition ASN.1 d'une requête `Take Control` est donnée sur la figure 7.8.

```
TakeControl-Request::= SEQUENCE
  {
  semaphoreName               [0] ObjectName,
  namedToken                  [1] IMPLICIT Identifier OPTIONAL,
  priority                    [2] IMPLICIT Priority DEFAULT 64,
  acceptableDelay             [3] IMPLICIT Unsigned32 OPTIONAL,
  controlTimeOut              [4] IMPLICIT Unsigned32 OPTIONAL,
  abortOnTimeOut              [5] IMPLICIT BOOLEAN OPTIONAL,
  relinquishIfConnectionLost  [6] IMPLICIT BOOLEAN DEFAULT TRUE,
  applicationToPreempt        [7] IMPLICIT ApplicationReference OPTIONAL,
  controlMode                 [8] IMPLICIT ControlMode OPTIONAL
  }

ControlMode::= INTEGER {
  shared               (0),
  exclusive            (1)
  }
```

FIG. 7.8 - Définition ASN.1 du service `Take Control` étendu

On détermine le mode d'acquisition d'un jeton (et donc d'un sémaphore) au moyen du SE qui le détient. Chaque SE est donc étendu avec l'attribut `Control Mode` qui prend :

- la même valeur que celle du paramètre `Control Mode` dans la requête `Take Control` ou
- la valeur `normal` (3) si le paramètre `Control Mode` est absent de la requête `Take Control`.

Le traitement d'une requête `Take Control` par les serveurs reste dans une large mesure identique à celui spécifié dans la norme [ISO90a]. Les serveurs doivent en plus tenir compte des nouveaux cas suivants lors d'une requête `Take Control` :

1. `Control Mode = shared` :
   (a) tous les jetons du sémaphore sont libres : prise de contrôle d'un jeton en mode `shared`, incrémentation de 1 de `Number Of Owned Tokens`.
   (b) un jeton est pris en mode `shared` et d'autres jetons sont libres : prise de contrôle d'un jeton en mode `shared`, incrémentation de 1 de `Number Of Owned Tokens`.
   (c) un jeton est pris en mode `exclusive` et d'autres jetons sont libres : mise en attente de la requête. Le SE est mis dans la liste `List Of Requesters` même si des jetons sont libres.
   (d) un jeton est pris en mode `normal` et d'autres jetons sont libres : prise de contrôle d'un jeton en mode `shared`, incrémentation de 1 de `Number Of Owned Tokens`.

2. `Control Mode = exclusive`:

   (a) tous les jetons du sémaphore sont libres : prise de contrôle d'un jeton en mode `exclusive`, incrémentation de 1 de `Number Of Owned Tokens`.

   (b) un jeton est pris en mode `shared` et d'autres jetons sont libres : mise en attente de la requête. Le SE est mis dans la liste `List Of Requesters` même si des jetons sont libres.

   (c) un jeton est pris en mode `exclusive` et d'autres jetons sont libres : mise en attente de la requête. Le SE est mis dans la liste `List Of Requesters` même si des jetons sont libres.

   (d) un jeton est pris en mode `normal` et d'autres jetons sont libres : prise de contrôle d'un jeton en mode `exclusive`, incrémentation de 1 de `Number Of Owned Tokens`.

3. `Control Mode` absent de la requête : dans tous les cas où des jetons sont libres, il y a prise de contrôle normale (c'est-à-dire selon les spécifications MMS) d'un jeton indépendamment du mode de contrôle éventuel d'autres jetons.

Dans les cas où tous les jetons du sémaphore sont pris, la requête `Take Control` est toujours mise en attente indépendamment du mode utilisé.

Notre définition de l'extension du sémaphore MMS remplit alors les conditions suivantes :

1. Il ne peut y avoir plus d'un jeton pris en mode exclusif dans un sémaphore MMS.
2. Il ne peut y avoir à la fois des jetons pris en mode partagé et un jeton pris en mode exclusif.
3. Il peut y avoir autant de jetons pris en mode partagé que le sémaphore dispose de jetons.
4. Il peut toujours y avoir à la fois des jetons pris en mode normal et des jetons pris en mode partagé.
5. Il peut toujours y avoir à la fois des jetons pris en mode normal et un jeton pris en mode exclusif.

Les SEs qui vont dans l'état `HUNG` perdent leur mode de contrôle exclusif ou partagé pour revenir au mode normal. Les clients qui acquièrent le contrôle d'un jeton dont le SE est dans l'état `HUNG` peuvent utiliser le paramètre `Control Mode` pour re-spécifier le mode d'acquisition du jeton. Par ailleurs, nous étendons également le modificateur `Attach To Semaphore` avec le paramètre `Control Mode` de sorte que l'exécution d'une requête de service MMS puisse être conditionnée sur la prise de contrôle en mode partagé ou exclusif d'un sémaphore.

Quand plus de deux jetons d'un même sémaphore sont pris avec des modes différents (exclusif et normal ou partagé et normal) par un même client sur une même association, il n'est pas possible de préciser quel est le jeton à libérer lors du service `Relinquish Control`. A la section 2.3.2.3, nous avons déjà rencontré dans la norme MMS ce problème de différenciation qui apparaît quand les temporisateurs `Control Timeout` sont utilisés. La solution consistant à libérer le dernier jeton acquis n'est pas satisfaisante car elle limite les possibilités des clients. Nous proposons d'étendre également la requête du service `Relinquish Control` avec le paramètre optionnel `Control Mode`. L'exécution d'une requête `Relinquish Control` contenant ce paramètre est alors un peu similaire à l'exécution de cette même

requête quand un jeton nommé (`Named Token`) est spécifié :

1. <u>`Control Mode = shared`</u> (resp. `exclusive` ou `normal`) :
   (a) aucun jeton n'est pris en mode `shared` (resp. `exclusive` ou `normal`) : le serveur retourne une erreur.
   (b) au moins un jeton est pris en mode `shared` (resp. `exclusive` ou `normal`) sur la même association : libération d'un de ces jetons. Le jeton `shared` (resp. ou `normal`) libéré est indéterminé s'il y en a plus d'un ayant ce mode.
2. <u>`Control Mode` absent de la requête</u> :
   (a) aucun jeton n'est pris : le serveur retourne une erreur comme dans le cas de MMS classique.
   (b) au moins un jeton est pris : libération d'un de ces jetons indépendamment de son mode de contrôle. S'il y en a plus d'un, le jeton libéré est indéterminé.

Pour être complet, nous étendons également le paramètre `State` de la requête du service `Report Semaphore Entry Status` pour pouvoir obtenir uniquement les SEs correspondant aux jetons pris en mode normal (`owner-normal`) ou partagé (`owner-shared`) ou encore exclusif (`owner-exclusive`). La requête de service étendue `Report Semaphore Entry Status` est illustrée sur la figure 7.9.

```
ReportSemaphoreEntryStatus-Request ::= SEQUENCE
  {
  semaphoreName               [0] ObjectName,
  state                       [1] IMPLICIT INTEGER
        {
        queued                (0),
        owner                 (1),
        hung                  (2),
        owner-normal          (3),
        owner-shared          (4),
        owner-exclusive       (5)
        } ,
  entryIdToStartAfter         [2] IMPLICIT OCTET STRING OPTIONAL
  }
```

FIG. 7.9 - Définition ASN.1 du service `ReportSemaphoreEntryStatus` étendu

La principale limitation de notre extension réside dans le fait que le nombre de prises de contrôle en mode partagé est limité par le nombre de jetons du sémaphore.

La figure 7.10 montre la machine d'états d'un objet SE. Elle est étendue par rapport à la machine d'états définie par MMS (fig. 2.2) pour tenir compte des prises de contrôle en mode exclusif et partagé.

Avec cette extension le problème des lecteurs/rédacteurs se résout simplement comme le montre l'algorithme 7.A.4. Le seul inconvénient tient au fait que le nombre de lectures simultanées est limité par le nombre de jetons du sémaphore w. Par contre l'avantage gagné en terme de nombre de services MMS effectués et donc de temps d'exécution est évident.

Alors qu'il fallait dans la solution (1) effectuer 9 requêtes de service MMS pour le premier lecteur, 10 pour le dernier et 8 pour les autres lecteurs, il n'en faut maintenant plus que 2. Ceci représente

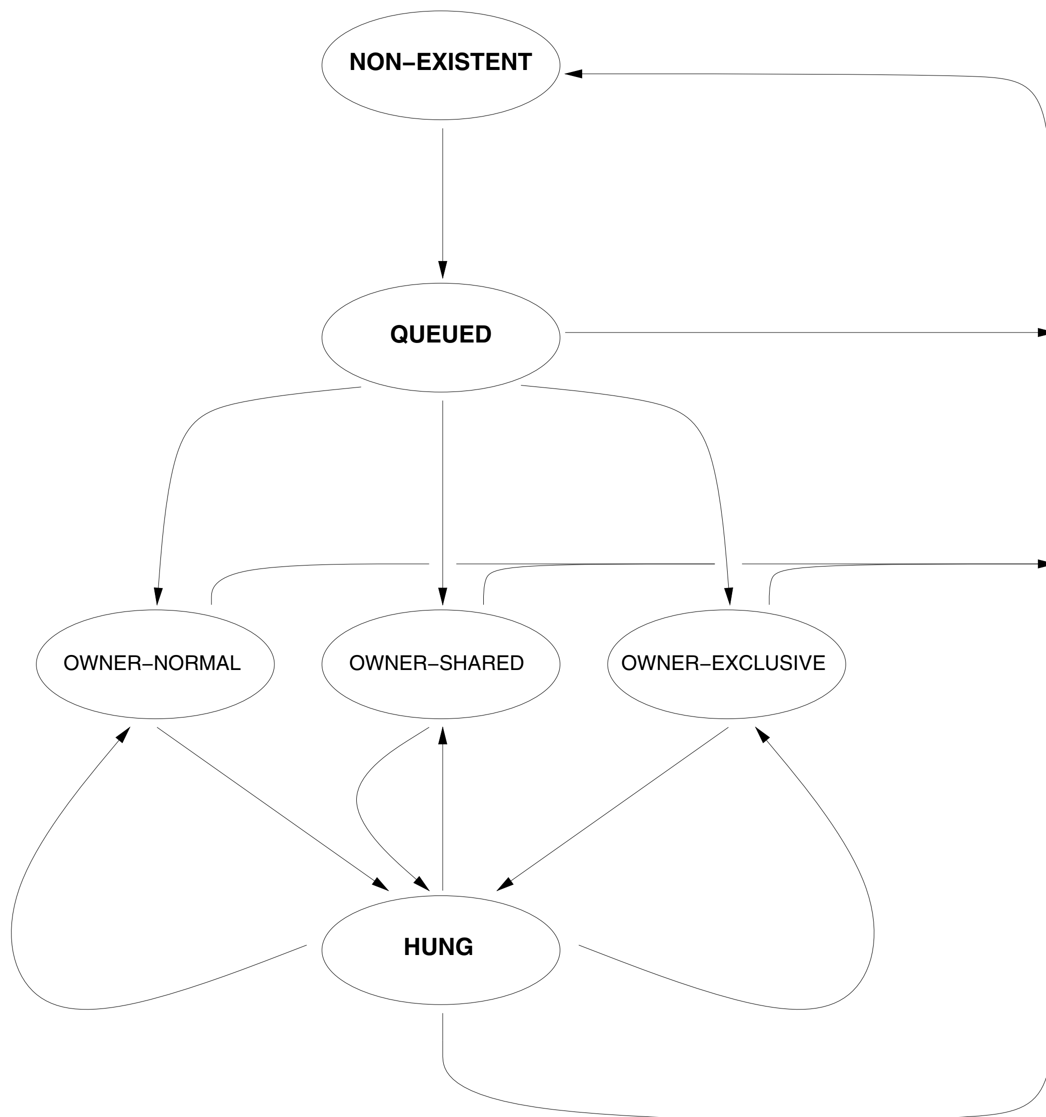


FIG. 7.10 - Le modèle étendu des objets SE

une diminution de 76% du nombre d'octets transmis pour exécuter l'algorithme de lecture[3]. L'octet supplémentaire nécessaire pour coder le paramètre `Control Mode` des requêtes `Take Control` étendues est totalement négligeable.

7.2.3 Le problème des philosophes

Le problème des philosophes est un problème classique et bien connu. Des philosophes sont assis autour d'une table et pensent (fig. 7.11). A tout moment, un philosophe peut désirer manger. Mais

[3] Calculé pour un lecteur différent du premier et du dernier; autrement, la diminution est plus grande encore.

```
procedure LECTEUR                          procedure REDACTEUR

begin                                      begin
  MMS_TakeControl(w, controlMode = shared);   MMS_TakeControl(w, controlMode = exclusive);
  ...                                        ...
  la lecture se fait ici                     l'écriture se fait ici
  ...                                        ...
  MMS_RelinquishControl(w);                  MMS_RelinquishControl(w);
end LECTEUR                                end REDACTEUR
```

Algorithme 7.A.4: Solution au problème des lecteurs/rédacteurs utilisant les modes partagé et exclusif

un philosophe ne peut manger que s'il dispose de deux fourchettes. Chaque philosophe partage sa fourchette de gauche (resp. droite) avec son voisin de gauche (resp. droite). Ainsi, quand un philosophe mange ses voisins ne peuvent manger au même moment. Le problème des philosophes représente de façon plus générale de nombreux problèmes rencontrés dans les systèmes où des processus entrent en conflit pour accéder à des ressources partagées. Nous allons adapter le problème des philosophes à l'environnement MMS.

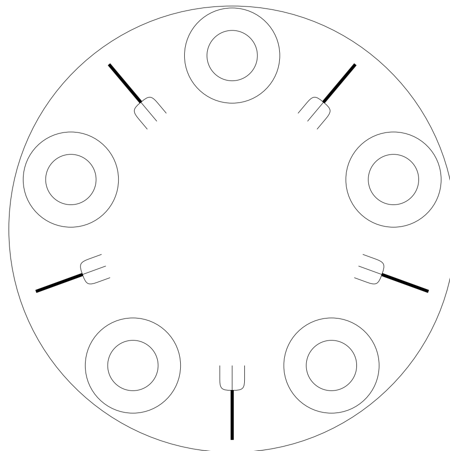


Fig. 7.11 - Le problème des philosophes

Nous illustrons avec l'algorithme 7.A.5 la solution classique au problème des philosophes [CRO75] pp. 32–35. Il y a N philosophes. Toutes les opérations sur les indices se font donc modulo N. L'algorithme 7.A.5 fait apparaître deux problèmes : initialisation à zéro des sémaphores `sempriv` et libération d'un sémaphore `sempriv` par un philosophe différent de celui qui détient ce sémaphore. Comme pour le problème des lecteurs/rédacteurs nous proposons une solution MMS basée uniquement sur les sémaphores et une solution basée sur les événements.

Pour ces deux solutions, les N philosophes sont représentés par N clients MMS. Nous utilisons un seul serveur MMS qui contient la variable MMS `etat` représentant un tableau de N entiers. Ce serveur contient également pour la première solution N sémaphores. Pour la deuxième solution il contiendra N ECs et N EEs.

```
 1: Global variables
 2:   etat    : array[0..N-1] of (PENSER, ATTENDRE, MANGER); /* initialisés à PENSER */
 3:   sempriv : array[0..N-1] of semaphore; /* initialisés à 0 */
 4:   mutex   : semaphore; /* initialisé à 1 */
 5: procedure TEST(i: integer)
 6: begin
 7:   if (etat[i] = ATTENDRE) and (etat[i+1] ≠ MANGER) and (etat[i-1] ≠ MANGER) then
 8:     etat[i]:= MANGER;
 9:     V(sempriv[i])
10:   end if
11: end TEST
12: procedure PHILOSOPHE(i: integer)
13: begin
14:   loop
15:     /* ------------ PENSER ------------ */
16:     P(mutex);
17:     etat[i]:= ATTENDRE;
18:     TEST(i);
19:     V(mutex);
20:     P(sempriv[i]);
21:     /* ------------ MANGER ------------ */
22:     P(mutex);
23:     etat[i]:= PENSER;
24:     TEST(i-1);
25:     TEST(i+1);
26:     V(mutex)
27:   end loop
28: end PHILOSOPHE
```

Algorithme 7.A.5: Solution classique au problème des philosophes

7.2.3.1 Solution basée sur les sémaphores MMS

De la même façon qu'à la section 7.2.2.2, pour résoudre le problème de la libération d'un sémaphore par un client autre que le détenteur du sémaphore nous utilisons l'option `Control Timeout` des requêtes `Take Control`. Toute requête `Take Control` se fait avec une valeur `Control Timeout` nulle de sorte que le SE correspondant se trouve immédiatement dans l'état `HUNG`. Le jeton du sémaphore n'est pas libéré mais simplement “suspendu”. A partir de ce moment, tout philosophe peut obtenir le contrôle de ce sémaphore par réquisition puis effectuer un `Relinquish Control`.

Chaque philosophe ne s'occupe que de la libération de ses voisins immédiats. Nous supposons donc que chaque philosophe connaît l'adresse de ses deux voisins ce qui évite de l'acquérir au moyen du service `Report Semaphore Entry Status` comme c'était le cas pour le problème des lecteurs/rédacteurs. Rappelons que cette adresse est nécessaire pour effectuer la réquisition des sémaphores. Pour simplifier, nous considérons que l'adresse du philosophe i est contenue dans la variable adr_i.

Le problème de l'initialisation à zéro des sémaphores `sempriv` est résolu en exécutant à l'initialisation de chaque philosophe i une prise de contrôle du sémaphore `sempriv[i]`. Cette prise de contrôle se fait avec le paramètre `Control Timeout` nul. Notons qu'aucun philosophe ne doit débuter l'algorithme avant que la phase d'initialisation de tous les philosophes ait été effectuée.

Notre solution se dérive alors facilement de l'algorithme 7.A.5 en remplaçant notamment :

- toutes les occurrences de `P(sempriv[i])` par `MMS_TakeControl (sempriv[i], control Timeout = 0)`.
- toutes les occurrences de `V(sempriv[i])` par la séquence `MMS_TakeControl(sempriv[i], adr); MMS_RelinquishControl(sempriv[i])` où `adr` représente l'adresse du philosophe qui va être autorisé à manger.

L'algorithme complet apparaît sur la figure 7.A.6.

```
 1:  Global variables
 2:    adr0, .., adrN-1: MMS Application Reference; /* initialisés aux adresses des philosophes */
 3:    s0, .., sN-1     : MMS Semaphore; /* initialisés à 0 par TakeControl(si) */
 4:    mutex            : MMS Semaphore; /* initialisé à 1 */
 5:    etat             : MMS Variable array[0..N-1] of (PENSER, ATTENDRE, MANGER); /* init PENSER */

 6:  procedure PHILOSOPHE(i: integer)

 7:    procedure TEST(k: integer)
 8:      e_local: array[0..N-1] of (PENSER, ATTENDRE, MANGER); /* Copie locale de etat */
 9:    begin
10:      e_local:= MMS_Read(etat);
11:      if (e_local[k] = ATTENDRE) and (e_local[k+1] ≠ MANGER) and (e_local[k-1] ≠ MANGER) then
12:        MMS_Write(etat[k], MANGER);
13:        MMS_TakeControl(sk, adrk);
14:        MMS_RelinquishControl(sk)
15:      end if
16:    end TEST

17:  begin
18:    loop
19:      /* ------------ PENSER ------------ */
20:      MMS_TakeControl(mutex);
21:      MMS_Write(etat[i], ATTENDRE);
22:      TEST(i);
23:      MMS_RelinquishControl(mutex);
24:      MMS_TakeControl(si, controlTimeout = 0);
25:      /* ------------ MANGER ------------ */
26:      MMS_TakeControl(mutex);
27:      MMS_Write(etat[i], PENSER);
28:      TEST(i-1);
29:      TEST(i+1);
30:      MMS_RelinquishControl(mutex)
31:    end loop
32:  end PHILOSOPHE
```

Algorithme 7.A.6: Solution MMS au problème des philosophes utilisant les sémaphores

7.2.3.2 Solution basée sur les événements MMS

L'algorithme de la solution basée sur les événements est illustré sur la figure 7.A.7.

Le principe de cette solution est de bloquer un philosophe i qui ne peut manger sur l'attente d'une notification (ligne 22). Quand le voisin de droite ou de gauche du philosophe i finit de manger, il autorise le philosophe i à manger à son tour en déclenchant un événement au moyen du service `Trigger Event` (lignes 30 et 34).

Nous supprimons la procédure TEST et optimisons la solution pour éviter les tests inutiles et les requêtes de service MMS redondantes. Ceci permet en outre d'éviter qu'un philosophe qui peut manger tout de suite ne s'envoie un événement à lui-même.

```
 1: Global variables
 2:   EC0 .. ECN-1: MMS Event Condition;
 3:   EE0 .. EEN-1: MMS Event Enrollment; /* EEi est initialement liés à ECi */
 4:   mutex        : MMS Semaphore; /* initialisé à 1 */
 5:   etat         : MMS Variable array[0..N-1] of (PENSER, ATTENDRE, MANGER); /* init PENSER */

 6: procedure PHILOSOPHE(i: integer)
 7:   ok: Boolean;
 8: begin
 9:   loop
10:     ok := TRUE;
11:     /* ------------ PENSER ------------ */
12:     MMS_TakeControl(mutex);
13:     MMS_Write(etat[i], ATTENDRE);
14:     e_local := MMS_Read(etat);
15:     if (e_local[i+1] ≠ MANGER) and (e_local[i-1] ≠ MANGER) then
16:       MMS_Write(etat[i], MANGER);
17:     else
18:       ok := FALSE;
19:     end if;
20:     MMS_RelinquishControl(mutex);
21:     if not ok then
22:       wait(MMS_EventNotification(ECi));
23:     end if
24:     /* ------------ MANGER ------------ */
25:     MMS_TakeControl(mutex);
26:     MMS_Write(etat[i], PENSER);
27:     e_local := MMS_Read(etat);
28:     if (e_local[i-1] = ATTENDRE) and (e_local[i-2] ≠ MANGER) then
29:       MMS_Write(etat[i-1], MANGER);
30:       MMS_TriggerEvent(ECi-1)
31:     end if;
32:     if (e_local[i+1] = ATTENDRE) and (e_local[i+2] ≠ MANGER) then
33:       MMS_Write(etat[i+1], MANGER);
34:       MMS_TriggerEvent(ECi+1)
35:     end if;
36:     MMS_RelinquishControl(mutex)
37:   end loop
38: end PHILOSOPHE
```

Algorithme 7.A.7: Solution MMS au problème des philosophes utilisant les événements

7.2.3.3 Le problème des philosophes résolu par XED

L'extension XED que nous avons définie au chapitre 5 permet de simplifier de façon radicale la solution au problème des philosophes. Comme nous venons de le voir un philosophe k peut manger dès que la condition suivante est satisfaite :

`(etat[k] = ATTENDRE) and (etat[k+1] ≠ MANGER) and (etat[k-1] ≠ MANGER)`

En associant un EC XED (noté EC_k) à l'expression prédicat précédente on autorise le serveur MMS à déclencher l'événement EC_k quand cette expression prédicat devient vraie. Nous lions maintenant un EE (noté EE_k) à EC_k. EE_k spécifie la transition `IDLE-TO-ACTIVE`. Le philosophe k reçoit donc une notification chaque fois que l'expression prédicat ci-dessus devient vraie. Finalement nous définissons un EA lié à EE_k dont l'action est `Write(etat[k], MANGER)`. Ainsi chaque fois que le prédicat devient vrai le serveur écrit dans `etat` que le philosophe k va manger. Il n'y a donc pas d'intervention du philosophe k. Le philosophe ne fait que signaler qu'il veut manger ou qu'il a fini de manger. Le sémaphore `mutex` n'est plus utilisé.

On aboutit alors à un algorithme extrêmement simple et qui s'exprime de façon très naturelle comme l'illustre l'algorithme 7.A.8. La table 7.1 compare l'efficacité des trois solutions MMS au problème des philosophes que nous venons d'exposer. Pour les solutions avec sémaphores et avec événements, nous donnons les mesures effectuées dans le meilleur cas et dans le pire des cas. Quelques soient le cas et la solution choisie, la supériorité de la solution XED apparaît clairement. Les courbes de la figure 7.12 montrent le temps d'exécution moyen de l'algorithme en fonction du nombre de philosophes. Ces mesures ont été effectuées avec notre implantation de MAP/MMS.

```
 1:  Global variables
 2:    etat : MMS Variable array[0..N-1] of (PENSER, ATTENDRE, MANGER); /* init PENSER */

 3:  procedure PHILOSOPHE(i: integer)

 4:  begin
 5:    loop
 6:      /* ------------ PENSER ------------ */
 7:      MMS_Write(etat[i], ATTENDRE);          /* Je signale que je veux manger */
 8:      wait(MMS_EventNotification(EC_i));     /* J'attends l'autorisation */
 9:      /* ------------ MANGER ------------ */
10:      MMS_Write(etat[i], PENSER);            /* Je signale que je ne mange plus */
11:    end loop
12:  end PHILOSOPHE
```

Algorithme 7.A.8: Solution MMS au problème des philosophes en utilisant XED

Notons toutefois que la solution XED implique les deux hypothèses suivantes :

- Le serveur MMS doit détecter l'événement EC_k et exécuter l'action `Write(etat[k], MANGER)` de façon atomique. Cela signifie que quand un événement est détecté, le processus de détection des événements ne peut se poursuivre tant que l'action associée à l'événement survenu n'a pas été exécutée. Dans le cas contraire, deux philosophes voisins pourraient être autorisés à manger en même temps. On peut réduire cette hypothèse en inhibant la poursuite de la détection des seuls ECs dont l'expression prédicat associée ne fait pas référence à la variable `etat`.

Caractéristiques mesurées	XED	Sémaphores		Evénements	
		Meilleur	Pire	Meilleur	Pire
Nombre de services MMS	3	10	21	8	15
Nombre d'octets transmis	650	2470	4603	1946	3285
Taux de réduction du nombre d'octets transmis (XED par rapport aux autres solutions)	-	74 %	86 %	67 %	80 %
Temps moyen d'exécution pour 1 seul philosophe (ms)	373	non mesuré		1216	

TAB. 7.1 - Comparaison entre les différentes solutions au problème des philosophes

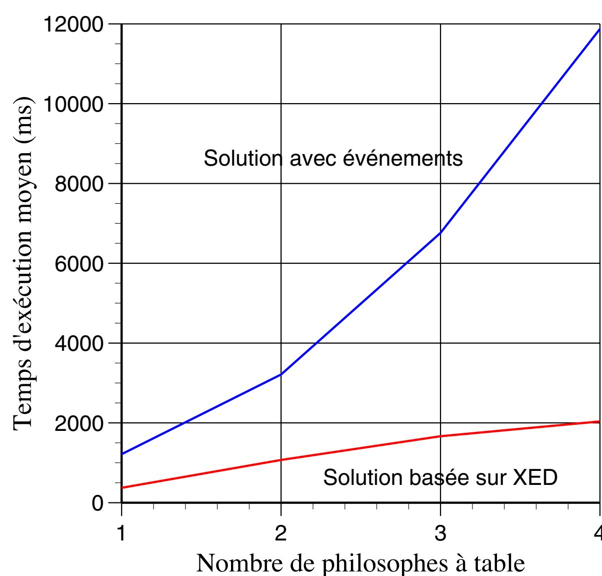


FIG. 7.12 - Le problème des philosophes avec et sans XED

– Le serveur MMS assure l'exclusion mutuelle des accès à chacune des cases du tableau `etat` (requête d'écriture dans une case de `etat` et lecture de cette case par XEDI pour la détection de l'événement). Nous avons déjà mentioné que cette hypothèse n'est que conseillée par la norme MMS mais devrait être naturellement satisfaite pour toute implantation "raisonnable" d'un serveur MMS.

7.3 Gestion répartie des événements et services MMS

7.3.1 Temps de validité d'une action événementielle

A la section 3.11.3, nous avons brièvement étudié comment rajouter un temps de validité à une action événementielle. Ce temps de validité détermine la durée maximale qui doit s'écouler entre le déclenchement d'un événement et l'exécution de l'action.

Nous allons maintenant étudier en détail comment s'intègre cette solution à MMS.

Création de l'EA : un client définit une action et lui fournit un temps de validité au moyen du

service `Define Event Action`. Ce service est donc étendu avec le paramètre optionel entier `Validity Interval` (fig. 7.13). Lors de la création de l'objet EA, la valeur `Validity Interval` est recopiée dans l'attribut de même nom de l'EA. On étend donc aussi l'objet EA avec ce nouvel attribut (fig. 7.14). Quand le paramètre `Validity Interval` est absent de la requête de service, l'attribut `Validity Interval` de l'EA prend la valeur `FOREVER`[4] (pour toujours). Il est possible que la VMD puisse déterminer lors de la création de l'EA que le temps de validité fournit et trop petit et que donc la VMD se retrouvera dans l'incapacité de traiter l'action dans le temps demandé. Dans ce cas, l'EA n'est pas créé et l'erreur MMS de classe `time-resolution` est retournée dans la réponse du service `DefineE vent Action`.

Apparition de l'événement : le temps auquel se produit l'événement est sauvegardé dans l'attribut `Transition Time` de l'objet EN. Ceci se fait dans la procédure 1 (EC Update) de traitement des événements décrite dans la norme MMS. Les temporisateurs liés aux actions événementielles comportant des temps de validité ne peuvent être démarrés tout de suite car à ce stade de traitement de l'événement la VMD ne sait même pas si des EAs liés à l'événement existent[5].

Traitement de l'action : ce n'est que lors de la procédure 5 ("Event Action Execution") que la VMD détermine les actions à exécuter. Pour chaque action il faut faire la différence entre le temps courant et le temps d'apparition de l'événement (notée δ). On soustrait ensuite δ au temps de validité de l'action.

- Si le résultat est négatif alors le temps de validité est écoulé. L'action n'est pas exécutée et aucun objet transaction n'est créé. La réponse de l'action incluse à la notification est négative et stipule l'erreur MMS `timeout` de classe `service-preempt`.
- Si le résultat est positif alors l'objet transaction nécessaire pour traiter l'action est créé. Le gestionnaire de temporisateurs est appelé pour démarrer une temporisation avec la valeur (`Validity Interval` - δ). Le temporisateur est identifié par les attributs `AAId` et `InvokeId`[6] de l'objet transaction. Nous étendons l'objet transaction avec l'attribut `Remaining Validity Interval`. Le temps restant avant la fin de l'interval de validité est constamment mis à jour dans l'attribut `Remaining Validity Interval` de l'objet transaction.

Expiration du temps de validité : lors de l'expiration du temporisateur, si l'action n'a pas commencé à être exécutée, l'objet transaction est détruit. Les objets EE ou SE traitant les modificateurs éventuels sont également détruits. La réponse de l'action incluse à la notification est négative et stipule l'erreur MMS `timeout` de classe `service-preempt`. Le traitement de l'événement se termine alors normalement.

Fin d'exécution de l'action : quand l'exécution de l'action se termine avant la fin du temps de validité, le temporisateur est arrêté et détruit. La réponse de l'action incluse à la notification est positive et stipule les résultats de l'action. Le traitement de l'événement se termine alors normalement.

Cette solution est de niveau de modification 4 car le service `Define Event Action` est étendu avec un paramètre optionnel. Si l'on suppose que les temps de validité sont pré-définis dans les EAs et que

[4] Cette valeur est définie dans la norme pour les intervalles de temps non spécifiés.

[5] Il peut ne pas y avoir d'EA du tout, ou alors il peut exister des EAs qui n'ont pas de temps de validité associé ou encore il peut exister des EAs avec des temps de validité mais le client n'a pas souscrit à la transition d'événement survenue.

[6] Ces valeurs sont générées de façon locale par le serveur et n'ont aucun lien avec une association existante. Ceci permet d'homogénéiser le traitement des actions et des requêtes de service.

```
DefineEventAction-Request::= SEQUENCE
  {
  eventActionName              [0] ObjectName,
  listOfModifier               [1] IMPLICIT SEQUENCE OF Modifier OPTIONAL,
  confirmed-Service-Request    [2] ConfirmedServiceRequest,
  validityInterval             [3] IMPLICIT Unsigned32 OPTIONAL
  }
```

FIG. 7.13 - Définition ASN.1 du service `Define Event Action` étendu

```
Object: Event Action

  Key Attribute: Event Action Name
  Attribute: MMS Deletable (TRUE, FALSE)
  Attribute: Confirmed Service Request
  Attribute: List Of Modifier
  Attribute: List Of Event Enrollment Reference
  Attribute: Validity Interval
  Attribute: Additional Detail
```

FIG. 7.14 - L'objet Action Evénementielle étendu avec l'intervalle de validité

les clients ne peuvent les fournir avec le service `Define Event Action` alors notre solution devient de niveau de modification 3. En effet, le protocole MMS est inchangé mais le comportement des serveurs est modifié. Enfin, si l'on suppose que le temps de validité ne sert que de guide à la VMD pour ordonner le traitement des actions et que les exécutions des actions ne sont pas annulées lors de l'expiration des temps de validité alors notre solution est une simple implantation particulière de la norme MMS et devient de niveau de modification 1.

Le temps de validité est particulièrement important quand l'action est effectuée sur un site distant comme nous le verrons à la section suivante. Remarquons qu'il n'est pas possible d'utiliser les modificateurs comme à la section 6.5.2 pour fournir les temps de validité des actions. En effet, la temporisation `Acceptable Delay` ne démarre que lorsque le modificateur correspondant de l'objet transaction est traité. Ceci se produit bien **après** l'apparition de l'événement. Notre temps de validité est défini comme la durée s'écoulant entre l'apparition de l'événement et l'exécution de l'action.

7.3.2 Exécution d'une action événementielle distante

Dans [Dak90] p. 39, Dakroury remarque que dans les applications industrielles la production d'un événement, la notification de cet événement et l'exécution de l'action associée se font souvent sur trois sites différents. L'auteur suggère alors de répartir les objets événementiels EC, EE et EA sur trois sites différents : l'EC sur le serveur où se produit l'événement, l'EA sur le serveur où doit s'exécuter l'action et l'EE sur la station qui informe les utilisateurs de l'événement[7]. Cette répartition n'est toutefois pas

[7] Il n'est pas clair si l'auteur considère que cette station est un serveur MMS comprenant un objet "Operator Station" ou le client devant recevoir la notification auquel cas ce client est aussi un serveur puisqu'il doit contenir l'EE.

étudiée plus en détail dans [Dak90]. Nous proposons une solution alternative présentant les mêmes avantages :

1. L'objet EC permet de détecter l'événement. Il est donc naturel que l'objet EC se trouve sur le serveur où se produit l'événement.

2. L'objet EE permet de savoir **à qui** envoyer la notification. Il permet de souscrire à l'événement. Il ne sert donc à rien de le mettre sur la station qui doit recevoir la notification. Il faudrait alors de toutes façons que le serveur où se produit l'événement dispose d'une information lui permettant de savoir où se trouve cet EE. C'est-à-dire qu'il faudrait un objet qui remplisse la fonction ancienne de l'EE. Une fois cet EE localisé, cet objet ne servirait à rien puisque le serveur connaît maintenant l'adresse de la station qui doit recevoir la notification. En outre l'EE permet de connaître l'action à effectuer lors de l'apparition de l'événement. Il est donc inutile d'aller chercher la référence de cette action sur un serveur distant quand on peut l'obtenir localement. Ne voyant pas l'utilité de mettre l'EE sur un serveur différent de celui où se produit l'événement, nous conservons la fonction MMS de l'EE et maintenons l'EE sur le serveur contenant l'EC.

3. L'objet EA permet d'exécuter l'action. Nous avons montré à la section 3.11.4.1 l'intérêt d'une solution permettant d'exécuter une action sur un serveur différent de celui où se produit l'événement. Il pourrait ainsi sembler intéressant que l'EA soit situé sur le serveur où l'action événementielle doit s'exécuter. Une délocalisation de l'EA exige une extension de l'objet EE pour connaître l'adresse du serveur contenant l'EA mais surtout la définition d'un nouveau service MMS dont la fonction est de demander l'exécution du service contenu dans un EA. En fait, l'important est que **l'action soit exécutée** sur ce serveur et non que l'EA soit défini dans celui-ci. Considérant que la délocalisation de l'EA ne respecte pas l'esprit de la norme MMS et introduit trop de modifications, nous proposons de conserver l'EA sur le serveur où se produit l'événement et d'autoriser l'exécution de l'action proprement dite sur un autre serveur.

Nous allons donc maintenant étudier plus en détail la proposition faite à la section 3.11.4.1 qui permet l'exécution des actions événementielles sur un site autre que le serveur où se produit l'événement.

```
DefineEventAction-Request::= SEQUENCE
  {
  eventActionName             [0] ObjectName,
  listOfModifier              [1] IMPLICIT SEQUENCE OF Modifier OPTIONAL,
  confirmed-Service-Request   [2] ConfirmedServiceRequest,
  serverApplication           [3] ApplicationReference OPTIONAL
  modifiersRemote             [4] IMPLICIT BOOLEAN DEFAULT FALSE
  }
```

FIG. 7.15 - Définition ASN.1 du service `Report Semaphore Entry Status` étendu

Le comportement des serveurs MMS supportant l'exécution d'actions distantes est le suivant :

Création de l'EA : un client définit un EA comportant une action distante au moyen du service `Define Event Action`. Nous étendons donc la définition du service `Define Event Action` avec le nouveau paramètre `Server Application`. Ce paramètre représente l'adresse du serveur MMS

```
Object: Event Action

  Key Attribute: Event Action Name
  Attribute: MMS Deletable (TRUE, FALSE)
  Attribute: Confirmed Service Request
  Attribute: List Of Modifier
  Attribute: List Of Event Enrollment Reference
  Attribute: Server Application
  Attribute: Modifiers Remote
  Attribute: Additional Detail
```

FIG. 7.16 - L'objet Action Evénementielle étendu avec l'adresse du serveur distant

qui doit exécuter l'action événementielle. Il est de type `Application Reference`. Ce type est déjà défini dans MMS [ISO90b]. Nous ajoutons également le paramètre `Modifiers Remote` qui n'a de sens que si le paramètre `List Of Modifier` n'est pas vide. Le paramètre `Modifiers Remote` est un booléen qui spécifie si les modificateurs attachés à l'action doivent être exécutés sur le serveur local ou sur le serveur distant représenté par `Server Application`. Dans la suite, nous qualifierons ces serveurs respectivement de *principal* et *secondaire*. La requête du service `Define Event Action` ainsi étendue apparaît sur la figure 7.15. Lors de la création de l'objet EA, les valeurs de `Server Application` et `Modifiers Remote` sont recopiées dans les attributs de même nom de l'EA. On étend donc aussi l'objet EA avec ces nouveaux attributs (fig. 7.16). Quand le paramètre `Server Application` est absent de la requête de service, cela signifie que l'action est locale et que l'on reste dans le cadre d'une utilisation classique de MMS. L'attribut `Server Application` de l'EA prend alors la valeur `UNDEFINED`.

Traitement de l'action après apparition de l'événement : c'est lors de la procédure 5 ("Event Action Execution") que la VMD détermine les actions à exécuter pour l'événement qui s'est produit. Pour chaque action, la VMD consulte l'attribut `Server Application` de l'EA pour savoir si l'action est locale ou distante :

- si la valeur est `UNDEFINED`, l'action est locale et son traitement se poursuit comme dans MMS classique. Dans ce cas, l'attribut `Modifiers Remote` doit avoir pour valeur `FALSE`. La valeur `TRUE` constitue une erreur.
- si la valeur est autre que `UNDEFINED` alors l'action doit s'exécuter sur un serveur secondaire. Le serveur principal crée un objet transaction pour traiter l'action. Si l'attribut `Modifiers Remote` est `FALSE` alors la liste de modificateurs de l'action est affectée à celle de l'objet transaction. Les modificateurs doivent donc être satisfaits localement avant que l'action ne soit envoyée au serveur secondaire. Si l'attribut `Modifiers Remote` est `TRUE`, les modificateurs doivent être satisfaits sur le serveur secondaire et sont envoyés avec l'action.

  Le serveur principal détermine ensuite s'il existe une association avec le serveur secondaire identifié par `Server Application`. Si tel n'est pas le cas, il ouvre cette association au moyen de la procédure 6 ("Establish Application Association") de traitement des événements. La requête de service représentant l'action est ensuite construite et envoyée sur l'association. Cette requête contient les éventuels modificateurs attachés à l'action uniquement si l'attribut `Modifiers Remote` est `TRUE`. Le gestionnaire d'événement se met alors en attente des résultats de cette action de la même manière qu'il attend les résultats d'une action locale.

Réception de la réponse de l'action : lors de la réception de la réponse de l'action, l'objet transaction est détruit et les résultats sont retournés au gestionnaire d'événements comme dans MMS classique. Le serveur peut alors continuer le traitement de l'événement et envoyer la notification avec les résultats de l'action distante. Notons que le client qui reçoit la notification peut être situé sur un site différent de celui du serveur qui a exécuté l'action distante.

La figure 7.17 reprend l'exemple de la section 3.11.4.1 en illustrant cette fois les deux possibilités d'utilisation des modificateurs dans l'exécution d'une action distante.

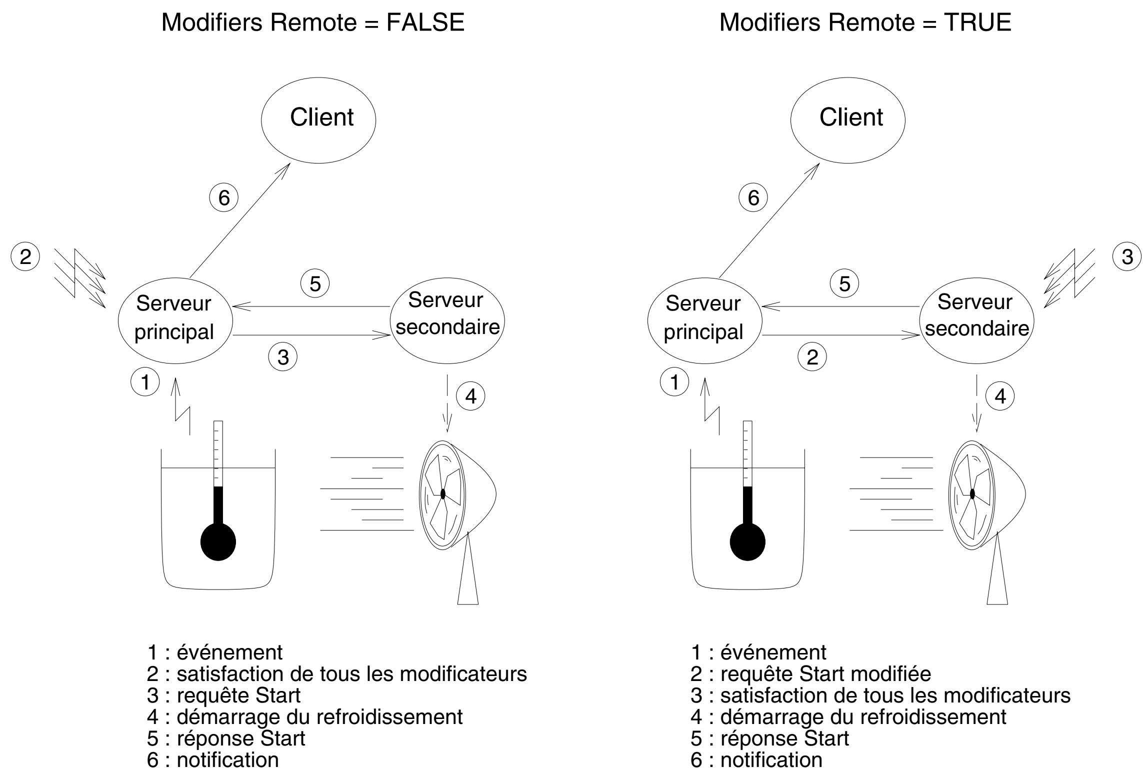


FIG. 7.17 - Séquencement des étapes lors d'une action distante avec modificateurs

L'exécution d'action distante s'intègre particulièrement bien à MMS sans modifier les fonctionnalités existantes. En particulier, les serveurs secondaires sur lesquels les actions sont exécutées n'ont pas besoin d'être étendus comme le serveur principal et peuvent donc rester de niveau de modification 1. Ceci vient en grande partie du fait qu'une action événementielle n'est autre qu'un service MMS et s'exécute de la même façon en créant un objet transaction. Notons que pendant l'exécution d'une action distante, il y a deux objets transaction qui représentent l'action. Le premier se situe sur le serveur principal en attente de la réponse de l'action. Le second se situe sur le serveur secondaire où s'exécute l'action.

Comme nous rajoutons deux paramètres optionnels à la requête `Define Event Action`, notre solution est de niveau de modification 4. Le comportement des serveurs MMS n'est toutefois pas modifié mais simplement étendu. Pour être complet, nous rajoutons également l'attribut `Server Application` à la réponse du service `Get Event Action Attributes` ce qui permet d'informer les clients du site sur lequel doit s'effectuer l'action (fig. 7.18).

Nous pouvons combiner l'extension proposée ici avec le temps de validité d'une action (section 7.3.1). Lors de l'expiration du temps de validité d'une action distante dont l'exécution est en cours, le serveur principal considère que l'action a échoué. Il construit une réponse négative stipulant l'erreur

```
GetEventActionAttributes-Response::= SEQUENCE
  {
  mmsDeletable                [0] IMPLICIT BOOLEAN DEFAULT FALSE,
  listOfModifier              [1] IMPLICIT SEQUENCE OF Modifier,
  confirmed-Service-Request   [2] ConfirmedServiceRequest,
  serverApplication           [3] ApplicationReference OPTIONAL
  }
```

FIG. 7.18 - Définition ASN.1 du service `Get Event Action Attributes` étendu

MMS `timeout` de classe `service-preempt` et l'inclut dans la notification. La notification est ensuite envoyée sans attendre les résultats de l'action distante.

Deux solutions sont alors envisageables :

1. Le serveur principal attend la réception des résultats de l'action distante. Lors de la réception de ces résultats, le serveur principal constate que l'attribut `Remaining Validity Interval` de l'objet transaction local est nul. L'objet transaction est alors détruit. Les résultats de l'action ne sont pas retournés au gestionnaire d'événements puisque la notification a déjà été envoyée.
2. Le serveur principal envoie une requête `Cancel` au serveur secondaire pour annuler l'action distante. Il détruit l'objet transaction quand une réponse positive au service `Cancel` lui parvient. Mais il est possible que l'action ne puisse être annulée ou que le serveur secondaire ait déjà envoyé la réponse de l'action quand la requête `Cancel` lui arrive. Dans ce cas, il est nécessaire d'adopter le même comportement que le point 1 ci-dessus.

L'automate des serveurs principal et secondaire illustrant l'exécution d'une action distante sont représentés sur la figure 7.19.

7.3.3 Exécution d'une requête de service MMS distante

Comme nous l'avons vu, l'exécution d'une action événementielle distante nécessite la création de deux objets transaction : le premier dans le serveur principal et le second dans le serveur secondaire. Nous pouvons ainsi étendre l'objet transaction avec l'attribut `Server Application` (fig. 7.20). Si cet attribut n'a pas pour valeur `UNDEFINED`, il identifie le serveur secondaire où s'exécute réellement l'action.

Cette extension de l'objet transaction nous amène naturellement à autoriser une exécution distante des requêtes de service MMS au même titre que les actions. Un serveur MMS (principal) qui reçoit une requête de service et qui se trouve dans l'incapacité de l'exécuter peut alors relayer cette requête vers un autre serveur (secondaire).

Les raisons pour lesquelles un serveur doit relayer une requête de service vers un autre serveur peuvent être multiples :

- **Répartition de la charge** : le serveur principal se trouve trop chargé et un autre serveur est capable d'effectuer la requête;

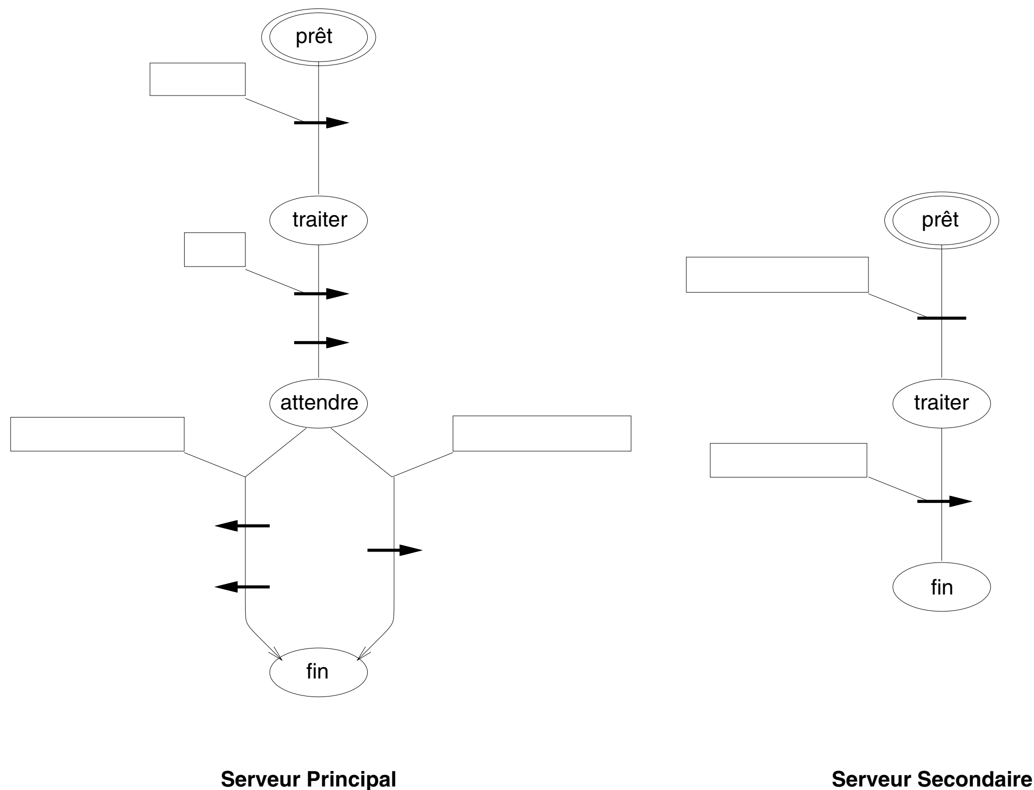


FIG. 7.19 - Automate de Nutt pour l'exécution distante des actions

- **Délocalisation des objets MMS** : pour des raisons de transparence, les clients ont une vision centralisée des objets MMS qui en apparence se situent tous sur le serveur principal. En fait, les objets sont répartis sur différents serveurs secondaires. Le serveur principal relaie les requêtes des clients vers les serveurs secondaires en fonction des objets à accéder.

- **Tolérance aux fautes** : le dispositif physique que modélise la VMD ou certaines fonctionnalités du serveur principal sont dans un état qui ne permet pas de répondre à la requête d'un client. Si l'on dispose d'un serveur de rechange, la requête peut être exécutée sur ce dernier serveur.

L'attribut `Server Application` de l'objet transaction du serveur principal contient l'adresse du serveur vers lequel la requête est relayée. Le serveur principal devient donc client du serveur secondaire. Lors de la réception de la réponse, le serveur principal la recopie dans la PDU réponse qu'il doit envoyer au client. Il détruit ensuite l'objet transaction correspondant. Le traitement des modificateurs éventuels est identique au cas des actions événementielles. Ces différentes étapes se font de façon transparente pour le client qui a effectué la requête. Pour ce client, tout se passe comme si le serveur principal exécute la requête de service. Notre extension est donc de niveau de modification 2. Si par contre nous souhaitons que les clients aient la possibilité de préciser le serveur secondaire pour chacune de leur requête, alors il faut ajouter les paramètres optionels `Server Application` et `Modifiers Remote` dans toutes les requêtes de service MMS. Ceci rend notre solution de niveau de modification 4. L'automate simplifié de notre extension intégrant le temps de validité est présenté sur la figure 7.21. L'automate du serveur secondaire est le même qu'à la figure 7.19.

Notons que notre extension est différente du protocole multi-serveurs défini par Dakroury et Elloy dans [DE89], [Dak90]. Dakroury étend les services MMS agissant sur les listes de variables et les invocations de programmes de façon qu'une requête de service MMS effectuée sur le serveur principal se divise en plusieurs requêtes envoyées aux serveurs secondaires. Par exemple, un objet MMS `Named`

```
Object: Transaction

  Key Attribute: InvokeId
  Key Attribute: Application Association Identifier
  Attribute: List Of Pre-execution Modifiers
  Attribute: Current Modifier Reference
  Attribute: Confirmed Service Request
  Attribute: List Of Post-execution Modifiers
  Attribute: Cancelable (TRUE, FALSE)
  Attribute: Server Application
  Attribute: Modifiers Remote (TRUE, FALSE)
```

FIG. 7.20 - L'objet transaction étendu

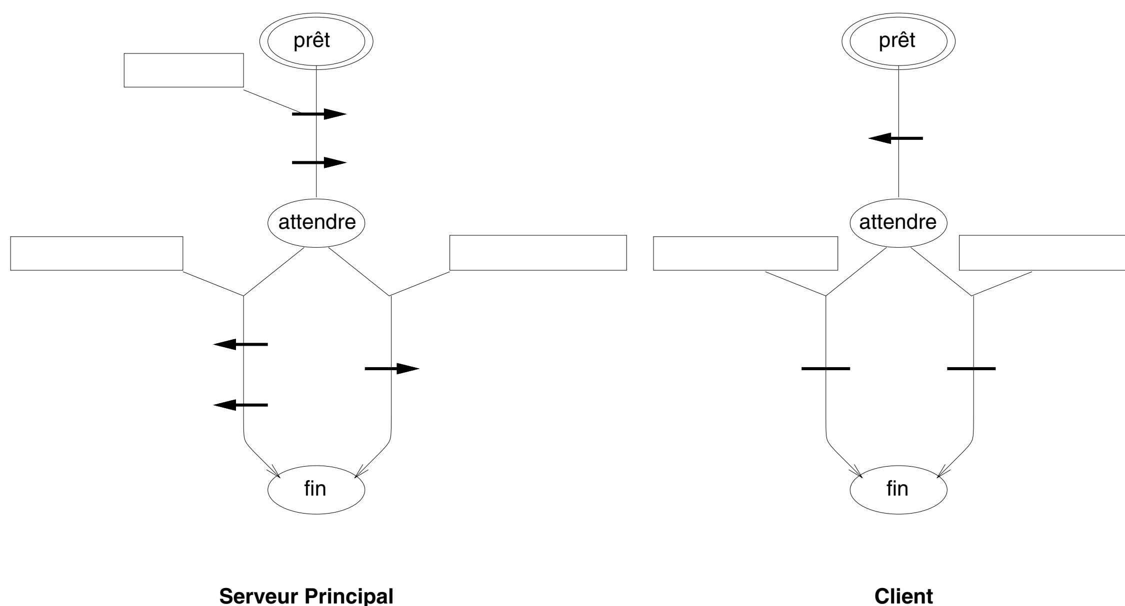


FIG. 7.21 - Automate de Nutt pour l'exécution distante des services

`Variable List` peut référencer des variables situées sur différents serveurs. Une requête de lecture de cet objet se divise en autant de requête `Read` qu'il y a de variables distantes dans les serveurs secondaires. Le serveur principal attend les réponses des serveurs secondaires, construit la réponse principale à partir des sous-réponses reçues et envoie cette réponse au client correspondant.

Dans notre extension, les objets MMS autres que les objets transaction sont inchangés. Les requêtes de service sont relayées intégralement d'un serveur à un autre. Et ceci peut se faire pour n'importe quelle requête MMS. La combinaison du protocole multi-serveurs et de notre extension permet une gestion réellement répartie des requêtes de service MMS et une exécution de ces requêtes de façon totalement transparente pour les clients.

7.4 Algorithmes de contrôle de la concurrence

7.4.1 Hypothèses de base

L'acquisition et la manipulation de données constituent une partie importante des applications industrielles et de productique. Le problème que rencontrent les développeurs de telles applications vient de l'hétérogénéité des données, du nombre de protocoles utilisés pour y accéder et de la complexité des interfaces de ces protocoles. Un des objectifs du projet ESPRIT CCE-CNMA était de faciliter le développement d'applications industrielles en offrant aux utilisateurs une plate-forme ouverte et portable qui intègre harmonieusement ces diverses technologies et cache aux développeurs d'applications la complexité et les différences d'accès à tous ces supports d'information [Ple94b], [CC95]. Dans le cadre de ce projet, nous avons développé un compilateur SQL qui donne aux utilisateurs la possibilité d'accéder aux données de systèmes industriels d'une façon simple, cohérente et homogène [Ple92b], [Cas95b]. Cette interface SQL exploite une plate-forme appelée CCE qui garantit la cohérence d'accès aux informations partagées. Cette garantie disparaît quand on ne dispose plus de CCE et que l'on adapte directement SQL sur MMS. Il est alors nécessaire de disposer d'algorithmes basés sur MMS qui assurent le contrôle de la concurrence d'accès aux données.

Cette section est consacrée à l'étude d'algorithmes assurant le contrôle de la concurrence dans l'environnement MMS (indépendamment de l'utilisation avec SQL citée ci-dessus). Nous montrons comment utiliser les sémaphores et événements MMS pour construire des algorithmes simples, exempts d'interblocages et destinés à la protection de ressources réparties et/ou répliquées dans un environnement de réseaux industriels. Nous avons publié ces algorithmes dans [Cas95a] et [Cas95c].

Nous nous intéressons donc au problème suivant. Etant donné un ensemble de ressources réparties sur différents serveurs MMS :

- comment assurer un accès sérialisé à des sous-ensembles de ces ressources à la manière des transactions dans les bases de données ?
- comment éviter, détecter et résoudre les interblocages qui peuvent éventuellement se former lors de ces tentatives d'accès ?

MMS ne fournit pas les outils nécessaires pour gérer facilement la répartition des ressources. Malgré cela, pour pouvoir garantir la compatibilité de nos solutions avec tout système MMS nous ferons l'hypothèse essentielle suivante : **MMS doit être utilisé sans aucune modification ou extension**. Nos solutions doivent donc être de niveau de modification 1. Le problème initial qui pouvait sembler trivial devient alors moins évident car de nombreuses techniques utilisées dans les systèmes distribués ou les bases de données réparties ne peuvent être adoptées ici. En fait, nos algorithmes doivent s'inscrire dans le cadre d'un environnement restreint caractérisé de la façon suivante :

1. **Le format des messages est fixé** : les PDUs MMS doivent être utilisées excluant tout autre type de communication. Les messages échangés entre les différents sites doivent donc suivre la syntaxe décrite dans [ISO90b]. Ces messages ne peuvent être construits en fonction des besoins des algorithmes.
2. **Pas d'estampillage des messages** : l'estampillage paraît difficilement réalisable avec MMS. On pourrait utiliser les attributs `InvokeId` des services MMS confirmés mais même dans ce cas, les serveurs MMS (étant par hypothèse inchangés) ne seraient pas capable d'exploiter cet estampillage.
3. **Pas de diffusion** : MMS ne permet pas la diffusion des requêtes de service.

4. **Passivité des gestionnaires de ressources**: seuls les sites qui demandent l'accès aux ressources partagées (les client MMS) sont actifs. Les sites qui renferment les ressources (les serveurs MMS) sont passifs en ce sens qu'ils ne peuvent qu'attendre les requêtes des clients[8].

5. **Faible connectivité** : pour rester aussi général que possible, nous supposons que les différents sites sont soit client, soit serveur mais pas les deux à la fois. Il n'y a donc pas de communication entre clients, ni de communication entre serveurs. En fait, les clients (resp. serveurs) ne connaissent même pas l'existence des autres clients (resp. serveurs). Ceci implique en particulier que les serveurs ne peuvent pas coopérer (par exemple pour résoudre les interblocages) comme c'est le cas pour de nombreux gestionnaires de bases de données distribuées.

Dans ce contexte, notre but n'est pas de proposer de nouveaux types d'algorithmes distribués génériques mais plutôt de déterminer ce qu'il est possible de faire avec MMS pour résoudre le problème de l'allocation de ressources réparties.

7.4.2 Modèle et notations

Le système considéré consiste en N clients MMS et M serveurs MMS. Les sites serveurs gèrent des ressources (variables, fichiers, machines industrielles, ...). Les sites clients ont besoin d'accéder à ces ressources. Bien qu'il puisse y avoir plusieurs ressources dans un seul serveur MMS, pour des raisons de simplicité nous considérons que chaque serveur i contient une seule ressource R_i. L'expression *ressource globale* se réfère au sous-ensemble de ressources auxquelles un client veut accéder à un moment donné. Chaque ressource de ce sous-ensemble est appelée *composante* de la ressource globale. Chacune de ces composantes est donc sur un serveur différent. On note $R = (R_1, ..., R_M)$ la ressource globale R dont les composantes R_i sont respectivement réparties sur les serveurs i. Par exemple un fichier et ses différentes copies de sauvegarde est une ressource globale. On peut trouver dans [Dak90] un autre exemple d'un tel scénario. Deux convoyeurs sont gérés et représentés par deux VMDs différentes. Chaque VMD contient une variable qui contrôle le mouvement de son convoyeur. Il est nécessaire d'écrire au même moment la même valeur dans chacune des variables pour que les convoyeurs bougent ensembles. Si deux clients effectuent une écriture simultanée avec des valeurs différentes, il peut arriver que les convoyeurs se déplacent dans des directions opposées ou avec des vitesses différentes. Un scénario identique est également décrit dans [RZI90].

Pour assurer la sérialisation des accès aux ressources, nous utilisons une méthode similaire au verrouillage à deux phases (2PL ou "Two Phase Locking") [EGLT76]. A chaque composante R_i est associée un sémaphore banalisé MMS à un seul jeton (noté S_i). Un client doit obtenir le contrôle de tous les sémaphores associés aux composantes qu'il veut accéder avant de pouvoir faire une quelconque opération d'accès aux ressources (2PL conservatif [BHG87] pp. 58–59). Si un sémaphore ne peut être obtenu le client est mis en attente sur ce sémaphore. Les sémaphores sont libérés lorsque toutes les opérations d'accès ont été effectuées. Comme nous associons un sémaphore MMS à chaque composante, nous ne pouvons pas faire la distinction entre accès exclusif et accès partagé. Les accès aux composantes se font donc toujours en mode exclusif. Nous ne faisons aucune hypothèse quant à l'ordre dans lequel les requêtes de contrôle/libération des sémaphores sont issues des différents clients.

Par analogie avec les bases de données, nous dirons qu'un client qui tente d'accéder à une ressource globale effectue une *transaction*. Les clients qui doivent interrompre leur transaction pour résoudre les interblocages sont appelés *victimes*.

---

[8] Un serveur peut décider indépendamment des clients d'envoyer une requête de service non confirmée. Les circonstances amenant à une telle décision ne sont pas normalisées par MMS. Cette décision est donc propre à chaque application serveur et ne peut être exploitée ici.

Enfin, les caractéristiques des média de communication sont celles du profil MAP [Gen88]. Par conséquent, la communication entre deux sites est fiable : aucun message n'est perdu, dupliqué ou délivré avec erreur. Sur une association MMS, les messages sont délivrés dans l'ordre dans lequel ils ont été envoyés et sont traités par les serveurs dans cet ordre. Par contre, nous ne pouvons faire d'hypothèse ni sur le temps mis par un message pour aller d'un site à un autre, ni sur le temps de traitement dans les serveurs excepté que ces temps sont finis.

La plupart des requêtes de service MMS sont effectuées en mode synchrone. Les clients attendent donc de recevoir la réponse à une requête avant de poursuivre. Dans nos algorithmes, ces requêtes sont préfixées par "MMS_". Dans certains cas les clients ont besoin de poursuivre leur exécution après avoir effectué une requête de service. Une telle requête est donc asynchrone et est préfixée par "MMS_Async_".

7.4.3 Résolution des interblocages par temporisation

7.4.3.1 L'algorithme de base

Les limitations 4 et 5 de la section 7.4.1 nous amènent à centraliser l'intelligence des algorithmes sur les clients MMS. Un client doit lui-même déterminer quand il peut poursuivre une transaction (les sémaphores sont libres) ou interrompre et différer cette transaction (il y a interblocage). Ces simples observations nous permettent de faire un parallèle avec les protocoles de réseaux locaux CSMA (Carrier Sense Multiple Access) et Ethernet. Dans ces protocoles, le canal de communication est passif et chacune des stations doit seule déterminer si elle peut émettre ou si elle doit interrompre son émission suite à une collision et recommencer plus tard. Pour dériver les deux algorithmes suivants nous avons fait un parallèle entre les stations et les clients MMS, entre le médium de communication et les sémaphores dans les serveurs MMS, et entre les collisions et les interblocages.

L'algorithme de base est présenté sur la figure 7.A.9. Cet algorithme expose le principe de l'algorithme plus évolué qui sera décrit à la section 7.4.3.2. Comme S_i dénote le sémaphore présent dans le serveur i, nous introduisons les variables d'entrée X_i pour souligner que l'ordre et le nombre de requêtes `Take Control` sont propres à chaque client. Par exemple dans le cas d'une variable globale constituée de trois variables réparties sur les serveurs 1, 2 et 3, l'algorithme peut être exécuté avec $(X_1, X_2, X_3) = (S_1, S_2, S_3)$ ou $(X_1, X_2, X_3) = (S_2, S_1, S_3)$ ou même $(X_1) = (S_3)$ si chaque variable est la réplique des autres et que l'opération effectuée est une lecture. K ($K \leq M$) dénote le nombre total de sémaphores à saisir lors d'une transaction.

Comme nous l'avons décrit plus haut, le principe de base de l'algorithme consiste simplement à prendre le contrôle de tous les sémaphores associés aux composantes à accéder, à effectuer les opérations d'accès puis à libérer tous les sémaphores acquis. L'utilisation de sémaphores peut évidemment mener à des conflits[9] ainsi qu'à des interblocages.

L'algorithme de base soupçonne qu'il y a interblocage quand une transaction est bloquée en attente d'un sémaphore pendant un temps donné c'est-à-dire quand la durée d'un conflit dépasse une valeur prédéfinie. Pour fixer les temps maximaux d'attente des transactions nous utilisons l'option `Acceptable Delay` des requêtes `Take Control` (ligne 6). Si le temporisateur `Acceptable Delay` d'une requête `Take Control` expire avant que le contrôle du sémaphore soit acquis alors cette requête est automatiquement annulée. Le client qui a effectué la requête reçoit une réponse négative lui spécifiant la fin de temporisation (ligne 7). La résolution d'un interblocage se fait par libération de tous les sémaphores détenus quand un client reçoit une réponse négative suite à l'expiration d'un temporisateur `Acceptable Delay` (lignes 8 à 10). Ceci permet aux autres transactions de poursuivre leur exécution.

[9] On appelle *conflit* le fait qu'une transaction demande un sémaphore détenu par une autre transaction.

```
 1: local variables
 2:   i, j, D1 : integers;

 3: begin
 4:   START: MMS_TakeControl_Request(X1);

 5:   for i in (2..K) do
 6:     MMS_TakeControl_Request(Xi, acceptableDelay = D1);
 7:     if NEGATIVE MMS_TakeControl_Response received then
 8:       for j in (i-1 .. 1) do
 9:         MMS_RelinquishControl_Request(Xj)
10:       end for;
11:       goto START
12:     end if
13:   end for;

14:   /* Accès aux ressources ici */

15:   for i in (K..1) do
16:     MMS_RelinquishControl_Request(Xi)
17:   end for
18: end
```

Algorithme 7.A.9: Algorithme de base

La transaction annulée est alors redémarrée.

L'adoption des temporisateurs pour détecter les interblocages est motivée par les raisons suivantes :

- l'algorithme résultant est simple;
- c'est souvent la seule solution dans un système constitué de sites qui ne peuvent pas coopérer pour résoudre un problème;
- l'implantation dans l'environnement MMS s'avère très facile.

Toutefois l'utilisation de temporisateurs présente certains inconvénients. Tout d'abord le fait qu'un temporisateur `Acceptable Delay` expire ne garantit pas qu'il y a bien interblocage. L'exécution d'une transaction peut simplement être ralentie pour des raisons indépendantes de la présence d'un interblocage. En ce sens, on ne peut pas réellement parler de **détection** d'un interblocage. L'algorithme ne peut que **supposer** qu'il y a interblocage. Par contre, tout interblocage implique une attente indéfinie et donc l'expiration d'un temporisateur [BHG87].

Le choix de la durée de temporisation est également un inconvénient. Une durée trop courte engendre des redémarrages intempestifs. Une durée trop longue tend à ralentir l'exécution des transactions car l'algorithme ne détecte pas d'interblocage avant l'expiration d'un temporisateur. La valeur du temporisateur dépend donc de la charge du système considéré. Comme dans tous les algorithmes où les temporisateurs sont utilisés pour détecter des interblocages, le choix de la durée de temporisation est un problème délicat. Généralement cette durée doit être réglée en fonction des caractéristiques du système (voir à ce sujet [CP85], [BHG87], [JTK89], [FHRT93]).

Dans notre cas nous avons adopté une durée définie de façon aléatoire. Nous en donnons la justification à la section suivante. Par ailleurs, il est intéressant d'ajouter à cette durée une valeur proportionnelle au nombre de sémaphores couramment détenus par la transaction. Ceci est équitable car

une transaction possédant plusieurs sémaphores (*transaction longue*) a ainsi moins de chance d'être annulée et redémarrée qu'une transaction possédant peu de sémaphores (*transaction courte*).

Par ailleurs, il faut aussi éviter que les temporisations de plusieurs transactions interbloquées n'expirent en même temps. Le redémarrage simultané de deux transactions interbloquées ne nuit pas à la validité de l'algorithme mais réduit ses performances. Prenons le cas de deux transactions T_1 et T_2 en situation d'interblocage. Nous supposons que le temporisateur de T_1 a une durée de temporisation AD_1 et expire le premier au temps δ_1. Nous cherchons à estimer la valeur de temporisation AD_2 de T_2 pour savoir dans quelle mesure le temporisateur de T_2 expire également. Soient k le nombre de sémaphores que détient T_1 au temps δ_1, RC le temps de réponse typique moyen d'une requête `Relinquish Control` et TC celui d'une requête `Take Control` en l'absence de conflit. Enfin req_1 (resp. req_2) est l'instant où la requête `Take Control` de T_1 (resp. T_2) est bloquée. C'est donc également l'instant où le temporisateur de T_1 (resp. T_2) est armé. La transaction T_2 est redémarrée si le temps δ_2 d'expiration de son temporisateur est tel que :

$$\delta_1 \leq \delta_2 \leq \delta_1 + \frac{TC}{2} + (k_{T_2} - 1)RC + \frac{RC}{2} \tag{7.1}$$

Dans cette équation nous avons négligé les temps de traitement dans les serveurs MMS devant les temps de communication dans la pile de protocole. Nous avons montré dans [CV95] que sur notre réseau MAP cette approximation pouvait être faite. L'équation 7.1 exprime simplement le fait que T_2 est redémarrée si son temporisateur expire avant que le sémaphore attendu par T_2 soit libéré. Ce sémaphore est le ${k_{T_2}}^{eme}$ a être libéré par T_1. Comme $k_{T_2} \leq k$, δ_2 est majoré par $(\delta_1 + \frac{TC}{2} + \frac{(2k-1)RC}{2})$ qui représente le temps de libération de tous les sémaphores détenus par T_1.

Pour être sûr que AD_2 n'expire pas également, il suffit que les équations suivantes soient assurées selon le cas dans lequel on se trouve :

1. si $req_2 \geq \delta_1$ alors

$$AD_2 > \frac{TC}{2} + \frac{(2k_{T_2} - 1)RC}{2} \tag{7.2}$$

2. si $req_1 \leq req_2 \leq \delta_1$ alors

$$AD_2 > AD_1 + \frac{TC}{2} + \frac{(2k_{T_2} - 1)RC}{2} \tag{7.3}$$

3. si $req_2 \leq req_1$ alors

$$AD_2 > AD_1 + req_1 - req_2 + \frac{TC}{2} + \frac{(2k_{T_2} - 1)RC}{2} \tag{7.4}$$

Ces équations comportent 3 inconnues pour T_2 : req_1, AD_1 et k_{T_2} qui ne permettent pas à T_2 de déterminer la valeur de AD_2 avant d'envoyer la requête `Take Control` correspondante.

Cette étude nous permet au moins d'obtenir des estimations sur l'ordre de grandeur souhaitable de AD_2 pour notre implantation de l'algorithme sur réseau MAP. En affectant $TC = RC = 65$ ms et $k = k_{T_2} = 5$ on obtient $AD_2 > 325$ ms pour le cas 1. Pour $k = k_{T_2} = 10$ on obtient $AD_2 > 650$ ms toujours pour le cas 1.

Sûreté et vivacité

Les propriétés de *sûreté* et de *vivacité* sont importantes pour assurer le bon fonctionnement de nos algorithmes [Ray92]. Dans notre cas, ces propriétés se traduisent de la façon suivante :

1. **sûreté** : à tout instant et pour toute ressource globale il y a au plus un client MMS qui accède à cette ressource;
2. **vivacité** : tout processus qui demande l'accès à une ressource doit pouvoir y accéder dans un temps fini. La propriété de vivacité traduit le fait que toute transaction se termine dans un temps fini. Elle comporte ainsi deux aspects :
   (a) toutes les situations d'interblocage sont résolues;
   (b) l'algorithme n'entre pas dans une situation (non interbloquée) où au moins un client ne progresse pas dans l'acquisition des sémaphores.

Pour tous nos algorithmes, les propriétés de sûreté et vivacité ont fait l'objet d'une vérification formelle avec l'outil SPIN [Hol93]. Les algorithmes codés en langage PROMELA sont fournis en annexe E.

En ce qui concerne l'algorithme de base, la propriété de sûreté est assurée de façon évidente. Tous les sémaphores associés à une ressource globale doivent être pris avant qu'un client puisse accéder à cette ressource. Aucun autre client ne peut donc détenir au même moment un seul de ces sémaphores et accéder à la ressource ou à un sous-ensemble des composantes de la ressource. Ce résultat découle directement de l'adoption de la méthode 2PL.

La propriété de vivacité 2(a) est également vérifiée. Une situation d'interblocage est un état stable du système qui se traduit par un cycle d'attente entre différentes transactions. Cette attente engendre l'expiration du temporisateur ayant la plus petite durée et donc la libération de tous les sémaphores détenus par la victime. Le sémaphore impliqué dans l'interblocage est donc aussi libéré et le cycle d'attente est rompu.

La propriété de vivacité 2(b) ne peut pas être strictement garantie par l'algorithme. Cette situation est d'ailleurs détectée par SPIN. Prenons le cas de deux transactions interbloquées. Il est parfaitement possible de tomber dans la situation suivante :

1. Les temporisateurs `Acceptable Delay` des deux transactions comportent des délais suffisamment proches pour que les deux transactions soient annulées et redémarrent en même temps;
2. Les deux transactions redémarrent et acquièrent les mêmes sémaphores dans le même ordre que précédemment. Elles se retrouvent donc en situation d'interblocage;
3. Retour au point 1.

Dans ce cas, l'algorithme s'exécute mais ne progresse pas. Cette situation provient directement de la façon dont est conçu l'algorithme. Dans les réseaux locaux comme Ethernet, on ne peut pas non plus théoriquement garantir que plusieurs stations désirant émettre ne vont pas continuellement entrer en collision. Pratiquement, les stations peuvent émettre car la probabilité que les paquets soient ré-émis en même temps après une collision est faible. C'est l'indétermination dans le choix du moment de ré-émission qui permet la communication.

Nous appliquons la même idée pour nos algorithmes. Ceci explique pourquoi nous avons adopté des durées aléatoires pour les temporisateurs `Acceptable Delay`. La propriété de vivacité 2(b) ne peut

donc être garantie mais l'indétermination dans le choix des temporisations permet à l'algorithme de se terminer dans des situations où la compétition n'est pas excessive. Une mesure de la vivacité 2(b) est présentée dans la section suivante.

7.4.3.2 Amélioration de l'algorithme

Dans cette section, nous allons améliorer l'algorithme de base. Le nouvel algorithme obtenu est numéroté algorithme 7.A.10. Nous lui avons essentiellement apporté quatre améliorations :

1. Le défaut majeur de l'algorithme de base provient du redémarrage des transactions. Dès qu'un temporisateur expire, la transaction correspondante est redémarrée et prend le contrôle de sémaphores qui peuvent être nécessaires à l'exécution d'autres transactions. Ceci risque d'être inutile si le sémaphore qui a causé la fin de temporisation est toujours détenu. Nous noterons désormais ce sémaphore S_{abort}.

   Le sémaphore S_{abort} n'appartient pas à la victime. Il peut donc être utilisé pour détecter quand il sera libéré, c'est-à-dire quand la transaction qui le détient sera terminée (ou annulée). Nous faisons donc en sorte que la victime ne puisse redémarrer que lorsque S_{abort} est libéré. Pour éviter d'avoir à consulter périodiquement l'état du sémaphore S_{abort}, il suffit d'envoyer une requête `Take Control(`S_{abort}`)` sans paramètre `Acceptable Delay`. La requête est alors mise en attente jusqu'à ce que le sémaphore soit libre. **La réception de la réponse à cette requête `Take Control` est le signal qui autorise la victime à redémarrer** (ligne 7). Cette façon de procéder est aussi à rapprocher du protocole Ethernet : une station ne ré-émet pas tant que la ligne est occupée.

2. L'utilisation des priorités des sémaphores MMS permet également d'améliorer notre algorithme. Nous utilisons le paramètre `Priority` des requêtes `Take Control` pour augmenter les chances d'exécution des transactions qui subissent de fréquents redémarrages. Ainsi, lors de la première exécution d'une transaction les requêtes `Take Control` se voient affectées une priorité normale que nous fixons à 126. A chaque redémarrage la priorité de ces requêtes est diminuée d'une unité[10] (ligne 15). De cette façon, les transactions anciennes augmentent leur chance d'exécution [BG81].

3. Nous voulons éviter qu'une victime n'affecte l'exécution des transactions qui n'ont pas redémarré et qui utilisent ou attendent S_{abort}. La requête `Take Control` sur un sémaphore S_{abort} s'effectue donc toujours avec une priorité inférieure aux priorités des transactions actives. Comme nous avons fixé à 126 le niveau de priorité normale, la priorité basse d'attente sur S_{abort} est égale à 127. C'est la plus basse priorité dans MMS.

4. Pour accélérer l'exécution d'une transaction, nous effectuons toutes les requêtes MMS `Relinquish Control` en mode asynchrone. Cela signifie que toutes les requêtes de libération des sémaphores sont envoyées en bloc sans attendre la réponse de chacune d'entre elles pour poursuivre. Nous avons montré dans [CV95] que l'utilisation du mode asynchrone permet de gagner un temps non négligeable sur le temps de réponse total des services exécutés.

Sûreté et vivacité

Les commentaires sur les propriétés de sûreté et vivacité sont les mêmes pour l'algorithme 7.A.9 et pour l'algorithme 7.A.10. La sûreté et la vivacité 2(a) de l'algorithme de base étant acquises, les modifications qui ont conduit à l'algorithme 7.A.10 ont essentiellement pour but de favoriser la vivacité 2(b).

---

[10] Jusqu'à la priorité maximum qui est 0. Après les priorités restent inchangées.

```
 1: local variables
 2:   i, j, k, priority, detectionPriority, D1 : integers;

 3: begin
 4:   k ← 1;
 5:   priority ← normalPriority;
 6:   detectionPriority ← normalPriority + 1;

 7:   START: MMS_TakeControl_Request(X_k, detectionPriority); /* Attente sur S_abort */

 8:   for i in (1 .. K) except k do
 9:     MMS_TakeControl_Request(X_i, D1 , priority);
10:     if NEGATIVE MMS_TakeControl_Response received then
11:       for j in (i-1 .. 1) ∪{k} do
12:         MMS_Async_RelinquishControl_Request(X_j)
13:       end for;
14:       k ← i;
15:       priority ← priority − 1; /* Augmenter la priorité des victimes */
16:       goto START
17:     end if
18:   end for

19:   /* Accès aux ressources ici */

20:   for i in (K .. 1) do
21:     MMS_Async_RelinquishControl_Request(X_i)
22:   end for
23: end
```

Algorithme 7.A.10: Algorithme amélioré

Nous verrons plus loin que ces modifications ont effectivement permis d'obtenir une meilleure vivacité. Nous cherchons maintenant à améliorer encore ces résultats. Dans les réseaux locaux Ethernet, la solution retenue pour diminuer le nombre de collisions entre paquets est d'ajouter un délai avant la ré-émission d'un paquet. En particulier, avec l'algorithme Binary Exponential Backoff (BEB) [MB76], ce délai est choisi de façon aléatoire avec une moyenne qui dépend du nombre de collisions passées. Nous adoptons dans les algorithmes la même idée et rajoutons une instruction d'attente durant un temps D_2 entre les lignes 15 et 16 de l'algorithme 7.A.10. Nous pouvons ainsi comparer les cinq cas suivants :

1. Pas de délai D_2 et redémarrage immédiat (algorithme de base 7.A.9).
2. Pas de délai D_2 et redémarrage uniquement quand le sémaphore S_{abort} est libre.
3. Génération aléatoire du délai D_2 et redémarrage uniquement quand le sémaphore S_{abort} est libre.
4. Génération aléatoire du délai D_2 auquel on ajoute une valeur proportionnelle au nombre de transactions en cours et redémarrage uniquement quand le sémaphore S_{abort} est libre.
5. Génération du délai D_2 basée sur l'algorithme BEB et redémarrage uniquement quand le sémaphore S_{abort} est libre.

Nous proposons comme mesure de la vivacité le rapport

$$V = \frac{\text{nombre d'accès à la ressource globale}}{\text{nombre total de tentatives d'accès}}$$

Ce rapport représente la capacité des clients à accéder à la ressource lorsqu'il y a compétition. C'est une mesure de la facilité avec laquelle les clients obtiennent les sémaphores. On peut aussi considérer V comme l'efficacité de l'algorithme dans une situation donnée. Quand V est nul ou proche de 0 alors les clients sont en compétition perpétuelle et ne peuvent pas accéder à la ressource. On se trouve alors typiquement dans un cas où la vivacité 2(b) n'est pas assurée car l'algorithme ne se termine pas. Intuitivement, V décroît avec le nombre de transactions actives.

Nous avons représenté sur la figure 7.22 les variations du rapport V en fonction du nombre de transactions actives pour les cinq cas décrits ci-dessus. Pour obtenir ces courbes nous avons simulé le comportement des différents algorithmes. Le nombre de serveurs MMS (donc de sémaphores) est fixé à trois. Tous les clients demandent continuellement 100 fois tous les sémaphores. Au départ, le nombre de transactions actives est donc égal au nombre de clients. L'ordre de demande des sémaphores est déterminé de façon aléatoire et recalculé après chaque accès aux ressources. La temporisation D_1 est aussi choisie de façon aléatoire entre 0.5 et 5.5 secondes. Les temps de réponse de tous les services sont compris entre 60 ms et 70 ms. Les courbes de la figure 7.22 doivent être interprétées de la façon suivante : par exemple sur la courbe 1 pour 6 clients la valeur de V est 0.2. Cela signifie que sur l'ensemble des tentatives effectuées pour accéder à la ressource, 20 % ont réussi. Les 80 % restantes sont dues à des fins de temporisation ayant entraîné des redémarrages.

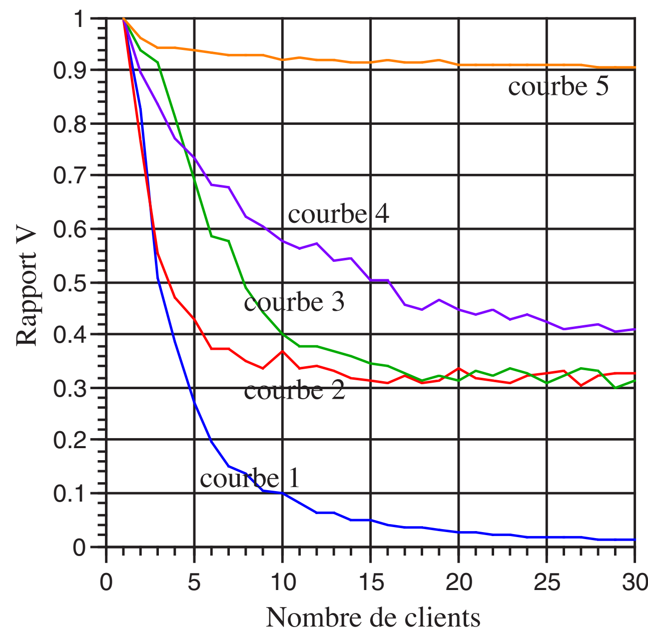


FIG. 7.22 - Variations de V en fonction du nombre de clients

Il n'est pas surprenant de constater que pour l'algorithme de base 7.A.9 (courbe 1) V décroît très fortement quand le nombre de clients augmente. La définition de l'algorithme amélioré 7.A.10 (courbe 2) avait pour but d'améliorer cette situation. On voit très nettement que le changement est important. Nos modifications ont donc bien permis d'améliorer le comportement de l'algorithme de base.

L'addition d'un délai aléatoire D_2 (compris entre 0 et 10 secondes) avant que la victime ne redémarre (courbe 3) améliore bien l'algorithme 7.A.10 mais pas de façon significative. A partir de 10 clients, les courbes 2 et 3 restent en effet très proches. En fait, le délai D_2 est plafonné à 10 secondes et n'évolue pas en fonction du nombre de transactions actives. Ainsi, à partir d'une certaine valeur l'effet du délai aléatoire D_2 s'atténue considérablement. En effet, à délai D_2 constant, plus le nombre de transactions actives augmente, plus les chances qu'une victime retrouve S_{abort} occupé sont grandes.

Une solution plus adaptée est donc de faire évoluer D_2 en fonction du nombre de transactions actives

(courbe 4). On obtient effectivement un meilleur taux de réussites V que pour les essais précédents. Enfin, étant donné la similitude entre nos algorithmes et les réseaux locaux comme Ethernet, nous avons adapté l'algorithme BEB à notre situation. La courbe 5 montre les résultats obtenus quand le délai D_2 est choisi de façon aléatoire entre 0 et 2^R où R est le nombre de redémarrages déjà subis par la transaction. Le rapport V de la courbe 5 est de loin le meilleur de tous ceux obtenus.

Les figures 7.23 et 7.24 montrent respectivement le temps de réponse moyen du système[11] et le temps de réponse moyen d'une transaction. Nous appelons temps de réponse moyen du système le temps nécessaire pour que les transactions de tous les clients terminent. La figure 7.25 montre le nombre moyen de redémarrages par transaction. On constate encore plus sur ces figures l'inefficacité de l'algorithme de base 7.A.9. Les résultats obtenus concordent avec les précédents et le cas 5 reste toujours le meilleur.

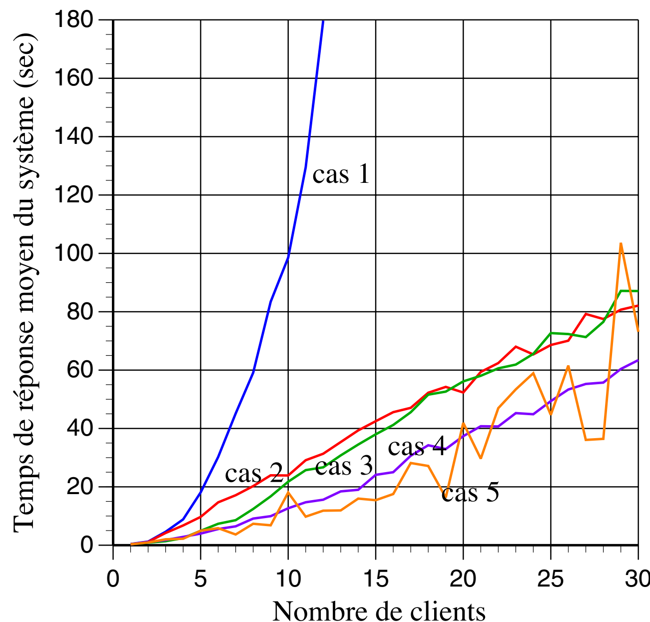


FIG. 7.23 - Temps de réponse moyen du système

En fait, une étude plus détaillée nous permet de nuancer ces résultats. Pour la courbe 5, le délai D_2 augmente très rapidement de sorte qu'une transaction peut être bloquée pendant un laps de temps très important. Dans un tel cas, les autres transactions ont beaucoup plus de chances de s'exécuter et peuvent même accéder à la section critique plusieurs fois de suite. On observe alors une occupation inégale de la section critique qui est monopolisée par certaines transactions. Le tableau 7.2 donne le nombre moyen d'accès aux ressources par transaction au moment où toutes les transactions ont pu accéder au moins une fois aux ressources. On constate que l'algorithme de base est le plus équitable (le plus proche de 1) alors que l'algorithme avec délais D_2 générés par BEB ne l'est pas du tout (bien que beaucoup plus efficace). Cela signifie en particulier que si les transactions qui se terminent n'étaient pas immédiatement redémarrées on pourrait observer des périodes pendant lesquelles tous les sémaphores sont libres et certaines transactions en attente d'expiration de D_2.

On observe d'ailleurs l'apparence inégale de la courbe des temps de réponse du cas 5 sur la figure 7.23. Ces inégalités proviennent de ce que certaines transactions doivent attendre un temps très long avant d'être redémarrées. Si ceci se produit alors que la plupart des transactions ont effectué tous leurs accès et ne s'exécutent plus, alors le temps de réponse moyen peut se retrouver considérablement augmenté.

[11] Le temps d'accès aux ressources n'est pas compté.

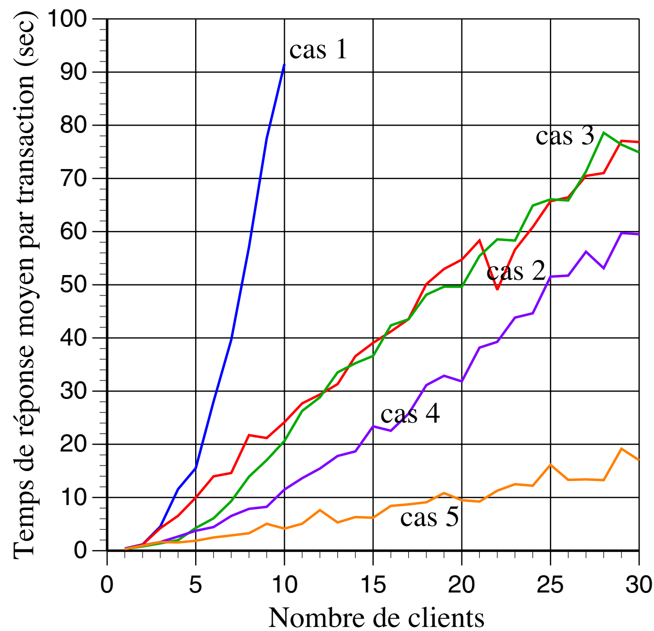


FIG. 7.24 - Temps de réponse moyen par transaction

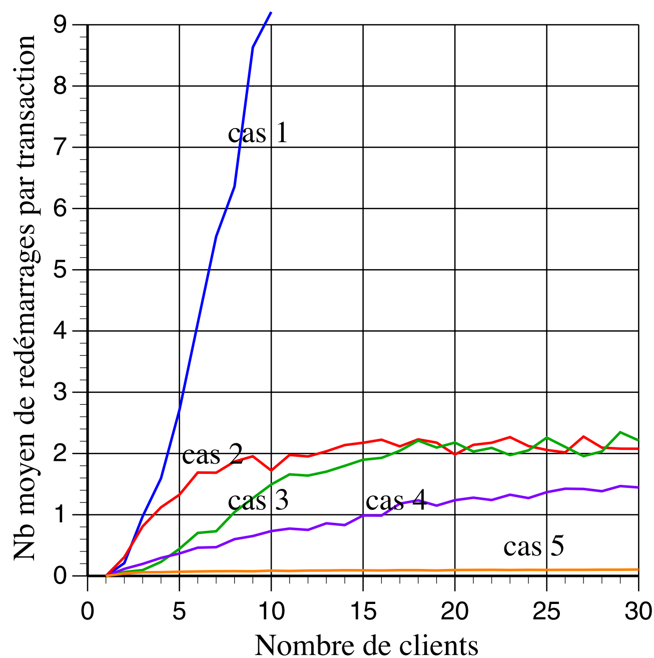


FIG. 7.25 - Nombre moyen de redémarrages par transaction

Tous les résultats obtenus dans cette section se situent dans le cadre d'une compétition intensive entre transactions. En effet, jusque là toutes les transactions cherchaient à acquérir tous les sémaphores. Nous donnons sur les figures 7.26, 7.27 et 7.28 la vivacité obtenue dans des cas où la compétition entre transactions est moins forte. Les transactions choisissent de façon aléatoire les sémaphores à acquérir parmi 100. Pour la figure 7.26, le nombre de sémaphores choisi est fixé à 3. Pour la figure 7.27 ce nombre est fixé à 5 et pour la figure 7.28 à 10.

On distingue très nettement sur la figure 7.28, l'importance d'avoir un délai D_2 qui s'adapte à la charge du système c'est-à-dire au nombre de clients en présence (cas 4 et cas 5). En l'absence d'un tel

	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5
Nb moyen d'accès	1.1	3.1	3.7	4.8	11.6

TAB. 7.2 - Nombre moyen d'accès aux ressources par transaction lorsque toutes les transactions y ont accédé au moins une fois.

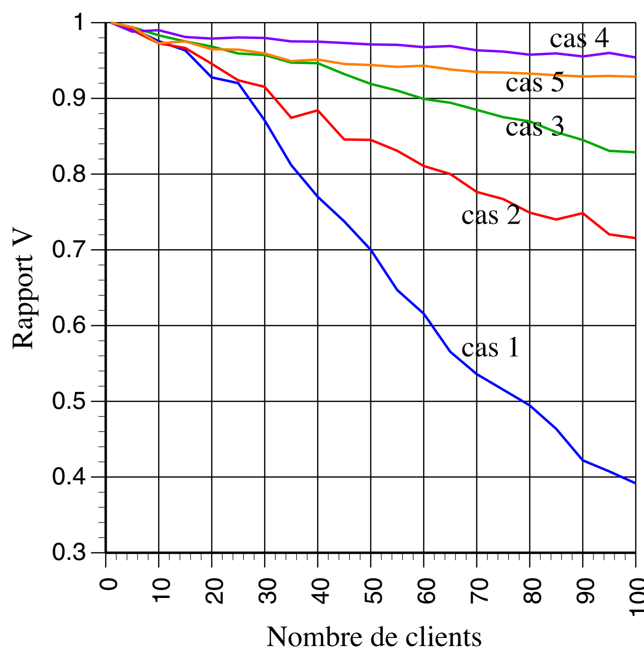


FIG. 7.26 - Variations de V en fonction du nombre de clients et pour 3 sémaphores

délai, la vivacité s'effondre rapidement (cas 1, 2 et 3).

Nombre de messages échangés

Le nombre de requêtes[12] MMS nécessaires pour exécuter complètement une transaction T est donné par les formules suivantes :

Dans le meilleur cas : $\boxed{N_{req} = 2M}$ C'est le nombre de requêtes nécessaires en l'absence de compétition (M `Take Control` et M `Relinquish Control`).

Dans les autres cas : $\boxed{N_{req} = 2M + \sum_{i=1}^{R}(2k_i + 1)}$ où R est le nombre de tentatives infructueuses avant l'accès à la ressource c'est-à-dire le nombre de redémarrages. k_i est le nombre de sémaphores détenus par la transaction lors de la i^{eme} tentative.

Dans le pire des cas : $\boxed{N_{req} = 2M(R+1) - R}$ Le cas le plus défavorable survient quand $k_i = M - 1$ pour tout i. Nous avons donc toujours $\boxed{N_{req} \leq 2M(R+1) - R}$ Remarquons que ce cas est extrêmement peu probable. Il faut en effet qu'à chaque expiration de temporisation de la transaction T il y ait une seule autre transaction T' détenant un seul sémaphore et que T détienne tous les sémaphores disponibles sauf 1.

[12] Comme nous comptons les requêtes, il faut multiplier ce nombre par deux pour avoir le nombre de messages ou PDUs MMS échangés (requête et réponse).

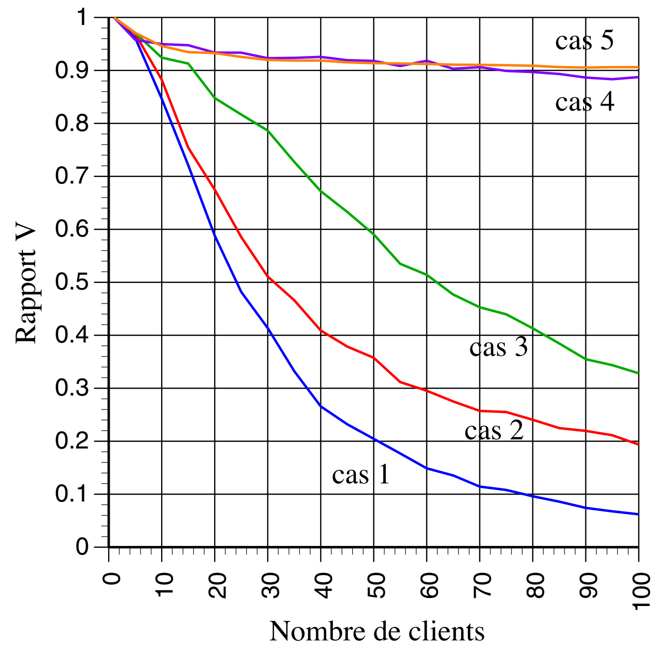


FIG. 7.27 - Variations de V en fonction du nombre de clients et pour 5 sémaphores

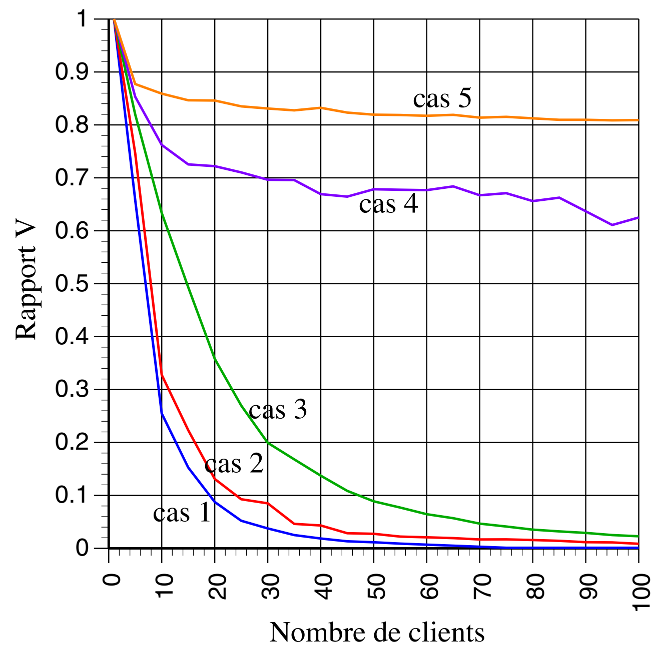


FIG. 7.28 - Variations de V en fonction du nombre de clients et pour 10 sémaphores

Il est possible d'obtenir une estimation du nombre moyen de redémarrages que subit une transaction donnée suite à des interblocages. Notons $\bar{R}$ ce nombre. Pour simplifier le problème, nous allons faire certaines hypothèses.

Hypothèse 1 *A tout instant, le système contient exactement N transactions actives. Dès qu'une transaction se termine elle est remplacée par une autre issue du même client.*

Hypothèse 2 *Chaque transaction cherche à acquérir K sémaphores. Le choix de ces K sémaphores parmi M se fait de façon aléatoire.*

L'hypothèse 1 implique que nous nous plaçons dans le cas défavorable de l'algorithme 7.A.9. L'algorithme 7.A.10 et ses dérivés obligent les victimes à acquérir S_{abort} pour pouvoir redémarrer. Les victimes n'étant pas actives, il peut y avoir moins de N transactions actives à un moment donné.

L'hypothèse 2, implique que tous les sémaphores ont les mêmes chances d'être acquis.

Notons p_i la probabilité qu'une transaction fasse partie d'un interblocage de taille[13] i. Pour tout $j \geq 1$, $p_{N+j} = 0$ puisqu'il n'y a pas plus de N transactions en cours. Par ailleurs, une transaction étant bien formée, elle ne s'interbloque pas avec elle-même donc $p_1 = 0$.

Nous allons encore faire l'hypothèse essentielle suivante.

Hypothèse 3 *En présence d'un interblocage de taille i, $i-1$ transactions redémarrent et une transaction réussit à acquérir ses sémaphores et termine.*

L'hypothèse 3 nous permet de déterminer la probabilité (notée α) pour qu'une transaction subisse un interblocage et redémarre. α est donnée par l'équation suivante :

$$\alpha = \sum_{j=2}^{N} \frac{j-1}{j} p_j \tag{7.5}$$

La probabilité pour qu'une transaction soit exécutée après avoir été redémarrée j fois est donnée par :

$$a_j = \alpha^j (1 - \sum_{j=2}^{N} p_j + \sum_{j=2}^{N} \frac{p_j}{j}) = \alpha^j (1-\alpha) \tag{7.6}$$

On en tire alors la valeur moyenne de R :

$$\bar{R} = \sum_{j=1}^{\infty} j a_j = \sum_{j=1}^{\infty} j \alpha^j (1-\alpha) \tag{7.7}$$

En factorisant $\alpha(1-\alpha)$ et en remarquant que $j\alpha^{j-1} = (\alpha^j)'$, on obtient :

$$\bar{R} = \alpha(1-\alpha) \frac{d}{d\alpha} \sum_{j=0}^{\infty} \alpha^j \tag{7.8}$$

Etant donné que $0 \leq \alpha < 1$, la série géométrique $\sum_{j=0}^{\infty} \alpha^j$ converge vers $\frac{1}{1-\alpha}$. Finalement on a :

$$\bar{R} = \alpha(1-\alpha) \underbrace{\frac{d}{d\alpha}(\frac{1}{1-\alpha})}_{\frac{1}{(1-\alpha)^2}} = \frac{\alpha}{1-\alpha} \tag{7.9}$$

[13] Un interblocage de taille i est un cycle d'attente comprenant i transactions.

Rappelons que $\bar{R}$ est le nombre moyen de redémarrages subits par une transaction suite à des interblocages. En fait, il faut également prendre en compte le nombre de redémarrages suite à des conflits qui ne sont pas des interblocages. On doit alors ajouter dans l'équation 7.5 un terme qui tienne compte des redémarrages suite à conflit. Notons W la probabilité qu'une transaction ait à attendre sur un sémaphore avant un redémarrage ou avant une exécution réussie. W prend donc en compte l'attente sur conflit qu'il y ait ou non interblocage. On obtient alors une nouvelle valeur de α :

$$\alpha \;=\; (\sum_{j=2}^{N} \frac{j-1}{j} p_j) + (W - \sum_{j=2}^{N} p_j)\beta \tag{7.10}$$

Le second terme est la probabilité qu'une transaction soit en attente sur un sémaphore mais pas dans un interblocage et soit redémarrée. β dénote la probabilité qu'une transaction en attente non interbloquée redémarre.

Dans ce cas, la probabilité a_j pour qu'une transaction soit exécutée après avoir été redémarrée j fois est :

$$a_j \;=\; \alpha^j (1 - W + \sum_{j=2}^{N} \frac{p_j}{j} + (W - \sum_{j=2}^{N} p_j)(1-\beta)) \;=\; \alpha^j (1-\alpha) \tag{7.11}$$

On calcule la nouvelle valeur de $\bar{R}$ de la même façon que précédemment. L'équation 7.9 reste donc adaptée, il suffit de remplacer α par la valeur donnée dans l'équation 7.10.

Il reste à exprimer α en fonction des valeurs K, M et N caractéristiques du système. Soit w la probabilité qu'une requête `Take Control` d'une transaction donnée doive attendre la libération d'un sémaphore. Nous allons utiliser les résultats décrits dans [GR93] pp. 428–429. Ces résultats nécessitent de faire l'approximation $KN \ll M$ i.e. la plupart des M sémaphores sont toujours libres. Dans notre cas, ceci s'applique essentiellement pour les résultats de la figure 7.26 et dans une moindre mesure de la figure 7.27. En considérant que chaque transaction détient environ $\frac{K}{2}$ sémaphores quand une requête `Take Control` se bloque sur un sémaphore pris, on a $w = \frac{K(N-1)}{2M}$. Comme chaque transaction effectue K requêtes `Take Control`, $W = 1 - (1-w)^K$. L'approximation $KN \ll M$ nous permet de poser $W \simeq Kw = \frac{(N-1)K^2}{2M}$.

La probabilité qu'une transaction T_1 soit dans un interblocage de taille 2 est la probabilité que T_1 soit en attente sur T_2 (i.e. W) et que T_2 soit en attente sur T_1 (i.e. $\frac{W}{N-1}$). On a donc :

$$p_2 \;=\; \frac{W^2}{N-1} \;=\; \frac{(N-1)K^4}{4M^2} \tag{7.12}$$

La probabilité qu'une transaction soit dans un interblocage de taille i est proportionnelle à W^i. Il est montré dans [TR91] et [GR93] qu'il est possible de négliger les interblocages de taille différente de 2 (toujours pour le cas où $KN \ll M$). En d'autres termes, $\forall i > 2,\ p_i \ll p_2$ donc on peut faire l'approximation $\forall i > 2,\ p_i = 0$.

On obtient ainsi une valeur approchée de α par l'expression :

$$\alpha \;\simeq\; \frac{p_2}{2} + (W - p_2)\beta \tag{7.13}$$

En supposant que la moitié des transactions rencontrant un conflit autre qu'un interblocages redémarrent (i.e. $\beta = \frac{1}{2}$), on a :

$$\alpha \;\simeq\; \frac{(N-1)K^2}{4M} \tag{7.14}$$

On en tire la valeur approchée de $\bar{R}$:

$$\bar{R} \simeq \frac{(N-1)K^2}{4M-(N-1)K^2} \tag{7.15}$$

La figure 7.29 montre la différence entre les valeurs de $\bar{R}$ obtenues en théorie et par simulation pour $\beta = 0.032$ et dans les conditions de la figure 7.26. Les courbes concordent pour un nombre de clients inférieur à 16 environ. On constate clairement ensuite une divergence dès que l'approximation $KN \ll M$ n'est plus valable.

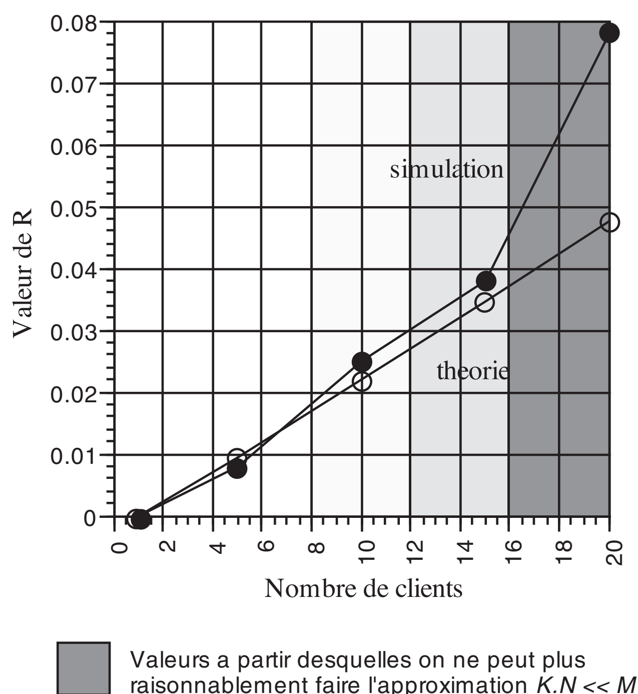


FIG. 7.29 - Différences entre les valeurs de $\bar{R}$ obtenues en théorie et par simulation

Pour établir les différentes équations ci-dessus, nous avons dû faire un certain nombre d'hypothèses et d'approximations qui ne nous permettent pas d'obtenir une valeur satisfaisante de $\bar{R}$ dans le cas où la compétition est forte. Toutefois, nous ne chercherons pas à estimer de façon plus précise la valeur de $\bar{R}$ pour ne pas nous éloigner de notre sujet principal qui est l'étude des systèmes dans l'environnement MMS. Nous avons voulu ramener le problème de l'allocation de ressources dans l'environnement MMS à celui identique de l'allocation de ressources dans les bases de données réparties avec la méthode 2PL. Dans le domaine des bases de données il existe de nombreuses études sur les performances de la méthode 2PL que nous pouvons avantageusement appliquer ici. Ces études permettent d'obtenir sous certaines approximations des résultats tels que ceux que nous venons d'étudier : la probabilité qu'une transaction soit interbloquée, le nombre moyen de transactions impliquées dans un interblocage, le nombre de transactions redémarrées pour résoudre un interblocage, etc (voir par exemple [JTK89], [SL90] et [TR91]). Ces résultats sont souvent obtenus au prix d'hypothèses ne reflétant pas toujours la réalité du système considéré et font donc toujours l'objet de recherches pour obtenir des modèles plus conformes à la réalité [BHG87] p. 87.

7.4.3.3 Transactions urgentes

Un des avantages de l'algorithme 7.A.10 est qu'il permet de servir les transactions urgentes. Une transaction dont les priorités des `Take Control` sont élevées et qui comporte de long temps d'attente D_1

s'exécutera avant les autres. En fait, on peut donner à une telle transaction la priorité maximum et des temps d'attente D_1 infinis[14]. Elle ne subira ainsi jamais de redémarrage. Pour éviter les interblocages, il faut toutefois s'assurer que c'est la seule transaction urgente active. Nous ne nous étendrons pas sur la façon d'obtenir cette garantie. Notons toutefois qu'une solution simple consiste à utiliser un sémaphore à un jeton situé dans un serveur MMS connu de tous les clients. Le jeton de ce sémaphore doit appartenir à tout client qui désire démarrer une transaction urgente. Il est libéré à la fin de la transaction.

7.4.3.4 Conclusion sur ces premiers algorithmes

Nous avons défini des algorithmes adaptés à MMS dans le but d'assurer l'allocation cohérente de ressources réparties. Ces algorithmes résolvent les problèmes d'interblocages. Ils utilisent des techniques connues dans les domaines des bases de données (2PL) et cherchent à mimer le comportement de réseaux locaux comme Ethernet. Leurs avantages principaux sont les suivants :

- Ils sont parfaitement adaptés à MMS et ne nécessitent aucune modification des serveurs ou du protocole MMS. Ils sont donc de niveau de modification 1 et s'adaptent à tout système MMS supportant la gestion de sémaphores.

- Ces algorithmes sont réalisés de telle façon que les clients soient totalement ignorants de l'existence des autres clients. Par conséquent des clients peuvent être rajoutés ou retirés à tout moment. Ils n'ont pas besoin de s'annoncer, de s'insérer dans un anneau ou de passer par une phase de reconfiguration. Ceci est une caractéristique importante car comme le souligne Grant dans [Gra92], l'ajout ou le retrait dynamique de stations et d'applications sans interrompre le fonctionnement normal du système est une composante intégrale des systèmes informatiques industriels.

- Nous n'avons pu faire la différence entre la consultation ou la modification des ressources car MMS ne permet pas l'acquisition des sémaphores en modes partagé ou exclusif. Nous avons défini à la section 7.2.2.5 une extension aux sémaphores MMS qui permet au client de préciser le mode d'acquisition d'un sémaphore. Nos algorithmes restent adaptés à cette extension. Ils ne nécessitent aucune modification et s'appliquent de façon identique. Cette extension doit permettre une réduction des interblocages et des attentes dues aux conflits [GR93] p. 429.

Mais nous devons aussi souligner les inconvénients suivants :

- Ces algorithmes sont en partie basés sur des délais aléatoires et sont par conséquent non déterministes. Ils ne peuvent être utilisés avec des applications ayant des contraintes de temps strictes.

- Ces algorithmes ne sont pas efficaces dans des systèmes où la compétition est importante et où le nombre de requêtes d'accès aux mêmes ressources est élevé.

- La détection d'interblocage par temporisation n'est pas adaptée aux systèmes dont la charge est élevée [CP85]. Une fin de temporisation ne garantit pas un interblocage et peut contribuer à augmenter inutilement la charge du système en redémarrant une transaction non interbloquée.

[14] C'est-à-dire ne jamais utiliser l'option `Acceptable Delay` pour cette transaction.

7.4.4 Détection des interblocages : la méthode des sondes

Nous présentons dans cette section un autre algorithme dont le but est également d'assurer l'allocation cohérente de ressources réparties avec MMS mais qui permet cette fois une détection plus explicite des interblocages.

Le modèle, notations et hypothèses sont les mêmes que dans les sections précédentes. Notre algorithme utilise le principe des *sondes* ("probes") initialement dû à Chandy *et al.* [CMH83]. Différentes versions de l'algorithme de Chandy *et al.* ont été proposées pour améliorer ses performances ou pour l'adapter à des conditions particulières comme la tolérance aux fautes [SN85], [CKST89], [KS91] et [LM93]. Le principe des sondes est le suivant. Toute transaction bloquée sur un sémaphore peut émettre dans le réseau un message spécial appelé *sonde*. La transaction ou le client qui décide d'émettre la sonde est appelée *initiateur*. La sonde circule depuis les transactions bloquées vers les transactions qui détiennent un sémaphore (et vice-versa). Etant donné qu'un interblocage crée un cycle de dépendance entre les transactions, on détecte un interblocage dès qu'une sonde retourne à son initiateur. Avec MMS, la sonde est représentée par les services d'événements `Trigger Event` et `Event Notification`.

Dans l'algorithme de Chandy *et al.*, les transactions sont effectuées par des processus et chaque processus communique directement avec son contrôleur local. Les contrôleurs gèrent les verrous (sémaphores) utilisés par les transactions. Ces contrôleurs ont une autonomie propre. Ils doivent communiquer entre eux et sont en grande partie responsables de la résolution des interblocages. Dans MMS, les contrôleurs sont les serveurs. Rappelons que selon les hypothèses 4 et 5 de la section 7.4.1, les serveurs ne peuvent pas communiquer entre eux ni décider d'actions à entreprendre autres que celles définies par la norme MMS [ISO90a]. Nous ne pouvons donc pas faire jouer aux serveurs MMS le même rôle que les contrôleurs de verrous de [CMH83].

Dans notre algorithme nous nous servons des serveurs comme de simples relais pour propager la sonde. Sur chaque serveur i $(1 \leq i \leq M)$ les objets suivants sont définis :

- un sémaphore S_i;
- l'objet EC associé au sémaphore noté EC_i[15];
- un objet EA noté EA_i contenant le service `Get Event Condition Attributes` s'appliquant à EC_i.

Notre algorithme est présenté sur la figure 7.A.11. Chaque client MMS est identifié par un numéro unique[16]. Un client k $(1 \leq k \leq N)$ tente d'acquérir séquentiellement les sémaphores dont il a besoin (ligne 15). Comme pour les algorithmes précédents, les requêtes `Take Control` utilisent le paramètre `Acceptable Delay` qui déterminent le temps D pendant lequel la transaction attendra l'obtention du sémaphore. Pour la première de ces requêtes il n'est pas nécessaire d'utiliser ce paramètre puisque la transaction ne détient pas de sémaphore et ne peut donc pas faire partie d'un interblocage (ligne 13).

Quand un sémaphore S_i est acquis avant expiration du temporisateur `Acceptable Delay`, une réponse positive est retournée au client (ligne 19). Ce client définit alors un EE noté EE_i liant EC_i et EA_i. Le client précise ainsi qu'il est le détenteur du sémaphore S_i. Cet EE sera utilisé pour recevoir des sondes éventuelles. Le client continu ensuite l'exécution de l'algorithme est tente d'acquérir d'autres sémaphores ou accède aux ressources désirées (ligne 40). A la fin de la transaction, tous les sémaphores sont libérés et tous les EEs créés sont détruits (lignes 42–45).

[15] Rappelons qu'il est spécifié par la norme MMS qu'à tout sémaphore MMS est associé un objet EC. On pourrait très bien utiliser un autre EC mais l'exploitation de l'EC existant nous permet une économie d'objets MMS et de messages (pour créer un autre EC).

[16] Nous supposons que les numéros d'identification des clients sont pré-établis.

```
 1: local variables
 2:    probeReceived, processNext   : booleans
 3:    UnlockedSemaphores           : set of integers
 4:    i, j, D, initiatorID         : integers /* initialisés localement */
 5:
 6: begin
 7:    UnlockedSemaphores ← {1, ..., K}
 8:    while UnlockedSemaphores ≠ ∅ loop
 9:       processNext    ← FALSE
10:       probeReceived  ← FALSE
11:       i              ← Min(UnlockedSemaphores)
12:       if Card(UnlockedSemaphores) = K then
13:          MMS_Async_Take_Control(S_i,normalPriority) /* Non bloquant, pas de temporisation */
14:       else
15:          MMS_Async_Take_Control(S_i, D, normalPriority) /* Non bloquant */
16:       end if
17:
18:       repeat
19:          when POSITIVE MMS_Take_Control_Response(S_i, ...) received
20:             Remove i from UnlockedSemaphores
21:             processNext ← TRUE
22:             MMS_Define_Event_Enrollment(EE_i attached to EC_i and EA_i)
23:
24:          when NEGATIVE MMS_Take_Control_Response(S_i, ...) received
25:             MMS_Trigger_Event(EC_i, clientID)
26:             MMS_Async_Take_Control(S_i, lowPriority) /* Non bloquant, pas de temporisation */
27:
28:          when MMS_Event_Notification(EC_j, initiatorID) received
29:             if clientID = initiatorID or probeReceived then
30:                probeReceived ← TRUE
31:                MMS_Delete_Event_Enrollment(EE_j)
32:                MMS_Relinquish_Control(S_j)
33:                Add j to UnlockedSemaphores
34:             else
35:                MMS_Trigger_Event(EC_i, initiatorID)
36:             end if
37:       until processNext
38:    end while
39:
40:    /* Accès aux ressources ici */
41:
42:    for i in (1 .. K) loop
43:       MMS_Delete_Event_Enrollment(EE_i)
44:       MMS_Relinquish_Control(S_i)
45:    end for
46: end
```

Algorithme 7.A.11: Détection des interblocages par propagation d'une sonde

Par contre, quand un temporisateur `Acceptable Delay` expire sur l'attente d'un sémaphore S_i une réponse négative est retournée au client (ligne 24). Cette réponse déclenche l'émission d'une sonde. Un client k émet une sonde avec le service `Trigger Event(`EC_i`, ` k`)` appliqué à l'objet EC_i. Le paramètre `Priority` de la requête `Trigger Event` contient la valeur k, c'est-à-dire l'identité du client k (ligne 25). Avant le déclenchement de l'événement EC_i dans le serveur, cette priorité k est affectée à l'objet EC_i. L'événement EC_i est ensuite déclenché et ce qui provoque l'envoi d'une notification d'événement au client l détenant le sémaphore S_i (ligne 28). Cette notification contient l'identité du client k initiateur de la sonde. En effet, le déclenchement de l'événement EC_i implique l'exécution de l'action EA_i qui permet d'obtenir la priorité de EC_i. Nous notons *sonde(*EC_i*, k)* la notification provenant de l'EC EC_i et transportant l'identité k. Quand un client l reçoit une sonde *sonde(*EC_i*, k)* plusieurs cas peuvent se présenter :

1. Le client l est en attente d'un sémaphore S_j et détient bien S_i :
   (a) Le client l est l'initiateur de la sonde ($k = l$). Dans ce cas, un interblocage est identifié. Le sémaphore S_i détenu par le client l est libéré. EE_i est détruit puisque l n'est plus le détenteur de S_i et la sonde n'est pas retransmise (lignes 30–33).
   (b) Le client l n'est pas l'initiateur de la sonde ($k \neq l$) mais a déjà émis et reçu sa sonde en retour. Dans ce cas, le sémaphore S_i détenu par le client l est libéré. EE_i est détruit puisque l n'est plus le détenteur de S_i et la sonde n'est pas retransmise (lignes 30–33).
   (c) Le client l n'est pas l'initiateur de la sonde ($k \neq l$) et n'a jusqu'alors pas reçu de sonde en retour (qu'il en ait émis une ou non). Dans ce cas, la sonde est simplement retransmise en déclenchant l'événement correspondant au sémaphore sur lequel le client l est en attente (requête `Trigger Event(`EC_j`, ` k`)`) (ligne 35).
2. Le client l est en attente d'un sémaphore S_j mais ne détient pas S_i : ce cas anormal peut se produire quand une requête `Trigger Event(`EC_i`)` arrive au serveur i alors que S_i est pris par le client l mais que la notification parvienne au client l après qu'il ait libéré S_i. Dans ce cas, la sonde est ignorée et n'est pas retransmise. Dans la suite, on qualifie une telle sonde de *non pertinente*[17].
3. Le client l a obtenu tous ses sémaphores et est en train d'accéder à ses ressources ou de terminer la transaction. Dans ce cas, toutes les notifications qui lui parviennent sont ignorées (lignes supérieures à 38). En fait, on pourrait détruire les EEs dès la ligne 39 avant d'accéder aux ressources. Le déclenchement d'un événement sur l'un des ECs associés aux sémaphores détenus par la transaction n'aurait alors aucun effet.

Le cas 1(c) ci-dessus permet à la sonde de se propager de site en site comme l'illustre la figure 7.30. Dans le cas 1(a), un client qui reçoit sa sonde en retour ne doit libérer que le sémaphore identifié par la sonde[18]. Ceci est suffisant pour résoudre l'interblocage et ne génère pas tous les messages d'annulation nécessaires à l'algorithme 7.A.10. Cependant, ce client peut encore détenir des sémaphores nécessaires à d'autres transactions. Il doit donc s'attendre à recevoir d'autres sondes et libérer tous les sémaphores identifiés par ces sondes qu'il y ait interblocage ou non (cas 1(b)). Les conflits et les interblocages sont les deux paramètres qui affectent les performances générales des systèmes où les verrous sont utilisés pour protéger les ressources. Les cas 1(a) et 1(c) permettent de détecter et de résoudre les interblocages. Le cas 1(b) permet de diminuer les conflits.

[17] La sonde *sonde(*EC_i*, k)* est dite *pertinente* si le client qui la reçoit détient S_i.

[18] Nous pouvons éviter d'annuler toute la transaction car contrairement à la méthode 2PL classique aucune ressource n'est ici accédée tant que **tous** les sémaphores ne sont pas pris.

Il est important de noter que nous appelons *résolution d'un interblocage* le fait que le cycle d'attente initial formé entre les transactions soit détruit ou ait simplement changé d'apparence. Les cas suivants constituent une résolution de l'interblocage initial :

- il n'y a plus de cycle entre les transactions;
- le cycle entre les transactions s'est agrandi ou réduit;
- le cycle entre les transactions est le même mais formé par un ensemble de sémaphores différent de celui qui formait le cycle avant la résolution.

La résolution d'un interblocage peut donc impliquer la formation d'un autre interblocage. Il est alors possible qu'une sonde émise lors de l'interblocage initial circule aussi dans l'interblocage suivant. Une sonde peut également être émise à un moment où il n'y a pas d'interblocage. Un interblocage peut ensuite se former après l'émission de cette sonde. Il se peut alors que la sonde circule dans ce nouvel interblocage. Une telle sonde est appelée *sonde résiduelle*.

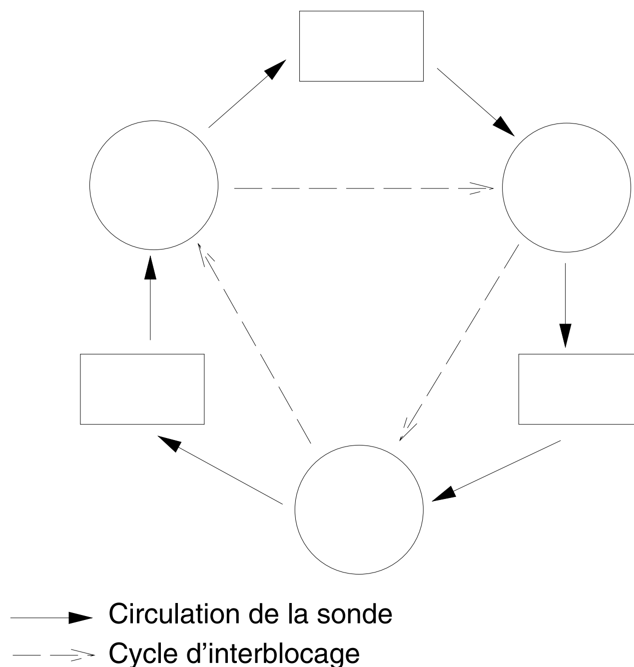


FIG. 7.30 - Circulation de la sonde entre clients et serveurs

Un client victime d'une fin de temporisation sur un sémaphore S_{abort} se met en attente de S_{abort} et ne peut poursuivre qu'après obtention de ce sémaphore (ligne 26). Pour les mêmes raisons que celles de l'algorithme 7.A.10, cette attente se fait avec une priorité basse. L'obtention de S_{abort} est donc le signal qui permet à ce client d'acquérir d'autres sémaphores. **Pendant cette attente, le client ne peut que libérer des sémaphores. Il n'envoie pas de sondes et ne tente pas d'obtenir d'autres sémaphores.**

7.4.4.1 Sûreté et vivacité

La validité de cet algorithme a été démontrée avec l'outil SPIN. Le code PROMELA de l'algorithme apparaît dans l'annexe E dans le cas de 3 clients et 3 serveurs.

La propriété de sûreté est assurée pour les mêmes raisons que celles des algorithmes 7.A.9 et 7.A.10.

Pour démontrer la vivacité 2(a), il faut montrer que tout interblocage est résolu. Nos tests et simulations avec l'outil SPIN dans le cas de 3 clients et 3 serveurs n'ont pas décelé d'interblocages non résolus. Dans le cas général, nous laissons cette démonstration de la vivacité 2(a) sous forme de conjecture.

Pour démontrer la vivacité 2(b), il faut s'assurer que l'algorithme ne rentre pas dans une situation où au moins une des transactions ne progresse pas. Ceci implique que les deux points suivants doivent être exclus :

1. au moins une transaction est bloquée en attente d'un message qui n'arrive jamais;
2. au moins une transaction exécute l'algorithme indéfiniment sans jamais accéder aux ressources demandées.

Nous ne considérons pas ici les pertes de messages dues à des fautes d'un des composants de notre système, celui-ci étant considéré comme fiable. Une transaction T reste donc bloquée indéfiniment que si elle ne reçoit jamais de réponse à une requête `Take Control`. Si de nouvelles transactions sont constamment démarrées alors il n'est pas possible de garantir le point 1 ci-dessus. En effet, il est toujours possible de démarrer une transaction qui demande justement le sémaphore S_{abort} attendu par T. Comme l'attente sur S_{abort} se fait avec une priorité basse, T ne peut accéder à S_{abort}.

En ce qui concerne le point 2, la vivacité 2(b) ne peut non plus être satisfaite pour les mêmes raisons que celles pour les algorithmes 7.A.9 et 7.A.10. On peut effectivement toujours positionner les temporisateurs à des combinaisons de valeurs telles qu'une transaction reste toujours dans un cycle d'interblocage. L'utilisation de délais aléatoires doit assurer une meilleure vivacité comme pour les premiers algorithmes.

7.4.4.2 Quelques remarques

Utilisation des priorités MMS

L'utilisation de la priorité d'un EC est la seule solution simple permettant à la sonde de transporter l'identité de son initiateur. Nous avons donc détourné l'usage initial de la priorité d'un EC (l'ordonnancement du traitement des événements) au profit de notre algorithme. Le principal inconvénient est de limiter le nombre de clients au nombre maximum de priorités c'est-à-dire à 128.

Notons aussi que notre algorithme suppose que le déclenchement de l'événement EC_i et l'exécution de l'action EA_i correspondante se font de façon **atomique**. En effet, si plusieurs transactions sont bloquées sur le même sémaphore S_i et émettent une sonde au même moment, il faut garantir que la priorité de l'EC capturée par l'action EA_i est bien celle écrite dans l'EC lors du déclenchement de l'événement par `Trigger Event`. Prenons le cas de deux transactions T_a et T_b initiées par les clients a et b respectivement. Ces deux transactions sont bloquées sur le sémaphore S_i et émettent une sonde au même moment. En l'absence d'exécution atomique de l'action et du déclenchement de l'événement EC_i les opérations peuvent se dérouler dans l'ordre suivant dans le serveur :

1. réception de `Trigger Event(`EC_i`, `a`)`
2. réception de `Trigger Event(`EC_i`, `b`)`
3. priorité de $EC_i := a$
4. priorité de $EC_i := b$

5. exécution de l'action déclenchée par a : lecture priorité de EC_i
6. envoi de `EventNotification` pour a (avec priorité de $EC_i = b$)
7. exécution de l'action déclenchée par b : lecture priorité de EC_i
8. envoi de `EventNotification` pour b (avec priorité de $EC_i = b$)

Le client détenant le sémaphore S_i reçoit deux sondes qui semblent provenir d'une même transaction (ici b). Notre hypothèse d'atomicité s'avère donc nécessaire pour garantir le bon fonctionnement de l'algorithme 7.A.11. Cette exigence sur l'atomicité ne fait pas partie des spécifications MMS. L'architecture des serveurs étudiée au chapitre 4 ne l'impose donc pas. On peut toutefois facilement l'adapter pour satisfaire cette hypothèse.

Le temps d'attente avant d'émettre une sonde

La détermination du temps D avant qu'une transaction n'émette la sonde n'est pas aussi critique que pour les algorithmes 7.A.9 et 7.A.10. Il ne s'agit pas ici de supposer qu'il y a un interblocage et d'annuler une transaction mais de démarrer le processus de détection. On pourrait donc choisir d'effectuer la détection d'interblocage de façon régulière chaque fois qu'il y a conflit en fixant le temps D à une valeur constante. Chandy *et al.* notent à ce propos dans [CMH83] que les temporisations sont utiles pour réduire le nombre de sondes émises dans le réseau. Toutefois il faut toujours faire un compromis entre un temps trop court qui risque de générer trop de sondes et un temps trop long qui risque de ralentir la détection des interblocages et donc le temps de réponse des transactions. Il faut aussi favoriser au mieux la vivacité 2(b) comme dans le cas des algorithmes 7.A.9 et 7.A.10 et éviter que les temporisateurs de transactions différentes n'expirent toujours en même temps. Nous conservons donc l'idée de la génération aléatoire des temps D. On peut rajouter à ce temps une valeur constante qui fixe la durée minimale avant qu'une sonde ne puisse être émise.

Nous préférons la méthode par temporisation à celle utilisée dans [SN85] et [CKST89] qui consiste à ordonner les transactions avec des priorités et à envoyer une sonde dès qu'une transaction est en conflit avec une autre transaction de priorité plus basse. Si ce dernier cas était adopté, il faudrait avec MMS acquérir la priorité des transactions dès qu'il y a conflit. Ceci peut se faire avec le service `Report Semaphore Entry Status` mais augmenterait de façon significative le nombre de messages et le nombre de sondes.

7.4.4.3 Comparaison entre les algorithmes 7.A.10 et 7.A.11

Nous allons calculer le coût, en terme de nombre de messages, de la résolution d'un interblocage pour les deux algorithmes 7.A.10 et 7.A.11. Un service MMS compte pour deux messages ou PDUs MMS (requête et réponse).

Prenons deux transactions T_1 et T_2 qui nécessitent d'acquérir respectivement K_1 et K_2 sémaphores. Il y a K sémaphores communs entre ces deux transactions. Supposons qu'un interblocage se forme entre les deux transactions alors que T_1 détient k_1 sémaphores dont k sémaphores communs avec T_2 ($2 \leq k \leq K - 1$ et $2 \leq k_1 \leq K_1 - 1$). La temporisation de T_1 est la première à expirer. Pour les deux algorithmes, T_1 ne peut alors plus acquérir de sémaphores tant que T_2 ne s'est pas terminée.

Coût engendré par l'algorithme 7.A.10 : $\boxed{C_1 = 4k_1}$ Il y a de la part de T_1, k_1 requêtes de libération `Relinquish Control` et k_1 requêtes `Take Control` pour retrouver le même nombre de sémaphores qu'avant annulation.

Coût engendré par l'algorithme 7.A.11 : $\boxed{C_2 = 11k + 3}$ La propagation de la sonde envoyée par T_1 implique 2 `Trigger Event` et 2 `Event Notification`. La libération du sémaphore identifié par cette sonde provoque 1 `Relinquish Control` et 1 `Delete Event Enrollment`. La libération des $k-1$ sémaphores communs restant nécessitent $k-1$ `Relinquish Control` et autant de `Delete Event Enrollment`. Ces libérations se font suite à la réception de $k-1$ sondes non retransmises soit $k-1$ `Trigger Event` et $k-1$ `Event Notification`. T_1 doit ensuite reprendre les k sémaphores libérés soit k `Take Control` et k `Define Event Enrollment`.

L'algorithme 7.A.10 est donc supérieur à l'algorithme 7.A.11 si $C_1 < C_2$ i.e. si $\boxed{k_1 < \frac{11k+3}{4}}$ Pour que le coût d'utilisation de l'algorithme 7.A.11 soit moindre que celui de l'algorithme 7.A.10 dans le cas de deux transactions il faut donc que le nombre total de sémaphores détenus par la victime soit un peu moins de 3 fois supérieur au nombre de sémaphores communs détenus. Ceci tend à montrer que l'intérêt de l'algorithme 7.A.11 réside dans les systèmes où les ressources sont relativement peu partagées par les transactions, et où les transactions sont plutôt longues.

Toutefois ce résultat peut difficilement se généraliser dans le cas où il existe plus de deux transactions actives. En effet, il est alors probable que le nombre moyen de redémarrages d'une transaction dans le cas de l'algorithme 7.A.10 soit plus grand que celui d'une transaction exécutant l'algorithme 7.A.11.

7.4.4.4 Conclusion sur la méthode des sondes

Nous avons présenté un algorithme de détection/résolution des interblocages basé sur MMS et utilisant la technique dite des sondes. Comme pour les précédents algorithmes, nous n'avons introduit aucune modification des serveurs ou du protocole MMS de sorte que l'algorithme 7.A.11 est parfaitement adapté à MMS. Cet algorithme peut donc être utilisé avec tout système MMS supportant la gestion de sémaphores et d'événements. Nous avons toutefois dû faire l'hypothèse d'atomicité du déclenchement d'un événement et du traitement de l'action correspondante pour assurer un fonctionnement correct de notre algorithme. A l'heure actuelle, le manque de produits MMS implantant les événements MMS et l'inexistence d'applications utilisant pleinement les fonctionnalités des événements MMS ne nous permettent pas de dire si cette hypothèse est réaliste.

Cette étude montre que malgré les limites de MMS dans le domaine de la gestion de ressources réparties il est possible de construire des algorithmes transactionnels entièrement basés sur MMS. Ces algorithmes ne font pas appel à des outils différents de MMS comme c'est le cas par exemple dans [DE95] qui utilise la norme TP [ISO92a].

7.5 Conclusion

Dans ce chapitre, nous avons proposé un ensemble d'outils optimisés et testés pour certains d'entre-eux permettant une meilleure intégration des applications réparties dans l'environnement MMS. Nous avons montré comment les problèmes classiques tels que rendez-vous, lecteurs/rédacteurs ou problème des philosophes se résolvent avec MMS. Certaines extensions à la norme ont été proposées pour optimiser ces solutions.

Nous avons montré qu'il était facile d'étendre la norme MMS de façon à autoriser une exécution répartie des services et actions événementielles.

En fin de chapitre, des algorithmes assurant le contrôle de la concurrence avec MMS ont fait l'objet d'une étude détaillée. Ces algorithmes s'inspirent des techniques trouvées dans les bases de données et les réseaux locaux de communication comme Ethernet. Ils permettent de garantir un accès sérialisé

aux données situées sur les serveurs MMS à la manière des transactions dans les bases de données. Ces algorithmes sont exempts d'interblocages.

Le tableau 7.3 fournit un récapitulatif des algorithmes, solutions et extensions proposées dans ce chapitre. Pour être complet, nous y ajoutons les extensions et solutions proposées dans les autres chapitres de cette thèse.

Solution ou extension	**Description**	**Niveau de modification**
Rendez-vous MMS simples	Réalisation de rendez-vous entre clients MMS en utilisant les sémaphores et les événements.	1
Rendez-vous MMS multiples	Réalisation de rendez-vous multiples entre clients MMS.	1
Lecteurs/rédacteurs	Deux solutions MMS au problème des lecteurs/rédacteurs.	1
Problème des philosophes	Trois solutions MMS et XED au problème des philosophes.	1 (2 pour XED)
Modes partagé et exclusif pour les sémaphores	Extension du sémaphore MMS pour autoriser la prise de contrôle en mode partagé ou exclusif	4
Temps de validité d'une action	Association d'un temps d'exécution au plus tard à une action événementielle.	3 ou 4
Exécution d'une action distante	Délocalisation de l'exécution d'une action événementielle sur un serveur secondaire.	2 ou 4
Exécution d'une requête distante	Délocalisation de l'exécution d'une requêtes de service sur un serveur secondaire.	2 ou 4
Contrôle de la concurrence (temporisation)	Algorithme de contrôle de la concurrence d'accès à des ressources MMS réparties sur différents serveurs. Interblocages résolus par temporisation.	1
Contrôle de la concurrence (sondes)	Algorithme de contrôle de la concurrence d'accès à des ressources MMS réparties sur différents serveurs. Interblocages détectés par la méthode des sondes.	1
XED	Extension à la détection d'événements permettant le déclenchement d'un événement lors de la réalisation d'une expression prédicat fournie par l'utilisateur.	2
XES	Extension permettant de fournir des priorités aux requêtes de service MMS pour ordonner le traitement de ces requêtes dans les serveurs.	2
Modificateurs et temps-réel	Exploitation du modificateur `Attach To Semaphore` pour contrôler et ordonnancer l'exécution des requêtes de service MMS.	2 ou 3

TAB. 7.3 - Tableau récapitulatif des solutions et extensions proposées

Chapitre 8

Conclusion

Les thèmes directeurs de cette thèse ont donc été les suivants :

– l'**exploration** détaillée de la norme MMS,

– l'**exploitation** au mieux de ses possibilités et

– l'**amélioration** significative de certaines de ses fonctionnalités.

Il nous est apparu important d'effectuer une analyse détaillée de certains aspects de la norme MMS pour pouvoir comprendre avec précision quelles sont les possibilités offertes par celle-ci dans son état actuel. La complexité de cette norme et la méconnaissance de certaines fonctionnalités telles que les événements et les sémaphores n'ont pas favorisé l'adoption de MMS dans les applications industrielles actuelles ni l'exploitation significative de ses capacités.

Cette thèse représente un des premiers efforts effectués en vue d'une compréhension et d'une intégration complète de la plupart des fonctionnalités offertes par la norme. Nous avons pu constater que MMS introduit de réelles limitations soit par manque de précision, soit par excès de liberté laissée aux concepteurs, soit par l'absence de fonctions répondant mieux aux besoins des applications utilisateurs malgré l'apparente simplicité de leur intégration à MMS. Cependant, il nous est aussi apparu qu'une exploitation poussée de la norme ouvre des horizons peu connus et peu explorés mais qui permettent souvent de résoudre de nombreux problèmes.

8.1 Principales contributions

Nous récapitulons ici les principales contributions de cette thèse :

1. **Analyse détaillée des sémaphores MMS.** Nous avons cherché à ramener les sémaphores MMS à des concepts connus tels que les sémaphores de Dijkstra. Ceci nous a permis d'identifier les problèmes et limitations liés au concept de sémaphore MMS. En particulier, il est apparu qu'il n'est pas possible d'utiliser les sémaphores MMS pour effectuer de la signalisation entre clients de façon classique.

2. **Analyse détaillée des événements MMS.** Là encore, nous avons comparé les événements MMS à d'autres types d'événements rencontrés dans différents systèmes. Avec cette comparaison nous avons fait apparaître les limitations liées au concept d'événement MMS. Ces deux analyses (sémaphores et événements) sont un tremplin pour pouvoir d'une part exploiter en détail toutes

les possibilités offertes par ces mécanismes et pour proposer d'autre part des extensions qui optimisent leur comportement et leur utilité ou encore comblent certaines lacunes.

3. **Proposition d'une architecture générique des serveurs** MMS intégrant l'ensemble des fonctionnalités d'un serveur. Avec cette architecture, nous pensons avoir établi une vision claire de la structure d'un serveur MMS et directement exploitable pour des implantations. Notre architecture se veut volontairement indépendante de tout support matériel, langage ou système d'exploitation. Nous avons effectué une implantation d'un serveur MMS supportant la quasi-totalité des services et fonctions proposés par la norme.

4. **Proposition d'une extension à la détection d'événements** MMS. Cette extension permet de définir les conditions d'apparition d'un événement au moyen d'un prédicat. Des expressions prédicat complexes référençant plusieurs variables MMS peuvent être employées. Cette extension est particulièrement souple, s'adapte mieux aux besoins des utilisateurs et s'intègre tout à fait à MMS sans modifier en quoi que ce soit la norme existante. Nous avons en particulier effectué une implantation de cette extension que nous avons intégrée à notre serveur MMS. Quelques mesures de performance ont été proposées.

5. **Proposition d'une extension permettant l'association de priorités** et d'échéances aux requêtes de service MMS. Cette extension est un exemple typique montrant comment exploiter au mieux les possibilités offertes par la norme sans entraver la compatibilité avec les applications existantes.

6. **Analyse de MMS dans le contexte de systèmes temps-réel.** Nous avons notamment montré comment utiliser des algorithmes classiques et connus dans le contexte d'un serveur MMS. Ces algorithmes ont été utilisés pour l'établissement d'associations temps-réel, l'acceptation des requêtes de service et la détection des événements. Nous avons aussi montré que certaines des fonctionnalités de MMS telles que les modificateurs pouvaient être avantageusement exploitées pour ordonnancer l'exécution des requêtes MMS ainsi que pour contrôler le respect des contraintes de temps.

7. Proposition d'une méthode simple pour permettre l'**exécution sur un serveur secondaire de toute requête de service** ou d'action événementielle. Cette méthode nécessite d'étendre le protocole MMS mais montre comment effectuer simplement et à peu de frais avec MMS une délocalisation de l'exécution des requêtes de service.

8. **Résolution de problèmes classiques avec MMS.** Le but de cette étude était double. Tout d'abord il s'agissait d'illustrer l'utilisation de MMS avec des problèmes bien connus. Dans le même temps, ceci nous a permis de fournir quelques outils entièrement basés sur MMS pour résoudre des problèmes représentatifs de nombreux paradigmes entre applications concurrentes tels que les rendez-vous, les lecteurs/rédacteurs et le problème des philosophes.

9. **Algorithmes de contrôle de la concurrence** avec MMS. Avec ces algorithmes, nous avons montré qu'il est possible de construire sur MMS des applications assurant un contrôle de l'allocation répartie des ressources à la manière des transactions dans les bases de données. Nous nous sommes toutefois occupés uniquement de l'aspect sérialisation de l'accès au ressource. Ceci est particulièrement intéressant dans un environnement industriel où l'information peut être dispersée sur un grand nombres de serveurs différents. Ces algorithmes restent toutefois limités à des situations où la compétition pour l'accès aux ressources est faible si l'on souhaite obtenir des performances satisfaisantes.

   Ces algorithmes nous ont là encore permis d'illustrer en détail comment exploiter les sémaphores MMS. Nous avons également abondamment utilisé les événements MMS pour implanter le concept de sonde dans la détection des interblocages distribués.

8.2 Perspectives futures

L'étude de la norme MMS que nous venons de faire laisse encore la porte ouverte à de nombreux travaux.

Tout d'abord, il est apparu que certaines fonctionnalités de MMS profiteraient grandement de modifications ou d'extensions. Nous avons souvent pu proposer de telles extensions. Celles-ci sont simples et s'intègrent tout à fait à l'esprit de MMS. L'extension à la détection d'événements proposée à la section 5.2 ou la différenciation entre les modes d'acquisition d'un sémaphore étudiée à la section 7.2.2.5 sont quelques exemples de telles extensions s'intégrant harmonieusement à la norme. Une modification de la norme en ce sens serait d'autant plus souhaitable que ces changements peuvent se faire à peu de frais dans la mesure où l'esprit de la norme MMS est respecté et où la compatibilité avec l'existant normatif reste bien souvent assurée.

Nous avons donc proposé quelques solutions aux limitations rencontrées mais n'avons pu résoudre tous les problèmes identifiés. Il reste ainsi à effectuer un effort d'analyse et de conception de solutions adéquates et naturellement adaptées à l'environnement MMS qui pourrait par exemple intervenir dans une optique de révision de la norme. Cette analyse pourrait également s'inscrire dans le cadre plus général d'une réflexion sur ce qui caractérise la capacité d'une norme de communication à évoluer et à accueillir des extensions [Che96].

MMS et le temps-réel est un autre domaine important que nous n'avons pu que traiter partiellement. Nous avons montré comment utiliser et exploiter les capacités de la norme MMS actuelle pour résoudre certains problèmes. Mais il est probable qu'une révision de la norme en vue de doter les systèmes MMS de capacités temps-réel constituerait une solution plus satisfaisante. Dans ce contexte, une norme d'accompagnement de MMS pour les systèmes temps-réel par exemple pourrait se révéler intéressante.

Par ailleurs, il reste à définir plus précisément l'architecture d'un serveur MMS qui intègre les algorithmes et solutions que nous avons proposés pour le temps-réel; puis à implanter un tel serveur. L'architecture des serveurs étudiée dans cette thèse est suffisamment générale pour constituer un point de départ. Mais le serveur implanté pour les besoins du chapitre 4 n'est pas un serveur doté de fonctionnalités temps-réel.

Annexe A

Conventions

On note en *italique* les mots et expressions qui apparaissent pour la première fois dans le texte et/ou sont généralement définis à cet endroit.

On note en **gras** les mots, expressions et phrases dont on veut souligner l'importance.

On note en `courier` les mots et expressions en rapport avec la programmation (les noms des services MMS, leurs paramètres, etc.)

On note entre crochets [xx] les références à des documents externes listés dans la bibliographie.

Annexe B

Abbréviations utilisées

AAId Application Association Identifier *(Identificateur d'association entre applications)*
ASE Application Service Element (*Elément de service de la couche application*)
ASN.1 Abstract Syntax Notation 1 (*Notation de syntaxe abstraite 1*)
BER Basic Encoding Rules (*Règles de codage de base*)
BNF Backus-Naur Form (*Forme de Backus-Naur*)
CCE CIM Computing Environment
CMISE Common Management Information Service Element
CMIP Common Management Information Protocol
CNMA Communications Network for Manufacturing Applications
CORBA Common Object Request Broker Architecture
COSS Common Object Services Specification
DCE Distributed Computing Environment
DM Deadline Monotonic algorithm
EA Event Action (*Action Evénementielle*)
EC Event Condition (*Condition Evénementielle*)
EDF Earliest Deadline First algorithm (*Algorithme à échéance la plus proche en premier*)
EE Event Enrollment (*Enregistrement Evénementiel*)
EM Event Monitor (*Moniteur d'Evénements*)
ETP Event Transition Processor (*Processeur de Transitions d'Evénements*)
HOOD Hierarchical Object Oriented Design (*Conception orientée-objet hiérarchique*)
IDL Interface Definition Language (*Langage de définition d'interfaces*)
LLF Least Laxity First algorithm (*Algorithme à laxité la plus proche en premier*)
MAP Manufacturing Automation Protocol
MMPM MMS Protocol Machine (*Machine de protocole MMS*)
MMS Manufacturing Message Specification
MMSI MMS Interface
NIST National Institute of Standards and Technology
OMG Object Management Group
OS Operating System (*Système d'exploitation*)
PCP Priority Ceiling Protocol
PDU Protocol Data Unit (*Unité de donnée protocolaire*)
PI Program Invocation (*Invocation de programme*)
POO Programmation Orientée-Objet
PPCM Plus Petit Commun Multiple
RM Rate Monotonic algorithm
SE Semaphore Entry (*Rubrique de sémaphore*)

SQL Structured Query Language
TO Transaction Object (*Objet transaction*)
TP Transaction Processor (*Processeur de transactions*)
VMD Virtual Manufacturing Device (*Machine virtuelle de fabrication*)
XED eXtended Event Detection (*Détection étendue des événements*)
XEDI XED Interpreter (*Interpréteur XED*)
XES eXtended Execution of Services (*Exécution étendue des services*)

Annexe C

Lexique des termes utilisés

association : connexion logique entre un client et un serveur MMS sur laquelle les requêtes de service sont effectuées.

association temps-réel : association sur laquelle toute requête MMS effectuée est exécutée par le serveur dans les temps impartis.

MMS classique : dénote un comportement des serveurs MMS qui est défini dans la norme [ISO90a]. Généralement employé pour comparer ce comportement avec celui d'une extension proposée.

mode synchrone : mode de fonctionnement d'un client MMS défini dans MMSI [Gen88] qui implique que le client ayant envoyé une requête de service reste bloqué jusqu'à réception de la réponse correspondante.

mode asynchrone : mode de fonctionnement d'un client MMS défini dans MMSI [Gen88] qui autorise un client ayant envoyé une requête de service à ne pas rester bloqué en attente de la réponse correspondante. Le client peut alors effectuer d'autres tâches utiles et envoyer d'autres requêtes de service.

protocole :

1. désigne le format des données échangées par les applications MMS décrit dans le document [ISO90b] en ASN.1 [ISO87a].
2. au sens large du terme désigne l'ensemble des règles qui mènent à l'entente mutuelle entre deux entités communicantes.

rubrique de sémaphore : objet MMS Semaphore Entry.

service modifié : service dont l'exécution est conditionnée sur l'apparition d'un ou plusieurs événements et/ou la libération d'un ou plusieurs sémaphores. On dit que la requête contient un ou des modificateurs.

session MMS : ouverture d'une association, envoi et réception de services MMS, fermeture de l'association.

variable anonyme : objet MMS UnnamedVariable.

variable nommée : objet MMS NamedVariable.

Annexe D

Tableau récapitulatif des services MMS

n°	Service MMS	Bloquant	Suspensible	Gestionnaires accédés	Gestionnaire responsable
0	Status	non	non	-	VMD
1	GetNameList	non	non	Tous	VMD
2	Identify	non	non	-	VMD
3	Rename	non	non	Tous	VMD
4	Read	non	oui/non	Domaines	Variables
5	Write	non	oui/non	Domaines	Variables
6	GetVariableAccessAttributes	non	non	Domaines	Variables
7	DefineNamedVariable	non	non	Domaines	Variables
8	DefineScatteredAccess	non	non	Domaines	Variables
9	GetScatteredAccessAttributes	non	non	Domaines	Variables
10	DeleteVariableAccess	non	non	Domaines	Variables
11	DefineNamedVariableList	non	non	Domaines	Variables
12	GetNamedVariableListAttributes	non	non	Domaines	Variables
13	DeleteNamedVariableList	non	non	Domaines	Variables
14	DefineNamedType	non	non	Domaines	Variables
15	GetNamedTypeAttributes	non	non	Domaines	Variables
16	DeleteNamedType	non	non	Domaines	Variables
17	Input	oui	oui	-	Station Op.
18	Output	oui	oui	-	Station Op.
19	TakeControl	oui	oui/non	Domaines	Sémaphores
20	RelinquishControl	non	oui/non	Domaines	Sémaphores
21	DefineSemaphore	non	oui/non	Domaines, Evénements	Sémaphores

Tab. D.1 - Tableau récapitulatif

n°	Service MMS	Bloquant	Suspensible	Gestionnaires accédés	Gestionnaire responsable
22	DeleteSemaphore	non	oui/non	Domaines, Evénements	Sémaphores
23	ReportSemaphoreStatus	non	non	Domaines	Sémaphores
24	ReportPoolSemaphoreStatus	non	non	Domaines	Sémaphores
25	ReportSemaphoreEntryStatus	non	non	Domaines	Sémaphores
26	InitiateDownloadSequence	non	oui	-	Domaines
27	DownloadSegment	non	oui	Variables Sémaphores Evénements	Domaines
28	TerminateDownloadSequence	non	oui	-	Domaines
29	InitiateUploadSequence	non	oui	-	Domaines
30	UploadSegment	non	oui	-	Domaines
31	TerminateUploadSequence	non	oui	-	Domaines
32	RequestDomainDownload	oui	oui	-	Domaines
33	RequestDomainUpload	oui	oui	-	Domaines
34	LoadDomainContent	oui	oui	-	Domaines
35	StoreDomainContent	oui	oui	-	Domaines
36	DeleteDomain	oui	oui	Variables Sémaphores Evénements	Domaines
37	GetDomainAttributes	non		-	Domaines
38	CreateProgramInvocation	non	non	Domaines, Evénements	Programmes
39	DeleteProgramInvocation	non	non	Domaines, Evénements	Programmes
40	Start	non	oui	-	Programmes
41	Stop	non	oui	-	Programmes
42	Resume	non	oui	-	Programmes
43	Reset	non	oui	-	Programmes
44	Kill	non	oui	-	Programmes
45	GetProgramInvocationAttributes	non	non	-	Programmes
46	ObtainFile	oui	oui	-	Fichiers
47	DefineEventCondition	non	non	Domaines Variables	Evénements
48	DeleteEventCondition	non	non	Domaines	Evénements
49	GetEventConditionAttributes	non	non	Domaines	Evénements
50	ReportEventConditionStatus	non	non	Domaines	Evénements
51	AlterEventConditionMonitoring	non	non	Domaines	Evénements
52	TriggerEvent	non	oui/non	Domaines	Evénements
53	DefineEventAction	non	non	Domaines	Evénements
54	DeleteEventAction	non	non	Domaines	Evénements
55	GetEventActionAttributes	non	non	Domaines	Evénements

TAB. D.2 - Tableau récapitulatif (suite)

n°	Service MMS	Bloquant	Suspensible	Gestionnaires accédés	Gestionnaire responsable
56	ReportEventActionStatus	non	non	Domaines	Evénements
57	DefineEventEnrollment	non	non	Domaines	Evénements
58	DeleteEventEnrollment	non	non	Domaines	Evénements
59	AlterEventEnrollment	non	non	Domaines	Evénements
60	ReportEventEnrollmentStatus	non	non	Domaines	Evénements
61	GetEventEnrollmentAttributes	non	non	Domaines	Evénements
62	AcknowledgeEventNotification	non	oui/non	Domaines	Evénements
63	GetAlarmSummary	non	oui	Domaines	Evénements
64	GetAlarmEnrollmentSummary	non	oui	Domaines	Evénements
65	ReadJournal	non	oui	-	Journaux
66	WriteJournal	non	oui	-	Journaux
67	InitializeJournal	non	oui	-	Journaux
68	ReportJournalStatus	non	oui	-	Journaux
69	CreateJournal	non	oui	-	Journaux
70	DeleteJournal	non	oui	-	Journaux
71	GetCapabilityList	non	non	-	VMD
72	FileOpen	non	oui	-	Fichiers
73	FileRead	non	oui	-	Fichiers
74	FileClose	non	oui	-	Fichiers
75	FileRename	non	oui	-	Fichiers
76	FileDelete	non	oui	-	Fichiers
77	FileDirectory	non	oui	-	Fichiers
78	UnsolicitedStatus	non	non	-	VMD
79	InformationReport	non	non	-	Variables
80	EventNotification	non	non	-	Evénements
81	AttachToEventCondition	oui	non	Domaines	Evénements
82	AttachToSemaphore	oui	non	Domaines	Sémaphores
83	Conclude	non	non	Sémaphores	Associations
84	Cancel	non	non	Sémaphores Evénements	Associations
-	Initiate	non	non	-	Associations
-	Abort	non	non	Tous	Associations

TAB. D.3 - Tableau récapitulatif (suite)

Annexe E

Code PROMELA pour la vérification des algorithmes proposés

```
/*******************************************************************************************
 *                                                                                         *
 *      CODE PROMELA POUR LA VÉRIFICATION DU RENDEZ-VOUS MMS PAR LA METHODE DE AKAZAN      *
 *                                    SIMPLIFIEE                                           *
 *                                                                                         *
 *******************************************************************************************/

# define false                 0
# define true                  1

/*******************************************
 *             Services MMS               *
 *******************************************/

# define takeControl           1
# define relinquishControl     2


/*******************************************
 *             Associations               *
 *******************************************/

chan ass_A1_Serveur = [0] of { bit };
chan ass_A2_Serveur = [0] of { bit };
chan ass_Serveur_A1 = [0] of { bit };
chan ass_Serveur_A2 = [0] of { bit };
chan ass_S1 = [0] of { bit };
chan ass_S2 = [0] of { bit };


/*******************************************
 *           Variables globales           *
 *******************************************/

byte test_variable = 0;      /* Teste que les applications A1 et A2 effectuent bien un rendez-vous */
byte waiting_1     = false;  /* Permet d'effectuer le service ReportSemaphoreEntryStatus sur S1    */
byte waiting_2     = false;  /* Permet d'effectuer le service ReportSemaphoreEntryStatus sur S2    */


/*******************************************
 * Processus APPLICATION 1                *
 *******************************************/

proctype A1()
{
  assert(test_variable == 0);           /* Assure que A2 attend A1 */

  /* --- Simulation du modificateur --- */
```

```
  atomic{
        waiting_1 = true;
        ass_S1!takeControl;
        }
  ass_S1!relinquishControl;            /* Rendez-vous */


  test_variable = 1
}


/*****************************************
 * Processus APPLICATION 2              *
 *****************************************/

proctype A2()
{

  assert(test_variable == 0);          /* Assure que A1 attend A2 */

  /* --- Simulation du modificateur --- */

  atomic{
        waiting_2 = true;
        ass_S2!takeControl;
        }
  ass_S2!relinquishControl;            /* Rendez-vous */


  test_variable = 1
}


/*****************************************
 * Processus APPLICATION Ardv           *
 *****************************************/

proctype Ardv()
{

  ass_S1!takeControl;
  ass_S2!takeControl;
  run A1();
  run A2();

  do
  :: (waiting_1 == true) -> break  /* Simule le service ReportSemaphoreEntryStatus */
  od;

  do
  :: (waiting_2 == true) -> break  /* Simule le service ReportSemaphoreEntryStatus */
  od;

  ass_S1!relinquishControl;
  ass_S2!relinquishControl;

}


/*****************************************
 * Processus SEMAPHORE 1                *
 *****************************************/

proctype SemaphoreS1()
{
byte semaphore = 1;

end:    do
:: (semaphore == 1) -> end1: ass_S1?takeControl;
   semaphore = 0
```

```
:: (semaphore == 0) -> end2: ass_S1?relinquishControl;
   semaphore = 1
od
}


/******************************************
 * Processus SEMAPHORE 2                  *
 ******************************************/

proctype SemaphoreS2()
{
byte semaphore = 1;

end: do
:: (semaphore == 1) -> end1: ass_S2?takeControl;
   semaphore = 0
:: (semaphore == 0) -> end2: ass_S2?relinquishControl;
   semaphore = 1
od
}


/******************************************
 * Processus d'initialisation             *
 ******************************************/

init
{
   atomic{
run SemaphoreS1();
run SemaphoreS2();
        run Ardv()
   }
}
```

```
/*********************************************************************************************
 *                                                                                           *
 *      CODE PROMELA POUR LA VÉRIFICATION DU RENDEZ-VOUS BASE SUR LES EVENEMENTS MMS         *
 *                                                                                           *
 *********************************************************************************************/


/********************************************
 *            Services MMS                  *
 ********************************************/

# define triggerEvent      1
# define eventNotification 2


/********************************************
 *            Associations                  *
 ********************************************/

chan ass_A1_Serveur = [0] of { bit };
chan ass_A2_Serveur = [0] of { bit };
chan ass_Serveur_A1 = [0] of { bit };
chan ass_Serveur_A2 = [0] of { bit };


/********************************************
 *          Variables globales              *
 ********************************************/

byte test_variable = 0;


/********************************************
 * Processus SERVEUR MMS                    *
 ********************************************/

proctype Serveur()
{
end:  do
      :: ass_A1_Serveur?triggerEvent -> ass_Serveur_A2!eventNotification
      :: ass_A2_Serveur?triggerEvent -> ass_Serveur_A1!eventNotification
      od
}


/********************************************
 * Processus APPLICATION 1                  *
 ********************************************/

proctype A1()
{
  assert(test_variable == 0);           /* Assure que A2 attend A1 */

  ass_A1_Serveur!triggerEvent;          /* Rendez-vous */
  ass_Serveur_A1?eventNotification;

  test_variable = 1
}


/********************************************
 * Processus APPLICATION 2                  *
 ********************************************/

proctype A2()
{

  assert(test_variable == 0);           /* Assure que A1 attend A2 */

  ass_Serveur_A2?eventNotification;     /* Rendez-vous */
  ass_A2_Serveur!triggerEvent;
```

```
  test_variable = 1
}


/******************************************
 * Processus d'initialisation             *
 ******************************************/

init
{
   atomic{
        run Serveur();
        run A1();
        run A2()
   }
}
```

```
/*******************************************************************************************
 *                                                                                         *
 *       CODE PROMELA POUR LA VÉRIFICATION DES RENDEZ-VOUS MULTIPLES BASE SUR MMS          *
 *                      (ici entre 3 applications A1, A2 et A3)                            *
 *                                                                                         *
 *******************************************************************************************/


/******************************************
 *             Services MMS               *
 ******************************************/

# define triggerEvent      1
# define eventNotification 2


/******************************************
 *             Associations               *
 ******************************************/

chan ass_A0_Serveur = [0] of { bit };
chan ass_Serveur_A0 = [0] of { bit };
chan ass_A1_Serveur = [0] of { bit };
chan ass_Serveur_A1 = [0] of { bit };
chan ass_A2_Serveur = [0] of { bit };
chan ass_Serveur_A2 = [0] of { bit };
chan ass_A3_Serveur = [0] of { bit };
chan ass_Serveur_A3 = [0] of { bit };


/******************************************
 *           Variables globales           *
 ******************************************/

byte test_variable = 0;


/******************************************
 * Processus SERVEUR MMS                  *
 ******************************************/

proctype Serveur()
{
end:  do
      :: ass_A1_Serveur?triggerEvent -> ass_Serveur_A0!eventNotification
      :: ass_A2_Serveur?triggerEvent -> ass_Serveur_A0!eventNotification
      :: ass_A3_Serveur?triggerEvent -> ass_Serveur_A0!eventNotification
      :: ass_A0_Serveur?triggerEvent ->
           ass_Serveur_A1!eventNotification;
           ass_Serveur_A2!eventNotification;
           ass_Serveur_A3!eventNotification
      od
}


/******************************************
 * Processus APPLICATION 0                *
 ******************************************/

proctype A0()
{
  assert(test_variable == 0);          /* Assure que les applications s'attendent */

  ass_Serveur_A0?eventNotification;
  ass_Serveur_A0?eventNotification;
  ass_Serveur_A0?eventNotification;

  ass_A0_Serveur!triggerEvent;

  test_variable = 1
}
```

```
/*****************************************
 * Processus APPLICATION 1               *
 *****************************************/

proctype A1()
{
  assert(test_variable == 0);          /* Assure que les applications s'attendent */

  ass_A1_Serveur!triggerEvent;
  ass_Serveur_A1?eventNotification;    /* Rendez-vous */

  test_variable = 1
}


/*****************************************
 * Processus APPLICATION 2               *
 *****************************************/

proctype A2()
{

  assert(test_variable == 0);          /* Assure que les applications s'attendent */

  ass_A2_Serveur!triggerEvent;
  ass_Serveur_A2?eventNotification;    /* Rendez-vous */

  test_variable = 1
}


/*****************************************
 * Processus APPLICATION 3               *
 *****************************************/

proctype A3()
{

  assert(test_variable == 0);          /* Assure que les applications s'attendent */

  ass_A3_Serveur!triggerEvent;
  ass_Serveur_A3?eventNotification;    /* Rendez-vous */

  test_variable = 1
}

/*****************************************
 * Processus d'initialisation            *
 *****************************************/

init
{
   atomic{
        run Serveur();
        run A0();
        run A1();
        run A2();
        run A3()
   }
}
```

```
/*******************************************************************************************
 *                                                                                         *
 *      CODE PROMELA POUR LA VÉRIFICATION DE LA SOLUTION BASÉE SUR LES ÉVÉNEMENTS MMS      *
 *                          DU PROBLÈME DES LECTEURS/RÉDACTEURS                            *
 *                                                                                         *
 *******************************************************************************************/


# define false                  0
# define true                   1


/*******************************************
 *            Services MMS               *
 *******************************************/

# define takeControl            0
# define relinquishControl      1
# define defineEE               2
# define deleteEE               3
# define triggerEvent           4
# define eventNotification      5


/*******************************************
 *            Associations               *
 *******************************************/

chan associationW  = [0] of { bit };            /* vers sémaphore W       */
chan associationM  = [0] of { bit };            /* vers sémaphore MUTEX   */
chan associationS  = [0] of { byte,byte };      /* lecteurs vers serveur  */
chan associationS1 = [0] of { byte };           /* serveur vers lecteur 1 */
chan associationS2 = [0] of { byte };           /* serveur vers lecteur 2 */
chan associationS3 = [0] of { byte };           /* serveur vers lecteur 3 */


/*******************************************
 *          Variables globales           *
 *******************************************/

byte critical_section = 0;
byte readCount        = 0;


/*******************************************
 * Processus LECTEUR                     *
 *******************************************/

proctype Reader(byte number)
{
  byte was_first       = false;
  byte local_readCount = 0;

  associationM!takeControl;
  readCount = readCount + 1;
  if
  :: (readCount == 1) -> was_first = true;
     associationW!takeControl;
     associationS!defineEE(number)
  :: (readCount != 1)
  fi;
  associationM!relinquishControl;

  /* Entrée en section critique */

  assert(critical_section == 0);
  printf("Reader %d in critical section\n", number);

  /* Sortie de la section critique */

  associationM!takeControl;
```

```
  readCount = readCount - 1;
  local_readCount = readCount;
  if
  :: (readCount == 0) ->

        if
        :: (was_first == true) -> associationS!deleteEE(number);
           associationW!relinquishControl
        :: (was_first == false) ->
           associationS!triggerEvent(number)
        fi

  :: (readCount != 0)
  fi;

  associationM!relinquishControl;

  if
  :: (was_first == true) && (local_readCount != 0) ->
     if
     :: (number == 1) -> associationS1?eventNotification
     :: (number == 2) -> associationS2?eventNotification
     :: (number == 3) -> associationS3?eventNotification
     fi;
     associationS!deleteEE(number);
     associationW!relinquishControl

  :: (was_first == false) || (local_readCount == 0)
  fi

}


/*******************************************
 * Processus REDACTEUR                     *
 *******************************************/

proctype Writer(byte number)
{
  associationW!takeControl;

  /* Entrée en section critique */

  critical_section = critical_section + 1;
  assert(critical_section == 1);
  critical_section = critical_section - 1;

  /* Sortie de la section critique */

  associationW!relinquishControl;

}


/*******************************************
 * Processus SERVEUR MMS                   *
 *******************************************/

proctype Server()
{
  byte number  = 0;
  byte whos_ee = 0;

  end: do
  :: associationS?triggerEvent(number) ->
        if
        :: (whos_ee == 1) -> associationS1!eventNotification
        :: (whos_ee == 2) -> associationS2!eventNotification
        :: (whos_ee == 3) -> associationS3!eventNotification
        fi
  :: associationS?defineEE(number) -> assert(whos_ee == 0); whos_ee = number
```

```
  :: associationS?deleteEE(number) -> whos_ee = 0
  od
}


/*******************************************
 * Processus SEMAPHORE W                   *
 *******************************************/

proctype SemaphoreW()
{
        byte semaphore = 1;

        do
        :: (semaphore == 1) -> end: associationW?takeControl;
           semaphore = 0
        :: (semaphore == 0) -> associationW?relinquishControl;
           semaphore = 1
        od
}


/*******************************************
 * Processus SEMAPHORE MUTEX               *
 *******************************************/

proctype SemaphoreM()
{
        byte semaphore = 1;

        do
        :: (semaphore == 1) -> end: associationM?takeControl;
           semaphore = 0
        :: (semaphore == 0) -> associationM?relinquishControl;
           semaphore = 1
        od
}


/*******************************************
 * Processus d'initialisation              *
 *******************************************/

init
{
   atomic {
        run SemaphoreW();
        run SemaphoreM();
        run Server()
   };
   atomic{
        run Reader(1);
        run Reader(2);
        run Reader(3);
        run Writer(1);
        run Writer(2)
    }
}
```

```
/*******************************************************************************************
 *                                                                                         *
 *    CODE PROMELA POUR LA VÉRIFICATION DE LA SOLUTION UTILISANT LES EVENEMENTS MMS AU      *
 *                             PROBLÈME DES PHILOSOPHES                                    *
 *                                                                                         *
 *******************************************************************************************/

# define false                  0
# define true                   1

# define nb_philosophes         5               /* Nombre de philosophes autour de la table */

/******************************************
 *        Etats d'un philosophe           *
 ******************************************/

# define PENSER                 0
# define ATTENDRE               1
# define MANGER                 2

/******************************************
 *            Services MMS                *
 ******************************************/

# define takeControl            0
# define relinquishControl      1
# define triggerEvent           4
# define eventNotification      5


chan to_philo_0 = [3] of { byte }     /* Connexion serveur vers philosophe 0 (notification) */
chan to_philo_1 = [3] of { byte }     /* Connexion serveur vers philosophe 1 (notification) */
chan to_philo_2 = [3] of { byte }     /* Connexion serveur vers philosophe 2 (notification) */
chan to_philo_3 = [3] of { byte }     /* Connexion serveur vers philosophe 3 (notification) */
chan to_philo_4 = [3] of { byte }     /* Connexion serveur vers philosophe 4 (notification) */

chan to_server = [0] of { byte,byte } /* Connexion philosophes vers serveur (triggerEvent)  */

chan to_mutex = [0] of { bit }        /* Canal de communication avec le mutex              */

chan connections[nb_philosophes]      /* Tableau de connexions vers les philosophes        */

byte etat[nb_philosophes];            /* Tableau etat courant des philosophes (MANGER,     */
                                      /* PENSER, ATTENDRE).                                */



/******************************************
 * Processus PHILOSOPHE                   *
 ******************************************/

proctype Philosophe(byte number; chan from_server)
{
  byte wait;

START:

  /* >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PENSER <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< */

  /* ---------------------------------------------------------------------- */

atomic{

  to_mutex!takeControl;

  etat[number] = ATTENDRE;
  wait = false;

}

  /* Les voisins sont-ils en train de manger ? */
```

```
  if
  :: (etat[(number+nb_philosophes-1) % nb_philosophes] != MANGER)
       && (etat[(number+1) % nb_philosophes] != MANGER)
       -> etat[number] = MANGER;
  :: else -> wait = true;
  fi;

  to_mutex!relinquishControl;

  /* ----------------------------------------------------------------------- */

  if
  :: (wait == true) -> from_server?eventNotification
  :: else
  fi;

  /* >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> MANGER <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< */

  /* S'assurer que les voisins du philosophe qui mange ne sont pas aussi en  */
  /* train de manger.                                                        */

  assert((etat[number] == MANGER) &&
         (etat[(number+nb_philosophes-1) % nb_philosophes] != MANGER) &&
         (etat[(number+1) % nb_philosophes] != MANGER));

  /* ----------------------------------------------------------------------- */

  to_mutex!takeControl;

  etat[number] = PENSER;

  /* Mon voisin de gauche attend il pour manger et le voisin de gauche de    */
  /* mon voisin de gauche est-il en train de manger ?                        */

  if
  :: (etat[(number+nb_philosophes-1) % nb_philosophes] == ATTENDRE)
       && (etat[(number+nb_philosophes-2) % nb_philosophes] != MANGER)
       -> etat[(number+nb_philosophes-1) % nb_philosophes] = MANGER;

          /* Si non, signaler a mon voisin de gauche qu'il peut manger      */

          to_server!triggerEvent(connections[(number+nb_philosophes-1) % nb_philosophes])
  :: else
  fi;

  /* Mon voisin de droite attend il pour manger et le voisin de droite de    */
  /* mon voisin de droite est-il en train de manger ?                        */

  if
  :: (etat[(number+1) % nb_philosophes] == ATTENDRE)
       && (etat[(number+2) % nb_philosophes] != MANGER)
       -> etat[(number+1) % nb_philosophes] = MANGER;

          /* Si non, signaler a mon voisin de droite qu'il peut manger      */

          to_server!triggerEvent(connections[(number+1) % nb_philosophes])
  :: else
  fi;

  to_mutex!relinquishControl;

  /* ----------------------------------------------------------------------- */

  goto START

}


/*******************************************
 * Processus SERVEUR MMS                   *
```

```
 ******************************************/

/* Le serveur n'est utilise que pour envoyer la notification d'evenement. On */
/* suppose que tous les ECs et EEs sont deja definis.                        */

proctype Server()
{
  chan assoID;

  end: do
  :: to_server?triggerEvent(assoID) -> assoID!eventNotification
  od
}


/******************************************
 * Processus MUTEX                        *
 ******************************************/

proctype Mutex()
{

end:    do
        :: to_mutex?takeControl -> to_mutex?relinquishControl;
        od
}


/******************************************
 * Processus d'initialisation             *
 ******************************************/

init
{
  byte counter = 0;

atomic{

  connections[0] = to_philo_0;
  connections[1] = to_philo_1;
  connections[2] = to_philo_2;
  connections[3] = to_philo_3;
  connections[4] = to_philo_4;

  run Mutex();
  run Server();

  do
  :: counter == nb_philosophes -> break
  :: else ->
        etat[counter] = 0;
        run Philosophe(counter, connections[counter]);
        counter = counter + 1
  od;

}

}
```

```
/*******************************************************************************************
 *                                                                                         *
 *   CODE PROMELA POUR LA VÉRIFICATION DE LA SOLUTION MMS XED AU PROBLÈME DES PHILOSOPHES   *
 *                                                                                         *
 *******************************************************************************************/

# define false                      0
# define true                       1

# define nb_philosophes             5        /* Nombre de philosophes autour de la table */

/*******************************************
 *        Etats d'un philosophe          *
 *******************************************/

# define PENSER                     0
# define ATTENDRE                   1
# define MANGER                     2

/*******************************************
 *            Services MMS               *
 *******************************************/

# define eventNotification          1


chan to_philo_0 = [3] of { byte }     /* Connexion serveur vers philosophe 0 (notification) */
chan to_philo_1 = [3] of { byte }     /* Connexion serveur vers philosophe 1 (notification) */
chan to_philo_2 = [3] of { byte }     /* Connexion serveur vers philosophe 2 (notification) */
chan to_philo_3 = [3] of { byte }     /* Connexion serveur vers philosophe 3 (notification) */
chan to_philo_4 = [3] of { byte }     /* Connexion serveur vers philosophe 4 (notification) */

chan connections[nb_philosophes]      /* Tableau de connexions vers les philosophes         */

byte etat[nb_philosophes];            /* Tableau etat courant des philosophes (MANGER,      */
                                      /* PENSER, ATTENDRE).                                 */


/*******************************************
 * Processus PHILOSOPHE                  *
 *******************************************/

proctype Philosophe(byte number; chan from_server)
{

START:

  /* >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> PENSER <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< */

  /* ---------------------------------------------------------------------- */

  etat[number] = ATTENDRE;                /* Je signale que je veux manger */

  from_server?eventNotification;       /* J'attends l'autorisation de manger */

  /* ---------------------------------------------------------------------- */

  /* >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> MANGER <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< */

  /* S'assurer que les voisins du philosophe qui mange ne sont pas aussi en  */
  /* train de manger.                                                        */

  assert((etat[number] == MANGER) &&
         (etat[(number+nb_philosophes-1) % nb_philosophes] != MANGER) &&
         (etat[(number+1) % nb_philosophes] != MANGER));

  /* ---------------------------------------------------------------------- */

  etat[number] = PENSER;                /* Je signale que je ne mange plus */
```

```
  /* ---------------------------------------------------------------------- */

  goto START

}


/******************************************
 * Processus SERVEUR MMS                  *
 ******************************************/

/* Le serveur n'est utilise que pour envoyer la notification d'evenement. On */
/* suppose que tous les ECs et EEs sont deja definis.                        */

proctype Server()
{

  end: do
  :: (etat[0] == ATTENDRE) && (etat[4] != MANGER) && (etat[1] != MANGER) ->
        etat[0] = MANGER; connections[0]!eventNotification
  :: (etat[1] == ATTENDRE) && (etat[0] != MANGER) && (etat[2] != MANGER) ->
        etat[1] = MANGER; connections[1]!eventNotification
  :: (etat[2] == ATTENDRE) && (etat[1] != MANGER) && (etat[3] != MANGER) ->
        etat[2] = MANGER; connections[2]!eventNotification
  :: (etat[3] == ATTENDRE) && (etat[2] != MANGER) && (etat[4] != MANGER) ->
        etat[3] = MANGER; connections[3]!eventNotification
  :: (etat[4] == ATTENDRE) && (etat[3] != MANGER) && (etat[0] != MANGER) ->
        etat[4] = MANGER; connections[4]!eventNotification
  od;

}



/******************************************
 * Processus d'initialisation             *
 ******************************************/

init
{
  byte counter = 0;

atomic{

  connections[0] = to_philo_0;
  connections[1] = to_philo_1;
  connections[2] = to_philo_2;
  connections[3] = to_philo_3;
  connections[4] = to_philo_4;

  run Server();

  do
  :: counter == nb_philosophes -> break
  :: else ->
        etat[counter] = 0;
        run Philosophe(counter, connections[counter]);
        counter = counter + 1
  od;

}

}
```

```
/*********************************************************************************************
 *                                                                                           *
 *       CODE PROMELA POUR LA VÉRIFICATION DE L'ALGORITHME DE CONTRÔLE DE LA CONCURRENCE      *
 *                               UTILISANT LE PRINCIPE DES SONDES                            *
 *                                                                                           *
 *********************************************************************************************/


# define false                  0
# define true                   1


/*******************************************
 *              Services MMS               *
 *******************************************/

# define takeControl            0
# define relinquishControl      1
# define defineEE               2
# define deleteEE               3
# define triggerEvent           4
# define eventNotification      5


/*******************************************
 *              Associations               *
 *******************************************/

chan ToSem1 = [0] of { bit }
chan ToSem2 = [0] of { bit }
chan ToSem3 = [0] of { bit }

chan ToServer1 = [0] of { byte,byte };
chan ToServer2 = [0] of { byte,byte };
chan ToServer3 = [0] of { byte,byte };

chan ToClient1 = [5] of { byte,byte,byte };
chan ToClient2 = [5] of { byte,byte,byte };
chan ToClient3 = [5] of { byte,byte,byte };

chan semaphores[9];
chan servers[9];
chan clients[3];

byte passed[9];
byte inverse[9];

/*******************************************
 *            Variables globales           *
 *******************************************/

byte critical_section = 0;


/*******************************************
 * Processus CLIENT                        *
 *******************************************/

proctype Client(byte clientID; chan association)
{
  byte initiatorID      = 0;
  byte current          = 0;
  byte serverID         = 0;
  byte probe_received   = false;
  byte owned_semaphores = 0;
  byte counter = 0;

  do :: counter < 3 ->

START:
```

```
  do
  :: (owned_semaphores != 3) ->

  {

  atomic{
        probe_received = false;
        if
        :: (passed[clientID*3-3] == false) -> current = clientID*3-3
        :: (passed[clientID*3-2] == false) -> current = clientID*3-2
        :: (passed[clientID*3-1] == false) -> current = clientID*3-1
        fi;
}

  do

  :: semaphores[current]?takeControl ->
        atomic{
              servers[current]!defineEE(clientID);
              owned_semaphores = owned_semaphores + 1;
              passed[current] = true;
              goto START
              }

  :: timeout ->
        if
        :: (owned_semaphores > 0) -> servers[current]!triggerEvent(clientID)
        :: else
        fi;
        goto NEXT

  :: association?eventNotification(initiatorID, serverID) ->

        /* La sonde recue ici ne peut avoir pour initiateur ce client.  */
        /* Ceci est prouve par le l'instruction suivante :              */

        assert(clientID != initiatorID);

        /* On ne fait donc que relayer la sonde vers le client suivant. */

        servers[current]!triggerEvent(initiatorID)

  od;

NEXT:

  do

  :: semaphores[current]?takeControl ->
        atomic{
              servers[current]!defineEE(clientID);
              owned_semaphores = owned_semaphores + 1;
              passed[current] = true;
              goto START
              }

  :: association?eventNotification(initiatorID, serverID) ->

        /* La sonde recue ici peut avoir pour initiateur ce client. */
        /* On va donc tester son origine et relacher un semaphore   */
        /* ou relayer la sonde vers le client suivant.              */

        if
        :: (clientID == initiatorID) || (probe_received == true) ->
            atomic
               {
               probe_received = true;
               passed[inverse[clientID*3-4+serverID]] = false;
               owned_semaphores = owned_semaphores - 1;
               }
```

```
            servers[serverID-1]!deleteEE(clientID);
            semaphores[serverID-1]!relinquishControl

        :: else ->
            servers[current]!triggerEvent(initiatorID)
        fi

  od;

  }

  :: (owned_semaphores == 3) -> break          /* Tous les semaphores sont acquis */

  od;

  /* ------------------------- ENTREE SECTION CRITIQUE ------------------------- */

  /* Les instructions suivantes garantissent la protection de la section critique. */

  critical_section = critical_section + 1;
  assert(critical_section == 1);
  critical_section = critical_section - 1;

  /* ------------------------- SORTIE SECTION CRITIQUE ------------------------- */


  /* Liberation des semaphores detenus et destructions des EEs crees.             */

  servers[clientID*3-3]!deleteEE(clientID);
  semaphores[clientID*3-3]!relinquishControl;

  servers[clientID*3-2]!deleteEE(clientID);
  semaphores[clientID*3-2]!relinquishControl;

  servers[clientID*3-1]!deleteEE(clientID);
  semaphores[clientID*3-1]!relinquishControl;

  atomic{
        passed[clientID*3-3] = false;
        passed[clientID*3-2] = false;
        passed[clientID*3-1] = false;
        owned_semaphores = 0;
        counter = counter + 1
        }

  :: else -> break

  od
}


/*******************************************
 * Processus SERVEUR MMS                   *
 *******************************************/

proctype Server(byte serverID; chan association)
{
  byte whos_ee     = 0;
  byte clientID    = 0;
  byte initiatorID;

  end: do
  :: association?triggerEvent(initiatorID) -> clients[whos_ee-1]!eventNotification(initiatorID, serverID)
  :: association?defineEE(clientID) -> assert(whos_ee == 0); whos_ee = clientID
  :: association?deleteEE(clientID) -> assert(whos_ee == clientID); whos_ee = 0
  od
}


/*******************************************
 * Processus SEMAPHORE                     *
```

```
 ******************************************/

proctype Semaphore(chan association)
{

end:   do
       :: association!takeControl -> association?relinquishControl;
       od
}


/******************************************
 * Processus d'initialisation             *
 ******************************************/

init
{
   atomic{

   semaphores[0] = ToSem1;
   semaphores[1] = ToSem2;
   semaphores[2] = ToSem3;
   semaphores[3] = ToSem2;
   semaphores[4] = ToSem3;
   semaphores[5] = ToSem1;
   semaphores[6] = ToSem3;
   semaphores[7] = ToSem2;
   semaphores[8] = ToSem1;

   servers[0] = ToServer1;
   servers[1] = ToServer2;
   servers[2] = ToServer3;
   servers[3] = ToServer2;
   servers[4] = ToServer3;
   servers[5] = ToServer1;
   servers[6] = ToServer3;
   servers[7] = ToServer2;
   servers[8] = ToServer1;

   clients[0] = ToClient1;
   clients[1] = ToClient2;
   clients[2] = ToClient3;

   passed[0] = false;
   passed[1] = false;
   passed[2] = false;
   passed[3] = false;
   passed[4] = false;
   passed[5] = false;
   passed[6] = false;
   passed[7] = false;
   passed[8] = false;

   inverse[0] = 0;   /* Indice dans passed du semaphore 1 pour le client 1 */
   inverse[1] = 1;   /* Indice dans passed du semaphore 2 pour le client 1 */
   inverse[2] = 2;   /* Indice dans passed du semaphore 3 pour le client 1 */
   inverse[3] = 5;   /* Indice dans passed du semaphore 1 pour le client 2 */
   inverse[4] = 3;   /* Indice dans passed du semaphore 2 pour le client 2 */
   inverse[5] = 4;   /* Indice dans passed du semaphore 3 pour le client 2 */
   inverse[6] = 8;   /* Indice dans passed du semaphore 1 pour le client 3 */
   inverse[7] = 7;   /* Indice dans passed du semaphore 2 pour le client 3 */
   inverse[8] = 6;   /* Indice dans passed du semaphore 3 pour le client 3 */

   }

   atomic{
       run Semaphore(ToSem1);
       run Semaphore(ToSem2);
       run Semaphore(ToSem3);
       run Server(1, ToServer1);
       run Server(2, ToServer2);
```

```
        run Server(3, ToServer3)
    }

    atomic{
        run Client(1, ToClient1);
        run Client(2, ToClient2);
        run Client(3, ToClient3)
    }
}
```

Annexe F

Scénarios d'application

Nous avons regroupé dans cette annexe trois exemples d'utilisation de MMS (et principalement des événements et sémaphores) dans le cadre de scénarios d'application industrielle. Bien que réels, ces exemples ne sont que des cas d'étude et ne prétendent en aucun cas résoudre tous les problèmes posés par le scénario d'application considéré. Notre but ici n'est que d'illustrer une utilisation possible de MMS.

F.1 Installation d'embouteillage de produits chimiques

F.1.1 Le scénario

Nous allons prendre l'exemple d'une installation d'embouteillage. Cet exemple est tiré de [HS92] pp. 7–10 et adapté à MMS.

L'installation considérée est composée d'un réservoir dans lequel deux liquides chimiques A et B doivent réagir. Le pH du mélange doit être maintenu constant. Une fois cette réaction terminée, le liquide résultant est utilisé pour remplir des bouteilles de différents volumes.

La réaction entre les liquides A et B nécessite de chauffer le mélange qui peut devenir explosif si la température est trop élevée. Il faut donc en permanence contrôler la température du liquide dans le réservoir et régler le chauffage en conséquence.

Les bouteilles à remplir sont amenées une par une depuis un magasin devant une vanne de remplissage provenant du réservoir. Les bouteilles arrivent depuis différentes lignes et sont positionnées sur une plateforme en vue de leur remplissage. Cette plateforme est dotée d'un capteur de poids pour savoir quand une bouteille est remplie. Le volume des bouteilles est constant sur une même ligne (pour simplifier nous n'avons représenté qu'une seule ligne sur la figure F.1). Le traitement que subissent les bouteilles de différentes lignes avant et après le remplissage est différent. Seul le remplissage est commun à toutes les lignes de sorte que chaque ligne de bouteilles est contrôlée par un processus différent (ici une application client MMS).

Les différents capteurs du système sont les suivants :

- niveau du mélange dans le réservoir;
- pH du mélange dans le réservoir;
- température du mélange dans le réservoir;

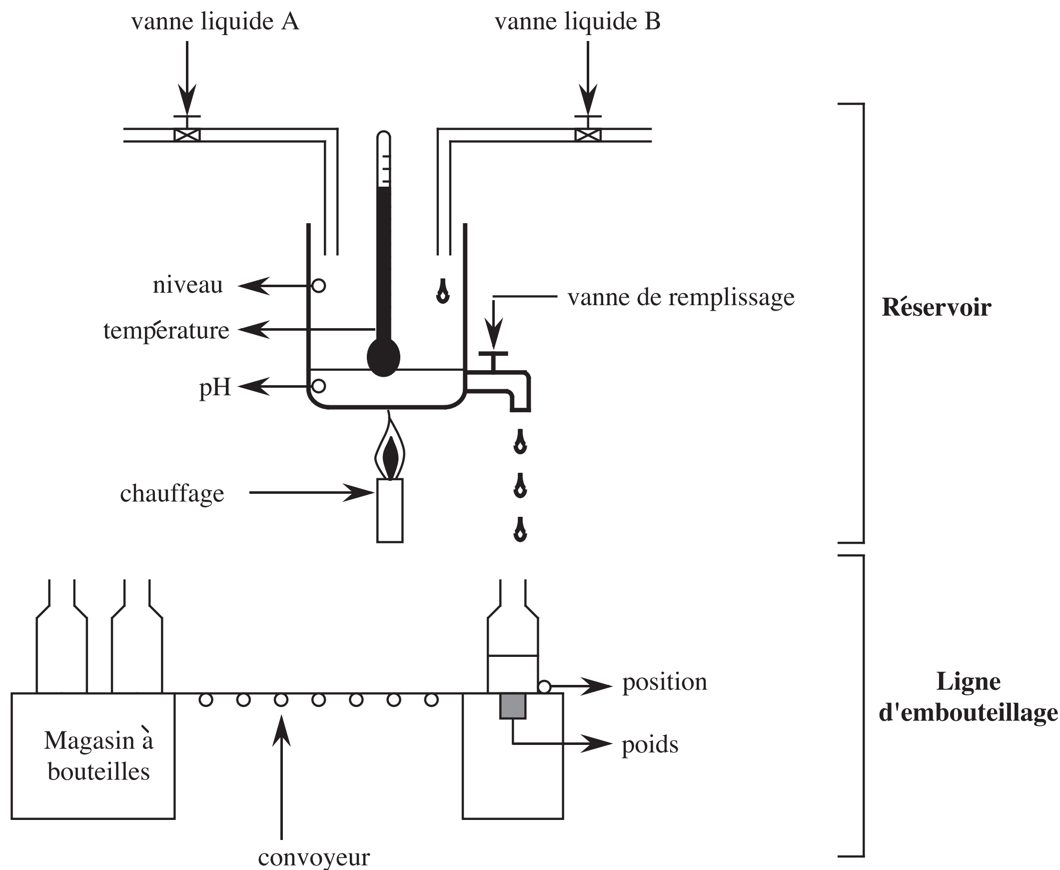


FIG. F.1 - Architecture générale de l'installation

– poids des bouteilles sur la plateforme;

– positionnement des bouteilles sur la plateforme.

Les actionneurs utilisés sont :

– l'ouverture/fermeture de la vanne pour le liquide A;

– l'ouverture/fermeture de la vanne pour le liquide B;

– l'ouverture/fermeture de la vanne pour le remplissage des bouteilles;

– le contrôle du dispositif de chauffage;

– le contrôle du mouvement des convoyeurs des différentes lignes de bouteilles.

Ce dispositif est représenté sur la figure F.1. Le système logiciel contrôlant l'ensemble du dispositif est relativement complexe. Nous allons uniquement faire ressortir une utilisation possible des événements et sémaphores MMS dans le cadre de cette installation sans entrer dans les détails du contrôle de ce système.

F.1.2 Modélisation avec MMS

Nous modélisons le réservoir et l'ensemble du dispositif de chauffage et de contrôle des vannes par une VMD unique notée VMD_{res}. Les convoyeurs permettant le déplacement des bouteilles sont tous

modélisés par une seconde VMD (notée VMD_{lignes}).

Il existe autant de clients MMS que de processus de contrôle pour l'embouteillage (donc que de lignes d'embouteillage). Chaque client est responsable du traitement complet de bout en bout des bouteilles d'un volume donné. Le point commun de ces différents processus se situe au niveau du remplissage des bouteilles. Il faut donc éviter qu'un client cherche à positionner une bouteille sur la plateforme occupée par une autre bouteille en cours de remplissage. La plateforme de remplissage constitue donc une ressource en accès mutuellement exclusif.

Nous associons à chaque capteur et à chaque actionneur une variable MMS située dans la VMD du dispositif correspondant. Les clients peuvent lire les variables associées aux capteurs pour obtenir le résultat des mesures et peuvent écrire dans les variables associées aux actionneurs pour déclencher l'action désirée. Chacun des actionneurs contrôlant le mouvement des convoyeurs des différentes lignes de bouteilles n'est accessible que par le client responsable de cette ligne.

F.1.2.1 Contrôle de la réaction chimique

Les capteurs de pH et de température permettent aux clients de savoir quand la réaction est terminée. Ceux-ci peuvent être consultés par les clients au moyen du service `Read`. Le client dont une des bouteilles se trouve sur la plateforme de remplissage effectue alors un `Write` sur la variable MMS modélisant la vanne de remplissage.

Le contrôle de la température du réservoir s'effectue au niveau de VMD_{res}. Nous utilisons les événements MMS de la façon suivante. Un EC de type scruté (noté EC_{temp}) est associé au capteur de température. Quand celui-ci dépasse un certain seuil, l'événement associé est déclenché. Tous les clients souscrivent à cet événement de sorte qu'ils sont informés du dépassement de température et peuvent interrompre momentanément leur processus. Par ailleurs, il existe un objet EA également lié à EC_{temp} dont le service est d'écrire dans la variable contrôlant le chauffage (dans ce cas pour l'éteindre). Cette façon de faire est étudiée en détail à la section 5.2. On utilise un scénario similaire pour rétablir le chauffage lorsque la température est trop basse.

On peut également contrôler l'arrivée des liquides A et B dans le réservoir de la même façon en associant un EC au capteur de niveau de liquide du réservoir et éviter ainsi tout risque de débordement. Le fait d'utiliser une action événementielle permet une réaction plus rapide que dans le cas où un client serait responsable du chauffage ou du contrôle des vannes A et B. En effet, ce client devrait d'abord recevoir la notification de l'événement considéré puis effectuer l'ordre correspondant. Dans le cas présent, l'ordre est effectué localement par VMD_{res} dès que surgit l'événement.

F.1.2.2 Contrôle de l'embouteillage

Le contrôle de l'embouteillage est représenté sur la figure F.2.

L'accès à la plateforme de remplissage est contrôlé par un sémaphore MMS banalisé (ou étiqueté) S à un seul jeton défini dans VMD_{lignes}. Quand un client est prêt pour remplir une bouteille (bouteille disponible, réaction dans le réservoir terminée), il doit au préalable obtenir le contrôle du sémaphore S. Ceci se fait au moyen du service `Take Control`. Si le sémaphore est occupé alors il existe déjà une bouteille en cours de remplissage et le client doit attendre. Quand le client obtient le sémaphore il peut écrire dans la variable associée au déplacement de la ligne de bouteilles dont il est responsable pour amener la bouteille sur la plateforme et la remplir.

Selon les besoins des applications considérées on peut utiliser les paramètres du service `Take Control` pour rendre plus prioritaire un client par rapport à un autre ou pour éviter une attente trop longue de la libération de la plateforme, etc.

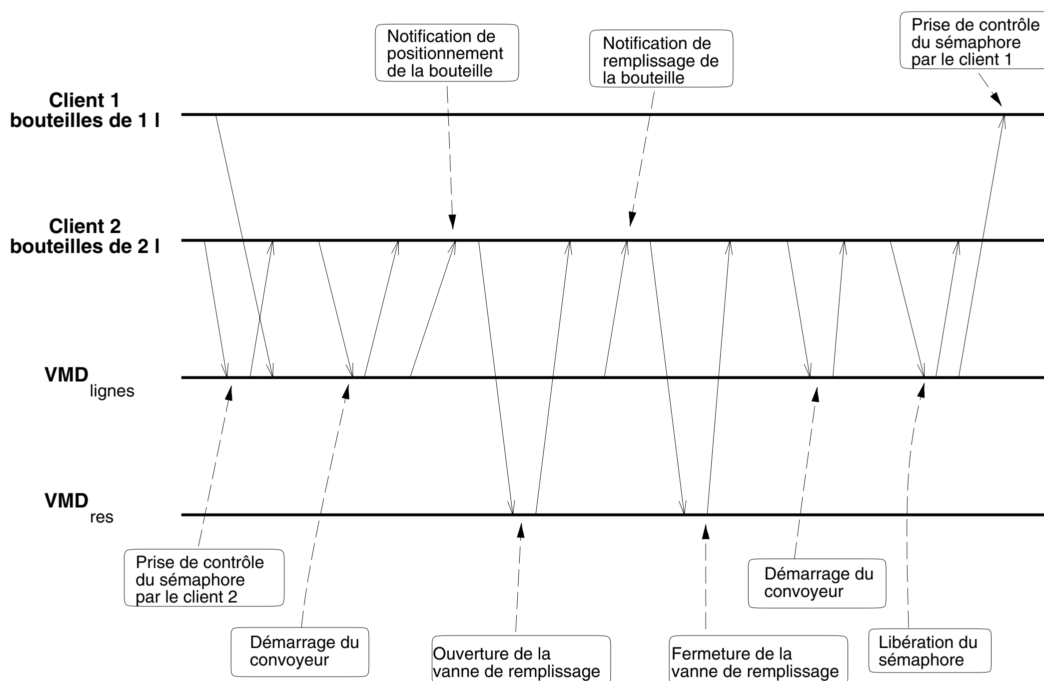


FIG. F.2 - Contrôle de l'embouteillage

On associe au capteur de position un événement qui là encore utilise une action permettant l'arrêt du convoyeur lorsque la bouteille est positionnée sur la plateforme. Par ailleurs, cet événement informe aussi le client concerné qu'il peut alors ouvrir la vanne de remplissage. Un autre EC associé au capteur de poids permet de savoir quand la bouteille est pleine. Toutefois, cette fois on ne peut pas utiliser une action pour arrêter le remplissage puisque l'événement se produit dans un serveur différent de celui commandant la vanne de remplissage. Il appartient donc au client qui reçoit la notification d'événement d'ordonner à VMD_{res} d'arrêter le remplissage. Nous proposons à la section 7.3.2 une solution plus simple qui évite de passer par le client. Notons également l'intérêt de la reconfiguration de XED pour changer les conditions d'apparition de l'événement en fonction du volume des bouteilles.

F.2 Scénario Renault : serveurs de terminaux

F.2.1 Le scénario

L'exemple qui suit décrit les exigences de comportement d'un système industriel réel chez le constructeur d'automobiles Renault. Nous ne décrirons pas tout le contexte de ce système en détail et renvoyons le lecteur à [Fuh93] pour plus d'informations.

L'environnement dans lequel nous nous plaçons est un atelier flexible où se produit l'assemblage de divers véhicules. Le système étudié est composé des éléments suivants :

- des *terminaux* à partir desquels des ordre sont entrés par des opérateurs humains;
- des *serveurs de terminaux* (ou ST) pouvant être connectés à plusieurs terminaux à la fois. Les STs sont des serveurs MMS;
- des *serveurs d'applications* (ou SA). Les SAs sont responsables de l'exécution d'un certain nombre d'*applications*. Ils sont liés à une base de données permettant d'accéder aux informations repré-

sentant une application. Un SA est un client MMS connecté à un ou plusieurs ST(s). Il utilise principalement les services `Input` et `Output`;

– des *applications*. Celles-ci définissent la séquence d'opérations intervenant entre un opérateur et un SA. L'exécution d'une de ces séquences d'opérations par un SA est appelée *travail*. Les opérateurs demandent le démarrage d'une application au moyen d'un terminal. Il ne peut y avoir qu'un seul travail actif par terminal. Mais une même application peut s'exécuter simultanément avec plusieurs terminaux.

Les SAs sont organisés par groupes en fonction des fonctionnalités qu'ils offrent. Les membres d'un groupe ne sont normalement pas connus des STs. Chaque groupe permet d'accéder à un certain ensemble d'applications. Dans chaque groupe, seul un SA communique avec un ST. Dans une certaine mesure, chaque membre d'un groupe peut être considéré comme une réplique des autres membres de ce groupe. L'architecture générale du système est représentée sur la figure F.3.

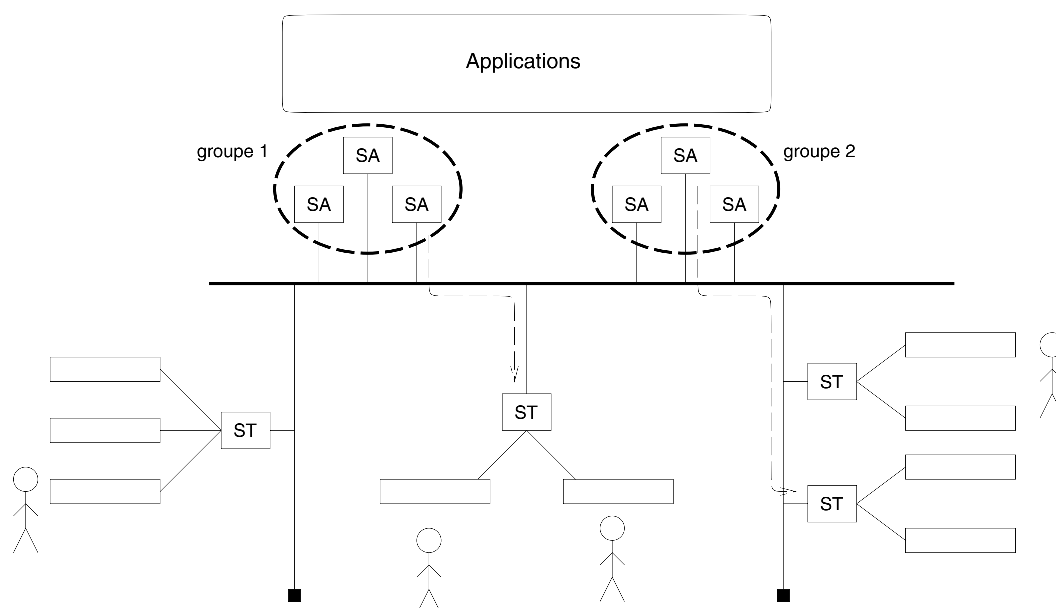


FIG. F.3 - Architecture générale du système

Le problème est de rendre le système tolérant aux fautes des SAs. On entend ici par faute l'arrêt pur et simple (autre que la terminaison normale) de l'exécution d'un SA. Dans un tel cas, tous les travaux actifs correspondant aux terminaux connectés au SA fautif doivent être rétablis avec un autre SA ou avec une réplique du SA en panne. Ce rétablissement doit être transparent pour les opérateurs. Pour assurer cette tolérance aux fautes, il nous faut réaliser les trois points suivants :

– informer les SAs de l'occurrence d'une panne;

– permettre la reprise automatique des travaux en cours après une panne;

– assurer que cette reprise ne se fait que par un seul SA.

F.2.2 Exemple de solution

F.2.2.1 Choix initial de l'application à exécuter

Chaque ST contient pour chaque application deux ECs prédéfinis nommés EC_{Init} et EC_{Fail} respectivement. Chaque SA définit deux EEs liés aux ECs correspondants aux seules applications que le SA peut exécuter. Par ailleurs, à chaque EC_{Fail} correspond un objet EA dont le service est de lire une variable $V_{travaux}$ dont la signification sera précisée plus loin.

Initialement, l'opérateur demande le démarrage d'une application à partir de son terminal. Cette opération déclenche dans le ST associé l'événement modélisé par EC_{Init}. Tous les SAs ayant souscrit à cet événement reçoivent alors une notification. Ces SAs correspondent à ceux capables d'exécuter l'application demandée par l'opérateur. Mais un seul SA doit servir l'opérateur. Il faut donc décider de celui qui va exécuter l'application requise.

Une solution simple consiste à associer à chaque terminal (donc à chaque objet "Operator Station") un sémaphore MMS banalisé (ou étiqueté) à un seul jeton. Tous les SAs qui reçoivent une notification effectuent ensuite une requête `Take Control` sur le sémaphore correspondant au terminal d'où vient la demande de démarrage de l'application. Un seul SA peut acquérir le sémaphore. Ce sera le SA qui exécutera l'application demandée. Pour éviter que les autres SAs restent en attente inutilement sur le sémaphore, on utilise l'option `Acceptable Delay` de la requête `Take Control`. Ce paramètre prend une valeur faible ou nulle de sorte que tous les SAs dont la requête arrive après la prise de contrôle du sémaphore sont informés qu'un autre SA est chargé d'exécuter l'application.

Cette phase d'"élection" du SA est illustrée sur la figure F.4.

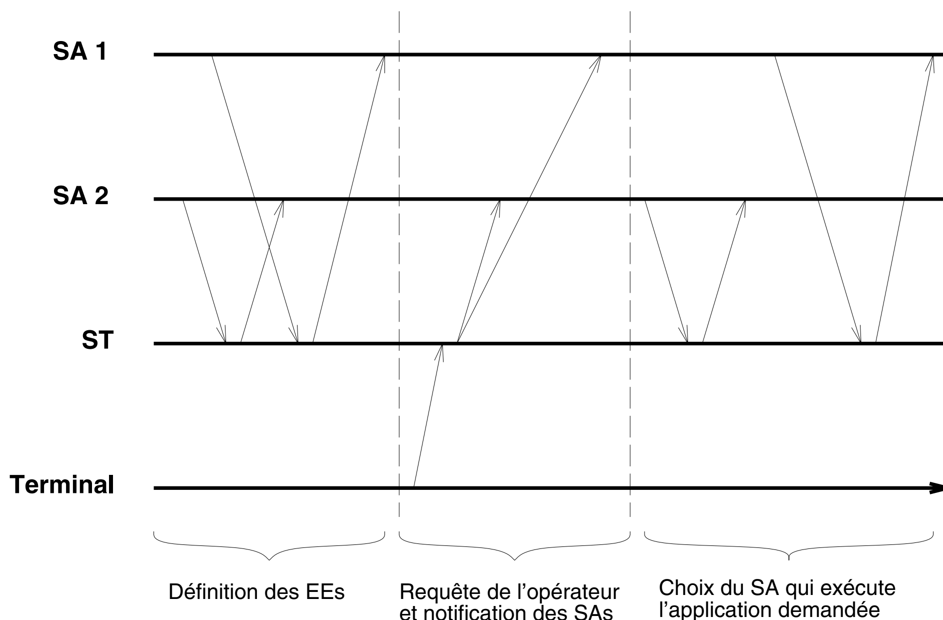


FIG. F.4 - Choix initial de l'application à exécuter

Le SA élu pour exécuter l'application libère le sémaphore lorsque l'exécution de cette dernière est terminée. Nous examinons dans la section suivante ce qui se passe en cas de panne de ce SA.

F.2.2.2 Exécution de l'application

Une fois le SA choisi, il initialise le travail correspondant à l'application sélectionnée. Ce travail consiste en une série d'étapes[1] dont chacune est numérotée. Le numéro d'une étape est communiqué au ST avant l'exécution de l'étape au moyen du service MMS `Write`. La variable MMS $V_{travaux}$ est utilisée dans le ST pour maintenir le numéro de l'étape en cours. En fait, la variable $V_{travaux}$ est une liste de structures à deux champs : le nom d'un objet "Operator Station" ayant demandé l'exécution de l'application considérée et le numéro de l'étape en cours pour ce terminal.

Si le SA tombe en panne, nous supposons que tous les STs ayant une association avec ce SA reçoivent un service `Abort`. Ces STs sont alors informés de la panne et déclenchent l'événement EC_{Fail} correspondant à l'application dont le SA est en panne. Tous les SAs capables d'exécuter cette application reçoivent donc une notification d'événement. Le déclenchement de l'événement implique l'exécution de l'action événementielle associée. La notification transporte donc le numéro de l'étape en cours d'exécution lors de la panne pour tous les travaux exécutant la dite application. Les SAs peuvent donc connaître tous les travaux interrompus, l'étape actuelle de ces travaux et le nom du terminal utilisé par ces travaux. Ces SAs se retrouvent alors dans la situation initiale exposée à la section précédente et effectue chacun une requête `Take Control` pour "élire" le SA qui continuera l'exécution au point où elle en était avant la panne.

Pour autoriser une prise de contrôle du sémaphore par un de ces SAs, le sémaphore acquis par le SA en panne doit auparavant avoir été libéré. Pour ceci nous exploitons l'option `Relinquish If Connection Lost` des requêtes `Take Control`. Nous affectons donc à VRAI ce paramètre dans toutes les requêtes `Take Control` effectuées par les SAs de sorte que toute perte d'association entraîne automatiquement la libération du sémaphore considéré.

La figure F.5 illustre le déroulement que nous venons de décrire.

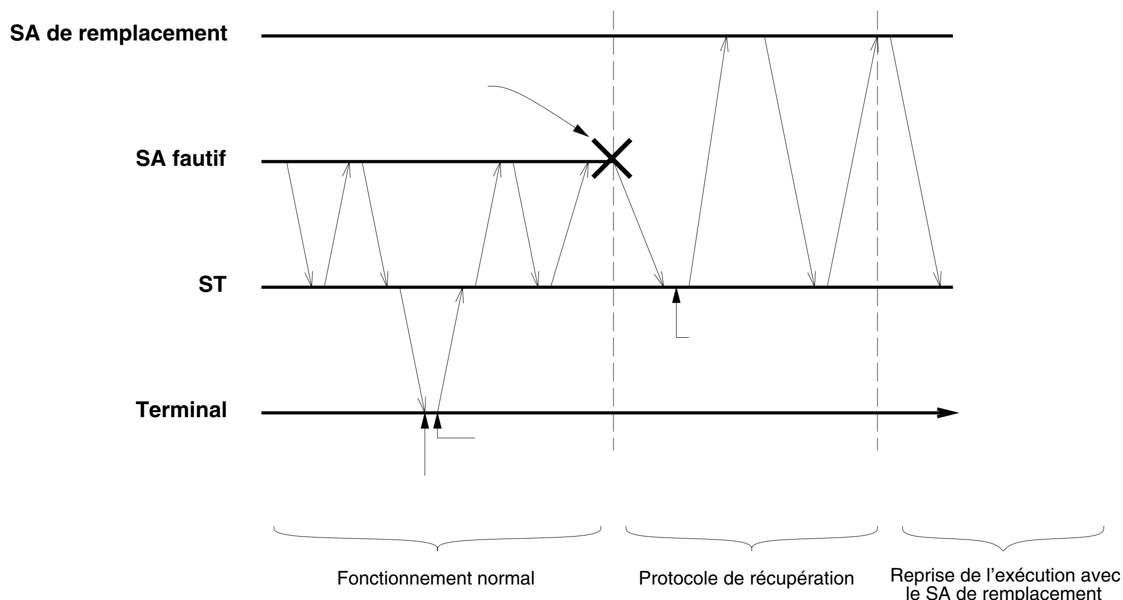


FIG. F.5 - Exécution de l'application et reprise lors d'une panne

Pour des raisons de concision, la solution présentée ici est simplifiée et légèrement modifiée par rapport à la solution originale proposée dans [Fuh93]. Il est clair que d'autres problèmes se posent tels que la nécessité de savoir si une étape a été ou non exécutée lorsqu'il y a une panne. Ceci est

[1]Typiquement, une étape est simplement un service `Input` ou `Output`.

important pour ne pas exécuter deux fois la même étape ou au contraire pour être sûr de bien l'exécuter une fois. Nous renvoyons le lecteur à [Fuh93] et [FMD94] pour une étude de ce type de problèmes. Ce scénario nous a toutefois permis d'illustrer l'utilité des événements MMS ainsi que de certaines options méconnues des sémaphores MMS.

F.3 Serveur d'impression

Dans les deux exemples précédents, nous n'avons utilisé que des sémaphores MMS à un seul jeton. Nous proposons dans la suite un scénario d'application comportant un sémaphore avec plusieurs jetons. L'exemple le plus classique d'utilisation des sémaphores avec plusieurs jetons consiste à faire jouer à un serveur MMS le rôle de serveur d'impression pour un nombre N de clients et n d'imprimantes. La figure F.6 illustre un tel cas pour $n = 3$.

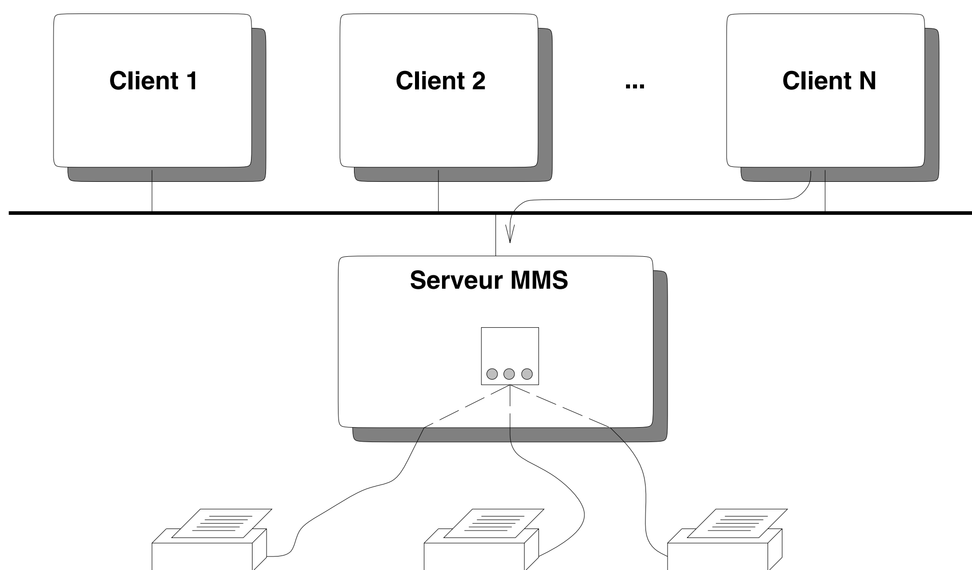


FIG. F.6 - Serveur d'impression

L'accès aux imprimantes est contrôlé par un sémaphore MMS banalisé (noté S) disposant de n jetons. Tout client désirant imprimer doit d'abord obtenir un jeton du sémaphore S. La prise de contrôle d'un jeton se fait avec le service `Take Control`. Une prise de contrôle réussie autorise le client à imprimer. Lors de l'acquisition d'un jeton, le serveur décide de l'imprimante qui sera utilisée par le client correspondant. Quand l'impression est terminée, le client libère le jeton détenu au moyen du service `Relinquish Control`.

Notons que l'on peut également utiliser un sémaphore étiqueté où chaque jeton correspond à une imprimante spécifique. Les jetons sont alors nommés et un client peut décider de l'imprimante utilisée en cherchant à obtenir le contrôle d'un jeton particulier.

Bibliographie

[ABRW91] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, GA, USA, 1991.

[AM95a] J. Akazan and Z. Mammeri. MMS and Time. In *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, June 1995.

[AM95b] J. Akazan and Z. Mammeri. On Tasks Synchronization with the MMS Protocol. *Journal of Real-Time Systems*, 9(3):265–287, November 1995.

[AM95c] J. Akazan and Z. Mammeri. Real-time extensions to MMS. In J.-D. Decotignie, editor, *Proceedings of IEEE International Workshop on Factory Communication Systems*, pages 193–200, Leysin, Switzerland, October 1995.

[And79] S. Andler. Synchronization Primitives and the Verification of Concurrent Programs. *Operating Systems: Theory and Practice*, pages 67–99, 1979.

[And91] G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49–90, March 1991.

[App85] Apple. *Inside Macintosh.* Addison-Wesley, 1985.

[ASU89] A. Aho, R. Sethi, and J. Ullman. *Compilateurs principes, techniques et outils.* InterEditions, 1989.

[Aud90] N. C. Audsley. Deadline Monotonic Scheduling. Technical report, Department of Computer Science, University of York, September 1990.

[Bac86] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc, 1986.

[BBF+70] C. Bétourné, J. Boulenger, J. Ferrié, C. Kaiser, S. Krakowiak, and J. Mossière. Process Management and Ressource Sharing in the Multiaccess System ESOPE. *Communications of the ACM*, 13(12):727–733, December 1970.

[BBHM95] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Applications. In *2nd International Workshop on Services in Distributed and Networked Environments*, pages 148–155, Whistler, British Columbia, June 1995.

[BCP+95] G. Berthet, P. Castori, P. Pleinevaux, F. Restrepo, K. Vijayananda, and F. Vamparys. Interoperability testing: Experience with MAP and CNMA. In M. H. Hamza, editor, *Proceedings of the 13th IASTED International Conference on Applied Informatics*, pages 60–63, February 1995.

[BDD+84] F. Browaeys, H. Derriennic, P. Desclaud, H. Fallour, C. Faulle, J. Febvre, J. E. Hanne, M. Kronental, J. J. Simon, and D. Vojnovic. Sceptre: proposition de noyau normalisé pour les exécutifs temps réel. *Technique et Science Informatiques*, 3(1):45–62, 1984.

[Ber93] G. Berthet. ASNADAC Conception et réalisation d'un compilateur ASN.1 vers Ada. Technical Report 175, EPFL-DI-LIT, 1993.

[BG81] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[BG91] M. Brill and U. Gramm. MMS: MAP application services for the manufacturing industry. *Computer Networks and ISDN Systems*, 21:357–380, November 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[Bla92] U. Black. *Network Management Standards*. Uyless Black Series on Computer Communications. McGraw-Hill, 1992.

[Boo86] G. Booch. *Software Engineering with Ada*. Benjamin Cumblings, second edition, 1986.

[BTE98] K. Botcherby, Y. Trinquet, and J. P. Elloy. Event Management and Rendezvous Concept in a Distributed Real Time Operating System. In M. G. Rodd and Th. Lalive d'Epinay, editors, *Proceedings of DCCS'88*, pages 43–48. IFAC, 1998.

[BTW95] A. Burns, K. Tindell, and A. Wellings. Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[Bur] A. Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. In Sang H. Son, editor, *Principles of Real-Time Systems*. Prentice Hall.

[Bur91] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, pages 116–128, 1991.

[BW90] A. Burns and A. J. Wellings. The Notion of Priority in Real-time Programming Languages. *Computer Languages*, 15(3):153–152, 1990.

[Cas94a] P. Castori. Dealing with Concurrency in MMS Servers. Technical Report 94/89, EPFL-DI-LIT, November 1994.

[Cas94b] P. Castori. XED: A Simple but Powerful Extension to MMS Events. In M. H. Hamza, editor, *Proc. of the IASTED International Conference on SYSTEMS AND CONTROL'94*, pages 1–5, June 1994.

[Cas95a] P. Castori. An Event-based Algorithm for Distributed Deadlock Detection/Resolution. In INRIA, editor, *Proceedings of the European Research Seminar on Advances in Distributed Systems*, pages 311–316, L'Alpe d'Huez, France, April 1995.

[Cas95b] P. Castori. An SQL Preprocessor for CIM Applications. In *Proceedings of INFORSID'95*, pages 381–391, Grenoble, France, May-June 1995.

[Cas95c] P. Castori. Distributed Concurrency Control Algorithms with an OSI Application Layer Protocol in Heterogeneous Environment. In *Proceedings of the Second International Conference on Industrial Automation*, volume 1, pages 219–224, Nancy, France, June 1995.

[Cas95d] P. Castori. On the Scheduling of MMS Services. In J.-D. Decotignie, editor, *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 219–224, Leysin, Switzerland, October 1995.

[Cas95e] P. Castori. Semaphores revisited with MMS. *Operating Systems Review*, 29(3):49–63, July 1995.

[CC95] ESPRIT Project 7096 CCE-CNMA. *CCE: An Integration Platform for Distributed Manufacturing Applications*. Springer-Verlag, 1995.

[Che96] N. Chevassus. Commentaires et notes de lecture sur cette thèse, March 1996. Private communication.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[CK96] P. Castori and S. Koppenhoefer. The Reader-Writer Problem Revisited with MMS. In *Proc. of the IASTED International Conference on APPLIED INFORMATICS'96*, Innsbruck, Austria, February 1996.

[CKST89] A. N. Choudary, W. H. Kohler, J. A. Stankovic, and D. Towsley. A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution. *IEEE Transactions on Software Engineering*, 15(1):10–17, January 1989.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, January 1985.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):633–646, October 1984.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[Con92] CNMA Consortium. *CNMA Implementation Guide Revision 5.1*. ESPRIT Project 5104 CNMA, November 1992.

[Coo91] J. E. Cooling. *Software Design for Real-time Systems*. Chapman and Hall, 1991.

[CP85] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems*. Mc Graw Hill, 1985.

[CP93] P. Castori and P. Pleinevaux. Structuration de la Gestion d'Événements MMS. *AGEN Mitteilungen*, 56–57:33–39, June 1993.

[CP95a] P. Castori and P. Pleinevaux. A Generic Architecture for MMS Servers. In J.-D. Decotignie, editor, *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 211–218, Leysin, Switzerland, October 1995.

[CP95b] P. Castori and P. Pleinevaux. Mechanisms to Ensure Ordered Execution in MMS. In *Proceedings of ETFA'95*, volume 2, pages 455–464, Paris, France, October 1995.

[CPR86] Y. F. Chen, A. Prakash, and C. V. Ramamoorthy. The Network Event Manager. In *Proceedings of the Computer Networking Symposium*, pages 169–178, Washington DC, November 1986.

[CPVV95] P. Castori, P. Pleinevaux, K. Vijayananda, and F. Vamparys. Interoperability testing in an Implementation of ISO/OSI Protocols. In L. M. Camarinha-Matos and H. Afsarmanesh, editors, *Balanced Automation Systems*, pages 241–252. Chapman & Hall, July 1995.

[Cri89] F. Cristian. A Probabilistic Approach to Distributed Clock Synchronization. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 288–296, Newport Beach, California, June 1989. Computer Society Press.

[CRO75] CROCUS. *Systèmes d'exploitation des ordinateurs.* Informatique. Dunod, 1975.

[CV95] P. Castori and F. Vamparys. Les Événements du Protocole MMS : Performances et Analyse. In *Proceedings of CFIP'95 Ingéniérie des Protocoles*, pages 167–182, May 1995.

[Dak90] Y. Dakroury. *Spécification et validation d'un protocole de messagerie multi-serveurs pour l'environnement MMS.* PhD thesis, Université de Nantes, École Nationale Supérieure de Mécanique, November 1990.

[DE89] Y. Dakroury and J-P. Elloy. A New Multi-Server Concept for the MMS Environment. In *Proc. of 9th IFAC Workshop on DCCS*, September 1989.

[DE95] Y. Dakroury and J. P. Elloy. A Distributed Transaction Processing Facility For the MMS Specification. In *Proceedings of the IEEE Symposium on Computers and Communication*, pages 407–413, Alexandria, Egypt, June 1995.

[DHM93] B. Delatte, M. Heitz, and J.F. Muller. *HOOD Reference Manual 3.1.* HOOD Technical Group, masson (paris) and prentice hall (london) edition, 1993.

[Dig82] Digital. *VAX Software Handbook.* Digital Equipment Corporation, 1982.

[Dij68a] E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. 1968. Academic Press.

[Dij68b] E. W. Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dij71] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138, 1971.

[DJJ89] M. Donner, D. Jameson, and W. Moran Jr. Events: A Structuring Mechanism For A Real-Time Runtime System. In *Proceedings of Real-Time Systems Symposium*, pages 22–30, Santa Monica, California, December 1989.

[Dun91] N. Dunstan. Semaphores for Fair Scheduling Monitor Conditions. *Operating Systems Review*, 25(3):27–31, July 1991.

[EC86] J. P. Elloy and D. Creusot. Study of MAP MMS Service Element Devoted to Events. Technical report, Renault-Automation, October 1986.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 11(19):624–633, November 1976.

[Ell87] J. P. Elloy. Analyse de la Proposition Event Management Functional Unit. Technical report, Renault-Automation, January 1987.

[EMR95] J.-P. Elloy, P. Molinaro, and R. Ricordel. A Temporal Extension to the MMS Protocol with KerNext Tool. In J.-D. Decotignie, editor, *Proceedings of the IEEE International Worshop on Factory Communication Systems*, pages 225–234, Leysin, Switzerland, October 1995.

[FHRT93] P. A. Franaszek, J. R. Haritsa, J. T. Robinson, and A. Thomasian. Distributed Concurrency Control Based on Limited Wait-Depth. *IEEE Trans. on Parallel and Distributed Systems*, 4(11):1246–1264, November 1993.

[FMD94] C. P. Fuhrman, S. Messina, and J.-D. Decotignie. Fault Tolerance Techniques Integrating MMS and Isis. In *Proceedings of IEEE Symposium on Emerging Technologies & Factory Automation*, Tokyo, Japan, 1994.

[Fou93] Open Software Foundation. *OSF DCE Application Development Guide.* Prentice-Hall, Inc, 1993.

[Fuh93] C. Fuhrman. Requirements Specification Renault Terminal Server Scenario, 1993. EPFL-LIT Internal Report.

[FV90] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communication*, 8(3):368–379, April 1990.

[GB95] T. M. Ghazalie and T. P. Baker. Aperiodic Servers in a Deadline Scheduling Environment. *Real-Time Systems Journal*, 9:31–67, 1995.

[Gen88] General Motors. *Manufacturing Automation Protocol. Version 3.0*, August 1988.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[Gra92] Dr C. R. Grant. MMS and Time-critical Communications in CIM. In *Proc. of the Eighth CIM-Europe Annual Conference*, pages 110–116, May 1992.

[Gro94] Object Management Group. *Common object services specification*, volume 1. J. Wiley and Sons, March 1994.

[Hab72] A. N. Habermann. Synchronization of Communicating Processes. *Communications of the ACM*, 15(3):171–176, March 1972.

[HB90] M. A. Huras and S. Bourbonnais. Manufacturing Message Specification: Achieving device independence on the factory floor. In *Proc. of IEEE Industrial Automation Conference and Exhibition*, April 1990.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, pages 666–677, August 1978.

[Hol] G. J. Holzmann. *Basic Spin Manual.* AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[Hol93] G. J. Holzmann. Design and Validation of Protocols: a Tutorial. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.

[HS92] W. A. Halang and K. M. Sacha. *Real-Time Systems Implementation of Industrial Computerised Process Automation.* World Scientific, 1992.

[IBFW86] J. D. Ichbiah, J. G. P. Barnes, R. J. Firth, and M. Woodger. *Rationale for the design of the Ada programming language.* Honeywell/Alsys, 1986.

[IC91] ISO/IEC and CCITT. *ISO/IEC 9595, CCITT X.710, Common Management Information Service Element Definition (CMISE).* International Standards Organization, 1991.

[Ins83] American National Standards Institute. *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.* ANSI, February 1983.

[ISO86] ISO/IEC. *Manufacturing Message Service for Bidirectional Transfer of Digitally Encoded Information, Part 1: Service Definition (Draft 5).* International Standards Organization, June 1986.

[ISO87a] ISO. *ISO 8824 Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).* International Standards Organization, May 1987.

[ISO87b] ISO/CCITT. *X.225/ISO 8327 Information processing systems - Open Systems Interconnection - Basic connection oriented session protocol specification.* ISO/CCITT, 1987.

[ISO88] ISO. Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1) ISO/IEC 8825, CCITT X.209, 1988.

[ISO90a] ISO/IEC. *ISO/IEC 9506-1, Industrial Automation Systems - Manufacturing Message Specification - Part 1: Service Definition.* International Standards Organization, 1990.

[ISO90b] ISO/IEC. *ISO/IEC 9506-2, Industrial Automation Systems - Manufacturing Message Specification - Part 2: Protocol Specification.* International Standards Organization, 1990.

[ISO92a] ISO. *ISO 10026/1, DTP, Part 1: OSI TP Model.* International Standards Organization, 1992.

[ISO92b] ISO. MMS Revised Defect Report 9506 / 3 and 4, 1992. ISO/TC 184/SC 5/WG 2.

[ISO92c] ISO. Technical Report of the TCCA rapporteurs' group of ISO/TC 184/SC 5/WG 2 identifying user requirements for systems supporting time critical communications, 1992. ISO/TC 184/SC 5/WG 2.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[JTK89] B-C. Jenq, B. C. Twichell, and T. W. Keller. Locking Performance in a Shared Nothing Parallel Database Machine. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):530–543, December 1989.

[Kop95] H. Kopetz. Real-Time Distributed Systems. In INRIA, editor, *Proceedings of the European Research Seminar on Advances in Distributed Systems*, pages 64–85, L'Alpe d'Huez, France, April 1995.

[Kra87] S. Krakowiak. Systèmes d'exploitation répartis. *Techniques et Science Informatiques*, pages 151–161, February 1987.

[KS91] A. D. Kshemkalyani and M. Singhal. Invariant-Based Verfification of a Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*, 17(8):789–799, August 1991.

[KS95] G. Koren and D. Shasha. D^{over} An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems. *Society for Industrial and Applied Mathematics*, 24(2):318–339, April 1995.

[LE93] A. M. Lister and R. D. Eager. *Fundamentals of Operating Systems.* The MacMillan Press Ltd, fifth edition, 1993.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM93] P.Y. Li and B. McMillin. Fault-Tolerant Distributed Deadlock Detection/Resolution. In *Proc. of COMPSAC'93*, pages 224–230, Phoenix, Arizona, November 1993.

[LMB92] J. Levine, T. Mason, and D. Brown. *Lex & Yacc.* Unix programming tools. O'Reilly, 1992.

[LPV94] M. Li, P. Pleinevaux, and F. Vamparys. Performance Evaluation of a Reduced OSI Stack. *Internetworking - Research & Experience*, pages 71–87, June 1994.

[LR80] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[LSM90] J. E. Lumpp, H. J. Siegel, and D. C. Marinescu. Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems. pages 476–483, 1990.

[LW82] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-time Tasks. *Performance Evaluation (Netherlands)*, 2:237–250, 1982.

[MA95] Z. Mammeri and J. Akazan. An approach to make MMS real-time. In *Proceedings of the IEEE International Symposium on Industrial Electronics ISIE'95*, volume 2, pages 586–591, Athens, Greece, July 1995.

[Mar81] A. J. Martin. An Axiomatic Definition of Synchronization Primitives. *Acta Informatica*, 16:219–235, 1981.

[MB76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[McE94] C. McElhone. Adapting and Evaluating Algorithms for Dynamic Schedulability Testing. Technical report, Department of Computer Science, University of York, February 1994.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor Scheduling in a Hard Real-time Environment. In *Proceedings of the 7th Texas Conference on Computer Systems*, pages 5.1–5.12, November 1978.

[Mey90] J. D. Meyer. Interoperability testing for HP MAP 3.0. *Hewlett-Packard Journal*, pages 50–53, August 1990.

[MLCS] D. C. Marinescu, J. E. Lumpp, T. L. Casavant, and H. J. Siegel. Models for monitoring and debugging tools for parallel and distributed software. *to appear in Journal of Parallel and Distributed Computing.*

[Mok83] A. K. L. Mok. *Fundamental Design Problems of Distributed Systems For The Hard Real-Time Environment.* PhD thesis, Laboratory of Computer Science, Massachssets Institute of Technology, 1983. MIT/LCS/TR-297.

[NC90] C. Nokely and G.K. Christensen. MMS Manufacturing Message Specification - The Most Important Building Block of CIM. In *International Conference in Automotive Technology and Automation*, 1990.

[NIS93] NIST. Workshop for Implementors of OSI. Stable Implementation Agreements for Open Systems Interconnection Protocols: Part 20 - MMS, December 1993.

[Nus86] H. Nussbaumer. *Informatique Industrielle II.* Presses Polytechniques Romandes, 1986.

[Nus91a] H. Nussbaumer. *Téléinformatique III.* Presses Polytechniques Romandes, 1991.

[Nus91b] H. Nussbaumer. *Téléinformatique IV.* Presses Polytechniques Romandes, 1991.

[Nut72] G. J. Nutt. Evaluation nets for computer system performance analysis. In *Proceedings of the Fall Joint Computer Conference*, pages 279–286, 1972.

[Ode94] J. J. Odell. Events and their specification. *Journal of Object-Oriented Programming*, July-August 1994.

[Pim90] J. R. Pimentel. *Communication Networks for Manufacturing.* Prentice Hall, 1990.

[Pir95] M. Pirazzi. Projet de Diplôme - Portage des gestionnaires d'événements et de sémaphores MMS en C, May 1995. EPFL - Laboratoire d'Informatique Technique.

[Ple90] P. Pleinevaux. *Architecture de Communication pour le Temps Réel Strict.* PhD thesis, École Polytechnique Fédérale de Lausanne, 1990.

[Ple92a] P. Pleinevaux. A General Architecture for an MMS Server, 1992. EPFL-LIT Internal Report.

[Ple92b] P. Pleinevaux. An SQL Interface to CCE, December 1992. ESPRIT CCE-CNMA Report TC.7096.92.04.

[Ple94a] P. Pleinevaux. An Analysis of the MMS Object Model. *IEEE Trans. on Industrial Electronics*, 41(3):265–268, June 1994.

[Ple94b] P. Pleinevaux. Integration of Industrial Applications: The CCE-CNMA Approach. In *Proceedings of IMSE'94, European Workshop on Integrated Manufacturing Systems Engineering*, pages 511–517, Grenoble, France, December 1994.

[Ple94c] P. Pleinevaux. Support of MMS services by major vendors, Version 1, June 1994. Document available using URL http://litwww.epfl.ch/~mms/mms_megapics.txt or using anonymous ftp at litsun.epfl.ch (128.178.151.135) in file pub/MMS/megapics.txt.

[Pre75] L. Presser. Multiprogramming Coordination. *Computing Surveys*, 7(1):21–44, March 1975.

[Ray92] M. Raynal. *Synchronisation et état global dans les systèmes répartis.* Eyrolles, 1992.

[Ray93] M. Raynal. Protocoles simples pour l'implémentation répartie des sémaphores. *Annales des Télécommunications*, 48(5–6):260–267, 1993.

[RCP93] T. Richner, P. Castori, and P. Pleinevaux. Architecture of an MMS Event Manager. In *Proceedings of COMPEURO'93*, pages 336–343, May 1993.

[Rea86] J. F. Ready. VRTX: A Real-Time Operating System for Embedded Microprocessor Applications. *IEEE Micro*, pages 8–17, August 1986.

[Roc86] M. J. Rochkind. *Advanced UNIX Programming.* Prentice-Hall, Inc, 1986.

[Ros92] J. P. Rosen. What Orientation Should Ada Objects Take? *Communications of the ACM*, 35(11):71–76, November 1992.

[RZI90] M.G. Rodd, G.F. Zhao, and I. Izikowitz. RTMMS - An OSI-based Real-Time Messaging System. *Journal of Real-Time Systems*, 2(3):213–234, November 1990.

[Sch86] A. Schiper. *Programmation concurrente.* Presses polytechniques romandes, 1986.

[Sel90] J. Self. *Aflex - An Ada Lexical Analyzer Generator, Version 1.1.* Department of Information and Computer Science, University of California, Irvine, May 1990.

[Shr76] S. K. Shrivastava. A view of concurrent process synchronisation. *The Computer Journal*, 18(4):375–379, 1976.

[Sie93] Siemens. *SINEC TF/DDE - Manager (Release 02)*, 1993.

[SL90] S-C. Shyu and V. O. K. Li. Performance Analysis of Static Locking in Distributed Database Systems. *IEEE Transactions on Computers*, 39(6):741–751, June 1990.

[SN85] M. K. Sinha and N. Natarajan. A Priority Based Distributed Deadlock Detection Algorithm. *IEEE Transactions on Software Engineering*, 11(1):67–80, January 1985.

[SP88] A. Silberschatz and J. L. Peterson. *Operating System Concepts.* Addison-Wesley, alternate edition, June 1988.

[SR87] J. A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proceedings of the Real-Time Systems Symposium*, pages 146–157, December 1987.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, September 1990.

[SRSC90] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A Real-Time Locking Protocol. *IEEE Transactions on Computers*, 40(7):793–800, July 1990.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

[Sta82] E. W. Stark. Semaphore Primitives and Starvation-Free Mutual Exclusion. *Journal of the ACM*, 29(4):1049–1072, October 1982.

[Sta93] W. Stallings. *SNMP, SNMPv2, and CMIP. The Practical Guide to Network Management Standards.* Addison-Wesley, April 1993.

[SVB93] D. Sidou, K. Vijayananda, and G. Berthet. An architecture for the implementation of OSI protocols: support packages, tools and performance issues. In *Proceedings of SI-CON/ICIE'93*, pages 98–102, Singapore, 1993.

[TBW92] K. W. Tindell, A. Burns, and A. J. Wellings. Mode Changes in Priority Pre-emptively Scheduled Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 100–109, 1992.

[Tho90] L. M. Thompson. Using pSOS+ for Embedded Real-Time Computing. In *COMPCON Spring 1990: Thirty-Fifth IEEE Computer Society International Conference*, pages 282–288, February 1990.

[TR91] A. Thomasian and I. K. Ryu. Performance Analysis of Two-Phase Locking. *IEEE Transactions on Software Engineering*, 17(5):386–402, May 1991.

[TTSC94] D. Taback, D. Tolani, R. J. Schmalz, and Y. Chen. *Ayacc User's Manual, Version 1.1.* Department of Information and Computer Science, University of California, Irvine, March 1994.

[VDC92] A. Valenzano, C. Demartini, and G. Cena. CIMNET: A Prototype Network for Time Critical Applications in the Factory. In *Proceedings of the 17th Conference on Local Computer Networks*, pages 466–472, Minneapolis, 1992.

[VER90] VERDIX Corporation. *VADS Verdix Ada Development System, Version 6.0, Sun-4 SunOS*, August 1990.

[Vie91] C. Vieillefond. *Mise en oeuvre du 68000.* Sybex, 1991.

[VP94] F. Vamparys and P. Pleinevaux. Architecture of the LITCommEngine, February 1994. EPFL-DI-LIT Internal Report.

[VP95] F. Vamparys and P. Pleinevaux. Conception et architecture du LITCommEngine. In *Proc of CFIP'95 Rennes France*, pages 241–256, May 1995.

[VvL72] H. Vantilborgh and A. van Lamsweerde. On an extension of Dijkstra's semaphore primitives. *Information Processing Letters*, pages 181–186, 1972.

[Wac93] M. Wack. *Méthode de description des communications dans l'atelier de production : messagerie MMS.* Lavoisier, 1993.

[ZR91] G.F. Zhao and M.G. Rodd. MAP for Real-time Distributed Computer Control Systems. In *Proceedings of the IFAC Workshop on DCCS*, pages 89–94, 1991.

[ZS90] Q. Zheng and K. G. Shin. On the Ability of Establishing Real-Time Channels in Point-to-Point Packet-Switched Networks. *IEEE Transactions on Communications*, 42(2/3/4):1096–1105, February/March/April 1990.

Curriculum Vitae

Pierre CASTORI, Ch. de la Dent d'Oche 4
1024 Ecublens (SUISSE), Tel 41-21-691.23.38

Etat civil : Né à Clermont-Ferrand, France, le 2 Avril 1968. Français et Canadien.

Langues : Français, Anglais, Italien, Espagnol.

Janvier 1992 - maintenant : Assistant, Laboratoire d'Informatique Technique, Ecole Polytechnique Fédérale de Lausanne, Suisse.

Février 1991 - Juillet 1991 : Concepteur d'un langage et de son compilateur destiné à la programmation d'applications industrielles, pour la société ARDATEM, France.

1988 - 1991 : Diplôme d'Ingénieur de l'EERIE (Ecole pour les Etudes et la Recherche en Informatique et Electronique), Nîmes, France.

1986 - 1988 : Mathématiques Supérieures - Mathématiques Spéciales, Clermont-Ferrand, France.

1986 : Baccalauréat C, Ecole Française de Milan, Italie.

Liste des publications

Publications dans des revues scientifiques

1. P. Castori, P. Pleinevaux, "Structuration de la gestion d'événements MMS", *AGEN Mitteilungen*, No 56/57, pp. 33-39. Juin 1993.
2. P. Castori, "Semaphores revisited with MMS", *Operating Systems Review*, 29(3):49–63. Juillet

1995.

Publications dans des conférences

3. P. Castori, P. Pleinevaux, T. Richner, "Architecture of an MMS Event Manager", *Proceedings of Compeuro'93*, pp. 439-444. Paris-Evry, France. 23-27 Mai 1993.
4. P. Castori "XED : A Simple but Powerful Extension to MMS Events", *Proceedings of IASTED Systems and Control'94*, pp. 1–5. Lugano, Suisse. Juin 1994.
5. G. Berthet, P. Castori, F. Restrepo, P. Pleinevaux, F. Vamparys, K. Vijayananda, "Interoperability testing: Experience with MAP and CNMA", *Proceedings of IASTED Applied Informatics'95*, pp. 60–63. Innsbruck, Autriche. Février 1995.
6. P. Castori, "An Event-based Algorithm for Distributed Deadlock Detection/Resolution", *Proceedings of ERSADS'95*, pp. 311-316. L'Alpe d'Huez, France. Avril 1995.
7. P. Castori, F. Vamparys, "Les Evénements du Protocole MMS: Performances et Analyse", *Proceedings of CFIP'95, Ingéniérie des protocoles*, pp. 167–182, Rennes, France. Mai 1995.
8. P. Castori, "Distributed Concurrency Control Algorithms with an OSI Application Layer Protocol in Heterogeneous Environment", *Proceedings of IAIA'95*, pp. 219–224. Nancy, France. Mai 1995.
9. P. Castori, "An SQL Preprocessor for CIM Applications", *Proceedings of INFORSID'95*, pp. 381–391. Grenoble, France. Mai-Juin 1995.
10. P. Castori, P. Pleinevaux, F. Vamparys, K. Vijayananda, "Interoperability in an Implementation of ISO/OSI Protocols", *Balanced Automation Systems*, pp. 241–252. Chapman & Hall. Vittoria, Brésil. Juillet 1995.
11. P. Castori, P. Pleinevaux, "A Generic Architecture for MMS Servers", *Proceedings of WFCS'95*, pp. 211–218. Leysin, Suisse. Octobre 1995.
12. P. Castori, "On the Scheduling of MMS Services", *Proceedings of WFCS'95*, pp. 219–224. Leysin, Suisse. Octobre 1995.
13. P. Castori, P. Pleinevaux, "Mechanisms to Ensure Ordered Execution in MMS", *Proceedings of ETFA'95*, volume 2, pp. 455–464. Paris, France. Octobre 1995.
14. P. Castori, S. Koppenhoefer, "The Reader/Writer Problem revisited with MMS", *Proceedings of IASTED Applied Informatics'96*, pp. 1-5. Innsbruck, Autriche. Février 1996.
15. P. Castori, "Event Management with TASE.2", *Proceedings of IASTED High Technology in the Power Industry'96.* Banff, Canada. Juin 1996.
16. P. Castori, "Ordonnancement des requêtes de service MMS", *Proceedings of CFIP'96, Ingéniérie des protocoles.* Rabat, Maroc. Octobre 1996.