Modèles et algorithmes de partage de données cohérents pour le calcul parallèle et distribué à haut débit.

Soumeya Leila Hernane

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAEM Lorraine

Modèles et algorithmes de partage de données cohérents pour le calcul parallèle distribué à haut débit

THÈSE

en co-tutelle

présentée et soutenue publiquement le 20 Juin 2013

pour l'obtention du

Doctorat

de l'université de Lorraine

(spécialité informatique)

et

de l'université des sciences et de la technologie d'Oran (USTO)

(spécialité informatique)

par

Soumeya-Leila Hernane

Composition du jury

Président :	Ye-Qiong Song :	Professeur à l'université de Lorraine
Rapporteurs :	Bouziane Beldjilali :	Professeur à l'université d'Es-senia d'Oran
	Hervé Guyennet :	Professeur à l'universite de Franche-Comté
Examinateurs :	Abdelkader Benyettou :	Professeur à l'universite des sciences et de la technologie d'Oran
	Flavien Vernier :	Maître de conférences à l'université de Savoie
	Nouredine Melab :	Professeur à l'université des sciences et technologies de Lille
	Jens Gustedt :	Directeur de recherche à l'Inria
	Mohamed Benyettou :	Professeur à l'université des sciences et de la technologie d'Oran

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Remerciements

Nombreuses sont les personnes qui m'ont aidée et soutenue durant toutes ces années de thèse, et celles qui ont contribué d'une manière ou d'une autre à la réalisation de ce travail.

Tout d'abord, je tiens à exprimer ma sincère gratitude à mon directeur de thèse, le directeur de recherche **Jens Gustedt**, sans qui ce travail n'aurait pu aboutir. Je le remercie pour la confiance qu'il m'a accordée dès le début en acceptant de diriger cette thèse, pour m'avoir accueillie dans l'équipe AlGorille du LORIA, et de m'avoir guidée durant toutes ces années de thèse.

Je remercie tout aussi respectueusement et chaleureusement mon directeur de thèse, le Professeur **Mohamed Benyettou** pour son soutien, son implication dans ce projet, sa bienveillance permanente, ainsi que pour ses encouragements.

J'adresse mes plus vifs remerciements aux professeurs **Bouziane Beldjilali** et **Hervé Guyennet** pour avoir accepté de rapporter ce manuscrit.

Je remercie également tous les autres membres du jury, à commencer par Monsieur **Ye-Qiong Song**, pour avoir accepté de présider le jury de soutenance. J'exprime ma reconnaissance à Messieurs **Abdelkader Benyettou**, **Flavien Vernier** et **Nouredine Melab**, pour m'avoir fait l'honneur de faire partie du jury. Je les remercie profondément pour leur présence et pour l'intérêt qu'ils ont porté à ce travail de recherche.

Merci à tous les membres de l'équipe AlGorille de m'avoir accueillie, sans oublier **Martin** pour ses orientations et ses remarques judicieuses.

Mes remerciements s'adressent également à mes anciens collègues de bureau, **Constantinos** et **Thomas**, pour leur gentillesse, et pour m'avoir orientée quand cela a était nécessaire. Je n'oublierai pas de citer les ingénieurs de l'équipe, **Sébastien**, **Tina** et **Mehdi**, d'avoir toujours répondu présents, chaque fois que j'ai eu recours à leur savoir faire, que ce soit en Grid'5000 ou en SimGrid.

J'exprime ma gratitude à toutes les parties qui ont contribué au financement de cette thèse, ainsi qu'à toutes les personnes que j'ai eu le plaisir de côtoyer à Nancy, et qui m'ont aidée d'une manière ou d'une autre dans mes démarches, en particulier, mes amis **Mounira** et **Mumtaz**.

J'adresse une attention particulière à ma sœur **Yasmina**, pour m'avoir encouragée et soutenue durant toutes ces années de thèse, et à ma mère pour m'avoir accompagnée lors de la phase finale de ce projet. Et pour finir, je ne manquerai pas de reconnaître que, si j'en suis arrivée là, c'est grâce à mes parents.

À mes très chers parents.

Table des matières

Table des figures **7**

Liste des tableaux **9**

1 Introduction générale **11**
1.1 Motivations 13
1.2 Contributions 13
1.3 Organisation du manuscrit 14

2 Librairies de communication et exclusion mutuelle : État de l'art **17**
2.1 Interfaces et librairies de communication 18
2.1.1 Fonctions POSIX 18
2.1.2 Message Passing Interface (MPI) 20
2.1.3 *Open Multi Processing* (OpenMP) 21
2.1.4 Étude comparative entre les différents paradigmes 22
2.1.5 SimGrid 23
2.2 L'exclusion mutuelle 24
2.2.1 Propriétés de l'exclusion mutuelle 25
2.2.2 L'inter-blocage 25
2.3 Taxonomie d'algorithmes distribués d'exclusion mutuelle pour les *locks* exclusifs 26
2.3.1 Algorithmes à permission 27
2.3.1.1 Algorithme de Lamport 27
2.3.1.2 Algorithme Ricart-Argawala 27

2.3.1.3 Algorithme Roucairol-Carvalho 28
2.3.2 Algorithmes basés sur la passassion d'un jeton 28
2.3.2.1 Algorithme de Suzuki et Kasami 29
2.3.2.2 Algorithme de Singhal 29
2.3.2.3 Algorithme de Le Lann 29
2.3.2.4 Algorithme de Kerry Raymond 29
2.3.2.5 Algorithme Naimi-Tréhel 30
2.4 Algorithmes distribués d'exclusion mutuelle de groupes 30
2.4.1 Problème noté *l-exclusion mutuelle* 30
2.4.2 Algorithmes basés sur les coteries 32
2.4.3 Algorithmes structurels 33
2.5 Algorithmes d'exclusion mutuelle pour les *locks* exclusifs et partagés . . 34
2.6 Variantes de l'algorithme Naimi-Tréhel 36
2.7 Synthèse du chapitre 37

3 L'interface *Data Handover* 39
3.1 Définitions générales 40
3.2 L'interface *Data Handover* 41
3.2.1 Les principes de base de **DHO** 41
3.2.2 Description détaillée de **DHO** 41
3.2.3 Cycle de vie de l'interface **DHO** 42
3.2.3.1 Modélisation et spécification formelle 45
3.2.3.2 Les différents chemins du cycle **DHO** 46
3.3 Critères d'évaluation et paramètres d'exécution 48
3.3.1 Durées observées 48
3.3.2 Plate-forme matérielle d'expérimentations 49
3.3.3 Modèle et librairie de communication 49
3.4 Analyse des résultats expérimentaux 50
3.4.1 Plan des expériences 50
3.4.2 Étude des délais de création et de destruction du *handle* 51
3.4.3 Étude du *mapping* et de la mise à jour de la ressource 52
3.4.4 Analyse de la performance du cycle **DHO** pour les *locks* exclusifs en l'absence de retard au blocage 52
3.4.5 Déduction de la durée du cycle pour les *locks* partagés 55

3.4.6 Évaluation du cycle pour les *locks* asynchrones 56
3.4.7 Évaluation du cycle pour les *locks* exclusifs et partagés 56
3.5 Synthèse du chapitre 58

4 Algorithme dynamique d'exclusion mutuelle à *locks* exclusifs (ADEMLE) 59
4.1 L'Algorithme distribué d'exclusion mutuelle de Naimi et Tréhel 60
4.1.1 Les bases de l'algorithme 60
4.1.2 Requêtes concurrentes 62
4.2 Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs (ADEMLE) 65
4.2.1 Conditions nécessaires au maintien de la connectivité des deux structures 65
4.2.2 Operations atomiques et notion d'état 65
4.2.3 Traitement d'une requête dans l'**ADEMLE** 66
4.2.3.1 Requêtes concurrentes 68
4.2.3.2 Complexité des messages envoyés pour l'envoi d'une requête 68
4.2.3.3 Notation formelle des opérations atomiques 69
4.2.3.4 Conditions préliminaires à l'envoi d'une requête 70
4.3 Traitement de la déconnexion d'un processus dans l'**ADEMLE** 73
4.3.1 Contraintes de déconnexion d'un processus 75
4.3.2 Modalités de déconnexion d'un processus 76
4.3.3 Envoi d'un ***Parent*** sortant à un demandeur de jeton 78
4.3.4 Complexité de départ d'un processus dans l'**ADEMLE** 78
4.4 Preuve de l'**ADEMLE** 79
4.4.1 Propriété de *vivacité* 79
4.4.1.1 Plan de preuve 79
4.4.2 Propriété de *sûreté* 80
4.5 Synthèse du chapitre 80

5 Algorithme dynamique d'exclusion mutuelle à *locks* exclusifs et partagés (ADEMLEP) 83
5.1 Algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés (**AEMLEP**) 84
5.1.1 Administration des requêtes mixtes 85
5.1.2 Complexité des messages envoyés dans l'**AEMLEP** 86

5.1.3 L'entrée en **section critique** d'une chaîne de lecteurs 86
5.1.4 Libération de la **section critique** 87
5.2 Preuve de l'**AEMLEP** 89
5.2.1 Propriété de *vivacité* 89
5.2.2 Propriété de *sûreté* 89
5.3 Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs et partagés (**ADEMLEP**) 89
5.3.1 Déconnexion d'un processus appartenant à la chaîne de lecteurs 90
5.3.2 Départ du *gestionnaire de lecteurs* 90
5.3.3 Départ du ***prochain écrivain*** 92
5.4 Preuve de l'ADEMLEP 92
5.4.1 Propriété de *vivacité* 92
5.4.2 Propriété de *sûreté* 93
5.5 Synthèse du chapitre 94

6 *Data Handover* dans un système peer-to-peer 95
6.1 L'API **DHO** dans le paradigme client-serveur 96
6.2 Objectifs de notre système distribué 97
6.3 **DHO** dans un système *peer-to-peer* 97
6.3.1 Cycle de vie **DHO** 98
6.3.2 Gestion des requêtes exclusives 99
6.3.3 Déroulement d'un exemple pour un accès exclusif 101
6.3.4 Mobilité des *peers* 103
6.3.4.1 Déconnexion des *peers* 104
6.3.4.2 Connexion d'un *peer* 104
6.3.5 Gestion des requêtes mixtes 105
6.4 Environnement expérimental 107
6.4.1 Plate-forme matérielle 107
6.4.2 Plate-forme logicielle 108
6.4.3 Scénarios 108
6.5 Méthodologie d'évaluation 109
6.5.1 Durées observées 110
6.6 Évaluation des performances du système *peer-to-peer* pour les *locks* exclusifs 110

6.6.1 Paradigme Client-serveur vs système *peer-to-peer* en l'absence de retard au blocage . . . 111
6.6.2 Impact de la répartition des *peers* sur les sites de la grille . . . 116
6.6.3 Évaluation du cycle **DHO** pour les *locks* asynchrones . . . 118
6.6.4 Évaluation du surcoût induit par la déconnexion des *peers* . . . 120
6.7 Observation du cycle **DHO** pour des requêtes asynchrones en accès partagé . . . 122
6.8 Évaluation du cycle **DHO** pour des requêtes mixtes . . . 124
6.9 Synthèse du chapitre . . . 126

7 Conclusion générale et prespectives **127**
7.1 Synthèse des travaux réalisés . . . 127
7.2 Perspectives . . . 128

Glossaire **129**

Bibliographie **131**

Table des figures

2.1 SimGrid 23

3.1 Intéractions entre le *client*, le *gestionnaire de ressource* et le *gestionnaire de verrou* (Serveur) 42
3.2 Cycle d'un *client* **DHO** 42
3.3 Diagramme d'états d'un *handle* 43
3.4 Durées observées : T_{DHO}, T_{Wait} et T_{Blocking} 53
3.5 Observation du cycle en variant T_{Wblocked}. Le temps du *lock* est fixé à $T_{\text{locked}} = 5(s)$. 57

4.1 Exemple d'exécution de l'algorithme Naimi-Trehel 62
4.2 Exemple de requêtes concurrentes dans l'algorithme Naimi-Trehel [NT87] 64
4.3 Exemple d'envoi d'une requête dans l'**AMDELE** 74

5.1 Un scénario avec des requêtes mixtes 86
5.2 Départ d'un ***prochain écrivain*** au milieu de deux chaînes de lecteurs . . 92

6.1 Implication des *peers* dans l'appel des fonctions 99
6.2 Intéractions entre processus de chaque *peer* 101
6.3 Déroulement d'un exemple et illustration des étapes traversées par la paire d'états $\{$*handle*, *gestionnaire de verrou*$\}$ 103
6.4 Représentation des fonctions **DHO** de l'étape (e) de la figure 6.3 103
6.5 Diagramme d'états d'un *handle* **DHO** 106
6.6 Structure globale d'une application sous **DHO** 108
6.7 Comparaison du temps moyen du cycle $\overline{T_{\text{DHO}}}$ entre le modèle centralisé et le système *peer-to-peer*, en variant la taille et le temps d'occupation de la ressource, T_{locked}. 113
6.8 Comparaison des durées moyennes $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et $\overline{T_{\text{Blocking}}}$ entre le modèle centralisé et le système *peer-to-peer* en diversifiant la taille de la ressource et le temps applicatif T_{locked} 115

6.9 Gain observé suite à la répartition des *peers* sur les sites de la grille. La taille de la ressource est fixé à 50 Mo et la durée du *lock* (T_{locked} à 5(s)) . . 117
6.10 Observation des durées $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et $\overline{T_{\text{Blocked}}}$ dans un système à 50 *peers* en prenant une ressource de 50 Mo. 119
6.11 Coût produit par un sous ensemble de *peers* en mouvement sur la durée moyenne du cycle **DHO** 121
6.12 $\overline{T_{\text{DHO}}}$ et $\overline{T_{\text{Wait}}}$ pour les *locks* partagés dans un système à 50 *peers* en prenant une ressource de 50 Mo 123
6.13 Durées moyennes du cycle $\overline{T_{\text{DHO}}}$. Pour chaque valeur T_{locked} du premier cycle, plusieurs valeurs sont prises dans le deuxième. Les deux figurent exposent les mêmes résultats pour une ressource de 50 Mo. 124
6.14 Effet du retard au blocage, de part et d'autre dans les deux types d'accès. T_{locked} est fixé à 10(s) tandis que la taille de la ressource est de 50 MiB . . 126

Liste des tableaux

2.1 Étude comparative entre les paradigmes de partage de données 22
2.2 Taxonomie des algorithmes d'exclusion mutuelle 31

3.1 Table de transition d'états. Pour une question de lisibilité, seuls les états exclusifs sont représentés 46
3.2 Les délais de création et de destruction du *handle* 51
3.3 Durées observées en prenant une bande passante BW $= 1.25 \cdot 10^8 (B/s)$ et une latence L $= 10^{-5}(s)$. 54
3.4 Moyenne du cycle $\overline{T_{\text{DHO}}}$. La taille de la ressource est fixée à 100 `MiB`. La durée du *lock* (T_{locked}) est prise dans l'intervalle $0 \ldots 10(s)$. 55
3.5 Moyenne du cycle $\overline{T_{\text{DHO}}}$. La durée du *lock* ($T_{\text{locked}}$) et le retard au blocage (T_{Wblocked}) sont pris dans l'intervalle $0 \ldots 10$(s), la bande passante BW $= 1.25 \cdot 10^8$ (B/s) et la latence L $= 10^{-5}$(s). 57

6.1 États relatifs à l'ensemble $\{$*peer*, *handle*, *gestionnaire de verrou*$\}$ de notre système distribué. 104
6.2 Moyennes des durées observées (système *peer-to-peer*) 111
6.3 Diminution du cycle occasionné par la répartition des *peers* sur les sites de la grille 116
6.4 Coût généré par le mouvement d'un sous ensemble de *peers*. Les résultats correspondent aux *peers* qui achèvent un cycle **DHO** 121

Chapitre 1

Introduction générale

Sommaire

1.1 Motivations 13
1.2 Contributions 13
1.3 Organisation du manuscrit 14

La pluralité grandissante des technologies informatiques ces dernières années a fait jaillir de nouvelles idées pour d'innombrables applications multidisciplinaires. La puissance de calcul, la capacité de stockage, la vitesse de transfert et la mutualisation des ressources sont actuellement des termes récurrents des systèmes distribués.

La quête d'une puissance de calcul continuelle justifiée par des applications de plus en plus gourmandes contraint les chercheurs à reconsidérer les modèles et paradigmes de programmation.

Les grilles informatiques, les systèmes *Peer-to-peer* et le *Cloud computing* s'accordent pour atteindre cette puissance. Ils regroupent des infrastructures matérielles et logicielles qui se caractérisent par une grandeur d'échelle, voulue par les scientifiques. La finalité de ces systèmes est d'offrir des services aux utilisateurs par des d'ordinateurs dispersés sur tout le globe.

La définition du terme grille informatique ou *Grid Computing* est très éparse dans la littérature. Le concept a été introduit par Ian Foster [Fos01]. Il définit une grille comme étant une organisation virtuelle, fondée sur une architecture massivement distribuée, dédiée au calcul scientifique. Depuis, d'autres définitions ont enrichi la littérature pour faire valoir le service pour lequel elles sont dédiées. Ainsi, plusieurs projets ont suivi [CFK+99], [And04], [ACK+02] et [GNFC00]. Les grilles de calcul, de données, de stockage, *Desktop Grid* et le *Cloud computing* ont pour point commun la virtualisation des ressources. En revanche, ils se distinguent par des objectifs plus ou moins différents.

Les systèmes *Peer-to-peer* sont apparus pour décentraliser des systèmes, auparavant basés sur des serveurs de données confrontés à des problèmes de goulot d'étranglement et de passage à l'échelle. Dans un monde où des quantités considérables de données nu-

mériques sont produites chaque jour, ces systèmes apportent l'infrastructure nécessaire pour que les données soient accessibles. En particulier, les systèmes *Peer-to-peer* permettent de partager des objets répartis sur des ordinateurs selon une politique d'égal à égal. Gnutella [CRB+03] et Napster [CG01] sont des exemples typiques.

Quoiqu'ils ciblent des objectifs différents, les grilles informatiques et les systèmes *Peer-to-peer* sont exposés à des contraintes identiques et présentent certaines similitudes dans leur mode de fonctionnement. Nous pensons qu'un système hybride tire profit des services offerts par les deux systèmes.

Plus généralement, les systèmes distribués à grande échelle offrent l'avantage d'être économiques de par les services qu'ils proposent. Par contre, leur structure complexe peut être un handicap pour une utilisation optimale de leurs ressources. En effet, la simplicité d'usage dans un environnement aussi complexe est un objectif à atteindre. Nous focalisons dans cette thèse sur la tansparence d'accès aux ressources dans les systèmes hétérogènes à haut débit.

En réalité, la productivité de tels systèmes dépend essentiellement des modèles et paradigmes dont ils émanent. En remontant un peu dans l'histoire, nous voyons que ces paradigmes étaient classés dans les architectures à mémoire partagée ou à mémoire distribuée. Les deux paradigmes prennent en charge l'accès et la cohérence de données. Néanmoins, ils opèrent différemment : Les *threads* sont typiquement déployés sur les environnements à mémoire partagée (à l'image de `POSIX-Pthread` [TSUS] et `OpenMP` [ope]), alors que les librairies basées sur l'API MPI [mpia] sont largement utilisées dans les architectures à mémoire distribuée.

La performance de ces paradigmes a été démontrée dans la mesure où ils sont déployés dans leur environnement dédié. Ils se montrent toutefois moins éfficaces sur des plates-formes dont la structure et la taille ne sont pas connues à priori. En effet, les systèmes distribués à grande échelle vérifient ces propriétés. D'autres attributs non moins importants caractérisent ces systèmes et contraignent les chercheurs à reconsidérer les modèles et paradigmes existants. Ainsi, la méconnaissance de la taille du système, la mobilité des nœuds et la volatilité des ressources sont des facteurs qui complexifient d'autant plus leur mise en œuvre.

D'un autre côté, un vaste panel d'algorithmes distribués d'exclusion mutuelle ont accompagné cette avancée technologique. Ces algorithmes répondent au besoin de partage de ressources entre processus concurrents, selon des protocoles appropriés. Ainsi, l'on trouve des algorithmes structurels ou non, basés sur les permissions comme ceux fondés sur la passation d'un jeton. Par ailleurs, la complexité en terme de messages envoyés a fait l'objet de plusieurs travaux scientifiques.

Nous avons choisi de nous intéresser à un algorithme structurel d'exclusion mutuelle, basé sur la passation d'un jeton. La structure dynamique de l'algorithme de Naimi et Tréhel [NTA96a] se prête parfaitement à la grandeur d'échelle de ces systèmes.

Dans cette thèse, l'algorithme Naimi-Tréhel est repris, puis enrichi par de nouvelles propriétés qui satisfont les exigences citées plus haut.

1.1 Motivations

Dans [Gus06], l'auteur a mis l'accent sur les problématiques diverses que présentent les paradigmes les plus connus de leur catégorie (OpenMP et MPI). Par ailleurs, une interface de programmation dénommée *Data Handover* (**DHO**) a été proposée. Le présent travail s'inscrit en continuité de cet article. Dans notre approche, nous tirons profit des paradigmes et librairies de communication existantes, tout en considérant les aspects liés à l'hétérogénéité et à la scalabilité des systèmes actuels. Particulièrement, les propriétés suivantes doivent être vérifiées :

Accès cohérent aux ressources et scalabilité L'API **DHO** doit fournir des routines pour l'acquisition des ressources en lecture ou en écriture en un temps fini, avec une simplicité d'usage. Son intégration ne doit pas altérer le code existant de l'application. De plus, **DHO** doit assurer les propriétés d'intéropérabilité et de scalabilité.

Abstraction de données **DHO** introduit un niveau d'abstraction entre la donnée et la mémoire, par le biais d'objets appelés *handles*, selon une politique d'accès à la donnée. L'utilisateur invoque une ressource et demande son instanciation en mémoire locale par deux opérations conjointes : Le *locking* et le *mapping*. En outre, l'architecture sous-jacente du système doit être dissimulée à l'utilisateur. Ce dernier n'est astreint de connaître que l'identifiant de la ressource sollicitée.

Volatilité des nœuds Nous prônons pour une API qui évolue au sein d'un nuage informatique, où les nœuds hébergeant des ressources peuvent joindre comme ils peuvent quitter le système, sans que cela n'affecte le déroulement de l'application.

1.2 Contributions

Proposition du modèle **DHO** Nous proposons un modèle élaboré pour l'API **DHO** conformément aux prescriptions formulées dans [Gus06]. Nous décrivons les évènements successifs produits par l'enclenchement des routines de l'API. Ces évènements forment un cycle qui débute à l'insertion d'une requête pour terminer à la résiliation de la ressource. Nous présentons une modélisation du cycle de vie **DHO** par un automate d'états finis qui fait ressortir les phases requises pour un *locking/mapping*.

Nous développons ensuite une première approche dans laquelle les routines de l'API évoluent dans un contexte centralisé. Dans ce modèle, des *clients* se disputent l'accès en lecture ou en écriture à une ressource hébergée sur un serveur, par des requêtes organisées et servies selon un ordre FIFO. L'implémentation est basée sur les `Sockets` et est réalisée en exploitant la librairie `GRAS` [Qui06a] (*Grid Reality and Simulation*) et le modèle de communication de `SimGrid` [CLQ08]. Les expériences sont menées en simulation puis, reconduites sur un environnement réel : Un *Cluster* de la plate-forme Grid'5000.

Extensions de l'algorithme Naimi-Tréhel Notre deuxième contribution est d'ordre théorique. Comme nous l'avons précisé plus haut, notre motivation est de fournir un modèle approprié aux systèmes distribués hétérogènes. Pour cela, nous nous orientons vers l'algorithme de Naimi et Tréhel. Compte tenu de la complexité de sa structure, nous avons choisi de présenter nos extensions séparément dans différents algorithmes, avant de les réunir dans un algorithme récapitulatif. Notre contribution s'étale sur différentes phases :

D'abord, nous reconsidérons le procédé d'acheminement des requêtes afin de le réduire à une opération atomique. Cette contribution a pour but de rendre possible la disparition des nœuds du système, sans que le déroulement de l'application n'en soit affecté. Ainsi, l'algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs (**ADEMLE**) est notre première extension à l'algorithme Naimi-Tréhel. Nous apportons une description formelle avant de prouver théoriquement les propriétés de *sûreté* et de *vivacité* de l'exclusion mutuelle.

Comme son nom l'indique, l'algorithme d'exclusion mutuelle pour les *locks* exclusifs et **partagés** (**AEMLEP**) considère les deux types de requêtes.

Enfin, l'algorithme **dynamique** d'exclusion mutuelle pour les *locks* exclusifs et partagés (**ADEMLEP**) résulte d'une hybridation de l'**ADEMLE** et de l'**AEMLEP**. Nous en apportons la preuve théorique.

Intégration de nos algorithmes étendus au modèle DHO Finalement, nous proposons une architecture qui s'étend sur trois niveaux dans laquelle les algorithmes **ADEMLE**, **AEMLEP** et **ADEMLEP** opèrent au plus bas de l'échelle, grâce à des processus intéragissant avec les gestionnaires du niveau supérieur. Notre modèle s'apparente à un système *peer-to-peer* guidé par les routines **DHO** invoquées plus haut.

L'implémentation ardue de ce modèle exploite les mêmes outils que précédemment (libraire `GRAS` de `SimGrid`) . Nous menons les expériences en environnement réel, par une réservation préalable de nœuds dans Grid'5000. Suite à des *Benchmarks* poussés, le comportement de l'API **DHO** est analysé et comparé à l'approche contralisée.

1.3 Organisation du manuscrit

Le manuscrit ci-présent est organisé comme suit :

Le **chapitre 2** présente un état de l'art des paradigmes de programmation parallèle et distribuée et des algorithmes distribués d'exclusion mutuelle. Les fondements de bases de : *locking* en lecture/écriture, *mapping*, gestion des accès dans une mémoire commune et des communications entre processus sont énoncés. Une étude comparative entre ces paradigmes est présentée selon des critères qui nous intéressent. Enfin, nous terminons cette partie par la présentation du *Toolkit* `SimGrid`, qui inclut la librairie de notre choix (`GRAS`) dans la totalité de nos expériences.

La deuxième partie de ce chapitre dresse un panorama des différentes classes d'algorithmes distribués d'exclusion mutuelle. Entre les algorithmes à permission,

à passation de jeton, structurels et de groupe, une analyse détaillée est fournie, suivie des contributions apportées à l'algorithme sélectionné dans cette étude : L'algorithme **Naimi-Tréhel** [NTA96a].

Le **chapitre 3** présente l'interface *Data Handover*. Nous commençons par décrire les phases incontournables à l'accomplissement d'un cycle, que nous schématisons par un diagramme d'états et modélisons par un automate d'états finis. S'ensuit une étude expérimentale de notre approche, menée sur un *Cluster* de la grille Grid'5000. Nous mettons en avant les durées qui composent le cycle dans le procédé d'évaluation. Suite à des tests intensifs qui englobent une large variété de paramètres, nous étudions les performances de notre approche par le système des files d'attente de type M/M/1.

Les **chapitres 4** et **5** explicitent des extensions multiples à l'algorithme originel de Naimi et Tréhel. L'**ADEMLE** révise le parcours d'une requête dans la structure arborescente de l'algorithme de base, et énonce les dispositions de départ d'un processus. Nous démontrons par preuve de théorème les propriétés d'exclusion mutuelle de notre algorithme étendu.

Avant de présenter l'**ADEMLEP**, nous recentrons notre étude autour des demandes de *locks* exclusifs et partagés, d'où la proposition d'un deuxième algorithme **AEMLEP**. Enfin, l'**ADEMEP** conclut cette partie par l'agrégation de toutes les propriétés qui définissent notre apport théorique. Par ailleurs, l'analyse de la complexité en terme de messages envoyés prend part de cette étude.

Le **chapitre 6** expose notre système *peer-to-peer*, issu du modèle de base de **DHO** présenté au chapitre 3, et des algorithmes proposés dans les chapitres 4 et 5. Le tout est organisé dans une architecture en couches avec pour sommet l'API **DHO**. Ce chapitre synthétise notre travail par un modèle distribué dédié aux plates-formes hétérogènes à haut débit. L'évaluation des performances est effectuée selon un plan d'expériences analogue à celui de l'approche centralisée précédemment étudié (chapitre 3), en prenant comme base la plate-forme Grid'5000. Nous en tirons les singularités du modèle distribué.

Enfin, le **chapitre 7** conclut l'ensemble de nos travaux et ouvre des perspectives sur des contributions futures.

Chapitre 2

Librairies de communication et exclusion mutuelle : État de l'art

Sommaire

2.1 Interfaces et librairies de communication 18
2.1.1 Fonctions POSIX 18
2.1.2 Message Passing Interface (MPI) 20
2.1.3 *Open Multi Processing* (OpenMP) 21
2.1.4 Étude comparative entre les différents paradigmes 22
2.1.5 SimGrid 23
2.2 L'exclusion mutuelle 24
2.2.1 Propriétés de l'exclusion mutuelle 25
2.2.2 L'inter-blocage 25
2.3 Taxonomie d'algorithmes distribués d'exclusion mutuelle pour les *locks* exclusifs 26
2.3.1 Algorithmes à permission 27
2.3.2 Algorithmes basés sur la passassion d'un jeton 28
2.4 Algorithmes distribués d'exclusion mutuelle de groupes 30
2.4.1 Problème noté *l-exclusion mutuelle* 30
2.4.2 Algorithmes basés sur les coteries 32
2.4.3 Algorithmes structurels 33
2.5 Algorithmes d'exclusion mutuelle pour les *locks* exclusifs et partagés 34
2.6 Variantes de l'algorithme Naimi-Tréhel 36
2.7 Synthèse du chapitre 37

CEt état de l'art couvre deux volets. Le premier s'articule autour des APIs communément utilisés dans les architectures parallèles et distribuées. À la section 2.1, nous mettons l'accent sur les aspects de gestion des *locks* pour le partage de données en lecture/écriture avant de conclure par une étude comparative qui fait ressortir les singularités entre les différents paradigmes. À la fin de cette section, nous présentons les majeures parties du *Toolkit* `SimGrid` pour lequel nous avons opté.

Le deuxième volet aborde la problèmatique de l'exclusion mutuelle ainsi que les premières solutions qui y sont apportées avant de présenter une étude comparative entre les différentes classes existantes. Nous survolons les algorithmes d'exclusion mutuelle à entrées multiples et nous présentons les variantes les plus récentes de l'algorithme de Naimi et Tréhel (section 2.2).

2.1 Interfaces et librairies de communication

Cette section résume les librairies standardisées qui sont maîtresses par leurs capacités à faire valoir les environnements parallèles et/ou distribuées dans leur catégorie.

2.1.1 Fonctions POSIX

L'interface `POSIX` a été définie en 1988, elle regroupe les APIs et logiciels standards destinés au systèmes compatibles `UNIX`. En d'autres termes, elle définit une librairie qui manipule les *threads*, l'accès aux fichiers par les réseaux, la gestion des signaux entre processus, la gestion de la mémoire ainsi que les routines de verrouillage.

La standardisation de l'API *multi-threading* a été spécifée par IEE POSIX 1003.1c [TSUS]. Connue sous le nom de `POSIX-threads` ou `Pthreads`, elle inclut les routines nécessaires aux opérations de *locking* en lecture/écriture pour des objets référencés par `rwlock`. En voici quelques unes :

Listing 2.1 – L'API POSIX-Pthreads

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Initialement, l'objet `rwlock` est dans un état *Unlock*. L'appel de `pthread_rwlock_rdlock` traduit une demande d'un *lock* en lecture qui est satisfaite si :

(1) Aucun écrivain ne le détient et (2) Aucun écrivain n'est bloqué pour le même *lock*. Si les deux conditions ne sont pas vérifiées, la fonction diffère son retour jusqu'à l'obtention effective du *lock*.

Afin d'éviter la famine des écrivains, `Pthreads` favorise les demandes en écriture à raison de quatre pour un lecteur. La routine `pthread_rwlock_tryrdlock` tente d'acquérir le même *lock* en lecture sauf que le retour de celle-ci est immédiat. De ce fait, Il peut être annonciateur d'un échec.

De façon analogue, les fonctions `pthread_rwlock_wrlock` et `pthread_rwlock_trywrlock` se comportent de manière similaire lorsqu'il s'agit de demander un *lock* en écriture. Aussi, la fonction `pthread_rwlock_unlock` est appelée pour la libération du *lock*, qu'il soit en lecture ou en écriture. Pour ce qui est de la résiliation en lecture, l'objet en question demeure dans l'état *read-locked* jusqu'à résiliation de tous les lecteurs [Dha06].

L'API que nous proposons s'inspire fortement de `Pthreads` à la différence que nous appliquons l'équité totale entre les demandes reçues, au moyen d'une queue FIFO et d'une structure dynamique.

Sous un autre angle, le *file-locking* est un autre mécanisme offert par `POSIX` utilisé fréquemment pour le contrôle des fichiers. Il est fourni par la fonction `fcntl`. On distingue deux modes : Le *Mandatory* et l'*Advisory* [TOGBSIIS].

Dans le mode *Mandatory*, le noyau du système se charge de protéger le *lock* contre d'éventuelles tentatives d'accès par d'autres processus. En outre, le deuxième mode offre plus de flexibilité dans la mesure où les processus peuvent acquérir le statut du *lock* préalablement à une demande. Ainsi, ce mécanisme contraint les processus à prendre l'action à suivre. Ils doivent, soit procéder aux opérations d'entrée/sortie, soit attendre la libération du *lock* en question.

Le mode *Mandatory* supporte le *locking* par segments de fichiers. Autrement dit, les fichiers sont rangés en octets permettant la sélection d'une rangée à partir d'un *offset*. De ce fait, plusieurs processus peuvent potentiellement partager différents segments d'un même fichier avec des *locks* différents. L'activation de ce mode est précisée dans les fichiers individuels mais aussi dans le système de fichiers.

Listing 2.2 – Exemple de file locking

```
int fd = open ("M␣o␣n␣f␣i␣c␣h␣i␣e␣r" , O_WRONLY);
struct flock fl = {
 .l_type = F_WRLCK,
 .l_whence = SEEK_SET,
 .l_start = 0,
 .l_len = 102400,
 .l_pid = getpid(),
};
fcntl(fd , F_SETLKW, &fl) ;
```

L'exemple du listing 2.2 montre une structure `flock` à plusieurs champs à travers lesquels est défini le fonctionnement du *file-locking*. Par exemple, le champ `l_type` indique le type du *lock* (lecture ou écriture), tandis que les champs `l_start` et `l_len` précisent le segment du fichier sollicité.

Bien que nous ne l'ayons pas implémenté au cours de cette thèse, l'API que nous proposons adhère à ce mécanisme de verrouillage par tranches d'objets à l'image du mode *Mandatory*.

Dans le même sens, `mmap` est une autre fonction `POSIX` qui admet les parties d'un fichier. En fait, elle réalise le *mapping*, c'est-à-dire la projection en mémoire d'un segment d'un fichier à partir d'un *offset*.

Nous ouvrons la parenthèse sur l'API que nous proposons plus loin dans ce document, pour souligner que ces deux derniers mécanismes sont inclus dans l'API **DHO**. En effet, le *mapping/locking* sont deux opérations inhérentes soutenues par les fonctions de notre API. Elles découlent d'une réponse positive à une demande d'un *lock*.

2.1.2 Message Passing Interface (MPI)

Vu la croissance des architectures à mémoire distribuée, l'API MPI a été conçue pour faire communiquer des processus distants par passage de messages. Conçue en 1993-1994 [mpia], MPI est une bibliothèque de fonctions normalisée reconnue par les langages `C, C++` et `Fortran`. MPI peut être exploitée aussi bien sur les machines massivement parallèles à mémoire partagée, qu'entre les *Clusters* d'ordinateurs hétérogènes. Nous nous référons à la version 2 pour résumer l'essentiel de ses propriétés.

Pour pouvoir communiquer, les processus doivent se déclarer de manière à former un ensemble appelé **communicateur**. Les communications sont de type point à point, et se font par les fonctions `MPI_Send`, `MPI_Recv` et `MPI_sendrecv`. La fonction `MPI_bcast` permet de diffuser un message à tous les processus d'un même communicateur, tandis que la fonction `MPI_scatter` applique une diffusion sélective.

MPI prévoit une synchronisation globale entre processus par la fonction `MPI_barrier` qui bloque les processus à l'endroit où cette fonction est insérée dans le code source, le temps qu'ils atteignent tous le même stade.

D'un autre côté, MPI prévoit des communications bloquantes et *non-bloquantes*. Pour les premières, le processus ne se libère qu'une fois l'action terminée. Par exemple, `MPI_Send` traduit une action bloquante dont la terminaison n'est effective que lorsque le récepteur la reçoit entièrement.

Au contraire, les actions *non-bloquantes* permettent d'optimiser le temps de communication. Dans ce mode, la main est rendue juste après que l'appel soit inséré. Ainsi, un recouvrement est produit car le transfert de la donnée peut se réaliser en même temps que les calculs de l'application. Les fonctions `MPI_isend` et `MPI_irecv` effectuent respectivement l'envoi et la réception *non-bloquantes*. Notons que dans ce cas précis, il revient au programmeur de s'assurer de l'envoi et de la réception effectifs du message par le biais des fonctions `MPI_test` et `MPI_wait` [CDG+] par exemple.

MPI propose différents modes de transfert de données. Dans l'approche *Remote Memory Access* (RMA), le processus émetteur a directement accès à la mémoire du récepteur. Le transfert se fait par copies de mémoire à mémoire. Dans cette approche, les processus définissent une zone mémoire (fenêtre) visible et accessible par la fonction `MPI_win_create()`. Ainsi, le transfert est entouré des fonctions `MPI_win_lock()` et `MPI_win_unlock()`. On remarque que dans ce type d'envoi, le processus cible est passif.

L'achèvement du transfert se fait par une étape de synchronisation qui met en action le processus émetteur.

En outre, les envois *bufférisés* sont déployés par les fonctions `MPI_bsend` et `MPI_ibsend`. Le *buffer* se trouve du côté de l'émetteur et est géré par l'application. Cette technique présente le désavantage d'une importante consommation de ressources, d'où le risque de saturation. En revanche, l'inter-blocage est clairement évité.

Les variantes de MPI sont multiples. Elles proposent des techniques d'optimisation de transfert et de communication. Par exemple, MPI-IO [TGL99], [CFF+95] prend en considération les aspects matériels et logiciels abritant les processus cibles. Elle autorise les entrées/sorties collectives par des processus d'un même communicateur à des seg-

ments de fichiers. Les accès se font par des masques (*filetypes*) utilisés pour indiquer le positionnement du fichier. Les entrées/sorties sont bloquantes ou *non-bloquantes*. Dans ce dernier cas, elles présentent tout l'intérêt d'un recouvrement avec les calculs.

MPICH est une variante de MPI conçue dans le but d'optimiser la latence entre les processus communicants en considérant l'hétérogénéité des ressources matérielles [mpib]. L'objectif de cette librairie est d'obtenir un débit élevé face à une faible latence. MPICH superpose des modules dans une architecture en couches dans laquelle les routines MPI opèrent au niveau supérieur [GLDS96]. Le module dénommé *Abstract Device Interface* (ADI) se greffe au plus bas de l'architecture pour fournir des informations sur l'environnement d'exécution, gérer les communications [GL96] et assurer la portabilité entre les différentes structures du réseau.

2.1.3 *Open Multi Processing* (OpenMP)

Standardisée depuis 1997, OpenMP est une librairie de programmation dédiée aux architectures parallèles à mémoire partagée [ope]. Portable sur UNIX comme sur Windows, OpenMP est supportée par les langages de programmation `C`, `C++` et `Fortran`.

L'API est basée sur un ensemble de clauses, directives, subroutines et de variables d'environnement. Aussi, OpenMP suit un modèle multi-tâches dont le mode de communication est implicite. Dans le code, le programmeur spécifie simplement les régions à paralléliser.

Les tâches parallèles sont exécutées par des *threads* créés implicitement, à l'entrée d'une région parallèle. Ici, le programmeur n'a pas besoin de faire appel à une fonction élaborée (*fork*) à l'instar de `Pthreads`. Le nombre de *threads* est réglé par défaut selon le nombre de cœurs existants dans la machine.

Les *threads* sont identifiés par des numéros qui sont retournés par la fonction `OMP_GET_THREAD_NUM()`. Cette région comprend des variables partagées en mémoire dont l'accès en lecture comme en écriture se fait à l'aide d'une pile (*Stack*) propre à chaque *thread*.

La directive `parallel` permet de définir les régions de code parallèles. Aussi, le nombre de tâches nécessaires dans une région peut être fixé par la clause `NUM_THREADS`.

À la terminaison de la région parallèle, les *threads* disparaissent tandis que le processus maître poursuit son exécution séquentielle. Par ailleurs, la synchronisation des *threads* est requise pour maintenir la cohérence des données.

Notons que les *locks* sont gérés par OpenMP pour être appliqués aux variables partagées dans le cas où l'application l'exige. À titre d'exemple, le listing 2.3 expose la demande d'un *lock* pour la variable partagée `verrou` initialisée en amont [CL].

Listing 2.3 – Exemple de locking d'une variable

```
Static omp_lock_t verrou;
omp_init_lock(&verrou);

#pragma omp parallel for
        for(i=0; i<n; i++)
        {
```

```
        omp_set_lock(&verrou);
        ..Section critique..
        omp_unset_lock(&lock)
        }
omp_destroy_lock(&verrou);
```

2.1.4 Étude comparative entre les différents paradigmes

Cette première étude nous a permis de mettre en évidence la diversité des stratégies mises en œuvre par ces interfaces, et d'en ressortir les points les plus forts et les moins avantageux de chacune d'elles.

Concernant les architectures à mémoire partagée, OpenMP et `Pthreads` sont comparables du point de vue gestion de la mémoire. Elles sont toutes les deux *Multi-threading* mais elles diffèrent dans leurs stratégies de parallélisation.

OpenMP est fondée sur la notion de tâche. Elle est destinée à la parallélisation d'instructions intrinsèquement parallèles qu'il est assez trivial d'implanter dans le code. Néanmoins, elle pose des limites de résolution de problèmes liées au partage de la mémoire.

L'utilisation de la librairie `Pthreads` est plus fastidieuse, mais elle se montre plus flexible dans les schémas d'exécution parallèles.

API	**Locking**			**Gestionnaire de cohérence**	***Chunk***
	Objet	**Concurrent**	**Mode**		
`Pthreads`	Mémoire	*thread*	Les deux modes	Noyau	Non
`fcntl` (Mandatory)	Fichier	Processus	Bloquant	Noyau	Oui
`fcntl` (Advisory)	Fichier	Processus	*Non-bloquant*	Utilisateur	Non
MPI	Buffer	Processus	Les deux modes	Utilisateur	Non
OpenMP	Zone mémoire	*thread*	Bloquant	Compilateur	Non

TABLE 2.1 – Étude comparative entre les paradigmes de partage de données

Les performances de MPI ont été démontrées aussi bien sur les machines massivement parallèles à mémoire partagée que sur les environnements hétérogènes à mémoire distribuée. De plus, MPI offre plusieurs alternatives de transfert de données. Cependant, certaines d'entre elles s'avèrent coûteuses en mémoire et en bande passante.

En somme, nous voulons combiner certains mécanismes de la librairie `POSIX-Pthreads` en matière d'insertion de *locks* en lecture/écriture (à la différence que les demandes seront servies en FIFO pour notre API) à celles par exemple, offertes par MPI en matière de contrôle sur la donnée, entre l'émetteur et le récepteur. De plus, nous nous inspirons de la fonctionnalité fournie par le *file-locking* pour permettre le partage des segments de données entre plusieurs processus (mode *Mandatory*).

Notre vision est celle d'une API où les processus gardent continuellement le contrôle sur la donnée par le biais d'objets appelés *handles* et l'instanciation de la donnée en mémoire locale.

2.1.5 SimGrid

Avant de clore cette section, nous présentons l'outil de base de nos expérimentations. Plus qu'un outil,`SimGrid` [CLQ08] est une boite à outils qui a émergé ces dernières années, dans le but d'offrir des solutions pour la conduite des expériences sur les environnements distribués tels que les grilles, les systèmes *peer-to-peer* et les *Clouds*.

`SimGrid` est utilisé par une large communauté scientifique, car elle procure un confort dans la programmation des applications distribuées et parallèles, sur des plates-formes réelles. Par ailleurs, `SimGrid` est un logiciel libre, distribué selon la licence GPL et portable sur plusieurs systèmes (LINUX, Mac OS X et Windows).

La première version résulte d'un travail de recherche effectué par Henri Casanova qui avait pour but d'étudier l'ordonnancement d'algorithmes pour les applications parallèles, au dessus des environnements distribués [CLQ08]. `SimGrid v1.0` a vu le jour avec une simple API appelée (SG). Depuis, plusieurs versions ont suivi pour donner naissance à d'autres APIs rangées au sommet d'une structure modulaire et d'un noyau propre à `SimGrid` (figure 2.1).

Les modules de `SimGrid` sont interchangeables. Toutefois, nous nous référons à la version que nous avons exploitée (3.6.2) dans cet aperçu des librairies disponibles :

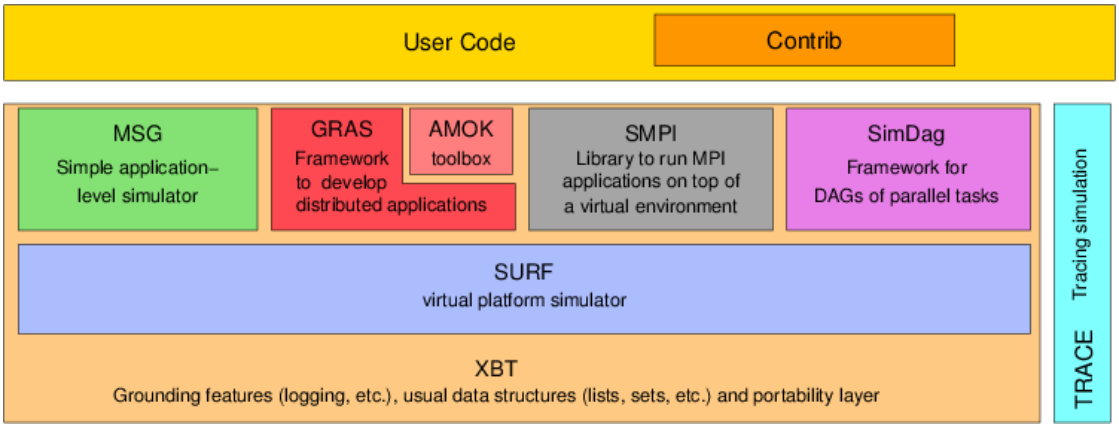


FIGURE 2.1 – SimGrid

MSG [CLQ08] est un simulateur de programmes distribués permettant l'étude d'heuristiques pour un problème particulier.

SMPI [CSG+11] reproduit les routines de MPI dans un environnement de simulation.

GRAS [Qui06a] (*Grid Reality and Simulation*) est l'interface que nous avons utilisée et intégrée à notre code applicatif. Basée sur les *Sockets*, elle permet de dépasser le

cap de la simulation pour exécuter les applications sur des plates-formes réelles. **AMOK** fournit des outils supplémentaires.

SimDag est le passage de SG à une nouvelle version. Cet environnement permet de simuler des tâches parallèles en se basant sur des graphes acycliques directs (DAGs).

Comme le montre la figure 2.1, le noyau de `SimGrid` est composé des modules suivants :

SURF est un module commun qui fournit tous les outils nécessaires à la création des plates-formes (en XML) pour le déploiement des applications.

XBT regroupe les outils de base de programmation (structures de données, types..).

Le modèle de communication de `SimGrid` sera décrit plus en détails dans les sections dédiées aux analyses expérimentales des chapitres 3 et 6.

2.2 L'exclusion mutuelle

Cette section fait ressortir les singularités du problème d'unicité et présente une vue d'ensemble sur les solutions apportées aussi diverses soient-elles, aux problèmes d'exclusion mutuelle. Nous commençons néanmoins par quelques définitions de base :

Section critique En programmation concurrente, une section est dite critique si son accès par des processus ou des *threads* concurrents entraîne un état incohérent. Généralement, elle résulte d'une ressource partagée et/ou modifiable. À un moment donné, un *thread* au plus est autorisé à activer dans cette section. Ainsi, une ressource non partageable est une ressource critique et toute section qui l'utilite est une **section critique**.

Le problème de contrôle dans la programmation concurrente a été abordé pour la première fois par Edsger Dijkstra [Dij65]. L'auteur a posé les requirements suivants pour assurer la séquentialité des opérations dans une **section critique** :

1. À un instant t un seul processus occupe la **section critique**.
2. Aucun ordre de priorité n'est assigné aux processus, de même que la vitesse relative aux processus n'est pas un facteur décisif pour l'entrée en **section critique**.
3. L'arrêt ou la suspension d'un processus quelconque du système ne doit en aucun cas bloquer les autres processus.
4. À la demande simultanée de plusieurs processus, la sélection finale doit se faire en un temps fini.

Par ces règles, l'atomicité des opérations sur une ressource dans une **section critique** est garantie. On parle alors d'exécution en exclusion mutuelle, *mutex* en anglais.

Le mot clé *lock* est souvent utilisé en programmation concurrente. Il reflète la notion de **section critique** associée à un bloc d'instructions, dans laquelle un objet particulier est concerné par le verrouillage.

Matériellement, une exclusion mutuelle est réalisable si les interruptions sont prohibées pendant qu'une variable partagée est modifiée au sein de la **section critique**.

Sémaphores Introduit par le même auteur [Dij65], un sémaphore est une variable entière et partagée S qui prend une valeur positive ou nulle. Le sémaphore est géré par deux opérations `Signal` et `Wait` qui agissent sur les processus pour un éventuel réveil ou une mise en attente. Entre autres, ces deux fonctions ont pour rôle de protéger la **section critique** et de synchroniser les processus.

En règle générale, les sémaphores sont implémentés dans le noyau du système d'exploitation, dont les valeurs sont conservées dans une table de la mémoire du noyau.

2.2.1 Propriétés de l'exclusion mutuelle

Depuis leur apparition, les mécanismes de synchronisation entre processus et d'exclusion mutuelle ont été adoptés tant dans les environnements parallèles à mémoire partagée que dans les environnements distribués. Différents par leur architecture, ces environnements ont pour point commun d'avoir des processus en compétition.

Tout algorithme qui prétend assurer l'exclusion mutuelle se doit de respecter les propriétés suivantes :

Sûreté (Safety) Rien de mauvais ne se produit. Deux processus ne peuvent en aucun cas partager la **section critique**. Cette propriété traduit l'atomicité des opérations au sein de la **section critique** et reprend donc la définition originelle de Dijsktra.

Vivacité (Liveness) Tout processus demandeur d'une **section critique** doit pouvoir y accéder au bout d'un temps fini. L'algorithme devra se préserver contre les famines et assurer l'équité entre processus. On dit qu'un algorithme est fort équitable s'il respecte l'ordre d'arrivée des processus dans les prises de décision. Pour y parvenir, une file d'attente est mise à disposition des processus pour les servir selon l'ordre FIFO.

Si l'équité faible assure l'absence de famine, elle admet néanmoins qu'un processus soit devancé par d'autres processus privilégiés, jouissant d'une priorité supérieure.

2.2.2 L'inter-blocage

L'interblocage (*deadlock* en anglais) est la pire des situations qui puisse arriver en programmation concurrente. Cet état reflète deux processus en attente mutuelle. Les processus bloqués dans cet état le sont définitivement. Dans la littérature, nous n'avons pas trouvé un dénouement à ce problème, car lorsqu'un processus détient une ressource en **section critique**, nul ne peut le forcer à la libérer.

Dans un système à N processus, l'inter-blocage peut se présenter sous forme d'un cycle dans un graphe d'attentes selon le schéma suivant :

Soit $\{p_0, p_1..p_n\}$ un ensemble de processus tel que :

p_0 attend p_1 attend...p_{n-1} attend p_n attend p_0

Ainsi, pour éviter l'inter-blocage dans un système à n processus compétitifs, le graphe d'attente doit être acyclique. Parmi les stratégies envisageables afin qu'un al-

gorithme soit exempt de ce problème, nous citons : La prévention, l'évitement et la détection.

Prévention La prévention impose souvent à priori des contraintes qui réduisent au maximum l'inter-blocage. Elles sont réalisables par des assertions insérées dans le code. Par exemple, on peut exiger la demande globale de toutes les ressources nécessaires, et la libération de celles-ci avant d'en demander de nouvelles. Cette solution consent parfois à la réservation des ressources non utilisées.

Evitement Le problème peut être évité par la connaissance préalable d'un schéma d'allocation de ressources. Le principe est de vérifier avant chaque étape d'exécution l'absence d'inter-blocage. Si tel est le cas, le processus est mis en attente. Le premier algorithme d'évitement a été introduit par Dijkstra [Dij68].

Détection Il s'agit de détecter à postériori les *deadlocks* pour y remédier à priori. Parmi les méthodes existantes dans la littérature, nous citons, les points de reprise et la réquisition des ressources détenues par un ou plusieurs processus. Pour cela, il convient de faire des choix pour désigner les processus victimes.

Par ailleurs, il existe des techniques de vérification automatique pour certaines propriétés dans les systèmes distribués. L'inter-blocage en fait partie. Ces techniques sont communément connues sous le nom de *Model-checking*. Ainsi, un *Model-checker* [CGP99] procède à l'exécution de tous les cas possibles et détecte ceux non conformes aux spécifications du système. Un retour au code source permet généralement de corriger les erreurs. Spin [Hol03] est le *Model-checker* le plus connu.

2.3 Taxonomie d'algorithmes distribués d'exclusion mutuelle pour les *locks* exclusifs

Une variété très large de publications a suivi celle de Dijsktra [Dij65] et a donné lieu à une multitude d'algorithmes d'exclusion mutuelle, pour la résolution des problèmes grandissants de partage de ressources dans les systèmes répartis.

La disparité des environnements distribués et les paradigmes les exploitant, ont été des facteurs prépondérants conduisant à l'évolution des algorithmes d'exclusion mutuelle. Ces algorithmes ont suivi d'emblée la progression des architectures, pour considérer celles basées sur l'envoi de messages autant que celles à mémoire partagée.

Les algorithmes d'exclusion mutuelle sont catalogués en deux grandes familles : Ceux basés sur les permissions, et ceux fondés sur la passation d'un jeton. Cette classification a été donnée pour la première fois par Michel Raynal [Ray91b]. Cela étant, elle existe de manière très éparse dans la littérature.

Nous nous intéressons dans cette thèse à la deuxième famille d'algorithmes, c'est-à-dire à ceux basés sur la passation d'un jeton, en raison des qualités offertes. Pour

commencer, nous passons en revue les algorithmes les plus marquants de leur catégorie.

2.3.1 Algorithmes à permission

Comme son nom l'indique, cette famille d'algorithmes repose sur la notion de permission. À la demande de la **section critique**, un processus doit demander, puis, attendre les permissions des autres processus. Donc, le droit d'entrée en section n'est acquis que si le processus demandeur a obtenu un nombre suffisant de permissions.

Soit p le processus demandeur. Dans cette approche, les processus n'ayant pas l'intention d'entrer en **section critique**, tout comme ceux qui ont déposé une demande postérieure, donnent leur accord à p. Autrement, c'est-à-dire les processus qui ont déposé une demande antérieurement à p, font attendre ce dernier jusqu'à ce qu'ils entrent puis libèrent la **section critique**.

2.3.1.1 Algorithme de Lamport

La littérature le présente parfois sous le nom d'algorithme de la boulangerie, car l'auteur s'est inspiré de la gestion d'une file d'attente d'un petit commerce, où des tickets d'ordre sont attribués aux clients [Lam78]. Dans cette approche, le mécanisme de synchronisation des messages est géré par le principe d'estampillage grâce à des horloges locales attribuées à chaque processus.

L'algorithme manipule trois types de messages munis de la valeur de l'horloge logique et de l'identifiant de son émetteur `(i,h)` :

1. `Request` pour l'envoi d'une requête.
2. `Release` diffusé à la libération de la **section critique**.
3. `Ack` envoyé au demandeur, à la réception du message `Request`.

Les processus partagent une zone de mémoire commune hébergeant une file d'attente mise à jour par les messages envoyés ainsi que par leur estampille. Le processus ayant la valeur d'horloge la plus ancienne entre en **section critique**.

Dans un système à n processus, p_i envoie $(n-1)$ messages pour la demande et la libération de la **section critique**. Il en reçoit autant pour pouvoir y accéder. L'algorithme vérifie bien les propriétés d'exclusion mutuelle. En revanche, il présente un inconvénient majeur en terme de complexité d'envoi de messages.

2.3.1.2 Algorithme Ricart-Argawala

Cet algorithme est une extension de celui de Lamport [Lam78]. L'amélioration porte sur la réduction du nombre de messages envoyés [RA81]. Lorsqu'un processus p_i transmet un message `Request` à un processus p_j, ce dernier ne lui envoie pas systématiquement un `Ack`. En fait, p_j répond par ce message s'il ne désire pas entrer en **section critique** ou s'il est lui même prioritaire. Dans ce dernier cas, le message `Ok` est envoyé

ultérieurement, c'est-à-dire à la libération de la **section critique**. Deux types de messages sont manipulés :

1. `Request` ici, la même horloge de Lamport est utilisée pour ordonner les requêtes.
2. `Ok` envoyé au processus demandeur, en guise de permission pour l'entrée en **section critique**.

Ainsi, p_i attend de recevoir $(n-1)$ messages de type `Ok` avant d'accéder à la **section critique**. Donc, en terme de complexité, le nombre total de messages envoyés est $2(n-1)$.

2.3.1.3 Algorithme Roucairol-Carvalho

Les algorithmes présentés jusque là sont basés sur des informations statiques. Autrement dit, la demande de la **section critique** ne tient pas compte de l'évolution de l'algorithme.

Une autre tentative d'amélioration a été proposée dans [CR84], dans laquelle un processus p_i limite l'envoi du message `Request` aux derniers processus demandeurs. Ainsi, si on exclut la première requête, la diffusion survient rarement. Quand p_i libère la **section critique**, il n'envoie un message qu'aux processus en attente de son accord.

Donc, la complexité est réduite pour varier entre 0 et $(2(n-1))$, ce qui constitue une amélioration de l'algorithme précédent. Cet algorithme figure parmi les algorithmes dynamiques d'exclusion mutuelle, car il tient compte d'une information variable.

2.3.2 Algorithmes basés sur la passassion d'un jeton

La finalité de l'exclusion mutuelle est d'assurer l'unicité d'accès à la **section critique**. Nous avons vu que dans un système à n processus, un seul est autorisé à y accéder. Les algorithmes à permission sont dépourvus de toute base structurelle. Guidés par une horloge commune, ils sont basés sur le paradigme de consensus qui requièrt l'autorisation de tous les autres processus ou d'une majeure partie d'entre eux. Le manque de scalabilité causé par l'absence de toute structure logique, est en partie responsable de la complexité élevée de ce type d'algorithmes.

Nous passons maintenant aux algorithmes fondés sur la notion de jeton. Ces algorithmes considèrent, en guise de remplacement des permissions, la possession d'un jeton comme seule condition pour l'entrée à la **section critique**. L'invariant suivant est posé :

Invariant 1. *Un système à n processus comprend un seul jeton dont l'accès à la **section critique** requièrt sa possession.*

Par cet invariant, l'unicité du jeton est assurée, de même que la propriété de *sûreté*. Partant de ce principe, la circulation du jeton est primordiale pour maintenir la *vivacité*. Les algorithmes ci-dessous s'accordent sur la notion de jeton et diffèrent par leur structure logique.

2.3.2.1 Algorithme de Suzuki et Kasami

Cet algorithme modélise l'ensemble des processus du système dans un grahe fortement connexe, et est basé simplement sur la diffusion. Lorsqu'un processus p_i désire entrer en **section critique**, il envoie un message à tous les autres. Celui qui détient le jeton le lui envoie directement. La structure logique est invariable et chaque requête requièrt N messages [SK85].

2.3.2.2 Algorithme de Singhal

Dans le souci permanent d'améliorer la complexité des messages, Singhal a proposé un algorithme qui maintient un sous ensemble de processus pour la diffusion des requêtes [Sin89]. L'algorithme de singhal repose sur la notion d'état. Le système garde en mémoire les états de chaque processus, ainsi que les séquences d'accès à la **section critique**. De ce fait, les états renseignent sur l'activité des processus ainsi que sur la possession du jeton.

Les processus adressent leurs requêtes uniquement à ceux qui réclament le jeton. L'idée est de restreindre l'envoi du message `Request` aux derniers processus demandeurs de la **section critique**.

Ainsi, le processus demandeur suppose que le jeton est forcément détenu par ceux qui l'on demandé. À la sortie de la **section critique**, le processus met à jour les informations le concernant, afin de maintenir un état cohérent du système.

Cet algorithme est classé parmi les algorithmes dynamiques non structurels avec une complexité qui varie entre 0 et n messages par requête.

2.3.2.3 Algorithme de Le Lann

Cet algorithme est fondé sur une structure logique en anneau unidirectionnel. Le jeton circule dans le sens des aiguilles d'une montre [Lan77]. Deux types de messages sont manipulés dans cet algorithme : `Request` et `token`.

Dans cette approche, les messages sont envoyés exclusivement entre voisins. Par conséquent, les processus ignorent la topologie globale du système.

La requête est diffusée de proche en proche dans la topologie circulaire. Lorsqu'un processus reçoit une requête, il la diffuse aussitôt à son voisin si le jeton n'est pas demandé. Autrement, la requête est bloquée momentanément, le temps de la libération de la **section critique**.

L'on remarque l'absence d'une mémoire globale dans cet algorithme. Aussi, le nombre de messages envoyés varie entre 1 et $(n-1)$.

2.3.2.4 Algorithme de Kerry Raymond

L'algorithme structurel de Kerry Raymond [Ray89b] organise les processus dans un arbre fixe, ayant pour racine le détenteur du jeton. Les requêtes sont propagées à travers

les nœuds de l'arbre pour se diriger vers la racine. En parcourant l'arbre, les requêtes sont enregistrées dans la queue de chaque processus.

Ainsi, lorsqu'un processus désire entrer en **section critique**, il insère sa demande dans la queue avant de l'acheminer.

Lorsque le message parvient au processus qui détient le jeton, si ce dernier a libéré la **section critique**, il envoie le jeton par le même chemin emprunté par la requête, dans le sens inverse.

En termes de complexité, le nombre moyen de messages nécessaires pour l'accès à la **section critique** est de l'ordre de $log(n)$.

2.3.2.5 Algorithme Naimi-Tréhel

L'algorithme de Naimi et Tréhel [NTA96b] utilise un arbre logique dynamique ainsi qu'une file d'attente de requêtes suspendues. Initialement, tous les processus pointent sur la racine de l'arbre qui détient le jeton. Cette variable est connue sous le nom de ***Parent***. Dans cette approche, le dernier processus demandeur de la **section critique** devient racine.

Les requêtes sont acheminées séquentiellement le long de l'arbre. Les processus intermédiaires mettent à jour leur variable ***Parent*** et pointent sur le processus demandeur. L'algorithme maintient une file *Suivant* des processus demandeurs. À chaque libération de la **section critique**, le processus se détache de cette file et passe le jeton à son successeur dans la file *Suivant*. La complexité des messages envoyés est de l'ordre de $log(n)$.

Cet algorithme couvre une bonne partie de cette thèse. Nous y reviendrons en détail dans le chapitre 4

2.4 Algorithmes distribués d'exclusion mutuelle de groupes

Nous avons donné un aperçu des algorithmes de référence traitant le problème d'exclusion mutuelle. Ils diffèrent par leur stratégie structurelle et leur complexité. En revanche, ils s'accordent tous sur l'exclusivité de la **section critique**.

Cette section parcourt les algorithmes qui admettent plusieurs processus dans la **section critique**. À l'instar des algorithmes classiques décrits précédemment, la littérature retrace une large variété d'algorithmes d'exclusion mutuelle de groupes, incluant ceux basés sur les permissions comme ceux fondés sur la passation d'un jeton.

2.4.1 Problème noté *l-exclusion mutuelle*

Dans un système à n processus, le problème noté *l-mutual exclusion* a été initié dans [FLBB79] afin d'élargir l'accès à une ressource donnée par l processus de manière concurrente. De toute évidence : $(1 \leq l < n)$.

Le terme *l-exclusion mutuelle* s'est alors répandu pour former une classe d'algorithmes traitant l'accès multiple à la **section critique**.

Algorithme	**Structure**	**Section critique**	**Diffusion**	**Complexité**
Lamport [Lam78]	-	Permission	Oui	$3(N-1)$
Ricart-Argawala [RA81]	-	Permission	Oui	$2(N-1)$
Roucairol-Carvalho [CR84]	-	Permission	Oui	$[0, 2(N-1)]$
Suzuki-Kasami [SK85]	Graphe statique	Jeton	Oui	N
Singhal [Sin89]	-	Jeton	Oui	$[0, N]$
Le Lann [Lan77]	Anneau statique	Jeton	Non	$[1, N-1]$
Raymond [Ray89b]	Arbre statique	Jeton	Non	$log(N)$
Naimi-Tréhel [NTA96b]	Arbre dynamique	Jeton	Non	$log(N)$

TABLE 2.2 – Taxonomie des algorithmes d'exclusion mutuelle

Dans [Ray89a], Kerry Raymond a étendu l'algorithme Ricart-Argawala (section 2.3.1.2) pour autoriser l entrées. Un processus qui désire entrer en **section critique** demande simplement $n-1$ permissions aux autres processus. D'autres solutions ont été proposées depuis, dans cette catégorie.

Michel Raynal a généralisé le problème de *l-exclusion mutuelle* à k parmi *l-exclusion mutuelle*, dans un système à l copies d'une ressource [Ray91a]. L'accès est permis par k processus sachant que $1 \leqslant k \leqslant l$.

Ainsi, les algorithmes traitant ce type de problème ont pour but de permettre à un nombre borné de processus, d'accéder concurremment à une ressource donnée.

Dans [Jou98], Yuh-Jzer Joung a énoncé le problème pour un système à m ressources. Il fait une analogie avec des philosophes qui pensent et parlent dans un forum. Sachant qu'une seule salle est mise à leur disposition, un philosophe ne peut y accéder que si la salle est vide ou si elle est occupée par d'autre philosophes qui discutent du même sujet. De ce fait, la notion de plusieurs sections a été introduite. Le même auteur a posé le problème d'allocation de ressources $(m, 1, k)$ pour modéliser l'exclusion mutuelle de groupes à capacité bornée, dans laquelle une ressource donnée peut être utilisée au plus par simultanément k processus du même groupe [Jou04].

Aux deux propriétés de *vivacité* et de *sûreté* requises pour tout algorithme d'exclusion mutuelle classique, s'ajoutent celles qui caractérisent les accès multiples. Ainsi, le problème d'exclusion mutuelle de groupe noté communément *(GME)* se distingue par les propriétés suivantes :

Exclusion mutuelle Si deux processus p et q exécutent simultanément la **section critique**, alors $S_p = S_q$. Par cette propriété, le système garantit qu'à un instant donné, une seule ressource est partagée à la fois.

Accès concurrent Si quelques processus demandent la même **section critique** alors qu'aucun processus ne demande une section différente, tous les processus demandeurs peuvent l'exécuter concurremment.

Les techniques basées sur les quorums et les coteries appartiennent à la catégorie des algorithmes à permissions traitant du problème d'exclusion de groupes.

2.4.2 Algorithmes basés sur les coteries

Afin de garantir qu'au plus k processus entrent concurremment en **section critique**, le concept de *k*-coteries a été défini comme suit [KFYA94] :

Soit $U = \{p_1, ..p_n\}$ un ensemble de processus. Un ensemble non vide C d'un ensemble non vide Q de U est appelé *k-coterie* si et seulement si les propriétés suivantes sont vérifiées :

Non-intersection Pour tout $l(1 \leqslant l \leqslant k-1)$ et pour tout l éléments $\{Q_1, ...Q_l \in C\}$ tel que $Q_i \cap Q_j = \oint (i \neq j)$, il existe un élément $Q \in C$ tel que $Q \cap Q_i = \oint$ pour $1 \leqslant i \leqslant l$. Cette propriété assure que même si l ($\leq k-1$) processus reçoivent la permission de l quorums arbitraires présents en **section critique**, un processus peut trouver un quorum Q qui n'intersecte pas avec chacun des l quorums.

Intersection Il existe au plus $k+1$ éléments $\{Q_1, ...Q_{k+1} \in C\}$ tel que $Q_i \bigcap Q_j = \oint$ pour tout $1 \leqslant i, j \leqslant k+1$. Par cette propriété, la présence d'au plus k éléments en **section critique** est vérifiée.

Minimalité Pour toute paire Q_i et Q_j dans C, $Q_i \nsubseteq Q_j$.

L'algorithme proposé dans [KFYA94] utilise l'estampille introduite par Lamport [Lam78] afin d'éviter l'inter-blocage et la famine. Les requêtes sont identifiées par leur processus correpondant et la valeur de l'estampille.

Dans un problème classique *1-mutual exclusion*, les permissions de tous les membres du quorum Q auquels appartient un processus p_i sont nécessaires.

Dans le cas de *k-exclusion mutuelle*, cette condition n'est plus de mise car le processus demandeur peut obtenir la permission d'un autre quorum en raison de l'existence de $k-1$ quorums qui n'intersectent pas avec Q.

En terme de complexité, $3 \mid Q \mid$ messages sont nécessaires dans le meilleur des cas. Pour preuve, l'algorithme manipule trois types de messages `request`, `release` et `Ok`. Ces derniers sont dirigés par un processus donné vers tous les processus du quorum sélectionné Q. Au pire des situations, le nombre de messages nécessaires peut atteindre $6N$ où N est le nombre total de processus.

Les auteurs proposent de borner le nombre de tentatives d'envoi de requêtes par l'utilisation d'une fonction appropriée.

Dans la même direction, le concept de *k-coteries non dominées* a été présenté dans [NM94], formellement définies comme suit :

Soit Q_1 et Q_2 des *k-coteries* d'un ensemble de processus U. Q_1 domine Q_2 si :

1. $Q_1 \neq Q_2$
2. $(\forall H \in Q_2)\exists G \in Q_1$ tel que $G \supseteq H$

Donc, une *k-coterie* Q_1 est *dominée* s'il existe une deuxième coterie Q_2 qui la domine. Deux méthodes sont proposées pour construire les *k-coteries non-dominées* : L'une basée sur le vote pondéré et l'autre sur la composition. Outre la disponibilité, les auteurs ont prouvé que les *k-coteries* non dominées sont plus résistantes aux pannes.

2.4.3 Algorithmes structurels

Les solutions présentées ci-dessus répondent à la problèmatique d'exclusion mutuelle à entrées multiples. Elles ont pour point commun deux propriétés.

Premièrement, elles requièrent toutes un certain nombre de permissions. Deuxièmement, elles ne font aucune hypothèse sur la topologie des nœuds. Les méthodes auxquelles les auteurs ont eu recours avaient pour but de réduire ce nombre afin de minimiser le coût produit par une requête.

En dépit des méthodes déployées (quorum et coteries), le constat est le même que pour les algorithmes d'exclusion mutuelle classique : La complexité en terme de messages envoyés pour les algorithmes à permissions demeure élevée.

Nous focalisons dans cette section sur les algorithmes où les processus sont organisés dans une structure logique.

Dans [WWV01], les auteurs proposent un algorithme d'exclusion mutuelle de groupes appelé *Reverse-link*, basé sur la circulation de jetons et adapté aux réseaux mobiles *ad hoc*. L'algorithme résout le problème de partage de m ressources par n processus concurrents. Il est donc classé k parmi *l-exclusion mutuelle*. L'algorithme maintient un graphe acyclique direct (DAG) ainsi que k jetons pour une ressource donnée. Nous le résumons comme suit :

Lorsqu'un processus tente d'accéder à une ressource partagée, il envoie une requête à son voisin dans la topologie. La requête s'achemine le long du lien de communication. Chaque processus maintient une file d'attente contenant les identificateurs des processus voisins à partir desquels il a reçu les requêtes.

Les processus sont ordonnés de sorte que le plus petit nombre soit attribué à celui qui détient le jeton.

Les requêtes sont dirigées vers le jeton. De ce fait, les processus choisissent une route. En cas de panne, les requêtes changent de chemin.

À chaque état d'exécution, le graphe acyclique est orienté selon la localisation du jeton.

L'algorithme présente l'avantage d'être tolérant aux pannes. Le papier reporte les résultats de son implémentation, cependant, la complexité n'a pas été clairement étudiée.

Ousmane Thiare présente une adaptation à cet algorithme en y ajoutant un message pour informer les processus voisins qu'ils peuvent accéder à la **section critique** de manière concurrente [TN06].

Dans [CDPV01], les auteurs présentent un algorithme basé sur la passation d'un jeton dans lequel les processus sont rangés dans un anneau unidirectionnel. Le mouvement du jeton suit un graphe arbitraire. Les processus ne sont pas identifiés, ne manipulent ni une structure de données ni une file d'attente.

L'algorithme manipule trois types de messages : `request_session`, `grant_session` et `release_session`. Le premier processus qui entre en **section critique** fait office de *leader*. Un processus ne peut envoyer un message qu'à son premeir voisin. De même, il ne peut en recevoir que de son deuxième voisin.

Lorsque le jeton est intercepté par le premier processus demandeur (supposé p) d'une section X, il devient *leader* et diffuse le jeton X à partir de son voisin, pour informer que la session X est ouverte. Tant qu'aucune autre session différente de X n'est réclamée, le jeton circule dans l'anneau et tout processus désirant la **section critique** pour la même ressource obtient satisfaction.

Les auteurs déclarent une complexité d'envoi de messages de l'ordre de N^s2 messages.

2.5 Algorithmes d'exclusion mutuelle pour les *locks* exclusifs et partagés

Dans cette thèse, notre intérêt porte sur le partage des ressources dans les systèmes distrtibués, dans le sens où les accès peuvent être aussi bien multiples qu'exclusifs. Les algorithmes présentés précédemment traitent la question de groupe mais ne proposent pas l'alternative de l'exclusivité d'accès.

Le problème d'exclusion mutuelle a été généralisé pour traiter les accès en lecture et/ou en écriture à une ressource donnée. Il fut énoncé par Dijkstra [Dij68] afin de soulever la problématique de la gestion des bases de données. Un rédacteur est généralement un processus léger (`thread`) qui détient l'exclusivité sur la donnée en vue d'y apporter une modification. Les lecteurs peuvent accéder simultanément à une donnée. L'exclusion mutuelle est donc assurée, d'une part, entre les processus rédacteurs et d'autre part, entre les lecteurs et rédacteurs.

Dijsktra a posé le problème de famine que pourrait causer un flux continuel de lecteurs au détriment d'un rédacteur. Il a apporté une solution qui instaure un système de priorité au moyen des sémaphores [Dij68].

La diversité des architectures parallèles à mémoire partagée a suscité un intérêt particulier des chercheurs qui ont inondé la littérature par des algorithmes efficaces de

synchronisation entre processus, en réponse aux besoins continuels des accès cohérents à la mémoire.

Après avoir parcouru quelques travaux, nous avons retenu quelques termes récurrents que nous définissons ci-dessous [hYA94] :

Attente active On dit qu'un processus est en attente active (*Busy-waiting*) s'il tente de vérifier de manière répétitive la véracité d'une condition.

Spin-local C'est un *lock* qu'un processus en attente active tente d'acquérir en boucle. Le *Spin-local* présente un intérêt dans les multi-processeurs pour des tâches atomiques au sein d'une mémoire commune.

Processus de contention Dans les multi-processeurs, les processus de ce type partagent un lien physique et s'accordent pour désigner celui qui entre en **section critique**, selon une politique appropriée. À la libération de cette dernière, le processus de contention reprend son cours.

Point de contention C'est le nombre maximum de processus en contention à un instant donné.

Remote memory references En abréviation **RMRs**, ce terme fait référence aux accès en mémoire par un processus pour une opération en lecture ou en écriture. Les chercheurs prennent en considération le **RMR** pour mesurer la performance des algorithmes.

Yan et Anderson ont été les précurseurs pour avoir utilisé le mécanisme du `Spin-local`, appliqué aux instructions atomiques de lecture/écriture, avec une complexité de l'ordre de $Olog(n)$ [hYA94]. Ici, la compléxité est mesurée en comptant les **RMRs**.

Les auteurs ont analysé la contention dans l'exclusion mutuelle à lecteurs/rédacteurs pour conclure que le temps moyen de réponse d'un algorithme, est étroitement lié à ce paramètre. Ainsi, des techniques basées soit sur une queue soit sur un arbre arbitraire, sont apparues pour réduire la contention en exclusion mutuelle.

Yan et Anderson ont intégré dans leur algorithme un arbre binaire, dans lequel les instances de deux processus sont intégrées arbitrairement. Dans leur approche, l'exclusion mutuelle est appliquée par paire de processus à chaque niveau de l'arbre.

Quelques algorithmes ont suivi pour réduire la contention des lecteurs/rédacteurs dans les systèmes à mémoire partagée.

Dans [AjK01], les auteurs font la liaison entre la performance des algorithmes abritant le *Spin-local* et la quantité du trafic d'interconnexion générée. Par conséquent, ils considèrent la valeur la plus élevée du **RMR** produite par le *Spin-local* d'un processus, afin de mesurer la complexité de ces algorithmes.

Ainsi, un algorithme d'exclusion mutuelle est adaptatif si sa complexité est fonction du nombre de processus en contention.

Dans [CS94], les auteurs affirment que le nombre de processus de contention doit être raisonable afin de maintenir un temps de réponse acceptable.

D'autres modèles formels sont apparus dans la littérature. La file d'attente *Queue-Read Queue-Write* (QRQW) a été introduite dans le modèle PRAM pour la concurrence d'accès en lecture/écriture à des espaces à mémoires partagées [GMR94]. Les auteurs estiment que le coût est proportionnel au nombre des lecteurs/rédacteurs présents dans n'importe quel espace mémoire à une étape donnée. Dans ce modèle, le temps d'exécution d'une opération en lecture ou en écriture sur une variable est une fonction linéaire du nombre de processus accédant à la même variable concurremment.

Le modèle QRQW-PRAM reflète les propriétés de contention de la majorité des machines parallèles de l'époque.

2.6 Variantes de l'algorithme Naimi-Tréhel

Parmi les classes d'algorithmes d'exclusion mutuelle, nous optons pour ceux basés sur la passation d'un jeton car, souvent, ces algorithmes sont dotés d'une structure logique. En effet, bien que des techniques aient été appliquées pour tenter de réduire leur complexité, la diffusion des messages dans les algorithmes à permission entrave leur scalabilité dans le contexte des environnements à grande échelle.

L'algorithme Naimi-Tréhel [NTA96a] présente l'avantage de préserver une complexité logarithmique en plus de son pouvoir d'adaptation aux mouvements des requêtes, de manière à privilégier les demandes les plus fréquentes. Par ailleurs, la structure arborescente sur laquelle il repose, favorise son extension. En effet, la jonction de nouveaux éléments se résume à la sélection d'une feuille de l'arbre. Cette question est étudiée plus loin dans ce document (chapitre 4).

L'algorithme originel a suscité un grand intérêt des chercheurs. C'est pourquoi, plusieurs extensions ont été proposées afin de répondre adéquatement aux exigences actuelles des systèmes distribués.

Julien Sopena a proposé une version tolérante aux pannes, en préservant la même complexité logarithmique [SABS05]. Afin de préserver l'ordre logique des requêtes émises même en cas de panne, l'auteur a ajouté un mécanisme d'acquittement de requêtes et a introduit des temporisateurs de détection et de recouvrement de pannes. L'auteur s'est appuyé sur une plate-forme en environnement réel pour l'évaluation des performances.

Une deuxième approche a été proposée par le même auteur pour composer de façon hiérarchique et générique des algorithmes d'exclusion mutuelle, grâce à des nœuds coordinateurs implantés dans chaque *Cluster*, dans le but de distinguer les algorithmes *intra-cluster* des algorithmes *inter-cluser* [SLAAS07].

Dans la même direction, l'extension de l'algorithme qui a fait l'objet de l'article [BAS06] considère l'aspect hétérogène induit par les latences *inter-clusters* dans les grilles. Cet algorithme favorise les requêtes *intra-Clusters* jusqu'à un certain nombre d'accès successifs. Pour cela, les auteurs ont réorganisé les nœuds en deux niveaux hiérarchiques au moyen d'un *Proxy*.

Dans la version proposée dans [QV09a] les requêtes adressent des demandes pour des parties de *locks* asynchrones. Dans cet algorithme, la notion de ressource est clairement associée au *lock*. Ainsi, l'algorithme *Split Waiting Queues* (SWQ) fractionne la file d'attente *Suivant* lorsque des parties de la ressource sont réclamées. L'impact des paramètres suivants a été étudié expérimentalement : Durée du *lock*, taille de la ressource, degré de fractionnement et nombre de nœuds.

A notre connaissance, la version présentée dans [WM00] est la seule à introduire les *locks* en lecture et en écriture. Dans cet algorithme, deux types de requêtes sont insérées dans la file d'attente. Ainsi, la **section critique** est accédée simultanément par des processus à demandes successives en lecture. La passation du jeton est basée sur des acquittements envoyés entre processus successifs dans la file *Suivant*. Expérimentalement, les auteurs ont étudié l'influence du nombre de nœuds, ainsi que de la proportion des requêtes en lecture sur la durée du *lock*. Notons que les deux dernières versions manquent d'une étude théorique.

2.7 Synthèse du chapitre

Les points abordés dans ce chapitre sont un bon point de départ pour la suite de notre étude. Compte tenu des APIs présentées, nous déduisons qu'elles sont intrinsèquement liées au schéma matériel pour lequel elles ont été conçues. OpenMp et `POSIX-Pthreads` ont été pensées pour parfaire des opérations massivement parallèles dans les multi-processeurs à mémoire partagée.

Cependant, il est difficile de maintenir le contrôle sur les *threads* comme il est également difficile de garantir la cohérence de données dans une plate-forme distribuée à grande échelle avec OpenMP.

D'un autre côté, certains modes de transferts offerts par MPI sont inadéquats dans un schéma à mémoire partagée. Le mécanisme de *bufférisation* pose des limites dans le partage des ressources de grande taille. En outre, la collection préalable des processus par l'utilisation d'objets (*communicators*) s'oppose aux fondements des grilles informatiques et des systèmes *peer-to-peer*.

Nous reprenons les principes de base de *locking* en lecture/écriture offerts par les fonctions de `POSIX-threads`. Nous nous inspirons également du service fourni par `mmap` pour la projection de la donnée en mémoire.

En revanche, nous optons pour une API qui évolue dans un système la localisation et le nombre de processus sont méconnus de l'utilisateur. En d'autres termes, nous voulons assurer la transparence d'accès aux données.

L'étude des algorithmes distribués d'exclusion mutuelle nous a permis d'avoir une idée plus concise sur les différentes stratégies mises sur place. Les algorithmes non structurels sont souvent basés sur les permissions et la diffusion des requêtes. Leur performance est incontestée dans un système borné. Leur évolution est toutefois compromise dans un contexte étendu et modulable pour le coût qu'ils génèrent.

Les algorithmes strcuturels nous ont intéressés car ils adhèrent au principe de voisinage. En effet, cette propriété facilite la connexion comme la déconnexion des nœuds

du système, sans subir de lourdes conséquences. Il suffit de s'annoncer à deux nœuds connectés pour joindre un anneau par exemple (algorithme de Le lann), ou simplement une feuille dans une structure en arbre (algorithme Naimi-Tréhel).

Les algorithmes à jeton présentent l'avantage de n'attendre qu'un message (jeton) pour accéder à la **section critique**. Nous avons opté pour l'algorithme de Naimi et Tréhel pour les deux propriétés citées ci-dessus mais pas seulement. Le caractère dynamique de la structure arborescente de cet algorithme contribue à minimiser davantage le nombre des messages envoyés car, les nœuds qui composent l'arbre s'adaptent à l'envoi des requêtes et se rapprochent des nœuds à grandes fréquences de demandes.

Les chapitres 4 et 5 explicient nos contributions sur l'algorithme Naimi-Tréhel, tandis que les chapitres 3 et 6 décrivent notre API tout en analysant son comportement face à deux architectures différentes.

Chapitre 3

L'interface *Data Handover*

Sommaire

3.1 Définitions générales 40
3.2 L'interface *Data Handover* 41
3.2.1 Les principes de base de **DHO** 41
3.2.2 Description détaillée de **DHO** 41
3.2.3 Cycle de vie de l'interface **DHO** 42
3.3 Critères d'évaluation et paramètres d'exécution 48
3.3.1 Durées observées 48
3.3.2 Plate-forme matérielle d'expérimentations 49
3.3.3 Modèle et librairie de communication 49
3.4 Analyse des résultats expérimentaux 50
3.4.1 Plan des expériences 50
3.4.2 Étude des délais de création et de destruction du *handle* 51
3.4.3 Étude du *mapping* et de la mise à jour de la ressource 52
3.4.4 Analyse de la performance du cycle **DHO** pour les *locks* exclusifs en l'absence de retard au blocage 52
3.4.5 Déduction de la durée du cycle pour les *locks* partagés 55
3.4.6 Évaluation du cycle pour les *locks* asynchrones 56
3.4.7 Évaluation du cycle pour les *locks* exclusifs et partagés 56
3.5 Synthèse du chapitre 58

Nous avons passé en revue les standards de communication pour l'exécution des programmes parallèles et distribués en présentant les techniques utilisées pour l'accomplissement des opérations de *locking* d'objets en lecture/écriture (section 2.1). Il en ressort que la performance de chaque interface dépend de l'architecture matérielle sous-jacente (section 2.7).

Les paradigmes basés sur l'envoi de messages (MPI) nécessitent une recopie de la donnée dans un *buffer*, alors que les paradigmes à mémoire partagée doivent gérer la concurrence d'accès entre plusieurs processus.

Nous but est de concevoir une API qui tire profit des fonctionnalités principales offertes par les standards existants, tout en tenant compte des aspects liés aux systèmes hétérogènes à haut débit.

Ce chapitre présente l'interface **DHO**. Elle englobe les caractéristiques des deux architectures [Gus06]. Ainsi, **DHO** est un paradigme de programmation qui combine donnée et mémoire, et qui s'intègre facilement dans le code d'une application. Par cette API, nous visons à dissimuler l'architecture interne du système distribué au niveau applicatif.

Après une description complète et détaillée de la structure de base de l'API (section 3.2), nous présentons une modélisation appropriée qui fait ressortir des états pendant l'accomplissement d'un cycle **DHO** (section 3.2.3.1). La section 3.3 expose la méthodologie adoptée pour une première évaluation de l'API.

Nous étudions dans ce chapitre, l'approche client-serveur, basée sur une gestion centralisée de la ressource. La section 3.4 expose l'interprétation des résultats obtenus suite à des tests approfondis menés en simulation puis en environnement réel.

3.1 Définitions générales

Pour commencer, il nous paraît utile de définir quelques termes liés à **DHO**.

Ressource C'est une donnée hébergée sur une machine quelconque dans un système distribué. Dans ce chapitre, nous considérons que la ressource est hébergée dans un seul serveur.

Mémoire Dans notre API, la ressource et la mémoire sont étroitement liés. À l'aide d'une fonction **DHO**, le contrôle sur la ressource partagée est maintenu par le processus qui la demande, dès lors que cette ressource est instanciée en mémoire locale.

Requête Une requête est créée suite à une demande lancée par un appel d'une fonction **DHO** qui cible une donnée particulière. Le programmeur n'est pas tenu de connaître la localisation de cette dernière.

État Un état reflète une situation particulière d'une entité vis-à-vis d'une requête ou d'une ressource. Les états facilitent la compréhension du modèle. Dans ce chapitre, deux entités sont concernées par la notion d'état.

Atomicité Cette propriété traduit l'accomplissement d'un cycle **DHO** sans interruption, depuis l'insertion de la requête jusqu'à la libération de la ressource instanciée en mémoire locale.

3.2 L'interface *Data Handover*

Pour commencer, nous énoncons les propriétés qui doivent satisfaire notre API.

3.2.1 Les principes de base de DHO

Simplicité **DHO** doit assurer une simplicité d'usage, de façon à ne pas contraindre le programmeur d'apporter des modifications majeures dans son application. Elle doit pouvoir s'intégrer facilement à d'autres librairies existantes afin de garantir sa portabilité.

Performance Notre API doit assurer au moins la même performance que les autres interfaces. Son évaluation doit couvrir un spectre très large de *Benchmarks*.

Intéropérabilité **DHO** doit se rallier autant que possible, aux autres APIs normalisées, afin d'assurer sa portabilité sur différentes architectures matérielles et logicielles.

Abstraction des données Le programmeur ne doit pas se soucier de la structure interne de l'API. Son rapport avec le système se limite à un objet intermédiaire, qui fait référence à la donnée.

3.2.2 Description détaillée de DHO

DHO introduit un niveau d'abstraction entre la ressource et la mémoire à travers des objets appelés *lock handles* , conformément à une politique de contrôle d'accès. Les *clients* **DHO** se disputent l'accès à la ressource dont la concrétisation est réalisée par le verrouillage (*locking*) et le chargement en mémoire locale (*mapping*), au moyen de processus applicatifs appelés *gestionnaires de ressources*. Nous rappelons que ces opérations sont soutenues par l'API `Pthreads` et les fonctions `fcntl` et `mmap` dans `POSIX`.

Dans un premier temps, nous supposons que la ressource est hébergée dans un seul serveur qui aura à gérer toutes les demandes. La gestion distribuée est étudiée plus loin dans ce document (chapitre 6). Le propriétaire de la ressource (*serveur*) traite les demandes tout en assurant la cohérence d'accès grâce à un *gestionnaire de verrou* . Pour un *client* donné, plusieurs *handles* peuvent être rattachés à la même ressource, si une séquence d'accès lui est adressée. Par un schéma représentatif (figure 3.1), nous illustrons les intéractions possibles entre le *gestionnaire de ressource* (*client*) et le *gestionnaire de verrou* (serveur).

Plusieurs évènements surviennent depuis l'insertion de la requête jusqu'à la résiliation de la ressource, conduisant le *client* et le *handle* à acquérir plusieurs états. Ces derniers attestent la position du *client* au regard de la ressource. Par exemple, *Idle* dénote l'état initial d'un *client* donné (figure 3.2).

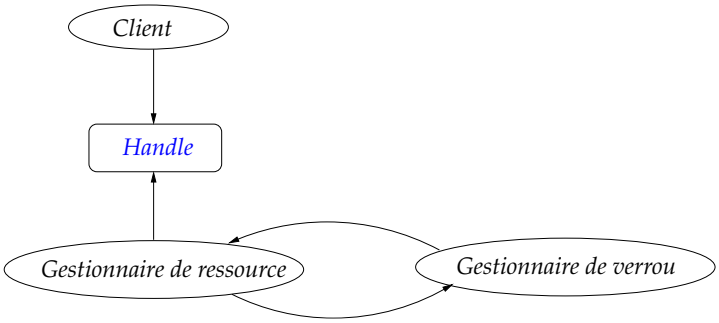


FIGURE 3.1 – Intéractions entre le *client*, le *gestionnaire de ressource* et le *gestionnaire de verrou* (Serveur)

À un degré plus bas, des états abstraits sont attribués au *handle*. Ils sont engendrés par, d'une part, les initiatives du *client*, et d'autre part par les interactions entre le *gestionnaire de ressource* et le *gestionnaire de verrou*. Par exemple, *invalid* représente l'état initial du *handle* (figure 3.3).

Les durées de séjour du *client* et du *handle* dans chaque état sont des mesures non moins insignifiantes. Nous les étudions dans la partie expérimentale de ce chapitre (section 3.3.1). Certaines de ces durées sont applicatives (déterminées dans le code par le *client*) alors que d'autres sont observées. Par exemple, le *client* devra déterminer la durée du *lock* ou alors le temps qu'il doit prendre avant de passer au mode bloquant.

Au terme des négociations entre le *gestionnaire de ressource* et le *gestionnaire de verrou*, la ressource est instanciée en mémoire locale du *client*, par une fonction qui retourne un pointeur sur l'adresse mémoire de la ressource.

Cependant, le *client* tente au début de créer une *Socket* avec le serveur qui héberge la ressource. Ainsi, par l'appel de la fonction `dho_create`, le nom de la ressource est pris comme argument. En retour, le serveur envoie la taille de la ressource. La *Socket* est donc créée et le *handle* devient *valid* .

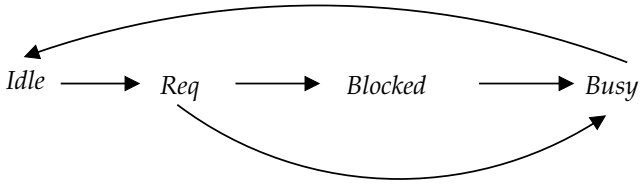


FIGURE 3.2 – Cycle d'un *client* **DHO**

3.2.3 Cycle de vie de l'interface DHO

L'acquisition de la ressource en mémoire locale (*locking/mapping*) est un processus complexe qui fait intervenir toutes les entités citées plus haut (section 3.2). Le *client* est le premier acteur qui déclenche une action, suite à laquelle d'autres événements se

succèdent. Toutes les fonction de l'API prennent le *handle* comme argument .

Le cycle de vie **DHO** se traduit par l'ensemble d'étapes et états que le *client* et le *handle* traversent, avant l'aboutissement de la requête (figure 3.3).

DHO reconnaît deux modes d'accès, celui en écriture (ew) pour un seul écrivain en accès exclusif, et celui en lecture (cr), dédié à plusieurs lecteurs simultanés en accès partagé. Au niveau du serveur, le *gestionnaire de verrou* concède l'accès à la ressource selon l'ordre d'arrivée et d'enregistrement des requêtes, conformément à une stratégie FIFO (contrairement à `POSIX-Pthreads` qui privilégie quatre rédacteurs au détriment d'un lecteur). Le *handle* encapsule toutes les informations relatives à la requête.

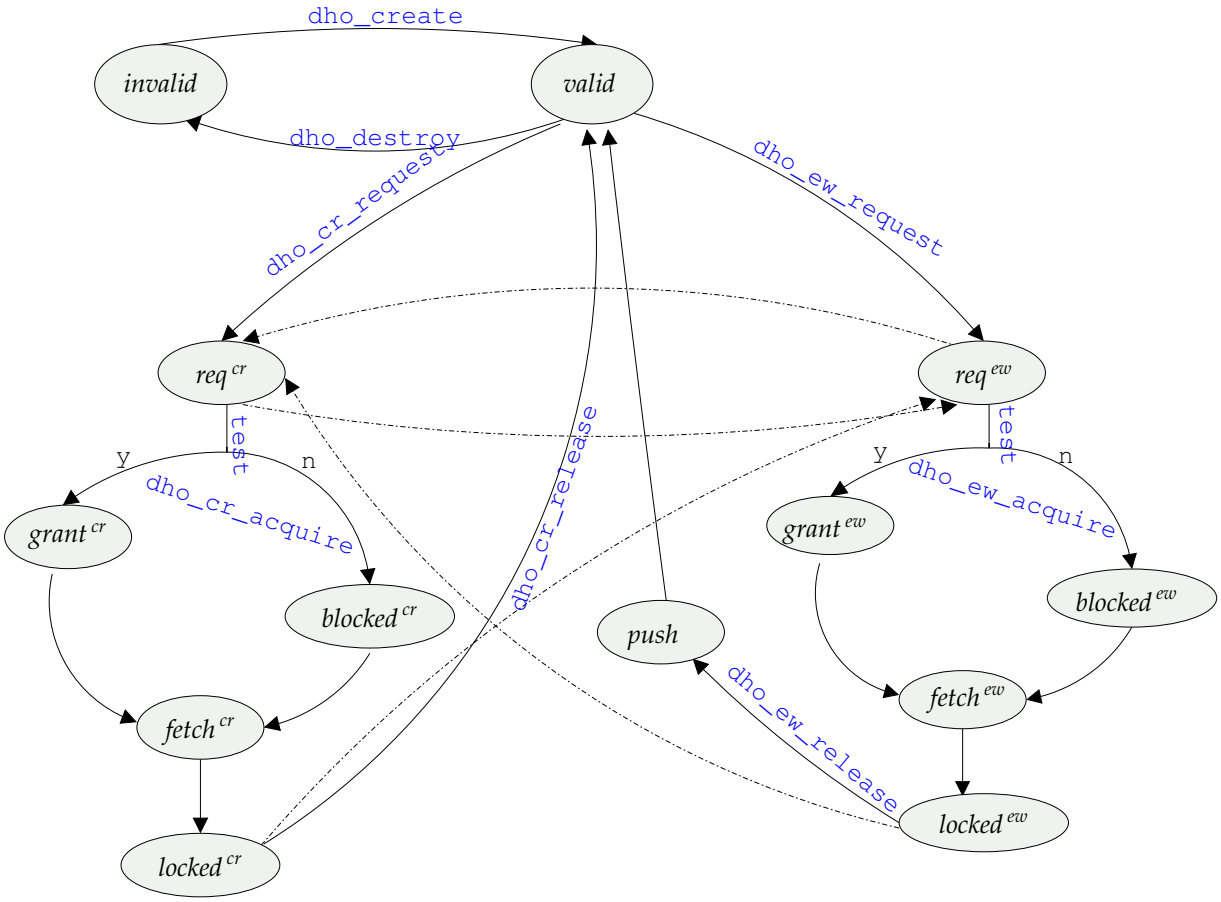


FIGURE 3.3 – Diagramme d'états d'un *handle*

Phase 1 Après une durée d'inactivité T_{idle}, le client envoie une demande d'accès en lecture à la ressource, par l'appel de la fonction `dho_cr_request`.

Dans le cas d'un éventuel *locking/mapping* exclusif déjà présent, la requête n'est pas servie immédiatement par le *gestionnaire de verrou* de la ressource, mais est plutôt insérée dans une queue (file d'attente).

Par ailleurs, le *gestionnaire de ressource* correspondant à la requête, reçoit un accusé de réception (`ACK`) de la part du *gestionnaire de verrou*. Celui-ci est retourné par la fonction `dho_cr_request` qui confirme l'enregistrement de la requête dans la queue.

Ainsi, le *client* passe à l'état *Req* en mode *non-bloquant* et le *handle* devient *req* $^{\text{cr}}$. Des événements analogues surgissent suite à un appel de la fonction `dho_ew_request` pour un accès exclusif. Dans ce cas, le *handle* devient *req* $^{\text{ew}}$. Rappelons que ces deux modes bloquant et *non-bloquant* opèrent séparément dans MPI et `POSIX-Pthreads`.

Phase 2 Une fois la ressource libérée d'un précédent *locking/mapping* qui inhibait l'accès en phase 1, le *gestionnaire de ressource* en est informé et le *handle* regagne l'état *grant* $^{\text{cr}}$, respectivement *grant* $^{\text{ew}}$. Notons par $T_{\text{WaitGrant}}$, la durée requise pour atteindre cet état.

Autrement, le *client* peut à tout moment parfaire le *lock* par l'appel de la fonction `dho_cr_acquire`, respectivement `dho_ew_acquire`. Il demeure cependant *bloquant* tant que le *lock* n'est pas obtenu. De même, le *handle* occupe l'état *blocked* $^{\text{cr}}$, respectivement *blocked* $^{\text{ew}}$. Le *client* s'accorde un temps T_{WBlocked} avant de franchir cette étape.

Phase 3 Dès lors que que le *lock* est accordé, le *gestionnaire de ressource* modifie l'état du *handle*. *fetch* $^{\text{ew}}$, respectivement *fetch* $^{\text{cr}}$ sont des états intermédiaires durant lesquels le *mapping* est réalisé. Ainsi, T_{fetch} est le temps requis pour cette opération.

Une fois le *mapping* réalisé, le *handle* devient *locked* $^{\text{cr}}$, respectivement *locked* $^{\text{ew}}$. La fonction `dho_cr_acquire`, respectivement `dho_ew_acquire` retourne un pointeur de la ressource. Ainsi, une copie de la ressource est instanciée localement, le *client* devient *locked* et le *handle locked* $^{\text{ew}}$, respectivement *locked* $^{\text{cr}}$.

Notons par T_{grant} le temps estimé avant l'appel de la fonction `dho_test`, et T_{blocked} la durée de séjour du *client* et du *handle* dans les états *Blocked* et *blocked* $^{\text{ew}}$, respectivement *blocked* $^{\text{cr}}$.

Phase 4 Suite à un temps applicatif noté T_{locked}, le *client* appelle la fonction `dho_cr_release`, respectivement `dho_ew_release` pour libérer la ressource. Dans le cas où le *lock* est exclusif, la modification de la donnée est propagée au serveur durant l'état transitoire *push* , pendant un temps T_{push}.

Le *gestionnaire de ressource* notifie la résiliation de la ressource par une mise à jour de l'état *valid* attribué au *handle*. Par conséquent, le *client* reprend son état initial *Idle*. Le cycle de vie **DHO** est ainsi accompli.

Le *client* peut à nouveau faire appel aux fonctions `dho_cr_request` ou `dho_ew_request`, selon qu'il souhaite accéder à la ressource en lecture ou en écriture. Le listing 3.1 présente un usage minimal de l'API **DHO**.

Listing 3.1 – Un exemple basique d'utilisation de l'API DHO

```
char const* name;
```

```
dho_t *a;
double DELAY, T_WBlocked, T_Lock;
dho_create(name, &a);
do {
  dho_ew_request(a);
  sleep(T_WBlocked);
  dho_test(a);
  dho_ew_acquire(a);
  sleep(T_Locked);
  dho_ew_release(a);
} while(time < DELAY);
dho_destroy(a);
```

Tant qu'il n'est pas *bloqué* , le *client* dispose de la fonction `dho_test` qui lui permet de connaître le stade de sa requête. En retour, la fonction `dho_test` renvoie l'état du *handle*. Aussi, il est possible d'interrompre le cycle **DHO** à n'importe quelle phase, par l'appel de la fonction `dho_destroy`. La conséquence directe de cet appel est la remise à l'état *invalid* au *handle* par le *gestionnaire de ressource*.

Il est important de noter que T_{Wblocked} et T_{locked} sont des durées non observées, mais plutôt imposées par l'application.

Autres cas spécifiques Nous avons présenté dans la section 3.2.3 les conséquences de l'utilisation des routines **DHO**, selon un ordre logique et un enchaînement classique d'actions entreprises par le *client*. Néanmoins, l'API **DHO** assume toutes les combinaisons possibles et agit en conséquence. Ainsi, le *client* peut faire appel à n'importe quelle fonction, et n'est donc pas tenu de respecter l'ordre logique du cycle tel que nous l'avons décrit plus haut.

En outre, le non respect du cycle **DHO** conduit son propriétaire (le *client*) à rejoindre immédiatement l'état initial *valid* . Par conséquent, le *handle* associé perd éventuellemnt son *lock* ainsi que son rang dans la queue. De même, toutes les durées estimées ($T_{\text{WaitGrant}}$ et T_{WBlocked}) sont initialisées.

Par exemple, si le *client* sollicite un accès en lecture par l'appel de la fonction `dho_cr_request`, il obtient un rang dans la queue et probablement le *lock*, et relance ensuite un appel `dho_ew_request` ; le *lock* et l'ancienne requête sont immédiatement annulés par le *gestionnaire de verrou*. Toutefois, la nouvelle requête est insérée dans la queue et reçoit un nouvel `ACK`. Un tel comportement est illustré par les lignes en pointillés de la figure 3.3.

3.2.3.1 Modélisation et spécification formelle

La ressource ne peut occuper que trois états potentiels, à savoir *libre* pour le cas où aucune requête ne lui est adressée, *accordé(ew)* pour le cas d'un seul écrivain, et *accordé(cr)* pour le cas de plusieurs lecteurs simultanés. Afin d'illustrer au mieux le

	valid	*req* ew	*grant* ew	*blocked* ew	*fetch* ew	*locked* ew	*push*	*invalid*
`create`	*valid*	*valid*	*valid*	–	–	*valid*	–	*valid*
`request`ew	*req* ew	*req* ew	*req* ew	–	–	*req* ew	–	*invalid*
`test`	*valid*	*req* ew |*grant* ew	*grant* ew	–	–	*locked* ew	–	*invalid*
`acquire`ew	*valid*	*blocked* ew	*fetch* ew	–	–	*locked* ew	–	*invalid*
`release`ew	*valid*	*valid*	*valid*	–	–	*push*	–	*invalid*
`destroy`	*invalid*	*invalid*	*invalid*	–	–	*invalid*	–	*invalid*

TABLE 3.1 – Table de transition d'états. Pour une question de lisibilité, seuls les états exclusifs sont représentés

comportement du cycle **DHO** à travers le *handle*, nous le modélisons par un automate déterministe à états finis (DFSA) [Sip97].

Soit $M = (Q, \Sigma, \delta, q_0, F)$ un 5-tuple, constitué de :

1. Un ensemble fini d'états Q d'un *handle* où :

$$Q = \{\mathit{valid}\ , \mathit{invalid}\ , \mathit{blocked}^{\,\mathrm{cr}}, \mathit{fetch}^{\,\mathrm{cr}}, \mathit{grant}^{\,\mathrm{cr}}, \mathit{locked}^{\,\mathrm{cr}}, \mathit{req}^{\,\mathrm{cr}},\ \mathit{blocked}^{\,\mathrm{ew}}, \mathit{fetch}^{\,\mathrm{ew}}, \mathit{grant}^{\,\mathrm{ew}}, \mathit{locked}^{\,\mathrm{ew}}, \mathit{req}^{\,\mathrm{ew}}, \mathit{push}\ \}$$

2. Un ensemble fini de symboles d'entrée Σ correspondant aux routines de l'API **DHO**, où :

$$\Sigma = \{\texttt{create}, \texttt{destroy}, \texttt{test}, \texttt{request}^{\mathrm{cr}}, \texttt{acquire}^{\mathrm{cr}}, \texttt{release}^{\mathrm{cr}},\ \texttt{request}^{\mathrm{ew}}, \texttt{acquire}^{\mathrm{ew}}, \texttt{release}^{\mathrm{ew}}\}$$

3. Un état de départ $q_0 \in Q$ où $q_0 = \mathit{invalid}$
4. Un ensemble d'états acceptant $F \subseteq Q$, où $F = \{\mathit{invalid}\ , \mathit{valid}\ \}$
5. Une fonction de transition $\delta : Q \times \Sigma \mapsto Q_\delta$ définie par la table de transition d'états (table 3.1).

3.2.3.2 Les différents chemins du cycle DHO

Soit T_{DHO} la durée d'un cycle **DHO**. La figure 3.3 démontre parfaitement les deux chemins envisageables, pour chaque type d'accès. Il en découle quatre chemins possibles :

Chemin 1 Accès partagé en mode *non-bloquant*

$$\{req^{\,\mathrm{cr}} \rightarrow grant^{\,\mathrm{cr}} \rightarrow fetch^{\,\mathrm{cr}} \rightarrow locked^{\,\mathrm{cr}} \rightarrow valid\ \}$$

Ce chemin s'accomplit lorsque le *client* demeure *non-bloquant* durant tout le cycle. Usuellement, ce cas de figure apparaît avec les premières requêtes enregistrées, ou dans des scénarios composés d'une succession de demandes en lecture. La durée totale du cycle est exprimée par :

$$T_{\mathrm{DHO}} = T_{\mathrm{WaitGrant}} + T_{\mathrm{grant}} + T_{\mathrm{fetch}} + T_{\mathrm{locked}} + T_{\mathrm{idle}}. \tag{3.1}$$

Chemin 2 Accès partagé en mode *bloquant*

$$\{req^{\,\mathrm{cr}} \rightarrow block^{\,\mathrm{cr}} \rightarrow fetch^{\,\mathrm{cr}} \rightarrow lock^{\,\mathrm{cr}} \rightarrow Valid\ \}$$

Ce chemin se produit lorsque le *client* passe au mode bloquant, suite à un appel de la fonction `dho_cr_acquire`. Dans ce schéma, plusieurs requêtes sont enregistrées par le *gestionnaire de ressource* et attendent d'être servies. Le temps du cycle correspondant est représenté par :

$$T_{\mathrm{DHO}} = T_{\mathrm{Wblocked}} + T_{\mathrm{blocked}} + T_{\mathrm{fetch}} + T_{\mathrm{locked}} + T_{\mathrm{idle}}. \tag{3.2}$$

Chemin 3 Accès exclusif en mode *non-bloquant*

$$\{req^{\,\mathrm{ew}} \rightarrow grant^{\,\mathrm{ew}} \rightarrow fetch^{\,\mathrm{ew}} \rightarrow lock^{\,\mathrm{ew}} \rightarrow Push\ \rightarrow Valid\ \}$$

De même que le chemin 1, ce scénario survient lorsque le *client* sollicite la ressource en mode écriture alors que cette dernière est plus ou moins libre. Cela arrive fréquemment avec les premières requêtes enregistrées, ou lorsque la queue n'est pas surchargée. La durée du cycle **DHO** est exprimée comme suit :

$$T_{\mathrm{DHO}} = T_{\mathrm{WaitGrant}} + T_{\mathrm{grant}} + T_{\mathrm{fetch}} + T_{\mathrm{locked}} + T_{\mathrm{push}} + T_{\mathrm{idle}}. \tag{3.3}$$

Chemin 4 Accès exclusif en mode *bloquant*

$$\{req^{\,\mathrm{ew}} \rightarrow block^{\,\mathrm{ew}} \rightarrow fetch^{\,\mathrm{ew}} \rightarrow lock^{\,\mathrm{ew}} \rightarrow Push\ \rightarrow Valid\ \}$$

Cette situation représente le cas le plus général en terme d'accès exclusif. Plusieurs requêtes devancent celle enregistrée dans la queue. Le *client* passe habituellement au mode bloquant avant d'obtenir le *lock*. La durée du cycle **DHO** correspondant est définie par :

$$T_{\mathrm{DHO}} = T_{\mathrm{Wblocked}} + T_{\mathrm{blocked}} + T_{\mathrm{fetch}} + T_{\mathrm{locked}} + T_{\mathrm{push}} + T_{\mathrm{idle}}. \tag{3.4}$$

3.3 Critères d'évaluation et paramètres d'exécution

Le reste de ce chapitre présente une première expertise de l'API sur la base d'une gestion centralisée de la ressource. Par cette étude, nous visons à analyser le comportement de notre librairie, notamment du délai du cycle **DHO**.

L'évaluation de notre API nous astreint à considérer quelques facteurs relatifs aux délais qui sont inclus dans le cycle, et ceux qui se rapportent aux composants matériels et logiciels du cadre expérimental.

Les différents paramètres pris en compte dans l'analyse des performances de **DHO** sont présentés ci-dessous :

3.3.1 Durées observées

Nous planifions ci-après (section 3.4) des scénarios dans lesquels les *clients* entament plusieurs cycles **DHO**. Pour faciliter notre étude, nous supposons T_{idle} *nul* dans toutes nos expériences. Autrement dit, le *client* redemande le *lock* aussitôt qu'il l'a résilié dans un cycle antérieur. Ainsi, les *clients* se disputent l'accès à la ressource selon le paradigme client-serveur.

En outre, nous nous appuyons sur les fondements du système M/M/1 des files d'attente pour formaliser le mécanisme **DHO** [STF99]. Notons que cette file est caractérisée par une arrivée poissonnienne de taux λ et une durée de service exponentielle de taux μ. La file d'attente M/M/1 est considérée comme un processus de naissance et de mort.

Par ailleurs, la queue **DHO** se singularise par deux types de requêtes. Nous prenons N_c *clients* comme source d'arrivée des requêtes.

Tout au long de son parcours, la requête suit un chemin et franchit des étapes. Nous basons notre étude sur l'observation du temps du cycle et des durées qui le composent.

Soit T_{Wait} le temps d'attente d'une requête dans la queue, c'est-à-dire le temps écoulé entre l'appel de la requête et la fin du *mapping*. Cela traduit les évènements qui précèdent le passage du *handle* à l'état *locked* . L'estimation de cette durée est étroitement liée au chemin suivi par le *handle* dans le cycle (bloquant ou *non-bloquant*). Ainsi, lorsque le blocage est absent du cycle, le temps T_{wait} est calculé comme suit (chemins 1 et 3) :

$$T_{\text{Wait}} = T_{\text{WaitGrant}} + T_{\text{grant}} + T_{\text{fetch}}. \tag{3.5}$$

De même, l'équation suivante résulte des chemins 2 et 4 du cycle relatif au mode *bloquant* :

$$T_{\text{Wait}} = T_{\text{Wblocked}} + T_{\text{blocked}} + T_{\text{fetch}}. \tag{3.6}$$

Soit T_{Blocking} le temps durant lequel le *client* demeure bloquant avant l'obtention du *lock*. En d'autres termes, le temps entre l'appel de la fonction `dho_ew_acquire` ou `dho_cr_acquire` et la fin du *mapping*.

$$T_{\text{Blocking}} = T_{\text{blocked}} + T_{\text{fetch}}. \tag{3.7}$$

Les durées T_{Wait} et T_{Blocking} sont observées conjointement avec T_{DHO}.

3.3.2 Plate-forme matérielle d'expérimentations

Nos expériences se déroulent en deux phases : Dans une première étape, nous menons des tests en simulation en prenant comme modèle de déploiement un *Cluster* dont les caractéristiques matérielles correspondent à celles qui existent dans une plate-forme réelle. Nous voulons d'abord nous assurer de la bonne qualité de notre application avant de franchir la seconde étape. Les mêmes tests sont exécutés par la suite, sur ladite plate-forme en réel.

En réalité, la plate-forme d'exécution correspond à la grille Grid'5000 [gri]. L'infrastructure matérielle de la grille est décrite ultérieurement dans ce document (section 6.4.1 du chapitre 6). Nous nous intéressons pour l'instant au *Cluster* déployé dont la description réalistique est la suivante :

```
<cluster id="Nancy" prefix="Nancy-" suffix=".Grelon.grid5000.fr"
radical="1-120" power="3.185E9" bw="1.25E8" lat="1.0E-4"
bw_bb="1.25E9" laT_bb="1.0E-4"/>
```

Chaque nœud du *Cluster* est relié à un *backbone* par un lien privé. Le *backbone* est interconnecté à la grille grâce au réseau **Renater**[1]. Nous enregistrons le cumul des latences, et le minimum de bande passante pour les deux liaisons. La réservation des nœuds est gérée par le système de *batch* `OAR`. Il permet de spécifier nos besoins en matière de ressources matérielles pour les expériences proprement dites. Ainsi, les processus associés aux *clients* (les *gestionnaires de ressources*) sont distribués sur les 119 nœuds, alors qu'un nœud du *Cluster* héberge la ressource (serveur).

3.3.3 Modèle et librairie de communication

Dans la section 2.1.5, nous avons donné un aperçu global de `SimGrid`. Notre choix a été porté sur ce *Toolkit* pour deux raisons :

Premièrement, il permet aisément de déployer des plate-formes par dessous des programmes distribués, et ce pour n'importe quelle librairie. Comme nous l'avons vu dans la section 2.1.5, le module **SURF** situé en bas de l'architecture modulaire de `SimGrid` offre cette opportunité.

Le deuxième motif pour lequel nous avons choisi `SimGrid`, concerne la librairie exploitée et le modèle de communication. Nous le justifions ci-dessous :

`SimGrid` s'appuie sur un modèle de communication [VL09] qui garantit une bonne précision pour les messages de taille supérieure à 100 `KiB`, d'où l'équation suivante :

$$T = \alpha \cdot L + \frac{S}{\min(\beta \cdot bw, \gamma')}, \tag{3.8}$$

Où $\alpha = 10.4$ et $\beta = 0.92$ sont des facteurs de correction.

1. Le réseau national de télécommunications pour la Technologie, l'enseignement et la recherche

Cette garantie est cruciale pour nos expériences car nous envisageons des *Benchmarks* avec des tailles de la donnée allant jusqu'à 1 Go.

γ' est un paramètre qui fixe la taille de la fenêtre (γ_{TCP}) dans les connexions TCP.

Nous avons : $\gamma' = \gamma_{\text{TCP}}/2L$.

γ_{TCP} est fixé à 10^7, tandis que L est la latence cumulée :

$L = 2\ell + \ell_{bb}$.

Par conséquent, et selon l'équation 3.8, chaque flux démarre réellement après αL. Le flux TCP atteint 92% de la bande passante physique.

De toutes les manières, nous procédons à des variations dans la latence et dans la bande passante dans certaines de nos expériences, afin de nous assurer de la bonne performance du modèle.

Dans `SimGrid`, nous avons opté pour l'interface *Grid Reality And Simulation environment* à base de `sockets`, `GRAS` ([Qui06b], [CLQ08]) que nous avons intégrée à notre code source écrit en lange C. `GRAS` est l'une des librairies de la boite à outils `SimGrid` [VL09], qui met à disposition des utilisateurs, les fonctionnalités pour le développement et la simulation d'applications distribuées sur des environnements hétérogènes. Le point fort de `GRAS` est qu'il permet de redéployer une expérience en simulation, sur une plate-forme réelle d'exécution, sans avoir à modifier ni à recompiler le code.

`GRAS` fait communiquer des processus sur la base de messages et d'évènements. Un bref aperçu du modèle est donné ci-dessous :

Les messages sont envoyés par la fonction `gras_msg_send` qui prend comme argument une charge utile. Les processus attachent des fonctions dites `Callback` aux messages reçus. Ces fonctions décrivent l'action à suivre lorsqu'un message rattaché à un `Callback` est reçu.

Aussi, les messages reçus ne sont pas traités à leur arrivée mais plutôt lorsque les processus déclarent qu'ils sont prêts à recevoir un tel évènement. `gras_msg_handle` est par exemple une fonction utilisée à cette fin. Ainsi, les messages qui ne sont pas traités sur le champ, sont insérés dans la file d'attente associée du processus qui les reçoit.

Outre la notion d'évènement, les processus peuvent attendre explicitement un message par la fonction `gras_msg_wait`.

3.4 Analyse des résultats expérimentaux

l'API **DHO** a été implémenée et exécutée avec succès, aussi bien en simulation que sur le *Cluster* de Grid'5000. Dans la présente section, nous analysons le comportement du modèle **DHO**, suite à l'invocation des routines qui lui sont associées.

3.4.1 Plan des expériences

Nous démarrons notre étude par l'observation des temps relatifs aux phases initiale (T_{create}) et terminale (T_{destroy}) de l'expérience, afin de mesurer les retards induits par la

ℓ, ℓ_{bb}(s)	$T_{(Create)}$(s)	$T_{(Destroy)}$(s)
10^{-5}	$145.6 \cdot 10^{-5}$	$72.8 \cdot 10^{-5}$
$5 \cdot 10^{-5}$	$728 \cdot 10^{-5}$	$364 \cdot 10^{-5}$
10^{-4}	$145.6 \cdot 10^{-4}$	$72.8 \cdot 10^{-4}$

TABLE 3.2 – Les délais de création et de destruction du *handle*

création et la destruction du *handle*.

Sont observées par la suite les durées de *mapping* (T_{fetch}) et de mise à jour (T_{push}) de la ressource. Par ces expériences, nous ciblons le modèle de communication de `SimGrid`. Notre but dans cette partie est de vérifier la consommation de la bande passante et de confirmer la bonne qualité du modèle.

L'évaluation du cycle **DHO** constitue la majeure partie de cette étude. Afin de couvrir tous les aspects du modèle, nous prévoyons plusieurs *Benchmarks* incluant les deux types de requêtes.

Le temps applicatif T_{Wblocked} sera négligé au début des tests. Par contre, nous étudions l'influence du temps d'occupation (T_{locked}) ainsi que celle de la taille de la ressource.

3.4.2 Étude des délais de création et de destruction du *handle*

La création du *handle* est l'opération qui suscite un premier dialogue entre le *gestionnaire de ressource* (client) et le *gestionnaire de verrou* (serveur). Le passage à l'état *valid* pour le *handle* traduit (1) la présence de la ressource demandée sur le serveur, et (2) l'ouverture d'une *socket* entre le *client* et le serveur.

Inversement, la destruction du *handle* rompt le lien entre le *client* et le serveur pour repositionner le *handle* à l'état *invalid* . Nous étudions l'influence de la latence de communication sur les durées T_{create} et T_{destroy}. À cet effet, nous prenons trois valeurs de latences ℓ et ℓ_{bb} : 10^{-5}, $5. \cdot 10^{-5}$ et 10^{-4}(s).

D'après la table 3.2, T_{create} est d'au moins $145.6 \cdot 10^{-5}$ et d'au plus $145.6 \cdot 10^{-4}$. Nous déduisons la relation suivante :

$$T_{(create)} \simeq 2 \cdot \alpha \cdot \eta \cdot L \tag{3.9}$$

Aussi, le délai observé pour la phase destruction du *handle* varie entre $72.8 \cdot 10^{-5}$ et $72.8 \cdot 10^{-4}$ d'où la relation suivante :

$$T_{(destroy)} \simeq \alpha \cdot \eta \cdot L \tag{3.10}$$

$\alpha \simeq 10.4$, comme nous l'avons indiqué auparavant, cette constante est le facteur de correction du modèle de communication de `Simgrid` (équation 3.8).

$\eta \simeq 7$, ce facteur est le coût enregistré par notre application.

Ces résultats sont cohérents puisque le création du *handle* nécessite plus d'échanges entre le *client* et le serveur.

3.4.3 Étude du *mapping* et de la mise à jour de la ressource

Cette partie des expériences concerne l'observation des délais durant lesquels le *locking/mapping* (T_{fetch}) et la propagation de la donnée (T_{push}) sur le serveur sont effectués. Ces durées dénotent simplement le temps de communication entre le *client* et le serveur. Pour cela, nous diversifions la latence et la bande passante comme précédemment, mais aussi la taille de la donnée avec des valeurs comprises entre 100 `KiB` et 1 `GiB`. Les *clients* demandent l'accès en écriture à la ressource à 100 reprises, ce qui revient à accomplir 100 cycles **DHO**. Les *clients* gardent le *lock* correspondant à la ressource durant un temps compris dans l'intervalle $0 \ldots 10$ (T_{locked}).

En dépit de toutes ces variations, nous avons constaté que T_{push} correspondait exactement au temps T (équation 3.8) annoncé dans SimGrid :

$$T_{\text{push}} = \alpha \cdot L + \frac{S}{\min(\beta \cdot bw, \gamma')}. \tag{3.11}$$

Aussi, T_{fetch} est d'environ T_{push} selon l'équation suivante :

$$T_{\text{fetch}} = T_{\text{push}} + \alpha \cdot L. \tag{3.12}$$

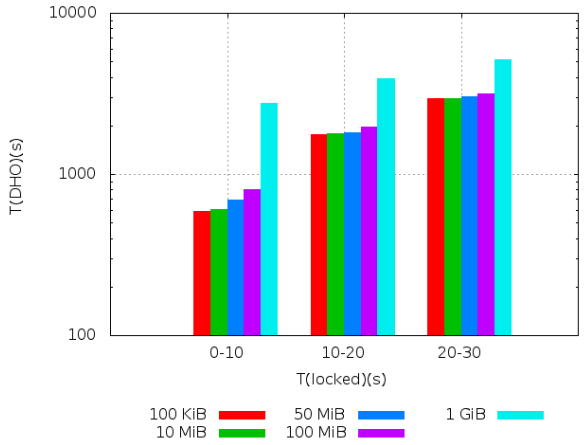
Résultat. Par les deux formules $T_{\textbf{push}} = T$ et $T_{\textbf{fetch}} \simeq T_{\textbf{push}}$, nous constatons que la bande passante est totalement consommée. Ceci témoigne de la bonne qualité du modèle de communication de `SimGrid`.

3.4.4 Analyse de la performance du cycle DHO pour les *locks* exclusifs en l'absence de retard au blocage

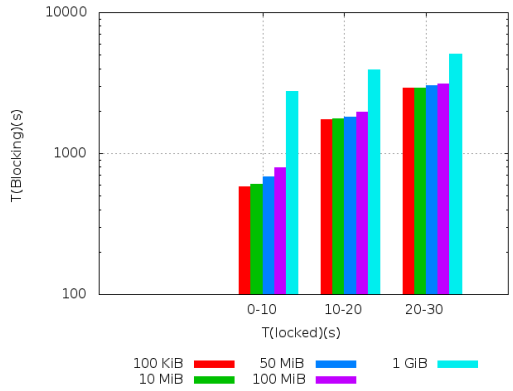
L'analyse du cycle **DHO** prend part dans cette section. Cependant, nous voulons encore une fois consolider notre choix porté sur le modèle de `SimGrid`. En d'autres termes, nous nous assurons que l'ensemble (**DHO**, librairie `GRAS` et modèle de communication de `SimGrid`) progresse sans difficulté et sans perte de performances. Ainsi, nous aspirons à travers ces expérimentations, à mesurer l'adéquation de **DHO** face à des circonstances variables et extrêmes.

Notre étude repose sur l'observation des durées qui composent le cycle (section 3.3.1). Les premières expériences s'appliquent aux requêtes exclusives uniquement.

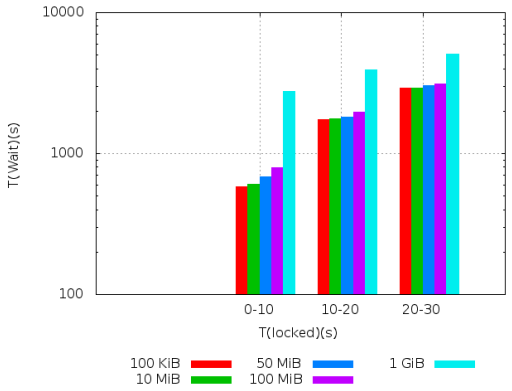
Comme nous l'avons précisé plu haut, nous fixons $T_{\text{Wblocked}} = 0$ (ce retard est étudié un peu plus loin) de sorte que le *client* appelle l'une des fonctions *acquire* dès qu'il reçoit un accusé de réception du serveur par son *gestionnaire de verrou*. La durée du *lock* est variée avec des valeurs comprises dans les intervalles $0 \ldots 10$, $10 \ldots 20$ et $20 \ldots 30$ (s) (table 3.3). De plus, la taille de la ressource est diversifiée avec des valeurs comprises entre 100 `KiB` et 1 `GiB`. Les durées reportées sur la table 3.3 dénotent les moyennes de 100 exécutions.



(a) Observation du cycle $\overline{T_{\text{DHO}}}$ en fonction à la durée d'occupation du *lock* $\overline{T_{\text{locked}}}$



(b) Observation du temps de blocage $\overline{T_{\text{blocking}}}$ en fonction à la durée d'occupation du *lock* $\overline{T_{\text{locked}}}$



(c) Observation du temps d'attente d'une requête $\overline{T_{\text{Wait}}}$ dans la queue en fonction à la durée d'occupation du *lock* $\overline{T_{\text{locked}}}$

FIGURE 3.4 – Durées observées : T_{DHO}, T_{Wait} et T_{Blocking}

Ressource	$\overline{T_{\text{locked}}}(s)$	$\overline{T_{\text{Wait}}}(s)$	$\overline{T_{\text{Blocking}}}(s)$	$\overline{T_{\text{DHO}}}(s)$
100 `KiB`	0-10	$5.833 \cdot 10^2$	$5.833 \cdot 10^2$	$5.883 \cdot 10^2$
	10-20	$17.449 \cdot 10^2$	$17.4491 \cdot 10^2$	$17.599 \cdot 10^2$
	20-30	$29.164 \cdot 10^2$	$29.164 \cdot 10^2$	$29.414 \cdot 10^2$
10 `MiB`	0-10	$6.024 \cdot 10^2$	$6.024 \cdot 10^2$	$6.075 \cdot 10^2$
	10-20	$17.67 \cdot 10^2$	$17.67 \cdot 10^2$	$17.821 \cdot 10^2$
	20-30	$29.336 \cdot 10^2$	$29.336 \cdot 10^2$	$29.587 \cdot 10^2$
50 `MiB`	0-10	$6.921 \cdot 10^2$	$6.867 \cdot 10^2$	$6.867 \cdot 10^2$
	10-20	$18.240 \cdot 10^2$	$18.087 \cdot 10^2$	$18.087 \cdot 10^2$
	20-30	$30.452 \cdot 10^2$	$30.197 \cdot 10^2$	$30.197 \cdot 10^2$
100 `MiB`	0-10	$7.942 \cdot 10^2$	$7.942 \cdot 10^2$	$8.002 \cdot 10^2$
	10-20	$19.583 \cdot 10^2$	$19.583 \cdot 10^2$	$19.742 \cdot 10^2$
	20-30	$31.282 \cdot 10^2$	$31.282 \cdot 10^2$	$31.541 \cdot 10^2$
1 `GiB`	0-10	$27.569 \cdot 10^2$	$27.569 \cdot 10^2$	$27.712 \cdot 10^2$
	10-20	$39.20 \cdot 10^2$	$39.444 \cdot 10^2$	$39.444 \cdot 10^2$
	20-30	$50.86 \cdot 10^2$	$50.86 \cdot 10^2$	$51.204 \cdot 10^2$

TABLE 3.3 – Durées observées en prenant une bande passante BW $= 1.25 \cdot 10^8 (B/s)$ et une latence L $= 10^{-5}(s)$.

Premièrement, nous constatons que le temps d'attente d'une requête T_{wait} équivaut environ au temps de blocage, T_{blocking} du *handle*. Ceci est conforme aux équations 3.6 et 3.7 puisque T_{Wblocked} est nul.

Deuxièmement, l'écart entre les durées observées pour les ressources (entre 100 `KiB` et 10 `MiB`) est minime. Cela s'explique par le fait que les valeurs de (T_{fetch}) et (T_{push}) sont négligeables pour les petites tailles de la ressource (formules 3.6 et 3.7).

Le cycle **DHO** croît inévitablement avec la durée du *lock* (figure 3.4(a)). Toutefois, l'écart entre les délais T_{wait}, T_{blocking} et T_{DHO} n'est pas très significatif pour les ressources de grande taille (table 3.3). En effet, tandis que le temps du cycle est de $31.541 \cdot 10^2$ pour une ressource de 100 `MiB` par exemple, il est seulement de $51.204 \cdot 10^2$ pour une taille 10 fois plus grande. Nous interprétons ce résultat par la prédominance de la durée du *lock* sur le cycle. Quant à la longueur de la queue ($\overline{q}$), elle est d'environ 118 pour tous les cas.

La même interprétation est déduite pour les durées d'attente d'une requête dans la queue (T_{wait}), et de blocage (T_{blocking}) du *handle* (figures 3.4(b), 3.4(c)).

Outre les valeurs recueillies, nous mesurons T_{Dho} en prenant 2 valeurs de bande passante et 3 valeurs de latence. La table 3.4 reporte les délais observés pour une durée du *lock* prise dans l'intervalle $0 \ldots 10$, et une taille de la ressource fixée à 100 `MiB`.

Les résultats obtenus jusqu'à présent nous amènent à conclure que le cycle **DHO**

Latence(s)	Bande passante(B/s)	$\overline{T_{\text{DHO}}}(s)$
10^{-5}	$1.25 \cdot 10^8$ 10^9	800.20 607.36
$5 \cdot 10^{-5}$	$1.25 \cdot 10^8$ 10^9	800.69 610.82
10^{-4}	$1.25 \cdot 10^8$ 10^9	802.61 609.72

TABLE 3.4 – Moyenne du cycle $\overline{T_{\text{DHO}}}$. La taille de la ressource est fixée à 100 `MiB`. La durée du *lock* (T_{locked}) est prise dans l'intervalle $0 \dots 10(s)$.

dépend principalement de T_{locked}, T_{fetch}, T_{push}, de la longueur de la queue ainsi que de quelques facteurs liés à la latence d'où :

$$\overline{T_{\text{DHO}}} = [\overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \overline{T_{\text{push}}} + \alpha \cdot \eta \cdot L](\overline{q} + 1) \tag{3.13}$$

Où α = 10.4 et $\eta \simeq 7$.

Notons que le temps consacré par le *gestionnaire de verrou* à chaque requête se traduit dans la théorie des files d'attente par le temps moyen de service, noté $1/\mu_1$. Il est déduit de la formule 3.13 :

$$\frac{1}{\mu_1} = \overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \overline{T_{\text{push}}} \tag{3.14}$$

D'après les fondements du système M/M/1 des files d'attente, ρ représente le rapport entre le taux moyen d'arrivée (λ) et le taux moyen de service (μ) ; $\rho = \frac{\lambda}{\mu}$.

La longueur moyenne des requêtes dans un système stable est égale à leur fréquence moyenne d'arrivée (λ) multipliée par le temps du cycle (T_{DHO}) d'où :

$\overline{q} = \lambda \cdot T_{\text{DHO}}$

Le système est stable lorsque $\rho < 1$.

Résultat. Pour toutes les valeurs du cycle (table 3.3), selon la formule 3.13 et la loi de *Little's* [Lit61], $\rho < 1$.
L'API **DHO** présente un système stable pour les requêtes exclusives.

3.4.5 Déduction de la durée du cycle pour les *locks* partagés

À partir de la formule 3.13, nous pouvons inférer la moyenne du cycle pour les demandes exclusivement en accès partagé :

$$\overline{T_{\text{DHO}}} = [T_{\text{fetch}} + \alpha \cdot \eta \cdot L](\overline{q} + 1) + T_{\text{locked}}. \tag{3.15}$$

Une série d'expériences similaire relative à ce type d'accès a été conduite et a confirmé la formule 3.15. Nous soulignons que, dans ce cas, le cycle suit le chemin 1 (section 3.2.3.2). De cette façon, le *client* et le *handle* ne séjournent pas dans le mode bloquant. En effet, les requêtes de ce type sont servies quasi immédiatement.

Notons également que le délai T_{locked} de l'ensemble des requêtes précédentes, ainsi que T_{push} n'ont aucun impact sur le cycle et de ce fait, n'interviennent pas dans la formule 3.15. Cela implique que dans ce scénario, une requête ne dépend que du délai du *mapping* de ceux qui la précèdent dans la queue.

En effet, le *gestionnaire de verrou* (serveur) n'attend que la propagation de la ressource en mémoire locale du *client* pour traiter la requête suivante. Nous déduisons que le temps de service $1/\mu_2$ pour les demandes d'accès partagé est approximativement T_{fetch}.

Résultat. L'intervention du *gestionnaire de verrou* pour les requêtes partagés se résume à l'opération de *mapping*. Le temps de service $1/\mu_2 \simeq T_{\text{fetch}}$

3.4.6 Évaluation du cycle pour les *locks* asynchrones

Les résultats obtenus à ce stade, attestent d'une robustesse et de la stabilité du modèle. Néanmoins, le temps T_{Wblocked} a été négligé jusqu'à présent. En fait, dans les calculs asynchrones, ce délai exprime le retard du *client* sur le mode bloquant. Ce mode peut être dispensé si le *lock* est accordé d'avance. Sémantiquement, l'utilisation de l'API **DHO** suppose ce retard à l'appel de l'une des fonctions *acquire*. Il permet à l'application de poursuivre son exécution indépendamment de la ressource.

Ainsi, nous étudions l'influence de ce délai applicatif sur la durée du cycle. Pour cela, nous fixons le temps du *lock* à 5(s) ($T_{\text{locked}} = 5$).

La figure 3.5 montre que ce retard ne rallonge pas le cycle. Il demeure quasi constant en dépit des différentes valeurs de T_{Wblocked}. En observant de plus près les durées qui pèsent sur le cycle, nous constatons que la différence se joue entre la durée d'attente d'une requête, et le temps de blocage. En effet, lorsque T_{wait} croît, T_{Blocking} décroît et inversement. Cette observation s'avère avantageuse pour notre modèle puisqu'aucun surcoût n'est enregistré dans le cycle.

Résultat. L'absence de surcoût induit par le retard T_{Wblocked} démontre la présence d'un recouvrement (*Overlapping*) présent entre le *gestionnaire de verrou* (serveur) et l'application (*client*). Alors que le premier est occupé à traiter une requête, le deuxième poursuit l'exécution de son application indépendamment de l'API.

3.4.7 Évaluation du cycle pour les *locks* exclusifs et partagés

Le dernier scénario fusionne les requêtes en lecture et en écriture. Ainsi, les *clients* appellent de manière altérnée la fonction `dho_ew_request` puis, `dho_cr_request`

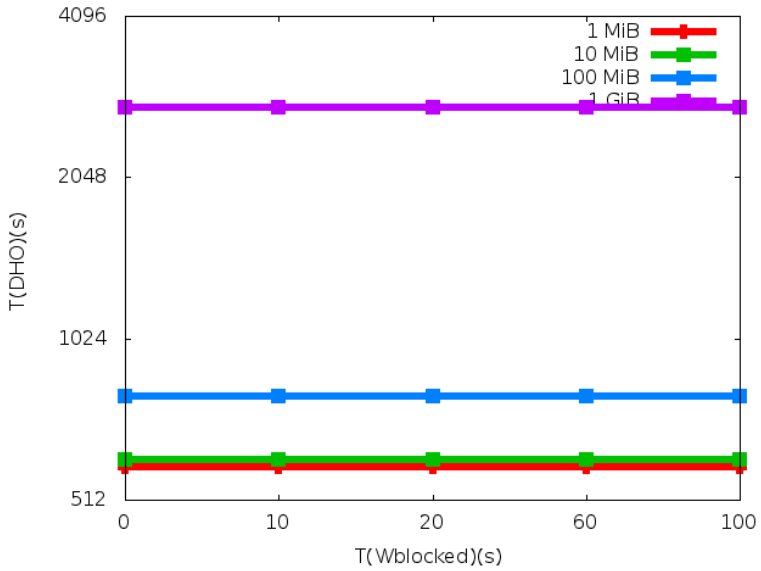


FIGURE 3.5 – Observation du cycle en variant T_{Wblocked}. Le temps du *lock* est fixé à $T_{\text{locked}} = 5(s)$.

après la libération du *lock* exclusif. Les durées applicatives T_{locked} et T_{Wblocked} sont prises dans l'intervalle $0..10$. Les moyennes des durées observées sont repportées dans la table 3.5.

$\overline{T_{\text{DHO}}}$(s)	$\overline{T_{\text{Wait}}}$(s)	$\overline{T_{\text{Blocking}}}$(s)	$\bar{q}$
$4.463 \cdot 10^2$	$4.441 \cdot 10^2$	$4.13 \cdot 10^2$	$1.143 \cdot 10^2$

TABLE 3.5 – Moyenne du cycle $\overline{T_{\text{DHO}}}$. La durée du *lock* ($T_{\text{locked}}$) et le retard au blocage (T_{Wblocked}) sont pris dans l'intervalle $0 \ldots 10$(s), la bande passante BW $= 1.25 \cdot 10^8$ (B/s) et la latence L $= 10^{-5}$(s).

Puisque notre modèle s'apparente à celui M/M/1 des files d'attente, nous déduisons que le taux moyen de service μ dans ce cas de figure est noté par :

$$\mu = \frac{\mu_1 + \mu_2}{2}. \tag{3.16}$$

Où $1/\mu_1$ et $1/\mu_2$ sont les temps moyens de service relatifs respectivement aux *locks* exclusifs et partagés. Par ailleurs, les calculs attestent que ρ avoisine la valeur 0.4 ($\rho < 1$).

3.5 Synthèse du chapitre

Nous avons présenté dans ce chapitre, une première étude de l'API **DHO** selon l'approche client-serveur. Son élaboration requiert l'intéraction entre les *gestionnaires de ressources* au niveau des *clients* et le *gestionnaire de verrou* associé à la ressource à l'échelle du serveur. La modélisation du cycle par un automate d'états finis nous a permis d'avoir une vision plus précise sur l'enchaînement des évènements enclenchés par les routines de l'API.

Les expériences déployées pour l'analyse de l'API, ont couvert une large variation de paramètres, relatifs aussi bien à la plate-forme d'expérimentation (bande passante et latence de communication) qu'à ceux issus de l'application (délais applicatifs et taille de la ressource).

Les délais observés pour les *locks* en lecture comme en écriture attestent d'une bonne performance de **DHO**. De plus, la stabilité du modèle a été confirmée et validée par le système M/M/1 des files d'attente.

Sur le plan expérimental, nous déduisons que notre sélection de : (1) l'environnement `GRAS` à base de *sockets*, (2) la taille de la fenêtre γ_{TCP} et (3) du modèle de communication de SimGrid est justifiée.

DHO admet les *locks* asynchrones. Ils constituent le cas d'utilisation générale de l'API. Ce type de *locks* confirme que dans l'architecture sous-jacente de **DHO**, le contrôle d'une requête par le serveur couvre le calcul de l'application. Cette observation présente un atoût à notre modèle car, elle dénote l'absence d'un surcoût.

Enfin, cette étude originelle du modèle **DHO**, conformément au paradigme client-serveur, a fait l'objet d'une publicaion [HGB11]. Ces premiers résultats nous ont conduit à envisager une deuxième approche totalement distribuée. Ce fut l'objet d'une étude élaborée rapportée dans les chapitres 4, 5 et 6 où d'autres stratégies et formalismes sont énoncés.

Chapitre 4

Algorithme dynamique d'exclusion mutuelle à *locks* exclusifs (ADEMLE)

Sommaire

4.1 L'Algorithme distribué d'exclusion mutuelle de Naimi et Tréhel . . . 60
4.1.1 Les bases de l'algorithme . . . 60
4.1.2 Requêtes concurrentes . . . 62
4.2 Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs (ADEMLE) . . . 65
4.2.1 Conditions nécessaires au maintien de la connectivité des deux structures . . . 65
4.2.2 Operations atomiques et notion d'état . . . 65
4.2.3 Traitement d'une requête dans l'**ADEMLE** . . . 66
4.3 Traitement de la déconnexion d'un processus dans l'ADEMLE . . . 73
4.3.1 Contraintes de déconnexion d'un processus . . . 75
4.3.2 Modalités de déconnexion d'un processus . . . 76
4.3.3 Envoi d'un ***Parent*** sortant à un demandeur de jeton . . . 78
4.3.4 Complexité de départ d'un processus dans l'**ADEMLE** . . . 78
4.4 Preuve de l'ADEMLE . . . 79
4.4.1 Propriété de *vivacité* . . . 79
4.4.2 Propriété de *sûreté* . . . 80
4.5 Synthèse du chapitre . . . 80

NOus avons parcouru dans le chapitre 2 les algorithmes fondamentaux d'exclusion mutuelle existants dans la littérature. Ces algorithmes se répartissent en deux grandes familles : Ceux fondés sur le principe de consensus et de l'élection, et ceux basés sur la circulation d'un jeton. Ces derniers présentent une faible complexité en terme d'envoi de messages.

Notre travail repose sur l'algorithme distribué d'exclusion mutuelle de Naimi et Tréhel [NTA96a]. Sa faible complexité n'est pas le seul facteur qui justifie notre choix. En vue d'une intégration ultérieure à un modèle distribué de partage de données (voir le chapitre 6), nous considérons que parmi les algorithmes distribués de sa catégorie, celui de Naimi et Tréhel offre une structure interne qui est la plus adaptée à des environnements distribués à grande échelle.

Nous commençons ce chapitre par présenter les fondements de l'algorithme Naimi-Tréhel, en mettant l'accent sur les requêtes concurrentes (section 4.1). Le reste de ce chapitre décrit notre première extension à l'algorithme de Naimi et Tréhel régie par l'**ADEMLE**, dont l'objectif est de fournir un aspect dynamique aux processus qui le composent (section 4.2).

L'**ADEMLE** reconsidère la façon dont les requêtes sont traitées par les processus du système, aussi bien dans l'envoi que dans la réception (section 4.2.3). Nous apportons une description formelle à notre algorithme étendu dans laquelle des états sont attribués au processus (section 4.2.3.3). La déconnexion des processus est étudiée dans tous ses aspects. Ainsi, les modalités de départ d'un processus donné sont exposées, de même que la complexité des opérations incluses dans l'**ADEMLE**.

Enfin, nous donnons une peuve théorique de l'algorithme étendu pour ainsi vérifier les propriétés de *sûreté* et de *vivacité* de l'exclusion mutuelle (section 4.4).

4.1 L'Algorithme distribué d'exclusion mutuelle de Naimi et Tréhel

L'algorithme Naimi-Tréhel est une brique de base pour les algorithmes distribués d'exclusion mutuelle. Basé sur la circulation d'un jeton à travers des processus, il assure l'unicité d'accès à la **section critique** et présente l'avantage d'avoir un faible surcoût en terme de transmission de messages.

4.1.1 Les bases de l'algorithme

Le point marquant de l'algorithme est dans les deux structures logiques sur lesquelles il repose. Nous les décrivons ci-après :

La première structure est arborescente à caractère dynamique, son évolution est rattachée aux requêtes émises. La *racine* représente le dernier processus qui a émis une demande d'accès à la **section critique**. Nous appelons cette structure l'arbre ***Parent***. Initialement, tous les processus pointent sur la même *racine* qui est en possession du jeton.

Cette situation reflète le point de départ de l'algorithme. Pendant leur émission, les requêtes sont propagées le long de l'arbre jusqu'à la *racine*.

La deuxième structure est une queue distribuée sur laquelle le jeton circule d'un processus à un autre. Elle se présente sous forme d'une file d'attente composée de

requêtes (labélisées par les noms de leurs processus correspondants) non encore satisfaites.

Les requêtes sont servies selon l'ordre FIFO. Nous appelons cette queue, la file *Suivant*.

Soit $..p_j, p_{j+1}, ..$ un ensemble ordonné de processus dans cette file.

p_j pointe sur p_{j+1}, son prochain processus dans cette file. Ainsi, p_{j+1} est le *Suivant* à avoir demandé l'accès en **section critique**, et c'est à lui que revient le jeton après que p_j l'ait quitté.

Lorsque p_j désire entrer en **section critique**, il envoie une demande à son ***Parent***, p_i par exemple (Comme nous l'avons indiqué, ce processus est initialement la *racine* de l'arbre ***Parent***). En attendant de recevoir le jeton, p_j devient *racine* de l'arbre parental. À partir de là, deux cas se présentent :

1. Le ***Parent*** de p_j, (p_i) n'est pas *racine*. Dans ce cas, il transmet la requête de p_j à son propre ***Parent***. Ainsi, la requête se propage de ***Parent*** à ***Parent*** à travers l'arbre jusqu'à atteindre l'ancienne *racine*. Toute l'ascendance pointe sur p_j qui devient le nouveau ***Parent***.
2. Le ***Parent*** de p_j (p_i) est *racine* de l'arbre. Si p_i a déjà quitté la **section critique**, il transmet directement le jeton à p_j.

   Par contre, si p_i est en **section critique** ou est en attente de recevoir le jeton, il pointe son *Suivant* sur p_j dans la file *Suivant*. Notons qu'un processus a deux pointeurs, l'un sur son ***Parent*** dans la structure arborescente et l'autre sur son *Suivant* dans la file.

Chaque processus maintient des variables locales qu'il met à jour au fur et à mesure que l'algorithme évolue. Nous les définissons comme suit :

Token_present : Variable booléenne dont la valeur est *true* si le processus détient le jeton, *false* autrement.

Requesting_cs : Variable booléenne dont la valeur est *true* si le processus réclame la **section critique**.

Suivant : Le prochain processus à entrer en **section critique**. La variable *Suivant* est *nulle* au début de l'algorithme, et lorsque le processus est dernier dans la file.

Parent : Chaque processus a un ***Parent***, il peut être *null* s'il s'agit de la *racine*.

Dans l'algorithme Naimi-Tréhel, les processus envoient deux types de messages :

Request(j) : Ce message est envoyé par un processus donné à son ***Parent***, pour demander l'accès en **section critique**.

Token : Ce message est envoyé par un processus à son *Suivant* dans la file, pour lui passer le jeton.

La propriété suivante est un invariant :

Invariant 2. *Au terme de la demande d'entrée en* ***section critique****, la racine de l'arbre* ***Parent*** *devient la tête de la file Suivant.*

L'algorithme Naimi-Tréhel est fondé sur un modèle distribué qui garantit l'unicité du jeton et qui assure les propriétés de *sûreté* et de *vivacité* (algorithme 4.1).

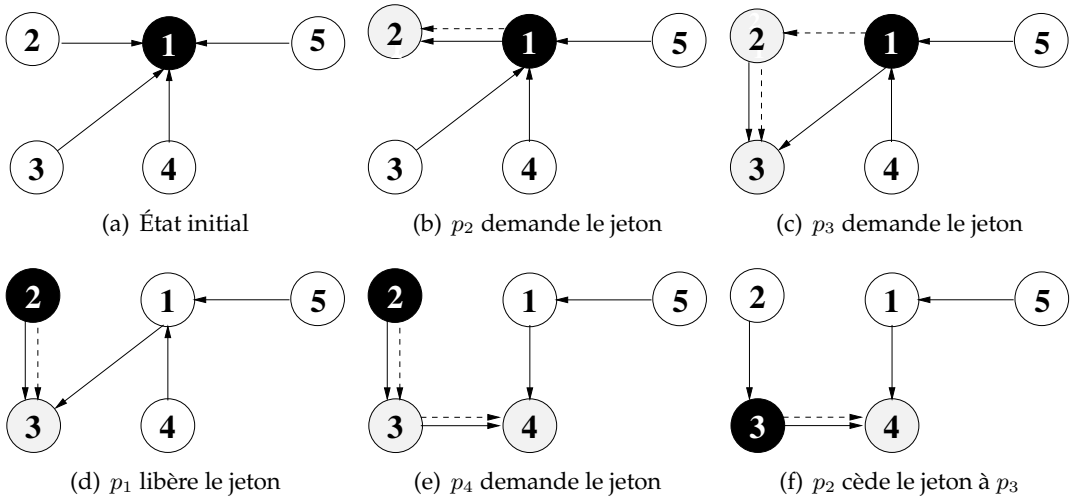
L'exemple d'exécution illustré dans la figure 4.1 résume le déroulement de l'algorithme, suite à l'émission de quelques requêtes par quelques processus. Les traits en pointillés désignent la file *Suivant*.

Initialement, p_1 détient le jeton (figure 4.1(a)). P_1 est donc *racine* de l'arbre et est ***Parent*** de tous les processus restants.

p_2 demande le jeton à son ***Parent*** (figure 4.1(b). En conséquence, p_1 pointe sur le nouveau ***Parent*** p_2, et positionne son *Suivant* sur le même processus.

À son tour, p_3 demande le jeton (figure 4.1(c)) à p_1. Par la suite, p_1 transmet la requête à son nouveau ***Parent*** p_2, qui à son tour met à jour ses deux variables, ***Parent*** et *Suivant*.

Dans la figure 4.1(d), p_1 quitte la **section critique** et transmet le jeton à p_2. P_4 demande à son tour le jeton (figure 4.1(e)). Ensuite, p_1 et p_3 pointent leur ***Parent*** sur p_4, et p_3 pointe son *Suivant* sur p_4. Finalement, p_3 obtient le jeton par p_2 (figure 4.1(f)).



(a) État initial (b) p_2 demande le jeton (c) p_3 demande le jeton

(d) p_1 libère le jeton (e) p_4 demande le jeton (f) p_2 cède le jeton à p_3

FIGURE 4.1 – Exemple d'exécution de l'algorithme Naimi-Trehel

4.1.2 Requêtes concurrentes

L'exemple ci-dessus expose l'exécution de l'algorithme suite à des demandes consécutives d'accès à la **section critique** (figure 4.1). Nous attirons l'attention sur le fait que dans l'algorithme de Naimi et Trehel, un ***Parent***, p_1 par exemple, de deux processus, peut recevoir deux requêtes simultanées de p_2 et p_3 par exemple. Dans ce cas, p_1 est confronté à deux situations :

Soit que la requête de p_2 est prise en charge en premier, d'où p_3 est mis en attente et est donc déconnecté temporairement de l'arbre parental, soit inversement.

Algorithme 4.1 Algorithme Naimi-Tréhel

Fonctions fondamentales de l'algorithme Naimi-Tréhel

Initialisation
1: **si** ($p_j \neq$ *racine*) **alors**
2: ***Parent*** $\leftarrow$ *racine* **Token_present** $\leftarrow$ *null*
3: **sinon**
4: ***Parent*** $\leftarrow$ *null*
5: **Token_present** $\leftarrow$ *true*
6: **fin si**
7: *Suivant* $\leftarrow$ *null*
8: ***Requesting_cs*** $\leftarrow$ *false*

Envoi d'une requête par p_j
9: ***Requesting_cs*** $\leftarrow$ *true*
10: **si** (***Parent*** $\neq$ *null*) **alors**
11: p_j envoie le message ($request[j]$) à son ***Parent***
12: ***Parent*** $\leftarrow$ *null*
13: p_j attend le jeton
14: **fin si**

Réception d'une requête, supposons par p_i le ***Parent*** de p_j
15: **si** (***Parent*** = *null*) **alors**
16: **si** (***Requesting_cs*** = *true*) **alors**
17: *Suivant* $\leftarrow p_j$
18: **sinon**
19: p_i envoie le message (token) à p_j
20: **fin si**
21: **sinon**
22: p_i envoie le message ($request[j]$) à son ***Parent***
23: **fin si**
24: ***Parent*** $\leftarrow p_j$

Libération de la **section critique** par p_j
25: ***Requesting_cs*** $\leftarrow$ *false*
26: **si** (*Suivant* $\neq$ *null*) **alors**
27: p_j envoie le message (token) à son *Suivant*
28: *Suivant* $\leftarrow$ *null*
29: **fin si**

La figure 4.2 expose un exemple de Naimi et Tréhel [NT87] dont le scénario est le suivant : Initialement, p_1 détient le jeton (figure 4.2(a)). P_3 réclame la **section critique** par l'envoi d'une requête à son ***Parent***. À son tour, p_1 pointe son ***Parent*** et son *Suivant* sur p_3 (figure 4.2(b)).

Par la suite, p_2 et p_5 réclament le jeton. Ils envoient tous les deux une requête à p_1 et se considèrent aussitôt *racine* de l'arbre, en pointant leur variable respective ***Parent*** sur *null*. P_1 transmet la requête de p_2 à p_3 et prend p_2 comme ***Parent*** (figure 4.2(c)). Pendant ce temps, p_5 est isolé de l'arbre.

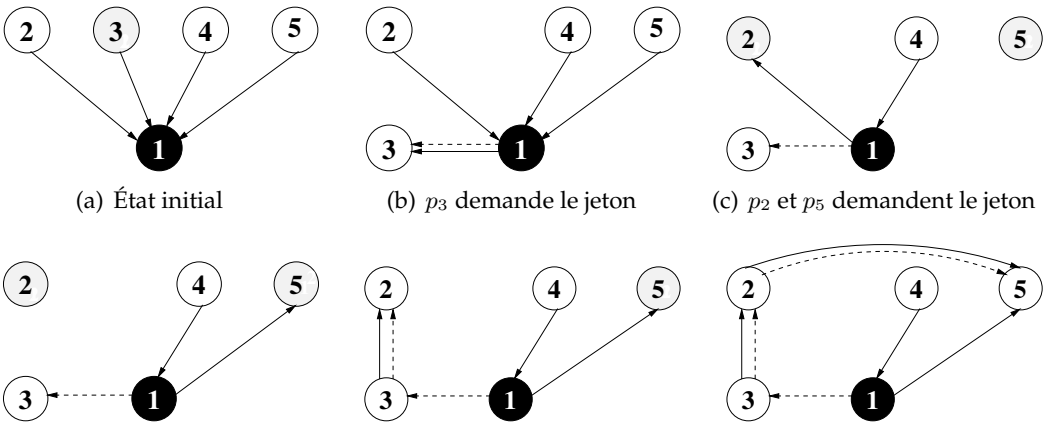
P_1 saisit la requête de p_5 dès qu'il a transmis celle de p_2 à p_3. Étant le nouveau ***Parent*** de p_1, p_2 reçoit la requête de p_5. Ce dernier devient désormais le nouveau ***Parent*** de p_1. À ce moment là, p_2 se détache à son tour de l'arbre (figure 4.2(d)). Dans les figures 4.2(e) et 4.2(f), les requêtes de p_2 et de p_5 sont respectivement accomplies.

L'algorithme stipule que les processus émetteurs deviennent *racine*. Leur variable ***Parent*** devient *null* dès lors que la requête est envoyée. Dans un système à n processus, *n-1* processus peuvent demander le jeton de manière concurrente. Cette situation génère un éclatement de l'arbre ***Parent*** par n processus disjoints.

L'adjonction d'un aspect dynamique supplémentaire à l'algorithme complexifie d'autant plus sa réalisation, pour le motif que l'algorithme que nous proposons comporte des processus volatiles.

En conséquence, l'isolement des processus, ne serait-ce que momentané est irrecevable dans un système où les processus disposent de l'éventualité de départ. De ce fait, dans le souci de maintenir une cohérence totale des deux structures, nous proposons une adaptation de l'algorithme Naimi-Tréhel de sorte qu'une requête au plus soit traitée par un processus donné (section 4.2).

Notons que cet exemple a été présenté dans [NT87] dans le contexte de défaillances des nœuds.



(a) État initial (b) p_3 demande le jeton (c) p_2 et p_5 demandent le jeton

(d) Les requêtes p_2 p_5 en cours de traitement (e) p_5 en transit (f) requêtes accomplies

FIGURE 4.2 – Exemple de requêtes concurrentes dans l'algorithme Naimi-Trehel [NT87]

4.2 Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs (ADEMLE)

Cette section expose notre première contribution à l'algorithme Naimi-Tréhel. Il s'agit d'avoir des processus susceptibles de joindre et de quitter le système, tout en garantissant les propriétés de *sûreté* et de *vivacité* de l'exclusion mutuelle. De prime abord, nous explicitons les faits **connexion** et **déconnexion** d'un processus du système.

Connexion d'un processus au système Pour faire partie du système distribué, il suffit aux processus de choisir un ***Parent*** parmi ceux existant dans l'arbre, puis, de l'affecter à leur variable associée. Cette opération le maintient à nouveau dans la structure parentale et cela suffit pour le reconnecter au système. Aussi, ce processus aura sa place dans la file *Suivant* dès l'envoi d'une requête.

Déconnexion d'un processus du système La déconnexion d'un processus est volontaire et réalisable sous conditions. Cette action suscite tout l'intérêt de notre algorithme car elle accentue la difficulté de sa mise en œuvre. Une bonne partie de notre travail théorique repose sur la potentialité de déconnexion des processus qui se présente doublement : Détachement de l'arbre ***Parent*** et suppression de la file *Suivant*. Les démarches que nous entreprenons s'articulent autour d'une structure modifiée, de variables ajoutées et de quelques nouveaux échanges dirigés entre processus.

4.2.1 Conditions nécessaires au maintien de la connectivité des deux structures

Pour commencer, nous imposons les règles suivantes :

Le système ne doit pas comporter des processus isolés. Ces derniers ne doivent en aucun cas se détacher de l'arbre ***Parent*** en dehors de l'opération de déconnexion. De ce fait, la connectivité doit être préservée en toutes circonstances.

Un processus donné p_j ne traite qu'une requête à la fois. Dans le cas où plusieurs requêtes sont réceptionnées, elles sont mises en attente, puis traitées une à une par p_j.

Contrairement à l'algorithme de base (algorithme 4.1, ligne 28), p_j maintient le lien avec son *Suivant* dans la file, bien qu'il ait obtenu le *lock*. Il peut toutefois se déplacer pour atteindre la tête de file, si et seulement s'il envoie une nouvelle requête, après avoir libéré l'ancien *lock*.

4.2.2 Operations atomiques et notion d'état

Les nouvelles exigences décrivent la conduite que doit adopter un processus donné p_j, face à la réception de requêtes concurrentes. Le chevauchement des opérations en provenance de processus disjoints est clairement écarté. En premier lieu, nous associons de nouvelles variables à chaque processus dans l'**ADEMLE** :

Send : Variable booléenne symbolisant l'état de p_j par rapport à une requête en voie d'émission. *Send* est *false* initialement. Elle devient *true* si p_j envoie une requête en cours de traitement.

Req : Liste des requêtes reçues par p_j, vide initialement.

Wait_ack : Variable booléenne faisant référence à l'accusé de réception (ACK) d'une requête envoyée par p_j. **Wait_ack** est initialement *false*.

***Précédent* :** Chaque processus connaît celui qui le précède dans la file *Suivant*. Une fois que p_j prend place dans cette file, il pointe sa variable *Précédent* sur le processus qui aura le jeton avant lui. *Précédent*,*Suivant* forment désormais une liste doublement chaînée.

***fils* :** Cette variable élargit la terminologie de notre algorithme étendu. Nous l'utilisons pour désigner les processus qui ont pour parent p_j.

En second lieu, nous instaurons la notion d'état comme suit :

Soit dans un système distribué, $\{p_0..p_j..p_n\}$ un ensemble fini de processus représentatifs d'une branche dans la structure parentale, auxquels nous attribuons des états, où p_0 est sa *racine*. Ces états sont abstraits. Ils n'ont pas pour rôle de renseigner sur la présence du jeton ou sur l'accès à la **section critique**, mais plutôt de déterminer le stade d'une requête vis à vis d'un processus p_j. Nous les définissons comme suit :

Idle* :** L'état *Idle* indique que p_j est au repos. À partir de là, il peut, soit envoyer une demande à son ***Parent pour l'accès à la **section critique**, pour basculer à l'état *Sending*, soit être occupé à recevoir des requêtes en provenance de ses processus *fils*. Notons que la liste **Req** est incontestablement *null* à cet état.

Sending* :** Cet état indique que p_j a envoyé une requête à son ***Parent et qu'il attend la fin de son acheminement dans l'arbre. La liste **Req** associée à p_j est *null*.

***Busy* :** p_j enregistre une requête, c'est à dire la prend en charge. Il n'est pas en mesure d'en enregistrer d'autres. Cependant, il peut les réceptionner en les affectant à la liste **Req**.

L'algorithme 4.2 précise le point de départ de notre système dans lequel tous les processus sont au repos. Notons qu'ils pointent tous sur la même *racine* (p_0). En d'autres termes, l'ensemble des initialisations des variables traduit l'état *Idle* de chaque processus.

Pour représenter ce point de départ, considérons une branche de l'arbre ***Parent*** composée d'une séquence de processus $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ où p_n est sa *racine*. Dans cette séquence, p_{j+1} est le ***Parent*** de p_j, tandis que p_j est le *fils* de p_{j-1}. Nous supposons qu'initialement, p_0 détient le jeton. P_0 est donc le ***Parent*** de tous les processus restants. Les sections ci-dessous décrivent le déroulement de l'**ADEMLE**.

4.2.3 Traitement d'une requête dans l'ADEMLE

Le terme "envoi d'une requête" est remplacé par "traitement d'une requête". En effet, l'envoi d'une requête prend désormais une définition plus large que celle dans

Algorithme 4.2 Initialisation (**ADEMLE**)

Initialisation des variables pour un processus donné p_j. P_0 est le cas particulier de la *racine*

1: **Send** $\leftarrow false$
2: **Req** $\leftarrow null$
3: **Wait_ack** $\leftarrow null$
4: *Suivant* $\leftarrow null$
5: *Précédent* $\leftarrow null$
6: ***Can_request*** $\leftarrow true$
7: **si** $(racine = p_0)$ **alors**
8: ***Parent*** $\leftarrow null$
Token_present $\leftarrow true$
9: **sinon**
10: ***Parent*** $\leftarrow p_0$
Token_present $\leftarrow false$
11: **fin si**

l'algorithme originel. Pour cause, son parcours est plus significatif. Nous le décrivons ci-dessous :

À l'instar de l'algorithme de base, p_j demande le jeton à son ***Parent*** p_{j+1}, à un instant t. La variable **Send** est positionnée à *true* préalablement à l'envoi d'une requête. Ceci traduit le passage de p_j de l'état *Idle* à l'état *Sending*.

p_{j+1} réceptionne la requête de p_j et l'insère dans la liste **Req**, s'il a lui même envoyé une requête non encore accomplie. La requête subit le même sort dans le cas où p_{j+1} est occupé par une autre requête. Autrement dit, p_{j+1} ne doit en aucun cas être *Sending* ni *Busy* pour enregistrer et traiter une requête.

Lorsque p_{j+1} est disponible, c'est-à-dire, lorsqu'il devient *Idle*, il prend en charge la requête de p_j qui sera forcément en tête de la liste **Req**. Ensuite, si le ***Parent*** de p_{j+1} (p_{j+2}) existe, p_{j+1} le transmet à p_j et attend de recevoir un `ACK` de la part de ce dernier. La requête de p_j est ainsi enregistrée.

Une fois informé du ***Parent*** de p_{j+1}, p_j réenvoie sa demande à p_{j+2}. De la même manière, si p_{j+2} est *Idle*, alors il envoie son propre ***Parent*** à p_j, et attend de recevoir un `ACK`. Autrement, la requête de p_j est mise en instance et est insérée dans la liste **Req** de p_{j+2}.

Le processus se poursuit le long des ascendants de la branche $\{..p_j, p_{j+1}..p_n\}$ jusqu'à ce que p_n soit atteint. À ce stade, p_n envoie un message à p_j l'informant qu'il est *racine* de l'arbre. Aussi, p_n pointe sa variable *Suivant* sur p_j.

Nous pouvons dire que notre algorithme diffère de celui de Naimi et Tréhel par le fait que p_j suit sa propre requête, pas à pas à travers la branche de ses ascendants dans l'arbre.

À son tour, p_j enregistre p_n en tant que *Précédent*. De cette façon, un nouveau lien se crée au bout de la liste chainée (*Précédent* =p_n,*Suivant* =p_j).

Les processus $p_{j+1}..p_n$ positionnent leur nouveau ***Parent*** sur p_j.

4.2.3.1 Requêtes concurrentes

Etudions à présent le déroulement de l'**AMDELE** suite à des requêtes concurrentes. Admettons que p_{j+1} est ***Parent*** d'un autre *fils* p_i à partir duquel dérive une autre branche. Prenons l'exemple de la figure 4.2 où p_1 est le ***Parent*** à la fois de p_2 et de p_5.

Supposons que p_1 reçoit, à quelques instants près, une demande de p_5 alors que celle de p_2 est en voie d'acheminement dans l'arbre. Sachant qu'il n'a pas encore reçu d'`ACK` en provenance de p_2, p_1 insère la requête de p_5 dans la liste **Req**. Pendant ce temps, contrairement à l'algorithme de base (ligne 12 de l'algorithme 4.1), p_5 garde toujours son ancien ***Parent***. Par conséquent, son lien avec l'arbre ***Parent*** subsiste tant que sa propre requête n'est pas accomplie.

La fin de traitement d'une requête se concrétise par l'envoi des `ACKs` à son ***Parent*** p_{j+1} et à tous ses ascendants (branche $\{p_{j+2}, ..., p_n\}$) dans l'arbre. Ce n'est qu'à ce moment là que les processus de la même branche peuvent éventuellement passer aux requêtes suivantes, qui occupent la tête de leurs listes **Req** correspondantes. Ainsi, les requêtes sont servies selon l'ordre FIFO.

Finalement, p_j positionne son ***Parent*** à *null*, sa variable **Send** à *false* et rejoint l'état *Idle*.

La succession des événements montre clairement que les requêtes ne sont pas piétinées. Aussi, l'algorithme **AMDELE** conserve l'ordre logique des requêtes émises par tout processus p_j.

4.2.3.2 Complexité des messages envoyés pour l'envoi d'une requête

Dans l'algorithme de base, le nombre moyen de messages nécessaires à l'acheminement d'une requête, dépend de sa distance avec la *racine* de l'arbre ***Parent***. Dans [NTA96a], les auteurs ont démontré que la complexité est de l'ordre de *O(log(n))* , où n est le nombre total de processus dans le système.

Nous allons nous conformer à ce résultat pour démontrer que la complexité de notre algorithme est aussi logarithmique.

Soit une branche $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ dans l'arbre ***Parent***.

Dans l'algorithme originel, le message envoyé par p_j à son ***Parent*** p_{j+1} pour la demande d'accès en **section critique**, déclenche l'envoi d'autres messages consécutifs, le long de ses ascendants.

D'après les spécifications de notre algorithme (section 4.2.2), nous pouvons aisément percevoir les messages additionnels :

Chaque ascendant dans la branche $p_{j+1}..p_n$ envoie un message à p_j incluant son propre ***Parent***. Contrairement à l'algorithme originel où les messages sont transmis d'un processus à son ascendant, p_j envoie un message à chaque ascendant. À la fin de

l'acheminement de la requête, p_j envoie les accusés de réception (`ACKs`) à son ***Parent*** et à tous ses ascendants.

De cette manière, le surcoût engendré par l'algorithme **AMDELE** n'implique pas l'ensemble des processus du système, mais uniquement la séquence qui compose la branche des ascendants du demandeur de jeton.

Tout compte fait, notre algorithme rajoute deux messages par requête. Nous pouvons conclure que la complexité de l'**AMDELE** est *O(log(n))* , donc équivalente à celle de l'algorithme Naimi-Tréhel.

Lemme 1. *L'arbre **Parent** est constamment connecté.*

Démonstration. Nous avons montré que dans l'algorithme de base, les processus sont dans certains cas, déconnectés momentanément de l'arbre ***Parent*** durant la transmission de leurs requêtes, entre le point de départ et la *racine* (section 4.1.1).

Inversement, l'**AMDELE** stipule que p_j garde son ***Parent*** en cours (p_{j+1}) pendant le transit de sa requête. Parallèlement, p_{j+1} et ses ascendants ($p_{j+2}..p_n$) pointent leur variable ***Parent*** sur p_j. Toutefois, p_j n'annule son ***Parent*** (*null*) qu'après avoir envoyé les `ACKs`, donc bien après que ses ascendants aient mis à jour leur nouveau ***Parent*** (ligne 30 de l'algorithme 4.3).

Donc, p_j ainsi que tout processus demandeur de jeton, dont la requête est en transit, n'est jamais détaché de son ***Parent*** . Par conséquent, l'arbre ***Parent*** est constamment connecté. □

Lemme 2. *Les variables Suivant et Précédent forment assidûment une liste doublement chaînée.*

Démonstration. Initialement, p_0 détient le jeton et la liste chaînée est vide (*Suivant =null*, *Précédent =null*).

Si p_j demande un jeton retenu par un autre processus, la requête se propage à travers l'arbre ***Parent*** (de p_{j+1} à p_n) qui est constamment connecté (lemme 1). À ce niveau, la liste chaînée (*Suivant*, *Précédent*) est actualisée par un nouveau lien qui s'établit entre p_j et p_n (algorithme 4.3, ligne 22 et algorithme 4.4, ligne 9).

De ce fait, la suite (*Suivant*, *Précédent*) forme continuellement une liste doublement chainée. □

4.2.3.3 Notation formelle des opérations atomiques

Cette section décrit formellement les séquences d'événements qui peuvent survenir localement chez un processus donné, à un instant t, $p_j(t)$.

Notre système n'utilise aucune horloge globale. Cependant nous allons recourir à une modélisation à temps discret, afin de faire valoir l'atomicité des différentes opérations pour un processus donné dans l'**AMDELE**.

Soit T=$\{t_0, t_1, ..t_k, ..t_n\}$ une séquence de temps discret, tel que $t_0 < t_1 < ..t_k.. < t_n$ relative à la réception des requêtes par p_j.

Soit T'=$\{t'_0, t'_1, ..t'_i, ..t'_n\}$ un ensemble de temps discret, relatif à la réception des accusés de réception (`ACKs`) pour les requêtes enregistrées. Rappelons que p_j n'enregistre une nouvelle requête qu'après avoir reçu l'`ACK` de celle traitée précédemment.

Nous pouvons déduire que $t_0 < t'_0 < t_1 < t'_1 < ..t_k < t'_k.. < t_n < t'_n$.

Soit $State(p_j(t))$ l'état de p_j à un instant t.

$State(p_j(t))$ est *Idle* s'il se trouve sans activité. En d'autres termes, p_j n'envoie pas une demande de jeton, ne reçoit pas de requête et n'attend pas un `ACK` d'un autre processus.

Donc, pour chaque instant *t* dans l'intervalle $[t'_j, t_{j+1}]$:

$$\text{STATE}(p_j(t)) = \textit{Idle} \Rightarrow \begin{cases} \text{SEND} = \textit{False} \\ \text{REQ} = Null \\ \text{WAIT\_ACK} = \textit{False} \end{cases}$$

Si $p_j(t)$ attend un `ACK` du processus dont la requête vient d'être enregistrée, il devient *Busy*. Pour chaque *t* dans $[t_j, t'_j]$, $State(p_j) = \textit{Busy}$:

$$\text{STATE}(p_j(t)) = \textit{Busy} \Rightarrow \begin{cases} \text{SEND} = \textit{False} \\ \text{REQ} \neq Null \\ \text{WAIT\_ACK} = \textit{True} \end{cases}$$

si $p_j(t)$ envoie une requête à son ***Parent***, ou continue d'envoyer les `ACKs` à ses ascendants, alors il prend l'état *Sending*.

$$\text{STATE}(p_j(t)) = \textit{Sending} \Rightarrow \begin{cases} \text{SEND} = \textit{true} \\ \text{REQ} = Null \\ \text{WAIT\_ACK} = \textit{False} \end{cases}$$

L'invariant suivant traduit les nouvelles règles concernant le traitement des requêtes dans l'**ADEMLE**

Invariant 3. *P_j doit être dans l'état* Idle *avant l'envoi ou l'enregistrement d'une requête. C'est la condition pour laquelle l'atomicité des opérations est garantie. En d'autres termes :*

Pour $t_1 < t_2$, si STATE*($p_j(t_1)$) est* Sending *et* STATE*($p_j(t_2)$) est* Busy *alors il existe un temps intermédiaire t' avec $t_1 < t' < t_2$ tel que* STATE*($p_j(t')$)*=Idle*.*

4.2.3.4 Conditions préliminaires à l'envoi d'une requête

La section 4.2.3 résume le déroulment d'une requête. En réalité, d'autres conditions, relatives au voisinage d'un processus (dans les deux structures), doivent être satisfaites et vérifiées avant qu'une demande d'accès en **section critique** ne soit effectivement envoyée.

Littéralement, p_j assujettit un ensemble de processus avant et pendant l'acheminement de sa requête.

p_j peut d'une part être bloqué par un autre processus, occupé lui aussi à envoyer sa propre requête. Le procédé de blocage est décrit un peu plus bas (requirements 3

et 4). Nous introduisons une variable booléenne ***Can_request***, initialement *true*, que p_j consulte en premier. Ainsi, l'envoi d'une requête est subordonné à la valeur associée à cette variable.

D'autre part, dans le but de préserver la connectivité de la liste chainée (*Suivant*, *Précédent*) et de l'arbre ***Parent***, le processus qui tente de récupérer le jeton s'acquièrt les états de ses voisins dans la liste chaînée, à savoir le *Suivant* et le *Précédent*. Il s'acquièrt également l'état de son ***Parent*** et ceux de ses ascendants.

Voisins de p_j Un processus est dit voisin de p_j s'il ne peut envoyer une requête en même temps que lui.

Tout compte fait, l'ordre chronologique des opérations qu'un processus doit parfaire en amont à l'envoi d'une requête est le suivant :

1. Premièrement, p_j consulte sa variable ***Can_request***. Si elle est *false*, p_j diffère son envoi.
2. P_j patiente encore s'il est *Busy*, de même si sa propre liste de requêtes réceptionnées **Req** n'est pas totalement vide. En somme, p_j devra attendre de rejoindre l'état *Idle* (invariant 3).
3. P_j ne peut envoyer une demande tant que son *Suivant* est dans l'état *Sending*. Quand ce dernier aura accompli sa demande, il se déplacera pour occuper la tête de la liste chaînée (*Suivant*, *Précédent*) (étape 4.2.3 de la section 4.2.3). À ce moment là, p_j est invité à mettre à jour son *Suivant*. Sitôt cela fait, il positionne la variable ***Can_request*** de ce dernier à *false* afin d'inhiber toute action de sa part.
4. De la même manière, la valeur correspondante à la variable ***Can_request*** du processus *Précédent* dans la liste chaînée est mise à *false*.
5. Le ***Parent*** de p_j et sa branche d'ancêtres sont astreints à ne pas réclamer l'entrée en **section critique**. Ils subissent le même sort que le *Suivant*, par l'affectation de la valeur *false* à leur variable associée ***Can_request***. Notons qu'à la fin de la demande, p_j se détache de quelques liens mais en crée de nouveaux.
6. À partir de là, p_j bascule à l'état *Sending*.

Par ces actions qui s'ajoutent à notre **AMDELE**, nous assurons l'atomicité des opérations d'envoi et de réception des requêtes pour un processus donné p_j.

Dans la figure 4.3, p_5 tente d'envoyer une demande à son ***Parent***, p_6. Les lignes rouges en pointillés dénotent la liste chaînée (*Suivant*, *Précédent*). Les cercles rouges représentent les processus bloqués par p_5 (ses voisins). Il s'agit des processus *Précédent* (p_3), *Suivant* (p_6) et de la branche parentale (p_8, p_1 et p_9). Ces processus s'immobilisent dans les deux structures (arbre ***Parent*** et liste chaînée (*Suivant*, *Précédent*)) le temps que la requête de p_5 s'achève.

Les lignes en bleu désignent les nouveaux liens établis après que p_5 ait fini d'envoyer sa requête. Au terme de cet envoi, p_6, p_8, p_1 et p_9 repositionnent leur ***Parent*** sur p_5. P_3 et p_6 établissent une liaison dans la liste chaînée (*Suivant*, *Précédent*), quant à p_5, il se positionne en tête de cette même liste.

Algorithme 4.3 Envoi d'une requête d'accès exclusif (**ADEMLE**)

Étapes préliminaires à l'envoi d'une requête. P_j bloque son voisinage

si (***Can_request*** $= false \vee$ **Req** $\neq null$) **alors**
  p_j attend
fin si
Can_request$(Suivant) \leftarrow false$
Can_request$(Précédent) \leftarrow false$
$k \leftarrow 1$
tant que $p_{j+k} \neq racine$ **faire**
  si ($Etat(p_{j+k}) =$ *Sending*) **alors**
    p_j attend
  fin si
  Can_request$(p_{j+k}) \leftarrow false$
  p_j demande le ***Parent*** de p_{j+k}
  $k++$
fin tant que

P_j change d'état et procède à l'envoi d'une demande d'accès exclusif.

P_j attend de recevoir le ***Parent*** correspondant à chaque ascendant de la branche dont il fait partie.

$Etat \leftarrow$ *Sending*
$k \leftarrow 1$
tant que ($p_{j+k} \neq racine$) **faire**
  p_j Envoi un message à p_{j+k} et attend la réponse
  si (réponse $\neq null$) **alors**
    $k++$
  sinon
    Précédent $\leftarrow p_{j+k}$
  fin si
fin tant que

P_j envoie les `ACKs` à ses ascendants, débloque son voisinage et redevient *Idle*

$k \leftarrow 1$
tant que ($p_{j+k} \neq racine$) **faire**
  p_j envoie un message (ACK) à p_{j+k}
  Can_request$(p_{j+k}) \leftarrow true$
fin tant que
Parent $\leftarrow null$
Can_request$(NEXT) \leftarrow true$
Can_request$(PREV) \leftarrow true$
$Etat(p_j) \leftarrow$ *Idle*

Algorithme 4.4 Réception d'une requête (**ADEMLE**)

Réception d'une requête. P_j traite les les requêtes reçues, en l'occurrence celle p_i (l'un de ses *fils*)

1: **si** (**Wait_ack** =true) **alors**
2: p_j insère la requête de p_i dans la liste **Req**
3: **sinon**
4: **tant que** (**Req** $\neq null$) **faire**
5: $Etat(p_j) \leftarrow$ *Busy*
6: **si** (***Parent*** $\neq null$) **alors**
7: p_j envoie son ***Parent*** (p_{j+1}) à p_i
8: **sinon**
9: *Suivant*$(p_j) \leftarrow p_i$
10: **fin si**
11: ***Parent*** $\leftarrow p_i$
12: p_j attend ack(p_i)
13: $Etat(p_j) \leftarrow$ *Idle*
14: **fin tant que**
15: **fin si**

Lemme 3. *Le Suivant et le Précédent n'envoient jamais une demande d'accès en* ***section critique*** *au même moment que leur voisin,* p_j*.*

$$\left.\begin{array}{r} \textsc{State}(p_j(t_1)) = \text{Sending} \\ \wedge \\ \textsc{State}(\textit{Suivant}(p_j)(t_2)) = \text{Sending} \end{array}\right\} \Rightarrow t_1 \neq t_2.$$

Démonstration. Nous procédons par le raisonnement par l'absurde. Supposons que t_1 = t_2. Cela veut dire que les variables ***Can_request*** associées à p_j et à son *Suivant* sont *true*. Donc, conformément à l'ordre chronologique des actions préalables à l'envoi d'une requête (section 4.2.3.4,action 3), p_j et son *Suivant* se bloquent mutuellement. Or, un seul événement peut se produire à la fois. Soit que le *Suivant* bloque son *Précédent* (dans ce cas p_j) en première instance, ou inversement.

La paire p_j, *Précédent* suit le même raisonnement. Par conséquent, nous pouvons affirmer qu'à un instant t, p_j est le seul à acquérir l'état *Sending*. □

4.3 Traitement de la déconnexion d'un processus dans l'ADEMLE

Nous allons décrire à présent la stratégie de déconnexion effective d'un processus donné du système distribué. En réalité, p_j ne se démet pas de l'arbre ***Parent*** ni de la chaîne (*Suivant*, *Précédent*) de manière subite. Il devra plutôt satisfaire certaines contraintes. La disparition d'un quelconque processus ne doit en aucun cas compro-

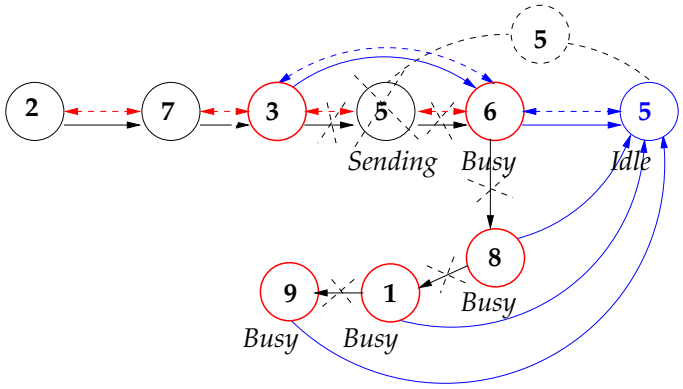


FIGURE 4.3 – Exemple d'envoi d'une requête dans l'**AMDELE**

mettre ni la cohérence de l'algorithme, ni la connectivité de l'arbre ***Parent***, ni celle de la liste chaînée (*Suivant*, *Précédent*).

La stratégie de départ partage néanmoins certaines opérations avec celles évoquées dans l'envoi d'une requête. En effet, dans les deux cas, p_j perd ses positions dans les deux structutres citées plus haut.

Nous poursuivons l'élaboration de notre **ADEMLE** qui est enrichi par de nouvelles variables et opérations.

Dans le souci continuel de maintenir la connectivité de l'arbre parental et celle de la liste chaînée, les variables suivantes sont ajoutées :

fils$[1..m]$ **:** Un tableau alloué dynamiquement pour contenir les processus *fils*.

Exit **:** Une variable booléenne initialisée à *false*, remise à *true* dès que p_j quitte le systeme.

Exiting **:** Une variable booléenne, initialisée à *false*, positionnée temporairement sur la valeur *true*, le temps que p_j négocie son depart du système.

L'activité de déconnexion requiert un nouvel état que nous associons à ceux précédemment énoncés (section 4.2.3.3) :

$$\text{STATE}(p_j(t)) = \textit{Exiting} \Rightarrow \begin{cases} \text{SEND} = \textit{false} \\ \text{REQ} = Null \\ \text{WAIT\_ACK} = \textit{False} \\ \textbf{del} = \textit{True} \\ \textbf{\textit{Exiting}} = \textit{True} \end{cases}$$

L'état *Exiting* stipule que p_j se prépare à quitter le système, ou plutôt négocie son départ avec ses voisins. En fait, cela dénote clairement que notre système autorise les processus à s'engager dans une seule opération à la fois. Donc, p_j peut soit envoyer, soit recevoir une requête ou planifier son départ.

Aussi, il est intelligible de constater que les variables ***Exit*** et ***Exiting*** doivent être *false* lorsque p_j est *Sending*, *Busy* ou *Idle*.

Les prédispositions que p_j devra adopter pour son départ s'apparentent à son état et aux états de ses voisins.

Tel que nous l'avons précisé, les prérequis de l'état *Exiting* correspondent à l'état *Sending* sur certains points.

Avant de continuer sur les faits, ouvrons une parenthèse sur les éléments qui composent les états de notre système. Par exemple, l'état *Busy* s'exprime dorénavant comme suit :

$$\text{STATE}(p_j(t)) = \textit{Busy} \Rightarrow \begin{cases} \text{SEND} = \textit{False} \\ \text{REQ} \neq Null \\ \text{WAIT\_ACK} = \textit{True} \\ \textbf{del} = \textit{False} \\ \textbf{\textit{Exiting}} = \textit{False} \end{cases}$$

Remarquons que désormais les deux variables ***Exit*** et ***Exiting*** font partie de chaque état.

Invariant 4. *p_j doit être* Idle *avant de regagner l'état* Exiting*. En d'autres termes :*

Pour $t_1 < t_2$, si STATE*($p_j(t_1)$) est* Exiting *et* STATE*($p_j(t_2)$) est* Sending *ou* Busy*, alors il existe un temps intermédiaire t' tel que :*

$t_1 < t' < t_2$ tel que STATE*($p_j(t')$)=Idle.*

Pour ainsi dire, entre deux activités, p_j doit obligatoirement reconquérir l'état *Idle*.

Revenons aux actions qui précèdent le départ de p_j. Nous introduisons une nouvelle variable conditionnelle à notre algorithme, ***Can_exit*** initialement *true*. Tout comme la variable ***Can_request***, ***Can_exit*** servira à p_j pour canaliser ses voisins, s'enquérir de leurs activités ; cela pour préparer un départ du système ou une nouvelle demande d'accès en **section critique**.

Sous un autre angle, p_j peut être immobilisé momentanément par son voisinage, à l'aide de l'une ou des deux variables en question, pour différer l'une des activités enoncées ci-dessus.

Les actions que p_j entreprend avant à son départ sont les suivantes :

4.3.1 Contraintes de déconnexion d'un processus

Pour qu'un processus p_j, se démette du système, il doit accomplir l'ensemble des opérations décrites ci-dessous :

- P_j consulte sa variable ***Can_exit*** et remet son départ à plus tard si celle-ci est *false*. Vu le caractère inhérent des deux activités et conjointement à la variable ***Can_request***, cette contrainte est associée aux actions préliminaires à l'envoi d'une requête (section 4.2.3.4).
- Tout comme la demande d'accès en **section critique**, p_j attend d'être *Idle* (section 4.2.3.4), et de finir avec de possibles requêtes contenues dans sa liste **Req**, qui le maintiendraient dans l'état *Busy*.

- P_j exécute les mêmes prérequis que dans la section 4.2.3.4 (condition 3). Pour retarder tout mouvement dans la liste chaînée, p_j immobilise momentanément son *Suivant* et son *Précédent* par les variables ***Can_request*** et ***Can_exit***.
- P_j consulte l'activité en cours de ses *fils*. Il diffère son départ si l'un d'entre eux se trouve *Sending*, *Busy* ou *Exiting*. Il patiente jusqu'à recevoir un message de leur part.

Notons que contrairement aux autres processus du voisinage, p_j ne bloque pas ses *fils* par les variables ***Can_request*** ou ***Can_exit*** lorsqu'il s'apprête à entamer une action car p_j perd son rôle de ***Parent*** pour ces *fils*.

Une fois ces conditions vérifiées, p_j bascule à l'état *Exiting* par le basculement des variables ***Exit*** et ***Exiting*** à la valeur *true*.

Invariant 5. *p_j quitte le systeme $\Rightarrow$ p_j met à jour les variables* **Exit** *et* **Exiting** *à* true *ssi*

$$\begin{cases} \textbf{\textit{Req}} = Null \\ \text{STATE}(p_j = \text{Idle}) \\ \text{STATE}(\textit{Suivant}(p_j), \textit{Précédent}(p_j), \textit{fils}[1..m](p_j) = \text{Idle}) \end{cases}$$

4.3.2 Modalités de déconnexion d'un processus

Les instructions suivantes sont exécutées chronologiquement par p_j. Elles identifient son départ effectif (algorithme 4.5) :

1. Tout d'abord, si p_j détient le jeton, il le passe immédiatement à son *Suivant*, si ce dernier existe. Autrement, si p_j se trouve en tête de la liste chaînée, il cédera le jeton à son *Précédent*.
2. P_j notifie son *Suivant* et son *Précédent* afin de les allier directement dans la liste chaînée (algorithme 4.5, ligne 29 et 30).
3. Si p_j est *racine* de l'arbre ***Parent***, alors il doit élire l'un de ses *fils* pour occuper ce rôle.
4. P_j transmet soit sa nouvelle *racine* élue, soit son ***Parent*** à ses *fils*.
5. Tous les *fils* de p_j mettent à jour leur nouveau ***Parent***. De cette façon, les processus *fils* maintiennent leur connexion à l'arbre ***Parent***.
6. À ce stade, p_j rétablit les valeurs initiales correspondantes aux variables ***Can_exit*** et ***Can_request*** de ses voisins.
7. P_j remet également sa propre variable ***Exiting*** à *false* et informe son ***Parent*** qu'il a achevé son départ.

Dans l'**ADEMLE**, la déconnexion d'un processus provoque quelques transformations dans l'arbre, sans que le déroulement des demandes d'accès en **section critique** n'en soit atteint.

Algorithme 4.5 Fonction EXIT (**ADEMLE**)

Fonction EXIT. P_j quitte le système

1: **si** (***Can_exit*** $= false \vee$ **Req** $\neq null$) **alors**
2: p_j attend
3: **fin si**
4: $(Etat(p_j) \leftarrow$ *Idle*
5: $($***Can_request*** $\wedge$ ***Can_exit***$)($*Suivant*$) \leftarrow false$
6: $($***Can_request*** $\wedge$ ***Can_exit***$)($*Précédent*$) \leftarrow false$
7: $k \leftarrow 1$
8: **tant que** $p_{j+k} \neq racine$ **faire**
9: **si** $(Etat(p_{j+k}) =$ (*Sending* $\vee$ *Exiting*) **alors**
10: p_j attend
11: **fin si**
12: $($***Can_request*** $\wedge$ ***Can_exit***$)(p_{j+k}) \leftarrow false$
13: $k++$
14: **fin tant que**
15: **pour tout** $i \in$ *fils*$[1..m]$ **faire**
16: **si** $(Etat(i) =$ *Exiting*) **alors**
17: p_j attend
18: **fin si**
19: **fin pour**

20: ***Exit*** $\leftarrow true$ /* départ effectif */
21: ***Exiting*** $\leftarrow true$
22: **si** (**Token_present** $= true$) **alors**
23: **si** (*Suivant* $\neq true$) **alors**
24: p_j envoie le jeton au *Suivant*
25: **fin si**
26: **sinon**
27: p_j envoie le jeton au *Précédent*
28: **fin si**
29: p_j envoie *Suivant* au *Précédent*
30: p_j envoie *Précédent* au *Suivant*
31: **si** $(p_j = racine)$ **alors**
32: $racine \leftarrow$ *fils*
33: **fin si**
34: **pour tout** $i \in$ *fils*$[1..m]$ **faire**
35: ***Parent***$(i) \leftarrow racine$ /* mise à jour de la nouvelle racine */
36: **fin pour**

37: $($***Can_request*** $\wedge$ ***Can_exit***$)($*Suivant*$) \leftarrow true$
38: $($***Can_request*** $\wedge$ ***Can_exit***$)($*Précédent*$) \leftarrow true$
39: $k \leftarrow 1$
40: **tant que** $p_{j+k} \neq racine$ **faire**
41: $($***Can_request*** $\wedge$ ***Can_exit***$)(p_{j+k}) \leftarrow true$
42: $k++$
43: **fin tant que**
44: ***Exiting*** $\leftarrow false$

4.3.3 Envoi d'un *Parent* sortant à un demandeur de jeton

Nous avons montré que grâce aux deux variables ***Can_request*** et ***Can_exit***, p_j peut maîtriser le mouvement de ses voisins, pour éviter qu'ils ne regagnent l'état *Sending* ou *Exiting*. Ce blocage temporaire permet à p_j de redessiner les liens nécessaires relatifs aux deux structures.

Toutefois, l'état *Busy* ne peut être dominé. En effet, p_j n'est pas en mesure de contrôler la réception des requêtes à destination de ses voisins. En fait, le problème provient des *fils* de p_j. Afin de mieux expliquer cette situation, prenons le scénario suivant :

Soit $p_0, ...p_{j-1}, p_j, p_{j+1}...p_n$ une branche de l'arbre ***Parent***, tel que p_j est le ***Parent*** de p_{j-1}. Supposons que p_{j-1} envoie une requête à p_j, qui est disponible. Pendant ce temps, p_{j+1} occupe l'état *Exiting*. Donc, p_j enregistre la requête de p_{j-1} et envoie son ***Parent*** à p_{j-1} (section 4.2.3). Cette situation présente une irrégularité dans la gestion de l'arbre parental. Il est évident que p_j ne devrait pas transmettre un ***Parent*** (p_{j+1}) en voie de disparition à p_{j-1}.

Pour remédier à ce problème, nous revenons à l'algorithme pour y apporter une légère rectification. Ainsi, plutôt que d'envoyer immédiatement p_{j+1} à p_{j-1}, p_j va demander la valeur correspondante à la variable ***Exiting*** de p_{j+1}. Si elle s'avère *true*, p_j attend qu'elle redevienne *false*. Ce basculement indique que les évènements suivants se sont produits :

Premièrement p_{j+1} a quitté le système, et deuxièmement p_j a éventuellement reçu un nouveau ***Parent***. Le cas échéant, c'est-à-dire si p_{j+1} est *racine*, p_j aura reçu le ***Parent*** élu parmi les *fils* de p_{j+1}.

Quoiqu'il en soit, p_{j-1} reçoit un ***Parent*** disponible. Il peut par la suite prendre les dispositions nécessaires à l'acheminement de sa requête en toute cohésion.

Les lignes ci-dessous sont donc ajoutées à la ligne 6 de l'algorithme 4.4 :

Algorithme 4.6

P_j met à jour un nouveau ***Parent*** avant de l'envoyer à p_i
1: p_j demande, puis reçoit ***Exiting***(***Parent***)
2: **tant que** ***Exiting*** $= true$ **faire**
3: p_j attend
4: **fin tant que**
5: p_j met à jour son nouveau ***Parent***

4.3.4 Complexité de départ d'un processus dans l'ADEMLE

La complexité moyenne d'envoi de messages par p_j lorsque ce dernier veut quitter le système est clairement de $0m$, où m est le nombre de *fils* directs.

Dans des cas extrêmes, c'est-à-dire dans le cas où les *fils* de p_j regroupent tous les processus du système, la complexité devient $0(n)$.

Il existe une alternative qui permet d'éviter une telle dégénérescence. En fait, la structure de l'arbre peut être modifiée dynamiquement, sous condition de préserver ses propriétés principales. Ceci est réalisable en utilisant les techniques d'équilibrage distribuées[NSSW87, BGM95]. La propriété structurale majeure qui devra être maintenue pour préserver l'exactitude de l'algorithme est relative à la *racine*, qui doit incontestablement représenter le dernier processus qui a demandé le jeton.

L'équilibre fourni par une telle structure assure une complexité maximale de messages de l'ordre de $O(\ logn)$, et ce, pour toutes les opérations relatives à un processus donné.

Nous pouvons également imposer un nombre borné de *fils* directs pour chaque processus. Par ce fait, la complexite globale des messages pour une opération de déconnexion rebascule à une échelle logarithmique de l'ordre de $O(\log n)$ pour toutes les opérations.

4.4 Preuve de l'ADEMLE

Cette section présente la preuve de notre algorithme, dont la validité résulte des propriétés de *sûreté* et de *vivacité* d'exclusion mutuelle.

4.4.1 Propriété de *vivacité*

Théorème 1. *(Vivacité) Si un processus p_j réclame la **section critique**, il l'obtient au bout d'un temps fini quelles que soient les circonstances.*

4.4.1.1 Plan de preuve

Pour démontrer la propriété de *vivacité*, nous procédons en deux étapes : D'abord, nous montrons que l'arbre ***Parent*** et la liste doublement chaînée (*Suivant*, *Précédent*) ne sont jamais déconnectés (lemme 4). Puis, nous démontrons que les requêtes sont servies dans le même ordre que celui où elles ont été insérées, bien que le processus qui les reçoit quitte le système (lemme 5).

Lemme 4. *L'arbre **Parent** et la liste chaînée (Suivant, Précédent) ne sont jamais déconnectés quelles que soient les circonstances.*

Démonstration. Il suffit de prouver que les lemmes 1 et 2 sont maintenus dans le cas de déconnexion de processus du système.

p_j est coupé de l'arbre s'il est séparé de sont ***Parent*** (p_{j+1}). Supposons qu'à l'instant t, p_{j+1} décide de quitter le système. Tant que ***Exiting*** est *true*, p_{j+1} prépare son départ.

En attendant, p_j garde p_{j+1} comme ***Parent*** jusqu'à ce qu'il en reçoive un autre, ou devienne lui même *racine* par élection. Donc, en aucun cas, les processus ne lâchent leur ancien ***Parent*** avant d'en avoir un nouveau. Par conséquent, l'arbre ***Parent*** n'est jamais déconnecté.

Par ailleurs, suivant le même raisonnement, p_j rétablit la liaison entre le *Suivant* et le *Précédent* avant de disparaitre. Donc, la liste chaînée n'est jamais déconnectée. □

Lemme 5. *Les requêtes sont servies dans le même ordre que celui où elles ont été insérées, bien que le processus qui les recoit quitte le système.*

Démonstration. Pour ce lemme, nous procédons par le raisonnement par l'absurde :

Soit p_{j+1} le ***Parent*** de p_j qui reçoit req_1 en premier et req_2 en second. Supposons que req_2 est enregistrée en premier à l'instant t_1 par exemple, et Req_1 à l'instant t_2. Nous avons $t_1 < t_2$.

Conformément à l'invariant 3, il existe un temps t' tel que $t_1 < t' < t_2$ et :

STATE($p_{j+1}(t_1)$)=*Busy*, STATE($p_{j+1}(t')$)=*Idle* et STATE($p_{j+2}(t_2)$)=*Busy*.

Supposons que p_{j+1} quitte le système à l'instant t'. Donc, selon l'invariant 4, sa propre liste doit être vide (**Req** $= null$). Or, vu que la requête Req_1 n'a pas encore été enregistrée, elle ne peut que se trouver dans la liste **Req** associée à p_{j+1}, d'où **Req** $\neq null$.

Nous déduisons de ce fait que Req_2 n'est jamais enregistrée avant Req_1. En conclusion, les requêtes sont toujours servies dans le même ordre où elles ont été émises. □

Démonstration. Puisque pour tout processus p_j, sa connexion à l'arbre par son ***Parent*** (p_{j+1}) est garantie quelle que soit l'évolution du système (lemme 4), et aussi longtemps que les requêtes sont servies selon l'ordre de leur émission (lemme 5), alors p_j finit par accéder à la **section critique**. Par conséquent, nous pouvons affirmer que notre **ADEMLE** vérifie la propriété de *vivacité*. □

4.4.2 Propriété de *sûreté*

Théorème 2. *(Sûreté) À tout moment, il n'y a qu'un seul jeton dans le système. En d'autres termes, le système garantit qu'au plus, un processus exécute la* ***section critique*** *à un instant* t.

Démonstration. Initialement, le système comprend un seul jeton détenu par la *racine*, p_0. L'algorithme progresse de manière à ce que le jeton passe d'un processus à un autre à travers la liste chaînée. Chaque processus bien que partant, cède le jeton à son *Suivant* ou, le cas échéant à son *Précédent* (section 4.3.2). Aussi, la liste chaînée est indivisible quelles que soient les circonstances (lemme 4). En conclusion, l'unicité du jeton est préservée d'où l'**ADEMLE** vérifie la propriété de *sûreté*. □

4.5 Synthèse du chapitre

Nous avons présenté dans ce chapitre un nouvel algorithme distribué d'exclusion mutuelle, basé sur celui de Naimi et Tréhel. Par l'algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs (**ADEMLE**), nous composons un système avec des processus interchangeables.

Le dynamisme apporté à l'algorithme originel n'altère nullement l'ordre des requêtes déposées dans la liste chaînée. Notre algorithme évolue tout en assumant les mouvements continuels des processus. Dans la réalité, ces processus sont des *peers* autonomes, distribués sur des sites hébergeant des ressources.

Notre algorithme maintient une complexité logarithmique et assure son passage à l'échelle. Par conséquent, l'**ADEMLE** s'accommode parfaitement aux propriétés de l'interface **DHO** (chapitre 3) d'où une intégration ultérieure (chapitre 6).

Afin de répondre pleinement aux prescriptions de l'interface **DHO**, nous poursuivons ce travail théorique par l'élaboration d'un deuxième algorithme qui s'inscrit en continuité de l'**ADEMLE**, dans lequel l'éventualité des accès partagés en lecture est étudiée. Le chapitre suivant conclut ce travail théorique par l'algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs et partagés (**ADEMLEP**).

Chapitre 5

Algorithme dynamique d'exclusion mutuelle à *locks* exclusifs et partagés (ADEMLEP)

Sommaire

5.1 Algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés (AEMLEP) 84
5.1.1 Administration des requêtes mixtes 85
5.1.2 Complexité des messages envoyés dans l'**AEMLEP** 86
5.1.3 L'entrée en **section critique** d'une chaîne de lecteurs 86
5.1.4 Libération de la **section critique** 87
5.2 Preuve de l'AEMLEP 89
5.2.1 Propriété de *vivacité* 89
5.2.2 Propriété de *sûreté* 89
5.3 Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs et partagés (ADEMLEP) 89
5.3.1 Déconnexion d'un processus appartenant à la chaîne de lecteurs 90
5.3.2 Départ du *gestionnaire de lecteurs* 90
5.3.3 Départ du ***prochain écrivain*** 92
5.4 Preuve de l'ADEMLEP 92
5.4.1 Propriété de *vivacité* 92
5.4.2 Propriété de *sûreté* 93
5.5 Synthèse du chapitre 94

Ce chapitre présente un deuxième apport à l'algorithme Naimi-Tréhel. Il s'inscrit en continuité de l'**ADEMLE** (chapitre 4). Comme nous l'avons déjà mentionné dans l'introduction générale, notre objectif dans cette thèse est de concevoir une API basée sur les fondements de *Data Handover* dont l'élaboration a déjà été présentée dans un contexte centralisé (chapitre 3). Rappelons que dans ce modèle, les *clients*

tentent d'accéder à une ressource en lecture, ou en écriture par l'appel des fonctions **DHO**.

Dans ce chapitre, nous voulons parvenir à un modèle dans lequel deux types d'accès à la **section critique** sont admis, et ce dans le but de parfaire le *locking/mapping* en lecture/écriture.

Dans notre approche, nous nous imposons l'obligation de servir les requêtes quelles qu'elles soient, selon le même ordre de leur insertion, c'est-à-dire en FIFO. Néanmoins les demandes successives de *locking* de type lecture partageront la **section critique**. Le jeton prendra alors une deuxième tournure dès lors qu'il sera détenu par un lecteur succédant à un écrivain. Ainsi, si d'autres lecteurs se suivent, ils seront autorisés à accéder à la **section critique**.

La concrétisation de ce modèle requiert de revoir les modalités d'accès à la **section critique** décrites précédemment dans l'**ADEMLE** du chapitre précédent. Les propriétés de déconnexion et d'accès partagé sont fusionnées pour donner lieu à un deuxième algorithme étendu : Algorithme dynamique d'exclusion mutuelle à *locks* exclusifs et partagés (**ADEMLEP**).

Afin de simplifier la compréhension de notre modèle, nous considérons dans un premier temps, la juxtaposition des jetons en l'absence de déconnexion des processus. Ainsi, l'Algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés (**AEMLEP**) expose de nouveaux procédés pour le traitement de deux types de requêtes (section ci-dessous). Par ailleurs, la preuve et la complexité de l'**AEMLEP** sont également fournies (section 5.2).

Enfin, l'Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs et partagés (**ADEMLEP**) conclut la partie théorique de notre travail, par l'introduction et l'étude de la déconnexion des processus (section 5.3) tous types confondus. La preuve complète de l'algorithme est finalement présentée, là où les propriétés *sûreté* et *vivacité* sont démontrées (section 5.4).

5.1 Algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés (AEMLEP)

Avant toute chose, les processus associent un type à leur requête. Par conséquent, ils ont l'alternative de demander un jeton partagé. Ainsi, si plusieurs processus demandeurs d'un accès partagé se succèdent dans la liste chaînée (*Suivant*, *Précédent*), ils accèderont simultanément à la **section critique**.

Nous désignons par lecteur, tout processus demandant un accès partagé, et chaîne de lecteurs, une succession de lecteurs dans la liste chaînée (*Suivant*, *Précédent*).

Cette section évoque les conséquences relatives à l'adjonction du jeton partagé au jeton exclusif. Pour ce faire, nous complétons l'algorithme précédent (**ADEMLE**) par d'autres facteurs. De nouvelles démarches relatives au déroulement des requêtes s'imposent. Pour commencer, nous associons les variables locales suivantes à la structure définie dans les sections 4.2.2 et 4.3 du chapitre 4, à un processus donné p_j :

Type **:** Associé à la requête, le type prendra soit le caractère "W" pour l'accès exclusif c'est-à-dire en écriture, soit le caractère "R" pour l'accès partagé c'est-à-dire en lecture.

Gestionnaire des lecteurs : Ce rôle est attribué au premier processus qui demande un accès en lecture à la suite d'un processus qui a demandé un jeton exclusif, initialement *null*.

Nombre de lecteurs : Ce compteur est incrémenté par le *gestionnaire de lecteurs* , pour chaque lecteur de la liste chaînée (*Suivant*, *Précédent*) qui entre en **section critique**.

Prochain écrivain : Le premier processus demandant un jeton exclusif, suite à un ou plusieurs lecteurs.

5.1.1 Administration des requêtes mixtes

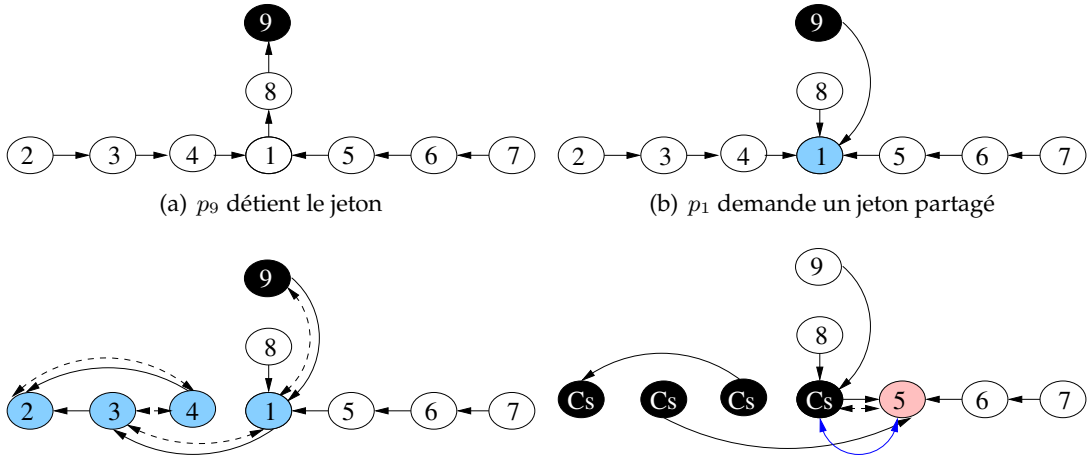
Il est trivial de constater que le remaniement de l'**ADEMLE** concerne la deuxième structure, à savoir la liste chaînée (*Suivant*, *Précédent*).

Soit $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ une séquence de processus dans la chaîne (*Suivant*, *Précédent*) avec des requêtes en suspens, tel que p_{j+1} est le *Suivant* de p_j, et p_{j-1} le *Précédent* de p_j. Rappelons que la *racine* de l'arbre met à jour son *Suivant* p_j lorsque ce dernier envoie une requête à son *parent* (section 4.2.3 du chapitre précedent).

Par la suite, p_j pointe son *Précédent* sur l'ancienne *racine*, c'est-à-dire p_{j-1} dans la chaîne. À ce moment là, p_j et p_{j-1} s'échangent mutuellement leur type de requête. Plusieurs cas se présentent :

- Si le type de p_{j-1} est W tandis que R est celui de p_j, p_j se proclame *gestionnaire de lecteurs*.
- Si pour p_j et p_{j-1} le type est R, p_{j-1} envoie en plus du type, le *gestionnaire de lecteurs*. Par la suite, p_j envoie un message au *gestionnaire de lecteurs* pour l'informer de sa présence. Notons que le *gestionnaire de lecteurs* peut être dans l'intervalle $[p_0..p_{j-1}]$.
- Si le type de p_{j-1} est R alors que celui de p_j est W, p_{j-1} va tout de même envoyer le *gestionnaire de lecteurs* à p_j. Ce dernier envoie donc un message à ce gestionnaire qui pointera son ***prochain écrivain*** sur lui. Ainsi, le *gestionnaire de lecteurs* et le ***prochain écrivain*** seront liés directement dans la chaîne.
- Si pour p_j et p_{j-1} le type est W, les deux processus s'échangent mutuellement leur type et mettent à jour respectivement leur variable *Suivant*, *Précédent*.

La figure 5.1 montre un exemple de quatre requêtes consécutives. Dans la figure 5.1(a) p_9 détient un jeton exclusif et occupe la **section critique**. P_1 devient le *gestionnaire de lecteurs* (figure 5.1(b)) suite à sa demande d'un accès partagé à la **section critique**. La figure 5.1(c) englobe les résultats des événements produits par l'envoi des requêtes partagées par p_3, p_4 et p_2 consécutivement. Dans la figure 5.1(d), la chaîne des lecteurs entre en **section critique** et p_5 envoie une demande d'accès exclusive.



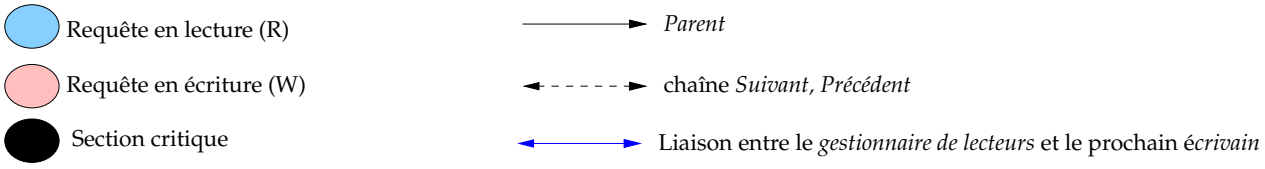
(a) p_9 détient le jeton

(b) p_1 demande un jeton partagé

(c) p_3,p_4 et p_2 demandent un jeton partagé

(d) La chaîne des lecteurs entre en **section critique** et p_5 demande un jeton exclusif

FIGURE 5.1 – Un scénario avec des requêtes mixtes



5.1.2 Complexité des messages envoyés dans l'AEMLEP

Par rapport à l'**ADEMLE**, nous comptabilisons trois messages additionnels par processus dans l'opération d'envoi d'une requête. Ils sont relatifs à l'échange du type de requête entre le *Suivant* et son *Précédent*, mais aussi à l'envoi du *gestionnaire de lecteurs* dans certains cas. Par conséquent, la complexité des messages envoyés dans le traîtement des requêtes demeure logarithmique *O(logn)* .

5.1.3 L'entrée en section critique d'une chaîne de lecteurs

Nous analysons, dans ce qui suit, l'entrée en **section critique** des processus dont les requêtes en lecture se succèdent dans la chaîne (*Suivant*, *Précédent*).

Soit $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ un sous ensemble de la chaîne (*Suivant*, *Précédent*) composé d'une séquence de lecteurs successifs. Dès que le *gestionnaire de lecteurs* (p_0) entre en **section critique**, il invite son *Suivant* à y accéder. De même, p_1 entre en **section critique** et invite son *Suivant* à le suivre. Le processus se poursuit tel une réaction en chaîne, jusqu'à ce que le dernier lecteur (p_n) entre en section (algorithme 5.1, ligne 11).

Lorsqu'un lecteur donné p_j entre en **section critique**, il avertit son *gestionnaire de lecteurs* par l'envoi d'un message, qui à son tour incrémente le *nombre de lecteurs*. En matière de complexité, nous comptons un message supplémentaire par processus.

Algorithme 5.1 Entrée en **section critique** (**AEMLEP**)

Réception du jeton par un processus donné p_j. P_j entre en **section critique**.

1: p_j entre en **section critique**
2: **si** ($type(Req) = r$) **alors**
3: **si** ($type(Précédent) = w$) **alors**
4: p_j = *gestionnaire de lecteurs*
5: p_j entre en **section critique**
6: *nombre de lecteurs* $++$
7: **sinon**
8: p_j entre en **section critique**
9: p_j envoie message au *gestionnaire de lecteurs*
10: **fin si**
11: **si** ($Suivant(p_j) \neq null \wedge type(Suivant) = r$) **alors**
12: p_j envoie message(**section critique**) à *Suivant*
13: **fin si**
14: **fin si**

Lemme 6. *Dès que le gestionnaire de lecteurs entre en **section critique**, toute la chaîne des lecteurs y accède.*

Démonstration. Puisque chaque processus connaît le type de requête émise par son *Suivant* (section 5.1.1), dès que le premier processus d'une chaîne de lecteurs (*gestionnaire de lecteurs*) entre en **section critique**, il alerte son *Suivant* pour y accéder dans le cas où ce dernier est lecteur. Le *Suivant* agit pareillement et la même action se répète jusqu'au dernier lecteur. Donc, toute la chaîne des lecteurs entre en **section critique**. □

5.1.4 Libération de la section critique

Soit $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ une chaîne de lecteurs.

Chaque lecteur qui libère la **section critique**, se doit d'informer le *gestionnaire de lecteurs* (p_0) qui, à la réception d'un message, décrémente le *nombre de lecteurs* (ligne 11 de l'algorithme 5.2).

Le *gestionnaire de lecteurs* garde le jeton même s'il a libéré la **section critique**. Il ne cède le jeton à un éventuel ***prochain écrivain*** que si le *nombre de lecteurs* est *null*. Nous rappelons que le *gestionnaire de lecteurs* et le ***prochain écrivain*** sont liés directement dans la structure de la chaîne (*Suivant*, *Précédent*). Donc, Ils possèdent un pointeur supplémentaire dans la chaîne.

Lorsque le ***prochain écrivain*** reçoit le jeton, il accède à la **section critique** en mode exclusif. Donc, le basculement du jeton d'un mode à un autre dépend du type de requêtes émises entre processus adjacents dans la chaîne.

Invariant 6. *Le gestionnaire de lecteurs ne cède le jeton au premier processus qui demande un accès exclusif au-delà de la chaîne des lecteurs, que si tous les lecteurs qui le précèdent dans la*

Algorithme 5.2 Libération de la **section critique** (**AEMLEP**)

Libération de la **section critique**. Traitement des requêtes mixtes
1: p_j libère la **section critique**
2: **si** ($type(Req) = w$) **alors**
3: **si** (*Suivant* $\neq null$) **alors**
4: p_j envoie message jeton au *Suivant*
5: **fin si**
6: **sinon si** ($p_j =$ *gestionnaire de lecteurs*) **alors**
7: **si** (***prochain écrivain*** $\neq null \wedge$ *nombre de lecteurs* $= null$) **alors**
8: p_j envoie message (jeton) au ***prochain écrivain***
9: **fin si**
10: **sinon**
11: p_j envoie message au *gestionnaire de lecteurs*
12: **fin si**

chaîne (Suivant, Précédent), y compris le gestionnaire de lecteurs, ont libéré la ***section critique****.*

Lemme 7. *Un processus p_j en possession d'un jeton exclusif ne le partage jamais avec un deuxième en accès partagé.*

Démonstration. Nous devons démontrer que deux processus en attente, qui diffèrent par le type de leurs requêtes correspondantes n'accèdent, jamais à la **section critique** simultanément.

Soit $\{p_1..p_{j-1}, p_j, p_{j+1}..p_n\}$ une chaîne de lecteurs contigus dont les requêtes sont en attente, tel que p_1 est le *gestionnaire de lecteurs* et p_{j+1} est le *Suivant* de p_j. Deux hypothèses se présentent :

Soit le cas d'un processus p_0 précédant p_1 dans la chaîne (*Suivant*, *Précédent*) accédant à la **section critique** grâce à un jeton exclusif. D'après le théorème 2 du chapitre 4, il est impossible que p_0 partage le jeton avec p_1.

D'un autre côté, supposons que le *gestionnaire de lecteurs* (p_1) détient le jeton partagé, et qu'il existe au moins un processus parmi ceux de la chaîne des lecteurs en **section critique**. Admettons que p_{n+1}, le premier processus à demander un jeton exclusif après la chaîne des lecteurs (***prochain écrivain***) est également en **section critique**. Dans ce cas, p_{n+1} partagerait la **section critique** avec au moins un processus de la chaîne des lecteurs. De ce fait, la présence de p_{n+1} en **section critique** implique que le jeton lui a été transmis par le *gestionnaire de lecteurs*.

Or, du moment qu'un processus au moins occupe la **section critique**, le *nombre de lecteurs* est positif. Nous savons que p_1 ne cède le jeton au ***prochain écrivain*** que si le *nombre de lecteurs* est *null* (algorithme 5.2, ligne 7). D'où, p_{n+1} ne peut entrer en **section critique** en même temps qu'un lecteur.

Par conséquent, quel que soit le type de jeton dont il dispose, un processus ne partage nullement la **section critique** avec un deuxième processus en quête d'un jeton d'un autre type. La propriété de *sûreté* est donc vérifiée. □

5.2 Preuve de l'AEMLEP

L'**AEMLEP** diffère de l'**ADEMLE**, que nous avons démontré dans le chapitre précédent, par l'introduction du mode partagé et de l'absence de la mobilité des processus. Donc, nous allons nous référer en majeure partie à la preuve de l'**ADEMLE** (section 4.4) du chapitre précédent.

5.2.1 Propriété de *vivacité*

Cette propriété a été vérifiée pour le cas de l'**ADEMLE** (théorème 1). Nous devons démontrer la préservation de la propriété de *vivacité* lorsque le mode partagé est associé au mode exclusif.

Théorème 3. *Si un processus réclame la* ***section critique*** *en mode partagé, il l'obtient au bout d'un temps fini.*

Démonstration. Du moment que toute requête émise par un quelconque processus est satisfaite (théorème 1), celle du *gestionnaire de lecteurs* aboutit au bout d'un temps fini. De ce fait, toute la chaîne des lecteurs entre en **section critique** (lemme 6). Par conséquent, tous les processus réclamant la **section critique** en mode partagé finissent par y accéder. □

5.2.2 Propriété de *sûreté*

Sachant qu'elle a déjà été démontrée pour l'**ADEMLE** (section 4.4.2), il suffit de montrer qu'elle est maintenue en présence d'un jeton partagé.

Théorème 4. *La* ***section critique*** *est occupée soit par un processus en possession d'un jeton exclusif, soit par une chaîne de lecteurs dont le jeton partagé est détenu par le gestionnaire de lecteurs.*

Démonstration. Nous avons démontré que deux processus en possession de deux jetons distincts ne peuvent jamais accéder à la **section critique** simultanément (lemme 7). Donc la propriété de *sûreté* est garantie lorsque le mode partagé est introduit dans l'algorithme. □

5.3 Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs et partagés (ADEMLEP)

Cette section expose notre dernier apport dans l'algorithme Naimi-Tréhel. L'**ADEMLEP** présente la synthèse de notre approche théorique. Après avoir introduit l'aspect de volatilité des processus dans l'**ADEMLE** (section 4.2 du chapitre 4) et le mode partagé dans l'**AEMLEP** (section 5.1), nous accumulons ces deux propriétés dans ce dernier algorithme.

Ainsi, nous étudions dans ce qui suit les conditions et les conséquences de déconnexion des processus du système.

Aux côtés des conditions nécessaires au détachement d'un processus donné p_j (chapitre 4, section 4.3.1) du système distribué, d'autres contraintes s'imposent, notamment celles rattachées au *gestionnaire de lecteurs*, ***prochain écrivain*** ou d'un quelconque processus appartenant à la chaîne des lecteurs.

5.3.1 Déconnexion d'un processus appartenant à la chaîne de lecteurs

Soit $\{p_0..p_j, p_{j+1}..p_n\}$ une chaîne de lecteurs telle que p_0 est le *gestionnaire de lecteurs*.

Soit ***lecteur_exit*** une variable booléenne administrée par ce gestionnaire, et associée à une suite de lecteurs dans la chaîne (*Suivant*, *Précédent*), initialement *false*.

Les dispositions suivantes sont prises avant tout départ d'un lecteur autre que le *gestionnaire de lecteurs* :

- Deux processus ne peuvent pas quitter le système simultanément. Cette nouvelle disposition permet au *gestionnaire de lecteurs* d'être à jour avec la structure de la chaîne qu'il administre.
- Dans le cas où p_j souhaite quitter le système, il demande l'autorisation au *gestionnaire de lecteurs* (p_0). Ce dernier consulte la valeur correspondante à la variable ***lecteur_exit***. Si elle s'avère *false*, p_j est autorisé à partir.
- Une valeur *true* associée à la variable ***lecteur_exit*** atteste qu'un autre processus de la chaîne des lecteurs prépare son départ. P_j devra donc attendre le basculement de cette variable.
- Sitôt *false*, le *gestionnaire de lecteurs* repositionne la variable ***lecteur_exit*** à *true*, rejoint l'état *Exiting* et commence son départ.
- En plus des contraintes imposées à p_j énoncées dans la section 4.3.2 du chapitre 4, p_j libère le *lock* s'il le détient puis, informe le *gestionnaire de lecteurs*. Ce dernier décrémente le *nombre de lecteurs*.

Invariant 7. *Dans une chaîne de lecteurs, un lecteur à la fois est autorisé à quitter le système.*
Soit $j \in 0..n$ $\text{STATE}(p_j(t)) = \text{Exiting} \Rightarrow \forall k \in 0..n$ *tel que* $k \neq j$ $\text{STATE}(p_k(t)) \neq \text{Exiting}$

5.3.2 Départ du *gestionnaire de lecteurs*

Le rôle prépondérant du *gestionnaire de lecteurs* dans la chaîne, le contraint à prendre des dispositions spécifiques à sa position. En fait, la chaîne des lecteurs n'est pas seule concernée par le départ du *gestionnaire de lecteurs*. Le ***prochain écrivain*** doit également en être informé car, c'est à lui que le *gestionnaire de lecteurs* cèdera le jeton.

Soit $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ une chaîne de lecteurs telle que p_0 est le *gestionnaire*.

Ainsi, les tâches spécifiques qu'un *gestionnaire de lecteurs* doit accomplir avant de se détacher du système sont les suivantes :

- Premièrement, p_0 vérifie la valeur correspondante à sa variable locale ***lecteur_exit*** et attend qu'elle soit *false*.

- Si aucun processus demandant un accès exclusif (***prochain écrivain***) ne suit la chaîne des lecteurs, p_0 inhibe l'accès à la **section critique** à tout nouveau lecteur.
- De même, p_0 patiente dans le cas où le ***prochain écrivain*** est lui même partant. Il se met par la suite en état *Exiting*.
- Au cas où les lecteurs sont en phase d'accès à la **section critique**, p_0 attend de recevoir le signal du dernier lecteur (section 5.1.3).
- Une fois tous les lecteurs en **section critique**, p_0 envoie le *nombre de lecteurs* à son *Suivant*. Donc, p_1 est invité à devenir le nouveau *gestionnaire de lecteurs*.
- Lorsque l'information parvient à p_1, celui-ci envoie un message à son *Suivant*. Ce dernier met à jour son nouveau *gestionnaire de lecteurs*. De même, p_2 alerte son voisin de ce changement et ainsi de suite jusqu'à ce que le ***prochain écrivain*** p_{n+1} par exemple, en soit informé. Ce dernier rétablit par la suite le lien avec le nouveau *gestionnaire de lecteurs*.
- Finalement, p_1 remet ***lecteur_exit*** à *false*.

Lemme 8. *Les lecteurs successifs doivent pouvoir accéder à la **section critique** même si leur gestionnaire se déconnecte du système.*

Démonstration. La preuve est assez triviale. D'une part, le jeton n'est pas perdu puisque la déconnexion du *gestionnaire de lecteurs* actuel, et celle de son successeur ne se produisent jamais simultanément (invariant 7).

D'autre part, l'information n'est jamais perdue pendant la mutation du *gestionnaire de lecteurs*, puisqu'elle ne prend effet qu'après l'entrée en **section critique** de toute la chaîne des lecteurs (section 5.3.2).

Par conséquent, le *gestionnaire de lecteurs* est constamment à jour avec les événements survenus dans la chaîne des lecteurs quelles que soient les circonstances. □

Lemme 9. *La chaîne des lecteurs ne partage jamais la **section critique** avec un processus qui demande un accès exclusif, même dans le cas du départ de l'un de ses processus.*

Démonstration. Nous devons démontrer que le lemme 7 est garanti dans les circonstances de départ.

(1) Supposons d'un côté qu'un processus en accès exclusif à la **section critique**, précédant une chaîne de lecteurs décide de quitter le système. Dans ce cas, le passage du jeton au *gestionnaire de lecteurs* est manifeste après que ce processus ait quitté la **section critique** (lemme 7).

D'un autre côté, supposons que la chaîne des lecteurs est en **section critique**. Si le ***prochain écrivain*** occupe également la **section critique**, cela veut dire que ces deux processus détiennent le même jeton.

Or, le *gestionnaire de lecteurs*, obtient le *lock* en premier. Donc, il entre en **section critique** ainsi que tous les autres lecteurs qui ne sont pas affectés par le départ de leur gestionnaire (lemme 8). Par ailleurs, le *gestionnaire de lecteurs* ne partage jamais le jeton avec un ***prochain écrivain*** (lemme 7). □

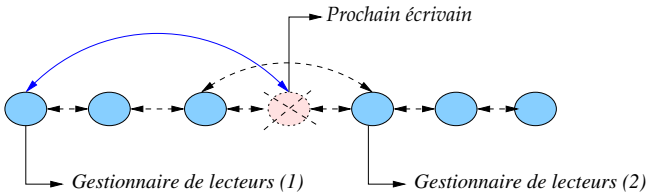


FIGURE 5.2 – Départ d'un ***prochain écrivain*** au milieu de deux chaînes de lecteurs

5.3.3 Départ du *prochain écrivain*

Sachant que le ***prochain écrivain*** est en liaison directe avec le *gestionnaire de lecteurs* dans la chaîne (*Suivant*, *Précédent*), ces deux processus ne peuvent pas quitter le système en même temps.

De ce fait, en plus des contraintes imposées pour le départ d'un processus p_j, si ce dernier est un ***prochain écrivain***, il devra alors se renseigner sur l'état du *gestionnaire de lecteurs* qui le devance dans la liste chaînée (*Suivant*, *Précédent*).

Le ***prochain écrivain*** patiente jusqu'à la terminaison du départ du *gestionnaire de lecteurs*, si ce dernier est sortant. Le *gestionnaire de lecteurs* envoie un message au ***prochain écrivain*** juste après avoir remis la variable ***Exiting*** à *false*.

Autrement, le ***prochain écrivain*** devient *Exiting* et en avertit aussitôt le *gestionnaire de lecteurs*.

Passons maintenant aux successeurs du ***prochain écrivain*** dans la liste chaînée : Si le *Suivant* du ***prochain écrivain*** est lecteur, il est getionnaire d'une autre chaîne de lecteurs ; ce dernier perdra systématiquement son rôle en tant que gestionnaire. En effet, ce processus sera lié, ainsi que d'éventuels successeurs, à la précédente chaîne des lecteurs.

Par contre, si la requête déposée par le *Suivant* du ***prochain écrivain*** dans la chaîne est de type exclusive (ligne 6 de l'algorithme 5.3), le ***prochain écrivain*** établira une liaison entre le *gestionnaire de lecteurs* et son *Suivant* de sorte que ce dernier devienne le nouveau ***prochain écrivain***. La figure 5.2 est un exemple illustratif d'une chaîne (*Suivant*, *Précédent*) composée de deux chaînes de lecteurs, séparées par un ***prochain écrivain*** en voie de déconnexion.

Les instructions de l'algorithme 5.3 relatives à la sortie d'un processus du système (algorithme 5.3, complèteront celles énoncées précédemment dans l'**ADEMLE**.

5.4 Preuve de l'ADEMLEP

5.4.1 Propriété de *vivacité*

La propriété de *vivacité* a été démontrée pour l'**AEMLEP**, donc elle est vérifiée pour les *locks* exclusifs et partagés en l'absence de l'aspect déconnexion des processus (théorème 3).

Aussi, du moment que tout processus demandant la **section critique** l'obtient au bout d'un temps fini en toutes circonstances, quel que soit le type de requête émise (théorème 1,lemme 8), notre algorithme satisfait la propriété de *vivacité*.

5.4.2 Propriété de *sûreté*

En l'absence de volatilité de processus, la propriété de *sûreté* est assurée pour l'**AEMLEP** (théorème 4). Par ailleurs, puisque les deux jetons ne sont jamais attribués simultanément quelles que soient les circonstances (lemmes 7, 9), alors l'**ADEMLEP** satisfait la propriété de *sûreté*.

Algorithme 5.3 Fonction EXIT (**ADEMLEP**)

Fonction EXIT pour un processus donné p_j (**ADEMLEP**.

1: **si** (type(p_j) = r) **alors**
2: p_j envoie message (***Exiting***) au *gestionnaire de lecteurs*
3: **si** (***Exiting*** (*gestionnaire de lecteurs*) = true) **alors**
4: p_j attend
5: **fin si**
6: **si** (*Suivant* $\neq null \wedge$ type(*Suivant*) = w) **alors**
7: p_j envoie le *gestionnaire de lecteurs* au *Suivant*
8: **fin si**
9: **sinon si** (***lecteur_exit*** = true) **alors**
10: p_j attend
11: **si** ($p_j \neq$ *gestionnaire de lecteurs*) **alors**
12: **si** p_j est en **section critique alors**
13: p_j libère la **section critique** et envoie un message au *gestionnaire de lecteurs*
14: **fin si**
15: **sinon**
16: p_j interdit l'accès à la **section critique** à tout processus
17: **si** (Etat(***prochain écrivain***)=*Exiting*) **alors**
18: p_j attend
19: **fin si**
20: **si** (p_j n'est pas en **section critique**) **alors**
21: **tant que** (un lecteur au moins est en **section critique**) **faire**
22: p_j attend
23: **fin tant que**
24: **fin si**
25: p_j envoie message (*nombre de lecteurs*) à son *Suivant*
26: **fin si**
27: **fin si**

5.5 Synthèse du chapitre

Ce chapitre conclut la partie théorique de notre travail dans cette thèse. Nous avons présenté différentes versions de l'algorithme originel. Dans notre démarche, nous avons préféré étudier une propriété à la fois dans plusieurs algorithmes distincts afin de faciliter la compréhension du modèle. Ainsi, L'algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés (**ADEMLE**) développe le départ volontaire des processus, tout en garantissant la connnectivité des deux structures dynamiques du système global.

L'algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés (**AEMLEP**) combine deux types d'accès, tout en assurant une équité totale pour toutes les requêtes émises. Enfin l'**ADEMLEP** courronne cette étude par l'agrégation des deux propriétés citées [HGB12].

L'**ADEMLEP** remplit pleinement les conditions pour une implantation dans une plate-forme à grande échelle. Nous en avons donné la preuve théorique et démontré que la complexité des messages préserve son échelle logarithmique.

Le chapitre suivant décrit notre système *peer-to-peer* dont le modèle est une hybridation de nos algorithmes étendus et du cycle de vie **DHO**. Il compose l'architecture interne de l'API. L'étude expérimentale qui suivra apportera une validation complète de notre approche distribuée.

Chapitre 6

Data Handover dans un système peer-to-peer

Sommaire

6.1 L'API DHO dans le paradigme client-serveur 96
6.2 Objectifs de notre système distribué 97
6.3 DHO dans un système *peer-to-peer* 97
6.3.1 Cycle de vie **DHO** 98
6.3.2 Gestion des requêtes exclusives 99
6.3.3 Déroulement d'un exemple pour un accès exclusif 101
6.3.4 Mobilité des *peers* 103
6.3.5 Gestion des requêtes mixtes 105
6.4 Environnement expérimental 107
6.4.1 Plate-forme matérielle 107
6.4.2 Plate-forme logicielle 108
6.4.3 Scénarios 108
6.5 Méthodologie d'évaluation 109
6.5.1 Durées observées 110
6.6 Évaluation des performances du système *peer-to-peer* pour les *locks* exclusifs 110
6.6.1 Paradigme Client-serveur vs système *peer-to-peer* en l'absence de retard au blocage 111
6.6.2 Impact de la répartition des *peers* sur les sites de la grille 116
6.6.3 Évaluation du cycle **DHO** pour les *locks* asynchrones 118
6.6.4 Évaluation du surcoût induit par la déconnexion des *peers* . . . 120
6.7 Observation du cycle DHO pour des requêtes asynchrones en accès partagé 122
6.8 Évaluation du cycle DHO pour des requêtes mixtes 124
6.9 Synthèse du chapitre 126

L'étude expérimentale présentée au chapitre 3 autour d'un modèle client-serveur a validé l'approche **DHO** et le modèle de communication.
Néanmoins, la centralisation de la ressource ne présente pas une solution en soi pour des applications distribuées sur des environnements de grande envergure telles que les grilles.

Ce chapitre est un rétrospectif de ce qui a été présenté jusqu'à présent. Le modèle *Data Handover* couplé aux algorithmes **ADEMLE**, **AEMLEP** et **ADEMLEP** constitue notre système distribué au dessous duquel l'API **DHO** évolue.

À titre comparatif, le modèle client-serveur est repris brièvement au début de ce chapitre (section 6.1) afin d'élucider les objectifs visés par le système distribué que nous proposons (section 6.2). Nous enchaînons par une présentation explicite du système proposé, en l'occurrence les retentissements des appels fonctionnels de **DHO** sur notre système (de la section 6.3.1 à la section 6.3.5).

La section 6.4 présente la plate-forme expérimentale et la base logicelle sur lesquelles nous nous sommes appuyés pour la mise en œuvre de notre système, tandis que la section 6.5 expose la méthodologie d'évaluation ainsi que les délais observés.

Le reste des sections est consacré à l'analyse des résultats expérimentaux. Nous étudions le comportement du système face à des requêtes exclusives et partagées, d'abord séparément, avant de conclure par les requêtes mixtes. Il ne demeure pas moins que la mobilité des *peers*, dans un système en cours d'évolution et les *locks* asynchrones font partie de notre procédé d'évaluation.

6.1 L'API DHO dans le paradigme client-serveur

Pour clarifier certains points relatifs à l'architecture interne de **DHO**, nous ouvrons une parenthèse sur l'étude que nous avons présentée dans le chapitre 3.

Le modèle décrit précédemment, repose sur le paradigme client-serveur dans lequel le *gestionnaire de verrou* situé au serveur, a la lourde tâche de dialoguer avec tous les *gestionnaires de ressources* associés aux *clients*. Dans ce schéma, nous pouvons dire que l'interface est exploitée au dessus d'une architecture à deux niveaux où le *gestionnaire de ressource* et le *handle* se rangent au degré inférieur (voir la figure 3.1 du chapitre 3).

Au niveau supérieur, le *client* ordonne l'accès à la ressource par le biais des fonctions **DHO**, tandis que les *gestionnaires de ressources* correspondants s'exécutent au niveau inférieur. Par ailleurs, les états attribués au *client* suite à l'appel d'une ressource donnée, découlent de ceux du *handle*. Ils témoignent du stade de la requête.

Nous soulignons que l'architecture interne de l'API est entièrement transparente pour l'utilisateur, quel que soit le paradigme adopté. Les routines appartenant à l'interface constituent la couche supérieure de **DHO** et demeurent inchangées.

Cependant, nous n'aspirons pas à un modèle où les ressources sont localisées et centralisées. Pour ne citer qu'un seul inconvénient, la seule défaillance du serveur compromettrait le fonctionnement de la librairie.

6.2 Objectifs de notre système distribué

Notre but est de garantir la portabilité du modèle **DHO** sur un système distribué de manière à être compatible avec les caractéristiques des grilles informatiques et des systèmes *peer-to-peer*. En d'autres termes, nous proposons un modèle basé sur les systèmes *peer-to-peer* qui évolue sur une grille informatique.

Plus précisément, notre système doit remplir les conditions suivantes :

- L'utilisateur doit pouvoir utiliser l'API en toute simplicité. Il doit ignorer aussi bien la localisation des ressources que la structure sous-jacente du système, aussi complexe soit-elle et aussi nombreux soient les nœuds et les *peers* qui la composent.
  Ainsi, tous les détails relatifs à la localisation des ressources à l'instar de ceux qui se rapportent aux caractéristiques physiques du réseau par exemple, doivent être occultés pour l'utilisateur.
- Lorsqu'il a besoin d'une ressource donnée pour son application, l'utilisateur doit la réclamer par son nom uniquement, au moyen d'une fonction **DHO**. Le système doit la lui fournir au bout d'un temps fini.
- Notre librairie doit pouvoir s'exécuter sur une plate-forme qui obéit à toutes les caractéristiques d'une grille, à savoir l'hétérogénéité et le passage à l'échelle ainsi que le dynamisme des nœuds.
- L'aspect dynamique traduit le départ volontaire des *peers* pour des raisons de maintenance ou d'indisponibilité temporaire ou définitive.
  Pour éviter toute confusion, nous précisons que le cas de la défaillance des nœuds n'est pas abordé, car il s'inscrit dans une thématique de recherche à part entière.
- Le dynamisme est provoqué par les *peers* qui composent notre système. Ainsi, les *peers* peuvent quitter comme ils peuvent joindre le système de façon transparente et abstraite pour l'utilisateur.
  Le mouvement des *peers* ne doit en aucun cas compromettre le déroulement des applications sous **DHO**.

6.3 DHO dans un système *peer-to-peer*

Notre système s'organise autour d'une architecture à trois niveaux qui résulte de l'agencement du cycle de vie **DHO** et des algorithmes dynamiques d'exclusion mutuelle à *locks* exclusifs et/ou partagés (**ADEMLE**, **AEMLEP** et **ADEMLEP**). La synthèse des trois chapitres précédents résume notre système distribué dont les propriétés sont définies ci-dessous :

Notre système distribué est composé d'un ensemble de *peers* qui, comme son nom l'indique, remplissent à la fois les fonctions de *client* et de *serveur*.

Les *peers* n'ont pas une connaissance globale de l'ensemble du système ; en revanche, chaque *peer* est connu de quelques autres *peers*. Ce nombre restreint de *peers* forme un voisinage.

Les ressources sont décentralisées. Tout *peer* peut demander une ressource comme il peut la proposer.

De manière abstraite, le *peer*, le *gestionnaire de ressource* et le *gestionnaire de verrou* composent respectivement l'ordre hiérarchique de l'architecture à trois niveaux associée à notre modèle.

Pour un *peer*, chaque ressource est associée à un *gestionnaire de ressource* et un *gestionnaire de verrou* qui interagissent continuellement pour son acquisition locale (*mapping*). À chaque ressource correspond un ou plusieurs *handles*. De plus, les *gestionnaires de verrou* du système entament des dialogues dans le but d'acquérir un jeton exclusif ou partagé.

Le *peer* fait intervenir le *gestionnaire de ressource* et le *gestionnaire de verrou* lorsqu'il veut demander une ressource, tandis que le *gestionnaire de verrou* seul est impliqué pour toute demande reçue. Ce dernier exécute les instructions au plus bas niveau de la structure, notamment ceux qui se rapportent à nos algorithmes étendus.

Le *gestionnaire de ressource* est le premier à saisir la requête d'une ressource donnée. Il est aussi responsable du *mapping/locking* de cette dernière localement.

Pendant que le système évolue, le *peer*, le *handle* et le *gestionnaire de verrou* conservent des informations à travers des états qui leurs sont assignés.

Les états du *peer* et du *handle* sont inhérents. Lorsque le *peer* est *locked* par exemple, on sait pertinemment que le *handle* est en état $locked^{\text{ew}}$ ou $blocked^{\text{cr}}$.

Si les états du *peer* et du *handle* révèlent la position de ces derniers vis-à-vis de la ressource et de l'accès à la **section critique**, l'état associé au *gestionnaire de verrou* dénote l'activité du *peer*.

En plus de l'état, le *handle* engloble d'autres variables auxquelles le *gestionnaire de ressource* a recours, notamment les *peers* qui composent le voisinage de p_j.

6.3.1 Cycle de vie DHO

Le cycle de vie de **DHO** que nous avons décrit dans la section 3.2.3 demeure inchangé. Cependant, nous allons y revenir pour détailler certains points en rapport avec la structure sous-jacente de notre système distribué.

Ainsi, dans le paradigme client-serveur, le *gestionnaire de ressource* intéragit uniquement avec le *gestionnaire de verrou* associé au serveur hébergeant la ressource. Toutes les actions sont déclenchées par le *client* puis, reprises par le *gestionnaire de ressource* et dirigées vers le *gestionnaire de verrou*.

Aussi, le passage d'un état à un autre, que ce soit pour le *client* ou pour le *handle*, dépend exclusivement des réponses envoyées par le *gestionnaire de verrou*.

A l'inverse, dans notre système distribué, plusieurs *peers* sont impliqués dans le cycle de vie de **DHO**. Un *peer* p_j dialogue avec trois *peers* distincts lorsqu'il réclame une ressource donnée, réalise son *mapping/locking* local ou lorsqu'il libère la ressource ; ces *peers* sont encapsulés dans le *handle*.

La figure 6.1 montre vers quels *peers* les fonctions **DHO** sont dirigées lorsqu'elles sont invoquées par l'utilisateur.

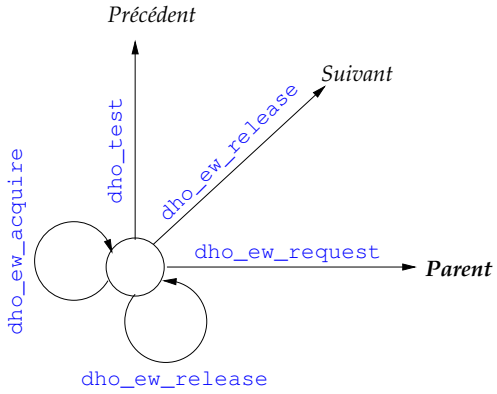


FIGURE 6.1 – Implication des *peers* dans l'appel des fonctions

Soit $\{p_0..p_{j-1}, p_j, p_{j+1}..p_n\}$ un ensemble de *peers* organisés dans une liste doublement chaînée, à l'image de l'algorithme **ADEMLE** (voir la section 4.2), tel que p_{j-1} est le *Précédent* de p_j et p_{j+1} est le *Suivant* de p_j.

Un *handle* est associé à chaque *peer* qui, suivant l'évolution du cycle, peut occuper les états suivants :

$$Q = \{valid\ , invalid\ , \textbf{\textit{Unlock}}\ , blocked^{\,\mathrm{cr}}, fetch^{\,\mathrm{cr}}, grant^{\,\mathrm{cr}}, locked^{\,\mathrm{cr}}, req^{\,\mathrm{cr}},\\ blocked^{\,\mathrm{ew}}, fetch^{\,\mathrm{ew}}, grant^{\,\mathrm{ew}}, locked^{\,\mathrm{ew}}, req^{\,\mathrm{ew}}\}$$

Cette liste est légèrement modifiée par rapport à celle énoncée dans le cas de la gestion centralisée (section 3.2.3.1 du chapitre 3), par l'ajout d'un état (*Unlock*) et, par la suppression de l'état (*push*). Les arguments sont donnés un peu plus bas (étapes 7 et 8 de la section 6.3.2).

6.3.2 Gestion des requêtes exclusives

Soit p_j un *peer* dans sa phase initiale représenté par l'état *Idle*. Le *handle* correspondant est donc *valid*. Cet état est atteint suite à l'appel de la fonction `dho_create`. Nous y reviendrons en détail un peu plus loin dans ce chapitre (section 6.3.4.2).

L'accomplissement du cycle de vie de **DHO** conformément aux trois niveaux associés au *peer*, au *gestionnaire de ressource* et au *gestionnaire de verrou* pour un accès exclusif, suit les étapes suivantes :

1. L'utilisateur fait appel à la fonction `dho_ew_request` pour demander un accès exclusif. Le *peer* associé bascule à l'état *Req*. À un degré plus bas, le *gestionnaire de ressource* correspondant contacte le *gestionnaire de verrou* correspondant au même *peer*.

Ce dernier demande un jeton exclusif à son ***Parent*** dans la structure la plus basse et rejoint l'état *Sending*. Le *gestionnaire de verrou* reprend son état *Idle* une fois qu'il a fini d'envoyer les ACKS à ses ancêtres dans la branche parentale (algorithme 4.3 du chapitre 4).

D'autre part, le *gestionnaire de ressource* assigne l'état *req* $^{\text{ew}}$ au *handle*. Ce dernier séjourne dans cet état tant que la ressource n'a pas été acquise.

2. À partir de là, le *gestionnaire de verrou* peut recevoir des demandes en provenance de ses *fils* et regagner l'état *Busy*.

3. Au terme de l'acheminement de la requête, p_j prend place en tête de la liste chaînée (*Suivant*, *Précédent*). Ainsi, le *gestionnaire de verrou* attend de recevoir le jeton de son *Précédent* (p_{j-1}). Pendant ce temps, l'application peut continuer son exécution durant un certain temps (T_{Wblocked}) indépendamment de la ressource.

   Selon le cycle de vie de **DHO**, deux situations se présentent :

   – Par son utilisateur, p_j décide de recourir à la fonction `dho_test` laquelle est dirigée par le *gestionnaire de ressource* vers le *gestionnaire de verrou* du même *peer*. Cette fonction renvoie l'état du *handle*. Celui-ci devient *grant* $^{\text{ew}}$ dans le cas où le jeton est déjà accordé.
   – Dans le cas où le retour de la fonction `dho_test` révèle la non attribution du verrou, p_j invoque la fonction `dho_ew_acquire` pour joindre l'état *Blocked*. Par conséquent, le *gestionnaire de ressource* met à jour l'état du *handle* pour la valeur *blocked* $^{\text{ew}}$.

4. Le *gestionnaire de verrou* de p_j est contacté par celui de p_{j-1} pour lui remettre le jeton après que le *gestionnaire de ressource* de ce dernier ait résilié la ressource. Ainsi, p_j est invité à entrer en **section critique**. Une fois le jeton acquis, le *gestionnaire de verrou* contacte le *gestionnaire de ressource* qui modifie l'état du *handle* pour la valeur *grant* $^{\text{ew}}$, si cette dernière n'est pas encore mise à jour (étape 3).

5. À ce stade, le *gestionnaire de ressource* contacte son homolgue du *peer Précédent* (p_{j-1}) et procède localement au *locking / mapping* de la ressource. Pendant ce temps, le *handle* rejoint temporairement l'état *fetch* $^{\text{ew}}$.

6. Le chargement de la ressource en mémoire locale et l'entrée en **section critique** du *gestionnaire de verrou* se concrétisent par l'achèvement du *mapping / locking*. Le *handle* et le *peer* deviennent respectivement *locked* $^{\text{ew}}$, *Locked*.

7. Après un temps fini, la résiliation de la ressource s'annonce par l'appel de la fonction `dho_ew_release` suite auquel p_j reprend son état initial *Idle*. Ce passage traduit l'accomplissement d'un cycle **DHO**.

   Par la suite, le *gestionnaire de ressource* attribue l'état *Unlock* et non pas l'état *valid* au *handle* comme dans le modèle centralisé (section 3.2.3).

   En fait, cet état exprime une phase intermédiaire durant laquelle le *peer* a déjà résilié la ressource mais détient encore le jeton. À partir de là, le *gestionnaire de verrou* est prêt ou bien, il a déjà cédé le jeton à un potentiel *Suivant*.

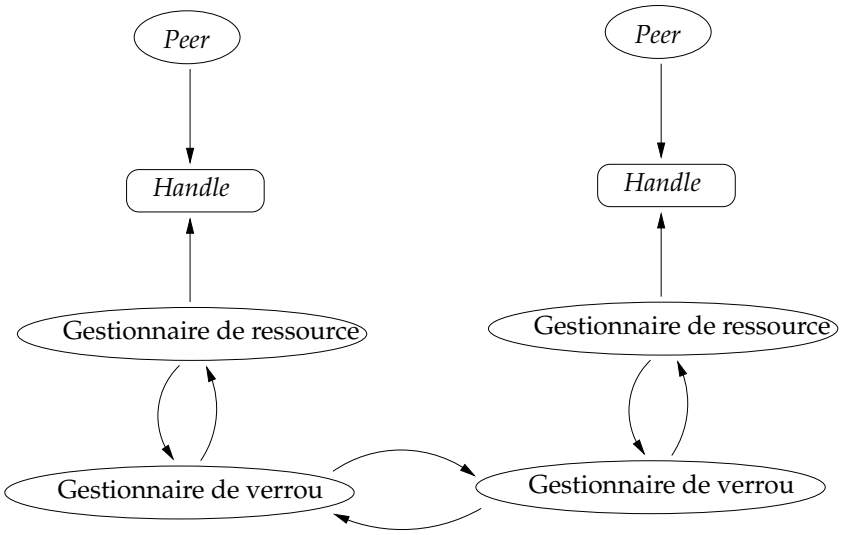


FIGURE 6.2 – Intéractions entre processus de chaque *peer*

8. Contrairement au modèle centralisé, la structure actuelle omet l'opération de mise à jour de la ressource.

   Rappelons que dans le paradigme client-serveur, le *gestionnaire de ressource* doit envoyer au serveur une version actualisée de la ressource afin de préserver une cohérence d'accès entre tous les *clients*.

   Par conséquent, cette opération n'a plus de raison d'être. En effet, lorsque le *Suivant* de p_j (p_{j+1}) réalisera le *locking/mapping*, il prélèvera inéluctablement une version actualisée de la ressource pendant l'état intermédiaire $fetch^{ew}$. Donc, le *handle* n'aura plus à occuper l'état *push* .

   La table 6.1 expose par ordre hiérarchique, une combinaison d'états possibles occasionnés par des évènements successifs déclenchés par les fonctions **DHO**, tandis que la figure 6.2 montre les intéractions entre les processus de chaque *peer* ainsi que le canal de dialogue entre deux *peers* différents.

   Par exemple l'ensemble $\{Req, fetch^{ew}, Busy\}$ atteste que le *peer* a émis une requête pour une ressource qui est en phase de chargement en mémoire locale, par le *gestionnaire de ressource*. De plus, le *gestionnaire de verrou* correspondant au même *peer* est occupé à traiter une autre requête qui lui est parvenue par l'un de ses *fils*.

6.3.3 Déroulement d'un exemple pour un accès exclusif

Nous reprenons l'exemple 4.1 du chapitre 4. Cet exemple illustre des *peers* auxquels sont assignés deux états ; l'un pour le *handle* et l'autre pour le *gestionnaire de verrou*. Par exemple, la paire d'états ($lock^{ew}$/*Busy*) traduit les faits suivants : Le *peer* est en **section critique** et dispose d'un accès exclusif à la ressource. En même temps, le *gestionnaire de verrou* associé traite une demande qui lui a été adressée par l'un de ses *fils* dans la

branche parentale.

Initialement, tous les *peers* sont connectés au *peer* p_1 en **section critique**. Ce dernier est leur ***Parent***. Par conséquent, tous les *peers* ont un *handle valid* mais sont encore inactifs (figure 6.3(a)).

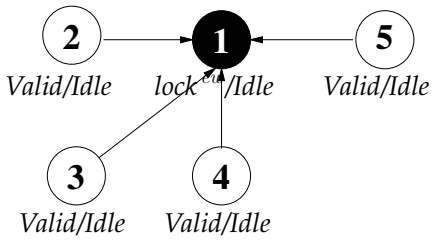
Dans la figure 6.3(b), p_2 appelle la fonction `dho_ew_request`, le *gestionnaire de ressource* dirige la requête vers p_1. À son tour, p_3 fait appel à la même fonction pour solliciter la ressource (figure 6.3(c)) puis, à la fonction `dho_ew_acquire`. Par la suite, p_3 devient *blocked* ^{ew} (figure 6.3(d)).

La *racine* p_1 libère la ressource par l'appel de la fonction `dho_ew_release` et p_2 entre en **section critique** (figure 6.3(d)). A son tour, p_4 réclame la ressource dont l'appel est dirigé vers p_1 (figure 6.3(e)).

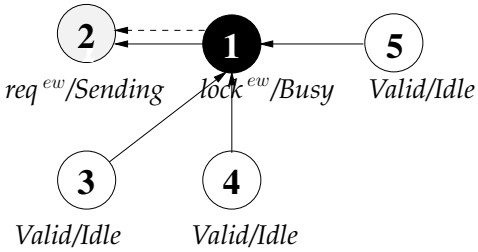
Finalement, p_2 libère la ressource et p_3 entre en **section critique**. Au même moment, p_4 appelle la fonction `dho_ew_acquire` et devient bloquant (figure 6.3(f)).

Nous signalons que certaines phases ont été omises. Par exemple le passage du *handle* à l'état intermédiaire *fetch* ^{ew} n'est pas représenté. Aussi, nous rappelons que tous les *gestionnaires de verrou* sont contraints de passer par l'état *Idle* avant d'occuper, celui de *Busy* ou de *Sending*.

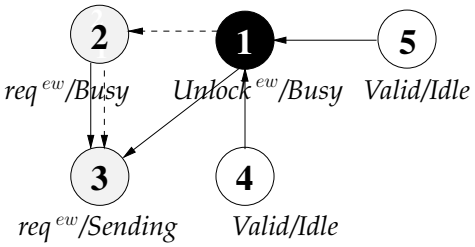
Par ailleurs, par souci de lisibilité, les fonctions **DHO** ne sont pas représentées dans cette figure mais nous les reportons dans la figure 6.4. Nous illustrons à titre indicatif la figure 6.3(e).



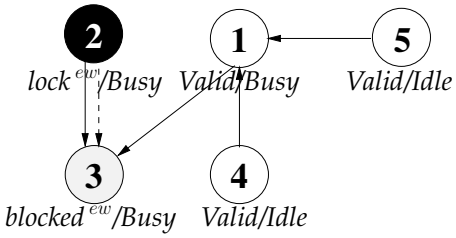
(a) p_1 est *racine* et ***Parent*** de tous les *peers*, tous les *handles* sont *Valid*



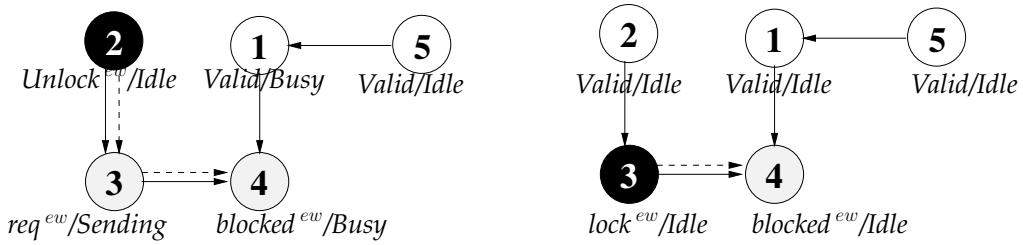
(b) p_2 envoie une requête à p_1



(c) p_3 envoie une requête à p_1



(d) p_1 libère la ressource, p_2 entre en **section critique** et p_3 passe au mode bloquant



(e) p_4 envoie une requête à p_1

(f) p_2 cède le jeton à p_3, p_4 devient bloquant

FIGURE 6.3 – Déroulement d'un exemple et illustration des étapes traversées par la paire d'états {*handle*, *gestionnaire de verrou*}

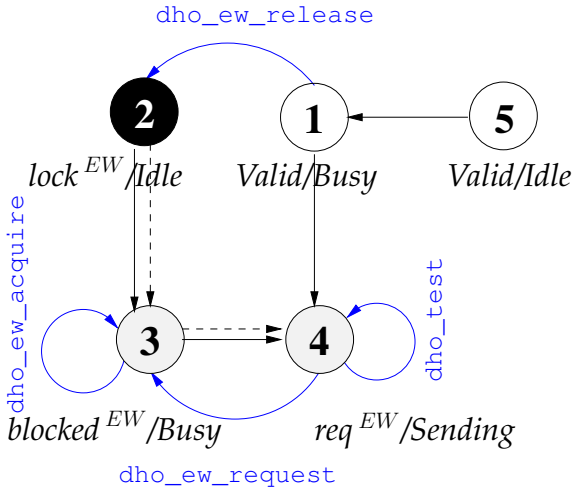


FIGURE 6.4 – Représentation des fonctions **DHO** de l'étape (e) de la figure 6.3

6.3.4 Mobilité des *peers*

Notre système *peer-to-peer* est subordonné à l'**ADEMLE** au plus bas niveau de la structure, du moins pour les *locks* exclusifs. Les modalités de départ des *gestionnaires de verrou* sont régies par cet algorithme (voir la section 4.3.2).

En fait, l'avènement de départ du *gestionnaire de verrou* résulte de l'appel d'une fonction **DHO** (`dho_destroy`) insérée par l'utilisateur dans le code applicatif. Le *gestionnaire de verrou* reçoit un signal du *gestionnaire de ressource* du même *peer*.

Nous présentons dans cette section les dispositions de déconnexion et de connexion d'un *peer* (p_j) du système.

peer	*Idle*				*Req*							*Blocked*		*Locked*	
handle	*valid*		*Unlock*		*req* ew			*grant* ew		*fetch* ew		*blocked* ew		*locked* ew	
g.verrou	*Idle*	*Busy*	*Idle*	*Busy*	*Idle*	*Sending*	*Busy*	*Idle*	*Busy*	*Idle*	*Busy*	*Idle*	*Busy*	*Idle*	*Busy*

TABLE 6.1 – États relatifs à l'ensemble {*peer*, *handle*, *gestionnaire de verrou*} de notre système distribué.

6.3.4.1 Déconnexion des *peers*

Nous avons vu dans la section 3.2.3 que l'utilisateur n'est pas tenu de respecter l'enchaînement classique du cycle **DHO** pour l'acquisition d'une ressource donnée. L'appel de la fonction `dho_destroy` est l'un des moyens qui permet de braver les règles. Usuellement, cette fonction est insérée dans le code lorsque l'application n'a plus besoin de la ressource représentée par son *handle*. De ce fait, toutes les informations relatives à la ressource sont perdues suite à la destruction du *handle*.

L'appel à cette fonction peut se faire juste après celui de `dho_ew_request` par exemple, préalablement à `dho_test` ou `dho_ew_release`.

Ainsi, lorsque l'utilisateur a recours à `dho_destroy`, le *gestionnaire de ressource* est saisi par cette demande et ouvre un dialogue avec le *gestionnaire de verrou*. Ce dernier s'il est *Idle*, vérifie que les conditions requises pour un tel départ sont bien satisfaites (section 4.3.1 du chapitre 4). Par la suite, il se déclare *Exiting*. Le départ proprement dit devient effectif par l'exécution des lignes de code de l'algorithme 4.5.

Dès que le *gestionnaire de verrou* s'annonce *Exiting*, si le *peer* est en **section critique**, le *gestionnaire de ressource* procède à l'acheminement de la ressource vers un autre *peer* potentiel du voisinage, le *Suivant* ou le *Précédent* le cas échéant. À ce moment là, le *handle* est détruit et devient *Invalid*, le *peer Exit*.

6.3.4.2 Connexion d'un *peer*

Nous avons vu dans la section 3.2 du chapitre 3 que la fonction `dho_create` sert à créer une *socket* avec le serveur hébergeant la ressource. Elle ne participe pas au cycle de vie **DHO** mais est généralement insérée en premier dans le code, lorsque l'utilisateur souhaite ultérieurement réclamer la ressource. En retour, elle renvoie la taille de cette dernière.

Dans le cas de notre système distribué, la ressource n'est pas localisée du fait de l'absence d'un serveur centralisé.

Donc, à la suite de l'appel de `dho_create` qui prend comme argument le nom de la ressource, le *gestionnaire de ressource* contacte le *gestionnaire de verrou* du même *peer* pour choisir arbitrairement un *peer* parmi ceux connectés au système. Ce *peer* sera le ***Parent*** de p_j dans la structure arborescente. Le ***Parent*** n'héberge pas forcément la ressource. Par contre, il détient les informations nécessaires pour l'acquisition de cette dernière.

Ensuite, le *gestionnaire de ressource* crée une *socket* avec ce ***Parent*** qui lui renvoie la taille de la ressource. Ainsi, le *handle* associé à p_j devient *valid*.

Notons que la fonction `dho_create` ne traduit pas une demande en soi pour l'acquisition de la ressource. Elle sert juste à associer un *handle* à une ressource donnée, connue uniquement par son nom ou son identifiant. Cette opération est abstraite et donc totalement transparente pour l'utilisateur.

6.3.5 Gestion des requêtes mixtes

L'algorithme dynamique d'exclusion mutuelle pour des requêtes exclusives et partagées (**ADEMEP**) remplit pleinement les objectifs visés. Il rend possible l'insertion des fonctions `dho_ew_request` ou `dho_cr_request` dans le code selon les besoins de l'application. L'utilisateur n'est pas tenu de respecter le cycle mais il en subit les conséquences.

Nous rappelons que le *gestionnaire de lecteurs* est un *gestionnaire de verrou* spécifique dont la tâche est de gérer l'accès à la **section critique** pour la chaîne des lecteurs qui lui succède et intéragir avec le *gestionnaire de ressource* du même *peer* (section 5.1 du chapitre précédent).

Le *peer* associé au *gestionnaire de lecteurs* a la particularité d'être le premier à demander une ressource en lecture à la suite d'une requête exclusive. Par ailleurs, il est le dernier à céder le jeton à un potentiel écrivain. Nous considérons le *gestionnaire de lecteurs* comme le maillon entre deux requêtes distinctes.

De façon abstraite et totalement transparente pour l'utilisateur, nous décrivons ci-dessous ce qu'il advient du *gestionnaire de ressource* et du *gestionnaire de verrou* pour le cas de ce *peer* et de la chaîne des lecteurs.

- Le *gestionnaire de verrou* est le premier à savoir qu'il conduit une liste de lecteurs. Cependant, il avise aussitôt son *gestionnaire de ressource* qui enregistre cette information dans le *handle*.
- Le *gestionnaire de ressource* est informé de l'arrivéée d'un écrivain (***prochain écrivain***) dans la chaîne (*Suivant*, *Précédent*) et l'enregistre dans le *handle*. Il fait partie désormais du voisinage.
- lorsque le *gestionnaire de lecteurs* réalise le *mapping* par son *gestionnaire de ressource*, il contacte aussitôt son *Suivant* pour la même action, qui à son tour opère pareillement. Le processus se poursuit tel une réaction en chaîne jusqu'au dernier lecteur.
- Nous savons que le *gestionnaire de lecteurs* garde le jeton tant que la **section critique** est occupée par au moins un lecteur. L'introduction du *gestionnaire de lecteurs* dans le système impose l'adjonction d'un nouvel état au *handle* spécialement pour ce *peer*.
  Donc, plutôt que d'affecter la valeur *Unlock* au *handle* lorsque le *gestionnaire de lecteurs* libère la **section critique**, son *gestionnaire de ressource* lui réserve l'état *RhUnlock* . Cet état montre que le *peer* en tête d'une chaîne de lecteurs a libéré la ressource mais, un des lecteurs au moins, demeure en **section critique**.
- Le passage à l'état *Unlock* se fait à la réception d'un signal du *gestionnaire de ver-*

rou, suite à l'annulation du *nombre de lecteurs*.
– le *gestionnaire de ressource* enregistre à nouveau cet état dans le *handle*. Il est désormais prêt à recevoir la demande du *locking/mapping* d'un potentiel ***prochain écrivain***

Départ du *gestionnaire de lecteurs*, du *prochain écrivain* ou d'un quelconque *peer* de la chaîne des lecteurs Les cironstances de départ reportées de ces *peers* sont reportées dans les sections 5.3.1, 5.3.2 et 5.3.3 du chapitre précédent prennent part dans notre système. Au niveau applicatif, l'utilisateur insère simplement un `dho_destroy` dans son code. De toute évidence, l'utilisateur ignore totalement les rôles confiés à ces *peers*.

Comme nous l'avons précisé plus haut, c'est le *gestionnaire de verrou* qui émet le signal pour le départ effectif du *peer*. Quand au *gestionnaire de ressource*, il est impliqué pour la réalisation du *mapping* pour un éventuel *Suivant* et pour l'affectation de l'état *invalid* au *handle*.

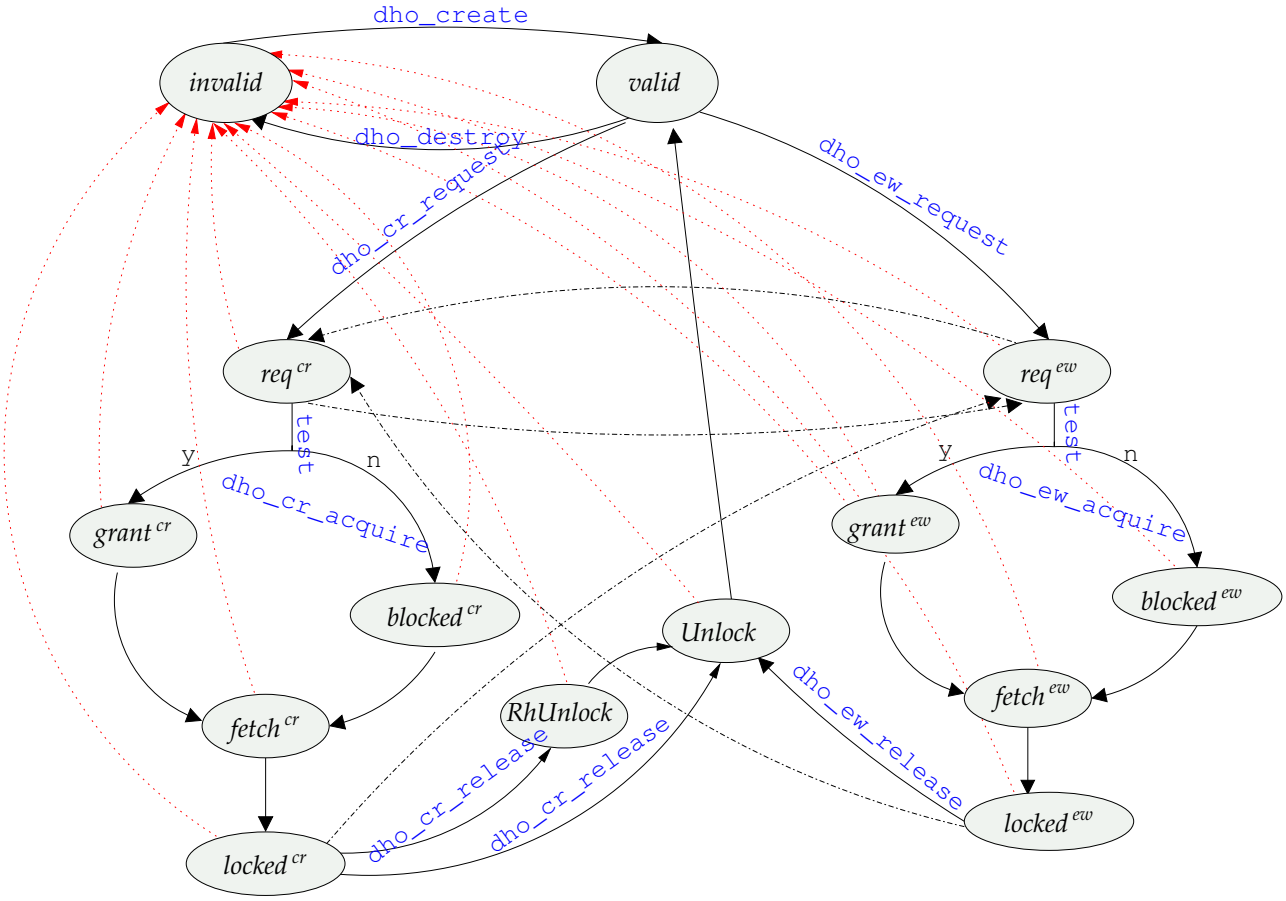


FIGURE 6.5 – Diagramme d'états d'un *handle* **DHO**

En somme, notre système requièrt l'ensemble des états suivants pour l'achèvement

d'un cycle **DHO**.

$$Q = \{valid\ ,\ invalid\ ,\ \textbf{\textit{RhUnlock}}\ ,\ \textbf{\textit{Unlock}}\ ,\ blocked^{\,\mathrm{cr}}, fetch^{\,\mathrm{cr}}, grant^{\,\mathrm{cr}}, locked^{\,\mathrm{cr}}, req^{\,\mathrm{cr}},\\ blocked^{\,\mathrm{ew}}, fetch^{\,\mathrm{ew}}, grant^{\,\mathrm{ew}}, locked^{\,\mathrm{ew}}, req^{\,\mathrm{ew}}\}$$

Aussi, la figure 6.5 reprend le diagramme d'états du chapitre 3 associé à un *handle* donné. Quelques modifications y sont apportées. Les pointillés noirs exposent le basculement d'états pour deux appels successifs et distincts, le premier étant non accompli. Exemple, `dho_ew_request` suivi de `dho_cr_request`. Les pointilliés rouges dénotent le retour à l'état initial *valid* suite à l'appel de la fonction `dho_destroy` et ce, quel que soit l'état en cours du *handle*.

6.4 Environnement expérimental

Nous présentons dans ce qui suit les moyens matériels et logiciels que nous avons déployés pour l'accomplissement des expériences.

6.4.1 Plate-forme matérielle

Grille informatique dédiée au calcul scientifique Les tests expérimentaux ont été réalisés sur la grille Grid'5000 [gri]. C'est une plate-forme scientifique mise à la disposition des chercheurs pour mener des expériences de grande échelle relatives aux applications parallèles et distribuées.

Composants de la grille Grid'5000 est en constante évolution. Actuellement, elle comprend 7244 cœurs distribués sur 1500 nœuds, répartis géographiquement sur 10 sites. Les processeurs de la grille sont des `AMD Opteron` et des `Intel Xeon` cadencés entre 1.6 et 3 Ghz. Les nœuds sont dotés d'une `RAM` dont la capacité varie entre 2 et 48 `Go`.

Chaque nœud dispose de 1 et 4 `cpus` et chaque `cpu` comprend entre 1 et 12 cœurs. Les sites sont répartis sur différentes villes françaises. Ils comprennent entre 1 et 4 *Clusters*.

Une laison `Ethernet` en fibre optique et des `VLANs` relient les différents sites de la grille au débit de 10 `Gbps`. L'ensemble de l'infrastructure réseau de Grid'5000 forme un *Backbone* fourni par le réseau national Français de télécommunications pour la technologie, l'enseignement et la recherche (Renater).

Afin d'étudier l'impact de l'hétérogénéité sur notre application, nous exploitons, à travers quelques unes de nos expériences, des nœuds répartis sur plusieurs sites. Notons cependant que la grille est sollicitée par une forte communauté de chercheurs, d'où la difficulté de trouver des nœuds disponibles en grand nombre.

6.4.2 Plate-forme logicielle

Réservation des nœuds L'exploitation des ressources de la grille se fait par une réservation préalable d'un certain nombre de nœuds. La grille est gérée par un gestionnaire de ressources *batch scheduler* dénommé : `Oargrid`.

Ainsi, la préparation des expériences nécessite de passer par cette phase. `OAR` nous permet de soumettre des jobs en mode interactif ou en mode passif, par l'utilisation de commandes telles que `oarsub` et `oargridsub`, selon que le job implore un ou plusieurs sites.

Modèle de communication Nous déployons la même base logicielle que dans nos expériences antérieures, à savoir la librairie `GRAS` de `SimGrid` basée sur les *sockets* dont la bonne performance a été démontrée (voir les sections 3.3.3 et 3.4.3 du chapitre 3).

La figure 6.6 présente la pile des modules de notre architecture. Elle traduit l'insertion de l'API **DHO** à la structure modulaire existante de `SimGrid`.

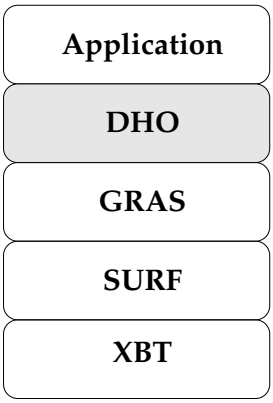


FIGURE 6.6 – Structure globale d'une application sous **DHO**

En outre, nous ouvrons une parenthèse sur un problème rencontré lors des premiers tests. Plusieurs échecs d'exécution se sont succédé, suite auxquels nos investigations nous ont amené à conclure que la libraire `GRAS` ne se prête pas encore aux environnements multi-cœurs.

En réalité, l'architecture interne de `GRAS` est `multi-threading`. Nous avons constaté que la sychronisation n'est pas garantie dans le même nœud, notamment dans le cas de notre application où deux processus sont en intéraction.

Nous avons pu éviter ce problème par la réservation d'un seul cœur par nœud. Ceci ne présente pas une baisse des performances. Des résultats analogues ont été observés suite aux tests conduits sur des processeurs *mono-cœur* et *multi-cœurs* en mode simulé.

6.4.3 Scénarios

Nous nous basons sur le même code applicatif que précédemment. Nous soulignons encore une fois que le modèle adopté constitue une couche interne et abstraite. Le *handle*

assure ce niveau d'abstraction et est pris comme argument dans toutes les fonctions **DHO**.

Le listing 6.1 présente un prototype de code pour les *locks exclusifs* dans lequel des cycles **DHO** sont accomplis. Les *peers* sont distribués sur les nœuds de la grille et chaque *peer* est représenté par deux processus, le *gestionnaire de ressource* et le *gestionnaire de verrou*. Ainsi, au lancement du script d'exécution deux processus sont générés sur chaque nœud.

Par ailleurs, le scénario d'exécution se présente comme suit : Chaque *peer* demande la ressource une centaine de fois (nombre de cycles). Toutefois, les dix premiers cycles ne sont pas pris en compte car nous voulons atteindre le régime stationnaire avant de prélever les mesures. Les expériences sont exécutées 10 fois et les mesures reportées sont déduites des moyennes.

Listing 6.1 – Invocation d'une ressource en boucle (100 fois)

```
char const* name;
dho_t *a;
double T_WBlocked, T_Lock;
dho_create(name, &a);

int cycle=1;
do {
  dho_ew_request(a);
  sleep(T_WBlocked);
  dho_test(a);
  dho_ew_acquire(a);
  sleep(T_Locked);
  dho_ew_release(a);
  sleep(T_idle);
} while(cycle < 100);
dho_destroy(a);
```

6.5 Méthodologie d'évaluation

Les choix pris pour certains paramètres concernent les éléments suivants :

- Plusieurs series de *benchmarks* sont prévues pour étudier les requêtes exclusives, partagées et mixtes.
- Le temps d'occupation de la ressource et donc de la **section critique** (T_{locked}) est diversifié. Nous étudions également l'impact du temps applicatif (T_{Wblocked}). Rappelons que cette durée exprime le délai prévu par l'application avant le passage du *peer* et du *handle* au mode bloquant.
- L'étude de l'influence des deux temps applicatifs cités plus haut tient compte de différentes tailles de la ressource.

– Les paramètres de passage à l'échelle et de l'hétérogénéité sont aussi analysés. Pour cela, nous varions le nombre de *peers* ainsi que le degré de répartition des *peers* sur les différents *Clusters* de la grille.
– L'impact de volatilité est étudié par une variation du nombre de *peers* en mouvement.

6.5.1 Durées observées

Nous nous intéressons aux applications où la ressource est fortement sollicitée, c'est-à-dire que nous n'envisageons pas un temps entre la résiliation de la ressource et une nouvelle demande, le but étant de tester le comportement de notre système dans des situations extrêmes. Ainsi T_{Idle} est *null* dans toutes les expériences.

Tout comme le modèle client-serveur, la durée du cycle **DHO** qualifie le système sur lequel il repose. En outre, la latence moyenne d'attente d'une requête (T_{Wait}) et la durée moyenne de blocage d'un *peer* (T_{Blocking}) font partie intégralement de notre étude.

Lorsque le *lock* est obtenu sans passer par le mode bloquant (voir les différents chemins du cycle dans le chapitre 3), T_{DHO} est exprimé comme suit :

$$T_{\text{DHO}} = T_{\text{WaitGrant}} + T_{\text{grant}} + T_{\text{fetch}} + T_{\text{locked}}. \tag{6.1}$$

Autrement, la durée du cycle est :

$$T_{\text{DHO}} = T_{\text{Wblocked}} + T_{\text{block}} + T_{\text{fetch}} + T_{\text{locked}}. \tag{6.2}$$

Par ailleurs, T_{Blocking} est défini par :

$$T_{\text{Blocking}} = T_{\text{blocked}} + T_{\text{fetch}}. \tag{6.3}$$

T_{Wait}, pour le chemin *non-bloquant* :

$$T_{\text{Wait}} = T_{\text{WaitGrant}} + T_{\text{grant}} + T_{\text{fetch}}. \tag{6.4}$$

Et T_{Wait}, pour le chemin bloquant :

$$T_{\text{Wait}} = T_{\text{Wblocked}} + T_{\text{blocked}} + T_{\text{fetch}}. \tag{6.5}$$

6.6 Évaluation des performances du système *peer-to-peer* pour les *locks* exclusifs

D'une part, des algorithmes distribuées d'exclusion mutuelle ont été proposés puis, validés théoriquement dans les chapitres 4 et 5. D'autre part, une première expertise du modèle de l'interface **DHO** a été présentée dans le chapitre 3.

Nous évaluons dans cette section les performances de l'interface **DHO** basée sur notre système *peer-to-peer* et les *locks* exclusifs. Comme nous l'avons précisé plus haut, nous tirons amplement profit du deuxième mode offert par `GRAS` (le mode réel).

Ressource	$\overline{T_{\text{locked}}}(s)$	$\overline{T_{\text{DHO}}}(s)$	$\overline{T_{\text{Wait}}}(s)$	$\overline{T_{\text{Blocking}}}(s)$
100 KiB	5	$3.564 \cdot 10^2$	$3.514 \cdot 10^2$	$3.518 \cdot 10^2$
	15	$5.519 \cdot 10^2$	$5.352 \cdot 10^2$	$5.469 \cdot 10^2$
	25	$18.668 \cdot 10^2$	$18.404 \cdot 10^2$	$18.418 \cdot 10^2$
1 MiB	5	$3.706 \cdot 10^2$	$3.64 \cdot 10^2$	$3.656 \cdot 10^2$
	15	$15.476 \cdot 10^2$	$15.315 \cdot 10^2$	$15.326 \cdot 10^2$
	25	$24.327 \cdot 10^2$	$24.071 \cdot 10^2$	$24.204 \cdot 10^2$
10 MiB	5	$5.313 \cdot 10^2$	$5.352 \cdot 10^2$	$5.469 \cdot 10^2$
	15	$16.653 \cdot 10^2$	$16.467 \cdot 10^2$	$16.476 \cdot 10^2$
	25	$26.091 \cdot 10^2$	$25.966 \cdot 10^2$	$26.087 \cdot 10^2$
50 MiB	5	$6.484 \cdot 10^2$	$6.421 \cdot 10^2$	$6.433 \cdot 10^2$
	15	$18.470 \cdot 10^2$	$18.308 \cdot 10^2$	$18.320 \cdot 10^2$
	25	$29.732 \cdot 10^2$	$29.466 \cdot 10^2$	$29.482 \cdot 10^2$
100 MiB	5	$6.644 \cdot 10^2$	$6.582 \cdot 10^2$	$6.594 \cdot 10^2$
	15	$18.903 \cdot 10^2$	$18.753 \cdot 10^2$	$18.74 \cdot 10^2$
	25	$31.612 \cdot 10^2$	$31.35 \cdot 10^2$	$31.362 \cdot 10^2$
1 GiB	5	$8.429 \cdot 10^2$	$8.367 \cdot 10^2$	$8.379 \cdot 10^2$
	15	$20.596 \cdot 10^2$	$20.434 \cdot 10^2$	$20.446 \cdot 10^2$
	25	$32.385 \cdot 10^2$	$32.123 \cdot 10^2$	$32.135 \cdot 10^2$

TABLE 6.2 – Moyennes des durées observées (système *peer-to-peer*)

6.6.1 Paradigme Client-serveur vs système *peer-to-peer* en l'absence de retard au blocage

Nous visons dans cette première serie de *Benchmarks* à mesurer le délai du cycle (T_{DHO}) et des durées qui le composent (T_{Wait} et T_{Blocking}). Nous poursuivons par une étude comparative entre les deux systèmes déployés au dessous de l'API **DHO**.

Pour cela nous prenons 120 *peers* d'un même *Cluster* en concurrence d'accès exclusif à une ressource. Les exécutions diffèrent par la durée du *lock* ainsi que par la taille de la ressource. Cependant, le retard applicatif T_{Wblocked} est négligé dans cette partie des tests ; il sera considéré ultérieurement dans ce chapitre.

D'une part, la table 6.2 et la figure 6.7 montrent que, pour chaque durée du *lock*, les durées qui composent le cycle sont rallongées par le temps T_{fetch} qui croît proportionnellement à la taille de la ressource. En revanche, les courbes des figures 6.7(a), 6.7(b) et 6.7(c) prennent des allures différentes en terme de croissance.

D'autre part, nous remarquons que pour des tailles relativement moyennes (10, 50 et 100 MiB), les deux courbes tendent à se superposer pour indiquer des valeurs très proches de T_{DHO} dans les deux modèles. Autrement, dans les cas extrêmes, le système

peer-to-peer affiche des valeurs inférieures. Pour en étudier les raisons, reprenons l'expression du temps moyen (T_{DHO}) (équation 3.13 du chapitre 3) :

$$\overline{T_{\text{DHO}}} = [\overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \overline{T_{\text{push}}} + \alpha \cdot \eta \cdot L](\overline{q} + 1) \tag{6.6}$$

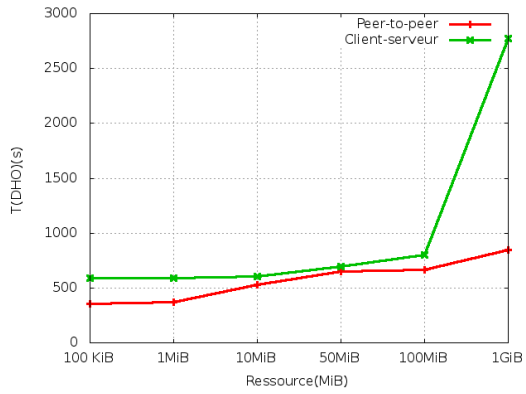
Où α = 10.4 et $\eta \simeq 7$. Cette formule a été déduite précédemment dans le cas du paradigme client-serveur. Le facteur $\overline{q}$ désigne la taille de la queue, il comptabilise le nombre de requêtes en attente. Pour le cas du système *peer-to-peer*, ce paramètre fait référence à la deuxième structure de notre algorithme **ADEMLE** à savoir la liste doublement chaînée (*Suivant*, *Précédent*).

Premièrement, le temps $\overline{T_{\text{push}}}$ n'est plus compté et ne fait plus partie du cycle, car le *peer Suivant* s'acquiert toujours une version actualisée de la ressource (étape 8 de la section 6.3.2).

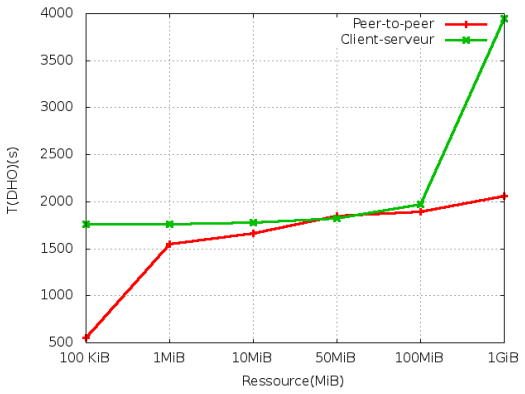
Par contre, le facteur η qui, rappelons le, désigne le retard induit par les échanges entre le *gestionnaire de ressource* et le *gestionnaire de verrou*, se montre 6 fois plus important dans le système *peer-to-peer*. De toute évidence, les échanges entre *peers* sont plus importants. La transformation partielle des deux structures : L'arbre ***Parent*** et la liste chaînée nécessitent l'envoi de plusieurs messages entre les *peers* et les membres de leur voisinage correspondants.

En outre, la taille de la queue est largement diminuée. En réalité, bien que le temps d'attente T_{wait} pour l'accès à la **section critique** soit subordonné à la taille de cette queue, en l'occurence de la chaîne (*Suivant*, *Précédent*), le traitement des requêtes n'est plus sérialisé étant donné la répartition de la charge entre les deux gestionnaires. Ceci explique cette diminution.

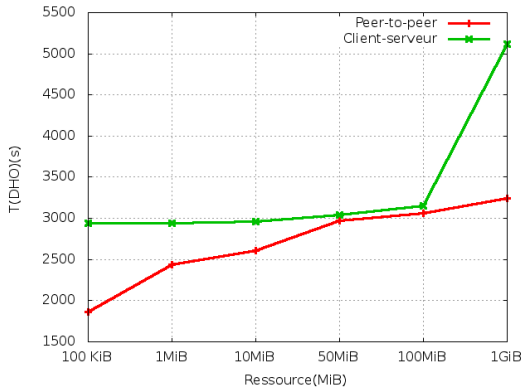
D'après nos calculs, alors qu'elle est d'une longueur constante dans le paradigme client-serveur (118), la taille de la queue varie entre 49 et 116 dans le système *peer-to-peer*.



(a) Temps moyen $\overline{T_{\text{DHO}}}$ pour 5(s) en **section critique**



(b) Temps moyen $\overline{T_{\text{DHO}}}$ pour 15(s) en **section critique**



(c) Temps moyen $\overline{T_{\text{DHO}}}$ pour 25(s) en **section critique**

FIGURE 6.7 – Comparaison du temps moyen du cycle $\overline{T_{\text{DHO}}}$ entre le modèle centralisé et le système *peer-to-peer*, en variant la taille et le temps d'occupation de la ressource, T_{locked}.

Par ailleurs, l'écart entre les durées moyennes du cycle est important pour les ressources de grande taille (1 Go par exemple). Notre interprétation est la suivante :

Nous savons que le paradigme client-serveur mobilise le seul *gestionnaire de verrou* dans toutes les opérations de *mapping*. Pendant ce temps, les autres requêtes attendent d'être servies une à une. Ainsi, un temps T_{fetch} considérable contribue à l'augmentation du temps de service dans ce modèle.

Au contraire, hormis la gestion distribuée de la ressource, la double fonction du *peer* fait qu'il peut traiter une requête en même temps qu'il peut accomplir un *locking / mapping* par exemple. La combinaison d'états *fetch* $^{\text{ew}}$ */ Busy* relative au couple (*handle*, *gestionnaire de verrou*) reflète une activité conjointe du *gestionnaire de ressource* et du *gestionnaire de verrou*.

Notre système *peer-to-peer* produit donc un recouvrement (*Overlapping*) entre le *gestionnaire de ressource* et le *gestionnaire de verrou* du même *peer* dans pas mal de circonstances, à l'instar des ressources de grande taille.

Aussi, le gain ressenti pour les ressources de taille minimale se rapporte d'une part, au temps négligé T_{fetch} et d'autre part, à une longueur de queue plus réduite que dans le modèle centralisé.

Le temps T_{DHO} s'exprime désormais comme suit :

$$\overline{T_{\text{DHO}}} = [\overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \alpha \cdot \delta \cdot L](\overline{q} + 1) \tag{6.7}$$

Où $\delta \simeq 43$

$\overline{q}$ est la longueur de la chaîne (*Suivant*, *Précédent*)

$\alpha = 10.4$

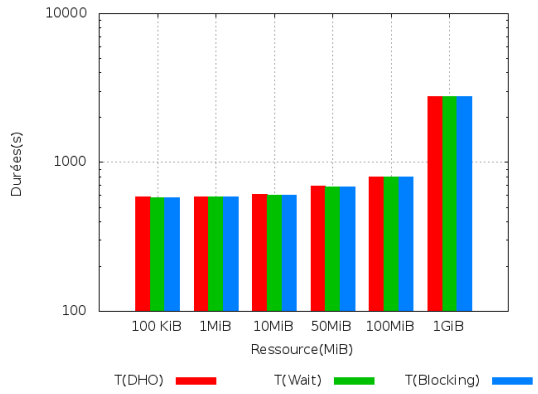
Nous déduisons que notre système est stable puisque le rapport entre le taux d'arrivée moyen (λ) et le taux moyen de service (μ) $\rho = \frac{\lambda}{\mu}$ est < 1 dans tous les cas ($\rho < 1$). Notons que le temps moyen de service pour notre système est défini par :

$$\frac{1}{\mu} = \overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} \tag{6.8}$$

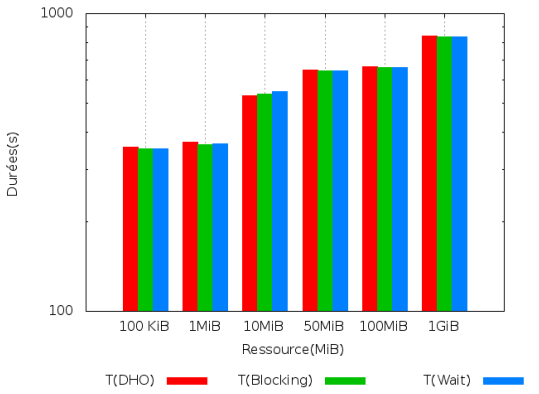
Tandis que ρ est : $\frac{(\overline{q}+1)\mu}{\overline{T_{\text{DHO}}}}$.

Les diagrammes en bâtons de la figure 6.8) exposent les écarts entre les moyennes des durées T_{DHO}, T_{Wait} et T_{Blocking} pour les deux systèmes. Nous remarquons que les temps d'attente d'une requête et de blocage d'un *client* avoisinent le temps T_{DHO} dans le système centralisé. En effet, le blocage des *clients* prend une grande part dans le cycle, et les requêtes séjournent plus longtemps dans la file d'attente.

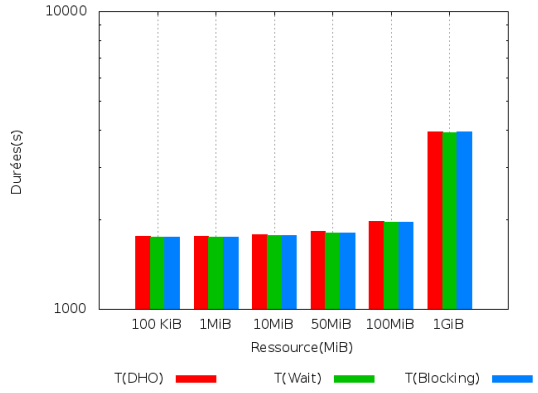
En outre, l'écart observé entre T_{Wait} et T_{Blocking} s'étend légèrement pour le cas du système *peer-to-peer*. Nous l'expliquons par le fait que le chemin *non-bloquant* n'est pas écarté du cycle (voir l'équation 6.4 et la section 6.5.1 de ce chapitre), contrairement au modèle client-serveur qui n'adopte que le chemin 3 (section 3.2.3.2 du chapitre 3).



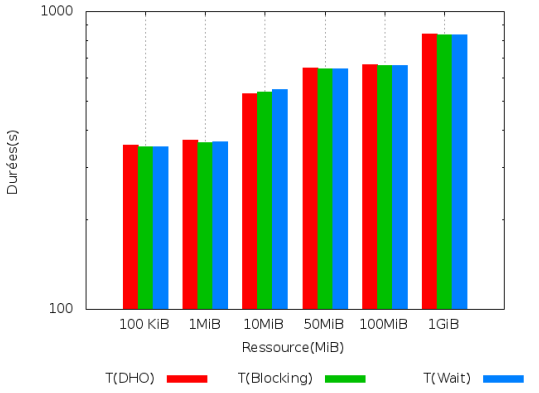
(a) Moyennes de $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et de $\overline{T_{\text{Blocking}}}$ dans le modèle client-serveur.$T_{\text{locked}} = 5s$



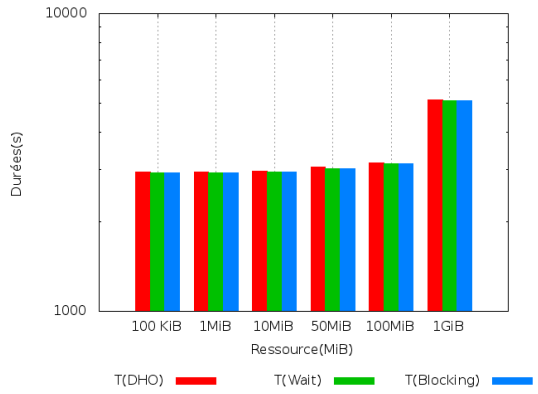
(b) Moyennes de $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et de $\overline{T_{\text{Blocking}}}$ dans le système *peer-to-peer*.$T_{\text{locked}} = 5s$



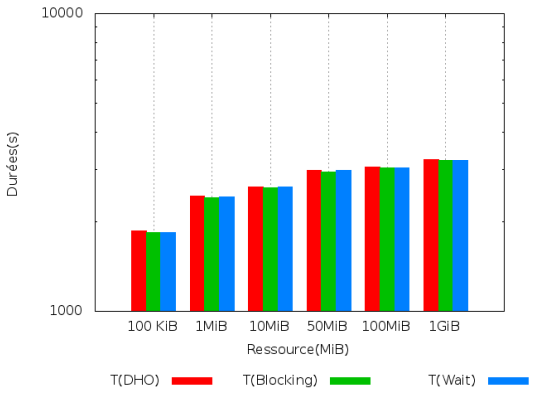
(c) Moyennes de $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et de $\overline{T_{\text{Blocking}}}$ dans le modèle client-serveur.$T_{\text{locked}} = 15s$



(d) Moyennes de $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et de $\overline{T_{\text{Blocking}}}$ dans le système *peer-to-peer*.$T_{\text{locked}} = 15s$



(e) Moyennes de $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et de $\overline{T_{\text{Blocking}}}$ dans le modèle client-serveur.$T_{\text{locked}} = 25s$



(f) Moyennes de $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et de $\overline{T_{\text{Blocking}}}$ dans le système *peer-to-peer*.$T_{\text{locked}} = 25s$

FIGURE 6.8 – Comparaison des durées moyennes $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et $\overline{T_{\text{Blocking}}}$ entre le modèle centralisé et le système *peer-to-peer* en diversifiant la taille de la ressource et le temps applicatif T_{locked}

Résultat. La distribution de la ressource sur plusieurs *peers* et le recouvrement produit par le modèle **DHO** contribuent à la réduction de la file d'attente globale du système (chaîne (*Suivant*, *Précédent*)), par conséquent à la diminution de la durée du cycle T_{DHO}, en dépit d'une latence non négligeable d'envoi de messages.

6.6.2 Impact de la répartition des *peers* sur les sites de la grille

Nous étudions maintenant l'impact de la distribution des *peers* sur les différents sites de la grille. Notons que les tests accomplis jusqu'à présent ont été menés sur un seul site, voire un seul *Cluster*. Aussi, nous prenons des mesures pour deux tailles du système *peer-to-peer* : 120 et 50 *peers*.

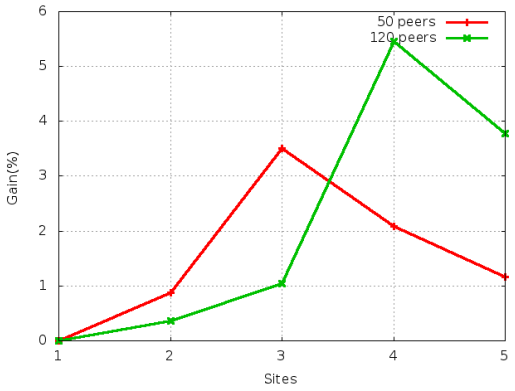
Nous observons les durées du cycle **DHO** en diversifiant le degré de répartition des *peers* sur les sites. Ainsi, les *peers* sont répartis consécutivement sur 1, 2, 3, 4 et 5 sites distants. Pour ce faire, nous prenons une ressource de 50 Mo et fixons la durée du *lock* (T_{locked}) à 5(s).

Contrairement à nos prévisions, la figure 6.9(a) montre que l'effet de bord dû aux latences *inter-sites* de la grille ne génère pas de surcoût. Au contraire, nous remarquons une légère baisse de la latence moyenne d'obtention du *lock* jusqu'à un certain degré. D'après le tableau 6.3, le gain enregistré varie entre 0.36 et 5.45 %. Nous interprétons ce résultat comme suit :

Sites	50 *peers*	120 *peers*
2	2.42(s) 0.88%	2.35(s) 0.36%
3	9.60(s) 3.51%	6.72(s) 1.036%
4	5.72(s) 2.08%	35.37(s) 5.45%
5	3.17(s) 1.56%	24.46(s) 3.77%

TABLE 6.3 – Diminution du cycle occasionné par la répartition des *peers* sur les sites de la grille

Premièrement, il est impensable d'admettre l'absence d'une latence *inter-sites*. Cette latence devrait influer directement sur le temps moyen de *mapping* ($\overline{T_{\text{fetch}}}$). Ensuite, en



(a) Gain en (%) de $\overline{T_{\text{DHO}}}$ pour 120 et 50 *peers* en fonction de la répartition des sites

FIGURE 6.9 – Gain observé suite à la répartition des *peers* sur les sites de la grille. La taille de la ressource est fixé à 50 Mo et la durée du *lock* (T_{locked} à 5(s))

obseravant l'équation 6.7 de plus près, nous déduisons que cette légère diminution ne peut provenir que de la longueur moyenne de la queue ($\overline{q}$). Ainsi, la répartition des *peers* sur un certain nombre de sites réduirait la longueur de la chaîne (*Suivant*, *Précédent*). Ce constat découle de l'activité des *peers*, voire des *gestionnaires de verrou* dans l'architecture la plus basse du système (algorithme **ADEMLE**).

Pour expliquer ce fait, reprenons la suite des événements provoqués par les instructions de notre code applicatif, spécialement au début du cycle. Initialement, le *gestionnaire de ressource* de chaque *peer* crée un *handle* avec la ressource correspondante (fonction `dho_create(name, &a)`). Pour ce faire, tous les *gestionnaires de verrou* pointent sur le même ***Parent***, c'est-à-dire sur le *peer* qui héberge la ressource et qui détient le jeton.

Examinons maintenant ce qui se produit lorsque tous les *peers* réclament la ressource. La première fonction invoquée dans le cycle (`dho_ew_request(a)`) fait que tous les *peers* afflûent sur le même *peer* ***Parent***. De ce fait, ce ***Parent*** se voit submergé de requêtes en attente dans sa propre file **Req**, particulièrement lorsque la latence entre ce dernier et l'ensemble des *peers* est négligeable (le cas d'un seul *Cluster*).

Par contre, la latence *inter-sites* produite par la distribution des *peers*, provoque un retard du flux qui écourte la longueur de la file (**Req**) du ***Parent***. Nous pouvons déduire qu'à un certain degré de répartition, les *gestionnaires de verrou* sont certes *Busy* mais moins longtemps que s'ils étaient synchronisés dans un même *Cluster*. C'est la raison pour laquelle la queue représentée par la liste chainée (*Suivant*, *Précédent*) est raccourcie, d'où un temps moyen du cycle diminué.

Résultat. Jusqu'à un certain degré de distribution, la latence *inter-sites* contribue au ralentissement de l'afflux des requêtes à destination du même *peer*, par conséquent à la réduction de la taille de sa file, donc de la latence moyenne d'attente d'une requête T_{Wait} et de la durée du cycle T_{DHO}.

De façon générale, nous pensons que le lancement asynchrone des exécutions provoquées par une fonction `random` au début des *Benchmarks*, est souhaitable pour notre application, particulièrement lorsque les *peers* sont distribués sur un même *Cluster*.

6.6.3 Évaluation du cycle DHO pour les *locks* asynchrones

Avant de commencer cette partie, nous signalons que pour la majorité de nos prochains *Benchmarks* nous prenons 50 *peers* au lieu de 120, car il nous a été de plus en plus difficile de trouver des nœuds disponibles dans la grille en grand nombre pour une longue période.

Lors des évaluations précédentes, nous avons toujours considéré une synchronisation entre l'envoi de la requête et la demande du jeton. L'absence de ce délai fait que T_{blocking} prend une bonne partie de T_{DHO} mais aussi du temps d'exécution global. Les applications qui n'envisagent pas un délai préalable à la demande effective de la ressource par la fonction `dho_ew_acquire`, sont fortement conditionnées par l'accès à la **section critique**, donc par l'acquisition de la ressource.

Nous procédons dans cette section à l'évaluation de l'impact du retard au blocage (T_{Wblocked}) pour les *locks* exclusifs. Dans notre approche, les applications sous **DHO** prévoient ce délai de sorte qu'elles puissent poursuivre leur exécution après l'envoi d'une requête (voir le cycle de vie de **DHO** dans le chapitre 3).

Le scénario prévoit la récupération de l'état du *handle* par la fonction `dho_test`. Au cas où il s'avère *grant* $^{\text{ew}}$, la fonction `dho_ew_acquire` est appelée pour passer directement à l'état *fetch* $^{\text{ew}}$ avant de basculer vers celui *locked* $^{\text{ew}}$.

L'équation suivante résume parfaitement cette situation, c'est-à-dire la récupération immédiate du *lock* en l'absence de blocage :

$$T_{\text{DHO}} = T_{\text{WaitGrant}} + T_{\text{grant}} + T_{\text{fetch}} + T_{\text{locked}}. \tag{6.9}$$

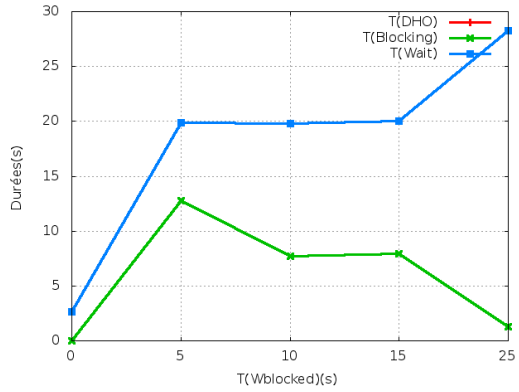
Autrement, le *peer* et le *handle* deviennent bloquants d'où :

$$T_{\text{DHO}} = T_{\text{Wblocked}} + T_{\text{blocked}} + T_{\text{fetch}} + T_{\text{locked}}. \tag{6.10}$$

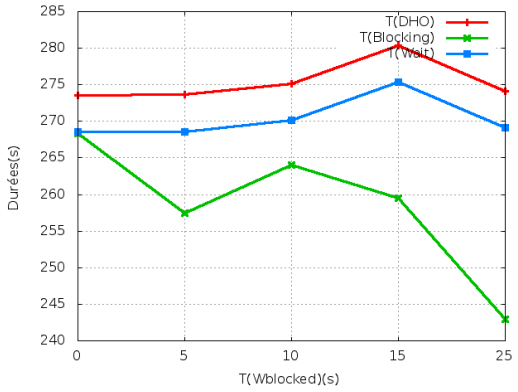
La figure 6.10 affiche les résultats obtenus suite à une diversification des durées applicatives T_{Wblocked} et T_{locked} (listing 6.1).

Pour comprendre les tracés des courbes, prenons le cas de la figure 6.10(a). Avec $T_{\text{locked}} = 0$ et $T_{\text{Wblocked}} = 0$, les *peers* demandent puis relâchent la ressource une fois le *mapping* achevé. Ce cas de figure dénote un temps de service $(1/\mu)$ équivalent à T_{fetch}.

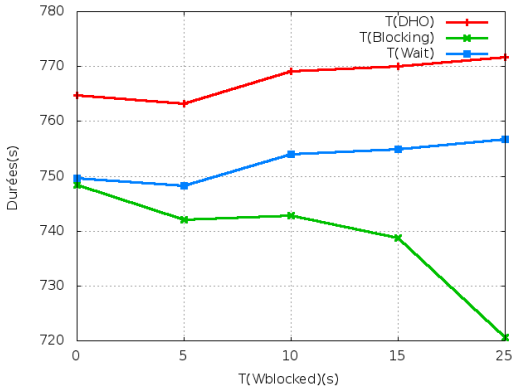
La présence de T_{Wait} pour des valeurs non nulles de T_{Wblocked} ne reflète pas un retard significatif dans la remise du jeton par le *gestionnaire de verrou*. Pour preuve, le *gestionnaire de ressource* notifie assez rapidement l'état *granted* ^{ew}. Le délai T_{Wait} atteste



(a) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 0(s) en **section critique**



(b) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 5(s) en **section critique**



(c) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 15(s) en **section critique**

FIGURE 6.10 – Observation des durées $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ et $\overline{T_{\text{Blocked}}}$ dans un système à 50 *peers* en prenant une ressource de 50 Mo.

plutôt que l'état *granted*^{ew} est prélevé postérieurement par le *handle* dans le laps de temps T_{grant}, par l'appel de `dho_test` dans l'application.

Dans le cas où $T_{\text{locked}} \neq null$ (figures 6.10(b) et 6.10(c)), nous pouvons dire qu'un retard moyen de T_{Wblocked} (5(s) par exemple) apporte un bon recouvrement à l'application. Des baisses sont enregistrées pour le temps du cycle T_{DHO} et pour la durée de blocage T_{Blocking}.

La croissance du retard au blocage T_{Wblocked} a deux incidences : (1) Tant qu'il ne prolonge pas la durée du cycle de manière conséquente, ce délai contribue à ce que le *peer* et le *handle* occupent l'état *Blocked* le moins longtemps et le plus tardivement possible. Ce retard influe positivement sur l'application qui peut évoluer pendant ce temps indépendamment des activités du *gestionnaire de ressource* et du *gestionnaire de verrou* ;

(2) espacer la fréquence d'accès à la **section critique**. Ainsi, la décroissance brutale de T_{DHO} pour des valeurs considérables de T_{Wblocked} traduit plutôt une diminution de la fréquence d'arrivée des requêtes (λ).

Nous voyons que T_{Blocking} et T_{Wait} sont reliés par l'inclusion pour un chemin du cycle (équations 6.3 et 6.5). En outre, les deux durées sont également corrélées par une relation inverse. Ce phénomène a été observé antérieurement dans le modèle client-serveur (section 3.4.6 du chapitre 3). Donc, nous ne faisons que confirmer la stabilité du modèle.

Résultat. Un léger surcoût de T_{DHO} dû à l'insertion de T_{Wblocked} ne représente pas une baisse de performances tant ce délai apporte un bon recouvrement entre le calcul et le contrôle de la donnée.

6.6.4 Évaluation du surcoût induit par la déconnexion des *peers*

Le listing ci-après expose un morceau de code dans lequel les fonctions `dho_create` et `dho_destroy` sont intégrées dans le cycle. Tous les appels **DHO** succédant à `dho_destroy` et précédant `dho_create` sont ignorés par le *gestionnaire de ressource* car, ces fonctions véhiculent un *handle* non valide (le cas de `dho_ew_acquire` et de `dho_ew_release` dans l'exemple).

Listing 6.2 – Exemple applicatif pour un peer autonome

```
dho_create(name, &a);
int cycle=1;
do {
  dho_ew_request(a);
  dho_test(a);
  dho_destroy(a);
  dho_ew_acquire(a);
  sleep(T_Locked);
  dho_ew_release(a);
  dho_create(name, &a);
} while(cycle < 100);
```

Pour ces expériences, le système est divisé en deux sous ensembles : (1) Dans l'un, les *peers* accomplissent un cycle ordinaire à l'image des autres *Benchmarks* ; (2) dans l'autre, les *peers* exécutent le code du listing 6.2 où l'on voit la destruction du *handle* puis, sa re-création au milieu du cycle, de sorte que le *peer* correspondant se déconnecte puis se reconnecte au système. Nous disons que ces *peers* sont en mouvement.

Vu le caractère inhérent des *peers* qui composent notre système, la latence produite par ce mouvement influe forcément sur les autres *peers*. Nous ne mesurons pas les durées correspondantes aux *peers* en mouvement car, le cycle est interrompu. Les répercussions de ces appels sont présentées dans la section 3.2.3 (voir les cas spécifiques).

Déconnexion	50 *peers*	120 *peers*
25%	2.27(s) 0.842%	4.27(s) 0.725%
33%	2.65(s) 0.98%	6.46(s) 1.05%
50%	4.29(s) 1.56%	10.34(s) 1.68%

TABLE 6.4 – Coût généré par le mouvement d'un sous ensemble de *peers*. Les résultats correspondent aux *peers* qui achèvent un cycle **DHO**

Nous observons plutôt les durées associées à l'ensemble des *peers* qui accomplissent un cycle classique non interrompu.

Nous avons choisi de prendre consécutivement un quart, un tiers, puis une moitié de *peers* en mouvement. Chaque test est réalisé avec 50 et 120 *peers*.

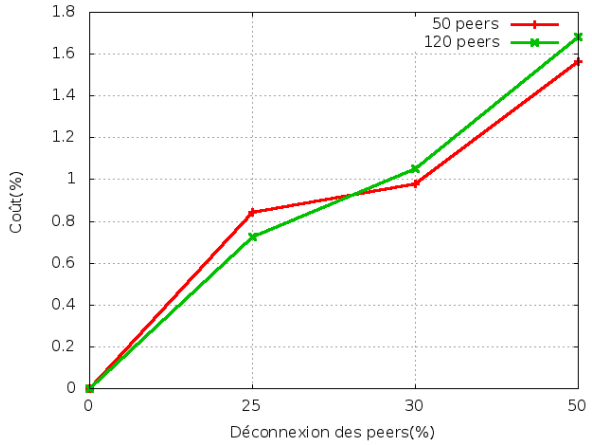


FIGURE 6.11 – Coût produit par un sous ensemble de *peers* en mouvement sur la durée moyenne du cycle **DHO**

À partir de la table 6.4 et de la figure 6.11, nous remarquons que les surcoûts sont approximatifs pour les deux tailles du système. Les latences enregistrées sont minimes bien qu'elles reflètent un départ par cycle. Nous estimons qu'un surcoût de 1.68% induit par la moitié des *peers* n'est pas nuisible à notre système. Ceci dit, nous imaginons une multiplication de ces valeurs en fonction de la fréquence de la mobilité des *peers*.

Aussi, les latences T_{create} et T_{destroy} ont déjà été évaluées dans la section 3.4.2 du chapitre 3. Rappelons que dans les pires des cas, elles sont respectivement de $145.6 \cdot 10^{-4}$ et de $72.8 \cdot 10^{-4}$. Par ailleurs, il n'existe pas de distinction entre le modèle client-serveur et le système *peer-to-peer* pour ces latences dans l'accomplissement de ces deux phases, car elles ne sont pas incluses dans le cycle.

Ces deux étapes ne nécessitent que quelques échanges entre le *peer* et son ***Parent*** par les *gestionnaires de verrou* ; Un aller-retour pour la destruction du *handle* et deux échanges lors de sa création. Reprenons les deux équations déduites précédemment :

$$T_{(create)} \simeq 2 \cdot \alpha \cdot \eta \cdot L \tag{6.11}$$

$$T_{(destroy)} \simeq \alpha \cdot \eta \cdot L \tag{6.12}$$

Où $\eta \simeq 7$ et $\alpha = 10.4$.

Pour conclure ce volet, nous pouvons attester que la dynamicité du système peut être maîtrisée en fonction du nombre des *peers* et de la fréquence de leur mobilité.

Résultat. L'inclusion des deux latences relatives à la connexion et à la déconnexion des *peers* dans le cycle modifie l'équation T_{DHO} comme suit :

$$\overline{T_{\text{DHO}}} = [\overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \alpha \cdot \delta \cdot L + \eta \cdot f \cdot p \cdot (2 \cdot T_{(create)} + T_{(destroy)})](\overline{q} + 1) \tag{6.13}$$

Où f est la fréquence de destruction du *handle* dans un cycle et p est la proportion de *peers* en mouvement dans le système.

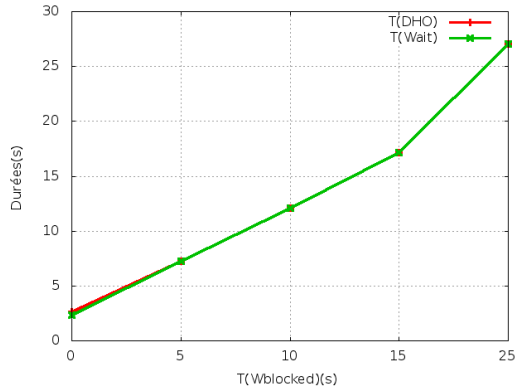
6.7 Observation du cycle DHO pour des requêtes asynchrones en accès partagé

Avant d'étudier le système avec des requêtes combinées, nous prélevons des mesures pour des accès exclusivement partagés en mode asynchrone, en diversifiant T_{Wblocked} et T_{locked}, avant d'observer T_{Wait} et T_{DHO}. Le listing 6.3 présente le bout de code dédié à ce type d'accès.

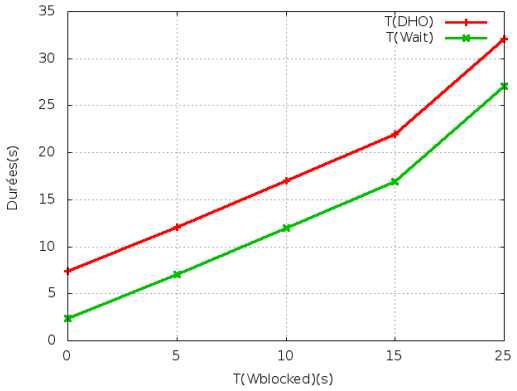
Listing 6.3 – Exemple applicatif pour des locks partagés

```
dho_cr_request(a);
dho_test(a);
sleep(Wblocked);
dho_cr_acquire(a);
sleep(T_Locked);
dho_cr_release(a);
```

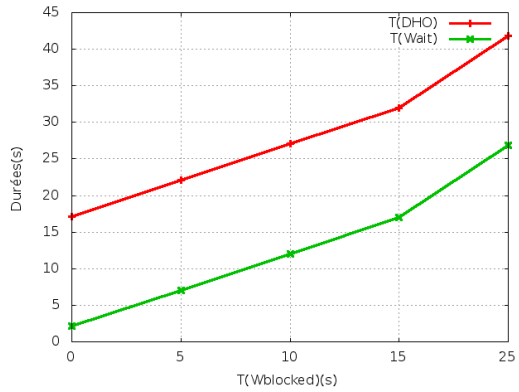
D'après la figure 6.12, on voit bien que T_{Wait} est constant malgré les variations des durées applicatives. Du fait du partage d'accès, le temps de service $(1/\mu)$ se résume à T_{fetch}. Le parcours du *handle* en termes d'états emprunte ce chemin dans le cycle qui n'enregistre aucun temps de blocage.



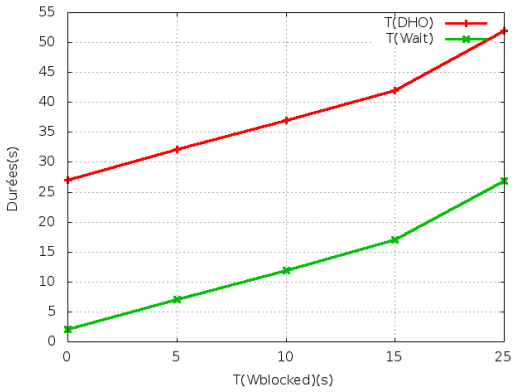
(a) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 0(s) en **section critique**



(b) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 5(s) en **section critique**



(c) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 15(s) en **section critique**



(d) Durées moyennes en fonction du délai T_{Wblocked} pour une occupation de 25(s) en **section critique**

FIGURE 6.12 – $\overline{T_{\text{DHO}}}$ et $\overline{T_{\text{Wait}}}$ pour les *locks* partagés dans un système à 50 *peers* en prenant une ressource de 50 Mo

.

$$\{req^{\,\text{cr}} \rightarrow grant^{\,\text{cr}} \rightarrow fetch^{\,\text{cr}} \rightarrow locked^{\,\text{cr}} \rightarrow valid\,\}$$

Pour toutes les requêtes émises, T_{Wait} est définit par :

$$T_{\text{Wait}} = T_{\text{WaitGrant}} + T_{\text{grant}} + T_{\text{fetch}}. \tag{6.14}$$

et T_{DHO} par :

$$T_{\text{DHO}} = T_{\text{WaitGrant}} + T_{\text{grant}} + T_{\text{fetch}} + T_{\text{locked}}. \tag{6.15}$$

En d'autres termes T_{DHO} est exprimé par :

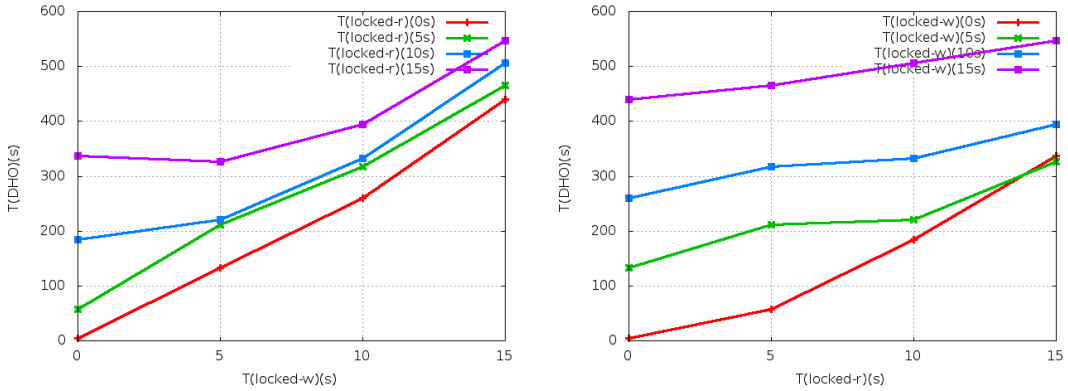
$$\overline{T_{\text{DHO}}} = [\overline{T_{\text{fetch}}} + \alpha \cdot \delta \cdot L](\overline{q} + 1) + \overline{T_{\text{locked}}} \tag{6.16}$$

Où $\overline{q}$ est quasi-nulle.

Résultat. L'absence de requêtes dans la queue nous a permis de déduire le coût des messages envoyés qui prend une bonne part dans l'acheminement d'une requête. Ce coût est d'environ $t = 2$(s). Il est couvert par le facteur δ dans l'équation 6.16 et par $T_{\text{WaitGrant}}$ ou T_{Wblocked} selon le chemin adopté dans le cycle.

6.8 Évaluation du cycle DHO pour des requêtes mixtes

Les accès exclusifs et partagés ont jusque là été analysés séparément. L'étude exhaustive de notre approche requiert l'exploration du modèle face à des requêtes distinctes de manière alternée, d'autant plus que cela représente le cas d'utilisation générale de l'API.



(a) Variation de T_{locked} dans les cycles à accès exlusif (b) Variation de T_{locked} dans les cycles à accès partagé

FIGURE 6.13 – Durées moyennes du cycle $\overline{T_{\text{DHO}}}$. Pour chaque valeur T_{locked} du premier cycle, plusieurs valeurs sont prises dans le deuxième. Les deux figurent exposent les mêmes résultats pour une ressource de 50 Mo.

Dans ces tests, les *peers* adressent des requêtes en lecture puis en écriture ou inversement sans retard au blocage (T_{Wblocked}), de manière à enchaîner deux cycles consécutifs. Au total, chaque *peer* accomplit 200 cycles. Nous diversifions le temps d'occupation de

la **section critique** T_{locked} par ordre croissant de sorte que pour chaque valeur prise dans le premier cycle, quatre autres valeurs sont prévues dans le deuxième.

La figure 6.13 affiche les mêmes moyennes de façon mutuelle pour chaque type d'accès. Nous remarquons une croissance des courbes avec une allure moins accélérée dans la figure 6.13(b) où plusieurs durées d'accès exlusifs sont prises pour la même valeur en accès partagé. Manifestement, les moyennes sont régentées par les cycles d'accès exclusifs.

Ainsi, le temps moyen de service serait $\mu = \frac{\mu_1+\mu_2}{2}$.

En dernier, nous étudions l'effet de l'asynchronisme sur le cycle, plus exactement l'effet du retard au blocage qui suit les deux types d'accès à la ressource. Comme ci-devant, le code applicatif est composé de deux cycles consécutifs. Pour ces tests, nous fixons le temps d'occupation de la ressource T_{locked} à 10(s) pour les deux types d'accès. Nous prévoyons alternativement différentes valeurs de T_{Wblocked} dans les deux cycles.

Listing 6.4 – Exemple applicatif pour des locks partagés

```
dho_cr_request(a);
dho_test(a);
sleep(Wblocked);
dho_cr_acquire(a);
sleep(T_Locked);
dho_cr_release(a);
dho_ew_request(a);
dho_test(a);
sleep(Wblocked);
dho_ew_acquire(a);
sleep(T_Locked);
dho_ew_release(a);
```

Au vu des différentes valeurs de T_{DHO}, nous observons une légère croissance du cycle. Cependant, il s'avère que les moyennes les plus élevées sont celles où les délais au blocage suivent les requêtes en lecture, et ce, dans tous les cas observés. De plus, nous enregistrons une légère baisse de T_{DHO} pour un retard de 5(s) dans les cycles d'accès exclusifs. Ainsi, pour $T_{\text{Wblocked}} = 10$, T_{DHO} est approximativement de 423(s) lorsque ce délai suit une demande d'accès exclusive, tandis qu'il est d'environ de 457(s) dans le cas où il retarde les cycles d'accès partagé.

Pour interpréter ces résultats, nous revenons à l'**AEMLEP**. La raison proviendrait de la circulation du jeton entre les *peers* de la chaîne des lecteurs, le *gestionnaire de lecteurs* et le ***prochain écrivain***. Nous savons que le *gestionnaire de lecteurs* ne cède le jeton au ***prochain écrivain*** que si tous les lecteurs ont libéré la **section critique**. Ainsi, le délai T_{Wblocked} retarderait l'acquisition de la ressource, par conséquent sa résiliation. Dans ce cas précis, nous pouvons imaginer que le *gestionnaire de lecteurs* tarde à envoyer le jeton au ***prochain écrivain***. Par ailleurs, un léger retard au blocage se révèle bénéfique s'il précède une demande de *lock* exclusif.

Donc la chaîne des lecteurs serait responsable du prolongement du temps d'attente

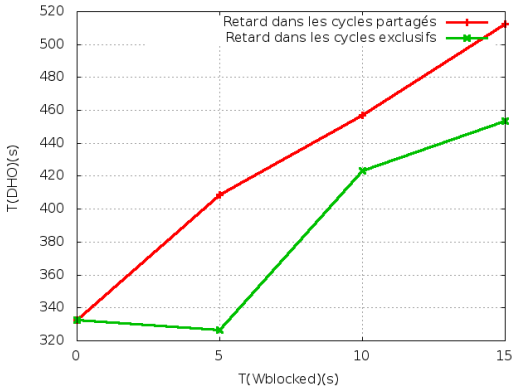


FIGURE 6.14 – Effet du retard au blocage, de part et d'autre dans les deux types d'accès. T_{locked} est fixé à 10(s) tandis que la taille de la ressource est de 50 `MiB`

d'une requête du ***prochain écrivain*** et de ceux qui le suivent dans la chaîne (*Suivant*, *Précédent*).

Résultat. Dans un système *peer-to-peer* mixte par ses types de *locks*, le surcoût généré par T_{Wblocked} est amoindri s'il est inséré dans les cycles à accès exclusif.

6.9 Synthèse du chapitre

Nous avons présenté dans ce chapitre un nouveau système *peer-to-peer* pour le partage des ressources. Les algorithmes que nous avons proposés aux chapitres 4 et 5 garantissent les propriétés indiscutables des systèmes distribués à grande échelle. La complexité de notre système ne remet pas en cause la propriété de transparence ni celle de la simplicité d'usage. Pour preuve, les *Benchmarks* sont les mêmes pour les deux modèles (centralisé et distribué).

Des algorithmes **ADEMLE**, **AEMLEP** et **ADEMLEP** dépend l'évolution interne du système, qui progresse suite à l'invocation des routines de l'API **DHO** au niveau applicatif. Notre système garantit l'acquisition de la ressource au bout d'un temps fini.

Notre modèle repose sur une intéraction permanente entre le *gestionnaire de ressource* et le *gestionnaire de verrou* qui assurent également des tâches en parallèle, en toute cohérence afin de répondre aux exigences de l'application. Dans cette approche, un recouvrement bénéfique est donc produit entre ces deux processus.

Sur une plate-forme réelle de grille informatique, nous avons conduit plusieurs series d'expériences pour lesquelles la stabilié du système a été démontrée.

Face à une multitude de variations, notre modèle a été étudié dans sa globalité ; les points suivants ont été abordés : Intensité des requêtes, hétérogénéité, passage à l'échelle, déconnexion des *peers*, temps d'occupation de la ressource et retard au blocage pour tout type de requêtes.

Chapitre 7

Conclusion générale et prespectives

Sommaire

7.1 Synthèse des travaux réalisés 127
7.2 Perspectives 128

LEs systèmes distribués tiennent actuellement une place prépondérante dans un monde où les technologies de l'information sont en évolution croissante. Généralement, la propriété de distribution est occultée pour le programmeur de l'application, qui perçoit le système comme une entité intégrée. Afin de bénéficier pleinement des services offerts par ces systèmes, des modèles adéquats doivent œuvrer pour atteindre un maximum de transparence.

Au cours de cette thèse, nous nous sommes intéressés au problème de partage et d'accès transparent aux ressources dans les systèmes à grande échelle. De l'étude présentée au **chapitre 2**, nous avons pu dégager une synthèse des différents paradigmes existants traitant de la problématique d'accès cohérent aux données. Aussi, l'état de l'art sur les algorithmes distribués d'exclusion mutuelle présentés dans le même chapitre nous a permis d'avoir une perception plus claire sur la suite de notre étude.

7.1 Synthèse des travaux réalisés

Partant de cette étude, nous avons proposé dans le **chapitre 3** l'interface *Data Handover*. **DHO** vérifie les propriétés de cohérence, de transparence d'accès et de simplicité d'usage. L'idée de base de notre interface est le niveau d'abstraction introduit entre la mémoire et la donnée par un *handle*.

L'API dispose des routines suivantes : La création et la destruction du *handle* (`dho_create`,`dho_destroy`) la réclamation d'une donnée en lecture comme en écriture (`dho_ew_request`,`dho_cr_request`), le test de l'état de la demande (`dho_test`) et l'acquisition et la résiliation de la donnée (`dho_ew_release`,`dho_cr_release`). Suite à une modélisation élaborée, nous avons présenté une méthodologie d'évaluation basée sur l'observation des retards induits par l'exercice des *clients* en quête d'une donnée partagée. L'implantation de notre

approche a été réalisée sur la base du modèle `SimGrid` et de la librairie *Grid Reality and Simulation* (`GRAS`). Le plan des expériences a couvert les deux modes prévus dans `GRAS` : La simulation et l'exécution en environnement réel, sur un *Cluster* de Grid'5000. Nous avons observé des résultats analogues dans les deux modes. Par ailleurs, les résultats recueillis ont démontré la stabilité et la performance de notre approche. Nous avons constaté que l'asynchronisme des *locks* envisagé par **DHO** apporte un recouvrement entre les calculs et le contrôle de la donnée.

Les **chapitres 4** et **5** exposent notre apport théorique sur l'algorithme de Naimi Tréhel. Il se résume comme suit :

- L'**ADEMLE** présente un algorithme d'exclusion extensible dans lequel le procédé d'acheminement des requêtes a été révisé et enrichi pour de nouvelles propriétés.
- L'**AEMLEP** introduit le jeton partagé à l'algorithme de base et présente un algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés.
- L'**ADEMLEP** présente un algorithme extensible d'exclusion mutuelle pour des accès partagés et exclusifs à une ressource. C'est un algorithme récapitulatif qui résume toutes les fonctionnalités ajoutées.

Nous avons démontré par preuve de théorème les propriétés de sûreté et de vivacité de l'exclusion mutuelle que vérifient tous les algorithmes cités ci-dessus. En outre, la complexité des messages envoyés a été étudiée et maintenue à l'échelle logarithmique.

Notre ultime contribution présentée au **chapitre 6** étale notre système *peer-to-peer* qui résulte de la juxtaposition du modèle de base de **DHO** et des algorithmes d'exclusion mutuelle proposés. Le tout forme une architecture à trois niveaux avec pour sommet les routines **DHO**. Notre approche vérifie toutes les revendications des systèmes distribués à grande échelle, à savoir : Scalabilité, transparence, hétérogénéité, autonomie des composants et simplicité d'usage au niveau applicatif.

L'implémentation a été réalisée avec les mêmes outils que précédemment et sur la même base matérielle Grid'5000. En dépit d'une variété trés large de paramètres en entrée, notre système s'est montré stable. Entre autres, les *Benchmarks* ont considéré la taille du système, les *locks* exclusifs, partagés et mixtes, la déconnexion des *peers* et le retard au blocage.

7.2 Perspectives

Parmi les fonctionnalités offertes par la fonction `POSIX fcntl`, le mode *Advisory* autorise les *locks* de segments d'un même fichier par plusieurs processus. En réalité, cette propriété est clairement revendiquée par notre API [Gus06]. **DHO** devrait autoriser les *locks* par tranches d'octets d'un *offset* car normalement **DHO** cible en grande partie les ressources de grande taille. Nous pensons donc doter notre API de cette fonctionnalité. Notons qu'un travail a déjà été réalisé dans ce sens [QV09b].

D'un autre côté, notre étude a focalisé sur le partage d'une seule ressource. Il serait intéressant de généraliser le concept de partage pour k ressources. Les expériences ont été menées sur la base de *Benchmarks*. Pour la suite, des applications réelles devraient utiliser notre API pour une validation plus complète.

Glossaire

Symboles

(Suivant,Précédent) Une file d'attente doublement chaînée de requêtes en attente. 63

DHO Data handover. 7

A

ADEMLE Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs. 8

ADEMLEP Algorithme dynamique d'exclusion mutuelle pour les *locks* exclusifs et partagés. 8

AEMLEP Algorithme d'exclusion mutuelle pour les *locks* exclusifs et partagés. 8

C

Cycle de vie DHO Le cycle commence dès lors que la demande d'un *lock* pour une ressource donnée est envoyée par une routine de l'API et s'achève par la libération de la ressource. 36

F

Fils Les processus qui ont pour p comme ***Parent*** sont les processus dits *fils* de p. 70

G

Gestionnaire de ressource Gestionnaire de ressource rattaché au *handle*, il prend en charge la requête et négocie son instanciation en mémoire locale. 36

Gestionnaire de verrou Gestionnaire de verrou associé à une ressource donnée. 36

Gestionnaire des lecteurs Dans la chaîne (*Suivant*,*Précédent*) à plusieurs requêtes suspendues, ce processus prend la tête d'une succession de lecteurs (Algorithmes **AEMLEP** et **ADEMLEP**). 77

GRAS Grid reality and simulation. GRAS est une librairie de SimGrid basée sur les Sockets BSD qui offre deux modes d'exécution : En simulation et en environnement réel. 8

H

Handle Objet utilisé comme argument dans toutes les routines de **DHO**. Il sert à fournir une abstraction entre la donnée et la mémoire. 35

L

Locking Verrouillage d'un objet dans une section critique dans pour y accéder en lecture ou en écriture. 7, 8

M

Mapping Instanciation de la donnée en mémoire locale. Le *locking* et le *mapping* sont deux opérations inéhrentes de notre API. 7

O

Overlapping Chevauchement entre deux activités conjointes : les instructions internes de l'application et le traîtement de la requête. 49

P

Parent Chaque processus pointe vers un autre processus dit ***Parent*** dans la structure arborescente. 63

Prochain écrivain Le premier processus qui suit une chaîne de lecteurs dans la chaîne *Suivant,Précédent*. 79

S

SimGrid *Toolkit* de plusieurs APIs et interfaces pour le développement d'applications parallèles et distribuées. 8

Bibliographie

[ACK+02] David P. ANDERSON, Jeff COBB, Eric KORPELA, Matt LEBOFSKY et Dan WERTHIMER : Seti@home : an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, novembre 2002.

[AjK01] James ANDERSON et Yong jik KIM : Adaptive mutual exclusion with local spinning. *In In Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, 2001.

[And04] David P. ANDERSON : Boinc : A system for public-resource computing and storage. *In Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[BAS06] Marin BERTIER, Luciana ARANTES et Pierre SENS : Distributed mutual exclusion algorithms for grid applications : A hierarchical approach. *J. Parallel Distrib. Comput.*, 66(1):128–144, janvier 2006.

[BGM95] Luc BOUGÉ, Joachim GABARRÒ et Xavier MESSEGUER : Concurrent AVL Revisited : Self-Balancing Distributed Search Trees. Rapport de recherche RR-2761, INRIA, 1995.

[CDG+] Jalel CHERGUI, Isabelle DUPAYS, Denis GIROU, Pierre-François LAVALLÉE, Dimitri LECAS et Philippe WAUTELET : Message passing interface. `http://www.idris.fr/data/cours/parallel/mpi/IDRIS_MPI_cours_couleurs.pdf`.

[CDPV01] Sébastien CANTARELL, Ajoy Kumar DATTA, Franck PETIT et Vincent VILLAIN : Token based group mutual exclusion for asynchronous rings. *In ICDCS*, pages 691–694, 2001.

[CFF+95] Peter CORBETT, Dror FEITELSON, Sam FINEBERG, Yarsun HSU, Bill NITZBERG, Jean-Pierre PROST, Marc SNIR, Bernard TRAVERSAT et Parkson WONG : Overview of the mpi-io parallel i/o interface, 1995.

[CFK+99] Ann CHERVENAK, Ian FOSTER, Carl KESSELMAN, Charles SALISBURY et Steven TUECKE : The data grid : Towards an architecture for the distribu-

ted management and analysis of large scientific datasets. *JOURNAL OF NETWORK AND COMPUTER APPLICATIONS*, 23:187–200, 1999.

[CG01] Bengt CARLSSON et Rune GUSTAVSSON : The rise and fall of napster : An evolutionary approach. *In Proceedings of the 6th International Computer Science Conference on Active Media Technology*, AMT '01, pages 347–354, London, UK, UK, 2001. Springer-Verlag.

[CGP99] Edmund M. CLARKE, Jr., Orna GRUMBERG et Doron A. PELED : *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[CL] Jalel CHERGUI et Pierre-François LAVALLÉE : Openmp multi-threaded parallelization for shared-memory machines. `http://www.idris.fr/data/cours/parallel/openmp/IDRIS_OpenMP_english_presentation.pdf`.

[CLQ08] Henri CASANOVA, Arnaud LEGRAND et Martin QUINSON : Simgrid : a generic framework for large-scale distributed experiments. *In Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.

[CR84] O.S.F. CARVALHO et G. ROUCAIROL : *Further Comments on Mutual Exclusion in Computer Networks*. Rapports de recherche / Université de Paris-Sud, Laboratoire de recherche en informatique. LRI, 1984.

[CRB+03] Yatin CHAWATHE, Sylvia RATNASAMY, Lee BRESLAU, Nick LANHAM et Scott SHENKER : Making gnutella-like p2p systems scalable. *In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 407–418, New York, NY, USA, 2003. ACM.

[CS94] Manhoi CHOY et Ambuj K. SINGH : Coping with contention (extended abstract). *In* Gerard TEL et Paul M. B. VITÁNYI, éditeurs : *Distributed Algorithms, 8th International Workshop, WDAG 94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, volume 857 de *Lecture Notes in Computer Science*, pages 89–100. Springer, 1994.

[CSG+11] Pierre-Nicolas CLAUSS, Mark STILLWELL, Stéphane GENAUD, Frédéric SUTER, Henri CASANOVA et Martin QUINSON : Single node on-line simulation of mpi applications with smpi. *In IPDPS*, pages 664–675, 2011.

[Dha06] D.M. DHAMDHERE : *Operating Systems*. McGraw-Hill Higher Education, 2006.

[Dij65] Edsger W. DIJKSTRA : Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569, 1965.

[Dij68] Edsger W. DIJKSTRA : Cooperating sequential processes. *In* F. GENUYS, éditeur : *Programming Languages : NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[FLBB79] Michael J. FISCHER, Nancy A. LYNCH, James E. BURNS et Allan BORODIN : Resource allocation with immunity to limited process failure, 1979.

[Fos01] Ian FOSTER : Grid technologies and applications : Architecture and achievements. *In in Proceedings of Computing in High Energy and Nuclear Physics (CHEP01*, 2001.

[GL96] William GROPP et Ewing LUSK : Mpich working note : The second-generation adi for the mpich implementation of mpi, 1996.

[GLDS96] William GROPP, Ewing LUSK, Nathan DOSS et Anthony SKJELLUM : A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, septembre 1996.

[GMR94] Phillip B. GIBBONS, Yossi MATIAS et Vijaya RAMACHANDRAN : The qrqw pram : accounting for contention in parallel algorithms. *In Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[GNFC00] Cécile GERMAIN, Vincent NÉRI, Gilles FEDAK et Franck CAPPELLO : Xtremweb : Building an experimental platform for global computing. *In Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, GRID '00, pages 91–101, London, UK, UK, 2000. Springer-Verlag.

[gri] Grid'5000. `http://www.grid5000.fr`.

[Gus06] Jens GUSTEDT : Data handover : Reconciling message passing and shared memory. *In* José Luiz FIADEIRO, Ugo MONTANARI et Martin WIRSING, éditeurs : *Foundations of Global Computing*, numéro 05081 de Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006.

[HGB11] Soumeya Leila HERNANE, Jens GUSTEDT et Mohamed BENYETTOU : Modeling and experimental validation of the data handover API. *In* Jukka RIEKKI, Mika YLIANTTILA et Minyi GUO, éditeurs : *GPC*, volume 6646 de *Lecture Notes in Computer Science*, pages 117–126. Springer, 2011.

[HGB12] Soumeya Leila HERNANE, Jens GUSTEDT et Mohamed BENYETTOU : A dynamic distributed algorithm for read write locks. *In PDP*, pages 180–184, 2012.

[Hol03] Gerard HOLZMANN : *Spin model checker, the : primer and reference manual*. Addison-Wesley Professional, first édition, 2003.

[hYA94] Jae heon YANG et James H. ANDERSON : A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:9–1, 1994.

[Jou98] Yuh-Jzer JOUNG : Asynchronous group mutual exclusion. *Distributed Computing*, 13:51–60, 1998.

[Jou04] Yuh-Jzer JOUNG : On quorum systems for group resources with bounded capacity. *In DISC*, pages 86–101, 2004.

[KFYA94] Hirotsugu KAKUGAWA, Satoshi FUJITA, Masafumi YAMASHITA et Tadashi AE : A distributed k-mutual exclusion algorithm using k-coterie. *Information Processing Letters*, 49:213–218, 1994.

[Lam78] Leslie LAMPORT : Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[Lan77] Gérard Le LANN : Distributed systems - towards a formal approach. *In IFIP Congress*, pages 155–160, 1977.

[Lit61] John D.C. LITTLE : A proof for the queuing formula : $L = \lambda W$. *Operations Research*, 3(9):383–387, 1961.

[mpia] Mpi-2 : Extensions to the message-passing interface. `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`.

[mpib] Mpich. `http://www.mpich.org`.

[NM94] Mitchell L. NEILSEN et Masaaki MIZUNO : Nondominated k-coteries for multiple mutual exclusion. *Inf. Process. Lett.*, 50(5):247–252, 1994.

[NSSW87] Otto NURMI, Eljas SOISALON-SOININEN et Derick WOOD : Concurrency control in database structures with relaxed balance. *In PODS*, pages 170–176. ACM, 1987.

[NT87] Mohamed NAIMI et Michel TREHEL : How to detect a failure and regenerate the token in the $\log(N)$ distributed algorithm for mutual exclusion. *In* Jan van LEEUWEN, éditeur : *WDAG*, volume 312 de *Lecture Notes in Computer Science*, pages 155–166. Springer, 1987.

[NTA96a] Mohamed NAIMI, Michel TREHEL et André ARNOLD : A $\log(N)$ distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34:1–13, April 1996.

[NTA96b] Mohamed NAIMI, Michel TREHEL et André ARNOLD : A log (n) distributed mutual exclusion algorithm based on the path reversal. *Journal of Parallel and Distributed Computing*, 34, 1996.

[ope] openmp. `http://www.openmp.or/wp`.

[Qui06a] Martin QUINSON : GRAS : a Research and Development Framework for Grid and P2P Infrastructures. *In* IASTED, éditeur : *18th IASTED International Conference on PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS*, Dallas, États-Unis, novembre 2006.

[Qui06b] Martin QUINSON : GRAS : a Research and Development framework for Grid services. Rapport de recherche RR-5789, INRIA, 2006.

[QV09a] Martin QUINSON et Flavien VERNIER : Byte-range asynchronous locking in distributed settings. *In PDP*, pages 191–195, 2009.

[QV09b] Martin QUINSON et Flavien VERNIER : Byte-range asynchronous locking in distributed settings. *In 17th Euromicro International Conference on Parallel, Distributed and network-based Processing - PDP 2009*, Weimar, Germany, 2009.

[RA81] Glenn RICART et Ashok K. AGRAWALA : An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, janvier 1981.

[Ray89a] Kerry RAYMOND : A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30(4):189–193, 1989.

[Ray89b] Kerry RAYMOND : A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, janvier 1989.

[Ray91a] Michel RAYNAL : A distributed solution to the k-out of-m resources allocation problem. *In ICCI*, pages 599–609, 1991.

[Ray91b] Michel RAYNAL : A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2):47–50, avril 1991.

[SABS05] Julien SOPENA, Luciana Bezerra ARANTES, Marin BERTIER et Pierre SENS : A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. *In Euro-Par*, pages 654–663, 2005.

[Sin89] Mukesh SINGHAL : A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Trans. Computers*, 38(5):651–662, 1989.

[Sip97] Michael SIPSER : *Introduction to the Theory of Computation*. PWS, Boston, 1997.

[SK85] Ichiro SUZUKI et Tadao KASAMI : A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4):344–349, 1985.

[SLAAS07] Julien SOPENA, Fabrice LEGOND-AUBRY, Luciana Bezerra ARANTES et Pierre SENS : A composition approach to mutual exclusion algorithms for grid applications. *In ICPP*, page 65, 2007.

[STF99] Warren SMITH, Valerie TAYLOR et Ian FOSTER : Using run-time predictions to estimate queue wait times and improve scheduler performance. *In Scheduling Strategies for Parallel Processing*, pages 202–219. Springer-Verlag, 1999.

[TGL99] Rajeev THAKUR, William GROPP et Ewing LUSK : On implementing mpi-io portably and with high performance. *In In Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, 1999.

[TN06] Ousmane THIARE et Mohamed NAIMI : %mohamed naimi + h. mohamed naimi = identisch ? a quorum-based algorithm for group mutual exclusion. *In ISCA PDCS*, pages 63–69, 2006.

[TOGBSIIS] 2004 Edition THE OPEN GROUP BASE SPECIFICATIONS ISSUE 6. IEEE STD 1003.1 : fcntl - file control. `http://pubs.opengroup.org/onlinepubs/009695399/functions/fcntl.html`.

[TSUS] Version 2. Copyright 1997 The Open Group THE SINGLE UNIX SPECIFICATION : pthread.h - threads. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html`.

[VL09] Pedro VELHO et Arnaud LEGRAND : Accuracy study and improvement of network simulation in the SimGrid framework. *In Simutools '09*, pages 1–10, Brussels, Belgium, 2009. ICST.

[WM00] Claus WAGNER et Frank MUELLER : Token-based read/write-locks for distributed mutual exclusion. *In Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 1185–1195, London, UK, 2000. Springer-Verlag.

[WWV01] Jennifer E. WALTER, Jennifer L. WELCH et Nitin H. VAIDYA : A mutual exclusion algorithm for ad hoc mobile networks. *Wirel. Netw.*, 7(6):585–600, novembre 2001.

Résumé

Data Handover est une librairie de fonctions adaptée aux systèmes distribués à grande échelle. **DHO** offre des routines qui permettent d'acquérir des ressources en lecture ou en écriture de façon cohérente et transparente pour l'utilisateur. Nous avons modélisé le cycle de vie de **DHO** par un automate d'état fini puis, constaté expérimentalement, que notre approche produit un recouvrement entre le calcul de l'application et le contrôle de la donnée. Les expériences ont été menées en mode simulé en utilisant la libraire `GRAS` de `SimGrid` puis, en exploitant un environnement réel sur la plate-forme Grid'5000. Par la théorie des files d'attente, la stabilité du modèle a été démontrée dans un contexte centralisé.

L'algorithme distribué d'exclusion mutuelle de Naimi et Tréhel a été enrichi pour offrir les fonctionnalités suivantes : (1) Permettre la connexion et la déconnexion des processus (**ADEMLE**), (2) admettre les *locks* partagés (**AEMLEP**) et enfin (3) associer les deux propriétés dans un algorithme récapitulatif (**ADEMLEP**). Les propriétés de *sûreté* et de *vivacité* ont été démontrées théoriquement.

Le système *peer-to-peer* proposé combine nos algorithmes étendus et le modèle originel *Data Handover*. Les gestionnaires de verrou et de ressource opèrent et interagissent mutuellement dans une architecture à trois niveaux. Suite à l'étude expérimentale du système sous-jacent menée sur Grid'5000, et des résultats obtenus, nous avons démontré la performance et la stabilité du modèle **DHO** face à une multitude de paramètres.

Mots-clés: *Data Handover*, cycle de vie **DHO**, gestionnaire de ressource, verrouillage en lecture/écriture, ADEMLEP, recouvrement, système dynamique.

Abstract

Data Handover is a library of functions adapted to large-scale distributed systems. It provides routines that allow to acquire resources in reading or writing in the ways that are coherent and transparent for users.

We modeled the life cycle of **DHO** by a finite state automaton and through experiments, we have found that our approach produced an overlap between the calculation of the application and the controle of the data.

These experiments were conducted both in simulated mode and in real environment (within Grid'5000). We exploited the `GRAS` library of the `SimGrid` toolkit. Several clients try to access the resource concurrently according the client-server paradigm. By the theory of queues, the stability of the model was demonstrated in a centralized environment.

We improved, the distributed algorithm for mutual exclusion (of Naimi and Trehel), by introducing following features : (1) Allowing the mobility of processes (**ADEMLE)**, (2) introducing shared locks (**AEMLEP**) and finally (3) merging both properties cited above into an algorithm summarising (**ADEMLEP**).

We proved the properties, *Safety* and *liveliness*, theoretically for all extended algorithms.

The proposed *peer-to-peer* system combines our extended algorithms and original *Data Handover* model. Lock and resource managers operate and interact each other in an architecture based on three levels.

Following the experimental study of the underlying system on Grid'5000, and the results obtained, we have proved the performance and stability of the model **DHO** over a multitude of parameters.

Keywords: *Data Handover*, **DHO** life cycle, resource handler, read/write lock, ADEMLEP, Overlapping, dynamic system.