CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique (ÉDITE)

Laboratoire CÉDRIC

THÈSE DE DOCTORAT

présentée par : **Michel SIMATIC**

soutenue le : **4 octobre 2012**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Discipline / Spécialité : **Informatique**

Contributions au rendement des protocoles de diffusion à ordre total et aux réseaux tolérants aux délais à base de RFID

THÈSE DIRIGÉE PAR

M. GRESSIER-SOUDAN Éric	*Professeur des Universités, CNAM*

RAPPORTEURS

M. DONSEZ Didier	*Professeur, Université Grenoble 1*
M. QUÉMA Vivien	*Professeur, Grenoble INP / ENSIMAG*

EXAMINATEURS

Mme DEMEURE Isabelle	*Professeur, Télécom ParisTech (Présidente)*
M. BIRMAN Kenneth P.	*Professeur, Cornell University*
M. FLORIN Gérard	*Professeur des Universités, CNAM (Invité)*
M. NATKIN Stéphane	*Professeur, CNAM*

Remerciements

You see, in every job that must be done,
there is an element of fun.
You find the fun, and snap !
The job's a game.
[Stevenson, 1964]

En rédigeant ce manuscrit, j'ai pris conscience combien une thèse est certes une personne qui rassemble des idées, mais surtout une équipe qui soutient cette personne, voire la porte, et, sans aucun doute, la supporte.

Que mon Herbe de Printemps soit ici remerciée, ainsi que les quatre magnifiques fleurs à qui nous avons donné vie ensemble et qui s'épanouissent un peu plus chaque jour.

Ils ne sont peut-être pas des fleurs (encore que...), mais quel soutien ! Je veux bien sûr parler de mes collègues : Éric du *CNAM-Cédric*, Denis, Gabriel, Djamel, Guy, Sophie, Sébastien, Chantal de l'équipe *MARGE*, Bruno, Brigitte et tous les autres membres du département *INF*, tous les autres collègues de *Télécom & Management SudParis*, et de nombreux étudiants de *Télécom SudParis* (mention spéciale pour Arthur Foltz, Damien Graux, Nicolas Hascoët, Nathan Reboud, et tous les membres de l'association Sing'INT).

Plus loin géographiquement, il me faut aussi remercier Romain Pellerin, Emmanuel Zaza, Annie Gentès, Aude Guyot, Camille Jutand, Isabelle Astic, Coline Aunis, mais aussi Didier Donsez, Vivien Quéma, Gautier Berthou. Plus loin dans le temps, il y a tous ceux qui m'ont aidé à faire tourner le train : Yves Eychenne, Bruno Kohen, Laurent Junot, Christophe Baradel, Edward Hurst-Frost, Olivier Orban, mais aussi Gérard Florin, Stéphane Natkin, Christophe Toinard, Ken Birman et bien sûr Flaviu Cristian.

Je serai incomplet si je ne mentionnais pas tous mes amis dont beaucoup font partie de l'association Vivre et Aimer : Claire et Olivier, Catherine et Damien, Marie-Odile et Jean-Paul, Catherine et Bruno, Marie-Noëlle et Thierry, Michèle et Jean-Louis, Marie-Pierre et Christian, Jeannine et Michel, Maud et Bruno, Pascale et Laurent, Anne-Marie et Bertrand, Bénédicte et François, Edouard et Marie-Dominique, Liane et Christelle, etc. Votre écoute et nos partages m'ont soutenu tout au long de ce manuscrit.

Vous qui lisez ce manuscrit, je vous ai peut-être oublié(e) dans les paragraphes précédents. Alors, soyez certain(e) que j'ai une pensée émue pour vous qui avez forcément été comme un « morceau de sucre qui aide la médecine (de la thèse) à couler ».

Et pour finir ou bien commencer à rentrer dans le vif du sujet, ma dernière pensée sera pour le Grand Barbu là-haut et son sens de l'humour presque aussi drôle que le mien...

Résumé

Dans les systèmes répartis asynchrones, l'horloge logique et le vecteur d'horloges sont deux outils fondamentaux pour gérer la communication et le partage de données entre les entités constitutives de ces systèmes. L'objectif de cette thèse est d'exploiter ces outils avec une perspective d'implantation.

Dans une première partie, nous nous concentrons sur la communication de données et contribuons au domaine de la diffusion uniforme à ordre total. Nous proposons le protocole des trains : des jetons (appelés trains) circulent en parallèle entre les processus participants répartis sur un anneau virtuel. Chaque train est équipé d'une horloge logique utilisée pour retrouver les train(s) perdu(s) en cas de défaillance de processus. Nous prouvons que le protocole des trains est un protocole de diffusion uniforme à ordre total. Puis, nous créons une nouvelle métrique : le rendement en termes de débit. Cette métrique nous permet de montrer que le protocole des trains a un rendement supérieur au meilleur, en termes de débit, des protocoles présentés dans la littérature. Par ailleurs, cette métrique fournit une limite théorique du débit maximum atteignable en implantant un protocole de diffusion donné. Il est ainsi possible d'évaluer la qualité d'une implantation de protocole. Les performances en termes de débit du protocole des trains, notamment pour les messages de petites tailles, en font un candidat remarquable pour le partage de données entre cœurs d'un même processeur. De plus, sa sobriété en termes de surcoût réseau en font un candidat privilégié pour la réplication de données entre serveurs dans le *cloud*. Une partie de ces travaux a été implantée dans un système de contrôle-commande et de supervision déployé sur plusieurs dizaines de sites industriels.

Dans une seconde partie, nous nous concentrons sur le partage de données et contribuons au domaine de la RFID. Nous proposons une mémoire répartie partagée basée sur des étiquettes RFID. Cette mémoire permet de s'affranchir d'un réseau informatique global. Pour ce faire, elle s'appuie sur des vecteurs d'horloges et exploite le réseau formé par les utilisateurs mobiles de l'application répartie. Ainsi, ces derniers peuvent lire le contenu d'étiquettes RFID distantes. Notre mémoire répartie partagée à base de RFID apporte une alternative aux trois architectures à base de RFID disponibles dans la littérature. Notre mémoire répartie partagée a été implantée dans un jeu pervasif qui a été expérimenté par un millier de personnes.

Mots clés : Systèmes répartis, Horloge logique, Vecteur d'horloges, Mémoire distribuée partagée, RFID, NFC, Diffusion uniforme à ordre total

Abstract

In asynchronous distributed systems, logical clock and vector clocks are two core tools to manage data communication and data sharing between entities of these systems. The goal of this PhD thesis is to exploit these tools with a coding viewpoint.

In the first part of this thesis, we focus on data communication and contribute to the total order broadcast domain. We propose trains protocol : Tokens (called trains) rotate in parallel between participating processes distributed on a virtual ring. Each train contains a logical clock to recover lost train(s) in case of process(es) failure. We prove that trains protocol is a uniform and totally ordered broadcast protocol. Afterwards, we create a new metric : the throughput efficiency. With this metric, we are able to prove that, from a throughput point of view, trains protocol performs better than protocols presented in literature. Moreover, this metric gives the maximal theoretical throughput which can be reached when coding a given protocol. Thus, it is possible to evaluate the quality of the coding of a protocol. Thanks to its throughput performances, in particular for small messages, trains protocol is a remarkable candidate for data sharing between the cores of a processor. Moreover, thanks to its temperance concerning network usage, it can be worthwhile for data replication between servers in the cloud. Part of this work was implemented inside a control-command and supervision system deployed among several dozens of industrial sites.

In the second part of this thesis, we focus on data sharing and contribute to RFID domain. We propose a distributed shared memory based on RFID tags. Thanks to this memory, we can avoid installing a computerized global network. This is possible because this memory uses vector clocks and relies on the network made by the mobile users of the distributed application. Thus, the users are able to read the contents of remote RFID tags. Our RFID-based distributed shared memory is an alternative to the three RFID-based architectures available in the literature. This distributed shared memory was implemented in a pervasive game tested by one thousand users.

Keywords : Distributed systems, Logical clock, Vector clocks, Distributed shared memory, RFID, NFC, Totally ordered broadcast

Avant-propos

La thèse présentée dans ce manuscrit a administrativement débuté en novembre 2009. Mais, en fait, elle est pétrie de l'expérience industrielle et académique de l'auteur qui a débuté sa vie professionnelle en septembre 1990 (cf. CV en annexe B page 229). Certains fruits de ce vécu apparaissent tels quels dans ce manuscrit. C'est le cas, par exemple, de la version monotrain du protocole de diffusion présenté en première partie et dont l'essentiel a été développé entre 1992 et 1994. Ces trois ans de thèse nous ont permis de formaliser ces résultats et de les étendre (cf. version multitrain). D'autres fruits apparaissent de manière beaucoup plus diffuse dans ce manuscrit, mais sous-tendent en fait l'ensemble de la démarche présentée. Le plus évident est la préoccupation constante de l'auteur d'exploiter au mieux les ressources disponibles et d'être en mesure d'estimer les performances d'un système dans une perspective d'implantation au sein d'une application industrielle. L'influence des 12 ans passés dans le monde industriel des télécoms et du contrôle-commande ne fait aucun doute. Mais les 10 ans passés dans le monde académique contribuent aussi largement : enseignement à des futurs ingénieurs passionnés par la technique informatique (cf. option de 3^e année que l'auteur coordonne), mais aussi recherche dans le domaine du mobile et du jeu vidéo (donc avec des ambitions importantes pour les applications, mais des ressources machines et réseaux intrinsèquement limitées).

Table des matières

1 Introduction **21**
1.1 Problématique scientifique 21
1.2 Contributions 22
1.3 Plan de thèse 23

I Communication de données dans un système réparti **25**

2 Introduction **27**

3 État de l'art des protocoles de diffusion uniforme à ordre total **31**
3.1 Spécification des protocoles de diffusion uniforme à ordre total 31
3.1.1 Définition d'un processus correct 31
3.1.2 Spécification de la diffusion uniforme à ordre total 32
3.1.3 Autres propriétés d'ordre 34
3.1.4 Conclusion 36
3.2 Classes de protocoles de diffusion uniforme à ordre total 36
3.2.1 Séquenceur fixe 36
3.2.2 Séquenceur mobile 37
3.2.3 Protocoles à base de privilège 38
3.2.4 Protocoles à base d'historique des communications 38
3.2.5 Accord des destinataires 40
3.3 Performances des protocoles de diffusion uniforme à ordre total 40
3.3.1 Performances des protocoles en l'absence de défaillances 40
3.3.2 Influence du service de gestion des participants au protocole 44
3.4 Conclusion 45

4 Protocole des trains **47**
4.1 Modèles 48
4.1.1 Modèle système 48

4.1.2 Modèle de performances 49
4.1.3 Conclusion 49
4.2 Définitions 49
4.3 Version monotrain du protocole 50
4.3.1 Présentation informelle de la version monotrain 50
4.3.2 Données 55
4.3.3 Messages 56
4.3.4 Fonctions et procédures 56
4.3.5 Algorithmes 58
4.3.6 Principes algorithmiques directement liés à l'uot-diffusion 60
4.3.7 Principes algorithmiques liés à la gestion du circuit de train 67
4.4 Version multitrain du protocole 75
4.4.1 Présentation informelle de la version multitrain 75
4.4.2 Données, messages, fonctions et procédures 80
4.4.3 Algorithmes 81
4.4.4 Principes algorithmiques 81
4.5 Conclusion 88

5 Propriétés du protocole des trains 91
5.1 Correction du protocole des trains 91
5.2 Performances du protocole des trains 105
5.2.1 Latence 105
5.2.2 Métrique de débit générique 106
5.2.3 Métrique de débit spécifique aux protocoles d'uot-diffusion 107
5.2.4 Rendement en termes de débit 109
5.3 Conclusion 117

6 Utilisation du protocole des trains pour une mémoire répartie partagée 119
6.1 Mise en œuvre du protocole des trains 119
6.1.1 Dépassement de capacité de l'horloge logique du train 119
6.1.2 Codage du circuit des trains dans l'entête du train 134
6.1.3 Régulation du débit d'émission dans le protocole des trains 135
6.1.4 Recherche du futur successeur sur le circuit des trains 136
6.1.5 Tests des performances 137
6.1.6 Améliorations envisageables 140
6.2 Gestion de groupes applicatifs 141
6.2.1 Interface de programmation du protocole des trains 142
6.2.2 Mise en œuvre de la notion de groupes 143

6.2.3 Mise en œuvre du transfert d'état . . . 146
6.2.4 Interface de programmation pour les tests de performance . . . 146
6.3 Mémoire répartie partagée basée sur de l'uot-diffusion . . . 147
6.3.1 Présentation générale . . . 147
6.3.2 Contraintes liées à la cohérence . . . 149
6.3.3 Intégration de diffusions fiables et ordonnées provenant d'entités ne participant pas à l'uot-diffusion . . . 150
6.3.4 Discussion . . . 153
6.3.5 Conclusion . . . 155
6.4 Conclusion . . . 155

7 Conclusions et perspectives **157**

II Partage de données dans un système réparti **161**

8 Introduction **163**

9 État de l'art **165**
9.1 Introduction à la technologie RFID . . . 165
9.2 Architecture centralisée . . . 167
9.2.1 Exemples d'applications . . . 167
9.2.2 Inadéquation aux contraintes du projet PLUG . . . 168
9.3 Architecture semi-répartie . . . 168
9.3.1 Exemple d'application . . . 169
9.3.2 Inadéquation aux contraintes du projet PLUG . . . 169
9.4 Architecture répartie . . . 170
9.4.1 Exemples d'application . . . 170
9.4.2 Inadéquation aux contraintes du projet PLUG . . . 172
9.5 Conclusion . . . 172

10 Mémoire répartie partagée à base d'étiquettes RFID **175**
10.1 Modèle du système . . . 175
10.2 Principes de cette mémoire . . . 177
10.3 Exemple d'application : un jeu pervasif . . . 179
10.4 Expérimentation de cette mémoire . . . 182
10.5 Discussion . . . 189
10.5.1 Travaux apparentés . . . 189
10.5.2 Analyse des limites . . . 190

10.6 Conclusion 192

11 Comparaison des différentes architectures RFID 193
11.1 Introduction 193
11.2 Critères de comparaison 194
11.2.1 Attributs fonctionnels 194
11.2.2 Attributs d'extensibilité 194
11.2.3 Attributs de coût 195
11.3 Évaluation de l'architecture centralisée 196
11.4 Évaluation de l'architecture semi-répartie 197
11.5 Évaluation de l'architecture répartie 198
11.6 Évaluation de la mémoire répartie partagée 199
11.7 Directives pour le choix d'une architecture 201
11.8 Conclusion 203

12 Conclusions et perspectives 205

Conclusion et perspectives 209

13 Conclusion et perspectives 209
13.1 Synthèse 209
13.2 Perspectives 210

Annexes 225

A Preuves des théorèmes 2, 3 et 4 225

B Curriculum Vitæ 229
B.1 État civil 229
B.2 Diplômes 229
B.3 Parcours professionnel 229
B.4 Recherche 230
B.4.1 Thématiques de recherche 230
B.4.2 Publications 230
B.4.3 Encadrement de jeunes chercheurs 233
B.4.4 Contrats de recherche 233
B.5 Enseignement 234
B.5.1 Synthèse 234

B.5.2 Détail des cours 235
B.5.3 Coordinations de cours 235
B.5.4 Encadrements de projets 236
B.6 Responsabilités 236
B.6.1 Encadrement 236
B.6.2 Gestion de contrats 236

Glossaire **239**

Liste des tableaux

6.1 Valeurs numériques de $gain_{lc\,sur\,b\,octets}$ 121
6.2 Résultats des tests sur le protocole des trains (version multitrain) 137
6.3 Comparaison entre le protocole des trains (version multitrain) et LCR . . . 140

10.1 Relation entre le taux d'indice correct de premier type et l'âge de l'indice . 184

11.1 Comparaison des architectures à base de RFID 201
11.2 Comparaison des architectures à base de RFID dans le cas du jeu PSM . . 203

Table des figures

2.1 Exemple d'architecture informatique industrielle à base de *P3200* 28

3.1 Diagramme de séquence de la contamination de processus corrects 33
3.2 Diagramme de séquence mettant en défaut la spécification d'ordre total de [Guerraoui *et al.*, 2010] 34
3.3 Diagramme de séquence montrant le nombre d'intervalles de temps requis pour acheminer les messages si les n processus multicastent 43
3.4 Diagramme de séquence montrant le nombre d'intervalles de temps requis pour acheminer les messages si les n processus exploitent au mieux les capacités du switch 43

4.1 Diagramme de séquence de la circulation d'un train entre 4 processus 52
4.2 Diagramme de séquence de l'insertion d'un processus 54
4.3 Diagramme de machine à états du protocole des trains 59
4.4 Diagramme de séquence expliquant ce qui pourrait se passer si un processus qui démarre ne se déclarait pas participant au protocole 67
4.5 Diagramme de séquence de l'insertion simultanée de deux processus dans un circuit vide (1/2) 69
4.6 Diagramme de séquence de l'insertion simultanée de deux processus dans un circuit vide (2/2) 70
4.7 Diagramme de séquence montrant pourquoi un processus non confirmé ne doit pas incrémenter l'identifiant du train 72
4.8 Diagramme de séquence montrant le suicide d'un processus qui découvre qu'il ne fait plus partie du circuit de train 74
4.9 Diagramme de séquence de circulation de plusieurs trains entre 3 processus 77
4.10 Diagramme de séquence montrant une anomalie d'ordre d'uot-livraisons quand le numéro de tour n'est pas utilisé 78
4.11 Diagramme de séquence (1/2) illustrant le rôle du numéro de tour 79
4.12 Diagramme de séquence (2/2) illustrant le rôle du numéro de tour 80
4.13 Diagramme de machine à états du protocole des trains (version multitrain) . 83

5.1 Structure d'un train dans le protocole des trains 111
5.2 Rendement du protocole des trains pour $n = 5$ processus 113
5.3 Rendements théorique et expérimental du protocole LCR pour $n = 5$ processus 115

6.1 Diagramme de séquence de la circulation d'un train avec dépassement de la capacité de l'horloge logique 124
6.2 Diagramme de séquence montrant une anomalie lors de la circulation d'un train avec dépassement de la capacité de l'horloge logique 125
6.3 Diagramme de séquence de la circulation d'un train entre n processus 126
6.4 Diagramme de séquence de la circulation d'un train entre n processus avec dépassement de la capacité de l'horloge logique du train 129
6.5 Autre diagramme de séquence de la circulation d'un train entre n processus avec dépassement de la capacité de l'horloge logique du train 130
6.6 Diagramme de séquence de la gestion du super-groupe en cas de démarrage simultané de deux processus 144
6.7 Mise en œuvre de la réplication d'objet avec de l'UTDN 148
6.8 Messages circulant entre un PLC et les répliques de P3200 152

10.1 Données présentes dans un système constitué de 2 étiquettes RFID et 3 téléphones mobiles 176
10.2 Données présente dans le jeu PSM 181
10.3 Fréquence du nombre d'étiquettes (y compris l'étiquette concernée) notifiées d'un changement 185
10.4 Boîte à moustache du temps (en minutes) requis pour qu'un certain nombre d'étiquettes soient notifiées d'un changement 185
10.5 Notion de période de validité 187
10.6 Fréquence des périodes de validité (T_{valid}) 188
10.7 Fréquence (en %) d'observation d'au plus n étiquettes périmées dans DM_{mobile} 188

Chapitre 1

Introduction

1.1 Problématique scientifique

Un système réparti est un système constitué d'un ensemble d'unités de calcul autonomes dotées de capacités de communication. Chacune de ces unités possède sa propre fonction dans l'ensemble du système. Mais, pour diverses raisons (partage de ressources, disponibilité, tolérance aux fautes, etc.), ces unités doivent coordonner leurs actions [Attiya et Welch, 2004]. Dans cette thèse, nous nous intéressons aux systèmes répartis *asynchrones*. Dans ces systèmes, le temps absolu ou même relatif auquel advient un événement ne peut pas être déterminé avec précision [Attiya et Welch, 2004]. En particulier, dans le cas où ces entités communiquent seulement par échange de messages, le délai de transmission est fini (sauf en cas de perte du message), mais imprévisible [Raynal et Singhal, 1995].

Cet asynchronisme complique fortement le développement et la preuve d'algorithmes répartis. En effet, dans un système réparti, on ne peut faire *a priori* aucune hypothèse sur l'ordre dans lequel les événements sont reçus. C'est pourquoi la communauté scientifique des systèmes répartis a introduit des outils pour déterminer l'ordre au sein de ces systèmes et donc limiter les effets de leur asynchronisme intrinsèque. Ainsi, Lamport définit la relation « Arrivé avant » et l'horloge logique [Lamport, 1978]. Cette horloge permet d'ordonner partiellement les événements advenant dans un système réparti : il devient possible d'implanter aisément un algorithme d'exclusion mutuelle réparti. Mais l'horloge logique perd des informations. Notamment elle ne permet pas à un processus de se rendre compte qu'il existe une relation de causalité entre deux événements. Par exemple, supposons qu'un processus p_0 envoie un même message m_0 à deux processus p_1 et p_2. p_1 réagit à la réception de m_0 en envoyant m_1 à p_2. Si le message m_0 met du temps à arriver au niveau de p_2, p_2 risque de prendre en compte m_1 avant m_0, ce qui peut être préjudiciable pour certaines applications. Le vecteur d'horloges est une des solutions à ce problème [Fidge, 1988, Mattern, 1988] : Elle permet de garantir le respect de la causalité. Ainsi, dans notre exemple, le processus p_2 est en mesure de retarder la livraison du message m_1 en attendant d'avoir reçu le message m_0 qui est à l'origine de ce message m_1 [Baldoni et Raynal, 2002].

L'horloge logique et le vecteur d'horloges sont exploités de deux manières différentes dans les systèmes répartis. La première manière consiste à les utiliser lors de la communication de données entre les ordinateurs du système. Ainsi, la réplication de processus basée

sur une approche *machine à états* [Schneider, 1990] requiert des propriétés additionnelles d'ordre, de qualité de livraison, etc. Cela conduit à spécifier la diffusion uniforme à ordre total (abrégé en uot-diffusion dans la suite) [Défago *et al.*, 2004, Guerraoui *et al.*, 2010]. Or, de nombreux protocoles d'uot-diffusion utilisent des vecteurs d'horloges pour respecter la causalité des messages. La deuxième manière consiste à exploiter ces outils lors du partage de données. Par exemple, le système d'exploitation *LOCUS* permet de répliquer des fichiers entre différentes machines. Pour détecter les incohérences entre les copies, notamment en cas de partition de réseau, chaque fichier est muni d'un vecteur de versions que nous considérons comme un vecteur d'horloges [Parker *et al.*, 1983].

Enfin, le passage à l'implantation de l'horloge logique et le vecteur d'horloges introduit un enjeu supplémentaire. Nous devons tenir compte des capacités limitées des machines (ordinateurs, routeurs, etc) utilisées. Ainsi, dès qu'un algorithme utilise un compteur, il est nécessaire de se poser la question du dépassement de la capacité de ce compteur sur un système industriel. Ainsi, [Attiya et Welch, 2004] remarque que le premier algorithme qu'il propose pour l'exclusion mutuelle répartie doit être revu pour tenir compte de cette contrainte physique. C'est pourquoi [Attiya et Welch, 2004] en propose une nouvelle version. De même, [Baldoni et Raynal, 2002] constate que les systèmes à base de vecteurs d'horloges sont également confrontés à ce problème pratique. En effet, les horloges des vecteurs peuvent également dépasser la capacité de la structure de données qui leur est attribuée.

Muni de l'ensemble de ces éléments, nous pouvons expliciter la problématique scientifique abordée dans cette thèse : comment exploiter au mieux l'horloge logique et le vecteur d'horloges dans des systèmes répartis avec une perspective d'implantation et de mise en exploitation industrielle ?

1.2 Contributions

Notre première contribution est le *protocole des trains*, un protocole d'uot-diffusion qui améliore le record, en termes de débit, établi par le protocole LCR [Guerraoui *et al.*, 2010]. Le principe de notre protocole reprend les idées du *protocole du train* de [Cristian, 1991], en y apportant les innovations suivantes : gestion de plusieurs trains simultanés dans l'anneau virtuel regroupant les processus participant au protocole, et gestion des participants intégrée au protocole grâce à l'utilisation d'horloges logiques. L'utilisation d'un train de messages présente plusieurs avantages par rapport au protocole LCR de [Guerraoui *et al.*, 2010] : 1) il permet de réduire la surcharge induite par le vecteur d'horloges associé à chaque message dans LCR, ce qui augmente mécaniquement le débit ; 2) il permet de regrouper naturellement les messages qu'une réplique souhaite uot-diffuser aux autres répliques, ce qui réduit le nombre d'interruptions de la machine hébergeant chaque réplique et permet donc de gagner en performances ; 3) il permet de regrouper naturellement les messages uot-diffusés entre les différentes répliques, ce qui réduit une fois encore le nombre d'interruptions de la machine hébergeant chaque réplique. Par ailleurs, comme dans LCR, nous optimisons l'utilisation du réseau en faisant circuler plusieurs messages (plusieurs trains de messages en l'occurrence) en parallèle dans le réseau. Nous proposons également un protocole de gestion des participants au protocole qui s'intègre harmonieusement à

notre protocole à base de train. En cas de défaillance d'une des répliques, l'algorithme de récupération consiste en l'émission d'un unique message. Notre protocole est donc plus performant que le protocole (à trois phases) proposé en cas de changement du groupe des participants dans l'idée initiale du train [Cristian, 1991]. Ce gain de performances peut être crucial dans le cas de certaines applications utilisant les protocoles d'uot-diffusion. De plus, cette intégration permet d'obtenir gratuitement (en termes de messages, pas en termes de temps) le caractère uniforme de l'uot-diffusion. En effet, il n'y a aucun besoin de message de validation dédié et/ou *piggy-backé*. Nous prouvons que notre protocole vérifie les propriétés attendues d'un protocole d'uot-diffusion. Nous sommes ainsi en mesure de compléter les informations concernant le protocole du train dans la synthèse [Défago *et al.*, 2004]. Nous analysons les points difficiles liés à la réalisation de ce protocole (qui est en cours d'implantation). Nous traitons notamment le problème du dépassement de capacité des horloges logiques. Notre solution s'affranchit des protocoles additionnels requis par [Yen et lu Huang, 1997] et des messages supplémentaires requis par [Baldoni, 1998].

Notre deuxième contribution est la métrique de *rendement en termes de débit* (et sa métrique associée, le *coût en termes de débit*). Cette métrique nous permet de comparer des protocoles d'uot-diffusion performants en termes de débit. De plus, elle permet de donner une limite théorique du débit maximum atteignable en implantant un protocole donné. Enfin, elle permet d'évaluer la qualité d'une implantation de protocole en positionnant les mesures expérimentales par rapport à la limite théorique.

Notre troisième contribution est une mémoire répartie partagée basée sur des étiquettes RFID (abrégée en RDSM pour RFID-*based Distributed Shared Memory*). S'appuyant sur des vecteurs d'horloges, cette mémoire est originale à deux points de vue : 1) même si elles n'ont aucune capacité de calcul, les étiquettes NFC utilisées dans l'architecture sont partie prenante dans la RDSM ; 2) ce sont les utilisateurs du système réparti qui font office de réseau de transport d'information pour cette RDSM. Nous positionnons cette RDSM par rapport aux architectures basées sur de la RFID et trouvées dans la littérature. Ainsi, nous proposons des directives pour choisir l'architecture à base de RFID la plus adaptée à une application donnée.

1.3 Plan de thèse

Dans une première partie, nous nous concentrons sur la communication de données dans un système réparti. Le chapitre 2 décrit le projet pour lequel ont été effectués nos travaux, donc les contraintes du système réparti cible : faibles répercussions sur l'utilisation des ressources (CPU, mémoire et réseau), charge induite par notre logiciel équilibrée entre les machines du système, et temps de récupération de défaillance d'une machine réduit. La communication de données qui a été développée s'appuie sur un protocole d'uot-diffusion. Le chapitre 3 dresse l'état de l'art de ces protocoles. Aucun ne répond de manière satisfaisante aux contraintes que nous avons posées. Aussi, le chapitre 4 présente le protocole des trains que nous avons conçu. Ensuite, le chapitre 5 prouve que le protocole des trains est effectivement un protocole d'uot-diffusion. Par ailleurs, ce chapitre évalue les performances théoriques du protocole, ce qui nous mène à spécifier et utiliser une nouvelle métrique : le rendement en termes de débit. Puis, le chapitre 6 décrit l'utilisation du protocole des

trains pour une mémoire répartie partagée. Il décrit les points difficiles de l'implantation du protocole des trains, et l'intégration des uot-diffusions à la mémoire répartie partagée utilisée dans le cadre de notre projet. Enfin, le chapitre 7 conclut cette partie.

Dans une seconde partie, nous nous concentrons sur le partage de données dans un système réparti. Le chapitre 8 décrit le projet pour lequel ont été effectués ces travaux, donc les contraintes du système réparti cible : l'application doit pouvoir connaître des données stockées dans des étiquettes RFID distantes, alors qu'elle ne peut pas accéder à un réseau informatique global. Le chapitre 9 dresse l'état de l'art des architectures système exploitant des étiquettes RFID. Aucune ne répond aux contraintes de manière satisfaisante. C'est pourquoi le chapitre 10 présente la RDSM que nous avons conçue et expérimentée dans le cadre d'un jeu ubiquitaire. Puis, le chapitre 11 compare notre RDSM aux architectures de la littérature. Ainsi, il propose des directives pour choisir l'architecture la plus adaptée à une application donnée. Enfin, le chapitre 12 conclut cette partie.

Le chapitre 13 conclut ce manuscrit et synthétise les pistes de recherche soulevées par ce travail.

Première partie

Communication de données dans un système réparti

Chapitre 2

Introduction

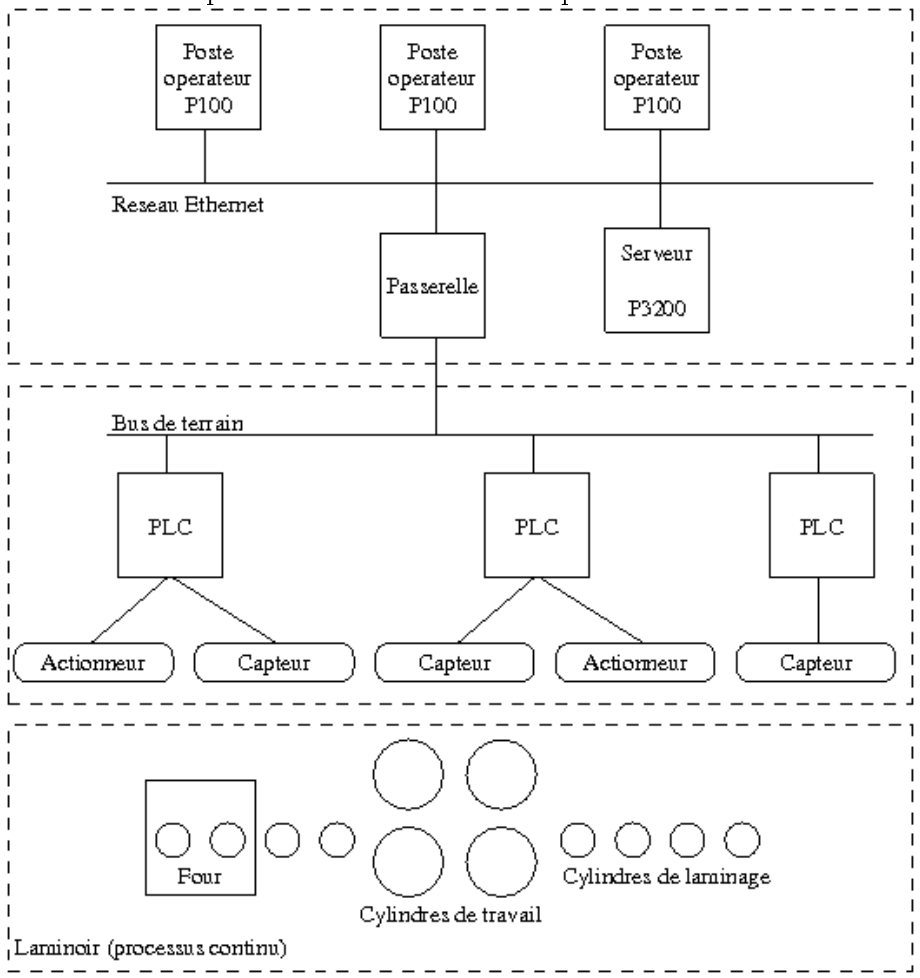
Cette première partie présente nos travaux liés à la communication de données dans un système réparti. Ces travaux ont été effectués dans le cadre du système de contrôle-commande et de supervision [1] *P3200* de *Cegelec*.

Ce type de système est utilisé pour gérer des processus industriels au sens large : laminoirs, chaînes de montage automobile, scènes de théâtre... Il s'inscrit dans l'architecture d'une application d'informatique industrielle typique. Illustrons cette architecture à travers l'exemple d'un laminoir (cf. figure 2.1) [Baradel *et al.*, 1995] : cette installation industrielle permet de réduire l'épaisseur d'une masse d'acier, la *brame*, pour fabriquer des tôles ou des plaques. Pour ce faire, on chauffe la brame dans un four. Des *cylindres de laminage* transportent cette brame entre une succession de *cylindres de travail* : ils l'écrasent pour obtenir l'épaisseur désirée. La vitesse de rotation des différents cylindres doit être synchronisée en continu, sous peine d'accidents. Pour ce faire, le site industriel est équipé de capteurs (vitesse de rotation, vitesse de défilement, température...) et d'actionneurs (contrôle de la température, du débit du liquide de refroidissement...). Les capteurs sont connectés à des automates programmables industriels (nommés PLC dans la figure selon l'acronyme de l'anglais *Programmable Logic Controller*). Ces derniers exécutent des boucles de contrôle qui acquièrent les données des capteurs, effectuent un traitement, et éventuellement commandent les actionneurs. Les PLC peuvent aussi dialoguer entre eux à travers un bus de terrain. Ils peuvent aussi envoyer des informations, via la passerelle et le réseau Ethernet, au système de contrôle et de supervision *P3200*. Ce dernier est un processus programmé en Objective-C [2] qui s'exécute sur une machine sous système Unix ou DEC/VMS. Il est chargé d'agréger les informations reçues des différents PLC (calcul de moyenne, historisation des valeurs...) et de faire des comparaisons à des seuils pour déclencher des alarmes si besoin. Les P100 sont des applications graphiques sous système Unix ou DEC/VMS qui, connectées au *P3200*, sont chargées de représenter graphiquement les données traitées par le *P3200*. À travers l'interface graphique du P100, un opérateur peut fixer une consigne (par exemple, réduire l'ouverture d'une vanne). Cette consigne est envoyée au *P3200* qui l'envoie à l'automate programmable concerné via la passerelle. Ce PLC envoie ensuite des

1. Le terme SCADA, acronyme de l'anglais *Supervisory Control And Data Acquisition*, est également utilisé.
2. L'Objective-C est essentiellement utilisé aujourd'hui pour les applications *iPhone*.

instructions à l'actionneur concerné.

FIGURE 2.1 – Exemple d'architecture informatique industrielle à base de *P3200*



L'objectif du projet auquel je contribue est de rendre le *P3200* tolérant aux fautes en tenant compte des contraintes suivantes :

1. impact faible sur le code existant (pour minimiser le coût final d'un *P3200* tolérant aux fautes),
2. coûts en matériel additionnel réduits (toujours pour minimiser le coût final d'un *P3200* tolérant aux fautes),
3. faibles répercussions sur l'utilisation des ressources (CPU, mémoire et réseau) pour préserver un maximum de ressources pour la fonction contrôle-commande et supervision, et ainsi ne pas avoir besoin de matériels additionnels,
4. charge induite par le mécanisme de tolérance aux fautes équilibrée entre les différentes machines (pour éviter qu'une machine ne rende plus qu'un service dégradé parce qu'elle est trop occupée par la fonction de tolérance aux fautes),

5. temps de récupération de fautes réduit (de sorte que l'opérateur humain puisse récupérer le plus rapidement possible une réplique de *P3200* opérationnelle en cas de problème).

Au moment où j'intègre le projet, le principe de la solution a été retenu. Le projet ne part pas sur une démarche « classique » de machines tolérantes aux fautes. Certes, cette démarche respecterait la contrainte #1. Mais, ces machines sont non seulement coûteuses (contradiction avec la contrainte #2), mais en plus leurs performances sont en retrait par rapport à des machines normales (contradiction avec la contrainte #3). De ce fait, les autres membres du projet ont choisi de répliquer des processus *P3200* sur des machines banalisées. Leur idée est d'exploiter la propriété d'attachement dynamique d'Objective-C pour répercuter chaque évolution d'un processus p vers toutes les répliques de p : sur le processus p, quand un objet o_1 invoque une méthode m sur un objet o_2, pour respecter l'attachement dynamique, Objective-C invoque systématiquement une procédure chargée de déterminer la méthode correspondant à m pour o_2. Au niveau de cette procédure, nous mettons en place un intercepteur qui diffuse (avec des garanties de livraison et d'ordre) cette invocation de méthode à l'ensemble des répliques [3]. Sur réception de cette diffusion, chaque processus invoque m sur sa propre instance de o_2. Ce mécanisme permet donc d'implanter une réplication d'objets entre différents processus [Eychenne *et al.*, 1992]. Dans le cas où un mécanisme additionnel est mis en place pour que p et toutes ses répliques démarrent dans le même état (toutes les répliques d'objet ont la même valeur), alors l'ensemble des processus évolue selon une approche « machine à états » : la tolérance aux fautes est assurée [Schneider, 1990].

Notez que la mise en place de ces mécanismes respecte la contrainte #1 (impact faible sur le code existant du *P3200*). En effet, outre la mise en place de l'intercepteur, il faut déterminer, pour chaque classe d'objets, les méthodes qui modifient l'état d'un objet (et dont l'invocation devra donc être diffusée) et mettre en place des méthodes de transfert d'état des objets d'une réplique de *P3200* déjà active vers une réplique qui démarre (et qui doit donc synchroniser son état avec celui des *P3200* déjà actifs).

Dans la suite de cette partie, je présente ma contribution à ce projet, c'est-à-dire une mémoire répartie partagée à base d'uot-diffusions. En effet, le mécanisme de réplication d'objets décrit précédemment peut être considéré comme un « *lazy cache algorithm* » [Afek *et al.*, 1993] : quand un processus p a besoin d'accéder en lecture au contenu d'un objet o, il lui suffit de consulter directement l'état de o (vu que p détient sa valeur dans sa mémoire qui peut être qualifiée de « cache » de p). En revanche, quand p a besoin d'invoquer une méthode sur o et que cette méthode modifie l'état de p, p diffuse cette mise à jour pour que toutes les répliques mettent à jour la valeur de o dans leur propre cache.

Suite à l'état de l'art sur l'uot-diffusion (cf. chapitre 3), pour respecter les contraintes #3 (impact faible sur l'utilisation des ressources), #4 (charge induite équilibrée entre les machines) et #5 (Temps de récupération de fautes réduit), nous avons conçu le *protocole* des *trains* (cf. chapitre 4) à partir de l'idée du *protocole* du *train* de [Cristian, 1991].

Le chapitre 5 étudie les propriétés de ce protocole : il démontre que le protocole des trains est effectivement un protocole d'uot-diffusion. Par ailleurs, il évalue ses performances

3. Dans le vocabulaire d'aujourd'hui, nous pourrions dire que nous ajoutons un aspect (de la programmation orientée aspect) au niveau de la méthode invoquée.

d'un point de vue théorique, selon les métriques recensées au chapitre 3. Cela nous amène à créer la métrique *rendement en termes de débit*. Cette métrique nous permet de montrer que le protocole des trains est actuellement le protocole le plus performant en termes de débit. Elle nous permet également de calculer la borne maximale atteignable avec ce protocole.

Puis, le chapitre 6 décrit l'utilisation du protocole des trains pour une application industrielle, en l'occurrence une mémoire répartie partagée pour le *P3200*. Nous résolvons les points difficiles liés à l'implantation du protocole des trains. Nous proposons notamment une solution originale pour la gestion du dépassement de capacité des horloges logiques utilisés par ce protocole. Nous décrivons ensuite la mise en œuvre de notre mémoire répartie partagée au dessus de notre protocole d'uot-diffusion.

Enfin, le chapitre 7 conclut cette partie en présentant les perspectives de nos travaux.

Chapitre 3

État de l'art des protocoles de diffusion uniforme à ordre total

Ce chapitre dresse un état de l'art des protocoles de diffusion uniforme à ordre total. Nous commençons par spécifier les propriétés de ces protocoles (cf. section 3.1). Puis nous synthétisons les protocoles qui ont déjà été imaginés (cf. section 3.2). Ensuite (cf. section 3.3), nous nous focalisons sur les études de performance qui ont été menées sur ces protocoles. La section 3.4 conclut ce chapitre.

3.1 Spécification des protocoles de diffusion uniforme à ordre total

Dans cette section, nous spécifions les protocoles d'uot-diffusion. La section 3.1.1 définit ce qu'est un processus *correct*. Ainsi, la section 3.1.2 peut spécifier l'uot-diffusion proprement dite. Enfin, la section 3.1.3 présente d'autres propriétés que peut vérifier un protocole d'uot-diffusion. La section 3.1.4 conclut cette section.

3.1.1 Définition d'un processus correct

La spécification des protocoles d'uot-diffusion requiert la définition de la notion de processus *correct*. Selon [Défago *et al.*, 2004], un processus *correct* est un processus qui n'exprime jamais de faute appartenant à l'une des classes suivantes :

- *arrêts francs (crash failures)* : arrêt définitif du processus, qui ne répond ou ne transmet plus ;
- *fautes par omission* : le processus omet d'effectuer certaines actions. Notamment, il peut omettre d'envoyer ou recevoir un message ;
- *fautes temporelles* : un résultat ou un message est délivré trop tard ou trop tôt par un processus ;
- fautes byzantines : le processus renvoie des valeurs fausses et/ou fait des actions quelconques fallacieuses. Un tel processus peut changer le contenu des messages, dupliquer des messages, envoyer des messages non sollicités, etc.

3.1.2 Spécification de la diffusion uniforme à ordre total

Pour spécifier la diffusion uniforme à ordre total, [Défago *et al.*, 2004] définit deux primitives *uot-diffusion(m)* et *uot-livraison(m)* où m est un message. Par ailleurs, il suppose que : 1) chaque message m est identifiable de manière unique ; 2) pour un message m donné, *uot-diffusion(m)* est exécuté au plus une fois.

Définition 1 (uot-diffusion et uot-livraison d'un message [Défago *et al.*, 2004])**.** *Soit un processus p. On dit que p fait une* uot-diffusion *(respectivement une* uot-livraison*) d'un message m ou bien que p* uot-diffuse *(respectivement* uot-livre*) lorsque p invoque la primitive* uot-diffusion(m) *(respectivement* uot-livraison(m)*).*

Soient les quatre propriétés suivantes :

Propriété 1 (Validité [Défago *et al.*, 2004, Guerraoui *et al.*, 2010])**.** *Si un processus correct uot-diffuse un message m, alors il uot-livre* in fine[1] *le message m.*

Propriété 2 (Accord uniforme [Défago *et al.*, 2004, Guerraoui *et al.*, 2010])**.** *Si un processus uot-livre un message m, alors tous les processus corrects uot-livreront* in fine *m.*

Propriété 3 (Intégrité [Défago *et al.*, 2004, Guerraoui *et al.*, 2010])**.** *Pour tout message m, tout processus uot-livre m au plus une fois, et seulement si m a été précédemment uot-diffusé par émetteur*(m)*.*

Propriété 4 (Ordre total [Guerraoui *et al.*, 2010])**.** *Pour tous messages m et* m'*, si un processus p uot-livre m sans avoir uot-livré* m'*, alors aucun processus* p' *uot-livre* m' *avant m.*

Définition 2 (Diffusion uniforme à ordre total [Guerraoui *et al.*, 2010])**.** *Un protocole de diffusion uniforme à ordre total (uot-diffusion) est un protocole qui respecte les propriétés 1, 2, 3 et 4.*

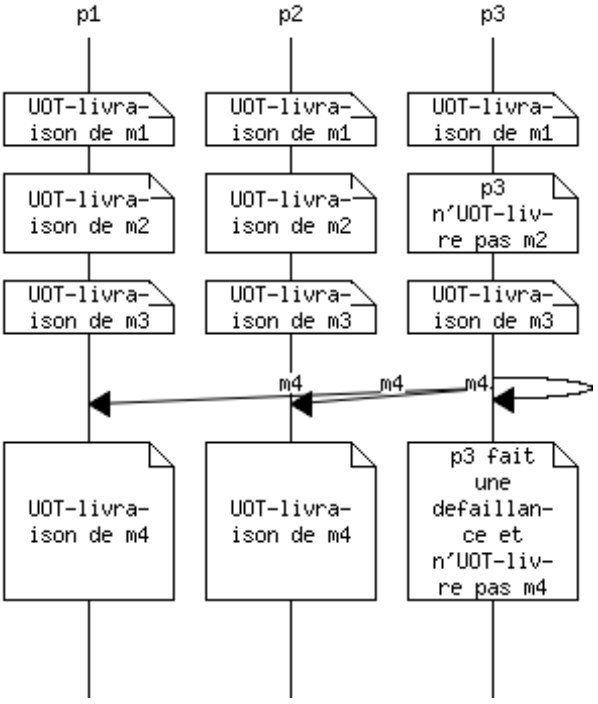
Les propriétés 1, 2 et 3 sont reprises de la littérature. Elles sont identiques dans [Guerraoui *et al.*, 2010] et dans [Défago *et al.*, 2004]. En revanche, pour la propriété 4 concernant l'ordre total, nous proposons une définition originale inspirée de [Guerraoui *et al.*, 2010]. Dans la suite de cette section, nous justifions notre choix.

Expliquons tout d'abord pourquoi nous préférons nous appuyer sur la spécification de [Guerraoui *et al.*, 2010] : la spécification de [Défago *et al.*, 2004] (elle-même tirée de [Hadzilacos et Toueg, 1994]) laisse la place au problème de la *contamination*. Selon [Défago *et al.*, 2004, Hadzilacos et Toueg, 1994], le problème de la contamination provient du fait que l'uot-diffusion n'empêche pas un processus défaillant de passer dans un état incohérent avant qu'il ne s'arrête définitivement. De ce fait, il peut uot-diffuser un message basé sur cet état incohérent et ainsi contaminer les autres processus (corrects). La figure 3.1 illustre ce problème. Le processus p_3 uot-livre les messages m_1 et m_3, mais pas m_2. De ce fait, son état est incohérent lorsqu'il uot-diffuse m_4. Quand p_1 et p_2 uot-livrent m_4, ils deviennent contaminés par l'état incohérent de p_3. Le processus p_3 peut avoir ce comportement, car

1. « *In fine* » est la traduction du mot anglais « *Eventually* ». Il doit être compris par « au bout du compte » ou « au bout d'un certain temps ».

[Défago *et al.*, 2004] spécifie une propriété d'ordre total moins forte que [Guerraoui *et al.*, 2010]. En effet, sa spécification est : « Si les processus p_i et p_j uot-livrent les messages m et m', alors p_i uot-livre m avant m' si et seulement si p_j uot-livre m avant m' ». De ce fait, comme le prédicat « Si les processus p_3 et $p_{j,j\in\{1,2\}}$ uot-livrent les messages m_2 et m_3 » est toujours faux, la propriété d'ordre total de [Défago *et al.*, 2004] est toujours vérifiée. Elle permet donc la contamination. En revanche, la propriété d'ordre total de [Guerraoui *et al.*, 2010] a pour spécification : « Pour tout message m et m', si un processus p_i uot-livre m sans avoir uot-livré m', alors aucun processus p_j uot-livre m' avant m. ». Dans l'exemple de la figure 3.1, si p_3 uot-livre m_3 sans avoir uot-livré m_2, alors aucun processus $p_{j,j\in\{1,2\}}$ ne peut uot-livrer m_2 avant d'uot-livrer m_3. Le message m_4 ne peut pas contaminer $p_{j,j\in\{1,2\}}$ avec un état incohérent pour ces derniers processus.

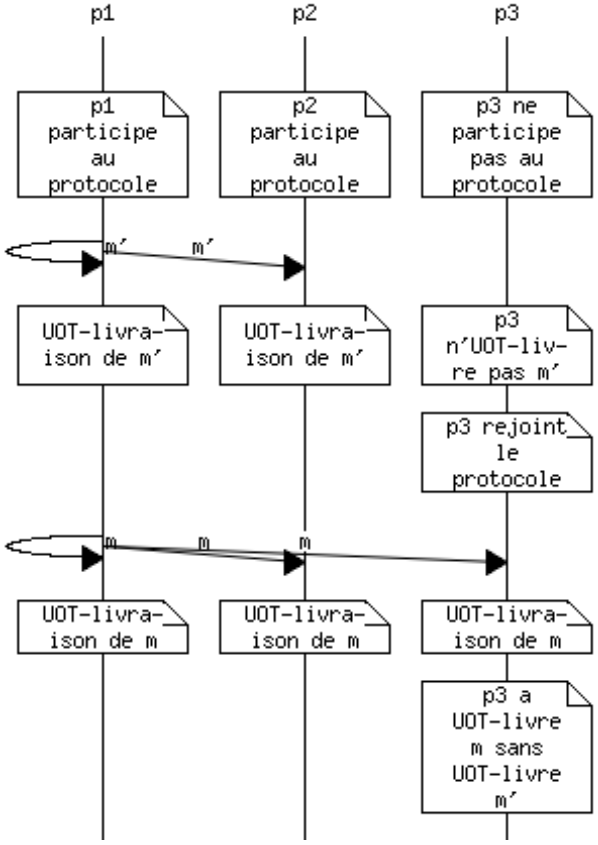
FIGURE 3.1 – Diagramme de séquence de la contamination de processus corrects (p_1, p_2), par un message (m_4), basé sur un état incohérent (p_3 a livré m_3, mais pas m_2)



Notez que la spécification de la propriété 4 (ordre total) par [Guerraoui *et al.*, 2010] convient aux protocoles d'uot-diffusion parce qu'elle est alors couplée à la propriété 2 (accord uniforme). En effet, sans l'accord uniforme, cette spécification peut être mise en défaut. Par exemple, elle ne gère pas le cas de la figure 3.2. En effet, dans ce diagramme de séquence, le processus p_3 uot-livre m sans uot-livrer m', *et* les processus p_1 et p_2 ont uot-livré m' avant m. On peut généraliser cet exemple : la spécification est mise en défaut dans les cas de messages uot-diffusés avant qu'un processus p donné rejoigne le protocole et des messages uot-diffusés après. Toutefois, si la propriété d'accord uniforme est vérifiée, elle induit que, soit les messages uot-diffusés avant l'arrivée de p sont uot-livrés après l'arrivée de p, soit les messages uot-diffusés avant l'arrivée de p sont uot-livrés avant l'arrivée de p

et ne sont pas pris en compte pour la propriété 4 : dans tous les cas, la propriété 4 est vérifiée.

FIGURE 3.2 – Diagramme de séquence mettant en défaut la spécification d'ordre total de [Guerraoui *et al.*, 2010]



3.1.3 Autres propriétés d'ordre

L'uot-diffusion que nous avons spécifiée ne garantit pas deux autres propriétés d'ordre qui peuvent intéresser certaines applications : l'ordre FIFO (cf. section 3.1.3.1) et l'ordre causal (cf. section 3.1.3.2). Par ailleurs, certaines applications peuvent tolérer de relâcher la contrainte d'uniformité. Nous étudions en section 3.1.3.3 ce que cela signifie.

3.1.3.1 Ordre FIFO

Telle qu'elle est spécifiée, l'uot-diffusion ne garantit pas que les messages sont uot-livrés dans l'ordre dans lequel ils ont été émis. Or, certaines applications peuvent avoir besoin de garanties FIFO (*First-In/First-Out*) traduites par la propriété 5.

Propriété 5 (Ordre FIFO [Hadzilacos et Toueg, 1994]). *Si un processus correct diffuse un message m avant de diffuser un message m', alors aucun processus correct ne livre m' avant d'avoir au préalable livré m.*

3.1.3.2 Ordre causal

D'autres applications peuvent requérir un ordre respectant la causalité des événements. Ce besoin est traduit par la propriété 6.

Propriété 6 (Ordre causal [Hadzilacos et Toueg, 1994]). *Si la diffusion d'un message m précède causalement la diffusion d'un message m', alors aucun processus correct ne livre m' à moins d'avoir livrer m au préalable.*

NB : Un protocole d'uot-diffusion qui respecte l'ordre FIFO respecte aussi l'ordre causal [Toinard *et al.*, 1999].

3.1.3.3 Ordre non-uniforme

La propriété 2 porte sur l'accord *uniforme*, c'est-à-dire qu'elle concerne l'ensemble des processus, ceux qui sont corrects et ceux qui ont défailli.

Cette propriété simplifie le développement des applications qui utilisent l'uot-diffusion. En effet, elle garantit que, dès qu'un processus uot-livre un message, *in fine* tous les processus l'uot-livreront. Mais cette simplification se fait au détriment des performances. En général, garantir le caractère uniforme requiert des ressources (CPU, mémoire et réseau) supplémentaires. De plus, le temps écoulé entre l'uot-diffusion du message par un processus et son uot-livraison par les différents processus est allongé. En effet, le protocole d'uot-diffusion doit garantir que tous les processus participant au protocole ont reçu le message et sont prêts à l'uot-livrer.

C'est pourquoi [Défago *et al.*, 2004] définit la propriété 7.

Propriété 7 (Accord). *Si un processus* **correct** *livre un message m, alors tous les processus corrects délivrent le message m* in fine.

On peut alors spécifier la diffusion (non-uniforme) à ordre total.

Définition 3 (Diffusion (non-uniforme) à ordre total). *Un protocole de diffusion (non-uniforme) à ordre total (nuot-diffusion) est un protocole qui respecte les propriétés 1, 7, 3 et 4.*

NB :

- Cette définition diffère de celle de [Défago *et al.*, 2004]. En effet, pour définir la nuot-diffusion, [Défago *et al.*, 2004] remplace la propriété 4 par un ordre total non-uniforme : « Si les processus p_i et p_j **corrects** uot-livrent les messages m et m', alors p_i uot-livre m avant m' si et seulement si p_j uot-livre m avant m' ». Du fait que, pour l'ordre total, nous avons adopté une spécification inspirée de [Guerraoui *et al.*, 2010], il n'y a pas de notion d'uniformité dans notre ordre total.

– [Guerraoui *et al.*, 2010] ne spécifie pas la diffusion (non-uniforme) à ordre total. Peut-être est-ce parce que, grâce à du *piggy-backing*, le protocole d'uot-diffusion présenté dans [Guerraoui *et al.*, 2010] offre l'uniformité « gratuitement », en fait sans message supplémentaire (mais avec une latence plus grande et des octets additionnels à transporter sur le réseau).

3.1.4 Conclusion

Dans ce chapitre, nous avons spécifié l'uot-diffusion à partir de la notion de processus correct et les propriétés de validité, accord uniforme, intégrité, et d'ordre total. Nous avons également défini l'ordre FIFO et l'ordre causal. Nous avons vu que si un protocole d'uot-diffusion garantit l'ordre FIFO, il garantit l'ordre causal. Enfin, nous avons spécifié la diffusion (non-uniforme) à ordre total qui, par rapport à l'uot-diffusion, remplace la propriété d'accord uniforme par une propriété d'accord (simple). Fort de ces spécifications, nous sommes maintenant en mesure d'étudier les différents protocoles d'uot-diffusion qui ont été proposés dans la littérature.

3.2 Classes de protocoles de diffusion uniforme à ordre total

[Défago *et al.*, 2004] distingue cinq classes de protocoles d'uot-diffusion que nous détaillons dans les sections suivantes : séquenceur fixe (cf. section 3.2.1), séquenceur mobiles (cf. section 3.2.2), à base de privilège (cf. section 3.2.3), à base d'historique des communications (cf. section 3.2.4), accord des destinataires (cf. section 3.2.5).

3.2.1 Séquenceur fixe

Dans l'algorithme à séquenceur fixe [Kaashoek *et al.*, 1989, Oestreicher, 1991], un participant joue le rôle particulier de séquenceur p_s : chaque station qui souhaite faire une uot-diffusion envoie son message à p_s (avec un numéro d'ordre local pour que p_s puisse détecter les trous éventuels). Ce dernier le diffuse à tous les participants (avec un numéro d'ordre global pour que chaque destinataire puisse repérer des trous éventuels).

Dans le cas où un accord uniforme est requis, sur réception de la diffusion de p_s, chaque participant renvoie un message d'acquittement. p_s diffuse alors un message pour signifier aux participants qu'ils peuvent uot-livrer le message.

Remarques :

– S'il y a plusieurs groupes de destinataires des uot-diffusions, les protocoles à séquenceurs fixes proposent que le participant séquenceur p_s diffuse les messages à tous les participants de tous les groupes. Chaque participant filtre ensuite le message selon qu'il fait partie du groupe destinataire ou non. [Florin et Toinard, 1992] propose une optimisation : p_s n'envoie le message qu'aux membres du groupe. p_s ne peut plus utiliser un numéro d'ordre global. En effet, un participant p pourrait s'affoler en constatant qu'il a des « trous » dans la réception des messages, alors qu'en fait p_s ne lui a pas envoyé certains messages qui étaient à destination de groupes dont p ne fait pas partie. p_s utilise donc un numéro d'ordre spécifique à chaque participant.

- Pour gérer les uot-diffusions, *JGroups* utilise uniquement la notion de participant séquenceur [Ban02]. En effet, il peut s'affranchir des numéros d'ordre local et global, vu qu'il s'appuie, pour la transmission des messages, sur des canaux évitant la perte ou le déséquencement des messages. Notez que *JGroups* ne gère pas la propriété d'accord uniforme [Guerraoui *et al.*, 2010].
- Les protocoles d'uot-diffusion ne se résument pas à un envoi de message du participant vers le partipant séquenceur et un retour de ce dernier. En effet, il faut éventuellement ajouter des messages de contrôle pour permettre à chaque participant de purger sa liste de messages en attente de confirmation. Certes, dans le meilleur des cas, chaque participant profite de la prochaine diffusion totale qu'il fait pour *piggy-backer* au séquenceur le fait qu'il a bien reçu la précédente diffusion. Dans le pire des cas, le rythme de diffusions totales est trop faible sur chaque participant et requiert donc l'envoi d'un message de contrôle pour valider la réception du multicast. Il y a alors des messages additionnels. [Oestreicher, 1991] propose une méthode pour ne pas gérer de messages de contrôle : le séquenceur ne garde que les k derniers messages envoyés (k étant une valeur configurée par l'utilisateur de l'algorithme) sans se préoccuper de savoir si ceux qui les ont précédés ont effectivement été reçus par les destinataires. Dans ce cas, un participant qui n'aurait pas reçu un tel message ne peut plus jamais le récupérer : il se suicide donc.
- La perte du participant séquenceur p_s requiert l'élection d'un nouveau séquenceur p'_s et la mise en place d'un algorithme spécifique entre p'_s et les autres participants pour que p'_s détermine où en était p_s dans ses diffusions de messages avant sa défaillance.
- [Défago *et al.*, 2004, Guerraoui *et al.*, 2010] estiment que le protocole `ABCAST` d'Isis [Schiper *et al.*, 1991] utilise un séquenceur. Effectivement, l'un des processus participant au protocole joue un rôle particulier : il est en charge d'informer les participants au protocole de l'ordre dans lequel ils doivent prendre en compte les messages diffusés de manière causale à l'aide du protocole `CBCAST`. Ce processus particulier peut donc être vu comme le processus séquenceur du protocole `ABCAST`. Mais le protocole `CBCAST` s'appuie sur des vecteurs d'horloges. Il est donc à base d'historique des communications. De ce fait, `ABCAST` est non seulement à base de séquenceur fixe, mais aussi d'historique des communications.
- Certains jeux vidéo multijoueurs utilisent une uot-diffusion à base de séquenceur fixe. Ainsi, dans le jeu *Darwinia*, chaque instance du jeu envoie à un serveur les mouvements de troupe ordonnés par le joueur [Kno09]. Le serveur renvoie alors, à chaque instance de jeu, une « lettre réseau » (numérotée pour qu'une instance de jeu puisse détecter les pertes de messages du serveur) avec la synthèse des instructions de tous les joueurs.

3.2.2 Séquenceur mobile

Dans les algorithmes à séquenceur mobile [Chang et Maxemchuk, 1984, Cristian et Mishra, 1993], le rôle de séquenceur passe d'un processus à l'autre. Pour ce faire, un jeton (contenant le numéro d'ordre) circule en permanence entre les processus.

Quand un processus p veut uot-diffuser un message m, il l'envoie à tous les participants. Sur réception de m, chaque participant stocke le message dans une queue de réception.

Quand la queue de réception du processus q détenteur du jeton contient au moins un message, q donne un numéro d'ordre au premier message dans cette queue, puis le diffuse avec le jeton.

3.2.3 Protocoles à base de privilège

Dans les protocoles à base de privilège, un processus qui souhaite faire une uot-diffusion ne peut le faire que quand il en reçoit le privilège.

Ainsi, dans le protocole Totem [Amir *et al.*, 1995], quand un processus p souhaite uot-diffuser un message m, il le stocke dans une queue d'envoi jusqu'à ce qu'il reçoive un jeton (qui circule en permanence entre les machines et qui contient un numéro d'ordre). p prend alors le premier message de sa queue d'envoi, lui ajoute le numéro d'ordre stocké dans le jeton, diffuse le message, incrémente le numéro d'ordre dans le jeton, et enfin envoie le jeton au processus suivant. Chaque participant uot-livre les messages selon l'ordre fixé par le numéro d'ordre.

Notez qu'une implantation des principes de Totem a été testée (sans suite) dans *JGroups* sous le nom de `TOTAL_ORDER` [Blagojevic, 1998].

L'algorithme du train [Cristian, 1991] reprend l'image d'un train (le jeton) constitué de wagons de messages. Quand un processus p reçoit le jeton, il prend en compte les messages stockés dedans. En particulier, si un message passe pour la deuxième fois, p considère que ce message a été vu par tous les autres processus : il peut l'uot-livrer en étant sûr que la propriété d'accord uniforme est respectée. Ensuite, p ajoute les messages de sa queue d'envoi en fin de train. Enfin, p envoie le nouveau jeton au processus suivant dans l'anneau virtuel sur lequel circule le jeton.

3.2.4 Protocoles à base d'historique des communications

Dans les protocoles à base d'historique des communications, quand un processus p souhaite uot-diffuser un message, il le diffuse immédiatement. Sur réception d'une diffusion m, un participant q attend d'être sûr qu'il n'y a pas un message précédant m, message que q n'aurait pas encore reçu. Lorsqu'il a cette certitude, q uot-livre le message.

Ainsi, Psync est un protocole permettant une diffusion fiable et causale des messages [Peterson *et al.*, 1989]. Une extension au protocole permet de rendre la diffusion ordonnée. L'envoi d'un message se déroule de la manière suivante. L'émetteur insère dans son message l'identifiant des messages reçus dont dépend causalement son message. Pour respecter la causalité, un récepteur ne livre le message que s'il a reçu (et livré) les prédécesseurs de ce message. Le respect de l'ordre est obtenu comme suit. Chaque récepteur conserve dans une file d'attente spéciale les messages (devant être ordonnés) qui ne dépendent que de messages déjà reçus et déjà livrés. Le récepteur attend d'être sûr qu'un des messages de la file d'attente a été reçu par tous les autres participants au protocole de diffusion. Cette certitude est obtenue après avoir reçu des messages de tous les autres participants indiquant que le message a été bien reçu (ce peut être des messages de contrôle ou bien des messages dépendant causalement de ce message). Il applique alors un algorithme

de tri [2] sur les messages de cette file d'attente spéciale et les livre à l'application.

Dans le cas du protocole ABCAST d'Isis, l'information de dépendance causale est rendue plus compacte en utilisant des vecteurs d'horloges [Schiper *et al.*, 1991]. L'ordre total est obtenu en utilisant un processus séquenceur parmi les participants au protocole.

Le protocole LCR utilise également des vecteurs d'horloges [Guerraoui *et al.*, 2010]. Mais il réussit à s'affranchir d'un algorithme de tri et d'un processus séquenceur en disposant les processus sur un anneau virtuel. Quand un processus souhaite uot-diffuser un message, il l'envoie à son successeur sur l'anneau virtuel, successeur qui l'envoie à son propre successeur, etc. De ce fait, le vecteur d'horloges qu'un participant p insère dans un message tient compte des messages uot-diffusés jusqu'à présent (puisque ces messages transitent forcément par p). Donc, le vecteur d'horloges permet d'ordonner totalement les messages.

Pour construire l'historique des communications, certains protocoles utilisent le temps physique. Ainsi, dans le protocole de [Cristian *et al.*, 1985], les horloges des participants au protocole sont synchronisées et les identifiants des participants au protocole sont ordonnés de manière strictement croissante. Un message est émis avec l'identifiant de l'émetteur et la date d'émission. Il est diffusé par inondation, c'est-à-dire un participant au protocole qui reçoit ce message (et qui ne l'a encore jamais reçu) le rediffuse à tous les autres participants : les éventuelles pertes de message sont compensées. Au niveau de chaque récepteur, un message est considéré comme livrable quand un certain laps de temps Δ [3] s'est écoulé depuis la date d'émission mentionnée dans le message. Les messages livrables sont livrés par ordre croissant de date et par ordre croissant d'identifiants des émetteurs en cas d'égalité de date.

[Guerraoui *et al.*, 2010] estime que, dans la pratique, l'hypothèse d'avoir des horloges synchronisées est peu réaliste, du fait notamment que la dérive d'horloge est difficile à détecter. Toutefois, des applications industrielles utilisent des protocoles à base de temps physique. C'est le cas notamment de plusieurs jeux vidéos de type « stratégie temps réel » qui utilisent cette classe de protocole. Ainsi, dans *Age of Empire*, chaque instance de jeu diffuse à toutes les autres instances de jeu les mouvements de troupes commandés par le joueur [Bettner et Terrano, 2001]. À intervalles réguliers, chaque instance trie les messages reçus dans un certain intervalle de temps (par heure d'émission du message et, si égalité, par identifiant de l'émetteur), puis les uot-livre à l'instance de jeu. Ce protocole tolère des latences inférieures à une demi-seconde (au-delà, le jeu n'est pas jouable) tant que la gigue du réseau reste faible. Notez que des incohérences peuvent apparaître entre les différentes instances, soit parce que des messages sont perdus, soit parce qu'un joueur triche et modifie artificiellement les données gérées par son instance de jeu. C'est pourquoi [Bettner et Terrano, 2001] complète ce protocole avec un échange périodique (entre les instances de jeu) d'une clé de hachage caractéristique de l'état du jeu. Si une instance est divergente par rapport aux autres, elle se suicide. [GauthierDickey *et al.*, 2004] propose de limiter préventivement la triche en cryptant les mouvements de troupes diffusés lors d'un

2. L'algorithme de tri est un algorithme quelconque : l'essentiel est que tous les participants appliquent le même algorithme de manière à livrer les messages dans le même ordre.

3. Ce laps de temps est calculé pour correspondre au temps requis pour être sûr que tous les participants ont reçu le message (compte-tenu des délais de transmission réseau, des pertes de message, et de la précision de la synchronisation des horloges).

intervalle de temps, et en ne fournissant la clé de décryptage qu'à l'intervalle de temps suivant.

3.2.5 Accord des destinataires

Dans cette classe de protocoles, les destinataires se mettent d'accord sur l'ordre de livraison.

Ainsi, dans [Birman et Joseph, 1987b], un participant qui souhaite diffuser un message m multicaste m à tous les autres. Ces derniers le stockent dans une file d'attente des messages non livrables en lui affectant une estampille locale (l'estampille locale étant incrémentée à chaque réception de message). Ils renvoient ensuite un acquittement avec cette estampille. Après réception de toutes ces estampilles, le participant émetteur calcule le maximum max de toutes ces estampilles. Puis, il renvoie un message contenant max. Sur réception de ce message, chacune des stations affecte l'estampille max à m et place m dans la queue des messages livrables. Les messages sont livrés à l'application par ordre croissant d'estampille.

3.3 Performances des protocoles de diffusion uniforme à ordre total

La section 3.2 a montré qu'il existait de nombreux protocoles d'uot-diffusion. Dans cette section, nous étudions les performances de ces protocoles en vue de les comparer. La section 3.3.1 s'intéresse aux performances des protocoles en l'absence de défaillances. Puis, la section 3.3.2 étudie l'influence du service de gestion des participants au protocole.

3.3.1 Performances des protocoles en l'absence de défaillances

[Urbán *et al.*, 2000b] distingue la prédiction de performances qui permet de donner des indications des performances d'un protocole avant son codage, et l'estimation des performances réalisée *a posteriori* soit dans le cadre d'une simulation, soit en analysant un système réel. Ces deux techniques sont complémentaires. La prédiction de performances permet d'orienter les choix de conception. L'estimation des performances peut confirmer ces décisions et permet de dimensionner les différents paramètres du système réel.

Dans cette section, nous nous focalisons sur les travaux liés à la prédiction de performances. En effet, la prédiction de performances fournit des indications plus générales, car indépendantes des hypothèses sur le système simulé (comme dans [Cristian et Mishra, 1995]) ou des caractéristiques du système évalué (comme dans [Guerraoui *et al.*, 2010]).

La prédiction de performances suppose la définition d'une (ou plusieurs) métrique(s), c'est-à-dire une valeur associée à un protocole qui n'a pas de réalité physique, mais permet de définir un classement entre les protocoles.

Classiquement, les métriques utilisées pour les algorithmes répartis sont la *complexité en temps* (c'est-à-dire le temps maximum écoulé entre le début de l'algorithme et sa terminaison) et la *complexité en messages* (c'est-à-dire le nombre maximum de messages échangés

entre les différents processus participant) [Attiya et Welch, 2004]. Mais [Urbán *et al.*, 2000b] remarque que ces deux métriques ne tiennent pas compte des capacités limitées des CPU et du réseau utilisés. C'est pourquoi il propose d'estimer les performances des algorithmes en considérant que ces algorithmes s'exécutent sur un système qui héberge n processus participant à l'algorithme. Ce système nécessite :

- λ unité(s) de temps CPU pour qu'un message soit traité par le processus qui souhaite émettre le message ($\lambda \in \mathbb{R}^{*+}$) [4],
- 1 unité de temps pour que le message transite sur le réseau (l'accès au réseau étant modélisé selon une politique de tourniquet),
- λ unité(s) de temps CPU pour que le message soit traité par un processus qui reçoit le message.

[Urbán *et al.*, 2000b] définit alors :

- la *métrique de latence* qui est le nombre d'unités de temps qui séparent le début et la fin de l'algorithme pour chaque uot-diffusion. Autrement dit, la latence correspond au nombre d'unités de temps écoulées entre le moment où un processus uot-diffuse un message et le moment où ce message est uot-livré à tous les participants au protocole ;
- la *métrique de débit* (notée $d\acute{e}bit_{gén}$ dans la suite, vu que cette métrique est générique à tout type d'algorithme réparti) qui vaut $\frac{1}{\max_{r \in \mathscr{R}_n} T_r(n,\lambda)}$ où $\mathscr{R}_n$ désigne l'ensemble des ressources du système (c'est-à-dire CPU_1,...,CPU_n et le réseau) et $T_r(n,\lambda)$ désigne la durée totale de l'utilisation de la ressource r pendant l'exécution de l'algorithme.

Grâce à ces deux métriques, [Défago *et al.*, 2003] compare les différentes classes de protocoles définies en section 3.2 : il effectue des représentations graphiques de ces métriques en fonction du nombre de processus n et de λ. Il montre ainsi que les protocoles à base d'historique et ceux à base de privilège ont des $d\acute{e}bit_{gén}$ d'une part similaires, et d'autre part, supérieurs (donc meilleurs) à ceux des protocoles à base de séquenceur (mobile ou fixe). En ce qui concerne la latence, les résultats varient selon que le réseau offre un multicast pour pouvoir envoyer un message à plusieurs destinataires simultanément ou non (seulement des messages point-à-point). Avec du multicast, les protocoles à base d'historique ont moins de latence (et sont donc meilleurs) que les algorithmes avec séquenceur fixe, eux-mêmes meilleurs que les protocoles à base de privilège. En cas de point-à-point, les protocoles à base d'historique n'ont la latence la plus faible que pour un nombre de processus réduit ou bien pour de grandes valeurs de λ (supérieures à 4 quand $n = 10$).

Ce classement doit toutefois être nuancé. Tout d'abord, les protocoles à base de privilège ont une tendance naturelle à agréger des petits messages en de gros messages [Birman, 2010]. En effet, en attendant l'arrivée du jeton, les petits messages que souhaite uot-diffuser un processus peuvent s'accumuler dans un seul et même gros message. De ce fait, d'une part, cela permet de réduire le nombre d'appels système effectués pour envoyer tous ces petits messages : la consommation CPU associée et donc la valeur de λ sont diminuées. D'autre part, il y a moins d'interruptions au niveau du récepteur des gros messages : là encore, la consommation CPU associée [Défago *et al.*, 2004, Friedman et van Renesse, 1997, Bauer

4. Le symbole λ peut sembler malheureux dans la mesure où il fait inévitablement penser à la théorie des files d'attente. Mais, il correspond à la notation retenue dans [Urbán *et al.*, 2000b]. C'est donc le symbole que nous avons retenu dans ce manuscrit.

et al., 2002, Smed *et al.*, 2001] et donc à nouveau la valeur de λ sont réduits. Par conséquent, la comparaison des protocoles à base de privilège et les autres classes de protocoles ne peut pas s'appuyer sur la même valeur de λ. Aussi, si le $d\acute{e}bit_{g\acute{e}n}$ des algorithmes est limité à cause de la valeur de λ, le $d\acute{e}bit_{g\acute{e}n}$ des protocoles à base de privilège est meilleur que le $d\acute{e}bit_{g\acute{e}n}$ des protocoles à base d'historique des communications.

Par ailleurs, [Urbán *et al.*, 2000b] suppose que l'accès au réseau se fait selon une politique de *tourniquet*. Or, [Guerraoui *et al.*, 2010] remarque qu'avec les matériels actuels : 1) une carte réseau peut émettre un message et simultanément en recevoir un autre, 2) si le réseau est géré par un *switch*, ce *switch* est en mesure d'acheminer plusieurs messages simultanément. [Guerraoui *et al.*, 2010] propose donc de modéliser le réseau selon une politique *à base de tours*. Découpons le temps en intervalle de temps k suffisamment longs pour que chaque processus envoie (éventuellement) un message, le *switch* les achemine, et le(s) processus le(s) reçoive(nt). Alors, chaque processus p_i exécute les étapes suivantes :

1. p_i calcule le message $m(i,k)$ pour l'intervalle de temps k,
2. p_i envoie $m(i,k)$ à tous les ou alors un sous-ensemble des processus,
3. p_i reçoit au plus un message envoyé à l'intervalle de temps k.

Fort de cette constatation, [Guerraoui *et al.*, 2010] propose de définir une nouvelle *métrique de débit* (notée $d\acute{e}bit_{UOT}$ dans la suite, vu que cette métrique est spécifique aux protocoles d'uot-diffusion). C'est le nombre moyen d'uot-diffusion(s) réalisée(s) par intervalle de temps (« réalisé » signifie que tous les processus ont uot-livré un message précédemment uot-diffusé).

[Guerraoui *et al.*, 2010] démontre le théorème suivant :

Théorème 1. *Pour un protocole de diffusion quelconque dans un système avec n processus et un réseau modélisé avec une politique à base de tours, le débit maximum μ_{max} (en termes de diffusions réalisées pendant un tour) vaut $n/(n-1)$ si chacun des n processus participant au protocole diffuse des messages et 1 sinon.*

Notez que μ_{max} vaut seulement 1 si tous les processus diffusent des messages en les multicastant aux $n-1$ autres processus. En effet, dans ce cas, pour effectuer n diffusions, n intervalles de temps sont requis (cf. figure 3.3). Donc, $\mu_{max} = n/n = 1$. Pour que μ_{max} atteigne $n/(n-1)$, il faut que chaque processus envoie à son successeur (sur un anneau virtuel) le message qu'il souhaite diffuser, qu'au tour suivant chaque processus fasse suivre le message à son propre successeur.... Alors, seuls $n-1$ intervalles de temps sont requis pour acheminer les n diffusions (cf. figure 3.4). Donc, $\mu_{max} = n/(n-1)$.

Le théorème 1 induit le lemme suivant :

Lemme 1. $\max d\acute{e}bit_{UOT} = n/(n-1)$ *s'il y a n émetteurs.*

[Guerraoui *et al.*, 2010] évalue ensuite $d\acute{e}bit_{UOT}$ pour les différentes classes d'algorithmes vues en section 3.2.

Pour les algorithmes à base de séquenceur, en ne tenant pas compte des possibilités de *piggy-backing* des messages d'acquittement de réception, il faut un intervalle de temps pour qu'un processus p_i envoie au processus séquenceur p_s un message m à uot-diffuser, un autre

FIGURE 3.3 – Diagramme de séquence montrant le nombre d'intervalles de temps requis pour acheminer les messages si les n processus multicastent

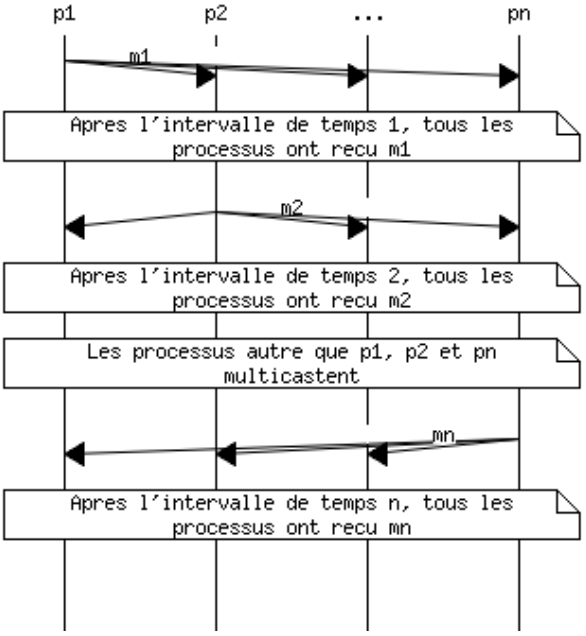
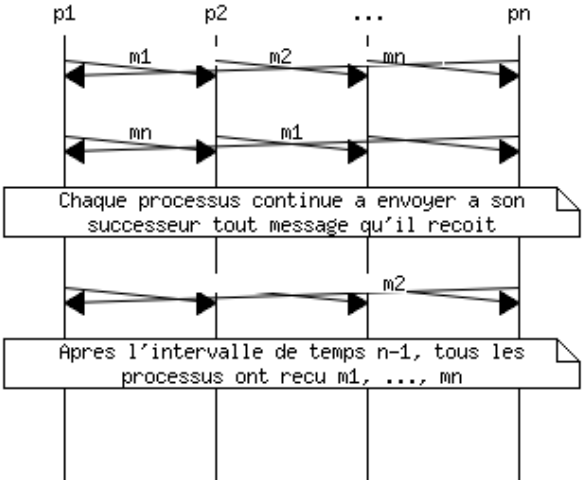


FIGURE 3.4 – Diagramme de séquence montrant le nombre d'intervalles de temps requis pour acheminer les messages si les n processus exploitent au mieux les capacités du switch



intervalle pour que p_s multicaste ce message, $n-1$ intervalles de temps pour que chaque processus acquitte la réception du message à p_s, et enfin un intervalle de temps pour que p_s multicaste un message indiquant que m peut être uot-livré. Au total, $1+1+(n-1)+1 = n+2$ intervalles de temps sont requis. $débit_{UOT}$ vaut donc $1/(n+2)$.

Pour les protocoles à base de privilège, [Guerraoui *et al.*, 2010] estime que, pour le protocole Totem [Amir *et al.*, 1995] représentatif de cette classe, $débit_{UOT}$ peut valoir au maximum 1. En effet, dans ce protocole, sur réception du jeton, un processus multicaste un message contenant le message à uot-diffuser et le transfert de jeton à un autre processus. Comme nous l'avons vu précédemment, dès qu'un protocole utilise du multicast, $\max debit_{UOT}$ est majoré par 1. Toutefois, comme nous le verrons à la section 5.2.4, pour le calcul de $\max debit_{UOT}$, Totem n'est pas le meilleur représentant des protocoles à base de privilège. En effet, le protocole des trains que nous développons dans cette thèse est plus performant.

Pour les protocoles à base d'historique des communications, [Guerraoui *et al.*, 2010] explique qu'ils ont en général un faible débit. En effet, ils induisent n^2 messages pour chaque uot-diffusion. Le protocole LCR fait toutefois exception. En effet, en disposant les processus sur un anneau virtuel, LCR limite le nombre de messages envoyés à $n-1$ pour chaque uot-diffusion. Par ailleurs, il optimise l'utilisation des capacités de transmission de messages simultanés du *switch*. Ainsi LCR permet d'atteindre le maximum théorique de $débit_{UOT}$. Nous en détaillons la démonstration. En effet, elle permet de mieux comprendre la démonstration que nous effectuerons pour le protocole des trains (cf section 5.2.3).

Démonstration. LCR *piggy-backe* les messages d'acquittement des messages déjà reçus sur les nouveaux messages à uot-diffuser. De ce fait, lors de la première série de $n-1$ intervalles de temps, même si chaque processus reçoit les n messages diffusés, *aucun* processus n'est autorisé à faire d'uot-livraison puisqu'il n'a pas reçu de message d'acquittement.

Lors de la deuxième série de $n-1$ intervalles de temps, chaque processus reçoit les messages d'acquittement *piggy-backés*. Les uot-diffusions de la première série sont donc réalisées.

Généralisons ce raisonnement : lors de la j^e ($j \geqslant 2$) série de $n-1$ intervalles de temps, les uot-diffusions des séries 1 à $j-1$ sont réalisées.

Donc : $débit_{UOT} = \frac{(j-1)\cdot n}{j\cdot(n-1)}$

De ce fait : $\lim_{j\to+\infty} debit_{UOT} = n/(n-1)$

□

La démonstration précédente repose sur le fait que n messages d'acquittement sont *piggy-backés* sur n uot-diffusions. De ce fait, à chaque tour, on a $(n+n)/(n-1)$ diffusions de messages. Or $2\cdot n/(n-1) > \mu_{max}$! Il est donc possible de dépasser μ_{max} et par extension $\max debit_{UOT}$. La section 5.2.3 (page 107) traite cet apparent paradoxe.

3.3.2 Influence du service de gestion des participants au protocole

Selon [Urban *et al.*, 2003], la plupart des protocoles d'uot-diffusion sont implantés à l'aide d'un service de gestion de groupes. Ces protocoles reprennent ainsi l'idée du système

Isis qui introduit un service de détection de défaillances en s'appuyant sur un service de gestion de groupes [Birman, 2010]. En procédant ainsi, ils simplifient la mise au point de leurs algorithmes dans le cas où il n'y a pas de défaillances. Un autre exemple est le protocole LCR de [Guerraoui *et al.*, 2010] qui s'appuie sur le service de gestion de groupes offert par l'intergiciel Spread [Spr98]. Par ailleurs, le protocole LCR comprend des algorithmes en l'absence de défaillances et des algorithmes en cas de changements dans la liste des participants. Il en est de même pour le protocole du train qui utilise un protocole à 3 phases pour calculer les membres du groupe [Cristian, 1991].

[Urban *et al.*, 2003] estime qu'on pourrait également gérer les défaillances de processus à l'aide d'un *détecteur de défaillances*. C'est pourquoi, à l'aide de simulations, il compare les performances en termes de latence de deux implantations de protocole d'uot-diffusion, l'une s'appuyant sur un détecteur de défaillances [Chandra et Toueg, 1996], l'autre s'appuyant sur un service de gestion de groupes utilisant une uot-diffusion à base de séquenceur fixe [Schiper *et al.*, 1991]. Il compare la latence, une fois que le système s'est stabilisé, selon 4 scénarios :

Scénario 1 il n'y a aucune défaillance, ni suspicion ;

Scénario 2 il y a des défaillances ;

Scénario 3 il n'y a aucune défaillance dans laquelle des processus corrects sont soupçonnés à tort d'avoir défailli ;

Scénario 4 il y a une latence transitoire après une défaillance.

[Urban *et al.*, 2003] observe les résultats suivants. Dans le scénario 1, il n'y a aucune différence entre les deux implantations. Dans le scénario 2 (respectivement 3 et 4), l'implantation à base de service de gestion de groupes est plus (respectivement moins) performante que l'implantation à base de détecteur de défaillances.

Il suggère donc en conclusion que les protocoles d'uot-diffusion doivent être implantés en combinant détecteur de défaillances et service de gestion de groupe. En effet, le détecteur de défaillances est plus réactif en cas de défaillances.

3.4 Conclusion

Dans ce chapitre, nous avons spécifié l'uot-diffusion. Puis, nous avons présenté les différentes classes de protocoles existantes : séquenceur fixe, séquenceur mobile, à base de privilège, à base d'historique des communications, et enfin accord des destinataires. Pour chaque classe, nous avons donné des exemples de protocoles issus de la littérature ou de l'industrie (notamment l'industrie du jeu vidéo).

Nous nous sommes ensuite intéressé aux performances de ces protocoles. Notre état de l'art montre que $d\acute{e}bit_{g\acute{e}n}$, le débit générique, est meilleur avec les protocoles à base de privilège qu'avec les protocoles à base d'historique des communications. De plus, optimiser le débit se fait au détriment de la latence. Par ailleurs, pour maximiser $d\acute{e}bit_{UOT}$, le débit en termes d'uot-livraisons, il faut utiliser des messages point-à-point et éviter les multicasts. Enfin, en combinant détecteur de défaillances et service de gestion de groupes, un protocole d'uot-diffusion gagne en réactivité en cas de défaillances.

Notre contribution au domaine de l'uot-diffusion est un protocole qui prend en compte l'ensemble de ces constats. Elle est présentée dans les chapitres suivants.

Chapitre 4

Protocole des trains

Le protocole des trains est inspiré de l'idée du protocole du train de [Cristian, 1991]. Dans ce dernier, un train de messages circule entre les processus participants répartis selon un anneau virtuel. En cas de départ ou d'arrivée d'un ou plusieurs processus, chaque processus participant exécute un algorithme à 3 phases pour décider des nouveaux membres du groupe des participants et déterminer les messages qui ont été uot-diffusés, mais non uot-livrés avant le changement dans la composition du groupe.

Dans ce chapitre, nous présentons nos contributions algorithmiques à [Cristian, 1991]. Ainsi, alors que [Cristian, 1991] ne fait qu'une description informelle du principe du protocole, nous décrivons ici complètement le protocole des trains. En outre, dans notre proposition, `ntr` trains ($\texttt{ntr} \in \mathbb{N}^*$) circulent en parallèle : comme nous le verrons au chapitre 5, cela nous permet d'obtenir des performances en termes de débit d'uot-diffusion supérieures à celles du meilleur des algorithmes actuels, en l'occurrence LCR [Guerraoui *et al.*, 2010]. Enfin, notre proposition intègre au protocole des trains la gestion de la liste des participants. Ainsi, nous n'avons pas besoin d'un algorithme à trois phases en cas de départ ou d'arrivée d'un ou plusieurs processus. D'ailleurs, grâce à l'horloge logique intégrée à chaque train, l'arrivée (respectivement le départ) d'un processus ne requiert que 2 (respectivement 1) établissement(s) de connexion TCP, et l'échange de $3 + \texttt{ntr}$ (respectivement $1 + \texttt{ntr}$) messages.

Ces contributions étendent nos travaux précédents [Eychenne et Simatic, 1996]. Ces derniers se cantonnaient à une version monotrain et ne fournissaient aucune preuve de la correction des algorithmes. De plus, nos travaux précédents manquaient de clarté quant aux principes algorithmiques utilisés : [Eychenne et Simatic, 1996] évoque un compteur qui s'avère être en fait une horloge logique. Enfin, nos travaux antérieurs ne géraient pas correctement le dépassement de capacité de l'horloge logique.

La section 4.1 présente le modèle du système et le modèle de performance que nous utilisons. La section 4.2 définit le vocabulaire dans le cadre du protocole des trains. Ensuite, la section 4.3 présente les algorithmes de la version monotrain du protocole des trains : un seul train circule entre les processus participant au protocole. Cette section a clairement une visée pédagogique : elle facilite la compréhension des algorithmes. Puis, la section 4.4 aborde la version multitrain (plusieurs trains circulent simultanément). Enfin, la section 4.5 conclut ce chapitre.

4.1 Modèles

Cette section présente le modèle du système (cf. section 4.1.1) et le modèle de performances (cf. section 4.1.2) que nous utilisons dans le cadre de notre contribution à l'uot-diffusion.

4.1.1 Modèle système

Notre modèle système est très proche du modèle système présenté dans [Guerraoui *et al.*, 2010]. En effet, le protocole des trains s'exécute au sein d'une petite grappe (« *cluster* » en anglais) de machines homogènes interconnectées par un réseau local. Chacun des n processus participant au protocole s'exécute sur une de ces machines (une machine peut héberger un ou plusieurs processus). Ces n processus sont répartis sur un anneau virtuel. De ce fait, chacun a une connexion TCP [1] ouverte avec son prédécesseur sur cet anneau et une autre connexion TCP ouverte avec son successeur sur cet anneau. Sur chacune de ces connexions TCP, les messages circulent en général toujours dans le même sens. Notre système supporte les arrêts francs de processus, les fautes par omission du réseau, et certaines fautes temporelles de processus. Si le processus p_p prédécesseur de p sur l'anneau virtuel fait un arrêt franc, p ouvre une nouvelle connexion TCP selon des modalités explicitées en section 4.3. Du fait de la simplicité des communications mises en œuvre, de l'homogénéité de l'environnement, et de la faible latence du réseau local, nous considérons que, quand la connexion TCP se rompt, c'est le processus à l'autre bout de la connexion qui a défailli [Dunagan *et al.*, 2004]. Par ailleurs, l'utilisation de TCP nous permet de gérer les fautes par omission du réseau. Toutefois, une simple connexion TCP est insuffisante pour détecter certaines fautes temporelles d'un processus. En effet, si un processus est suspendu suite à la réception du signal (Unix) `SIGTSTP`, comme c'est le système d'exploitation qui gère la connexion TCP, elle ne se ferme pas. Il en va de même si le processus rentre dans une boucle infinie qui l'empêche de consacrer du temps de calcul à la gestion du protocole des trains. C'est pourquoi une fois qu'un processus p a ouvert une connexion vers un processus q, q envoie périodiquement un message « battement de cœur » à p. Si p ne reçoit pas de message « battement de cœur » dans un certain laps de temps, il ferme la connexion TCP et se comporte comme si cette connexion avait été accidentellement fermée.

Remarques :

1. Nous utilisons TCP pour ses garanties suivantes : livraison des messages dans l'ordre de leur envoi, non-perte de messages (en l'absence de perte de processus ou de réseau), contrôle de flux entre 2 processus de manière à ne pas perdre d'octet, intégrité des messages (TCP garantit qu'aucun octet n'est modifié au cours de son transport), gestion des gros messages (qui peuvent nécessiter une fragmentation et un réassemblage), et enfin fermeture de la connexion en cas d'arrêt inopiné. La plupart de ces garanties sont également offertes par des intergiciels comme eNet [ENe03] ou bien JGroups [Ban02]. Nous pourrions donc envisager d'utiliser ces intergiciels en lieu et place de TCP. Il faudrait toutefois adapter les algorithmes du protocole des train,

1. Le flag `TCP_NODELAY` est positionné pour empêcher le gestionnaire TCP du système d'exploitation d'introduire un délai dans l'envoi des messages.

afin de pallier les garanties offertes par TCP et absentes de ces intergiciels.

2. Le réseau local auquel sont connectées toutes les machines de la grappe peut être implanté à l'aide d'un *switch*. Comme nous l'avons vu en section 3.3.1, le réseau peut donc acheminer des messages vers des processus différents dans un même intervalle de temps. Par ailleurs, il ne peut plus y avoir que deux types de fautes réseau : 1) le *switch* est défaillant et il n'y a alors plus aucun réseau ; 2) un câble réseau est rompu (ou débranché) entre une machine et le *switch*. Il ne peut donc jamais y avoir de partitionnement réseau où plusieurs machines de la grappe se retrouvent dans un sous-réseau pendant que les autres machines sont dans un autre sous-réseau : soit toutes les machines sont chacune dans un sous-réseau, soit plusieurs sont, chacune seule, dans un sous-réseau avec les autres machines connectées à un seul et même sous-réseau. Nous avons tiré parti de cette remarque pour simplifier la gestion des partitionnements (et donc l'implantation) dans le protocole des trains.

Dans cette section, nous avons défini le modèle du système où s'exécute le protocole des trains. Voyons maintenant le modèle de performances.

4.1.2 Modèle de performances

La section 4.1.1 décrit notre modèle système : une petite grappe de machines homogènes interconnectées par un réseau local (typiquement un *switch*). En conséquence, nous pouvons reprendre le modèle de performances décrit dans [Guerraoui *et al.*, 2010]. Nous le résumons ici. Chaque processus peut envoyer un message à un (*unicast*) ou plusieurs (*multicast*) processus au début de chaque tour. En revanche, ce processus peut recevoir un seul message d'un processus à la fin d'un tour. Si plus d'un message est envoyé au même processus pendant le même tour, ces messages seront reçus pendant différents tours.

4.1.3 Conclusion

Cette section a présenté le modèle système et le modèle de performances que nous utilisons pour le protocole des trains. Nous sommes maintenant en mesure de présenter ce protocole. Nous commençons par définir le vocabulaire utilisé.

4.2 Définitions

Cette section définit le vocabulaire utilisé dans le cadre du protocole des trains.

Le *circuit de train* (ou *circuit des trains*) est l'anneau virtuel sur lequel sont répartis les processus participant au protocole des trains. Nous employons l'expression *circuit de train* (respectivement *circuit des trains*) en version monotrain (respectivement multitrain) du protocole, car un seul *train* (respectivement plusieurs) circule(nt) sur le circuit.

Le *train* est un jeton qui circule sur le circuit. Il transporte les messages à uot-diffuser au sein de *wagons*.

Quand un processus p souhaite uot-diffuser un message, p l'ajoute à un *wagon*, c'est-à-dire une structure de données destinées à stocker les messages à uot-diffuser en attendant

le passage d'un train. Ce wagon est ajouté en queue du premier train qui passe.

Quand un processus p reçoit un train t, ce train est *récent* pour p quand t contient au moins un wagon que p reçoit pour la première fois. En revanche, ce train est *périmé* (ou *obsolète*) pour p quand t ne contient que des wagons que p a déjà reçus. Par extension, un train t est *récent* s'il existe un processus du circuit de train pour lequel t est récent. En revanche, t est *périmé* s'il est périmé pour tous les processus du circuit de train.

Chaque train contient une *estampille* qui est une simple *horloge logique* [Lamport, 1978] (respectivement une structure dont l'un des trois champs est une horloge logique) en version monotrain (respectivement multitrain).

Un message m est *stable* pour un processus p quand p a la garantie que tous les autres participants au protocole ont reçu m. Notez que la terminologie *message stable* est issue de [Guerraoui *et al.*, 2010]. [Chockler *et al.*, 2001] parle plutôt de *message sécurisé* (*safe message*, en anglais).

L'*adresse* d'un processus est la donnée qui identifie de manière unique un processus dans le cadre du protocole des trains.

L'*intergiciel protocolaire* est la couche logicielle qui met en œuvre le protocole des trains. La *couche applicative* s'adresse à cet intergiciel quand elle souhaite effectuer une uot-diffusion. L'intergiciel s'adresse à elle quand il peut effectuer une uot-livraison.

Cette section a présenté le vocabulaire utilisé tout au long de ce chapitre. Étudions maintenant la version monotrain du protocole des trains.

4.3 Version monotrain du protocole

Dans cette section, nous présentons les algorithmes de la version monotrain du protocole des trains : un seul train circule entre les processus participant au protocole.

La section 4.3.1 présente cette version de manière informelle. Ensuite, la section 4.3.2 spécifie les données utilisées pour la mise en œuvre du protocole des trains. Puis, la section 4.3.3 décrit les messages protocolaires échangés. Ensuite, la section 4.3.4 décrit les différentes fonctions et procédures utilisées par nos algorithmes. Puis, comme notre protocole est implanté à l'aide d'un automate à états, la section 4.3.5 présente l'algorithme de chacun de ces états. Dans le reste de cette section, nous analysons les principes qui sous-tendent ces différents algorithmes. Nous nous focalisons tout d'abord sur les principes directement liés à l'uot-diffusion (cf. section 4.3.6). Puis, nous détaillons les principes liés à la gestion du circuit de train (cf. section 4.3.7).

4.3.1 Présentation informelle de la version monotrain

Cette section présente de manière informelle la version monotrain. Notez que nous ne cherchons pas à être exhaustif dans cette section (notamment concernant les cas d'erreur). Nous souhaitons seulement faciliter la compréhension des algorithmes formels présentés à la section 4.3.5.

Le train ne circule que quand il y a au moins 2 processus qui participent au protocole. Il circule ensuite indéfiniment jusqu'à ce qu'il y ait strictement moins de 2 processus par-

ticipant au protocole. L'horloge logique du train est incrémentée de 1 à chaque fois que le train passe par un processus. Ainsi, dans le diagramme de séquence de la figure 4.1, 4 processus passent leur temps à faire circuler le train. Par exemple, le processus p_0 envoie le train $t_0(w_0)$ (c'est-à-dire le train d'horloge logique 0 transportant le wagon w_0) parce qu'il a reçu un train non représenté sur ce diagramme ; p_1 envoie $t_1(w_0, w_1)$ parce qu'il a reçu $t_0(w_0)$, etc. Quand un processus p est le seul participant au protocole et qu'un autre processus rejoint le protocole, c'est p qui est chargé de mettre en circulation le train.

Dans la suite de cette section, nous supposons qu'il y a au moins 2 processus participant au protocole. Notez qu'il y a alors autant de connexions TCP ouvertes que de participants, y compris dans le cas où il y a seulement deux participants (il serait alors possible de n'utiliser qu'une seule connexion TCP, mais ce cas particulier compliquerait inutilement les algorithmes).

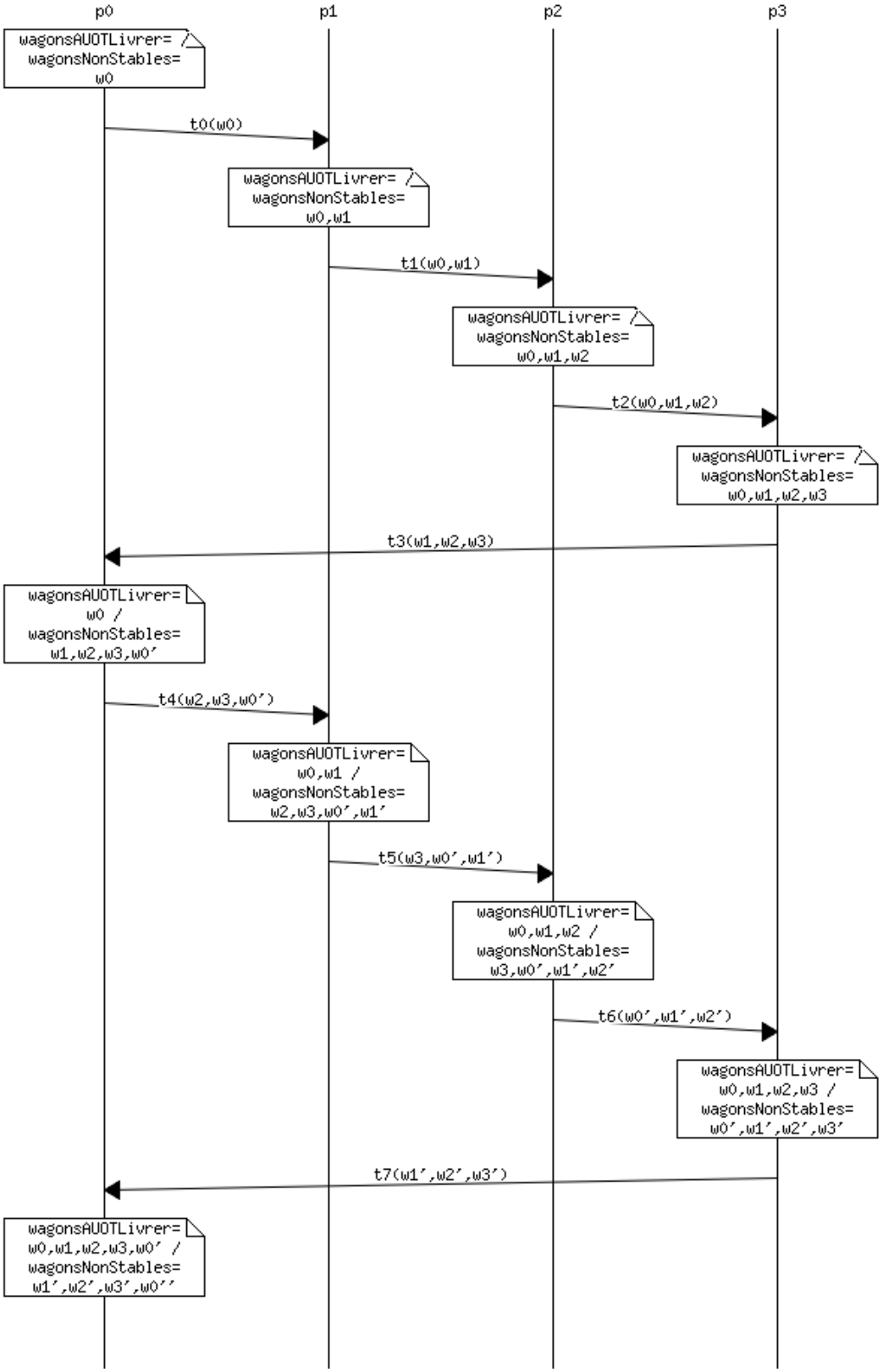
Quand un processus souhaite uot-diffuser un message, il le stocke dans un wagon. Par exemple, pendant que p_0 est en train d'envoyer le train t_0, si p_3 souhaite uot-diffuser un message, il le stocke dans le wagon w_3. Quand un processus reçoit un train, il ajoute le wagon des messages en attente à la fin du train qu'il envoie. Par exemple, quand p_3 reçoit le train t_2, il ajoute w_3 à la queue du train t_3 qu'il envoie. Quand un processus p reçoit un train, les messages contenus dans les wagons de ce train ne sont pas stables : p n'a aucune garantie que tous les autres participants ont reçu ces wagons. Par exemple, supposons que le processus p' a ajouté un wagon w à un train qui est ensuite reçu par p. p sait que tous les processus situés entre p' et p sur le circuit de train ont reçu w. En revanche, tous les processus situés entre p et p' n'ont pas reçu w. Aussi, comme les wagons ne sont pas stables, p les stocke dans la liste de wagons `wagonsNonStables`. Par exemple, sur réception de t_2, p_3 stocke les wagons w_0, w_1, w_2, w_3 dans `wagonsNonStables`. En revanche, quand un processus p reçoit un train, il a la garantie que tous les wagons qu'il avait stockés dans `wagonsNonStables` au tour de train précédent ont été vus par tous les participants. p peut donc les uot-livrer : ces wagons sont transférés de `wagonsNonStables` à `wagonsAUOTLivrer`. Ainsi, sur réception du train t_6, p_3 transfère les wagons w_0, w_1, w_2, w_3 de `wagonsNonStables` à `wagonsAUOTLivrer`.

Quand un processus envoie un train, il en retire le wagon qui appartient à son successeur dans le circuit de train. On économise ainsi de la ressource réseau. Par exemple, quand p_3 reçoit t_2, t_2 contient le wagon w_0 qui appartient à p_0, successeur de p_3 dans le circuit de train. Donc, p_3 retire le wagon w_0 du train t_3 qu'il envoie. La figure 4.1 montre que les wagons apparaissent dans le même ordre dans la variable `wagonsAUOTLivrer` des différents processus [2].

Étudions maintenant le comportement du protocole en cas de défaillance d'un processus p. De nombreux protocoles d'uot-diffusion s'appuient sur un service de gestion des membres extérieur au protocole [Birman, 2010, Chockler *et al.*, 2001]. Espérant les gains de réactivité promis par [Urban *et al.*, 2003] (et ainsi être conforme à la contrainte 5 présentée au chapitre 2), nous intégrons la gestion de défaillance de processus à nos algorithmes. De ce fait, en cas de défaillance d'un processus p, s'il y a plus de deux participants, nous reconstruisons le circuit de train de la manière suivante. La connexion TCP entre p et son prédécesseur p_p (respectivement successeur p_s) sur le circuit de train est rompue. p_p

2. Le lemme 13 page 103 démontre formellement la propriété d'ordre total.

FIGURE 4.1 – Diagramme de séquence de la circulation d'un train entre 4 processus



attend d'avoir un nouveau successeur. p_s établit une connexion avec p_p et lui envoie un message pour signifier à p_p que p_s est son nouveau successeur. p_p réagit en lui envoyant alors le dernier train qu'il a envoyé. Pour déterminer si un train reçu est récent ou périmé, un processus compare l'horloge logique du train reçu (noté *tr.st.lc*) avec celle du dernier train qu'il a envoyé (noté *dte.st.lc*). Si $tr.st.lc > dte.st.lc$, alors le train est récent. De ce fait, quand p_p renvoie à p_s le dernier train que p_p avait envoyé à p :

- si ce train est récent pour p_s, p_s le traite conformément à ce que nous avons décrit ci-dessus ;
- si ce train est périmé pour p_s, p_s l'ignore.

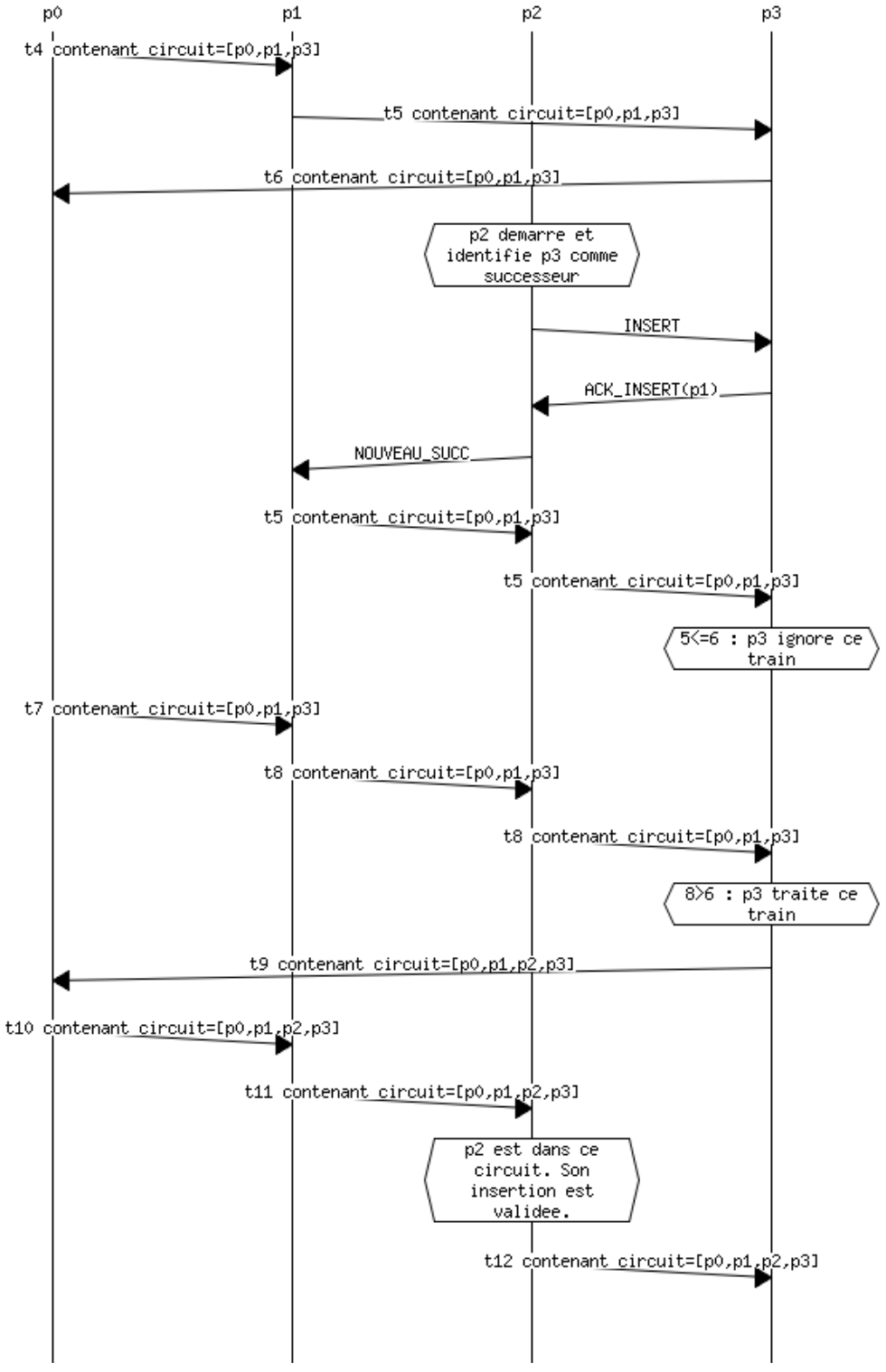
Par exemple, dans la figure 4.1, supposons que p_0 défaille juste après que p_1 envoie t_5. p_1 établit une connexion TCP avec p_3, prédécesseur de p_0 dans le circuit. p_3 envoie alors le dernier train qu'il a envoyé. Deux cas sont possibles :

- soit p_3 a déjà reçu t_6 et a donc envoyé t_7 à p_0. Par conséquent, t_7 s'est perdu avec la défaillance de p_0. p_3 envoie alors t_7 à p_1. Comme t_5 est le dernier train qu'a envoyé p_1 et que $7 > 5$, t_7 est récent pour p_1. Certes, t_7 contient le wagon w'_1 dont p_1 a déjà eu connaissance quand il a traité t_4. Mais, t_7 contient aussi les wagons w'_2 et w'_3 qui sont nouveaux pour p_1. C'est pourquoi le traitement de t_7 inclut la purge des wagons que p_1 connaît déjà (w'_1 dans notre exemple). De manière plus générale, cette purge consiste à retirer du train à traiter tout wagon émis par le processus lui-même ou bien par un processus qui ne fait plus partie du circuit de train ;
- soit p_3 n'a pas reçu t_6 et n'a donc pas envoyé t_7. Alors, p_3 envoie t_2 à p_1. Comme $2 < 5$, t_2 est obsolète pour p_1. Effectivement, t_2 ne contient que des messages dont p_1 a déjà eu connaissance : p_1 ignore le train t_2.

Étudions maintenant le cas où un nouveau processus p souhaite participer au protocole. Nous l'illustrons avec la figure 4.2 dans laquelle un processus p_2 s'insère dans un circuit de train $[p_0, p_1, p_3]$. Dans le cas où il y a déjà au moins un processus, p repère l'endroit où il veut s'insérer dans le circuit de train. Puis, il envoie le message `INSERT` à son futur successeur p_s (p_3 dans la figure 4.2). p_s note qu'il a un nouveau prédécesseur p et rompt sa connexion TCP avec son prédécesseur actuel p_p (p_1 dans la figure 4.2). p_s envoie ensuite le message `ACK_INSERT` à p avec l'adresse de p_p. p réagit en ouvrant une connexion avec p_p : p lui envoie le message `NOUVEAU_SUCC` pour signifier à p_p que p est son nouveau successeur. Chaque train transporte le circuit de train tel qu'il est perçu par le processus qui l'envoie. Tant que p ne se trouve pas dans le circuit d'un train qu'il reçoit, il fait suivre ce train à p_s sans aucun autre traitement. Quand p_s reçoit un train récent, il ajoute dans le circuit de ce train l'adresse de p. Puis, p_s fait suivre ce train. Ainsi, tous les autres participants au protocole sont avertis de l'insertion de p. *In fine*, p reçoit un train contenant cette information : il considère qu'il est désormais membre du circuit de train. Désormais, il peut incrémenter l'horloge logique et ajouter le wagon contenant les messages en attente d'être uot-diffusés.

Cette section a présenté de manière informelle la version monotrain. Afin de pouvoir être plus formel dans la présentation des algorithmes de cette version, la section suivante décrit les données qu'ils manipulent.

FIGURE 4.2 – Diagramme de séquence de l'insertion d'un processus (les wagons ne sont pas représentés pour simplifier le diagramme)



4.3.2 Données

Cette section décrit les données des algorithmes :

- `dte` : dernier train envoyé par le processus ;
- `myAddress` : adresse du processus qui exécute l'algorithme ;
- `nbAttentes` : nombre de fois que l'automate à états du processus est passé par l'état `AttenteAvantNouvelleTentativeInsertion` ;
- `NB_ATTENTES_MAX` : constante désignant le nombre maximum de fois qu'un processus peut passer par l'état `AttenteAvantNouvelleTentativeInsertion` ;
- `prédécesseur` : adresse du prédécesseur du processus courant ;
- `processusArrivés` : liste ordonnée des processus qui, depuis le dernier passage du train, ont informé le processus courant qu'ils souhaitaient rejoindre le circuit de train ;
- `processusPartis` : liste ordonnée des processus qui, selon le processus courant, ont quitté (volontairement ou involontairement) le circuit de train ;
- `successeur` : adresse du successeur du processus courant ;
- `tr` : dernier train reçu ;
- `wagonAEnvoyer` : wagon contenant les messages que le processus souhaite uot-diffuser. Ces messages sont en attente du passage du prochain train qui correspondra à l'uot-diffusion effective de ces messages ;
- `wagonsNonStables` : liste ordonnée de wagons non stables, chacun contenant une liste de messages provenant d'un des participants au protocole. Les wagons attendent dans cette liste jusqu'à ce que le processus reçoive la garantie que ces wagons sont stables ;
- `wagonsAUOTLivrer` : liste ordonnée de wagons, chacun contenant une liste de messages provenant d'un des participants au protocole et qui doivent être uot-livrés à la couche applicative [3].

Un wagon comprend :

- un champ `emetteur` contenant l'adresse de l'émetteur du wagon ;
- un champ `msgs` contenant la liste ordonnée de messages uot-diffusés par le processus émetteur.

Un train est structuré de la manière suivante :

- un entête contenant :
  - `st`, l'estampille de ce train. Dans la version monotrain, cette estampille est uniquement composée d'une horloge logique `lc` (pour « Logical Clock »). `lc` est un entier dont nous supposons pour l'instant qu'il ne peut pas être victime d'un dépassement de capacité [4]. `lc` sert à retrouver le train le plus récent sur le circuit en cas de perte du train ;
  - `circuit`, le circuit de train : ce champ contient la liste des identifiants de processus qui participent au protocole du train [5] ;
- `wagons`, la liste ordonnée des wagons transportés par ce train.

Ces données sont initialisées par l'algorithme 1.

3. Par exemple, cette uot-livraison peut être faite par un *thread* qui se charge d'extraire chaque wagon w de cette liste (en commençant par le début de cette liste), d'extraire de w chaque uot-diffusion m qu'il contient, et de livrer m à la couche applicative.

4. Voir la section 6.1.1 pour la gestion du dépassement de capacité.

5. Voir la section 6.1.2 pour un codage efficace de ce champ.

Algorithme 1 Initialisation des données de l'automate à états du protocole des trains (version monotrain)

```
1: nbAttentes ← 0
2: dte.st.lc ← 0
3: dte.circuit ← [ ]
4: processusArrivés ← [ ]
5: processusPartis ← [ ]
6: wagonAEnvoyer ← ⊥
7: wagonsNonStables ← [ ]
8: wagonsAUOTLivrer.emetteur ← myAddress
9: wagonsAUOTLivrer.msgs ← [ ]
10: prédécesseur ← ⊥
11: successeur ← ⊥
12: nextstate HorsCircuit
```

Cette section a présenté les données manipulées dans le cadre du protocole des trains. Nous étudions maintenant les messages échangés dans ce protocole.

4.3.3 Messages

Voici les différents messages échangés par les processus participant au protocole des trains :

- `ACK_INSERT` : réponse positive à une demande d'insertion dans le circuit de train (cf. message `INSERT`) ;
- `INSERT` : un processus p envoie ce message à un processus p_s pour demander à p_s que p soit inséré dans le circuit de train avant p_s ;
- `NAK_INSERT` : réponse négative à une demande d'insertion dans le circuit de train (cf. message `INSERT`) ;
- `NOUVEAU_SUCC` : un processus p envoie ce message à un processus p_p pour signaler à p_p que p est son nouveau successeur sur le circuit de train ;
- `PERTE_CONNEXION` : message reçu par un processus qui a perdu sa connexion TCP avec un autre processus ;
- `TRAIN` : message transportant les wagons, ces derniers contenant les uot-diffusions des différents processus ;
- uot-diffusion : message émis par l'application utilisatrice de l'intergiciel protocolaire ou bien l'intergiciel protocolaire lui-même. Ce message est à destination de la couche applicative des différents processus participant au protocole.

Cette section a présenté les messages échangés dans ce protocole. Voyons maintenant les fonctions et procédures sur lesquelles s'appuie ce protocole.

4.3.4 Fonctions et procédures

Cette section décrit les différentes fonctions et procédures utilisées par les algorithmes présentés à la section 4.3.5 et dont le fonctionnement n'a pas besoin d'être détaillé par un

algorithme :

- `append(uneListe,unElément)` : procédure ajoutant l'élément `unElément` à la liste `uneListe` ;
- `envoyer(unMsg,uneAdresse)` : procédure envoyant le message `unMsg` au destinataire d'adresse `uneAdresse` *via* la connexion TCP précédemment établie avec ce destinataire. Dans le cas où le destinataire défaille pendant (respectivement a défailli avant) l'envoi de `unMsg`, cette procédure fait comme si elle avait pu effectivement envoyer tous les octets au destinataire. Or, elle n'en envoie qu'un certain nombre d'octets (respectivement aucun octet). Notez que cette procédure ne retourne pas avant d'avoir envoyé le message demandé (éventuellement partiellement, en cas de rupture de connexion). De ce fait, deux appels successifs à cette procédure ne risquent pas de mélanger les octets des messages envoyés ;
- `extend(uneListe,uneListe2)` : procédure ajoutant les éléments de la liste `uneListe2` à la liste `uneListe` ;
- `fermerConnexion(uneAdresse)` : procédure fermant la connexion TCP qui avait été précédemment établie avec le processus d'adresse `uneAdresse` ;
- `majCircuit(unCircuit,uneAdresse,listeProcessusArrivés,listeProcessusPartis)` : fonction renvoyant un circuit calculé à partir de `unCircuit` en lui ajoutant chaque adresse de processus trouvée dans `listeProcessusArrivés` avant le processus d'adresse `uneAdresse`, et en lui retirant toutes les adresses trouvées dans `listeProcessusPartis`. Pour gérer le cas où un processus s'insère plusieurs fois dans le circuit de train et le quitte plusieurs fois, dans le cas où une adresse apparaît plusieurs fois dans `listeProcessusArrivés` (respectivement `listeProcessusPartis`), la fonction ajoute (respectivement retire) plusieurs fois cette adresse au (respectivement du) circuit. Au final, l'adresse apparaît zéro ou bien une fois dans le circuit ;
- `msgInfoArrivée(uneAdresse,uneListeParticipants)` : fonction renvoyant un message de type uot-diffusion permettant à l'intergiciel d'informer la couche applicative de tous les processus qu'un processus d'adresse `uneAdresse` est arrivé au sein de la liste `uneListeParticipants` ;
- `msgsInfoDépart(uneListeAdresses)` : fonction renvoyant une liste de messages de type uot-diffusion permettant à l'intergiciel d'informer la couche applicative du départ de processus du protocole des trains (il y a un message par adresse trouvée dans `uneListeAdresses`) ;
- `nextstate(nouvelÉtat)` : procédure faisant passer l'automate dans l'état `nouvelÉtat`. Notez que les éventuelles instructions situées derrière l'invocation de `nextstate(nouvelÉtat)` sont ignorées (en ce sens, `nextstate()` a un comportement similaire à celui d'une instruction `goto`) ;
- `ouvrirConnexion(uneAdresse)` : fonction permettant d'ouvrir une connexion TCP vers le processus d'adresse `uneAdresse`. Elle renvoie `SUCCÈS` si la connexion a été établie et `ÉCHEC` sinon ;
- `participerAuProtocole(unBooléen)` : procédure appelée par un processus p avec le paramètre `unBooléen` à `true` (respectivement `false`) pour que p indique qu'il veut (respectivement ne veut pas) participer au protocole. En particulier, il est (respectivement n'est pas) à l'écoute de demandes de connexion d'autres processus participant

au protocole. Il y répond positivement (respectivement négativement) ;
- `prédécesseurDe(uneAdresse,uneListeParticipants)` : fonction renvoyant le prédécesseur du participant d'adresse `uneAdresse` dans la liste de participants au protocole `uneListeParticipants` ;
- `rechercherSuccesseur()` : Fonction renvoyant l'adresse d'un processus participant au protocole des trains [6] ;
- `recevoirMsg()` : fonction fournissant un message reçu (sur n'importe quel canal TCP ouvert). Cette fonction bloque jusqu'à ce qu'un message soit reçu. Dans le cas où une connexion est rompue, cette fonction renvoie `PERTE_CONNEXION`. Notez que cette fonction est atomique du point de vue de la réception des messages : si une connexion est rompue alors que `recevoirMsg()` était en train de recevoir un message sur cette connexion, cette fonction ignore ce message incomplet et renvoie `PERTE_CONNEXION` ;
- `sauvegardeJusquAuProchainNextstate(unMsg)` : procédure similaire à l'instruction `save` du langage *SDL*[7]. Vu que le protocole est implanté à l'aide d'un automate à états, cette procédure permet de sauvegarder le message `unMsg` jusqu'à ce que l'automate appelle l'instruction **nextstate** pour passer dans un nouvel état ou bien rester dans le même état [8]. Le message est alors à nouveau présenté à l'automate.

Cette section a présenté les fonctions et procédures dont le fonctionnement n'a pas besoin d'être spécifié par un algorithme. Nous sommes maintenant en mesure de détailler les algorithmes du protocole des trains.

4.3.5 Algorithmes

Cette section présente les algorithmes du protocole des trains, version monotrain. Ce protocole est mis en œuvre à l'aide d'un automate à états. Cet automate possède 4 états. La figure 4.3 présente son diagramme de machine à états.

Pour aider à comprendre le lien entre cet automate à états et la couche applicative, voici l'algorithme 2 qui décrit la procédure `uotDiffusion(unMessage)`. Cette dernière permet à la couche applicative de signifier à l'intergiciel (mettant en œuvre le protocole des trains) qu'elle souhaite uot-diffuser le message `unMessage`. Cette procédure ne peut être appelée que quand le protocole des trains est initialisé, c'est-à-dire que l'automate à états est dans l'état `DansCircuitSeul` ou bien `DansCircuitÀPlusieurs`.

Dans la suite, nous décrivons chaque état en langage algorithmique. En effet, nous estimons ce dernier langage plus approprié à présenter nos idées qu'un langage dédié aux automates à états comme *SDL*, *LOTOS*, *Esterel*, etc.

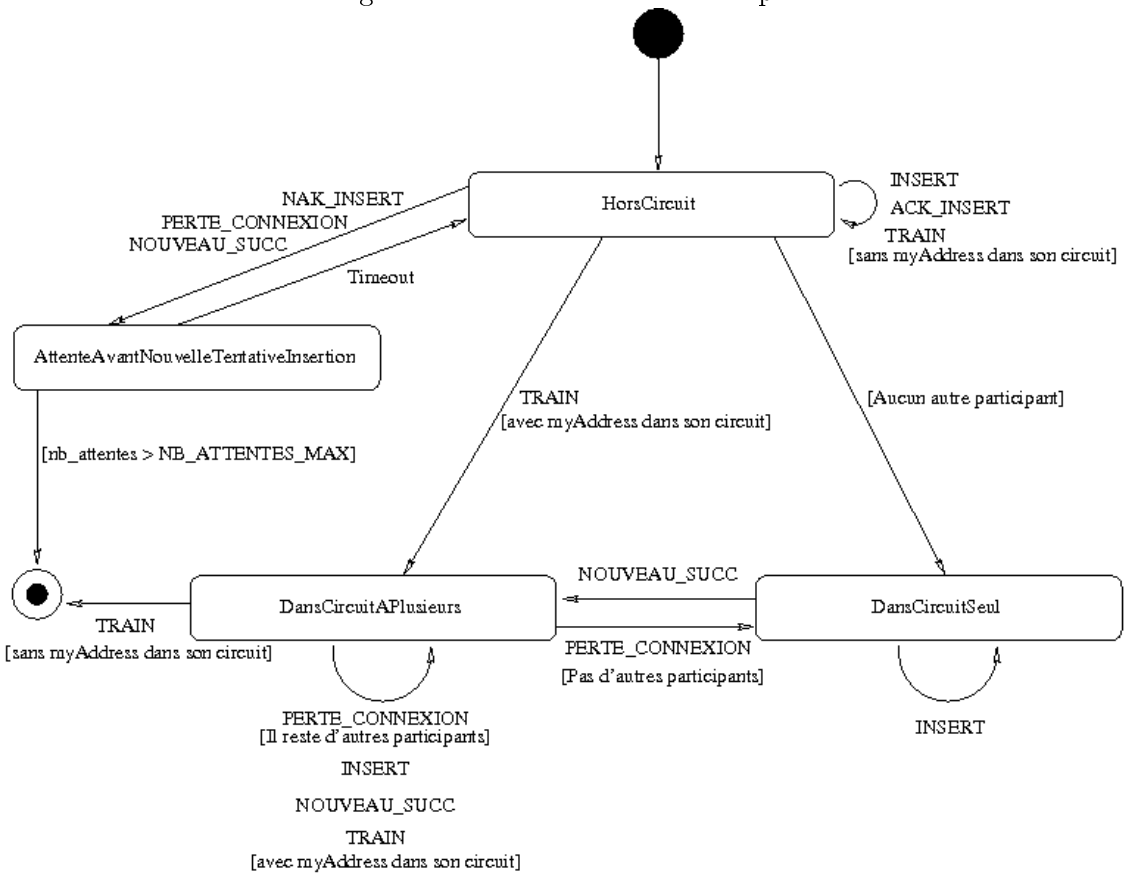
L'état `HorsCircuit` (cf. algorithme 3 page 61) est le premier état par lequel passe l'automate une fois qu'il a initialisé ses données (cf. algorithme 1 page 56). L'état `HorsCircuit`

6. Voir la section 6.1.4 pour les choix de mise en œuvre.

7. *SDL* est ici le sigle de *Specification and Description Language* qui n'a aucun lien avec la librairie graphique *Simple DirectMedia Layer*.

8. `sauvegardeJusquAuProchainNextstate` est ici différente de l'instruction `save` de *SDL*. En effet, en *SDL*, un message est sauvegardé jusqu'à ce que l'automate passe dans un état autre que l'état dans lequel il a été sauvegardé. Nous avons adopté une sémantique légèrement différente, car certains états décrits en sections 4.3 et 4.4 s'écriraient avec plusieurs états en *SDL*.

FIGURE 4.3 – Diagramme de machine à états du protocole des trains



Algorithme 2 Algorithme de la procédure `utoDiffusion(unMessage)` (version monotrain et multitrain)

```
1: append(wagonAEnvoyer.msgs,unMessage)
2: if étatAutomate == DansCircuitSeul then
3:    // Aucun train ne circule : on uot-livre le message immédiatement.
4:    append(wagonsAUOTLivrer,wagonAEnvoyer)
5:    wagonAEnvoyer.msgs ← [ ]
6: end if
```

permet au processus de déterminer s'il existe d'autres processus sur le réseau qui sont membres du circuit de train. Si c'est le cas, il identifie un successeur et lui envoie le message `INSERT` pour lui signifier qu'il veut s'insérer dans le circuit de train. Si ce n'est pas le cas, l'automate se considère seul dans le circuit de train : il passe dans l'état `DansCircuitSeul`.

L'état `AttenteAvantNouvelleTentativeInsertion` (cf. algorithme 4 page 62) est utilisé dans le cas où le processus détecte une tentative d'insertion simultanée avec un autre processus. Cet état lui permet d'attendre un laps de temps aléatoire. Cela garantit qu'un seul processus cherche à s'insérer dans le circuit de train à un instant donné : les algorithmes et leurs preuves de bon fonctionnement en sont simplifiées.

L'état `DansCircuitSeul` (cf. algorithme 5 page 62) sert dans le cas où un processus se considère seul sur le circuit de train. Il lui permet d'attendre la demande d'insertion d'un nouveau processus sur le circuit de train.

Enfin, l'état `DansCircuitÀPlusieurs` (cf. algorithme 6 page 63) sert aux processus membres du circuit de train (avec plusieurs membres dans ce circuit). Il permet à un processus de gérer la réception d'un train en testant s'il est récent ou périmé (fonction `estCeQueTrainRécent(tr.st.lc,dte.st.lc)` décrite par l'algorithme 8 page 64) et en le traitant s'il est récent (procédure `traiterTrainRécent` décrite par l'algorithme 7 page 64). L'état `DansCircuitÀPlusieurs` permet également de gérer les pertes de connexions TCP et les demandes d'insertion de nouveaux processus dans le circuit.

Cette section a présenté les algorithmes du protocole des trains (version monotrain). La section suivante détaille le rôle de certaines lignes contribuant à l'uot-diffusion.

4.3.6 Principes algorithmiques directement liés à l'uot-diffusion

Cette section analyse les principes qui sous-tendent les algorithmes présentés en section 4.3.5 en se concentrant sur la dimension uot-diffusion.

Quand la couche applicative d'un processus p souhaite uot-diffuser un message m, elle appelle `uotDiffusion(m)` : m est ajouté à `wagonAEnvoyer.msgs` (ligne 2.1 [9]).

Si p est seul sur le circuit de train, `wagonAEnvoyer` est immédiatement ajouté à `wagonsAUOTLivrer` (lignes 2.2 et 2.4). In fine, m est livré à la couche applicative.

Si p n'est pas seul sur le circuit de train, montrons tout d'abord qu'un train récent parvient *in fine* au niveau de p, ce qui fait que p ajoutera `wagonAEnvoyer` (qui contient m) au train destiné à son successeur. Soit p_0 (respectivement p_1) le processus qui arrive en premier (respectivement second) sur le circuit de train. p_1 envoie `NOUVEAU_SUCC` à p_0 pour lui signifier qu'il est son nouveau successeur. p_0 réagit en envoyant un train dont le circuit contient p_0 et p_1 (lignes 5.20 et 5.22). p_0 passe ensuite dans l'état `DansCircuitÀPlusieurs`. En recevant ce train, p_1 réagit en envoyant un train (ligne 3.38) dont l'identifiant est l'identifiant du train reçu plus un (lignes 3.35 et 3.36). p_1 passe ensuite dans l'état `DansCircuitÀPlusieurs`. Quand p_0 reçoit ce train, comme son identifiant est strictement supérieur à l'identifiant du dernier train que p_0 a envoyé, p_0 estime que ce train est récent (ligne 6.5). Il le traite donc : p_0 envoie un train (ligne 6.7) dont l'identifiant est l'identifiant

9. La notation « ligne 2.1 » signifie « ligne 1 de l'algorithme 2 ». La notation « lignes 2.1-5 » signifie « lignes 1 à 5 de l'algorithme 2 ».

Algorithme 3 État `HorsCircuit` de l'automate à états du protocole des trains (version monotrain)

```
1: participerAuProtocole(true)
2: successeur ← rechercherSuccesseur()
3: if successeur == myAddress then
4:   append(wagonAEnvoyer.msgs,msgInfoArrivée(myAddress,[myAddress]))
5:   append(wagonsAUOTLivrer,wagonAEnvoyer)
6:   wagonAEnvoyer.msgs ← [ ]
7:   nextstate DansCircuitSeul
8: end if
9: if ouvrirConnexion(successeur) == ÉCHEC then
10:  nextstate AttenteAvantNouvelleTentativeInsertion
11: end if
12: envoyer(INSERT,successeur)
13: repeat // Attente ACK_INSERT
14:   msg ← recevoirMsg()
15:   switch (msg.type)
16:   case NAK_INSERT or PERTE_CONNEXION:
17:     nextstate AttenteAvantNouvelleTentativeInsertion
18:   case INSERT:
19:     envoyer(NAK_INSERT(),msg.émetteur)
20:   end switch
21: until msg.type == ACK_INSERT
22: prédécesseur ← msg.ACK_INSERT.préd
23: if ouvrirConnexion(prédécesseur) == ÉCHEC then
24:   nextstate AttenteAvantNouvelleTentativeInsertion
25: end if
26: envoyer(NOUVEAU_SUCC,prédécesseur)
27: repeat // Attente confirmation que processus dans le circuit
28:   msg ← recevoirMsg()
29:   switch (msg.type)
30:   case PERTE_CONNEXION or NOUVEAU_SUCC:
31:     nextstate AttenteAvantNouvelleTentativeInsertion
32:   case INSERT:
33:     envoyer(NAK_INSERT(),msg.émetteur)
34:   case TRAIN:
35:     if myAddress ∈ msg.TRAIN.circuit then
36:       msg.TRAIN.st.lc ← msg.TRAIN.st.lc +1
37:     end if
38:     envoyer(msg,successeur)
39:     dte ← msg.TRAIN
40:   end switch
41: until msg.type == TRAIN and myAddress ∈ dte.circuit
42: append(wagonAEnvoyer.msgs,msgInfoArrivée(myAddress,dte.circuit))
43: nextstate DansCircuitÀPlusieurs
```

Algorithme 4 État `AttenteAvantNouvelleTentativeInsertion` de l'automate à états du protocole des trains (version monotrain et multitrain)

```
1: participerAuProtocole(false)
2: fermerConnexion(prédécesseur)
3: fermerConnexion(successeur)
4: prédécesseur ← ⊥
5: successeur ← ⊥
6: if nbAttentes > NB_ATTENTES_MAX then
7:    Signaler l'échec d'insertion à la couche applicative : elle décidera des suites à donner
8: end if
9: attente(valeur aléatoire entre 0 et 2^nbattentes*T)// T, constante de temps, est
   le temps maximum que peut attendre un processus lors de sa première attente
10: nbAttentes ← nbAttentes+1
11: nextstate HorsCircuit
```

Algorithme 5 État `DansCircuitSeul` de l'automate à états du protocole des trains (version monotrain)

```
1: prédécesseur ← myAddress
2: successeur ← myAddress
3: repeat // Attente demande d'insertion de la part d'un autre processus
4:    msg ← recevoirMsg()
5: until msg.type == INSERT
6: envoyer(ACK_INSERT(myAddress),msg.émetteur)
7: prédécesseur ← msg.émetteur
8: repeat // Attente connexion établie avec le successeur
9:    msg ← recevoirMsg()
10:   switch (msg.type)
11:   case PERTE_CONNEXION:
12:      nextstate DansCircuitSeul
13:   case INSERT:
14:      sauvegardeJusquAuProchainNextstate(msg)
15:   end switch
16: until msg.type == NOUVEAU_SUCC
17: successeur ← msg.émetteur
18: // On met en circulation le train en l'envoyant vers le successeur
19: dte.st.lc ← dte.st.lc +1
20: dte.circuit ← [myAddress, prédécesseur]
21: dte.wagons ← [ ]
22: envoyer(dte,successeur)
23: nextstate DansCircuitÀPlusieurs
```

Algorithme 6 État `DansCircuitÀPlusieurs` de l'automate à états du protocole des trains (version monotrain)

```
1: msg ← recevoirMsg()
2: switch (msg.type)
3: case TRAIN:
4:   if myAddress ∈ msg.TRAIN.circuit then
5:     if estCeQueTrainRécent(tr.st,dte.st) then // cf. algorithme 8
6:       traiterTrain(tr,dte,wagonsAEnvoyer,processusPartis,processusArrivés)
                                                                  // cf. algorithme 7
7:       envoyer(dte,successeur)
8:     end if
9:   else
10:    Signaler problème de partitionnement à la couche applicative : elle décidera des suites à
       donner (suicide par défaut)
11:  end if
12: case INSERT:
13:  fermerConnexion(prédécesseur)
14:  envoyer(ACK_INSERT(prédécesseur),msg.émetteur)
15:  prédécesseur ← msg.émetteur
16:  append(processusArrivés,prédécesseur)
17: case NOUVEAU_SUCC:
18:  fermerConnexion(successeur)
19:  successeur ← msg.émetteur
20:  envoyer(dte,successeur)
21: case PERTE_CONNEXION:
22:  if connexion perdue avec prédécesseur then
23:    // On cherche à se connecter à un prédécesseur en commençant par le prédécesseur actuel
24:    while prédécesseur ! = myAddress do
25:      if ouvrirConnexion(prédécesseur) == SUCCÈS then
26:        // Le prédécesseur trouvé est opérationnel
27:        envoyer(NOUVEAU_SUCC,prédécesseur)
28:        nextstate DansCircuitÀPlusieurs
29:      end if
30:      // Le prédécesseur trouvé a aussi quitté le circuit
31:      append(processusPartis,prédécesseur)
32:      prédécesseur ← prédécesseurDe(prédécesseur,dte.circuit)
33:    end while
34:    // Il n'y a plus aucun autre processus sur le circuit
35:    extend(wagonAEnvoyer.msgs,msgsInfoDépart(processusPartis))
36:    processusPartis ← [ ]
37:    extend(wagonsAUOTLivrer,wagonsNonStables)
38:    wagonsNonStables ← [ ]
39:    append(wagonsAUOTLivrer,wagonAEnvoyer)
40:    wagonAEnvoyer.msgs ← [ ]
41:    nextstate DansCircuitSeul
42:  else // Connexion perdue avec successeur
43:    successeur ← ⊥
44:  end if
45: end switch
46: nextstate DansCircuitÀPlusieurs
```

Algorithme 7 Traitement de la réception d'un train récent par un processus membre du circuit de train (version monotrain)

```
1: extend(wagonsAUOTLivrer,wagonsNonStables)
2: wagonsNonStables ← [ ]
3: dte.circuit ←
        majCircuit(tr.circuit,myAddress,processusArrivés, processusPartis)
4: processusArrivés ← [ ]
5: extend(wagonAEnvoyer.msgs,msgsInfoDépart(processusPartis))
6: processusPartis ← [ ]
7: dte.wagons ← [ ]
8: for all w ∈ tr.wagons tel que emetteur(w) ∈ dte.circuit \ { myAddress } do
9:   append(wagonsNonStables,w)
10:  if emetteur(w) != successeur then
11:    append(dte.wagons,w)
12:  end if
13: end for
14: append(wagonsNonStables,wagonAEnvoyer)
15: append(dte.wagons,wagonAEnvoyer)
16: wagonAEnvoyer.msgs ← [ ]
17: dte.st.lc ← tr.st.lc +1
```

Algorithme 8 Fonction `estCeQueTrainRécent(tr_st,dte_st)` sans gestion des dépassements de capacité (version monotrain)

```
1: return (tr_st.lc > dte_st.lc)
```

du train reçu plus un (ligne 7.17). p_1 fait ensuite un traitement semblable à p_0, etc. Ainsi, le train circule perpétuellement. Si un nouveau processus p' arrive, le traitement de son insertion se déroule de manière similaire à celle de p_1 à une nuance près : une fois que le processus p' est inséré avant son successeur p'_s, tant que p' n'a pas reçu de train contenant son adresse, p' se contente de faire suivre sans aucun traitement les trains qu'il reçoit. Quand p'_s reçoit un train récent de p', p'_s intègre l'adresse de p' au circuit de train et fait suivre ce train. De ce fait, *in fine*, p' reçoit un train avec un circuit où p' apparaît. Ce n'est pas forcément le circuit que p'_s avait mis dans le train puisqu'un processus après p'_s a pu modifier ce circuit pour signaler, par exemple, le départ d'un processus. Mais, c'est un circuit où il y a p'. p' réagit en envoyant un train (ligne 3.38) dont l'identifiant est l'identifiant du train reçu plus un (lignes 3.35 et 3.36). p' passe ensuite dans l'état `DansCircuitÀPlusieurs`. En cas d'insertion, le train circule perpétuellement. Traitons maintenant le cas où un processus p'' quitte le circuit de train [10]. S'il reste au moins 2 processus participants au protocole, au minimum, le prédécesseur p''_p et le successeur p''_s de p'' participent encore au protocole. En tant que prédécesseur, p''_p ne fait que noter la perte de connexion avec p'' (lignes 6.21, 6.22 et 6.43). En effet, c'est au successeur p''_s que revient la charge d'établir la connexion entre p''_p et p''_s pour réparer le circuit de train (lignes 6.22–23). p''_s envoie `NOUVEAU_SUCC` à p''_p. p''_p réagit en envoyant son `dte` (lignes 6.17 et 6.20). Si ce train est récent pour p''_s, p''_s le traite et le train continue à circuler. Si ce train est périmé pour p''_s, cela signifie que p''_s a précédemment envoyé un train plus récent à son propre successeur : ce train progresse actuellement sur le circuit de train. *In fine*, il parvient à p''_p qui envoie alors un train récent à p''_s. Le train continue donc aussi à tourner. S'il reste moins de 2 processus participant au protocole après la défaillance de p'', le train ne circule plus. Par rapport au problème initial « p a stocké une uot-diffusion m dans `wagonAEnvoyer` », seuls 2 cas sont possibles : 1) si c'est p qui est le survivant, p uot-livre `wagonAEnvoyer` (lignes 6.21, 6.33 et 6.39) avant de passer dans l'état `DansCircuitSeul` ; 2) si c'est un autre processus qui a survécu, p a défailli avant de pouvoir uot-diffuser son message. Par conséquent, *in fine*, `wagonAEnvoyer` de p est emporté par un train (ou bien uot-livré si p est le seul survivant ou bien encore perdu si p est défaillant).

Détaillons maintenant ce qui se passe lorsqu'un processus p reçoit un train récent. p commence par uot-livrer les wagons qui n'étaient pas stables (ligne 7.1). En effet, si p vient de recevoir un train récent, c'est parce que son prédécesseur p_p avait lui-même reçu un train récent. p_p avait reçu un train récent parce que son propre prédécesseur lui avait envoyé un train récent. Et ainsi de suite jusqu'au moment où le successeur p_s de p avait reçu un train récent parce que son prédécesseur (p en l'occurrence) lui avait envoyé un train récent (en l'occurrence `dte`). p a donc la garantie que les wagons stockés dans `wagonsNonStables` avant l'envoi de `dte` ont été vus par l'ensemble des autres processus du circuit : ces wagons sont stables. Par conséquent, p peut les uot-livrer. Ainsi, le protocole des trains garantit la propriété d'accord uniforme (cf. propriété 2), sans avoir besoin qu'un message à uot-diffuser fasse deux tours de l'anneau virtuel comme dans [Cristian, 1991], ni besoin de faire circuler des informations de validation via *piggy-backing* sur les messages uot-diffusés comme dans [Guerraoui *et al.*, 2010].

10. Dans la suite, lorsqu'un processus quitte le circuit de train, nous considérons en général que ce départ est volontaire ou involontaire (défaillance du processus). Si ce n'est pas le cas, nous le mentionnons explicitement.

Une fois que p a uot-livré les wagons de `dte`, p calcule le circuit de train qu'il va stocker dans le train qu'il s'apprête à envoyer (ligne 7.3). Il part du circuit `tr.circuit`. Il y insère les processus mentionnés dans `processusArrivés` (nous verrons en section 4.3.7 que ces processus sont ceux qui se sont signalés à p par un message `INSERT`) avant sa propre position dans le circuit. Enfin, p retire les processus mentionnés dans `processusPartis` (c'est-à-dire ceux dont p a détecté le départ).

Par ailleurs, pour chaque processus p_i listé dans `processusPartis`, p ajoute à `wagonAEnvoyer` un message (ligne 7.5). Ce message signale le départ de p_i à la couche applicative de tous les participants au protocole. Tous ces messages sont intégrés au flux des uot-diffusions faites par la couche applicative de manière à être vus par tous les processus dans le même ordre au moment de leur uot-livraison.

Ensuite, p détermine les wagons qu'il peut considérer comme non stables et qui seront donc uot-livrés au tour de train suivant. p ne peut pas se permettre une simple recopie des wagons de `tr` dans `wagonsNonStables`, sinon p risque d'uot-livrer deux fois le même message : l'intégrité uniforme (propriété 3) serait mise à mal. Ainsi, dans la figure 4.1, imaginons que les processus p_1 et p_2 défaillent juste après que p_0 ait envoyé le train $t_4(w2, w3, w0')$. p_3 se connecte à p_0 et lui envoie le message `NOUVEAU_SUCC`. p_0 réagit en envoyant le dernier train qu'il a envoyé, c'est-à-dire t_4. Si p_3 recopiait directement les wagons de t_4 dans `wagonsNonStables`, cette variable contiendrait $[w_0, w_1, w_2, w_3, w_2, w_3, w'_0]$, soit deux fois w_2 et w_3. Pour éviter ce problème, p initialise `wagonNonStables` à `[ ]` (ligne 7.2). Puis, il n'ajoute que les wagons dont l'émetteur fait partie du circuit de train et qui n'est pas p (lignes 7.8 et 7.9). Cela garantit que p ne recopie pas de doublons dans `wagonsNonStables`. En effet, si seul le prédécesseur p_p de p est défaillant, le wagon w_p ajouté par p au passage précédent du train n'a pas été retiré du train par p_p (comme nous le verrons ci-dessous, en temps normal, p_p recopie avec la ligne 7.11 les wagons du train qu'il reçoit dans le train qu'il envoie, sauf pour son successeur, p en l'occurrence, grâce au test de la ligne 7.10). En ne transférant pas dans `wagonsNonStables` le wagon dont p est l'émetteur, nous pallions ce problème. De plus, si le prédécesseur p_{p_p} de p_p est défaillant en même temps que p_p, p_{p_p} n'a pas retiré le wagon w_{p_p} que p_p avait ajouté dans le train au tour précédent. Ce train étant déjà passé par p, p connaît déjà ce wagon w_{p_p}. Or, p reçoit à nouveau ce wagon parce qu'il a détecté les défaillances de p_p et p_{p_p}. p sait donc que p_p et de p_{p_p} ne font plus partie du circuit de train. En effet, parce que p a détecté les défaillances de p_p et de p_{p_p}, p a envoyé `NOUVEAU_SUCC` au prédécesseur de p_{p_p} et ce prédécesseur a envoyé un train contenant le wagon w_{p_p}. En ne transférant pas dans `wagonsNonStables` le wagon w_{p_p} dont l'émetteur p_p ne fait plus partie du circuit, nous pallions le problème que p_{p_p} n'a pas retiré ce wagon. Le même raisonnement s'applique si on suppose que le prédécesseur de p_{p_p} a aussi défailli.

Une fois que p a déterminé les wagons de `tr` qu'il peut considérer comme non stables, il ajoute `wagonAEnvoyer` à `wagonsNonStables` (ligne 7.14). En effet, `wagonAEnvoyer` appartient au train que p va envoyer (cf. paragraphe ci-dessous). Donc, au prochain tour de train, p aura la garantie que tous les autres processus du circuit ont reçu `wagonAEnvoyer`.

p construit ensuite le train destiné à son successeur p_s. Ce train est constitué de tous les wagons de `wagonsNonStables` (lignes lignes 7.8, 7.11 et 7.15), sauf le wagon de p_s de p (d'où le test de la ligne 7.10). En effet, p_s a déjà mémorisé ce wagon dans sa propre variable `wagonNonStables`. Il est inutile de consommer inutilement de la ressource réseau

en envoyant à p_s ce wagon qu'il connaît déjà.

Enfin, p augmente `dte.st.lc` (ligne 7.17) de manière à ce que p puisse être conscient que c'est lui, parmi les processus corrects, qui détient le train le plus récent.

Après nous être concentré sur les lignes des algorithmes contribuant à la dimension uot-diffusion, tournons-nous vers les lignes ayant trait à la gestion du circuit de train.

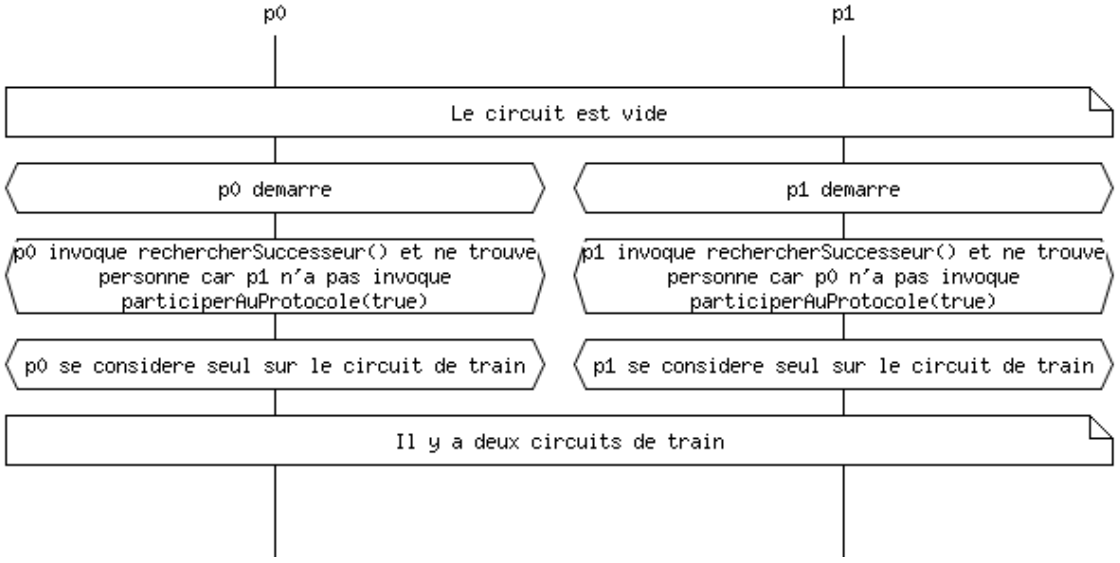
4.3.7 Principes algorithmiques liés à la gestion du circuit de train

Dans cette section, nous nous concentrons sur les principes liés à la gestion du circuit de train.

À la section 4.3.6, nous avons vu que, quand un processus p quitte le circuit de train : 1) son prédécesseur p_p ne fait que noter la perte de connexion p ; 2) son successeur p_s se charge d'établir la connexion entre p_p et p_s pour réparer le circuit de train. L'insertion d'un processus dans le circuit de train s'appuie sur ce principe.

Quand un processus p démarre, il commence par se déclarer participant au protocole (ligne 3.1). Cela permet d'éviter que deux processus qui démarreraient simultanément ne se considèrent chacun comme étant seuls sur le circuit de train : il y aurait alors simultanément deux circuits de train (cf. figure 4.4).

FIGURE 4.4 – Diagramme de séquence expliquant ce qui pourrait se passer si un processus qui démarre ne se déclarait pas participant au protocole



Puis, p recherche son futur successeur sur le circuit de train (ligne 3.2). S'il n'en trouve pas, p considère qu'il est seul sur le circuit de train. De ce fait, il uot-livre à la couche applicative un message contenant deux informations (lignes 3.4 et 3.5) : 1) il fait partie des participants au protocole d'uot-diffusion ; 2) il est le seul membre du circuit de train. Puis, p passe dans l'état `DansCircuitSeul` (ligne 3.7).

Étudions maintenant le cas où p n'est pas seul sur le circuit de train et trouve donc son futur successeur p_s. Il lui envoie le message `INSERT` (ligne 3.12). Deux cas peuvent se présenter :

1. p_s est membre confirmé du circuit de train, c'est-à-dire qu'il est dans l'état `DansCircuitSeul` ou bien dans l'état `DansCircuitÀPlusieurs`. p_s accepte la demande de p en renvoyant le message `ACK_INSERT` (ligne 5.6 ou ligne 6.14) avec en paramètre le prédécesseur p_p actuel de p_s. p_p deviendra le prédécesseur de p à la fin de l'insertion de p.

   Notez que :

   – si p_s est actuellement dans l'état `DansCircuitSeul`, il renvoie sa propre identité (ligne 5.6) : p_s sera non seulement le futur successeur de p, mais également son prédécesseur ;

   – si p_s est actuellement dans l'état `DansCircuitÀPlusieurs`, il ferme sa connexion avec p_p (ligne 6.13). Ce dernier mémorise qu'il n'a plus de successeur (ligne 6.43). Ainsi, si p_p reçoit un train récent avant que son nouveau successeur se soit connecté à lui, p_p n'essaye pas d'envoyer à p_s le train qu'il va préparer. En effet, cet envoi se ferait sur une connexion dont p_p sait qu'elle a été fermée.

   p se connecte ensuite à p_p et lui envoie `NOUVEAU_SUCC` (ligne 3.26).

2. p_s n'est pas un membre confirmé du circuit de train, c'est-à-dire qu'il est dans l'état `HorsCircuit`. Dans ce cas, p_s n'a aucune garantie qu'en cas de départ d'un processus du circuit de train, les processus actuellement membres confirmés de ce circuit sauront se reconnecter à p_s. De ce fait, il considère qu'il ne peut pas gérer la demande d'insertion de p dans de bonnes conditions et renvoie le message `NAK_INSERT` (lignes 3.19 et 3.33). p passe alors dans l'état `AttenteAvantNouvelleTentativeInsertion` : il y attend pendant une durée aléatoire dont la borne maximum double à chaque fois que p a besoin de faire une nouvelle attente pour réussir son insertion.

   Cette attente (inspirée de l'attente en cas de collision sur un réseau *ALOHA* [Robert, 2000]) avant une nouvelle tentative d'insertion sert notamment dans le cas de la tentative d'insertion simultanée de deux processus dans un circuit vide (cf. figures 4.5 et 4.6) [11].

Une fois que p a envoyé `NOUVEAU_SUCC` à p_p, il attend de recevoir la confirmation de son appartenance au circuit de train (lignes 3.27–41). En effet, quand p sera sûr que tous les processus actuellement membres confirmés du circuit de train sauront que p est dans le circuit, alors en cas de départ du successeur de p, p sera certain qu'un processus du circuit se reconnectera à lui et qu'il recevra *in fine* le train le plus récent circulant sur le circuit. Cette confirmation d'appartenance de p se fait de la manière suivante : lorsque le successeur p_s de p reçoit un train récent de p, il ajoute l'adresse de p dans `dte.circuit` et envoie `dte` à son propre successeur. De ce fait, quand p reçoit un train `tr` et qu'il trouve sa propre adresse dans `tr.circuit`, il a la garantie que tous les processus à la suite de p_s dans le circuit de train, donc au moins tous les processus membres confirmés du circuit de train, ont vu passer ce train : ils sont au courant que p fait désormais partie du circuit de train. De ce fait, p peut se considérer comme membre du circuit de train : p uot-diffuse,

11. En section 5, nous montrons que cette attente est finie.

FIGURE 4.5 – Diagramme de séquence de l'insertion simultanée de deux processus dans un circuit vide (1/2)

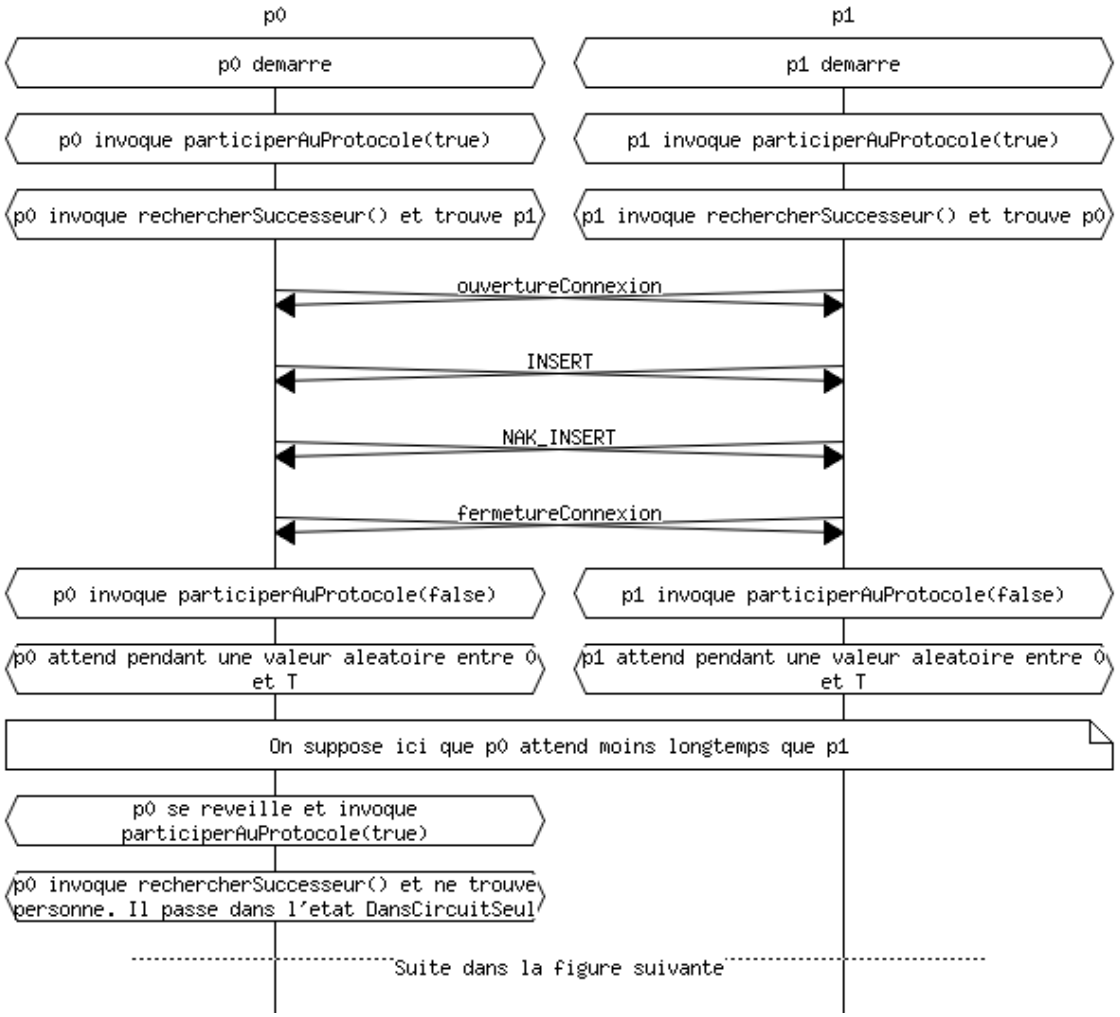
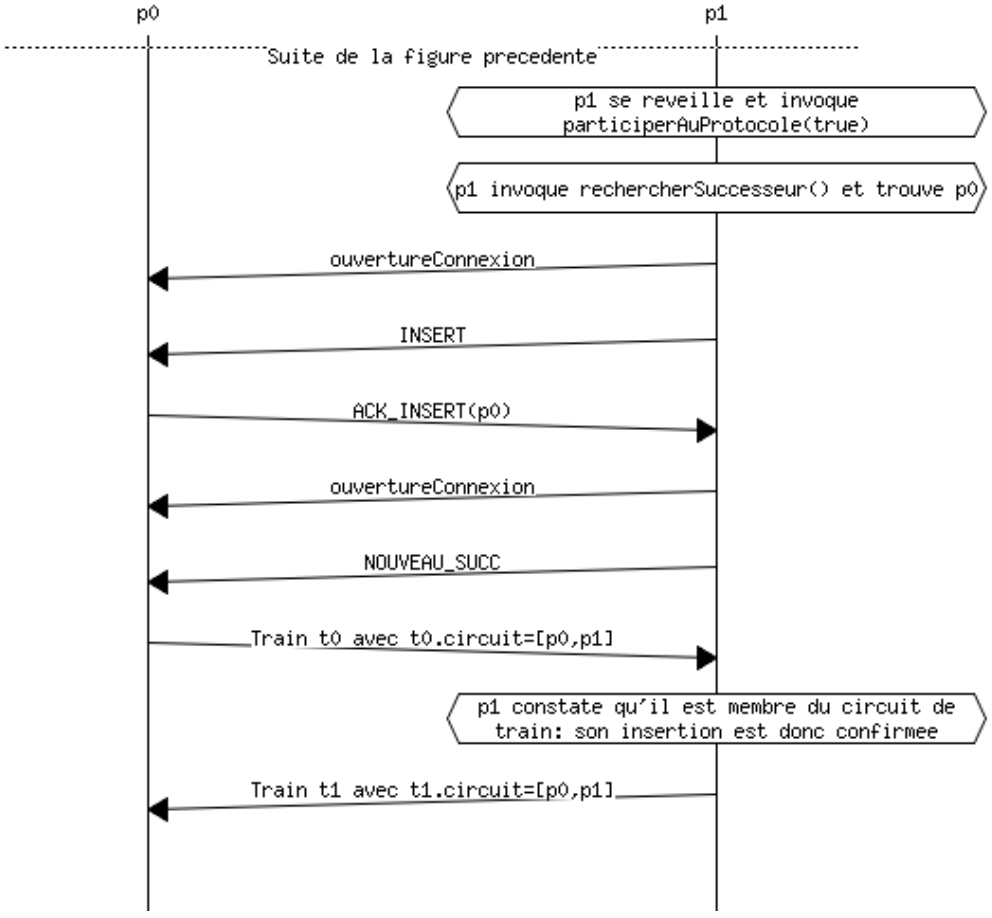


FIGURE 4.6 – Diagramme de séquence de l'insertion simultanée de deux processus dans un circuit vide (2/2)



à l'intention de la couche applicative, l'information de sa participation au protocole d'uot-diffusion accompagnée de la liste des membres du circuit de train (ligne 3.42). Puis, p bascule dans l'état `DansCircuitÀPlusieurs`.

En attendant cette confirmation, p laisse passer les trains qu'il reçoit sans y toucher (ligne 3.38). En particulier, il ne modifie pas l'identifiant du train reçu (cf. test à la ligne 3.35). Par ailleurs, pendant cette phase d'attente :

- p répond par un `NAK_INSERT` à tout message `INSERT` ;
- si p perd sa connexion avec son prédécesseur ou son successeur, il considère qu'il y a de fortes chances que sa demande d'insertion ne soit jamais confirmée : il passe dans l'état `AttenteAvantNouvelleTentativeInsertion` ;
- si p (de successeur p_s) reçoit un message `NOUVEAU_SUCC` de $p_{s'}$, p passe dans l'état `AttenteAvantNouvelleTentativeInsertion`. Nous procédons ainsi pour simplifier le traitement du cas de fermeture de connexion (évoqué dans l'alinéa précédent) [12].

Quand p reçoit le train qui lui confirme son appartenance au circuit de train, il incrémente l'identifiant de ce train (cf. test à la ligne 3.35 et l'incrémentation qui suit). Cette incrémentation sert surtout quand p rejoint un circuit où il n'y a qu'un seul processus p_0. En effet, p renvoie ainsi un train dont l'identifiant est strictement supérieur à $\texttt{dte}_{p_0}\texttt{.st.lc}$. p_0 considère donc que ce train reçu de p est récent et le traite donc [13].

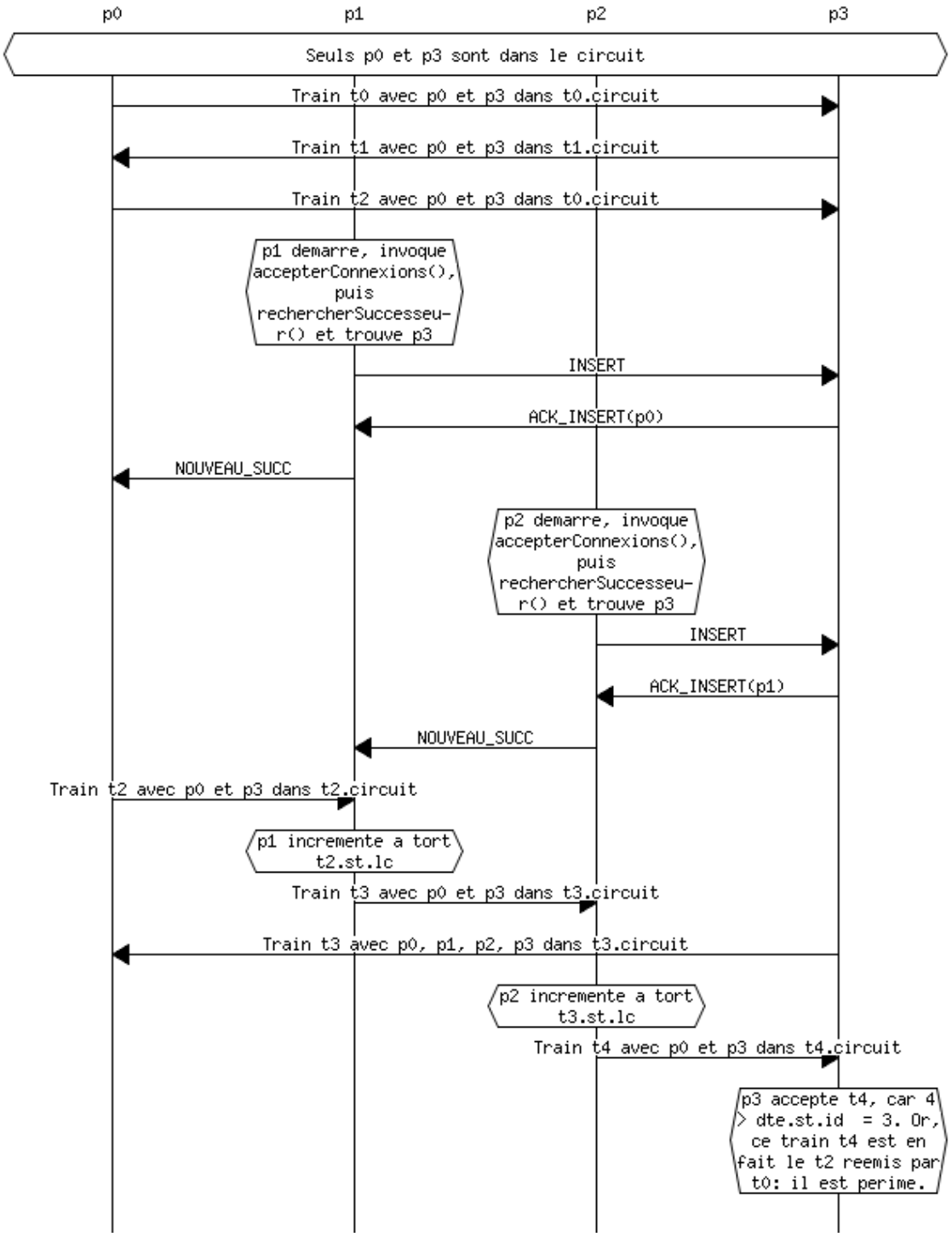
Le diagramme de séquence de la figure 4.7 illustre pourquoi un processus p ne doit surtout pas systématiquement incrémenter l'identifiant du train qu'il reçoit pendant qu'il attend la confirmation de son appartenance au circuit : cela pourrait incrémenter l'identifiant d'un train périmé. De ce fait, le processus détenteur du train le plus récent serait amené à considérer comme récent ce train périmé ; des wagons pourraient être uot-livrés deux fois : la propriété d'intégrité deviendrait caduque. Si p vérifie d'abord qu'il appartient au circuit de train reçu, il a la garantie que le train reçu est le plus récent. En effet, puisque p appartient au circuit de train reçu, ce train est la conséquence du train envoyé par le successeur de p après avoir reçu la demande d'insertion de p : il ne peut pas y avoir de train plus récent. p peut donc incrémenter l'identifiant de ce train sans aucune crainte.

La ligne 6.10 correspond à la réaction d'un processus quand il constate qu'il ne fait plus partie du circuit de train. En effet, comme l'illustre la figure 4.8, il peut arriver qu'un processus correct soit considéré comme défaillant par les autres processus et donc retiré de la liste des processus participant au protocole. Cela est dû au fait que, comme mentionné

12. Nous aurions pu aussi décider que p se contente de mémoriser l'identité de son nouveau successeur $p_{s'}$. En effet, ce message `NOUVEAU_SUCC` signifie que p_s a informé un autre processus $p_{s'}$ que p était le prédécesseur de p_s. Cela a pu advenir parce que $p_{s'}$ est le successeur actuel de p_s, que p_s a quitté le circuit de train et que $p_{s'}$, au courant que p est le nouveau prédécesseur de p_s, se connecte à p. Ce peut être aussi parce que $p_{s'}$ est un processus qui cherche à s'insérer au niveau de p_s, que p_s lui a indiqué que p était son futur prédécesseur et que donc $p_{s'}$ se connecte à p. Toutefois, pour que p puisse recevoir ce `NOUVEAU_SUCC` de $p_{s'}$, il faut que p_s ait fermé sa connexion avec p, puis envoyé `ACK_INSERT(p)` à $p_{s'}$. Donc, l'information de fermeture de connexion avec p_s va suivre la réception de ce message `NOUVEAU_SUCC` sachant que, dans la grande majorité des cas, il va le précéder. Ainsi, l'automate passera dans l'état `AttenteAvantNouvelle-TentativeInsertion`, à moins d'imaginer un traitement spécifique pour ce cas particulier. Nous avons choisi la voie de la simplicité en basculant dans l'état `AttenteAvantNouvelleTentativeInsertion` dès la réception de `NOUVEAU_SUCC`.

13. Si p n'avait pas incrémenté l'identifiant de train, p_0 aurait considéré que le train était périmé : le train n'aurait plus circulé entre p et p_0.

FIGURE 4.7 – Diagramme de séquence montrant pourquoi un processus non confirmé ne doit pas incrémenter l'identifiant du train (les ouvertures de connexion sont omises pour simplifier l'exemple)



au chapitre 4.1, nous avons ajouté à TCP un mécanisme de battement de cœur destiné à considérer défaillants les processus devenus trop lents. Ce processus (p_1 dans le cas de la figure 4.8) gêne alors la circulation du train en cherchant à se reconnecter systématiquement à son prédécesseur (p_0 dans le cas de la figure). En effet, si nous reprenons le diagramme de séquence de la figure 4.7 et que nous supposons que p_1 ne réagit pas en découvrant qu'il ne fait plus partie du circuit de train, alors :

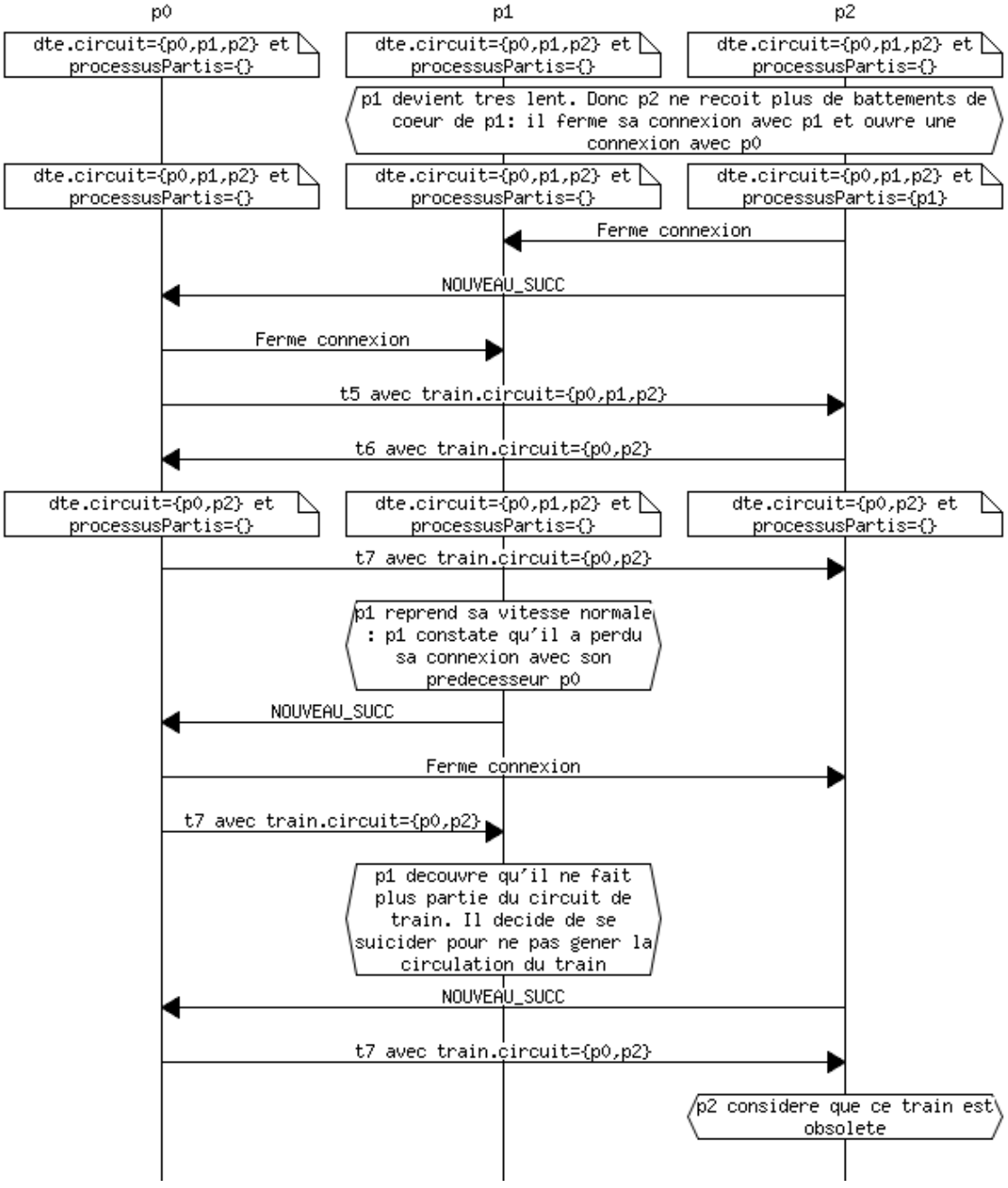
1. p_1 reprend sa vitesse normale ;
2. p_1 constate qu'il a perdu sa connexion avec son prédécesseur p_0. p_1 envoie `NOUVEAU_SUCC` à p_0. p_0 ferme sa connexion avec son successeur actuel p_2 (lignes 6.17 et 6.18). p_0 envoie `dte` à p_1 (ligne 6.20). Notez que, si jamais ce train est récent pour p_1, p_1 n'a personne à qui envoyer le train résultant puisque p_1 n'a plus de successeur ;
3. p_2 constate qu'il a perdu sa connexion avec son prédécesseur p_0. p_2 envoie `NOUVEAU_SUCC` à p_0. p_0 ferme sa connexion avec son successeur actuel p_1. p_0 envoie `dte` à p_2. Si le train est récent, p_2 le traite et envoie le train résultant à p_0 ;
4. p_1 procède comme au point 2 ;
5. p_2 procède comme au point 3 ;
6. etc.

Pour éviter ce problème, la réaction la plus simple à mettre en œuvre est que le processus qui constate qu'il n'est plus dans le circuit se suicide : il arrête ainsi de perturber son ancien prédécesseur avec des reconnexions intempestives. Toutefois, d'un point de vue applicatif, il se peut que ce soit ce processus qui détienne les données les plus à jour pour l'application. C'est pourquoi nous avons décidé que, si notre intergiciel protocolaire détecte ce problème, il prévient la couche applicative qui est sensée avoir mis en place un mécanisme dédié. Ce mécanisme lui permet de décider s'il faut effectivement tuer ce processus ou bien tuer tous les autres processus qui sont actuellement sur le circuit de train. Si la couche applicative n'a mis en place aucun mécanisme applicatif, l'intergiciel protocolaire adopte un mécanisme par défaut : le processus se suicide.

Ce comportement permet de gérer de manière satisfaisante les partitionnements réseau de notre système. En effet, en section (cf. section 4.1.1), nous avons vu que soit toutes les machines sont chacune dans un sous-réseau (le *switch* réseau est tombé en panne), soit plusieurs sont, chacune seule, dans un sous-réseau, et les autres machines sont connectées à un seul et même sous-réseau (le câble entre chaque station seule et le *switch* a été soit débranché, soit coupé). Il n'y a donc pas besoin d'une gestion de partitionnement réseau plus sophistiquée. Il est toutefois possible de l'améliorer en ajoutant un *thread* supplémentaire au sein de chaque processus p. Ce *thread* a pour objectif de rechercher des processus qui feraient partie d'un circuit de train dont p ne ferait pas partie : en cas de recherche fructueuse, ce *thread* déclenche le suicide de p ou bien signale le problème à la couche applicative pour permettre à cette dernière d'avoir la réaction appropriée.

Cette section a présenté comment les algorithmes gèrent le circuit de train. Elle conclut la présentation des algorithmes de la version monotrain du protocole, présentation qui a permis de mieux comprendre les principes de ces algorithmes. Aussi, nous pouvons désormais aborder les algorithmes de la version multitrain.

FIGURE 4.8 – Diagramme de séquence montrant le suicide d'un processus qui découvre qu'il ne fait plus partie du circuit de train



4.4 Version multitrain du protocole

Cette section généralise les algorithmes présentés en section 4.3 : plusieurs trains circulent en parallèle sur le circuit de train. Ainsi, nous augmentons le débit d'information transportée sur le réseau. La version multitrain s'appuie sur toutes les notions vues avec la version monotrain. La section 4.4.1 présente cette version de manière informelle. Puis la section 4.4.2 décrit les changements par rapport aux données, messages, fonctions et procédures de version monotrain. La section 4.4.3 décrit les changements par rapport aux algorithmes de la section 4.3.5. Enfin, la section 4.4.4 complète les sections 4.3.6 et 4.3.7 en expliquant les spécificités des algorithmes de la version multitrain par rapport à la version monotrain.

4.4.1 Présentation informelle de la version multitrain

Cette section présente de manière informelle la version multitrain. Notez que nous ne cherchons pas à être exhaustif dans cette section (notamment concernant les cas d'erreur). Nous souhaitons seulement faciliter la compréhension des algorithmes présentés à la section 4.4.3.

Une première idée pour avoir plusieurs trains circulant en parallèle serait d'avoir plusieurs circuits de train c, c'..., chacun hébergeant un train qui circule entre les processus p_0, $p1$... Cette idée n'est pas réaliste. En effet, elle n'offre aucune garantie de synchronisation entre les différents circuits. Ainsi, si le train t (qui circule sur c) est traité par p_0 avant le train t' (qui circule sur c'), il se peut très bien que t' arrive ensuite avant t au niveau de p_1.

La solution que nous avons adoptée est de faire circuler les trains t, t'... sur le même circuit de train (que nous nommons donc circuit *des trains*). En effet, vu que les canaux de communication de notre système garantissent l'ordre FIFO (cf. section 4.1), ils garantissent que t arrivera toujours avant t' au niveau des différents processus.

Les trains circulant sur le même circuit, leur estampille est désormais une structure constituée de trois nombres :

1. l'*identifiant* de ce train : ce nombre sert à différencier les trains qui circulent en parallèle sur le circuit ;
2. l'*horloge logique* de ce train, qui a la même rôle que l'horloge logique de la version monotrain (cf. section 4.3) ;
3. le *numéro de tour* dont fait partie ce train : cette valeur indique le nombre de rotations qu'a fait le train sur le circuit.

C'est pourquoi dans cette section, nous notons les trains $t_{\{i,j,k\}}$: $\{i, j, k\}$ désigne l'estampille du train ; i est son identifiant, j son horloge logique, et k son numéro de tour.

En multitrain, l'horloge logique ne suffit pas à distinguer les différents trains. Par exemple, supposons qu'un processus p reçoive t_2 de son prédécesseur p_p. Alors p envoie t_3 à son successeur p_s. Si jamais p reçoit ensuite t_4, ce t_4 est-il un train qui est derrière t_2 sur le circuit ? Ou bien est-ce le t_4 que p_s a construit à partir du t_3 reçu de p et que p_s envoie à p parce que tous les processus entre p_s et p se sont arrêtés ? p est incapable de répondre. Une solution à cette incapacité serait de mettre un grand écart entre les horloges logiques

des différents trains. Dans le cas où deux trains circulent en parallèle, il y aurait alors, par exemple, le train t_2 suivi du train t_{1042} sur le circuit. Cette solution résout le problème précédemment identifié : t_{1042} et t_4 sont distincts. Mais, apparaît un nouveau problème. S'il y a seulement deux processus sur le circuit, quand p reçoit t_2, il envoie t_3 à p_s. Ensuite, quand p_2 reçoit t_{1042}, il envoie t_{1043} à p_s. Or, comme il n'y a que deux processus sur le circuit, p_s envoie t_4 à p. Comme $4 < 1043$, p considère que t_4 est un train obsolète alors que ce n'est pas le cas. Pour résoudre ces deux problèmes, nous introduisons l'identifiant de train et le couplons avec l'horloge logique. Aussi, pour déterminer s'il doit traiter un train t reçu, un processus p regarde d'abord si l'identifiant de t correspond à l'identifiant id que p attendait. Puis, p compare l'horloge logique de t avec l'horloge logique du dernier train qu'il a envoyé avec pour identifiant id. Il détermine ainsi si t est récent ou obsolète.

La figure 4.9 illustre la circulation de plusieurs trains en parallèle sur le circuit.

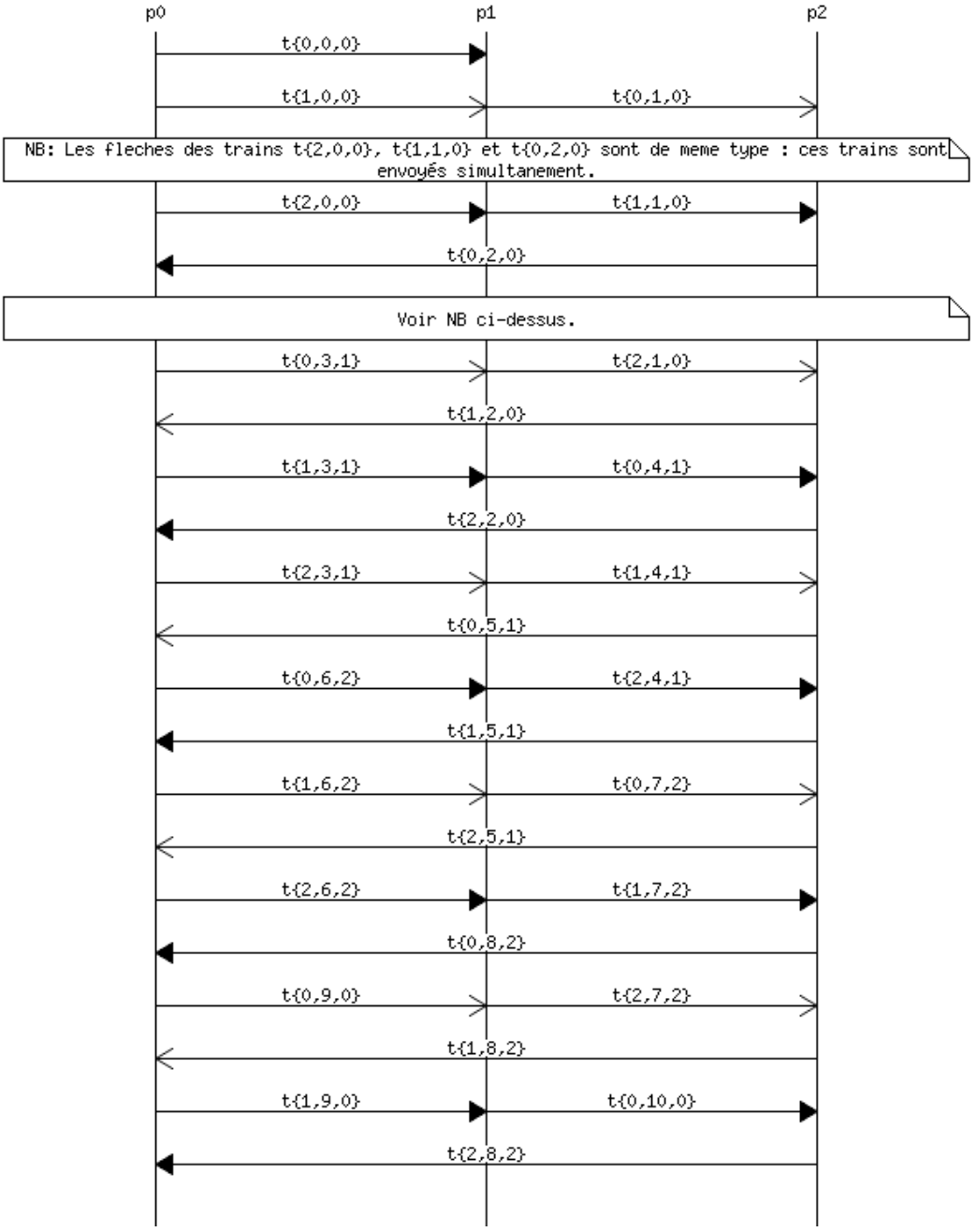
En cas de défaillance d'un processus p, son successeur p_s se connecte au prédécesseur p_p de p. p_p envoie à p tous les derniers trains qu'il avait envoyés à p. Par exemple, dans la figure 4.9, supposons que le processus p_1 a défailli juste après avoir envoyé le train $t_{2,4,1}$ et reçu le train $t_{\{0,6,2\}}$: p_2 se connecte à p_0 qui lui renvoie les trois [14] derniers trains envoyés à p_1, c'est-à-dire $t_{\{1,3,1\}}$, $t_{\{2,3,1\}}$ et $t_{\{0,6,2\}}$. p_2 s'attend à recevoir un train d'identifiant 0 (puisque le dernier train que p_2 a envoyé à p_0 est $t_{\{2,4,1\}}$). Par conséquent, p_2 ignore $t_{1,3,1}$ (respectivement $t_{2,3,1}$) puisqu'il a pour identifiant 1 (respectivement 2). Quant à $t_{\{0,6,2\}}$, p_2 s'intéresse maintenant à son horloge logique, c'est-à-dire 6. p_2 la compare avec l'horloge logique de $t_{\{0,5,1\}}$ (qui est le dernier train d'identifiant 0 que p_2 a envoyé), c'est-à-dire 5. $6 > 5$: $t_{\{0,6,2\}}$ est plus récent que $t_{\{0,5,1\}}$. p_2 traite $t_{\{0,6,2\}}$.

Intéressons-nous maintenant au numéro de tour stocké dans l'estampille du train. Cet attribut ne sert pas à distinguer les trains, mais à garantir l'ordre de livraison des wagons dans `wagonAUOTLivrer`. Pour ce faire, chaque wagon contient l'identifiant de son émetteur et le numéro de tour auquel ce wagon appartient, c'est-à-dire la valeur de l'attribut numéro de tour du train auquel le wagon est ajouté. Notons $w_{i\theta\alpha}$ le wagon α [15] envoyé par le processus p_i pendant le numéro de tour θ. Montrons l'intérêt de la notion de numéro de tour en raisonnant par l'absurde. La figure 4.10 illustre ce qui se passerait si on n'utilisait pas la notion de numéro de tour et que, comme en version monotrain, on uot-livrait un wagon quand le train qui l'a transporté a fait un tour de circuit : *in fine*, p_0 (respectivement p_1) uot-livre $w_{0.a}$, puis $w_{0.b}$ (respectivement $w_{0.a}$, puis $w_{1.a}$), ce qui est incorrect. L'utilisation de l'attribut numéro de tour pallie ce problème. Quand un processus reçoit un train d'identifiant id et de numéro de tour θ, il compare θ au numéro de tour du dernier train de même identifiant qu'a envoyé p. Si les deux valeurs sont différentes, p laisse θ inchangé. En revanche, si les deux valeurs sont égales, cela signifie que le train a fait un tour : p incrémente θ. De ce fait, quand p reçoit un train d'identifiant id et de numéro de tour θ, p a la garantie qu'il a reçu tous les wagons de numéro de tour $\theta - 1$. De plus, p a la garantie que tous les wagons de numéro de tour $\theta - 2$ ont été reçus par tous les participants : p peut uot-livrer ces wagons. Par exemple, dans la figure 4.11, p_0 reçoit $t_{\{0,5,1\}}$. Ce train a pour numéro de tour 1, donc le même numéro de tour que le dernier train de même identifiant

14. Ce nombre est le nombre de trains qui circulent en parallèle sur le circuit des trains. Il est stocké dans la variable *ntr* définie en section 4.4.2.

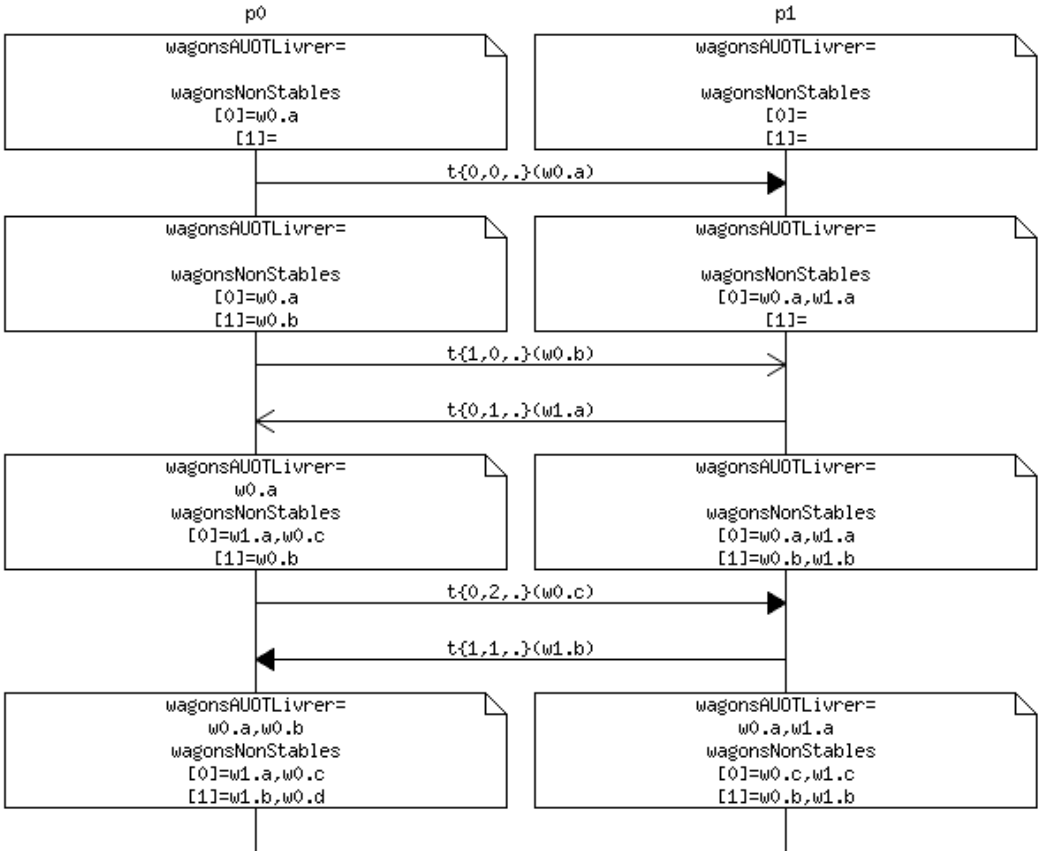
15. Dans cette notation, α est une lettre destinée à faciliter la distinction entre les différents wagons envoyés par le processus. α ne correspond à aucun attribut de l'estampille.

FIGURE 4.9 – Diagramme de séquence de circulation de plusieurs trains entre 3 processus (NB : un groupe de flèches de même type signifie que les messages sont envoyés simultanément)



que p_0 a envoyé, c'est-à-dire $t_{\{0,3,1\}}$. Aussi, p_0 incrémente le numéro de tour de ce train : il passe à $1 + 1 = 2$. Ce train a pour identifiant 0. Donc p_0 ajoute les wagons w_{11d} et w_{21d} de ce train ainsi que le wagon w_{02g} de `wagonAEnvoyer` (ce wagon a pour numéro de tour 2 puisque le train a pour numéro de tour 2) dans la liste `wagonsNonStables`$[id = 0]$. Cette dernière contient donc : $[w_{00a}, w_{10a}, w_{20a}, w_{01d}, w_{11d}, w_{21d}, w_{02g}]$. Les wagons ayant pour numéro de tour $2 - 2 = 0$ sont stables De ce fait, p_0 transfère w_{00a}, w_{10a} et w_{20a} de `wagonsNonStables`$[id = 0]$ à `wagonsAUOTLivrer`. En revanche, les wagons w_{01d}, w_{11d}, w_{21d} et w_{02g} n'ont pas 0 pour numéro de tour : ils restent dans `wagonsNonStables`$[id = 0]$. Notez que seuls les numéros de tour θ, $\theta - 1$ et $\theta - 2$ ont besoin d'être distingués : cela fait 3 valeurs de numéro de tour. Par conséquent, le numéro de tour a seulement besoin d'être une valeur dans l'intervalle $[\![0, 3[\![$. C'est pourquoi, dans la figure 4.9, quand p_0 reçoit $t_{\{1,8,2\}}$ de p_2, p_0 envoie $t_{\{1,9,0\}}$ à p_1.

FIGURE 4.10 – Diagramme de séquence montrant une anomalie d'ordre d'uot-livraisons quand le numéro de tour n'est pas utilisé (NB : les « . » représentent le numéro de tour inutilisé ; un groupe de flèches de même type signifie que les messages sont envoyés simultanément ; par souci de lisibilité, la figure ne représente pas les trains qui ont circulé avant l'envoi de $t_{\{0,0,.\}}$)



Cette section a présenté de manière informelle la version multitrain du protocole des

FIGURE 4.11 – Diagramme de séquence (1/2) illustrant le rôle du numéro de tour (NB : un groupe de flèches de même type signifie que les messages sont envoyés simultanément ; par souci de lisibilité, la figure ne représente pas les trains qui ont circulé avant l'envoi de $t_{\{0,0,0\}}$)

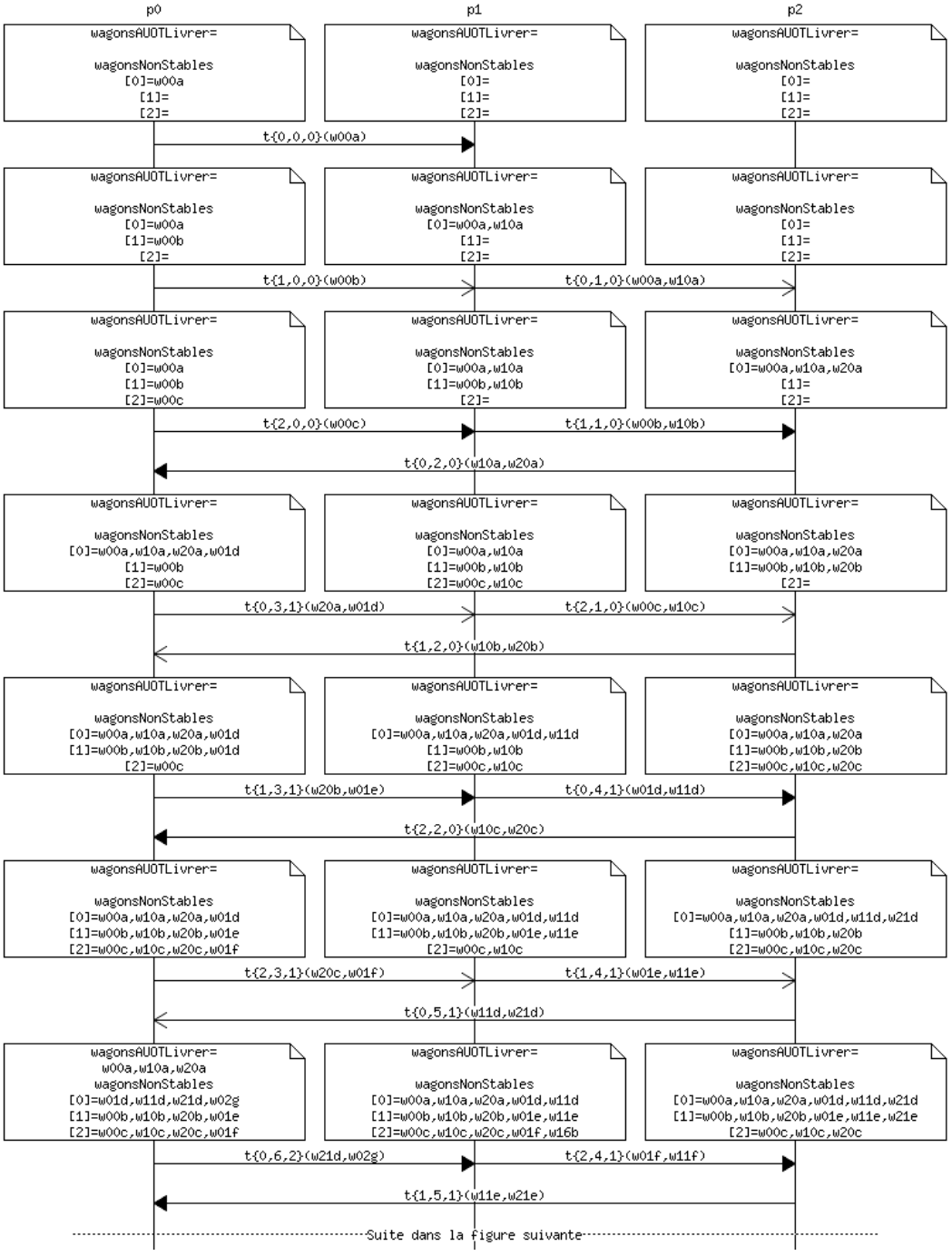
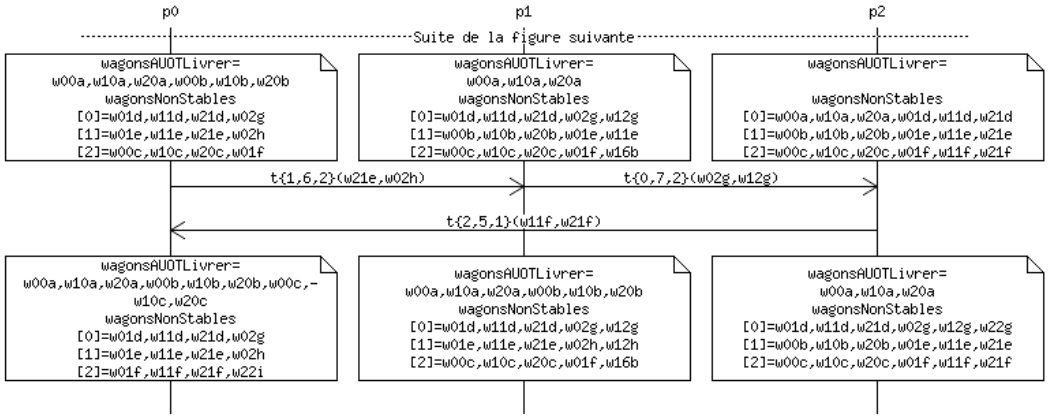


FIGURE 4.12 – Diagramme de séquence (2/2) illustrant le rôle du numéro de tour (NB : un groupe de flèches de même type signifie que les messages sont envoyés simultanément ; par souci de lisibilité, la figure ne représente pas les trains qui ont circulé avant l'envoi de $t_{\{0,0,0\}}$)



trains. Intéressons-nous maintenant aux conséquences du multitrain sur les données, les messages, les fonctions et procédures.

4.4.2 Données, messages, fonctions et procédures

La version multitrain utilise les mêmes messages qu'en version monotrain (cf. section 4.3.2). De même, les fonctions et procédures sont identiques (cf. section 4.3.4). En revanche, les données évoluent par rapport à la version monotrain (cf. section 4.3.2) :

1. `ntr` est une nouvelle variable. Elle contient le nombre de trains circulant en parallèle sur le circuit. Nous supposons que ce nombre est fixé au début de l'exécution du protocole et garde la même valeur tout au long de cette exécution. La section 6.1.6 propose une piste pour faire varier `ntr` pendant l'exécution du protocole ;
2. `NR` est une nouvelle constante. Égale à 3, elle caractérise le nombre de numéros de tour que nous avons besoin de distinguer. Quand un processus p reçoit un train de numéro de tour θ et qu'il doit l'incrémenter de un, p le fait modulo `NR` : $\theta \leftarrow (\theta + 1) \bmod \texttt{NR}$. Par ailleurs, pour déterminer les wagons stables, p le fait également modulo `NR` : $\theta \leftarrow (\theta - 2 + \texttt{NR}) \bmod \texttt{NR}$, le `+ NR` garantissant que la valeur obtenue est dans $[\![0, NR[\![$;
3. `die` est une nouvelle variable. Elle contient le numéro du dernier identifiant envoyé ;
4. `dte` joue le même rôle que quand elle a été définie en section 4.3.2, sauf qu'elle est désormais un tableau de `ntr` éléments ;
5. `wagonsNonStables` joue le même rôle que quand elle a été définie en section 4.3.2, sauf qu'elle est désormais un tableau (à deux dimensions) de `ntr` × `NR` éléments.

Un train est désormais structuré de la manière suivante :
– un entête contenant

  - `st`, l'estampille de train qui était un entier en section 4.3.2 et est désormais une structure contenant 3 champs :
    1. `id`, l'identifiant du train codé sous la forme d'un entier (de taille suffisante pour pouvoir contenir toutes les valeurs entre 0 et `ntr`-1) ; ce champ correspond à i dans la notation $t_{\{i.j,k\}}$;
    2. `lc` (pour « Logicial Clock »), l'horloge logique de ce train. Elle est incrémentée à chaque passage par un processus. C'est un entier dont nous supposons pour l'instant qu'il ne peut pas être victime d'un dépassement de capacité [16]. Ce champ correspond à j dans la notation $t_{\{i.j,k\}}$;
    3. `numTour`, le numéro de tour qu'effectue ce train. C'est un entier compris entre 0 et `NR`-1 ; ce champ correspond à k dans la notation $t_{\{i.j,k\}}$;
  - `circuit`, le circuit des trains (donnée inchangée par rapport à la section 4.3.2) ;
- `wagons`, la liste ordonnée des wagons transportés par ce train (donnée inchangée par rapport à la section 4.3.2).

Chaque wagon contient désormais un champ supplémentaire : `numTour`. Ce champ contient le numéro de tour du train auquel ce wagon est ajouté.

Ces données sont initialisées par l'algorithme 9.

Cette section a expliqué qu'il n'y avait aucun changement par rapport aux messages, fonctions et procédures de la version monotrain. Puis, elle a décrit les conséquences du multitrain sur les données. Intéressons-nous maintenant aux évolutions sur les algorithmes.

4.4.3 Algorithmes

Par rapport à la section 4.3.5 :

- le diagramme de machine à états évolue (cf. figure 4.13 page 83) ;
- l'algorithme de la procédure `uotDiffusion` est inchangé (cf. algorithme 2 page 59) ;
- l'algorithme de l'état `HorsCircuit` évolue (cf. algorithme 10 page 84) ;
- l'algorithme de l'état `AttenteAvantNouvelleTentativeInsertion` est inchangé (cf. algorithme 4 page 62) ;
- l'algorithme de l'état `DansCircuitSeul` évolue (cf. algorithme 11 page 85) ;
- l'algorithme de l'état `DansCircuitÀPlusieurs` évolue (cf. algorithme 12 page 86) ;
- l'algorithme de la procédure `traiterTrainRécent` évolue (cf. algorithme 13 page 87) ;
- l'algorithme de la fonction `estCeQueTrainRécent` évolue (cf. algorithme 14 page 87).

Après avoir présenté la version multitrain des algorithmes, nous détaillons leurs spécificités liées à l'aspect multitrain.

4.4.4 Principes algorithmiques

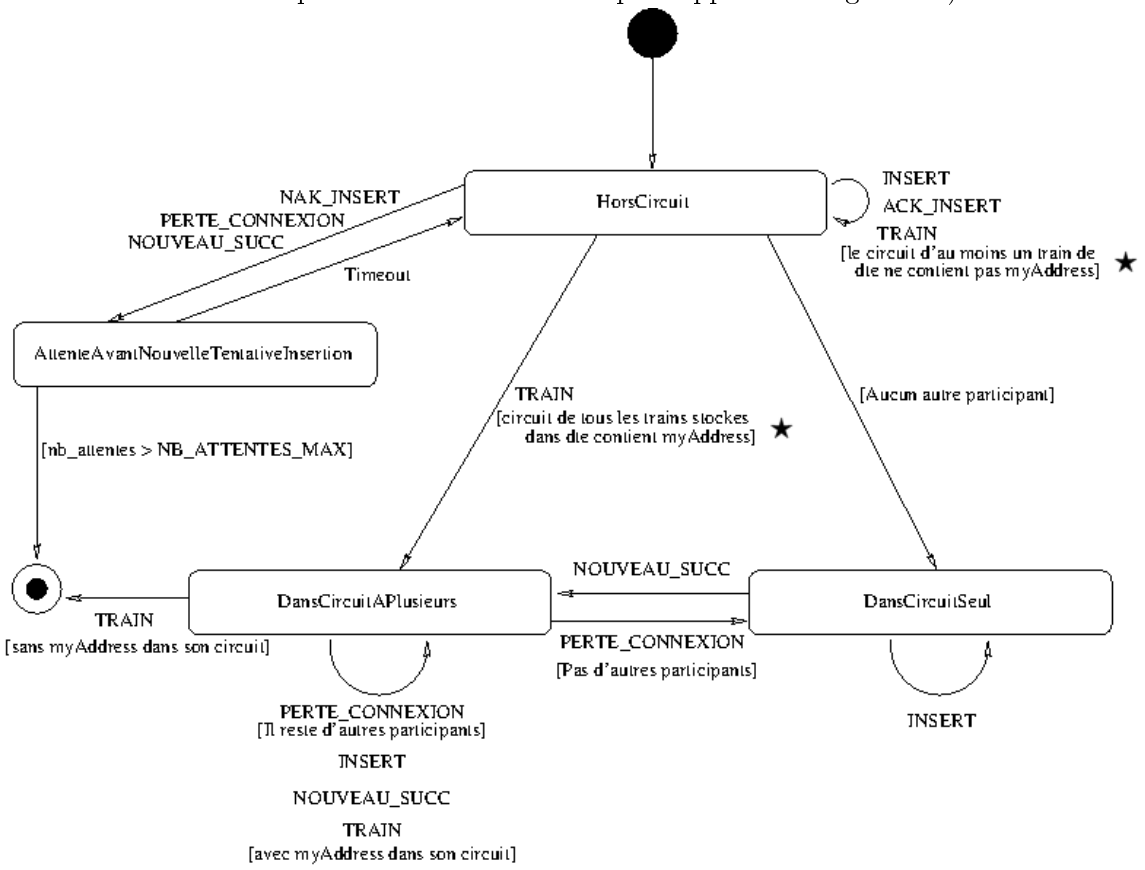
Cette section complète les sections 4.3.6 et 4.3.7 en expliquant les spécificités des algorithmes de la version multitrain par rapport à la version monotrain. Cette section est volontairement brève, car l'adéquation des algorithmes multitrains aux propriétés d'uot-diffusion sera démontrée au chapitre 5.

16. Voir la section 6.1.1.3 pour la gestion du dépassement de capacité.

Algorithme 9 Initialisation des données de l'automate à états du protocole des trains (version multitrain ; le caractère « ★ » indique les lignes ajoutées ou modifiées par rapport à l'algorithme 1)

```
 1: nbAttentes ← 0
 2: ntr ← lecture d'une variable d'environnement// ★
 3: die ← ntr - 1// ★
 4: for all id ∈ [[0, ntr[[ do // ★
 5:   dte[id].st.id ← id// ★
 6:   dte[id].st.lc ← 0// ★
 7:   dte[id].st.numTour ← 0// ★
 8:   dte[id].circuit ← [ ] // ★
 9:   dte[id].wagons ← [ ] // ★
10: end for// ★
11: for all id ∈ [[0, ntr[[, numTour ∈ [[0, NR[[ do // ★
12:   wagonsNonStables[id][numTour] ← [ ] // ★
13: end for// ★
14: processusArrivés ← [ ]
15: processusPartis ← [ ]
16: wagonAEnvoyer.emetteur ← myAddress
17: wagonAEnvoyer.numTour ← 0 // ★
18: wagonAEnvoyer.msgs ← [ ]
19: wagonsAUOTLivrer ← [ ]
20: prédécesseur ← ⊥
21: successeur ← ⊥
22: nextstate HorsCircuit
```

FIGURE 4.13 – Diagramme de machine à états du protocole des trains (version multitrain ; ; le caractère « ★ » indique les éléments modifiés par rapport à la figure 4.3)



Algorithme 10 État `HorsCircuit` de l'automate à états du protocole des trains (version multitrain ; le caractère « ★ » indique les lignes ajoutées ou modifiées par rapport à l'algorithme 3)

```
 1: participerAuProtocole(true)
 2: successeur ← rechercherSuccesseur()
 3: if successeur == myAddress then
 4:   append(wagonAEnvoyer.msgs,msgInfoArrivée(myAddress,[myAddress]))
 5:   append(wagonsAUOTLivrer,wagonAEnvoyer)
 6:   wagonAEnvoyer.msgs ← [ ]
 7:   nextstate DansCircuitSeul
 8: end if
 9: if ouvrirConnexion(successeur) == ÉCHEC then
10:   nextstate AttenteAvantNouvelleTentativeInsertion
11: end if
12: envoyer(INSERT,successeur)
13: repeat // Attente ACK_INSERT
14:   msg ← recevoirMsg()
15:   switch (msg.type)
16:   case NAK_INSERT or PERTE_CONNEXION:
17:     nextstate AttenteAvantNouvelleTentativeInsertion
18:   case INSERT:
19:     envoyer(NAK_INSERT(),msg.émetteur)
20:   end switch
21: until msg.type == ACK_INSERT
22: prédécesseur ← msg.ACK_INSERT.préd
23: if ouvrirConnexion(prédécesseur) == ÉCHEC then
24:   nextstate AttenteAvantNouvelleTentativeInsertion
25: end if
26: envoyer(NOUVEAU_SUCC,prédécesseur)
27: repeat // Attente confirmation que processus dans le circuit
28:   msg ← recevoirMsg()
29:   switch (msg.type)
30:   case PERTE_CONNEXION or NOUVEAU_SUCC:
31:     nextstate AttenteAvantNouvelleTentativeInsertion
32:   case INSERT:
33:     envoyer(NAK_INSERT(),msg.émetteur)
34:   case TRAIN:
35:     if myAddress ∈ msg.TRAIN.circuit then
36:       msg.TRAIN.st.lc ← msg.TRAIN.st.lc +1
37:     end if
38:     envoyer(msg,successeur)
39:     dte[msg.TRAIN.st.id] ← msg // ★
40:   end switch
41: until msg.type == TRAIN and ∀id ∈ ⟦0,ntr⟦ , myAddress ∈ dte[id].circuit // ★
42: die ← msg.TRAIN.st.id // ★
43: append(wagonAEnvoyer.msgs,msgInfoArrivée(myAddress,dte[die].circuit))// ★
44: nextstate DansCircuitÀPlusieurs
```

Algorithme 11 État `DansCircuitSeul` de l'automate à états du protocole des trains (version multitrain ; le caractère « ★ » indique les lignes ajoutées ou modifiées par rapport à l'algorithme 5)

```
1: prédécesseur ← myAddress
2: successeur ← myAddress
3: repeat // Attente demande d'insertion de la part d'un autre processus
4:   msg ← recevoirMsg()
5: until msg.type == INSERT
6: envoyer(ACK_INSERT(myAddress),msg.émetteur)
7: prédécesseur ← msg.émetteur
8: repeat // Attente connexion établie avec le successeur
9:   msg ← recevoirMsg()
10:  switch (msg.type)
11:  case PERTE_CONNEXION:
12:    nextstate DansCircuitSeul
13:  case INSERT:
14:    sauvegardeJusquAuProchainNextstate(msg)
15:  end switch
16: until msg.type == NOUVEAU_SUCC
17: successeur ← msg.émetteur
18: // On met en circulation tous les trains en les envoyant vers le successeur
19: for i= 1 to ntr do // ★
20:   id←((die + i) mod ntr // ★
21:   dte[id].st.lc ← dte[id].st.lc +1 // ★
22:   dte[id].circuit ← [myAddress, prédécesseur] // ★
23:   dte[id].wagons ← [ ] // ★
24:   envoyer(dte[id],successeur) // ★
25: end for // ★
26: nextstate DansCircuitÀPlusieurs
```

Algorithme 12 État `DansCircuitÀPlusieurs` de l'automate à états du protocole des trains (version multitrain ; le caractère « ★ » indique les lignes ajoutées ou modifiées par rapport à l'algorithme 6)

```
 1: msg ← recevoirMsg()
 2: switch (msg.type)
 3: case TRAIN:
 4:    if myAddress ∈ msg.TRAIN.circuit then
 5:       if estCeQueTrainRécent(tr.st,dte,die,ntr) then // cf. algorithme 14 ★
 6:          traiterTrain(tr,dte,wagonsAEnvoyer,processusPerdus,processusArrivés)
                                                                  // cf. algorithme 13 ★
 7:          die ← tr.st.id // ★
 8:          envoyer(dte[die],successeur) // ★
 9:       end if
10:    else
11:       Signaler problème de partitionnement à la couche applicative : elle décidera des suites à donner
          (suicide par défaut)
12:    end if
13: case INSERT:
14:    fermerConnexion(prédécesseur)
15:    envoyer(ACK_INSERT(prédécesseur),msg.émetteur)
16:    prédécesseur ← msg.émetteur
17:    append(processusArrivés,prédécesseur)
18: case NOUVEAU_SUCC:
19:    fermerConnexion(successeur)
20:    successeur ← msg.émetteur
21:    for i= 1 to ntr do // ★
22:       envoyer(dte[(die + i) mod ntr],successeur) // ★
23:    end for // ★
24: case PERTE_CONNEXION:
25:    if connexion perdue avec prédécesseur then
26:       // On cherche à se connecter à un prédécesseur en commençant par le prédécesseur actuel
27:       while prédécesseur ! = myAddress do
28:          if ouvrirConnexion(prédécesseur) == SUCCÈS then
29:             envoyer(NOUVEAU_SUCC,prédécesseur)
30:             nextstate DansCircuitÀPlusieurs
31:          end if
32:          append(processusPartis,prédécesseur)
33:          prédécesseur ← prédécesseurDe(prédécesseur,dte[die].circuit) // ★
34:       end while
35:       extend(wagonAEnvoyer.msgs,msgsInfoDépart(processusPartis))
36:       processusPartis ← [ ]
37:       for aNumTour = 1 to NR do // ★
38:          for i = 1 to ntr do // ★
39:             id ← ( die + i ) mod ntr // ★
40:             variable locale: numTour ← ( dte[id].st.numTour + aNumTour ) mod NR // ★
41:             extend(wagonsAUOTLivrer,wagonsNonStables[id][numTour]) // ★
42:             wagonsNonStables[id][numTour] ← [ ] // ★
43:          end for// ★
44:       end for// ★
45:       append(wagonsAUOTLivrer,wagonAEnvoyer)
46:       wagonAEnvoyer.msgs ← [ ]
47:       nextstate DansCircuitSeul
48:    else // Connexion perdue avec successeur
49:       successeur ← ⊥
50:    end if
51: end switch
52: nextstate DansCircuitÀPlusieurs
```

Algorithme 13 Traitement de la réception d'un train récent par un processus membre du circuit des trains (version multitrain ; il n'y a pas de caractère « ★ » pour indiquer les différences par rapport à l'algorithme 7, car les lignes sont pratiquement toutes différentes)

```
1: variable locale: id ← tr.st.id
2: variable locale: numTour ← tr.st.numTour
3: if numTour == dte[id].st.numTour then
4:   numTour ← (numTour + 1) mod NR
5: end if
6: extend(wagonsAUOTLivrer,wagonsNonStables[id][(numTour - 2 + NR) mod NR])

7: wagonsNonStables[id][(numTour - 2 + NR) mod NR] ← [ ]
8: if id == 0 then
9:   dte[id].circuit ←
          majCircuit(tr.circuit,myAddress,processusArrivés,processusPartis)
10:  processusArrivés ← [ ]
11:  extend(wagonAEnvoyer.msgs,msgsInfoDépart(processusPartis))
12:  processusPartis ← [ ]
13: else
14:  dte[id].circuit ← dte[0].circuit
15: end if
16: dte[id].wagons ← [ ]
17: for all w ∈ tr.wagons tel que emetteur(w) ∈ dte[id].circuit \ ({myAddress}∪
    processusPartis) do
18:  append(wagonsNonStables[id][w.numTour],w)
19:  if emetteur(w) != successeur then
20:    append(dte[id].wagons,w)
21:  end if
22: end for
23: dte[id].numTour ← numTour
24: wagonAEnvoyer.numTour ← numTour
25: append(wagonsNonStables[id][numTour],wagonAEnvoyer)
26: append(dte[id].wagons,wagonAEnvoyer)
27: wagonAEnvoyer.msgs ← [ ]
28: dte[id].st.lc ← tr.st.lc + 1
```

Algorithme 14 Fonction `estCeQueTrainRécent(tr_st,dte,die,ntr)` sans gestion des dépassements de capacité (version multitrain ; il n'y a pas de caractère « ★ » pour indiquer les différences par rapport à l'algorithme 8, car les lignes sont toutes différentes)

```
1: variable locale: idAttendu ← (die+1) mod ntr
2: if tr_st.id == idAttendu then
3:   return (tr_st.lc > dte[idAttendu].st.lc)
4: else
5:   return false
6: end if
```

`dte` est désormais un tableau. De ce fait :

1. dans l'état `HorsCircuit`, quand un processus découvre qu'il n'est pas seul sur le circuit des trains, il attend d'avoir reçu tous les trains circulant sur le circuit avant de passer dans l'état `DansCircuitÀPlusieurs` (ligne 10.39) ;
2. dans l'état `DansCircuitSeul`, quand un processus reçoit `NOUVEAU_SUCC`, il doit renvoyer tous les trains de `dte` en vue de les remettre en circulation (lignes 11.19–25) ;
3. dans l'état `DansCircuitÀPlusieurs`,
   - quand un processus reçoit un message `TRAIN` :
     - il vérifie que l'identifiant de ce train est l'identifiant qu'il attendait (ligne 12.5) ;
     - dans l'algorithme 13, le processus travaille sur la case de `dte` correspondant à l'identifiant du train reçu, c'est-à-dire la case `tr.st.lc` ;
     - une fois que le processus a construit le nouveau train à envoyer, il mémorise l'identifiant de ce train (ligne 12.7) avant de l'envoyer ;
   - quand un processus reçoit un message `NOUVEAU_SUCC`, comme il ne sait pas combien de messages ont été perdus au niveau de son précédent (dans le temps) successeur, il renvoie tous les trains stockés dans `dte` en commençant par le plus ancien (cf. boucle `for` des lignes 12.21–23).

Par ailleurs, plusieurs trains circulent désormais en parallèle sur le circuit des trains. La première conséquence est que, pour signaler les processus partis et les processus arrivés, nous avons décidé que les calculs de nouveaux circuits ne se font qu'au moment du passage du train d'identifiant 0 (lignes 13.8–12), les circuits des autres trains n'étant que des copies du circuit de train d'identifiant 0 (ligne 13.8, 13.13 et 13.14). Ainsi, quand un processus p modifie le circuit d'un train d'identifiant 0, il doit attendre que ce train ait fait le tour du circuit avant de faire un quelconque autre changement : p est certain que tous les processus ont pris en compte les changements qu'il a signalés précédemment. Toutefois, p ne peut pas se permettre d'attendre d'avoir reçu le train d'identifiant 0 pour purger les wagons obsolètes des autres trains. C'est pourquoi, à la ligne 13.17, p ne stocke pas dans le train à envoyer les wagons de `wagonsNonStables` dont l'émetteur se trouve dans `processusPartis`.

La deuxième conséquence est que, pour vérifier qu'un train reçu par un processus n'est pas obsolète, la fonction `estCeQueTrainRécent` (cf. algorithme 14) commence par regarder si l'identifiant de ce train correspond à l'identifiant que le processus attend. Si c'est le cas, la fonction renvoie le résultat de la comparaison des horloges logiques (comme pour l'algorithme 8). Sinon la fonction renvoie systématiquement la valeur `false`.

Cette section a présenté les spécificités de la version multitrain des algorithmes. Elle conclut la présentation de cette version multitrain.

4.5 Conclusion

Ce chapitre a présenté les algorithmes du protocole des trains. Les processus sont répartis sur un anneau virtuel : le circuit des trains. Des jetons, dénommés trains, circulent en parallèle sur ce circuit. Ils transportent les messages à uot-diffuser. Ils sont munis d'un numéro d'identifiant (qui permet de distinguer les différents trains) et d'une horloge logique (qui permet de retrouver les trains les plus récents en cas de perte de processus).

La gestion de l'anneau virtuel est intégrée aux algorithmes. Ainsi, le temps de récupération d'une défaillance d'un processus p n'est constitué que : 1) du temps de détection de la défaillance, 2) plus le temps de connexion du successeur p_s de p avec le prédécesseur p_p de p, 3) plus le temps de réémission par p_p à p_s des derniers trains envoyés par p_p à p, 4) plus le temps d'analyse par p_s du bien-fondé de ces réémissions (en comparant les horloges logiques reçues des trains de p_p avec les horloges logiques des trains dernièrement envoyées par p_s). En outre, la construction de l'anneau virtuel s'intègre harmonieusement avec le mécanisme de circulation des trains : les trains suffisent à valider l'arrivée d'un nouveau membre auprès de tous les anciens membres du circuit. Il n'y a pas besoin de faire circuler de messages d'un autre type.

Ce travail ouvre plusieurs perspectives d'un point de vue algorithmique. Nous avons superposé plusieurs anneaux sur un seul et même anneau : notre proposition pourrait-elle trouvé d'autres usages dans des algorithmes s'appuyant sur une topologie en anneau ? Par ailleurs, nous proposons un service de gestion des participants au protocole d'uot-diffusion. Est-ce que ce service pourrait être utilisé à d'autres fins que le protocole d'uot-diffusion. Enfin, de nombreux protocoles d'uot-diffusion s'appuient sur un service de gestion des membres extérieur au protocole [Birman, 2010]. Par exemple, le protocole LCR de [Guerraoui *et al.*, 2010] utilise le service de l'intergiciel *Spread* [Spr98]. Quelles seraient les conséquences de cette « externalisation » sur nos algorithmes ? Selon [Urban *et al.*, 2003], nous perdrions au moins en réactivité en cas de défaillances. Mais quels seraient les gains ?

Une autre perspective pour notre protocole est son utilisation au sein d'intergiciels qui nécessitent des débits élevés. En effet, notre protocole agrège les messages de manière naturelle au sein de wagons, puis de trains. Ce comportement laisse présager de bonnes performances en termes de débit. Nous pourrions donc intégrer notre protocole à des intergiciels en vue de mesurer les performances résultantes. Nous pensons plus particulièrement à *Hadoop Distributed File System* qui permet la réplication et la distribution de blocs de données entre des nœuds d'une grappe [Apa07], et à *Hazelcast* qui offre des structures de données Java réparties [Haz10]. Avant de nous engager dans une telle voie, nous nous proposons d'étudier notre protocole de manière théorique. Ainsi, nous démontrerons que c'est effectivement un protocole d'uot-diffusion. Et, nous estimerons les performances envisageables avec ce protocole.

Chapitre 5

Propriétés du protocole des trains

Ce chapitre s'intéresse aux propriétés du protocole des trains. La section 5.1 montre que ce protocole est un protocole d'uot-diffusion. Toutes les propriétés sont évaluées pour la version multitrain. La section 5.2 estime les performances de ce protocole d'un point de vue théorique. Cette estimation révèle les limites des métriques identifiées dans l'état de l'art. Aussi, nous définissons une nouvelle métrique : le rendement en terme de débit. Nous montrons que cette métrique permet non seulement de comparer finement des protocoles performants en termes de débit, mais d'estimer le débit maximum théorique qu'une implantation de cet algorithme peut atteindre. La section 5.3 conclut.

5.1 Correction du protocole des trains

Cette section montre que le protocole des trains (version multitrain) est correct, c'est-à-dire que c'est effectivement un protocole d'uot-diffusion.

Pour ce faire, nous commençons par démontrer que, lorsqu'un processus p démarre, p atteint l'état `DansCircuitSeul` ou l'état `DansCircuitÀPlusieurs`, ou bien p est dans l'état `HorsCircuit` (attendant d'avoir reçu des trains lui confirmant son insertion dans le circuit), ou bien encore p signale l'échec d'insertion à la couche applicative (cf. lemme 2). Puis, nous montrons que, lorsque p atteint l'état `DansCircuitSeul` (respectivement `DansCircuitÀPlusieurs`), p reste dans cet état ou bien passe dans l'état `DansCircuitÀPlusieurs` (respectivement `DansCircuitSeul`) (cf. lemme 3). Les lemmes et théorèmes suivants sont destinés à compléter le lemme 2 en contribuant à montrer qu'au bout de quelques secondes en moyenne et avec une forte probabilité, p atteint forcément l'état `DansCircuitSeul` ou l'état `DansCircuitÀPlusieurs` (cf. lemme 7) : 1) nous démontrons que l'envoi du message `NOUVEAU_SUCC` à un membre confirmé du circuit des trains provoque la (re)mise en circulation de trains qui ne circulaient pas(/plus) (cf. lemme 4) ; 2) nous montrons que les trains circulent en permanence dès qu'il y a au moins deux processus sur le circuit des trains (cf. lemme 5) ; 3) nous démontrons qu'en cas de défaillance d'un processus, un train récent est reçu par tous les processus restant dans le circuit des trains (cf. lemme 6) ; 4) nous estimons la probabilité que deux processus qui cherchent à rejoindre simultanément un circuit vide soit à nouveau en conflit après $k \in \mathbb{N}^*$ attente(s) dans l'état `AttenteAvantNouvelleTentativeInsertion` (cf. théorème 2) ; 5) nous estimons la proba-

bilité qu'au moins 2 processus parmi n processus qui démarrent simultanément n'arrivent pas à rejoindre le circuit des trains au bout de $\nu \in \mathbb{N}^*$ attentes (cf. théorème 3), et enfin 6) nous estimons le temps moyen d'attente pour rejoindre le circuit des trains quand n processus rejoignent simultanément un circuit vide (cf. théorème 4). Ainsi, nous pouvons démontrer le lemme 7. En y ajoutant une propriété sur le numéro de tour (cf. lemme 8) et l'ordre des trains (cf. lemme 9), nous sommes alors en mesure de montrer les propriétés de validité (cf. lemme 10), accord uniforme (cf. lemme 11), intégrité (cf. lemme 12) et ordre total (cf. lemme 13) définies à la section 3.1.2. Nous en concluons que le protocole des trains est un protocole d'uot-diffusion (cf. théorème 5). Enfin, nous montrons que le protocole des trains respecte l'ordre FIFO (cf. lemme 14) et l'ordre causal (cf. corollaire 1).

Lemme 2. *Lorsqu'un processus p démarre, il est dans l'état* `HorsCircuit`*. Après zéro ou plusieurs passages par l'état* `AttenteAvantNouvelleTentativeInsertion`*, puis à nouveau l'état* `HorsCircuit`*,* in fine*, p atteint l'état* `DansCircuitSeul` *ou l'état* `DansCircuitÀPlusieurs`*, ou bien p est dans l'état* `HorsCircuit` *(attendant d'avoir reçu des trains lui confirmant son insertion dans le circuit), ou bien encore p signale l'échec d'insertion à la couche applicative.*

Démonstration. Lorsque le processus p démarre, son initialisation des données du protocole l'amène dans l'état `HorsCircuit` (ligne 9.22).

Supposons que le processus p démarre alors qu'il n'y a aucun processus dans le circuit des trains et qu'aucun autre processus ne démarre simultanément. Alors, p ne découvre aucun autre successeur que lui-même. Il se retrouve dans l'état `DansCircuitSeul` (lignes 10.2–3 et 10.7).

Supposons que le processus p démarre alors qu'il n'y a aucun processus dans le circuit des trains et qu'un (ou plusieurs) autre(s) processus démarrent simultanément. p et l'autre (les autres) processus commence(nt) par indiquer qu'ils participent au protocole des trains (ligne 10.1). De ce fait, la recherche d'un successeur par p (ligne 10.2) renvoie un processus p_s différent de p. Dans le cas où p_s défaille juste après que p l'ait identifié comme successeur, p ne réussit pas à ouvrir de connexion avec p_s. p se retrouve dans l'état `AttenteAvantNouvelleTentativeInsertion` (lignes 10.9–10). Une fois dans cet état, p indique qu'il ne participe plus pour l'instant au protocole (ligne 4.1). Puis, si p a déjà fait plus de `NB_ATTENTES_MAX` attentes, p signale l'échec d'insertion à la couche applicative : elle décidera des suites à donner (lignes 4.6–7). Sinon p attend pendant une durée aléatoire, avant de repartir dans l'état `HorsCircuit` (lignes 4.9–11). Dans le cas où p_s ne défaille pas, p réussit sa connexion avec p_s et lui envoie le message `INSERT` (ligne 10.12). Vu que nous sommes dans le cas d'un démarrage simultané de processus, nous supposons ici que p_s est encore dans l'état `HorsCircuit`. p_s réagit donc en renvoyant `NAK_INSERT` (lignes 10.18–19 ou 32–33). La réception de ce message `NAK_INSERT` fait basculer p dans l'état `AttenteAvantNouvelleTentativeInsertion` (lignes 10.16–17). Remarquez que p réagit de la même manière si p_s défaille avant d'avoir renvoyé `NAK_INSERT`. En effet, cette défaillance entraîne une perte de connexion (message `PERTE_CONNEXION`) au niveau de p.

Supposons que le processus p démarre alors qu'il n'y a au moins un processus p_s dans le circuit des trains et qu'un (ou plusieurs) autre(s) processus démarre(nt) simultanément. Supposons également que p_s est le processus identifié par la recherche de successeur faite

par p (notez que si p identifie plutôt un processus en train de démarrer, il aura un comportement semblable à celui décrit au paragraphe précédent). Lorsque p envoie `INSERT` à p_s, comme p_s fait partie du circuit des trains, p_s est dans l'état `DansCircuitSeul` ou l'état `DansCircuitÀPlusieurs`. p_s renvoie `ACK_INSERT` avec l'adresse de son propre prédécesseur p_p (lignes 11.3–6 ou 12.13–15). Sur réception de ce `ACK_INSERT`, p ouvre une connexion avec p_p (p bascule dans l'état `AttenteAvantNouvelleTentativeInsertion` en cas d'échec) et lui envoie `NOUVEAU_SUCC` (lignes 10.22–26). p attend ensuite de recevoir `ntr` trains (d'identifiants différents) dont le circuit contient p (lignes 10.27–41). Tant que cette condition n'est pas remplie, p reste dans l'état `HorsCircuit`. En revanche, une fois que cette condition est remplie, p passe dans l'état `DansCircuitÀPlusieurs` (lignes10.44). Notez que, pendant l'attente des lignes 10.27–41, si p reçoit un message `NOUVEAU_SUCC` d'un autre processus ou bien si p perd sa connexion avec p_p ou p_s, p bascule dans l'état `AttenteAvantNouvelle-TentativeInsertion` (lignes 10.30–31). En revanche, pendant cette attente, si p reçoit un message `INSERT`, p renvoie `NAK_INSERT` (lignes 10.32–33). □

Lemme 3. *Lorsqu'un processus p atteint l'état `DansCircuitSeul` (respectivement `DansCircuitÀPlusieurs`), p reste dans cet état ou bien passe dans l'état `DansCircuitÀPlusieurs` (respectivement `DansCircuitSeul`). De plus, dans l'état `DansCircuitÀPlusieurs`, p peut signaler un problème de partitionnement à la couche applicative : cette dernière décide des suites à donner (suicide par défaut).*

Démonstration. Supposons qu'un processus p est dans l'état `DansCircuitSeul`. S'il reçoit le message `INSERT`, il renvoie `ACK_INSERT` (lignes 11.3–6). Si p reçoit ensuite `PERTE_CONNEXION`, il reste dans l'état `DansCircuitSeul` (lignes 11.11–12). En revanche, si p reçoit ensuite `NOUVEAU_SUCC`, il envoie `ntr` trains et passe dans l'état `DansCircuitÀPlusieurs` (lignes 11.8–26). Étudions maintenant le cas où p est dans l'état `DansCircuitÀPlusieurs`. Si p reçoit un train et qu'il constate que le train de ce circuit ne contient pas p, p signale un problème de partitionnement à la couche applicative (lignes 12.4 et 12.11). Cette dernière décide des suites à donner (suicide par défaut). Si p est dans le circuit de ce train, il le traite et reste dans l'état `DansCircuitÀPlusieurs` (lignes 12.6–8 et 12.52). Si p reçoit `INSERT` ou `NOUVEAU_SUCC`, il traite ce message et reste dans l'état `DansCircuitÀPlusieurs` (ligne 12.52). Si p reçoit `PERTE_CONNEXION`, si cette perte concerne son prédécesseur et si p ne trouve aucun autre processus auquel se connecter, il passe dans l'état `DansCircuitSeul` (lignes 12.26–47). Sinon, p reste dans l'état `DansCircuitÀPlusieurs` (ligne 12.52). □

Dans la suite, nous ignorons les cas où un processus dans l'état `DansCircuitÀPlusieurs` signale un problème de partitionnement à la couche applicative.

Les lemmes et théorèmes suivants sont destinés à compléter le lemme 2 en contribuant à montrer de manière probabiliste qu'*in fine*, p atteint forcément l'état `DansCircuitSeul` ou l'état `DansCircuitÀPlusieurs`.

Lemme 4. *L'envoi du message `NOUVEAU_SUCC` à un membre confirmé du circuit des trains provoque la (re)mise en circulation de trains qui ne circulaient pas(/plus). Les éventuels doublons de train que provoque cette remise en circulation sont supprimés au niveau du premier processus dans l'état `DansCircuitÀPlusieurs` que ces doublons rencontrent.*

Démonstration. Soit p, un processus qui est membre confirmé du circuit des trains. p est dans l'état `DansCircuitSeul` ou bien dans l'état `DansCircuitÀPlusieurs`. Supposons que p est dans l'état `DansCircuitSeul`. Alors, p est seul sur le circuit. Donc, p ne peut recevoir `NOUVEAU_SUCC` que parce qu'un processus p' cherche à rejoindre le circuit des trains. p' envoie à p le message `INSERT` (ligne 10.12). Comme p est dans l'état `DansCircuitSeul`, il réagit en renvoyant le message `ACK_INSERT` (ligne 11.6). p' renvoie alors le message `NOUVEAU_SUCC` (ligne 10.26). Sur réception de ce dernier, p réagit en envoyant `ntr` trains à p' (lignes 11.19–25) en mettant p et p' dans chaque circuit des trains (ligne 11.22). p envoie d'abord le train d'identifiant (`die` + 1) mod `ntr`, puis le train d'identifiant (`die` + 2) mod `ntr`, etc. p passe ensuite dans l'état `DansCircuitÀPlusieurs` (ligne 11.26). Comme nous utilisons un canal TCP entre p et p', p' reçoit les trains dans l'ordre dans lequel p les a envoyés. Quand p' reçoit chacun de ces trains, p' constate qu'il appartient au champ `circuit`, incrémente le champ `st.lc` (ligne 10.36) avant de renvoyer ce train à p. Revenons à p : il reçoit d'abord un train d'identifiant (`die` + 1) mod `ntr`, dont le circuit contient p, et dont le champ `st.lc` vaut 1 de plus que le champ `st.lc` du dernier train qu'il a envoyé. p le considère donc comme un train récent (lignes 12.4–5 et 14.2–3). Aussi p le traite (ligne 12.6). Il incrémente notamment `st.lc` (ligne 13.28) et envoie ce train à p' (ligne 12.8). De même pour le train d'identifiant (`die` + 2) mod `ntr`, etc.

Supposons maintenant que p est dans l'état `DansCircuitÀPlusieurs`. p peut recevoir `NOUVEAU_SUCC` soit parce que son successeur p_s défaille et que le successeur p_{s_s} de p_s se reconnecte à p, soit parce qu'un processus p' s'insère dans le circuit des trains entre p et p_s. Dans le cas où p_s défaille, il se peut qu'il détienne alors un (ou plusieurs) trains récents et qu'il n'ait pas le temps de les faire suivre à son successeur p_{s_s} avant la défaillance. La circulation de ces trains pourrait donc s'arrêter. Toutefois, comme p_s défaille, sa connexion avec p_{s_s} est rompue. p_{s_s} réagit en se connectant à p et en lui envoyant `NOUVEAU_SUCC` (lignes 12.24–34). p envoie alors tous les derniers trains qu'il avait envoyés à p_s (lignes 12.21–12.23). Grâce au test du caractère récent d'un train (ligne 12.5), p_{s_s} ignore les trains que p_s lui avait déjà fait suivre. En effet, la ligne 13.28 garantit que, si $\texttt{dte}_p\texttt{[st.id]}$ (qui correspond à $\texttt{tr}_{p_{s_s}}$ chez p_{s_s}) est un train que p_{s_s} a déjà vu passer, alors $\texttt{tr}_{p_{s_s}}\texttt{.st.lc}$ est inférieur ou égal à $\texttt{dte}_{p_{s_s}}\texttt{[tr.st.id].st.lc}$: p_{s_s} décide d'ignorer le train $\texttt{tr}_{p_{s_s}}$. En revanche, ce test du caractère récent d'un train permet de ne pas ignorer les trains qui ont été perdus au moment de la défaillance de p_s et qui ont donc été remis en circulation à raison par p. Étudions maintenant le cas où un processus p' s'insère entre p et p_s. p' est dans l'état `HorsCircuit`. Il envoie `INSERT` à p_s. Ce dernier ferme sa connexion avec p et répond `ACK_INSERT` à p' avec l'adresse de p en paramètre (lignes 12.13–17). p' envoie `NOUVEAU_SUCC` à p. Aussi, p envoie à p' tous les derniers trains qu'il avait envoyé à p_s (certains ont pu être perdus entre le moment où p_s a fermé sa connexion avec p et le moment où p' a envoyé `NOUVEAU_SUCC` à p). p' se contente de les faire suivre à p_s. En s'appuyant sur l'horloge logique, p_s supprime les éventuels doublons générés par cette remise en circulation. □

Lemme 5. *S'il y a au moins deux processus sur le circuit des trains, les trains circulent en permanence. Par ailleurs,* in fine*, chacun de ces processus est dans l'état* `DansCircuitÀPlusieurs`*.*

Démonstration. Pour cette démonstration, nous étudions tout d'abord le cas où un processus rejoint un circuit où il n'y avait qu'un seul processus. Ensuite, nous nous intéressons

au cas d'insertion d'un processus dans un circuit où il y a au moins deux processus. Puis, nous étudions le cas où un processus du circuit a défailli (et qu'il reste au moins deux processus). Enfin, nous considérons le cas d'une insertion et d'une défaillance simultanée.

Circulation perpétuelle lorsqu'un processus rejoint un circuit des trains ne contenant qu'un processus. Si un processus p est l'unique membre du circuit des trains, alors l'automate de p est dans l'état `DansCircuitSeul`. En effet :

- si p démarre seul sur le circuit des trains, alors il est dans l'état `HorsCircuit`, constate qu'il est seul en recherchant un successeur (lignes 10.2 et 10.3) et bascule dans l'état `DansCircuitSeul` (ligne 10.7) ;
- si p a été dans l'état `DansCircuitSeul`, a été contacté par un autre processus auquel il a envoyé `ACK_INSERT` (ligne 11.6), a constaté le départ de cet autre processus (ligne 11.11), alors il est resté dans l'état `DansCircuitSeul` (ligne 11.12) ;
- si p a été précédemment dans l'état `DansCircuitÀPlusieurs`, alors tous les autres processus ont quitté le circuit des trains, p a cherché un nouveau prédécesseur (lignes 12.26–34), n'en a pas trouvé et a basculé dans l'état `DansCircuitSeul` (ligne 12.47).

Supposons que le processus p' souhaite se joindre au circuit des trains sur lequel p est seul. Dans la démonstration du lemme 4, nous avons vu que p met en circulation tous les trains, puis bascule dans l'état `DansCircuitÀPlusieurs`. En plus, quand p' reçoit le dernier train mis en circulation par p (donc le train d'identifiant (`die` + `ntr`) mod `ntr`, c'est-à-dire `die`), p' constate qu'il a reçu tous les trains et qu'il était membre du circuit de chacun de ces trains. Il mémorise donc l'identifiant du dernier train qu'il a envoyé (ligne 10.42), puis bascule dans l'état `DansCircuitÀPlusieurs` (ligne 10.44). De ce fait, quand p envoie à p' le train d'identifiant (`die` + 1) mod `ntr`, p' traite ce train. Nous constatons qu'avec deux processus, les trains circulent en permanence et que les processus sont dans l'état `DansCircuitÀPlusieurs`.

Circulation perpétuelle lorsqu'un processus rejoint un circuit des trains contenant au moins deux processus dans l'état `DansCircuitÀPlusieurs`. Procédons par récurrence. Soit un processus p qui s'insère avant un processus p_s et après un processus p_p, p_s et p_p étant les seuls processus du circuit des trains qui sont dans l'état `DansCircuitÀPlusieurs` dans le circuit des trains [1]. Comme nous l'avons vu lors de la démonstration du lemme 4, lorsque p envoie le message `NOUVEAU_SUCC` à p_p pour lui signaler qu'il est son nouveau successeur, p_p renvoie à p tous les derniers trains qu'il a envoyés à p_s. Et p se contente de faire suivre à p_s ces trains reçus de p_p (ligne 10.38) : p permet ainsi aux trains de circuler perpétuellement sur le circuit. Lorsque p_s reçoit de p un train récent d'identifiant 0 (lignes 12.13 et 13.8), il ajoute p au circuit de ce train (ligne 13.9). Donc, pour les trains récents suivants qui ont un identifiant différent de 0, p_s met un circuit contenant p dans ce train

1. p_p et p_s ne sont pas dans l'état `AttenteAvantNouvelleTentativeInsertion`, puisqu'ils participent tous les deux au protocole. p_p et p_s ne sont pas non plus dans l'état `DansCircuitSeul`, sinon ils n'auraient pas conscience qu'ils sont l'un et l'autre voisins sur le circuit des trains. Si p_p est dans l'état `HorsCircuit`, l'envoi de `NOUVEAU_SUCC` par p à p_p fait que p_p ferme toutes ses connexions, avant de passer dans l'état `AttenteAvantNouvelleTentativeInsertion`. Donc la connexion avec p est fermée, ce qui fait que p passe dans l'état `AttenteAvantNouvelleTentativeInsertion`. Si p_s est dans l'état `HorsCircuit`, il réagit au message `INSERT` de p par un message `NAK_INSERT`. Donc, p passe dans l'état `AttenteAvantNouvelle-TentativeInsertion`. Par conséquent, il est légitime de se concentrer sur le cas où p_p et p_s sont dans l'état `DansCircuitÀPlusieurs`

(ligne 13.14). p_s envoie chacun de ces trains à son successeur, c'est-à-dire p_p (ligne 12.8). p_p fait de même vers son propre successeur, c'est-à-dire p (ligne 12.8). Puisque p a reçu `ntr` trains d'identifiants différents et contenant p dans leur circuit, p bascule dans l'état `DansCircuitÀPlusieurs` (ligne 10.44). p applique ensuite la même logique que les autres processus du circuit : le train circule en permanence. Ainsi, si nous supposons que le circuit contient deux processus dans l'état `DansCircuitÀPlusieurs`, et que le train circule en permanence, nous démontrons que, quand un troisième processus s'insère dans le circuit, il passe *in fine* dans l'état `DansCircuitÀPlusieurs`, et le train continue à circuler en permanence. Supposons qu'il y a n processus ($n \geqslant 2$) dans le circuit et que ces n processus sont dans l'état `DansCircuitÀPlusieurs`. En reprenant le même raisonnement que pour $n = 2$ ci-dessus, nous montrons qu'un $(n+1)^e$ processus qui s'insère dans le circuit passe *in fine* dans l'état `DansCircuitÀPlusieurs` et que le train continue à circuler en permanence. Par récurrence, la propriété est donc vraie pour tout $n \geqslant 2$.

Circulation perpétuelle en cas de défaillance de processus (et qu'il reste au moins deux processus dans le circuit). La démonstration du lemme 4 montre que la défaillance d'un processus n'interrompt pas la circulation des trains. Par récurrence, ce raisonnement peut être généralisé à la défaillance de plusieurs processus.

Circulation perpétuelle en cas d'insertion(s) dans un circuit avec au moins 2 processus avant l'insertion et de défaillance(s) simultanée(s)) de processus. Soit un processus p qui rejoint le circuit des trains, alors que ce circuit contient au moins 2 processus. p s'insère avant un processus p_s et après le processus p_p. Par hypothèse, le circuit des trains contient au moins deux processus avant l'insertion de p. Donc, p_p est différent de p_s. Supposons qu'un processus p' s'arrête pendant l'insertion de p. Quatre cas se présentent. Dans le cas où p' est différent de p_p, de p et de p_s, il n'y a aucune influence sur l'insertion de p : les éléments précédents de la démonstration s'appliquent. Dans le cas où p' est p, p_s se reconnecte à p_p et lui envoie `NOUVEAU_SUCC` : nous sommes dans le cas de la circulation perpétuelle en cas de défaillance de processus. Dans le cas où p' est p_p (respectivement p_s), alors p perd sa connexion avec son prédécesseur (respectivement successeur) : p se met en attente avant de faire une nouvelle tentative d'insertion (lignes 10.16 et 10.17 ou bien 10.30 et 10.31). p_s (respectivement p_p) se retrouve alors dans le cas de la défaillance de 2 processus, cas qui a été étudié précédemment. Ce raisonnement est généralisable à l'insertion simultanée de plusieurs processus et la défaillance simultanée de plusieurs processus.

□

Lemme 6. *En cas de défaillance d'un processus, un train récent est reçu par tous les processus restant dans le circuit des trains.*

Démonstration. Supposons qu'un processus p défaille alors que son prédécesseur p_p venait de lui envoyer un (ou plusieurs) train(s) que p n'a pas eu le temps de faire suivre à son successeur p_s. Soit $i \in [\![0, \texttt{ntr}[\![$ l'identifiant de l'un de ces trains perdus. Pour ce train d'identifiant i, nous devons considérer les deux cas suivants. 1) p n'a pas eu le temps de faire un quelconque traitement sur ce train : les wagons de ce train mémorisé dans $\texttt{dte}_{p_p}[\texttt{i}]$ sont les mêmes que les wagons qu'avait p. 2) p a eu le temps de traiter ce train, mais a été interrompu dans l'envoi de ce train à p_s (p_s ignore donc ce train incomplet). La seule information supplémentaire par rapport à la version mémorisée dans $\texttt{dte}_{p_p}[\texttt{i}]$ est le wagon

w_p que p a ajouté en queue de ce train (ligne 13.26). Comme p a défailli, ce n'est pas un processus correct. De ce fait, même si w_p contient des messages que p souhaitait uot-diffuser, la propriété 1 (validité) des protocoles d'uot-diffusion ne nous impose aucune contrainte quant à l'uot-livraison des messages contenus dans w_p. Nous pouvons donc notamment considérer que ces messages à uot-diffuser n'ont jamais existé. Alors, les wagons de ce train mémorisé dans $\texttt{dte}_{p_p}[\texttt{i}]$ sont les mêmes que les wagons qu'aurait eu p.

Dans les deux cas, p_p détient un train aussi à jour que celui de p au moment de la défaillance. Le lemme 4 conduit à la remise en circulation du train $\texttt{dte}_{p_p}[\texttt{i}]$ qui est donc reçu par tous les processus restant dans le circuit des trains. □

Avant de pouvoir démontrer le lemme 7, nous estimons : 1) la probabilité que deux processus qui cherchent à rejoindre simultanément un circuit vide soit à nouveau en conflit après $k \in \mathbb{N}^*$ attente(s) dans l'état `AttenteAvantNouvelleTentativeInsertion` (cf. théorème 2), 2) la probabilité qu'au moins 2 processus parmi n processus qui démarrent simultanément n'arrivent pas à rejoindre le circuit des trains au bout de $\nu \in \mathbb{N}^*$ attentes (cf. théorème 3), et enfin 3) le temps moyen d'attente pour rejoindre le circuit des trains quand n processus rejoignent simultanément un circuit vide (cf. théorème 4). Nous présentons ensuite une application numérique qui permet de donner des ordres de grandeur de ces probabilités.

Définition 4. *Pour pouvoir faire ces estimations, nous définissons les constantes suivantes :*

- $\nu \in \mathbb{N}^*$ *est l'abréviation de* `NB_ATTENTES_MAX`*. Donc,* ν *désigne le nombre maximum de fois qu'un processus* p *a le droit de faire un tirage aléatoire d'attente du fait que* p *cherche à entrer dans le circuit des trains en même temps qu'un autre processus ;*
- D *est le délai maximum qui s'écoule entre le moment où un processus* p *entre dans l'état* `HorsCircuit` *et le moment où il entre dans l'état* `DansCircuitÀPlusieurs` *(sans transiter par l'état* `AttenteAvantNouvelleTentativeInsertion`*).* D *inclut donc le temps pour* p *trouve un processus membre du circuit des trains, lui envoyer* `INSERT`*, en recevoir* `ACK_INSERT`*, envoyer* `NOUVEAU_SUCC`*, et recevoir des trains jusqu'à avoir reçu* `ntr` *trains contenant* p *dans leur circuit. Nous supposons que* $D \ll T$ *(rappelons que* T *est l'attente maximale que peut avoir un processus* p *lors de sa première attente, cf. ligne 4.9).*

Remarques :

1. Comme nous utilisons le protocole TCP, nous ne disposons pas d'une borne supérieure pour les transmissions de message. En effet, la seule garantie offerte par le protocole TCP est du *best effort*. Comme nous avons ajouté un mécanisme de « battement de cœur » qui permet de fermer une connexion TCP sur laquelle le délai de transmission est trop important, nous supposons que chaque transmission de message prend moins de 2 × période de « battement de cœur ».
2. D dépend linéairement du nombre de processus sur le circuit des trains (puisque le temps de rotation des trains dépend linéairement de ce nombre de processus) Théoriquement, il ne peut donc pas avoir de borne supérieur. Pour les calculs qui suivent, nous supposons que le nombre de processus dans le circuit des trains est borné et qu'il en est donc de même pour D.

Théorème 2. *Nous considérons le cas où le circuit des trains est vide. Soient deux processus p_1 et p_2 qui cherchent simultanément à rejoindre le circuit des trains. Notez qu'il n'y a pas d'autres processus qui cherchent à rejoindre le circuit pendant cette phase. Si les processus p_1 et p_2 effectuent leur $k^e, k \in \mathbb{N}^*$, attente(s) dans l'état* `AtenteAvantNouvelleTentativeInsertion`*, la probabilité qu'ils se retrouvent en conflit après cette k^e attente (et rebasculent donc dans l'état* `AtenteAvantNouvelleTentativeInsertion`*) est de l'ordre de :* $(2D-1)/(2^{k-1}T)$.

La preuve de ce théorème a une dominante mathématique qui tranche par rapport aux preuves à dominante algorithmique présentées jusqu'à présent. C'est pourquoi nous reportons cette preuve à l'annexe A page 225.

Théorème 3. *Considérons le cas où le circuit des trains est vide. Soient n processus $p_1,\ldots,p_{n,n\in[\![2,+\infty[\![}$ qui cherchent simultanément à rejoindre le circuit des trains. La probabilité qu'au moins 2 processus parmi $p_1,\ldots,p_n$ ne réussissent pas à s'insérer dans ce circuit au bout de ν tentatives est de l'ordre de :* $[n(n-1)(2D-1)]^\nu/(2^{\frac{\nu(\nu+1)}{2}}T^\nu)$.

De même que pour la preuve du théorème 2, nous reportons la preuve à l'annexe A page 225.

Soit $X_{n,n\in[\![2,+\infty[\![}$ la variable aléatoire donnant le temps qui s'écoule avant que n processus $p_1,\ldots,p_n$ qui cherchent simultanément à rejoindre un circuit des trains (qui ne contiendra que $p_1,\ldots,p_n$) réussissent à le faire. Si $p_1,\ldots,p_n$ attendent pour la k^e fois, $k \in \mathbb{N}^*$, l'attente de ces processus est une valeur aléatoire dans $[\![0, 2^{k-1}T-1]\!]$. Elle vaut donc $\frac{(2^{k-1}-1)T}{2} \simeq \frac{2^{k-1}T}{2}$ en moyenne.

La variable aléatoire X_2 prend donc les valeurs (approchées) $x_{n_1} = \frac{T}{2}; \ldots; x_{n_\nu} = \frac{2^{\nu-1}T}{2}$.

Comme pour la variable aléatoire X_2, la variable aléatoire X_n prend les valeurs (approchées) $\frac{T}{2}; \ldots; \frac{2^{\nu-1}T}{2}$.

Théorème 4. *Si nous supposons que T est suffisamment grand pour que* $\forall i \in [\![2,+\infty[\![, \frac{1}{T^i} \ll \frac{1}{T}$*, alors :* $E(X_n) \simeq \frac{T}{2} + \frac{n(n-1)(2D-1)}{4}$

De même que pour la preuve du théorème 2, nous reportons la preuve à l'annexe A page 225.

Application numérique. Supposons que $n = 4$ processus, $T = 10\,000$ millisecondes, $\nu = 10$ et $D = 200$ millisecondes. Alors, comme $D \ll T$, nous pouvons utiliser les théorèmes précédents. Par conséquent, la probabilité qu'au moins 2 processus parmi les n ne réussissent pas à s'insérer au bout de ν tentatives est de l'ordre de : $2 \cdot 10^{-20}$. Donnons un ordre de grandeur de ce phénomène : un joueur de Loto a environ 1 chance sur 14 millions de trouver les 6 bons numéros ; par conséquent, un utilisateur du protocole des trains a 10^{12} fois moins de chances d'observer ce phénomène que de gagner au Loto. De plus, le temps moyen d'attente en cas de tentative d'insertion simultanée dans un circuit initialement vide est : $E(X_n) \simeq 6\,200$ millisecondes.

Lemme 7. *Lorsqu'un processus p démarre, il est dans l'état* `HorsCircuit`*. Après zéro ou plusieurs passages par l'état* `AtenteAvantNouvelleTentativeInsertion`*, puis à nouveau*

l'état `HorsCircuit`*, au bout de quelques secondes en moyenne et avec une forte probabilité*[2]*, p se retrouve soit dans l'état* `DansCircuitSeul`*, soit dans l'état* `DansCircuitÀPlusieurs`*. Sa participation au protocole des trains est donc confirmée.*

Démonstration. Selon le lemme 2, *in fine*, p se retrouve dans l'état `DansCircuitSeul` ou l'état `DansCircuitÀPlusieurs`, ou bien p est dans l'état `HorsCircuit` (attendant d'avoir reçu des trains lui confirmant son insertion dans le circuit), ou bien encore p signale l'échec d'insertion à la couche applicative. Le théorème 3 montre que la probabilité que p signale l'échec d'insertion est suffisamment faible (de l'ordre de $2 \cdot 10^{-20}$ dans le cas de notre application numérique) pour être considérée comme nulle. Par ailleurs, les lemmes 5 et 6 montrent que p reçoit *in fine* `ntr` trains dont le circuit contient p. De plus, le théorème 4 montre que le temps avant de basculer dans l'état `DansCircuitSeul` ou dans l'état `DansCircuitÀPlusieurs` est en moyenne de quelques secondes (environ 6 secondes dans le cas de notre application numérique). □

Lemme 8. *Soient $i \in [\![0, \texttt{ntr}[\![, j \in \mathbb{Z}, k \in [\![0, \texttt{NR}[\![$. Pour tout $m \in \mathbb{N}^*$, si un processus p a envoyé un train $t_{\{i,j,k\}}$ à un instant donné, p considère que le train d'identifiant i qu'il reçoit au m^e tour après l'envoi de $t_{\{i,j,k\}}$ a pour numéro de tour $(k+m)$ mod* `NR`*.*

Démonstration. Supposons que le train $t_{\{i,j,k\}}$ envoyé par p fasse un tour de circuit sans que le circuit ne connaisse de changement (insertion ou départ de processus). p reçoit un train $t_{\{i,j',k'\}}, j' \in]\!]j, +\infty[\![, k' \in [\![0, \texttt{NR}[\![$. Dans le cas où $k' == k$, p incrémente le numéro de tour de ce train avant de le traiter (lignes 13.3 et 13.4). De ce fait, p considère que ce train a pour numéro de tour $(k+1)$ mod `NR`. Dans le cas où $k' \neq k$, alors k' vaut forcément $(k+1)$ mod `NR`. En effet, si $k' \neq k$, cela signifie qu'un processus du circuit des trains a incrémenté le numéro de tour de ce train (lignes 13.3 et 13.4). Et les successeurs de ce processus (dont p) n'ont pas fait de nouvel incrément puisque le test de la ligne 13.3 était forcément faux. Une fois encore, p considère que ce train a pour numéro de tour $(k+1)$ mod `NR`.

Supposons maintenant que le train $t_{\{i,j,k\}}$ envoyé par p fasse un tour de circuit et que le circuit connaisse une insertion d'un processus noté p'. Supposons que p' a envoyé `NOUVEAU_SUCC` à son prédécesseur. Tant que p' n'a pas reçu `ntr` trains dont le circuit contient p', p' reste dans l'état `HorsCircuit`. De ce fait, sur réception d'un train, p' ne modifie pas son numéro de tour. p' se contente de faire suivre le train reçu à son successeur en incrémentant éventuellement son horloge logique (lignes 10.35 à 10.38). Pour p, nous pouvons donc reprendre le raisonnement concernant le cas en l'absence d'insertion ou de départ : p considère donc que le train reçu a pour numéro de tour $(k+1)$ mod `NR`.

Supposons maintenant que le train $t_{\{i,j,k\}}$ envoyé par p fasse un tour de circuit et que le circuit soit témoin d'une défaillance d'un processus p'. Une fois que le circuit est reconstruit, il est simplement plus court. Pour p, nous pouvons donc reprendre le raisonnement concernant la circulation du train en l'absence d'insertion ou de départ : p considère que le train reçu a pour numéro de tour $(k+1)$ mod `NR`.

2. Cette moyenne et cette probabilité dépendent du nombre n de participants, du temps maximum d'attente T lors d'un premier conflit d'insertion, du délai maximum D pour qu'un processus trouve un successeur ou bien soit identifié comme successeur, et du nombre maximum ν de tentatives d'insertions infructueuses.

Faisons un changement de variable en prenant $k_1 = k+1$. En reprenant le raisonnement des paragraphes précédents, nous montrons qu'au bout d'un nouveau tour, p considère que le train reçu a pour numéro de tour $(k_1 + 1) \bmod \texttt{NR}$. Donc, au bout de deux tours, p considère que le train reçu a pour numéro de tour $((k+1)+1) \bmod \texttt{NR} = (k+2) \bmod \texttt{NR}$.

Par récurrence, nous concluons qu'au bout du m^e tour, p considère qu'il a reçu le train de numéro de tour $(k+m) \bmod \texttt{NR}$. □

Lemme 9. *Dans l'histoire du système, les trains circulent toujours dans l'ordre* $t_{\{0,.,0\}}$ *(le « . » représentant une valeur quelconque pour l'horloge logique), puis* $t_{\{1,.,0\}} \ldots$ *et* $t_{\{\texttt{ntr}-1,.,0\}}$*, suivi de* $t_{\{0,.,1\}} \ldots t_{\{\texttt{ntr}-1,.,1\}}$*, puis de* $t_{\{0,.,\texttt{NR}-1\}} \ldots t_{\{\texttt{ntr}-1,.,\texttt{NR}-1\}}$*, puis à nouveau* $t_{\{0,.,0\}}$*, etc. En particulier, un train de double jamais un autre train.*

Démonstration. Dans l'histoire du système, le premier processus p_0 qui démarre aboutit dans l'état `DansCircuitSeul` (cf. lemme 3). Sa variable `die` vaut alors `ntr`-1 (ligne 9.3). De ce fait, quand un processus $p_1 \neq p_0$ démarre et envoie `INSERT`, puis `NOUVEAU_SUCC` à p_0, ce dernier envoie les trains $t_{\{0,.,0\}} \ldots t_{\{\texttt{ntr}-1,.,0\}}$ à p_1 (lignes 11.19–25). Ces trains sont envoyés dans cet ordre sur le canal TCP entre p_0 et p_1, car la sémantique de la procédure `envoyer()` fait que deux appels successifs à `envoyer()` ne risquent pas de mélanger les octets des messages envoyés. De plus, ces trains sont reçus dans cet ordre par p_1, car ils circulent sur le même canal TCP qui garantit l'ordre de livraison des messages. p_1 reçoit donc les trains dans l'ordre $t_{\{0,.,0\}} \ldots t_{\{\texttt{ntr}-1,.,0\}}$. Par conséquent, p_1 les renvoie dans cet ordre à p_0 (lignes 10.35–38). p_0 augmente alors le numéro de tour (lignes 13.3 et 13.4). Ensuite, le système voit la circulation de $t_{\{0,.,1\}} \ldots$ et $t_{\{\texttt{ntr}-1,.,1\}}$. Les trains $t_{\{0,.,\texttt{NR}-1\}} \ldots t_{\{\texttt{ntr}-1,.,\texttt{NR}-1\}}$ circulent ensuite, puis à nouveau $t_{\{0,.,0\}} \ldots t_{\{\texttt{ntr}-1,.,0\}}$.

L'insertion d'un processus p_i dans le circuit des trains ne change rien à cet ordre de circulation. En effet, dans l'état `HorsCircuit`, p_i se contente de faire suivre le train sans incrémenter (respectivement en incrémentant) son horloge logique si le circuit de ce train ne contient pas (respectivement contient) p_i (lignes 10.35–38).

En cas de faute d'un processus p, son successeur p_s se connecte au prédécesseur p_p de p et envoie `NOUVEAU_SUCC` à p_p (lignes 12.24–29). Selon le lemme 4, les trains qui ne circulaient plus sont remis en circulation, et les éventuels doublons de train que provoque cette remise en circulation sont supprimés au niveau de p_s. □

Lemme 10 (Validité)**.** *Si un processus correct uot-diffuse un message* m*, alors il uot-livre* in fine *le message* m*.*

Démonstration. Après son démarrage, un processus p parvient *in fine* dans l'état `DansCircuitSeul` ou dans l'état `DansCircuitÀPlusieurs` (cf. lemme 7). Et il reste dans l'un de ces deux états (cf. lemme 3). Supposons maintenant que p (qui est correct) uot-diffuse un message m.

Considérons tout d'abord le cas où p est dans l'état `DansCircuitSeul`. L'algorithme de la procédure `uotDiffusion(m)` ajoute m au wagon `wagonAEnvoyer` qui est immédiatement ajouté à la liste `wagonsAUOTLivrer` (lignes 2.1–4).

Intéressons-nous maintenant au cas où p est dans l'état `DansCircuitÀPlusieurs`. L'algorithme de la procédure `uotDiffusion(m)` ajoute m au wagon `wagonAEnvoyer` (ligne 2.1).

Il y a alors deux possibilités : 1) soit tous les autres processus du circuit des trains défaillent avant que p ne reçoive un train récent, 2) soit il reste au moins un autre processus sur le circuit des trains. Si tous les autres processus défaillent (possibilité 1), p ajoute `wagonAEnvoyer` à `wagonsAUOTLivrer` (ligne 12.45). S'il reste au moins un autre processus (possibilité 2), comme les trains circulent en permanence (cf. lemme 5), p reçoit *in fine* un train récent $t_{\{i,j,k\}}, i \in [\![0, \texttt{ntr}[\![, j \in \mathbb{Z}, k \in [\![0, \texttt{NR}[\![$. Supposons que p stocke la valeur $k' \in [\![0, \texttt{NR}[\![$ dans la variable locale `numTour` de l'algorithme 13 (lignes 13.2–5). p affecte ensuite k' à `wagonAEnvoyer.numTour` (ligne 13.24). Puis, il ajoute ce wagon à `wagonsNonStables[`i`][`k'`]` (ligne 13.25). Ensuite, p envoie le train $t_{\{i,j+1,k'\}}$ (ligne 12.8). Dans le cas où ce train fait deux tours de circuit, d'après le lemme 8, p considère que le train reçu a pour numéro de tour $(k'+2)$ mod `NR`. Par conséquent, p ajoute `wagonsNonStables[`i`][`k'`]` à `wagonAUOTLivrer` (ligne 13.6). Dans le cas où ce train ne peut pas faire deux tours de circuit parce que tous les autres processus du circuit quittent le circuit, alors p ajoute également `wagonsNonStables[`i`][`k'`]` à `wagonAUOTLivrer` (lignes 12.37–44). □

Lemme 11 (Accord uniforme)**.** *Si un processus uot-livre un message m, alors tous les processus corrects uot-livreront* in fine *m.*

Démonstration. Soit un processus p qui uot-livre un message m. Dans le cas où p est seul sur le circuit des trains, p a uot-livré m : le lemme est vérifié. Dans le cas où il existe d'autres processus sur le circuit des trains, remarquons que, si p a été en mesure d'uot-livrer le message m, c'est parce que p a transféré m de la variable `wagonsNonStables[`i`][`k`]` (où i est l'identifiant du train qui a transporté m et k est le numéro de tour qui a été donné à m) à la variable `wagonsALivrer` (ligne 13.6). Il y a deux possibilités pour que m se soit retrouvé dans `wagonsNonStables[`i`][`k`]`. Dans la première possibilité, m est un message uot-diffusé par p lui-même. Alors, m a été stocké dans `wagonAEnvoyer` (ligne 2.1). Ce wagon a été transféré dans `wagonsNonStables[`i`][`k`]` (ligne 13.25), avant d'être ajouté à un train (ligne 13.26). Ce dernier a été envoyé sur le circuit des trains (ligne 12.8). Comme les trains circulent en permanence (cf. lemme 5) et qu'un train récent est reçu par tous les processus du circuit des trains (cf. lemme 6), nous avons la garantie que tous les processus corrects ont reçu le wagon avec le message m et l'ont stocké dans `wagonsNonStables[`i`][`k`]` (ligne 13.18). Grâce au lemme 8, nous pouvons affirmer qu'après deux tours du train qui a permis le stockage de m dans `wagonsNonStables[`i`][`k`]`, le contenu de `wagonsNonStables[`i`][`k`]` est transféré dans `wagonsALivrer` (ligne 13.6). Dans la deuxième possibilité, m est un message uot-diffusé par un processus p' avec $p' \neq p$. Alors, le wagon contenant m a été transféré d'un train reçu à la variable `wagonsNonStables[`i`][`k`]` (ligne 13.18). Nous avons la garantie que tous les processus corrects après p' et avant p sur le circuit des trains ont également transféré le wagon contenant m dans `wagonsNonStables[`i`][`k`]`. De plus, si p n'est pas le prédécesseur de p', p ajoute ce wagon au train à envoyer (lignes 13.19–21). Il envoie ensuite ce train à son successeur (ligne 12.8). Ainsi, tous les processus corrects après p et avant p' sur le circuit des trains reçoivent le wagon contenant m et le stockent dans `wagonsNonStables[`i`][`k`]`. Grâce au lemme 8, nous pouvons affirmer qu'après deux tours du train qui a permis le stockage de m dans `wagonsNonStables[`i`][`k`]`, le contenu de `wagonsNonStables[`i`][`k`]` est transféré dans `wagonsALivrer` (ligne 13.6). □

Remarques :

1. Comme dans le protocole LCR de [Guerraoui *et al.*, 2010], l'uniformité est obtenue sans message supplémentaire. Toutefois, le protocole des trains permet d'obtenir l'uniformité sans aucun octet supplémentaire, alors que LCR nécessite de *piggy-backer* sur les messages des octets pour confirmer que les messages deviennent stables. Ce *piggy-backing* induit deux conséquences : LCR impose que chaque message soit identifié de manière unique, ce qui ajoute des octets à chaque message ; chaque message contient des octets pour *piggy-backer* la confirmation, ce qui ajoute d'autres octets ;
2. Le protocole des trains peut être modifié pour devenir un protocole de diffusion *non uniforme* à ordre total : il suffit de modifier les lignes 13.6 et 13.7 pour travailler sur l'indice (`numTour` $-1+$ `NR`) mod `NR` au lieu de l'indice (`numTour` $-2+$ `NR`) mod `NR` comme actuellement. Ainsi, p n'a pas à attendre un nouveau tour de train pour faire la livraison. Mais, l'uniformité n'est plus assurée. Par exemple, une défaillance de p induit le risque que p ait livré des messages que les autres processus ne livreront jamais : ces messages sont ceux uot-diffusés par p et n'ayant pas eu le temps de quitter p.

Lemme 12 (Intégrité)**.** *Pour tout message m, tout processus uot-livre m au plus une fois, et seulement si m a été précédemment uot-diffusé par émetteur(m).*

Démonstration. Pour cette démonstration, nous montrons chaque implication de l'équivalence. Nous démontrons tout d'abord que, si un processus p uot-livre un message m, alors m a été précédemment uot-diffusé par *émetteur*(m). Puis, nous montrons que, si m a été précédemment uot-diffusé par *émetteur*(m), alors tout processus livre m au plus une fois.

Soit p un processus qui uot-livre un message m. Cela signifie que m a été stocké dans un wagon de la variable `wagonsAUOTLivrer`. Ce stockage a pu se faire parce que l'automate de p était dans l'état `DansCircuitSeul` et que p a souhaité uot-diffuser m (lignes 2.1–4). L'autre possibilité est que l'automate de p était dans l'état `DansCircuitÀPlusieurs` et que p a reçu un train d'identifiant i (respectivement p a découvert qu'il s'était retrouvé seul sur le circuit des trains). Alors, p a recopié `wagonsNonStables[`i`][`k`]` contenant m dans `wagonsAUOTLivrer` (ligne 13.6, respectivement 12.26–45). Or, pour pouvoir être dans `wagonsNonStables[`i`][`k`]`, m a été stocké par un processus dans sa variable `wagonAEnvoyer` parce que ce processus souhaitait uot-diffuser m.

Supposons maintenant qu'un processus p uot-diffuse un message m. m est stocké une seule fois dans `wagonAEnvoyer` (ligne 2.1). Dans le cas où p est dans l'état `DansCircuitSeul`, ce wagon est ajouté une seule fois à `wagonsAUOTLivrer` (lignes 2.2–4). Et, la ligne 2.5 garantit qu'un autre wagon contenant m ne sera pas ajouté à `wagonsAUOTLivrer`. Dans le cas où p est dans l'état `DansCircuitÀPlusieurs`, selon le lemme 5, p reçoit *in fine* un train récent $t_{\{i,j,k\}}, i \in [\![0, \texttt{ntr}[\![, j \in \mathbb{Z}, k \in [\![0, \texttt{NR}[\![$. Supposons que p stocke la valeur $k' \in [\![0, \texttt{NR}[\![$, dans la variable locale `numTour` de l'algorithme 13 (lignes 13.2–5). Ensuite, p ajoute `wagonAEnvoyer` à `wagonsNonStables[`i`][`k'`]` (respectivement `dte[`i`].wagons`) (ligne 13.25, respectivement 13.26). Et, la ligne 13.27 garantit qu'un autre wagon contenant m ne sera pas ajouté à `wagonsNonStables[`i'`][k'']` (respectivement `dte[`i'`].wagons`) $\forall i' \in [\![0, \texttt{ntr}[\![, \forall k'' \in [\![0, \texttt{NR}[\![$. Ensuite, `dte[`$i$`]` est envoyé sur le circuit des trains (ligne 12.8). Quand un autre processus p' reçoit ce train d'identifiant i, grâce à l'hor-

loge logique et le test `estCeQueTrainRécent` (ligne 12.5), p' ne le traite qu'une fois, même en cas de remise en circulation du train (cf. lemme 4). Ainsi, p' stocke le wagon contenant m dans `wagonsNonStables[`i`][`k'`]` (ligne 13.18). Deux tours de train plus tard (cf. lemme 8), chacun des processus ajoute `wagonsNonStables[`i`][`k'`]` à `wagonsAUOTLivrer` (ligne 13.6). Et, la ligne 13.7 garantit que m ne reste pas dans un wagon stocké dans `wagonsNonStables[`i`][`k'`]`. Il nous reste à vérifier qu'aucun processus p' ne peut recevoir un autre train contenant le wagon contenant m. Quand un processus p' reçoit un train t, il en retire l'éventuel wagon qui lui appartient, l'éventuel wagon qui appartient à son successeur et les éventuels wagons qui appartiennent à des processus qui ne font plus partie du circuit des trains ou bien dont p' a détecté le départ [3] (lignes 13.17–22). Ces derniers cas peuvent arriver quand des prédécesseurs p'_{p_i} de p' ont quitté le circuit et que p' a stocké leur identité dans `processusPartis`. Quand un train récent arrive, ce train peut contenir des wagons appartenant à p'_{p_i}. Deux cas sont possibles. Dans le premier cas, ces wagons ont déjà été vus par p'. Cela signifie qu'au passage de train précédent, p' les avaient recopiés dans le dernier train qu'il a envoyé (ligne 12.17–21). Donc, tous les successeurs de p' ont vu ces wagons qui n'ont pas été retirés du train du fait du départ de p'_{p_i}. p' doit donc purger ces wagons pour éviter que les processus du circuit des trains uot-livrent deux fois les messages de ces wagons. Dans le deuxième cas, ces wagons n'ont jamais été vus par p'. Dans ce cas, p' a la garantie d'être le premier processus correct à voir ces wagons. En effet, s'il y avait un processus correct p_c qui avait aussi vu ces wagons, cela signifierait que p_c est situé entre p'_{p_i} et p' dans le circuit des trains. Donc, ce serait p_c qui aurait détecté la défaillance de p'_{p_i} et p_c aurait déjà purgé ces wagons. En purgeant ces wagons, p' garantit que les messages stockés dans ces wagons sont uot-livrés zéro fois [4]. □

Lemme 13 (Ordre total)**.** *Pour tout message m et m', si un processus p uot-livre m sans avoir uot-livré m', alors aucun processus p' uot-livre m' avant m.*

Démonstration. Soient 2 messages m et m'. Soit p un processus qui uot-livre m sans avoir uot-livré m'. Nous raisonnons par contradiction en supposant qu'il existe un processus p' qui uot-livre m' avant m.

Il y a deux possibilités : soit p uot-livre m' *après* avoir uot-livré m, soit p n'uot-livre *jamais* m'. Considérons la possibilité que p uot-livre m' après avoir uot-livré m. Si un processus p_m souhaite uot-diffuser un message m, p_m stocke m dans `wagonAEnvoyer` (ligne 2.1). Soit $t_{\{i,.,k\}}(i \in [\![0, \texttt{ntr}[\![, k \in [\![0, \texttt{NR}[\![)$ le train qui passe au niveau de p_m juste après. p_m détermine le numéro de tour k' qu'il va utiliser pour ce train (lignes 13.2–5). Puis, p_m ajoute `wagonAEnvoyer` à `wagonsNonStables[`i`][`k'`]` au train qu'il envoie (lignes 13.26 et 12.8). Ainsi, `wagonAEnvoyer`, donc m, est reçu par l'ensemble des participants au protocole au bout d'un tour de circuit : chaque participant le stocke dans `wagonsNonStables[`i`][`k'`]`

3. p' peut avoir détecté des départs de processus sans pour autant signaler ces départs dans le circuit de train, car le train en cours de traitement n'a pas pour identifiant 0 (cf. lignes 13.13–14).

4. Nous aurions pu aussi décider que p' retransmette ces wagons aux autres processus. Ainsi, tous les processus corrects auraient uot-livré ces derniers messages de p'_{p_i} envoyés avant le départ des p_{p_i}. Ces wagons auraient été ensuite purgés par p' lors du passage suivant du train. Mais, cela aurait obligé à ce que chaque processus correct fasse une vérification supplémentaire (un wagon ne devant alors être purgé que si son processus émetteur n'appartenait pas au circuit des trains et si ce wagon n'appartenait pas à `dte[`i`].wagons`). Cela aurait entraîné une baisse de performance du protocole, pour un gain qui ne nous semblait pas intéressant.

(ligne 13.18). Au tour de train suivant, chaque participant transfère ce wagon et donc le message m qu'il contient dans `wagonsAUOTLivrer` en vue de l'uot-livrer (ligne 13.6). Considérons maintenant un processus $p_{m'}$ ($p_{m'} \neq p_m$) qui souhaite uot-diffuser un message m'. Quatre cas sont possibles : 1) m' est acheminé par un train antérieur à $t_{\{i,.,k\}}$; 2) m' est acheminé par le train $t_{\{i,.,k\}}$ et ce train n'est pas encore passé par p_m ; 3) m' est acheminé par le train $t_{\{i,.,k'\}}$ et ce train est déjà passé par p_m) ; 4) m' est acheminé par un train postérieur à $t_{\{i,.,k'\}}$. Dans le premier cas, vu qu'aucun train ne double un autre train (cf. lemme 9), nous avons la garantie que m' est uot-livré avant m sur l'ensemble des processus. Dans le deuxième cas, il y a deux sous-cas : a) $k' = k$ (ligne 13.2), b) $k' = (k+1) \bmod \texttt{NR}$ (ligne 13.4). Dans le premier sous-cas, au niveau de $p_{m'}$ et des successeurs de $p_{m'}$ jusqu'au prédécesseur de p_m, le wagon contenant m' est ajouté à `wagonsNonStables[`i`][`k`]`. À partir de p_m jusqu'au prédécesseur de $p_{m'}$, le wagon contenant m' est ajouté à `wagonsNonStables[`i`][`k`]`, suivi du wagon contenant m, car p_m a stocké ce wagon dans $t_{\{i,.,k\}}$ derrière le wagon contenant m'. À partir de $p_{m'}$ jusqu'au prédécesseur de p_m, le wagon contenant m est ajouté à `wagonsNonStables[`i`][`k`]`. Par conséquent, sur chaque processus, le wagon contenant m' apparaît avant le wagon contenant m dans `wagonsNonStables[`i`][`k`]` : m' est uot-livré avant m sur l'ensemble des processus. Dans le deuxième sous-cas, m est stocké dans un wagon qui a un numéro de tour ultérieur à celui de m' : m' est uot-livré avant m sur l'ensemble des processus. Pour le troisième cas, en reprenant le raisonnement utilisé pour le premier sous-cas, nous concluons que m est uot-livré avant m' sur l'ensemble des processus. Pour le quatrième cas, vu qu'aucun train ne double un autre train (cf. lemme 9), nous avons la garantie que m est uot-livré avant m' sur l'ensemble des processus. Dans tous les cas, un message donné est uot-livré avant un autre message donné sur l'ensemble des processus. Cette propriété est contradictoire avec la supposition que p uot-livre m' après avoir uot-livré m alors que p' uot-livre m après avoir uot-livré m'.

Considérons la possibilité que p n'uot-livre jamais m'. Comme nous supposons qu'il existe un processus p' qui uot-livre m' avant m, p' uot-livre m'. Donc, d'après le lemme 11 (accord uniforme), si p est un processus correct, il doit aussi uot-livrer m'. Nous aboutissons aussi à une contradiction. □

Théorème 5. *Le protocole des trains est un protocole d'uot-diffusion.*

Démonstration. Nous avons montré que le protocole des trains vérifie les lemmes 10, 11, 12 et 13. Donc, il vérifie les propriétés de validité, accord uniforme, intégrité uniforme et ordre total. Par conséquent, le protocole des trains est un protocole d'uot-diffusion. □

Lemme 14 (Ordre FIFO). *Si un processus correct diffuse un message m avant de diffuser un message m', alors aucun processus correct ne livre m' avant d'avoir au préalable livrer m.*

Démonstration. Soit un processus p qui uot-diffuse un message m avant d'uot-diffuser un message m'.

Supposons que m et m' sont transportés dans le même wagon. m apparaît dans le wagon avant m', puisque la primitive d'uot-diffusion a été d'abord invoquée avec m, puis avec m'. De ce fait, quand chaque processus correct uot-livre le contenu de ce wagon, il uot-livre d'abord m, puis m'.

Supposons que m et m' sont transportés dans deux wagons différents. Cela signifie que m est acheminé par un train et que m' est acheminé par un train ultérieur. Donc, sur chaque processus, m est transféré dans la variable `wagonAUOTLIvrer` avant m' : chaque processus uot-livre d'abord m, puis m'. □

Corollaire 1 (Ordre causal)**.** *Le protocole des trains est un protocole à diffusion causale.*

Démonstration. Le protocole des trains est un protocole d'uot-diffusion qui respecte l'ordre FIFO (cf. lemme 14). De ce fait, il respecte aussi l'ordre causal [Toinard *et al.*, 1999]. □

Dans cette section, nous avons démontré que le protocole des trains est un protocole d'uot-diffusion qui respecte également l'ordre FIFO et l'ordre causal. Étudions maintenant ses performances d'un point de vue théorique.

5.2 Performances du protocole des trains

Dans cette section, nous analysons les performances du protocole des trains d'un point de vue théorique. Pour ce faire, nous utilisons tout d'abord les métriques de la littérature, c'est-à-dire la latence (cf. section 5.2.1), le débit générique (cf. section 5.2.2) et le débit spécifique aux protocoles d'uot-diffusion (cf. section 5.2.3). Nous constatons que les deux métriques de débit donnent des résultats aberrants. En effet, le débit générique du protocole des trains est indépendant du nombre de participants au protocole, ce qui est intuitivement faux. De plus, pour le protocole des trains, le débit spécifique aux protocoles d'uot-diffusion est strictement supérieur au maximum théorique du débit de diffusions, ce qui est contradictoire. Ces résultats aberrants sont dus au fait que les métriques de débit de la littérature comptabilisent les messages qui circulent physiquement sur le réseau. Appelons-les *messages physiques*. Chacun de ces messages physiques peut contenir plusieurs messages. Appelons-les *messages applicatifs*. Le protocole des trains est conçu pour regrouper plusieurs messages applicatifs dans un même message physique. De ce fait, les métriques de débit de la littérature ne peuvent pas appréhender correctement le protocole des trains : elles sont inadaptées pour estimer ses performances en termes de débit. Aussi, nous définissons la métrique *rendement en termes de débit* : elle nous permet d'évaluer finement les capacités du protocole des trains en termes de débit (cf. section 5.2.4).

5.2.1 Latence

La section 3.3.1 définit la latence comme le nombre d'unités de temps qui séparent le début et la fin du protocole pour chaque uot-diffusion. Autrement dit, la latence correspond au nombre d'unités de temps écoulées entre le moment où un processus uot-diffuse un message et le moment où ce message est uot-livré à tous les participants au protocole.

Lemme 15. *Soit le modèle de performances de la section 3.3.1 selon lequel il faut :*
- *λ unité(s) de temps CPU pour qu'un message soit traité par le processus qui souhaite émettre le message ($\lambda \in \mathbb{R}^{*+}$),*
- *1 unité de temps pour que le message transite sur le réseau (l'accès au réseau étant modélisé selon une politique de tourniquet)*

– *λ unité(s) de temps CPU pour que le message soit traité par un processus qui reçoit le message.*

Soient n processus participant au protocole des trains. Nous supposons que n trains circulent en parallèle. Alors : $Latence(n, \lambda) = (2n - 1)(2\lambda + 1)$.

Démonstration. Soit un processus p qui décide d'uot-diffuser un message m. p stocke m dans la variable `wagonAEnvoyer`. Comme n trains circulent en parallèle sur le circuit, nous avons la garantie qu'un train est en cours de traitement au niveau du prédécesseur de p. Ce prédécesseur envoie ce train à p. De ce fait, sans attente, un train arrive au niveau de p : ce train emporte le wagon contenant m. Considérons le prédécesseur p_p de p. Comme p_p est forcément le dernier processus à uot-livrer m, $Latence(n, \lambda)$ est le temps écoulé entre l'envoi par p et l'uot-livraison par p_p. p_p doit attendre que le train contenant m lui parvienne, puis que ce train fasse un tour de circuit avant que p_p puisse uot-livrer m. p_p doit donc attendre : $[\lambda + 1 + (n - 2)(\lambda + \lambda + 1) + \lambda] + [\lambda + 1 + (n - 1)(\lambda + \lambda + 1) + \lambda] = (2n - 1)(2\lambda + 1)$. □

Notez que [Urbán *et al.*, 2000a] distingue l'utilisation d'un réseau n'autorisant que les communications point-à-point et celle d'un réseau autorisant aussi le multicast. Pour le protocole des trains qui n'utilise que des communications point-à-point, cette distinction n'a pas lieu d'être.

5.2.2 Métrique de débit générique

La section 3.3.1 définit, pour tout algorithme réparti, une métrique de débit générique avec $d\acute{e}bit_{g\acute{e}n} = \frac{1}{\max_{r \in \mathscr{R}_n} T_r(n,\lambda)}$ où $\mathscr{R}_n$ désigne l'ensemble des ressources du système (c'est-à-dire CPU_1,...,CPU_n et le réseau) et $T_r(n, \lambda)$ désigne la durée totale de l'utilisation de la ressource r pendant l'exécution de l'algorithme.

Lemme 16. *Soit le modèle de performances utilisé dans le lemme 15. Nous ajoutons à ce modèle la contrainte réseau qu'un seul message peut circuler entre 2 machines à un instant donné. Alors : $d\acute{e}bit_{g\acute{e}n}(n, \lambda) = \frac{1}{\max(1,2\lambda)}$.*

Démonstration. Cette démonstration s'inspire de la méthodologie décrite en annexe B de [Urbán *et al.*, 2000a]. Nous supposons que le système est à pleine charge et en régime stationnaire, c'est-à-dire que tous les processus effectuent suffisamment d'uot-diffusions pour que le système ait tout le temps des messages à envoyer, sans pour autant que la liste des uot-diffusions en attente de passage d'un train augmente indéfiniment. Pour pouvoir calculer $d\acute{e}bit_{g\acute{e}n}$, par hypothèse, nous considérons qu'à chaque période, seul un message peut circuler sur le réseau. Dans le cas du protocole des trains, cela revient à travailler avec la version monotrain : à tout instant, seul un train circule sur le circuit des trains.

En s'inspirant de la méthode de calcul de $d\acute{e}bit_{g\acute{e}n}$ présentée dans [Urbán *et al.*, 2000a], il serait tentant d'écrire qu'une uot-diffusion m nécessite que le train transite n fois pour que m soit transmis à l'ensemble des processus et à nouveau n transitions de train pour que chaque processus considère que m peut être uot-livré, soit $2 \cdot n$ messages sur le réseau. Succomber à cette tentation ne permettrait pas de tenir compte des spécificités du protocole des trains. En effet, quand un processus p reçoit un train $t_{\{i,.,k\}}, i \in [\![0, \mathtt{ntr}[\![, k \in [\![0, \mathtt{NR}[\![$, p uot-livre les n messages qui avaient été stockés dans la variable `wagonsNonStables[`i`][(`$k-$

2+NR) mod NR] lors des deux passages précédents du train d'identifiant i. De ce fait, quand le train fait le tour du circuit (ce qui représente n messages sur le réseau), chacun des n processus uot-livre n messages : une uot-diffusion requiert en moyenne $\frac{n}{n} = 1$ message sur le réseau.

Par ailleurs, comme nous considérons la version monotrain du protocole, à un instant donné, soit un processus consacre de la CPU pour traiter la réception du train (λ unité(s) de temps) et l'émission du train (λ unité(s) de temps), soit ce processus ne consacre aucune CPU au protocole du train car le train est traité par un autre processus. Un processus consacre donc au maximum 2λ unités de temps CPU.

Nous en concluons : $d\acute{e}bit_{g\acute{e}n}(n, \lambda) = \frac{1}{\max(1,2\lambda)}$. □

Remarques :

1. Pour le lemme 16, nous avons ajouté une contrainte conforme aux hypothèses de la section 3.3.1, mais plus restrictive que les contraintes du modèle réseau envisagé en section 4.1.2 pour notre protocole. L'intérêt de prendre cette contrainte plus forte est de pouvoir comparer notre protocole aux protocoles évalués dans [Urbán *et al.*, 2000b].
2. Nous constatons que $d\acute{e}bit_{g\acute{e}n}(n, \lambda)$ est indépendant de n, ce qui n'est le cas d'aucun des protocoles étudiés dans [Urbán *et al.*, 2000b]. Cette indépendance s'explique ainsi : de manière naturelle, le protocole des trains agrège les messages. Il suit donc intrinsèquement la meilleure optimisation possible pour un protocole d'uot-diffusion, c'est-à-dire regrouper au maximum les messages à uot-diffuser dans un même paquet [Friedman et van Renesse, 1997]. En effet, ce regroupement permet de rendre le nombre d'interruptions liées à la réception de messages indépendant de n et donc de limiter la charge CPU liée au traitement des différents messages [Smed *et al.*, 2001, Bauer *et al.*, 2002, Herlihy et Mohan, 2003]. Si elle s'explique, cette indépendance de $d\acute{e}bit_{g\acute{e}n}(n, \lambda)$ par rapport à n n'en reste pas moins insatisfaisante. En effet, elle donne à penser que le protocole des trains n'a aucune limite en termes de débit, ce qui est intuitivement faux. Le problème vient du fait que le protocole des trains travaille avec des messages applicatifs. De ce fait, cette métrique appréhende incorrectement son fonctionnement et donne des résultats aberrants.

La section suivante étudie l'autre métrique de débit de la littérature et analyse si cette métrique est confrontée au même problème d'aberration.

5.2.3 Métrique de débit spécifique aux protocoles d'uot-diffusion

La section 3.3.1 définit une métrique de débit spécifique aux protocoles d'uot-diffusion, $d\acute{e}bit_{UOT}$. Cette métrique est le nombre moyen d'uot-diffusion(s) réalisée(s) par intervalle de temps, sachant qu'une uot-diffusion d'un message m est *réalisée* quand tous les processus ont uot-livré m.

Lemme 17. *Pour le protocole des trains,* $d\acute{e}bit_{UOT_{trains}} = n$

Démonstration. Nous considérons le système défini à la section 4.1. De ce fait, n messages peuvent circuler en parallèle à partir du moment où deux messages n'ont pas le même

émetteur, ni le même récepteur.

Considérons la version multitrain du protocole des trains et supposons que n trains circulent en parallèle sur le circuit.

Supposons qu'il y a plusieurs trains au niveau d'un processus p. Comme p ne peut envoyer qu'un seul train par intervalle de temps, vu qu'il y a autant de trains que de processus, au bout de n intervalles de temps, nous obtenons 1 train par processus à chaque intervalle de temps.

À partir de cet instant, à chaque intervalle de temps, chaque processus p reçoit un train ayant comme identifiant $i \in [\![0, \texttt{ntr}[\![$. Notons $k' \in [\![0, \texttt{NR}[\![$ le numéro de tour qu'utilise alors p. Au bout de n intervalles de temps, le train d'identifiant i a fait un tour de circuit. Donc, p a stocké n wagons dans `wagonsNonStables[`i`][`k'`]`. Au bout de n nouveaux intervalles de temps, le train d'identifiant i a fait un nouveau tour de circuit. Donc, p transfère les n wagons de `wagonsNonStables[`i`][`k'`]` dans `wagonsAUOTLivrer`.

Par conséquent, lors du j^e ($j \geqslant 2$) passage d'un train au niveau d'un processus, seules les uot-diffusions des séries 1 à $j-2$ sont réalisées. Si nous supposons qu'il n'y a qu'un seul message par wagon : $\textit{débit}_{UOT_{trains}} = \frac{(j-2)\cdot n}{j}$

De ce fait : $\lim_{j\to+\infty} \textit{debit}_{UOT_{trains}} = n$ □

Le lemme 17 est contradictoire avec le lemme 1 (cf. section 3.3.1). En effet, selon ce dernier, $\max \textit{débit}_{UOT}$ vaut $n/(n-1)$ si chacun des n processus participant au protocole uot-diffuse des messages et 1 sinon. Or, d'après le lemme 17, dès que $n > 2$, $\textit{débit}_{UOT_{trains}} > \max \textit{débit}_{UOT}$. Cette contradiction provient du fait que, pour calculer μ_{max} dans le théorème 1, [Guerraoui *et al.*, 2010] considère les messages physiques, sans tenir compte des multiples messages applicatifs que ces messages physiques peuvent contenir. Or, ce désintérêt vis-à-vis des messages applicatifs entraîne déjà une contradiction avec le lemme 1 dans le cas du protocole LCR. En effet, pour démontrer que LCR atteint μ_{max}, [Guerraoui *et al.*, 2010] explique que LCR diffuse des messages de confirmation en les *piggybackant* sur des messages d'uot-diffusion pour économiser des messages. Donc, dans LCR, chaque message physique transporte un message applicatif d'uot-diffusion et également un message applicatif correspondant à l'acquittement d'une uot-diffusion qui a déjà été reçue par tous les processus. Nous constatons qu'avec les $n/(n-1)$ diffusions de messages physiques par période (la borne supérieure μ_{max} du théorème 1 est donc respectée), LCR utilise en fait $2 \times n/(n-1)$ diffusions de messages applicatifs par tour, soit $2 \times \mu_{max} > \mu_{max}$. Nous observons le même phénomène avec le protocole des trains. Chaque envoi de message physique transporte plusieurs messages applicatifs, chacun de ces derniers correspondant à une uot-diffusion. De ce fait, il est légitime de constater que $\textit{débit}_{UOT_{trains}} > \mu_{max}$. Ainsi, nous obtenons une explication de la contradiction apportée par $\textit{débit}_{UOT_{trains}}$.

Mais, en fait, la valeur de $\textit{débit}_{UOT_{trains}}$ entraîne également une aberration. En effet, dans la démonstration du lemme 17, nous avons supposé que chaque wagon livré contient un seul message. Dans la pratique, rien n'empêche d'avoir plus d'un message par wagon livré. Supposons que chaque wagon contient $m \geqslant 1$ messages. Alors $\textit{débit}_{UOT_{trains}} = n \times m$. Donc, plus nous mettons de messages dans chaque wagon, meilleur est le débit ! Par conséquent, $\textit{débit}_{UOT_{trains}}$ ne tient aucun compte des limites physiques du routeur via lequel communiquent tous les processus. Nous en concluons que, parce que $\textit{débit}_{UOT}$ ne considère

que les messages physiques et n'appréhende pas les messages applicatifs, il ne capture pas correctement la réalité d'un système.

C'est pourquoi la section suivante propose une métrique qui permet de prendre en compte les messages applicatifs et évalue l'algorithme des trains selon cette métrique.

5.2.4 Rendement en termes de débit

Cette section a pour objectif de pallier les limites de la métrique $débit_{UOT}$ de la section précédente.

Tout d'abord, nous définissons la métrique *rendement en termes de débit* qui répond à ce problème, puis les métriques résultantes : le *coût* (qui caractérise la part de réseau utilisée par le protocole pour obtenir les propriétés d'uot-diffusion) et *le débit maximum d'uot-livraison* en Mbps [5] (cf. section 5.2.4.1). Ensuite, nous estimons le protocole des trains selon ces métriques (cf. section 5.2.4.2). La section 5.2.4.3 propose des applications numériques. Puis, nous estimons le protocole LCR selon ces métriques, car LCR est le meilleur protocole d'un point de vue débit selon [Guerraoui *et al.*, 2010] (cf. section 5.2.4.4). Ensuite, nous montrons que le protocole des trains est théoriquement meilleur que le protocole LCR (cf. section 5.2.4.5). Enfin, nous concluons avec la section 5.2.4.6.

5.2.4.1 Introduction

Définition 5. *La métrique rendement en termes de débit (notée e comme la lettre « E » dans* throughput Efficiency*) est définie par :*

$$e = \frac{\textit{nombre d'octets effectivement uot-livrés par période}}{\textit{nombre maximum d'octets diffusables à tous les processus par période}}$$

Par définition, e mesure l'efficacité avec laquelle le protocole utilise la capacité réseau disponible pour faire des diffusions sur le réseau utilisés. Plus e est proche de 1, plus le protocole est efficace.

Définition 6. *Dans la littérature, un protocole est dit « symétrique » lorsque tous les processus exécutent le même algorithme [Attiya et Welch, 2004]. Dans notre étude, pour qu'un protocole soit dit symétrique, nous ajoutons la contrainte que les différents processus s'échangent des messages physiques de même type et de même taille, mais de contenus différents.*

Lemme 18. *Dans un réseau où toutes les communications se font en point-à-point, quand le protocole est symétrique (cf. défintion 6) :*

$$e = \frac{\textit{nombre d'octets applicatifs dans chaque message utiles aux uot-livraisons}}{\textit{nombre d'octets total dans le message}}$$

Démonstration. Du fait des hypothèses, le trafic réseau est issu seulement de messages point-à-point. De plus, comme le protocole est symétrique, chaque message contribue de la même manière à e. Le rendement pour un message correspond donc bien au rendement pour l'ensemble des messages. □

5. Mbps est le symbole du mégabit par seconde, un mégabit correspondant à un million de bits.

Définition 7. *Nous définissons la métrique coût (notée c) comme :* $c = 1 - e$

c matérialise la quantité de réseau utilisée par le protocole pour obtenir la fonction désirée.

Lemme 19. *Notons* $H_{point-à-point}$ *le débit maximum (en Mbps) d'une connexion point-à-point. Notons* $H_{uot-diffusion}$ *le débit maximum (en Mbps) d'uot-livraison d'un protocole d'uot-diffusion dont le rendement est* $e_{uot-diffusion}$*. Alors :*

$$H_{uot-diffusion} = e_{uot-diffusion} \cdot H_{point-à-point} \cdot n/(n-1)$$

Démonstration. D'après le théorème 1 page 42, il peut y avoir au maximum $n/(n-1)$ diffusions de messages physiques. Donc, $H_{point-à-point} \cdot n/(n-1)$ est le nombre maximum d'octets qui peuvent être diffusés à tous les processus chaque seconde. En appliquant la définition 5, nous concluons : $H_{uot-diffusion} = e_{uot-diffusion} \cdot H_{point-à-point} \cdot n/(n-1)$ □

5.2.4.2 Application au protocole des trains

e et c étant définis, nous évaluons le protocole des trains selon e (et c), grâce au lemme 18. Pour ce faire, nous estimons le nombre d'octets dans chaque train.

Nous supposons que nous n'avons pas besoin de faire du *bit padding*, c'est-à-dire de l'ajout d'octets pour que certains octets soient alignés sur des frontières d'octets. En effet, nous supposons que les processus participant au protocole s'exécutent sur des processeurs insensibles à l'alignement des octets (par exemple, des processeurs Intel récents [Sim12]) : un accès à un entier sur une frontière d'octets ne cause pas d'exception dans le processeur (alors que c'est le cas de certains processeurs ARM) et ne réduit pas les performances d'accès à ces données (en nécessitant, par exemple, deux accès au bus de données comme dans le cas des processeurs 68xxx) [6].

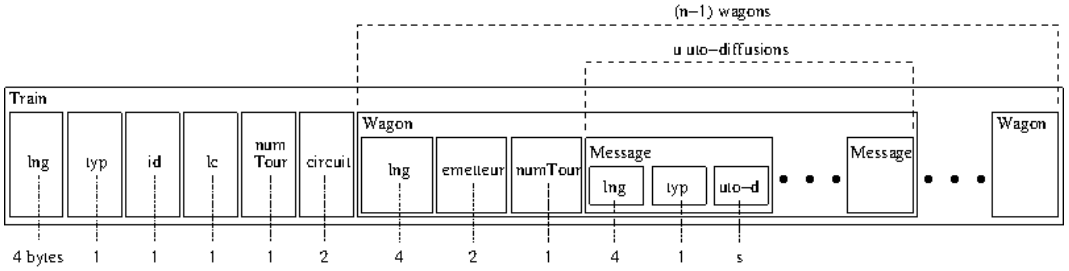
Pour être stocké dans un wagon w, un message m est précédé d'un entier `lng` (4 octets) permettant de retrouver la fin du message dans le wagon et d'un octet `typ` indiquant si le message est une uot-diffusion venant de l'application ou bien une information concernant une arrivée (respectivement un départ) de processus sur le (respectivement du) circuit des trains (cf. figure 5.1).

Un wagon commence par un entier `lng` (4 octets) permettant de retrouver la fin d'un wagon dans un train, suivi par un champ `emetteur` (que nous réduisons à 2 octets, cf. section 6.1.2), puis le numéro de tour auquel appartient ce wagon (1 octet). Suivent les messages stockés dans ce wagon.

Un train contient une longueur de train `lng` (4 octets), puis un octet indiquant que c'est un train (cela permet de distinguer les trains des autres messages protocolaires, comme `NOUVEAU_SUCC` par exemple) suivi par l'identifiant de train (1 octet), l'horloge logique (que nous réduisons à 1 octet, cf. section 6.1.1), 1 octet pour le numéro de tour, le circuit des trains (que nous réduisons à 2 octets, cf. section 6.1.2), et la liste des wagons. Supposons que chaque message uot-diffusé par l'application fait en moyenne s octets, que chaque wagon

6. Dans le cas des processeurs Intel, nous supposons que le codage du protocole des trains n'utilise pas certaines instructions `MMX` qui requièrent un alignement des données sur une frontière de 16 octets.

FIGURE 5.1 – Structure d'un train dans le protocole des trains



contient en moyenne u uot-diffusions, que chacun des n processus fait des uot-diffusions, et que n trains circulent en parallèle sur le circuit.

Lemme 20. $e_{trains} = \frac{(n-1)\cdot u\cdot s}{10+(n-1)\cdot[7+u\cdot(5+s)]}$

Démonstration. Le lemme 18 est applicable puisque le protocole des trains est un protocole symétrique et basé sur des communications point-à-point.

Si les n processus font des uot-diffusions, chaque train contient $n-1$ wagons puisqu'un processus n'envoie pas de train contenant le wagon de son successeur sur le circuit. De ce fait, la taille (en octets) de chaque train est : $4+1+1+1+1+2+(n-1)\cdot[4+2+1+u\cdot(4+1+s)] = 10+(n-1)\cdot[7+u\cdot(5+s)]$

Donc :

$$e_{trains} = \frac{\text{nombre d'octets applicatifs dans chaque message utiles aux uot-livraisons}}{\text{nombre d'octets total dans le message}} = \frac{(n-1)\cdot u\cdot s}{10+(n-1)\cdot[7+u\cdot(5+s)]}$$

□

Nous constatons que e_{trains} est une fonction de n, u et s. Les lemmes suivants nous permettent de démontrer qu'il n'existe pas de valeur optimale pour chacun de ces 3 paramètres.

Lemme 21. *Soit $u \to e_{trains}(u)$ la fonction qui associe le rendement e_{trains} du protocole des trains au nombre moyen u d'uot-diffusion(s) par wagons ($u \in [\![1, +\infty[\![$).*

Si au moins deux processus participent au protocole, alors la fonction $u \to e_{trains}(u)$ est strictement croissante.

Démonstration. Reprenons l'expression de e_{trains} du lemme 20 et dérivons la par rapport

à u :

$$\begin{aligned}\frac{\partial e_{trains}(u)}{\partial u} &= (n-1)\cdot s\cdot \frac{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\}-\{(n-1)\cdot(5+s)\cdot u\Big)}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}\\ &= (n-1)\cdot s\cdot \frac{10+(n-1)\cdot 7}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}\\ &= (n-1)\cdot s\cdot \frac{3+7n}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}\end{aligned}$$

Par hypothèse $n \geqslant 2$. Donc : $\forall u \in [\![1,+\infty[\![, \frac{\partial e_{trains}(u)}{\partial u} > 0$. Nous concluons que la fonction $u \to e_{trains}(u)$ est strictement croissante. □

Lemme 22. *Soit $n \to e_{trains}(n)$ la fonction qui associe le rendement e_{trains} du protocole des trains au nombre n de processus participant au protocole des trains ($n \in [\![2,+\infty[\![$).*

La fonction $n \to e_{trains}(n)$ est strictement croissante.

Démonstration. Reprenons l'expression de e_{trains} du lemme 20 et dérivons la par rapport à n :

$$\begin{aligned}\frac{\partial e_{trains}(n)}{\partial n} &= u\cdot s\cdot \frac{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)-\Big((n-1)\cdot[7+u\cdot(5+s)]\Big)}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}\\ &= u\cdot s\cdot \frac{10}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}\end{aligned}$$

Par conséquent : $\forall n \in [\![2,+\infty[\![, \frac{\partial e_{trains}(n)}{\partial n} > 0$. Nous concluons que la fonction $n \to e_{trains}(n)$ est strictement croissante. □

Lemme 23. *Soit $s \to e_{trains}(s)$ la fonction qui associe le rendement e_{trains} du protocole des trains à la taille moyenne s des messages uot-diffusés ($s \in [\![1,+\infty[\![$).*

Si au moins deux processus participent au protocole, alors la fonction $s \to e_{trains}(s)$ est strictement croissante.

Démonstration. Reprenons l'expression de e_{trains} du lemme 20 et dérivons la par rapport

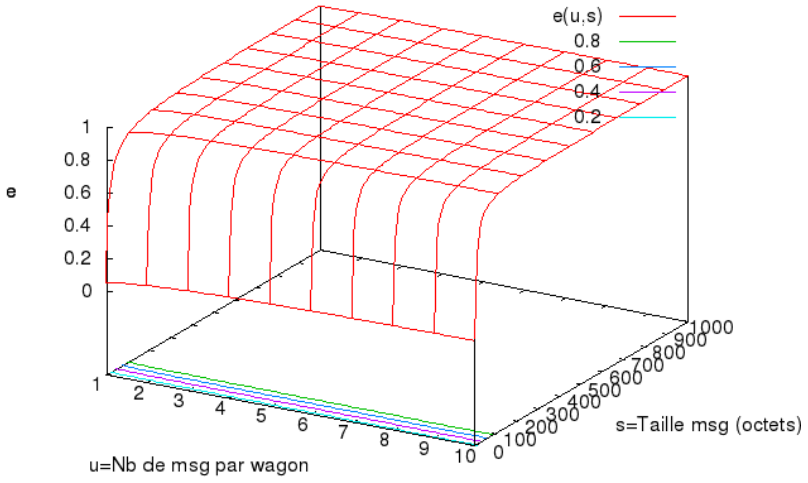
à s :

$$\frac{\partial e_{trains}(s)}{\partial s} = (n-1)\cdot u \cdot \frac{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big) - \Big(s\cdot(n-1)\cdot u\Big)}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}$$
$$= (n-1)\cdot u \cdot \frac{10+7\cdot(n-1)+5\cdot(n-1)\cdot u}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)^2}$$

Par hypothèse $n \geqslant 2$. Donc : $\forall s \in [\![1,+\infty[\![, \frac{\partial e_{trains}(s)}{\partial s} > 0$. Nous concluons que la fonction $s \to e_{trains}(s)$ est strictement croissante. □

Le lemme 21 permet de démontrer que plus un wagon contient de messages, meilleur est le rendement. Par ailleurs, le lemme 22 établit que le rendement augmente avec le nombre de participants au protocole. Enfin, le lemme 23 prouve que plus la taille moyenne des messages uot-diffusés est grande, meilleur est le rendement. Nous en concluons qu'il n'y a pas de valeur théorique optimale pour le nombre de messages dans un wagon, ni pour le nombre de participants, ni pour la longueur moyenne des messages. La figure 5.2 illustre graphiquement ces résultats mathématiques pour $n = 5$ processus.

FIGURE 5.2 – Rendement du protocole des trains pour $n = 5$ processus



Cette absence de valeur théorique optimale montre que notre métrique rendement ne modélise pas correctement les contraintes physiques de notre système. En effet, si nous augmentons de manière inconsidérée la taille des trains pour améliorer le rendement, intuitivement, le réseau et les ordinateurs fonctionneront plus lentement pour écluser la charge

induite par cette taille importante. Alors, mécaniquement, le débit du système diminuera. Notre métrique nécessite donc d'être affinée. C'est une perspective de cette thèse.

Notez que, grâce à l'expression de e_{trains}, nous obtenons les expressions de $c_{trains} = 1 - e_{trains}$ et $H_{trains} = e_{trains} \cdot H_{point-à-point} \cdot n/(n-1)$ via la définition 7 et le lemme 19.

5.2.4.3 Applications numériques

Supposons que $n = 5$ processus, $u = 1$ uot-diffusion par wagon, et $H_{point-à-point} = 93$ Mbps (nous reprenons ici le débit mesuré dans [Guerraoui *et al.*, 2010]). Pour $s = 100$ octets (respectivement 10 kB) :

- $e_{trains}(n = 5, u = 1) = 0,873$ (respectivement $0,999$),
- $c_{trains}(n = 5, u = 1) = 0,127$ (respectivement $0,001$),
- $H_{trains}(n = 5, u = 1) = 102$ Mbps (respectivement 116 Mbps).

Dans le cas où $s = 100$, considérons $u = 100$, c'est-à-dire la valeur de u uot-diffusions par wagon telle que $u \cdot s = 10$ kB. Nous obtenons :

- $e_{trains}(n = 5, u = 100) = 0,952$,
- $c_{trains}(n = 5, u = 100) = 0,048$,
- $H_{trains}(n = 5, u = 100) = 111$ Mbps.

5.2.4.4 Application au protocole LCR

Des échanges de courriels privés avec les auteurs de LCR nous permettent de déterminer la structure d'un message LCR utilisé pour le transport d'une uot-diffusion. Chaque message LCR comprend un type de message (codé sous la forme d'un `enum` C++, soit 4 octets), une adresse d'émetteur (4 octets), un identifiant de message (4 octets), l'émetteur du message acquitté (4 octets) et l'identifiant du message acquitté (4 octets) pour l'acquittement via *piggybacking*, le vecteur d'horloges (n participants $\times$ 4 octets), la taille du message transporté (4 octets) et enfin le message proprement dit (s octets).

Lemme 24. $e_{LCR} = \frac{s}{24+4 \cdot n+s}$

Démonstration. Le lemme 18 est applicable puisque LCR est un protocole symétrique et basé sur des communications point-à-point.

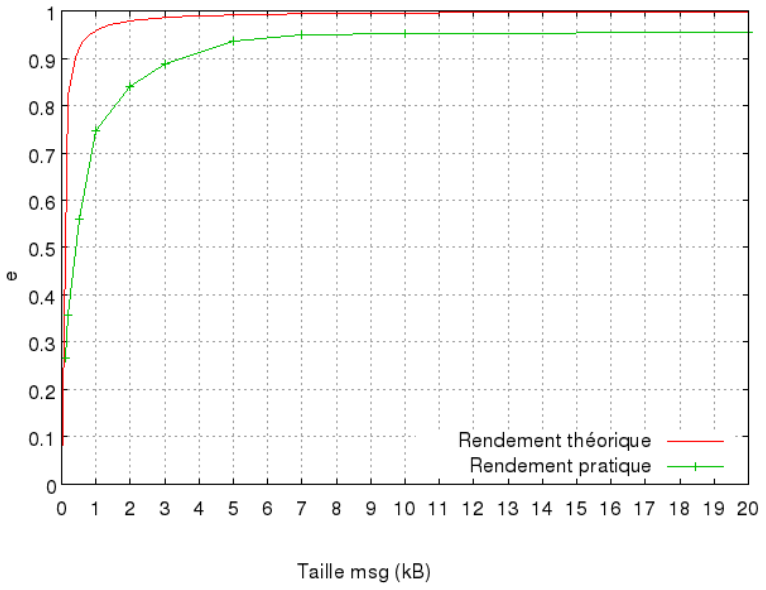
Donc :

$$\begin{aligned} e_{LCR} &= \frac{\text{nombre d'octets applicatifs dans chaque message utiles aux uot-livraisons}}{\text{nombre d'octets total dans le message}} \\ &= \frac{s}{24 + 4 \cdot n + s} \end{aligned}$$

□

La figure 5.3 permet de comparer les rendements théorique et expérimental du protocole LCR. Pour calculer le rendement expérimental, nous avons repris les débits expérimentaux présentés dans la figure 12 de [Guerraoui *et al.*, 2010] et nous les avons rapportés au débit de diffusion maximum en présence de 5 processus, soit 116,25 Mbps.

FIGURE 5.3 – Rendements théorique et expérimental du protocole LCR pour $n = 5$ processus



Nous observons un écart entre la théorie et la pratique. Ainsi, pour $n = 5$ processus et $s = 100$ octets (respectivement 10 kB), $e_{LCR_{théorique}}(n = 5) = 0,694$ (respectivement $0,999$) alors que $e_{LCR_{pratique}}(n = 5) = 0,301$ (respectivement $0,951$), soit un écart entre la théorie et la pratique de $-56,6\%$ (respectivement $-4,7\%$)

Après une analyse personnelle du code actuel LCR, nous pensons que l'essentiel de la perte de performances se situe au niveau de l'envoi et de la réception des messages. En effet, l'envoi (respectivement la réception) d'un message requiert 8 appels système `write` (respectivement `read`). Or, un appel système est toujours coûteux en termes de CPU. Et, au vu du code de LCR, il nous semble tout à fait possible de ne faire qu'un appel système pour envoyer (respectivement recevoir) un message en utilisant, par exemple, les appels système `writev` (respectivement `readv`). Toutefois, nous n'avons pas vérifié expérimentalement le bien-fondé de notre proposition.

Grâce à l'expression de e_{LCR}, nous obtenons aussi les expressions de $c_{LCR} = 1 - e_{LCR}$ et $H_{LCR} = e_{LCR} \cdot H_{point-à-point} \cdot n/(n-1))$ via la définition 7 et le lemme 19.

5.2.4.5 Comparaison entre le protocole des trains et le protocole LCR

Lemme 25. *Le protocole des trains a un meilleur rendement que celui de* LCR.

Démonstration. Comparons e_{trains} et e_{LCR} en déterminant le signe de $e_{trains} - e_{LCR}$.

$$
\begin{aligned}
e_{trains} - e_{LCR} &= \frac{(n-1)\cdot u\cdot s}{10+(n-1)\cdot[7+u\cdot(5+s)]} - \frac{s}{24+4\cdot n+s} \\
&= \frac{(n-1)\cdot u\cdot s\cdot(24+4\cdot n+s) - s\cdot\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)\cdot(24+4\cdot n+s)} \\
&= \frac{(n-1)\cdot u\cdot s\cdot(19+4\cdot n) - 10\cdot s - 7\cdot(n-1)\cdot s}{\Big(10+(n-1)\cdot[7+u\cdot(5+s)]\Big)\cdot(24+4\cdot n+s)}
\end{aligned}
$$

$e_{trains} - e_{LCR}$ est du signe de son numérateur. Notons $f(u,s,n)$, ce numérateur.
$\frac{\partial f(u,s,n)}{\partial u} = (n-1)\cdot s\cdot(19+4\cdot n) > 0$ dès que $n \geqslant 2$. Donc, f est croissante selon u.
$\frac{\partial f(u,s,n)}{\partial s} = (n-1)\cdot(19\cdot -7) + 4\cdot n\cdot(n-1) - 10 > 0$ dès que $n \geqslant 2$ et $s \geqslant 1$. Donc, f est croissante selon s.
$\frac{\partial f(u,s,n)}{\partial n} = 4\cdot(2\cdot n-1) + (19\cdot u-7)\cdot s > 0$ dès que $n \geqslant 2$ et $u \geqslant 1$. Donc, f est croissante selon n.

Or, $f(1,1,2) = 6 > 0$. Par conséquent : $\forall u \geqslant 1, \forall s \geqslant 1, \forall n \geqslant 2, f(u,s,n) > 0$
Nous concluons : $\forall u \geqslant 1, \forall s \geqslant 1, \forall n \geqslant 2, e_{trains} - e_{LCR} > 0$ □

Ce résultat doit toutefois être nuancé. En effet, nous pouvons effectuer des optimisations sur la taille des champs des messages LCR. Ainsi, le type de message (respectivement l'émetteur d'un message) pourrait être réduit à 1 octet (respectivement 2 octets) comme dans le protocole des trains. Par ailleurs, chaque élément du vecteur d'horloges pourrait être stocké sur 2 (voire 1) octet(s) à la place des 4 octets actuels. Toutefois, dans ce cas, il faudrait probablement s'appuyer sur la proposition de [Baldoni, 1998] pour gérer les dépassements de capacité de chaque élément. Une perspective de cette thèse est de vérifier le bien-fondé d'une telle optimisation, puis de recalculer $e_{trains} - e_{LCR}$ en tenant compte de l'ensemble de ces optimisations.

5.2.4.6 Conclusion

Cette section définit la métrique rendement en termes de débit e qui pallie les limites de $débit_{UOT}$, la métrique de débit spécifique aux protocoles d'uot-diffusion. Cette section définit également la métrique coût et la métrique résultante qui est le débit maximum d'uot-livraison, mesuré en mégabit par seconde.

Une première exigence de ces métriques est qu'elles requièrent de connaître précisément la structure des messages échangés par le protocole. Une seconde exigence est la rigueur dans le calcul du numérateur de e évoqué dans le lemme 18. En effet, ce numérateur est le nombre d'octets applicatifs dans chaque message *utiles aux uot-livraisons*. Or, cette définition n'offre aucun garde-fou contre la prise en compte d'octets inutiles. Par exemple, dans le cas du protocole des trains, imaginons que nous n'ayons pas perçu qu'un processus n'a pas besoin d'envoyer le wagon de son successeur à son successeur. Dans ce cas, en reprenant

la démonstration du lemme 20, nous obtenons : $e_{trainsInexact} = \frac{n \cdot u \cdot s}{10+n \cdot [7+u \cdot (5+s)]} > e_{trains}$. Par conséquent, cette métrique ne nous permet pas de détecter que nous transportons inutilement des informations dans le protocole.

Mais, la liste des avantages de ces métriques contrebalancent largement ces exigences :

- aucun autre détail de codage n'est requis ;
- nous pouvons estimer au plus tôt (et sans codage) le débit maximal que pourra atteindre un protocole ;
- ces métriques permettent de comparer précisément des protocoles en termes de débit. Ainsi, nous avons pu démontrer que le protocole des trains est plus performant en termes de débit que le protocole LCR (considéré comme l'algorithme optimal en termes de débit [Guerraoui *et al.*, 2010]) ;
- ces métriques permettent de calculer un débit maximum théorique, et donc la qualité de l'implantation, une fois que cette dernière est disponible. Ainsi, l'écart entre la théorie et la pratique pour LCR nous a amené à analyser le code de LCR et à proposer une piste d'amélioration ;
- ces métriques permettent d'évaluer les gains de performance envisageables en réalisant des optimisations sur la structure des messages utilisés.

Ces métriques laissent des questions ouvertes (qui sont autant de perspectives) : 1) quelle(s) méthode(s) mettre en place pour disposer d'un garde-fou contre la prise en compte d'octets inutiles évoquée ci-dessus ? 2) Pouvons-nous évaluer d'autres protocoles à l'aide de ces métriques ? 3) Pouvons-nous les coupler avec l'estimation de l'utilisation de la CPU comme le propose [Défago *et al.*, 2003] ? 4) Comment se positionneraient le protocole LCR par rapport au protocole des trains si nous effectuions d'importantes optimisations sur la taille des messages de LCR ? 5) Notre métrique de rendement donne à penser que, pour améliorer le rendement du protocole des trains, il suffit de charger infiniment chacun de ses trains ; comment affiner notre métrique pour qu'elle prenne mieux en compte les contraintes physiques du système ? 6) Quel est l'écart minimal entre l'estimation théorique du rendement et son estimation sur la base d'une implantation et d'expériences de performances ?

5.3 Conclusion

Dans ce chapitre, nous avons tout d'abord démontré que le protocole des trains est un protocole d'uot-diffusion qui respecte également l'ordre FIFO et l'ordre causal.

Puis, nous avons estimé ses performances de manière théorique. La latence est une fonction linéaire du nombre de participants au protocole. Le débit générique ($d\acute{e}bit_{gén}$) est indépendant du nombre de participants, car le protocole des trains agrège naturellement les messages au sein d'un même train. C'est pour cette même raison que son débit spécifique aux protocoles d'uot-diffusion ($d\acute{e}bit_{UOT}$) est égal à n. Cela nous a amené à définir des métrique plus précises pour comparer des algorithmes qui excellent en termes de débit : le rendement en termes de débit, le coût (qui caractérise la part de réseau utilisée par le protocole pour obtenir les propriétés d'uot-diffusion), et le débit maximum d'uot-livraison en mégabit par seconde. Grâce à ces métriques, nous avons montré que protocole des trains est théoriquement meilleur en termes de débit que LCR, le plus performant des protocoles

existants en termes de débit.

Nous nous proposons maintenant de confronter ces résultats théoriques à la pratique. Dans le chapitre suivant, nous nous intéressons aux conséquences liées à la mise en œuvre du protocole des trains.

Chapitre 6

Utilisation du protocole des trains pour une mémoire répartie partagée

Dans ce chapitre, nous nous intéressons à l'utilisation du protocole des trains pour l'amélioration de la tolérance aux fautes d'une application industrielle, en l'occurrence le système de contrôle-commande et de supervision *P3200* présenté dans le chapitre 2. La section 6.1 analyse tout d'abord la mise en œuvre du protocole lui-même. Puis, la section 6.2 définit la notion de groupe applicatif offerte à la couche applicative. Ensuite, la section 6.3 présente la mémoire répartie partagée que nous avons mise en œuvre dans le *P3200*. Cette mémoire rend l'utilisation de notre protocole d'uot-diffusion transparente pour la majorité du code du *P3200*. Elle est généralisable à tout logiciel qui a besoin d'améliorer sa tolérance aux fautes. Enfin, la section 6.4 conclut.

6.1 Mise en œuvre du protocole des trains

Dans cette section, nous nous intéressons à la mise en œuvre du protocole des trains et ses conséquences sur les algorithmes présentés au chapitre 4.

La section 6.1.1 étudie la gestion du dépassement de capacité de l'horloge logique du train. Puis, la section 6.1.2 s'intéresse au stockage du circuit des trains dans l'entête de chaque train. Ensuite, la section 6.1.3 introduit la régulation du débit d'émission. Nous étudions alors la recherche du futur successeur sur le circuit des trains (cf. section 6.1.4) et présentons l'expérience en cours pour évaluer les performances d'un point de vue pratique (cf. section 6.1.5). Enfin, nous décrivons les améliorations envisageables (cf. section 6.1.6).

6.1.1 Dépassement de capacité de l'horloge logique du train

Dès qu'un algorithme utilise un compteur, il est nécessaire de se poser la question du dépassement de la capacité de ce compteur sur un système industriel. Ainsi, [Attiya et Welch, 2004] remarque que le premier algorithme qu'il propose pour l'exclusion mutuelle répartie doit être revu pour tenir compte de cette contrainte physique. De même, [Baldoni et Raynal, 2002] constate que les systèmes à base de vecteurs d'horloges sont également

confrontés à ce problème pratique. En effet, les horloges des vecteurs peuvent également dépasser la capacité de la structure de données qui leur est attribuée. Dans cette section, nous proposons une solution pour gérer le dépassement de capacité de l'horloge logique du train (alors que le chapitre 4 supposait que cette horloge logique pouvait grandir indéfiniment).

La section 6.1.1.1 étudie tout d'abord les méthodes proposées dans la littérature pour prendre en compte ce problème : 1) ignorer le problème, 2) prévoir une structure de données assez grande pour ne pas observer le problème pendant la durée de vie du système, 3) utiliser une structure de données illimitée en capacité, 4) déclencher une remise à zéro de la variable quand elle atteint un certain seuil, 5) faire en sorte que l'algorithme puisse s'exécuter correctement avec des dépassements de capacité. Cette dernière méthode a été mise en œuvre dans le cadre du codage initial du protocole des trains. Nous montrons que cette mise en œuvre comporte une faute qui peut donner lieu à une erreur dans le système avec une probabilité estimée à $2 \cdot 10^{-11}$. Puis, la section 6.1.1.2 propose une gestion correcte du dépassement de la capacité de l'horloge logique du train. Elle s'appuie sur des valeurs non signées. Cette section montre que la capacité maximum de ces valeurs est minorée par le double du nombre maximum de processus sur le circuit des trains. Puis, elle présente les autres fondements théoriques de cette gestion du dépassement. Enfin, la section 6.1.1.3 décrit la mise en œuvre.

6.1.1.1 État de l'art des méthodes de gestion de dépassement de capacité

Cette section présente les cinq méthodes de gestion du problème de dépassement de capacité.

La première méthode consiste à ignorer le problème. Elle n'est pas réaliste dans le cas de l'utilisation dans un système industriel.

La deuxième méthode consiste à éviter le dépassement de capacité en stockant la donnée dans une structure de données suffisamment grande pour garantir que la donnée n'atteindra jamais la limite supérieure induite par cette structure de données. Ainsi, [Simatic, 2009a] propose de stocker l'heure du dernier changement d'une information sous la forme du nombre de secondes écoulées depuis un temps initial t_0. Comme cette information a besoin d'être mémorisée pendant au maximum 85 minutes (soit 5 100 secondes), 2 octets suffisent, sans risquer de dépasser la capacité de ce compteur. Quant à [Murphy et Picco, 2006], il utilise un compteur de versions pour chaque donnée gérée par LIME répliqué. Une analyse de son code révèle que cet entier est stocké sur 8 octets. Supposons qu'une donnée change un million de fois par seconde. Alors, LIME répliqué ne connaîtra pas de dépassement de capacité avant $256^{8\ octets}/10^6\ changements\ par\ seconde/86\,400\ secondes\ par\ jour/365\ jours\ par\ an \simeq 600\,000$ ans : la taille de 8 octets garantit un nombre quasi-illimité de changements de versions. Calculons la taille de la structure de données qui serait nécessaire pour stocker l'horloge logique selon cette méthode. Supposons que l'algorithme doive s'exécuter pendant une durée de 10 ans entre 5 processus et 50 rotations de train par seconde. Au bout de 10 ans, l'horloge logique atteint la valeur $5\ processus \times 50\ rotations\ par\ seconde \times 86\,400\ secondes\ par\ jour \times 365\ jours\ par\ an \times 10\ ans \simeq 8 \cdot 10^{10}$. 5 octets suffisent pour stocker cette valeur. Avons-nous intérêt à chercher une méthode consommant moins d'octets ? Pour décider, reprenons la démonstration du lemme 20 afin de calculer le rende-

ment en termes de débit selon le nombre b d'octets utilisés pour stocker l'horloge logique lc. Nous obtenons : $e_{train\_lc\,sur\,b\,octets} = \frac{(n-1)\cdot u\cdot s}{9+b+(n-1)\cdot[7+u\cdot(5+s)]}$. Nous pouvons alors estimer le gain de rendement en utilisant b octets au lieu de 5 octets pour stocker lc. En effet, $gain_{lc\,sur\,b\,octets} = \frac{e_{train\_lc\,sur\,b\,octets} - e_{train\_lc\,sur\,5\,octets}}{e_{train\_lc\,sur\,5\,octets}}$. Donc : $gain_{lc\,sur\,b\,octets} = \frac{\frac{(n-1)\cdot u\cdot s}{9+b+(n-1)\cdot[7+u\cdot(5+s)]} - \frac{(n-1)\cdot u\cdot s}{14+(n-1)\cdot[7+u\cdot(5+s)]}}{\frac{(n-1)\cdot u\cdot s}{14+(n-1)\cdot[7+u\cdot(5+s)]}} = \frac{5-b}{9+b+(n-1)\cdot[7+u\cdot(5+s)]}$. Le tableau 6.1 présente quelque valeurs numériques de $gain_{lc\,sur\,b\,octets}$: pour de faibles tailles de messages, la recherche d'un méthode moins consommatrice d'octets peut apporter un gain supérieur ou égal à 4% (respectivement 1%) pour $b = 1$ (respectivement 4) octet(s).

TABLE 6.1 – Valeurs numériques de $gain_{lc\,sur\,b\,octets}$ pour $u = 1$ message par wagon, b octets pour stocker lc, n participants et s octets par message

n participants \ s octets	$b = 4$ octets			$b = 1$ octet		
	10	100	1000	10	100	1000
2	3%	8‰	1‰	13%	3%	4‰
5	1%	2‰	0,2‰	4%	1%	0,1‰

La troisième méthode consiste à utiliser une structure de données évolutive. L'objectif est que le protocole utilise un nombre réduit d'octets pour stocker les données pendant une durée importante d'exécution, sans pour autant introduire de dépassement de capacité. *Treedoc* est un exemple de structure de données évolutive [Preguica *et al.*, 2009] : les données y sont représentées sous la forme d'atomes stockés dans un arbre binaire. Mais cette structure et les algorithmes associés pour le nettoyage de l'arbre binaire nous semblent inadaptés au cas d'une simple horloge logique qu'un algorithme ne fait qu'incrémenter. Une piste plus prometteuse (mais à notre connaissance jamais observée dans la littérature) est d'utiliser une librairie comme *GNU Multiprecision Arithmetic Library*[1]. En effet, cette librairie permet de stocker des nombres dont la seule limite est la taille de la mémoire. Pour stocker un entier, elle requiert 4 octets pour la longueur plus 1 octet par facteur 256 du nombre. L'utilisation d'une telle librairie résout donc le problème d'absence de limite. Toutefois, dans le cas du protocole du train, dès que l'horloge logique dépasse la valeur 255, la librairie nécessite $4 + 2 = 6$ octets. Si nous reprenons l'hypothèse d'un train qui tourne 50 fois par seconde entre 5 processus, la valeur 255 est atteinte en un peu plus d'une seconde : au bout d'une seconde, l'utilisation de la librairie est moins rentable que la deuxième méthode (qui ne requérait systématiquement que 5 octets).

La quatrième méthode de gestion du dépassement de capacité consiste à remettre à 0 l'identifiant quand il risque d'atteindre la valeur maximum. [Yen et lu Huang, 1997] propose une solution générale adaptée aux vecteurs d'horloge. Cette solution consiste à déclencher un algorithme de remise à zéro des éléments de vecteurs, quand des critères de déclenchement se trouvent réunis. L'inconvénient de cette méthode est qu'elle nécessite un algorithme supplémentaire (en l'utilisant, nous complexifierions donc le protocole des

1. `http://gmplib.org/`

trains). De plus, elle entraîne une interruption de l'algorithme normal pendant la phase de remise à zéro.

La dernière méthode de gestion de dépassement de capacité est de laisser le compteur dépasser la valeur maximale due à la structure de données utilisée et d'utiliser un test adapté à ce dépassement de capacité. Ainsi, dans le cas de vecteurs d'horloges utilisés dans un algorithme de diffusion respectant l'ordre causal, [Baldoni, 1998] propose une mise en œuvre modulo k de chaque élément de vecteur où k est la largeur de la fenêtre causale de chaque processus. Même si [Eychenne et Simatic, 1996] n'en fait pas état, dans le codage initial du protocole des trains (qui est une version monotrain), il y a une tentative de mise en œuvre de cette méthode de gestion de dépassement de capacité : l'horloge logique du train y est stockée sous la forme d'un entier signé stocké sur 4 octets. Un train est considéré comme actuel si l'expression suivante est vraie :

$$(tr.st.lc > dte.st.lc) \vee [(tr.st.lc < 0) \wedge (dte.st.lc > 0)] \qquad (\mathscr{E})$$

dans laquelle $dte.st.lc$ désigne l'horloge logique du dernier train envoyé et $tr.st.lc$ est l'horloge logique du train reçu. La figure 6.1 illustre la circulation du train avec cette expression : quand p_3 reçoit le train $t_{2\,147\,483\,647}$, il augmente cet identifiant de 1. Or, l'horloge logique est égale à `0x7fffffff` en hexadécimal. C'est pourquoi cet entier signé fait un dépassement de capacité : sa valeur hexadécimale passe à `0x80000000`, soit `-2 147 483 648` pour un entier signé. De ce fait, p_3 envoie le train $t_{-2\,147\,483\,648}$. p_0 accepte ensuite ce train car la sous-expression $[(tr.st.lc < 0) \wedge (dte.st.lc > 0)]$ de $\mathscr{E}$ est vraie. Plus tard, p_2 reçoit t_{-1}. Par conséquent, p_2 envoie le train t_0 que p_3 accepte car la sous-expression $(tr.st.lc > dte.st.lc)$ de $\mathscr{E}$ est vraie. Le dépassement de capacité semble donc correctement géré. Mais, le scénario présenté à la figure 6.2 révèle que l'expression $\mathscr{E}$ peut conduire à une erreur dans le protocole d'uot-diffusion. En effet, comme précédemment, p_2 envoie le train $t_{2\,147\,483\,647}$ à p_3. À cause du dépassement de capacité, ce dernier envoie le train $t_{-2\,147\,483\,648}$ à p_0 qui l'accepte. Mais, cette fois, nous supposons qu'à ce moment-là, p_3 défaille. Aussi, p_2 renvoie à son nouveau successeur p_0 son dernier train envoyé. Donc, p_0 reçoit le train $t_{2\,147\,483\,646}$. p_0 devrait le considérer comme périmé puisqu'il a reçu $t_{-2\,147\,483\,648}$ de p_3 juste avant. Mais, p_0 l'accepte, car la sous-expression $(tr.st.lc > dte.st.lc)$ de $\mathscr{E}$ est vraie. Or, le wagon w_1 (respectivement w_2) contenant des uot-diffusions de p_1 (respectivement p_2) et transporté par $t_{-2\,147\,483\,648}$ est aussi présent dans $t_{2\,147\,483\,646}$: *in fine*, p_0 uot-livre w_1 et w_2 2 fois, ce qui met en défaut la propriété d'intégrité uniforme de l'uot-diffusion. De plus, à partir du train $t_{2\,147\,483\,646}$, p_0 ajoute un wagon w_0 (que p_0 uot-livrera *in fine*) dans le train $t_{2\,147\,483\,647}$ que p_0 envoie alors à p_1. Pendant ce temps, en parallèle, suite à la réception de $t_{-2\,147\,483\,647}$, p_1 envoie $t_{-2\,147\,483\,646}$ à p_2 qui accepte ce train, car la sous-expression $[(tr.st.lc < 0) \wedge (dte.st.lc > 0)]$ de $\mathscr{E}$ est vraie. Quand p_1 reçoit le train $t_{2\,147\,483\,647}$, p_1 l'accepte puisque la sous-expression $(tr.st.lc > dte.st.lc)$ de $\mathscr{E}$ est vraie. De ce fait, p_1 uot-livrera *in fine* w_0. Il ajoute un wagon w_1' (que p_1 uot-livrera *in fine*) et envoie le train $t_{-2\,147\,483\,648}$ à cause du dépassement de capacité de l'horloge logique. Pendant ce temps, en parallèle, suite à la réception de $t_{-2\,147\,483\,645}$, p_0 envoie $t_{-2\,147\,483\,644}$ à p_1 qui accepte ce train, car la sous-expression $[(tr.st.lc < 0) \wedge (dte.st.lc > 0)]$ de $\mathscr{E}$ est vraie. Cependant, cette circulation erronée de deux trains en parallèle s'arrête. En effet, p_2 considère que $t_{-2\,147\,483\,648}$ est obsolète puisque l'expression $\mathscr{E}$ est fausse. Donc, p_2 n'uot-livrera jamais w_0 et w_1 : la propriété d'accord uniforme de l'uot-diffusion est mise en défaut. Notez que,

par ailleurs, en parallèle, p_1 reçoit $t_{-2\,147\,483\,644}$: p_1 envoie $t_{-2\,147\,483\,644}$ à p_2 qui l'accepte, etc. Dans la pratique, l'anomalie que nous venons de présenter n'a jamais été observée. En effet, sa probabilité est très faible. Comme nous le constatons avec la figure 6.2, la défaillance de p_3 doit arriver à un moment très particulier de la circulation du train, soit dans un cas sur le nombre de valeurs possibles pour un identifiant codé sur 4 octets, c'est-à-dire $256^4 = 4\,294\,967\,296$. Supposons qu'il existe encore une dizaine d'autres scénarios défavorables, et que le processus p_3 et la machine qui l'héberge ont un taux global de fiabilité de 99%[2]. Nous pouvons alors estimer la probabilité d'observer ce phénomène à : $10 \times (1 - 0,99) \times \frac{1}{4\,294\,967\,296} \simeq 2 \cdot 10^{-11}$. La probabilité d'observer cette erreur est donc très faible[3], mais non nulle : telle qu'elle a été mise en œuvre pour le protocole des trains, cette méthode n'est pas satisfaisante.

Nous avons présenté les cinq méthodes de gestion de dépassement de capacité. Pour chacune, nous avons analysé leur(s) limite(s), notamment l'incorrection de la mise en œuvre de la cinquième méthode pour le protocole des trains. La section suivante étudie une mise en œuvre correcte de cette cinquième méthode.

6.1.1.2 Théorie pour la gestion du dépassement de la capacité

Cette section présente l'ensemble des fondements théoriques pour une gestion correcte (et performante) du dépassement de la capacité de l'horloge logique du train. Pour simplifier les démonstrations, elle traite la version monotrain du protocole. La section 6.1.1.3 généralise l'approche à la version multitrain du protocole.

Soient :
- M : le nombre maximum de valeurs entières positives (ou nulle) que peut prendre l'horloge logique de train ;
- *dte.st.lc* : l'horloge logique du dernier train envoyé ; dans cette section et la suivante, nous supposons que cette horloge est stockée sous la forme d'un entier non signé pouvant valoir au maximum $M - 1$. Donc $dte.st.lc \in [\![0, M[\![$;
- *tr.st.lc* : l'horloge logique du train reçu. De même que *dte.st.lc*, $tr.st.lc \in [\![0, M[\![$;
- N : le nombre maximum de processus qui peuvent faire partie du circuit de train à tout instant ;
- n : le nombre de processus (notés p_0 à p_{n-1}) qui participent, à un instant donné, au protocole du train. Vu qu'il faut au minimum 2 processus pour que le train circule et qu'il y a au maximum N processus, $n \in [\![2, N]\!]$.

Théorème 6. *M est minoré selon l'inéquation suivante : $M \geqslant 2N + 1$*

Démonstration. Considérons que p_0 a envoyé le train d'horloge logique a ($a \in [\![0, M[\![$) à un instant donné. À l'aide de la figure 6.3, intéressons-nous au processus p_i ($i \in]\!]0, n-1]\!]$).

Tant que p_i n'a pas reçu le train $t_{a+n+i-1}$, il est susceptible de recevoir le train t_a. En effet, une fois que p_i a envoyé le train t_{a+i} à p_{i+1}, si les processus p_1 à p_{i-1} défaillent, et si

2. Ce taux est élevé, mais permet de se donner une marge d'erreur par rapport au nombre de scénarii défavorables.

3. Donnons un ordre de grandeur de ce phénomène : un joueur de Loto a environ 1 chance sur 14 millions de trouver les 6 numéros. Par conséquent, ce joueur a environ 3 000 fois plus de chances de gagner au Loto que d'observer cette erreur dans le protocole d'uot-diffusion.

FIGURE 6.1 – Diagramme de séquence de la circulation d'un train avec dépassement de la capacité de l'horloge logique

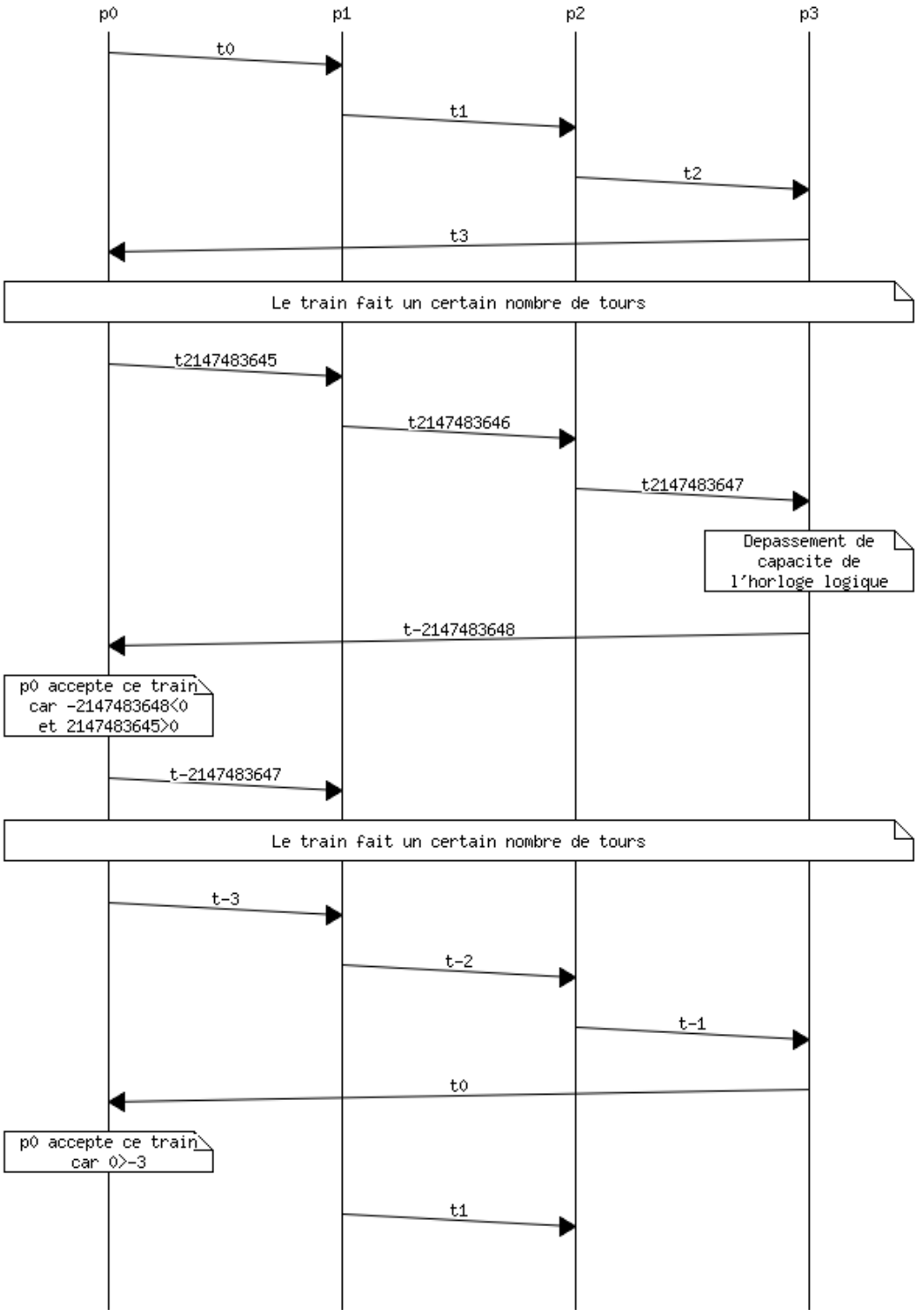


FIGURE 6.2 – Diagramme de séquence montrant une anomalie lors de la circulation d'un train avec dépassement de la capacité de l'horloge logique

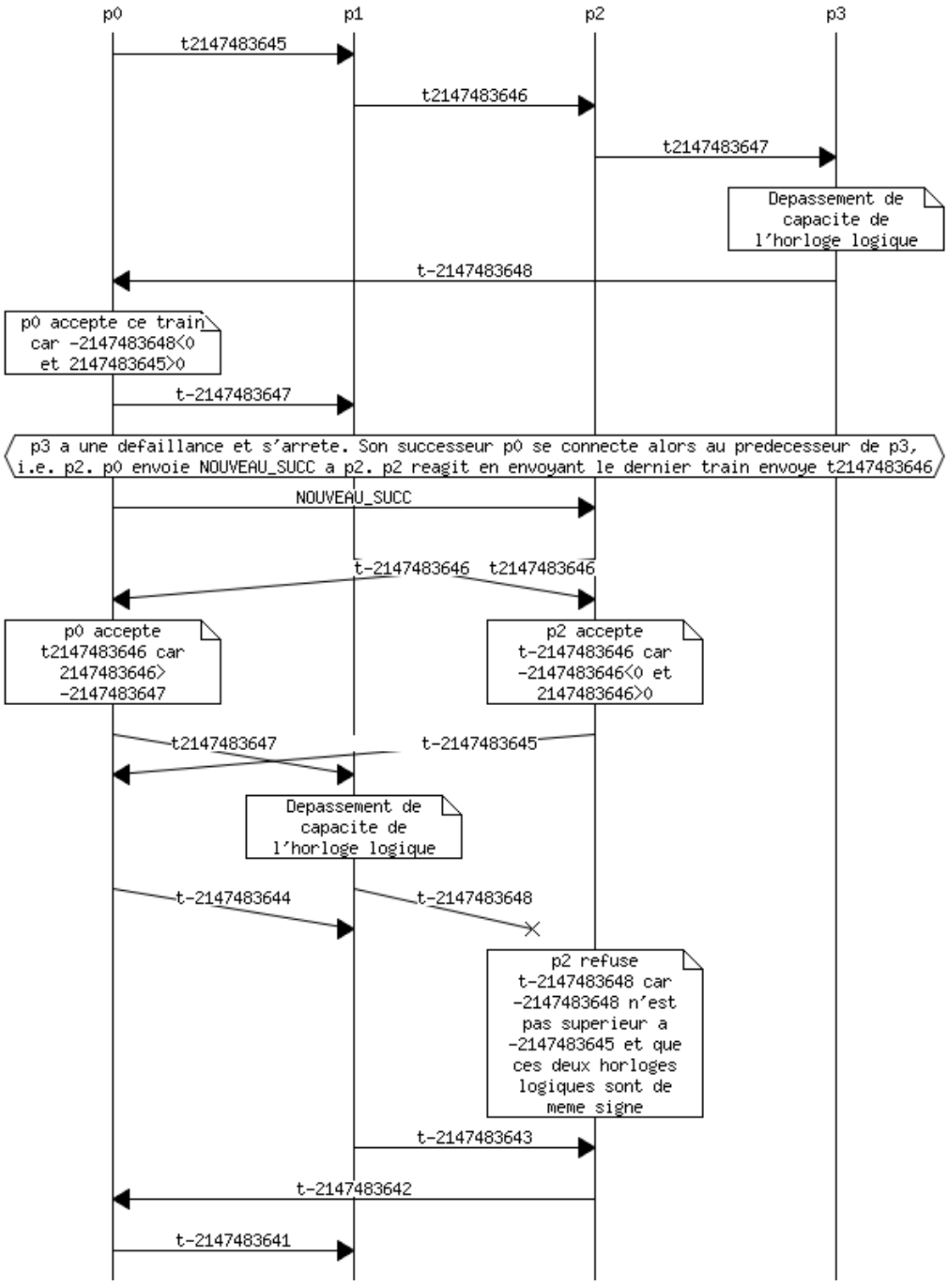
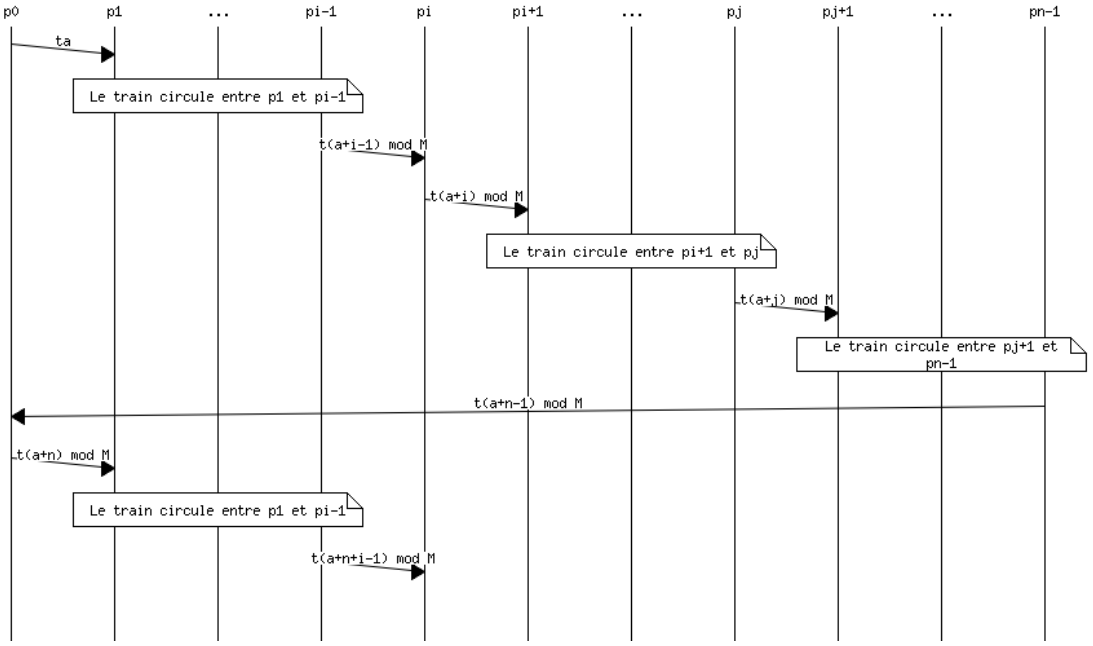


FIGURE 6.3 – Diagramme de séquence de la circulation d'un train entre n processus



p_0 n'a pas encore reçu t_{a+n-1} de p_{n-1} (ou bien, de manière plus générale, un train issu de p_j, $j \in [\![i+1, n-1[\![$ dans le cas où les processus p_{j+1} à p_{n-1} ont aussi défailli), p_0 envoie t_a à p_i.

Or, dans le cas d'une horloge logique à capacité limitée, en cas de dépassement de capacité, l'horloge logique du train $t_{a+n+i-1}$ ne doit pas pouvoir être confondue avec celle du train t_a, sinon le processus p_i pourrait accepter t_a alors que ce train est périmé pour p_i. Donc :

$$\begin{aligned}
&\forall a \in [\![0, M[\![, \forall i \in]\!]0, n-1]\!], \forall n \in [\![2, N]\!], && a > (a+n+i-1) \bmod M \\
&\forall i \in]\!]0, n-1]\!], \forall n \in [\![2, N]\!], \forall k \in \mathbb{N}^*, && 0 > n+i-1-kM \\
&\forall i \in]\!]0, n-1]\!], \forall n \in [\![2, N]\!], \forall k \in \mathbb{N}^*, && kM > n+i-1 \\
&\text{Pour } i = n-1,\ k = 1, \forall n \in [\![2, N]\!], && M > n+(n-1)-1 \\
&\text{Pour } n = N, && M > 2N-2 \\
&\text{On en conclut :} && M \geqslant 2N-1
\end{aligned}$$ □

Corollaire 2. *Il ne peut pas y avoir plus d'un dépassement de capacité de l'horloge logique du train sur 2 tours de train.*

Démonstration. Considérons que p_0 envoie le train d'horloge logique a ($a \in [\![0, M[\![$) à un instant donné. Après le deuxième tour de train, p_0 reçoit de p_{n-1} le train $t_{(a+n+n-1) \bmod M}$.

Or, $a \in [\![0, M[\![$ et $n \leqslant N$. Donc : $0 \leqslant a+2n-1 < M+2N-1$. Comme $M \geqslant 2N-1$ (cf. théorème 6), $0 \leqslant a+2n-1 < 2M$.

Par conséquent, soit $(a+2n-1) \bmod M = a+2n-1$ (il n'y a pas eu de dépassement

de capacité au cours des 2 tours de train), soit $(a + 2n - 1) \bmod M = a + 2n - 1 - M$ (il y a eu un seul dépassement de capacité au cours des 2 tours du train).

□

Théorème 7. *Si un processus reçoit un train d'horloge logique tr.st.lc et que* $tr.st.lc - dte.st.lc > 0$*, alors ce train est récent pour ce processus si et seulement si* $tr.st.lc - dte.st.lc < \frac{1+M}{2}$.

Démonstration. Notons $\mathscr{A}$ la proposition « Le train d'horloge logique *tr.st.lc* est récent pour ce processus ». Notons $\mathscr{B}$ la proposition « $tr.st.lc - dte.st.lc < \frac{1+M}{2}$ ». Pour démontrer que $\mathscr{A} \iff \mathscr{B}$, nous allons démontrer que $\mathscr{A} \implies \mathscr{B}$, puis que $\mathscr{B} \implies \mathscr{A}$.

1. Montrons tout d'abord que $\mathscr{A} \implies \mathscr{B}$.

   Soit un processus p_0 qui envoie un train $t_{a,a\in[\![0,M[\![}$ à un instant donné (cf. figure 6.3). Notre hypothèse de départ est la proposition $\mathscr{A}$. Donc, après avoir envoyé t_a, p_0 reçoit un train récent. Considérons que c'est le processus p_i ($i \in]\!]0, n-1]\!]$) qui envoie ce train récent (NB : $i \neq n-1$ si les processus p_{i+1}, …, p_{n-1} ont défailli). Comme le train est récent, son horloge logique vaut $(a + i) \bmod M$.

   D'après le corollaire 2, $(a + i) \bmod M$ vaut $a + i$ ou bien $a + i - M$. Montrons tout d'abord que $(a + i) \bmod M$ vaut forcément $a + i$. Pour ce faire, supposons que $(a+i) \bmod M$ vaut $a+i-M$ et montrons que nous aboutissons à une contradiction. Si $(a + i) \bmod M = a + i - M$, alors :

   $$\begin{aligned} tr.st.lc - dte.st.lc &= [(a + i) \bmod M] - a \\ &= [a + i - M] - a \\ &= i - M \end{aligned}$$

   Pour $i = n - 1$ et comme $n \leqslant N$:

   $$tr.st.lc - dte.st.lc \leqslant N - 1 - M$$

   Comme $N \leqslant \frac{1+M}{2}$ (d'après le théorème 6) :

   $$\begin{aligned} tr.st.lc - dte.st.lc &\leqslant \frac{1 + M}{2} - 1 - M \\ &\leqslant -\frac{1 + M}{2} \\ &< 0 \end{aligned}$$

   Nous aboutissons à une contradiction avec l'hypothèse du théorème : $tr.st.lc - dte.st.lc > 0$. Par conséquent, $(a + i) \bmod M = a + i$.

   Nous pouvons maintenant majorer $tr.st.lc - dte.st.lc$:

   $$\begin{aligned} tr.st.lc - dte.st.lc &= [(a + i) \bmod M] - a \\ &= [a + i] - a \\ &= i \end{aligned}$$

Pour $i = n-1$, comme $n \leqslant N$ et $N \leqslant \frac{1+M}{2}$ (d'après le théorème 6) :

$$tr.st.lc - dte.st.lc \leqslant \frac{1+M}{2} - 1$$

Par conséquent :

$$tr.st.lc - dte.st.lc < \frac{1+M}{2}$$

Nous en concluons : $\mathscr{A} \Longrightarrow \mathscr{B}$

2. Montrons maintenant que $\mathscr{B} \Longrightarrow \mathscr{A}$. Pour ce faire, nous allons démontrer la contraposée de cette proposition, c'est-à-dire : $\neg\mathscr{A} \Longrightarrow \neg\mathscr{B}$ (« Si le train est périmé, alors $tr.st.lc - dte.st.lc \geqslant \frac{1+M}{2}$ »).
   Considérons un processus p_i ($i \in [\![0, n-1]\!]$) qui a reçu un train périmé, ce train ayant une horloge logique telle que $tr.st.lc - dte.st.lc > 0$ (par hypothèse du théorème). Le seul cas où cela peut arriver est dû à un dépassement de capacité illustré par la figure 6.4. Appelons p_0 le processus où a lieu le dépassement de capacité (rappel : selon le corollaire 2, il ne peut y avoir qu'un seul dépassement de capacité sur 2 tours de train). Quand ce processus reçoit le train t_{M-1}, il renvoie le train t_0. Le train circule ensuite jusqu'à p_i qui envoie le train t_i. Pour que p_i puisse recevoir un train tel que $tr.st.lc - dte.st.lc > 0$, il faut que ce train ait été envoyé par un processus p_j ($j \in [\![i+1, n-1]\!]$), que p_j n'ait pas encore reçu le train t_i (ou bien le train résultant de t_i), et enfin que ce processus p_j (considérant que les processus p_{j+1}, …, p_{n-1}, p_0, …, p_{i-1} ont défailli) remette en circulation le dernier train qu'il a envoyé (t_{M-n+j} en l'occurrence) vers p_i.
   Donc :

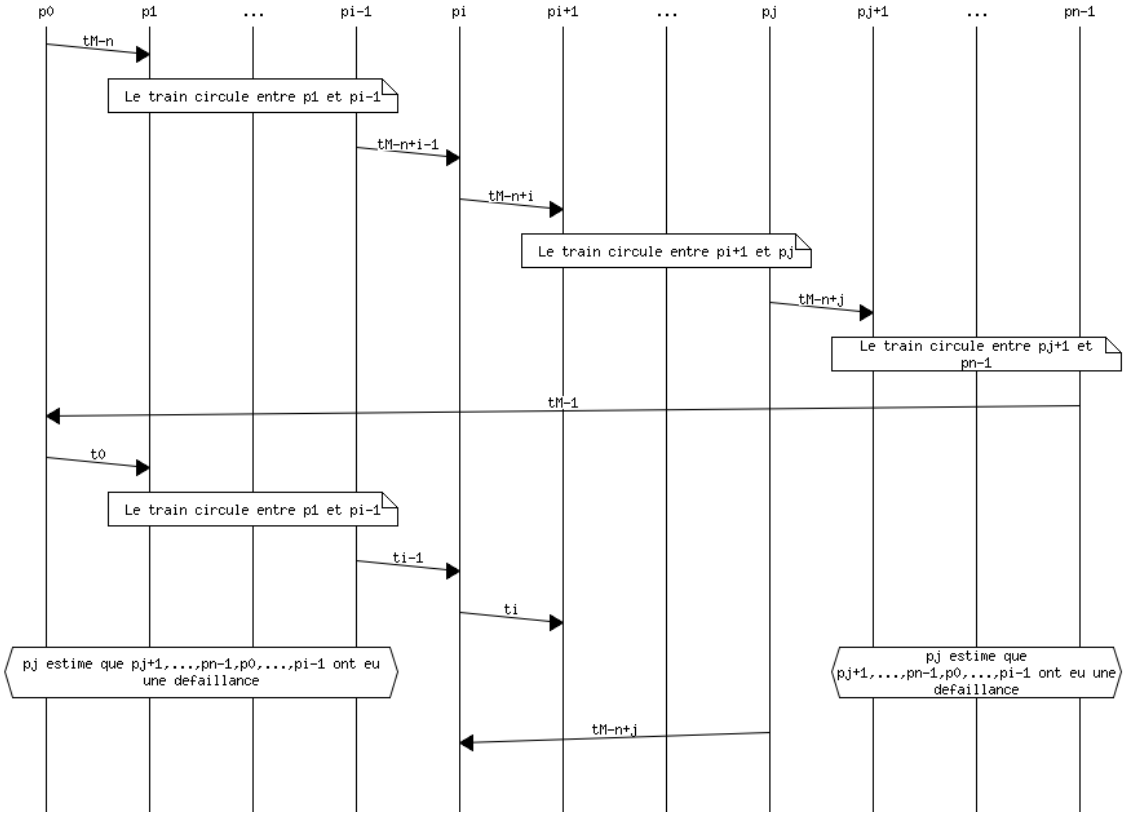
$\forall i \in [\![0, n-1[\![, \forall j \in [\![i+1, n-1]\!], \forall n \in [\![2, N]\!]$,	$tr.st.lc - dte.st.lc = (M - n + j) - i$
Pour $j = i+1$,$\forall i \in [\![0, n-1[\![, \forall n \in [\![2, N]\!]$,	$tr.st.lc - dte.st.lc \geqslant (M - n + i + 1) - i$
	$\geqslant M - n + 1$
Pour $n = N$,	$tr.st.lc - dte.st.lc \geqslant M - N + 1$
Comme $N \leqslant \frac{1+M}{2}$ (d'après le théorème 6) :	$tr.st.lc - dte.st.lc \geqslant M - \frac{1+M}{2} + 1$
Nous en déduisons :	$tr.st.lc - dte.st.lc \geqslant \frac{1+M}{2}$

   Nous concluons : $\neg\mathscr{A} \Longrightarrow \neg\mathscr{B}$ et donc $\mathscr{B} \Longrightarrow \mathscr{A}$

Nous avons démontré que $\mathscr{A} \Longrightarrow \mathscr{B}$ et que $\mathscr{B} \Longrightarrow \mathscr{A}$. Par conséquent : $\mathscr{A} \Longleftrightarrow \mathscr{B}$. □

Théorème 8. *Si un processus reçoit un train d'horloge logique tr.st.lc et que $tr.st.lc - dte.st.lc \leqslant 0$, alors ce train est récent pour ce processus si et seulement si $tr.st.lc - dte.st.lc < \frac{1-M}{2}$.*

FIGURE 6.4 – Diagramme de séquence de la circulation d'un train entre n processus avec dépassement de la capacité de l'horloge logique du train

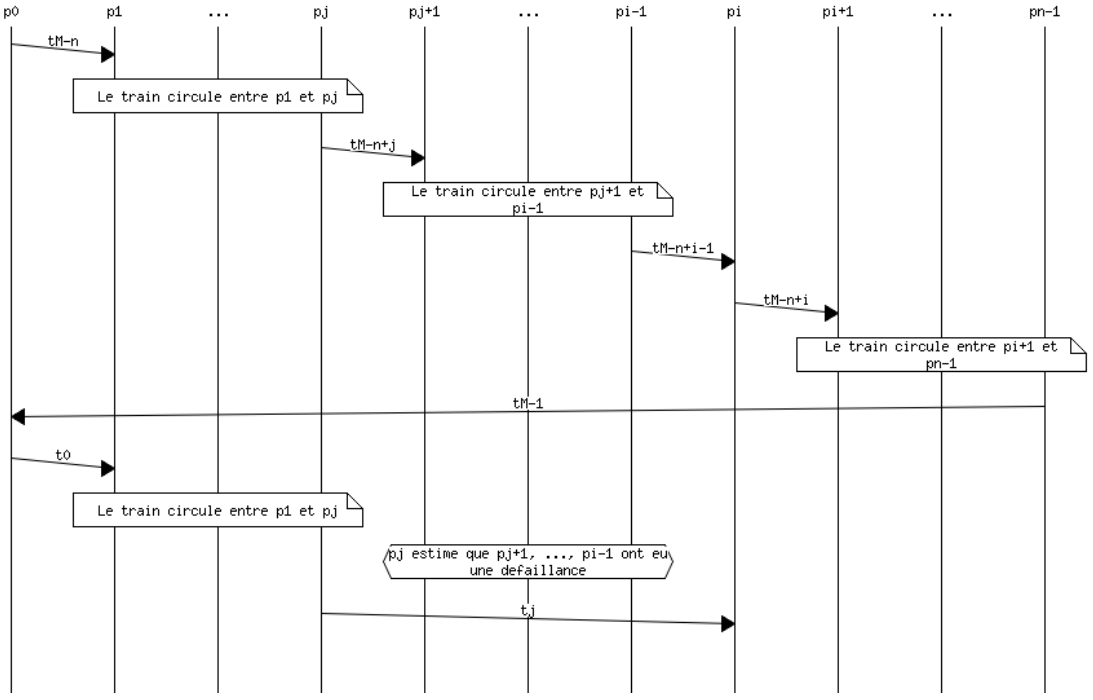


Démonstration. Notons $\mathscr{A}$ la proposition « Le train d'horloge logique $tr.st.lc$ est récent pour ce processus ». Notons $\mathscr{C}$ la proposition « $tr.st.lc - dte.st.lc < \frac{1-M}{2}$ ». Pour démontrer que $\mathscr{A} \iff \mathscr{C}$, nous allons démontrer que $\mathscr{A} \implies \mathscr{C}$, puis que $\mathscr{C} \implies \mathscr{A}$.

1. Montrons tout d'abord que $\mathscr{A} \implies \mathscr{C}$.

   Supposons que $\mathscr{A}$ est vérifiée au niveau du processus p_i ($i \in [\![1, n-1]\!]$). Par hypothèse du théorème, $tr.st.lc - dte.st.lc \leqslant 0$. Donc, p_i a reçu un train récent d'horloge logique inférieur à $dte.st.lc$: il y a eu dépassement de capacité de l'horloge logique du train au niveau de p_0 et le train a été envoyé par le processus p_j ($j \in [\![0, i-1]\!]$, *cf.* figure 6.5).

FIGURE 6.5 – Autre diagramme de séquence de la circulation d'un train entre n processus avec dépassement de la capacité de l'horloge logique du train



$\forall i \in [\![1, n-1[\![, \forall j \in [\![0, i-1[\![, \forall n \in [\![2, N]\!]$, $tr.st.lc - dte.st.lc = (j) - (M - n + i)$
$= (j - i) - M + n$

Pour $j = i - 1$ et $n = N$, $tr.st.lc - dte.st.lc \leqslant -1 - M + N$

Comme $N \leqslant \frac{1+M}{2}$ (d'après le théorème 6) : $tr.st.lc - dte.st.lc \leqslant -1 - M - \frac{1+M}{2}$
$\leqslant \frac{-1-M}{2}$
$< \frac{-1-M}{2} + 1$

Par conséquent : $tr.st.lc - dte.st.lc < \frac{1-M}{2}$

Nous en concluons : $\mathscr{A} \Longrightarrow \mathscr{C}$

2. Montrons maintenant que $\mathscr{C} \Longrightarrow \mathscr{A}$. Pour ce faire, nous allons démontrer la contraposée de cette proposition, c'est-à-dire : $\neg\mathscr{A} \Longrightarrow \neg\mathscr{C}$ (« Si le train est périmé pour ce processus, alors $tr.st.lc - dte.st.lc \geqslant \frac{1-M}{2}$ »).
Considérons que p_0 a envoyé le train t_a ($a \in [\![0, M[\![$, *cf.* figure 6.3) à un instant donné, puis le train $t_{(a+n) \bmod M}$.
Comme nous supposons $\neg\mathscr{C}$, p_0 reçoit alors un train périmé. Considérons que c'est le processus p_i ($i \in]\!]0, n-1]\!]$) qui envoie ce train. Puisque ce train est périmé, son horloge logique est $(a + i) \bmod M$.
Montrons tout d'abord qu'il n'y a pas eu de dépassement de capacité entre $a + i$ et $a + n$. Pour ce faire, supposons qu'il y ait eu dépassement de capacité. Dans ce cas, puisque d'après le corollaire 2, il ne peut pas y avoir plus de 2 dépassements de capacité sur 2 tours de train :

$$\exists k \in [\![0, 1]\!], ((a + i) \bmod M = a + i - kM) \wedge ((a + n) \bmod M = a + n - (k + 1)M)$$

De ce fait :

$$\begin{aligned} tr.st.lc - dte.st.lc &= [(a + i) \bmod M] - [(a + n) \bmod M] \\ &= [a + i - kM] - [a + n - (k + 1)M] \quad (\text{pour } k \in [\![0, 1]\!]) \\ &= i - n + M \end{aligned}$$

Pour $i = 1$ et $n = N$:

$$tr.st.lc - dte.st.lc \geqslant 1 - N + M$$

Comme $N \leqslant \frac{1+M}{2}$ (d'après le théorème 6) :

$$\begin{aligned} tr.st.lc - dte.st.lc &\geqslant 1 - \frac{1+M}{2} + M \\ &\geqslant \frac{1+M}{2} \\ &> 0 \end{aligned}$$

Nous aboutissons à une contradiction avec l'hypothèse du théorème : $tr.st.lc - dte.st.lc \leqslant 0$. Par conséquent, il n'y a pas de dépassement de capacité entre $a+i$ et $a+n$.

Nous pouvons maintenant minorer $tr.st.lc - dte.st.lc$:

$$\begin{aligned} tr.st.lc - dte.st.lc &= [(a+i) \bmod M] - [(a+n) \bmod M] \\ &= [a+i-kM] - [a+n-kM] \quad (\text{pour } k \in [\![0,1]\!]) \\ &= i-n \end{aligned}$$

Pour $i = 1$ et $n = N$:

$$tr.st.lc - dte.st.lc \geqslant 1 - N$$

Comme $N \leqslant \frac{1+M}{2}$ (d'après le théorème 6) :

$$tr.st.lc - dte.st.lc \geqslant 1 - \frac{1+M}{2}$$

Par conséquent :

$$tr.st.lc - dte.st.lc \geqslant \frac{1-M}{2}$$

Nous en concluons : $\neg\mathscr{A} \Longrightarrow \neg\mathscr{C}$ et donc $\mathscr{C} \Longrightarrow \mathscr{A}$

Nous avons démontré que $\mathscr{A} \Longrightarrow \mathscr{C}$ et que $\mathscr{C} \Longrightarrow \mathscr{A}$. Par conséquent : $\mathscr{A} \Longleftrightarrow \mathscr{C}$. □

Cette démonstration termine la présentation des fondements théoriques pour une gestion correcte du dépassement de la capacité de l'horloge logique du train. La section suivante présente comment implanter cette théorie.

6.1.1.3 Codage de la gestion du dépassement de la capacité

Cette section présente la mise en œuvre de la théorie présentée dans la section 6.1.1.2.

Toutes les horloges logiques de train sont des entiers non signés pouvant valoir au maximum $M-1$, avec $M \geqslant 2 \cdot N + 1$, N étant le nombre maximum de processus qui peuvent faire partie du circuit de train à tout instant.

En version monotrain, quand un processus p reçoit un train d'horloge logique `tr.st.lc`, la fonction qui permet de décider si un train est récent ou non (donc périmé) utilise `tr.st.lc` et `dte.st.lc` (l'horloge logique du dernier train que p a envoyé) selon l'algorithme 15 pour gérer tous les cas de dépassement de capacité du compteur de train.

Lemme 26. *La fonction `estCeQueTrainRécent(tr_st,dte_st)` (cf. algorithme 15) renvoie `VRAI` si et seulement si le train reçu est récent et `FAUX` s'il est périmé.*

Algorithme 15 Fonction `estCeQueTrainRécent(tr_st,dte_st)` avec gestion des dépassements de capacité (version monotrain)

```
1: variable locale: différence ← tr_st.lc − dte_st.lc
2: if différence > 0 then
3:    return (différence < (1+M)/2)
4: else
5:    return (différence < (1−M)/2)
6: end if
```

Démonstration. La ligne 15.1 permet de calculer la différence qui sert d'hypothèse aux théorèmes 7 et 8. Ensuite, le test de la ligne 15.2 nous permet de décider quel théorème nous appliquons. En effet, si `différence` est strictement positif, nous utilisons le théorème 7 : le test ($\texttt{différence} < \frac{1+M}{2}$) de la ligne 15.3 est vrai si et seulement si le train reçu est récent pour ce processus. En revanche, si `différence` est négatif ou nul, nous nous servons du théorème 8 : le test ($\texttt{différence} < \frac{1-M}{2}$) de la ligne 15.5 est vrai si et seulement si le train reçu est récent pour ce processus. Nous en concluons que la fonction `estCeQueTrainRécent(tr_st,dte_st)` renvoie `VRAI` si et seulement si le train reçu est récent et `FAUX` s'il est périmé. □

Pour implanter cet algorithme, nous proposons de stocker les horloges logiques de train dans des `unsigned char`. De ce fait, $M = 256$ et $N \leqslant \frac{1+M}{2} = 128$. Cette valeur de N est suffisante pour être au delà du nombre de stations où la latence induite par le protocole des trains est acceptable. Nous proposons ensuite de stocker la variable locale `différence` dans un `short` de manière à pouvoir faire des calculs signés et ne pas perdre en précision lors des calculs de différence.

Au final, nous constatons que la gestion du dépassement de la capacité de l'horloge logique du train requiert :

- un octet pour stocker l'horloge logique du train (et gérer ainsi jusqu'à 128 processus participant au protocole des trains),
- à chaque réception de train, le calcul d'une différence entre deux `short`, une comparaison à 0, suivie d'une comparaison (au lieu d'une simple comparaison, si nous n'avions pas le dépassement de capacité à gérer).

L'algorithme 16 généralise l'algorithme 15 au cas multitrain en tenant compte de la notion d'identifiant de train.

Lemme 27. *La fonction `estCeQueTrainRécent(tr_st, dte, dernièreInstanceEnvoyée, ntr)` (cf. algorithme 16) renvoie `VRAI` si et seulement si le train reçu a l'identifiant qui est attendu et si ce train est récent. Cette fonction renvoie `FAUX` dans les autres cas.*

Démonstration. La ligne 16.1 détermine l'identifiant attendu. Si l'identifiant du train reçu est différent, la fonction renvoie `FAUX` (lignes 16.2 et 16.10). Si l'identifiant du train reçu est égal, la ligne 16.3 permet de calculer la différence qui sert d'hypothèse aux théorèmes 7 et 8. Ensuite, en reprenant le raisonnement de la démonstration du lemme 26, nous montrons que les lignes 16.4–8 renvoient `VRAI` si et seulement le train reçu est récent. □

Algorithme 16 Fonction `estCeQueTrainRécent(tr_st, dte, dernièreInstanceEnvoyée, ntr)` avec gestion des dépassements de capacité (version multitrain)

```
1: variable locale: idAttendu ← (die + 1) mod ntr
2: if tr_st.id == idAttendu then
3:   variable locale: différence ← tr_st.lc − dte[idAttendu].st.lc
4:   if différence > 0 then
5:     return (différence < (1+M)/2)
6:   else
7:     return (différence < (1−M)/2)
8:   end if
9: else
10:  return false
11: end if
```

6.1.2 Codage du circuit des trains dans l'entête du train

Le protocole des trains nécessite de transporter le circuit des trains au sein de chaque train. Cela permet : 1) de signaler des changements dans la liste des participants au protocole, 2) de permettre à des processus qui se croient membres du circuit des trains de prendre conscience qu'ils ont été en fait retirés du circuit par d'autres processus. Cette section étudie trois méthodes pour implanter le circuit des trains. La première consiste à transporter systématiquement toutes les adresses des participants. Les deux autres ont pour objectif de ne pas transporter inutilement des octets concernant le circuit des trains et réduire ainsi la consommation de ressource réseau.

Dans cette section, nous supposons que l'adresse protocolaire d'un processus requiert 6 octets : 4 pour l'adresse IPv4 et 2 pour le numéro de port.

La première méthode de codage du circuit des trains au sein de chaque train consiste à utiliser un octet `nbAdresses` indiquant le nombre d'adresses listées dans le circuit stocké dans le train, suivi par la liste des adresses protocolaires des membres du circuit. Dans le cas où le circuit compte 5 processus, ce codage du circuit des trains requiert $1 + 5 \times 6 = 31$ octets dans chaque train.

La deuxième méthode de codage du circuit des trains s'inspire de la première. Son objectif est de ne transporter les adresses protocolaires des membres du circuit des trains que s'il y a eu un changement depuis le dernier passage du train. Pour ce faire, nous ajoutons au codage du circuit employé dans la première méthode, un octet `numCircuit` pour stocker le numéro de circuit. Ce numéro est utilisé de la manière suivante. Quand un processus p reçoit un train et qu'il modifie le circuit des trains, p fixe `nbAdresses` au nombre de membres dans le circuit, incrémente l'octet `numCircuit` et ajoute les adresses protocolaires de chaque membre du circuit des trains. Quand un processus p' reçoit un train et qu'il n'a pas de modification de circuit à signaler, si p' constate que `numCircuit` est différent du `numCircuit` du dernier train qu'il a envoyé, il recopie tel quel le circuit des trains. En revanche, si p' constate que `numCircuit` est égal au `numCircuit` du dernier train qu'il a envoyé, p' fixe `nbAdresses` à 0 et met juste dans le train `nbAdresses` et `numCircuit` (p' ne met aucune adresse protocolaire). Ainsi, le circuit des trains ne prend plus que 2 octets sur

chaque train en l'absence de changement de circuit. Notez que `numCircuit` permet à un processus p'' qui a été éjecté du circuit des trains de se rendre compte de cette éjection. En effet, quand p'' reçoit le train, si jamais ce train ne contient aucune adresse protocolaire, p'' peut quand même comparer le `numCircuit` stocké dans ce train avec le `numCircuit` dont il était conscient. S'il y a une différence, p'' sait qu'il a raté des changements dans le circuit des trains et qu'il ne fait plus partie de ce circuit. Par ailleurs, si le nombre maximum de processus dans le circuit est limité à 15, cette deuxième méthode permet de stocker le circuit des trains sur un seul octet en utilisant des champs de bits : 4 bits pour `nbAdresses` et 4 bits pour `numCircuit`. Toutefois, cette deuxième méthode présente un inconvénient. En effet, pendant que p'' est mis à l'écart, `numCircuit` risque d'évoluer de telle manière que, quand p'' reçoit un train, `numCircuit` corresponde au `numCircuit` dont p'' a conscience. Dans ce cas, p'' ne signale pas de problème de partitionnement à la couche applicative (lignes 12.4 et 12.11). Notez qu'intuitivement, la probabilité que ce problème soit observé est faible dès qu'il y a suffisamment de valeurs différentes (256 que nous pourrions peut-être réduire à 16, voire 8) dans `numCircuit`.

La troisième méthode de codage du circuit des trains pallie cet inconvénient. Chaque processus participant au protocole a accès à un fichier listant l'ensemble des adresses protocolaires possibles. Ce fichier est un fichier commun à l'ensemble des processus (qui y accèdent via NFS par exemple) ou bien un fichier spécifique à chaque processus, le contenu de chacun de ces fichiers étant commun à tous les participants. Si le nombre maximum d'adresses possible est limité (par exemple, 16), nous représentons alors le circuit des trains sous la forme d'un masque de bits signalant la présence ou l'absence des différentes adresses protocolaires dans le circuit. Au démarrage d'un processus p, p lit ce fichier et calcule le rang rg de la ligne contenant sa propre adresse protocolaire. Quand le successeur p_s de p veut indiquer que p fait désormais partie du circuit, il met à 1 le bit de rang rg dans le champ de bits représentant le circuit des trains et stocké dans chaque train. Le codage du circuit reste compact (2 octets pour stocker 16 adresses protocolaires différentes) et pallie l'inconvénient de la deuxième méthode. De plus, un processus peut tester son appartenance au circuit des trains avec une instruction `ET` bit à bit. En contrepartie, l'ordre des processus dans le circuit est imposé, ce qui induit des contraintes dans la recherche du futur successeur sur le circuit des trains (cf. section 6.1.4). De plus, cette troisième méthode présente l'inconvénient de nécessiter une procédure spécifique en cas de besoin de mise à jour du contenu du fichier (par exemple, si nous avons besoin d'introduire une machine ayant une adresse IP que nous n'avions pas prévue lors du démarrage du protocole).

Pour notre prototype évoqué en section 6.1.5, nous avons choisi la troisième méthode de codage avec un circuit des trains codé sur 2 octets. En effet, cette méthode nous semblait plus simple à mettre en œuvre.

6.1.3 Régulation du débit d'émission dans le protocole des trains

La mise en œuvre du protocole des trains peut amener à limiter la taille des wagons pour deux raisons : 1) le système d'exploitation impose que les messages confiés à TCP ne dépasse pas une certaine taille (par exemple, dans les années 1990, *DEC-VMS* ne permettait

pas d'écrire plus de 64 Kio[4] sur le descripteur système associé à un canal TCP) ; 2) nous souhaitons éviter qu'un des processus puisse stocker trop de messages d'un seul coup dans un wagon afin que le train ne soit pas ralenti par ce wagon très long.

Notez que cette dernière motivation ne doit pas être confondue avec le besoin d'équité auquel est confronté le protocole LCR [Guerraoui *et al.*, 2010]. Dans ce dernier, quand un processus p uot-diffuse un grand nombre de messages, avec le protocole LCR de base qui ne respecte pas l'équité, son successeur p_s dans l'anneau virtuel ne peut plus uot-diffuser de messages. En effet, p_s passe son temps à faire suivre les messages de p. [Guerraoui *et al.*, 2010] propose donc un mécanisme pour que p_s puisse décider de ne pas faire suivre les messages de p, le temps qu'il puisse uot-diffuser ses propres messages. Dans le protocole des trains, il n'y a pas besoin d'un tel mécanisme puisque, quand le processus p_s reçoit un train, p_s peut toujours ajouter `wagonAEnvoyer` derrière le wagon qu'a éventuellement mis p dans ce train.

Toutefois, `wagonAEnvoyer` peut devenir tellement long entre deux passages de train que le train résultant est également très long. De ce fait, le réseau met du temps à écouler ce train volumineux. C'est pourquoi lorsqu'un processus p décide de participer au protocole des trains, nous avons décidé que la borne supérieure de la taille des wagons, notée `BSW` dans la suite, était un paramètre du protocole. Ainsi, quand la couche applicative souhaite uot-diffuser un message m de taille s, si $s >$ `BSW`, la couche applicative doit attendre que `wagonAEnvoyer` soit vide avant de pouvoir confier son message à la couche intergicielle[5]. La couche applicative observe la même attente dans le cas où $s \leqslant$ `BSW` et que l'ajout de m à `wagonAEnvoyer` serait telle que la taille de `wagonAEnvoyer` dépasserait `BSW`. Nous régulons ainsi le débit de messages que la couche applicative souhaite uot-diffuser.

6.1.4 Recherche du futur successeur sur le circuit des trains

Lorsqu'un processus p décide de participer au protocole des trains, il doit trouver son futur successeur sur le circuit des trains. Pour cette recherche, p peut multicaster un message `QUI_EST_LÀ`, attendre des réponses, et décider que son futur successeur est le premier processus qui lui a répondu. Cette solution possède l'avantage d'être simple et de permettre de trouver rapidement un successeur pour p. Toutefois, elle ne garantit pas que, si plusieurs participants sont sur la même machine, ces participants sont adjacents sur le circuit des trains (pour éviter que le train ne rentre sur la machine, en sorte, revienne, etc.). En effet, cette solution n'offre aucune garantie que c'est un processus de la machine qui répondra en premier au multicast du `QUI_EST_LÀ` d'un autre processus de la même machine. Par ailleurs, si nous choisissons de mettre en œuvre le circuit des trains à l'aide de bits de présence dans un masque de bits (cf. section 6.1.2), le multicast peut amener à créer un anneau virtuel qui n'est pas conforme à l'ordre induit par le masque de bits. Par exemple, imaginons qu'au vu du masque de bits, les processus p_1, p_2 et p_3 doivent toujours apparaître dans cet ordre dans le circuit. À un instant donné, p_1 et p_3 sont dans le circuit. p_2 multicaste son message `QUI_EST_LÀ`. Supposons que p_1 répond avant p_3. Alors p_2 prend p_1 comme successeur. Donc, on obtient le circuit p_2, p_1 et p_3.

4. Kio est le symbole du kibioctet. 1 Kio $= 2^{10}$ octets $= 1\,024$ octets.

5. Nous émettons également un message d'anomalie dans le journal d'exécution de l'application pour signaler que l'application a uot-diffusé un message de taille strictement supérieure à `BSW`.

TABLE 6.2 – Résultats des tests sur le protocole des trains (version multitrain)

nb proc.	nb trains	taille msg (octets)	nb msg / wagon	H_{trains} pratique (Mbps)	H_{trains} théorique (Mbps)	écart théorie	CPU
5	5	10	2184	73,8	78,3	-5,7%	8,5%
5	5	100	312	105,5	111,9	-5,7%	5,8%
5	5	1000	32	110,3	116,9	-5,7%	5,5%
5	5	10000	3	110,5	117,4	-5,8%	5,6%
5	10	10	2184	76,1	78,3	-2,8%	8,8%
5	10	100	312	108,7	111,9	-2,8%	6,0%
5	10	1000	32	113,6	116,9	-2,8%	5,6%
5	10	10000	3	113,9	117,4	-3,0%	5,7%

C'est pourquoi, s'il existe un fichier listant l'ensemble des adresses protocolaires possibles accessible à partir de chaque processus participant au protocole, nous avons intérêt à ne pas utiliser de multicast : lorsque p décide de participer au protocole des trains, p essaye d'ouvrir une connexion TCP vers chaque adresse protocolaire présente dans le fichier, en commençant par le successeur de sa propre adresse dans le fichier. S'il réussit, p considère qu'il a trouvé son successeur. Cette solution possède l'avantage de garantir un certain ordre des processus dans le circuit virtuel. De ce fait, si des processus appartenant à la même machine sont listés de manière contiguë dans le fichier et que l'ordre des processus dans le fichier est conforme à l'ordre induit par le masque de bits, les problème identifiés précédemment avec le multicast ne peuvent pas se produire. En contrepartie, la recherche du futur successeur prend plus de temps. De plus, elle est limitée aux adresses protocolaires mentionnées dans ce fichier commun à tous les processus.

6.1.5 Tests des performances

Avec l'aide de deux étudiants de Télécom SudParis travaillant dans le cadre de leur projet de 2^e année, nous avons implanté la version multitrain du protocole des trains. Le code issu de ce projet comprend 3 600 lignes (commentaires inclus). Il est disponible à l'adresse `https://github.com/simatic/TrainsProtocol`. Cette section analyse les premières performances observées avec ce code.

Nos mesures ont été effectuées selon la méthodologie présentée dans [Guerraoui *et al.*, 2010]. Nous avons d'abord évalué $H_{point-à-point}$ à l'aide de Netperf [Jones 2007] : le *switch* HP ProCurve 2610-24 que nous utilisons offre un débit TCP point-à-point de 94 Mbps. Ensuite, nous avons effectué nos différents tests sur un réseau de machines Dell Precision T3500 équipées d'un processeur Intel Xeon W3530 (2,80 GHz) et de 4 Gio de mémoire vive, fonctionnant sous Linux Fedora 15. Nous avons appliqué la procédure suivante : 5 minutes de temps d'échauffement, 10 minutes de tests effectifs, 5 minutes de refroidissement. Nos résultats sont synthétisés dans la table 6.2. Notez que, pour assurer la régulation du débit d'émission (cf. section 6.1.3), nous avons limité la taille de `wagonAEnvoyer` à 32 Kio. De ce fait, par exemple, notre code ne peut pas stocker plus de 3 messages de 10 000 octets dans `wagonAEnvoyer`.

Lorsque 5 processus participent au protocole, l'écart entre le débit théorique du protocole des trains et son débit pratique est de l'ordre de $-5,7\%$ quand 5 trains circulent en parallèle et d'environ $-2,8\%$ quand 10 trains circulent en parallèle. La faible différence entre les débits expérimentaux et les débits théoriques nous permettent de penser que le code actuel ne contient pas de défaut entraînant une anomalie majeure de performance. Toutefois, il apparaît qu'il faut deux fois plus de trains que de participants au protocole pour se rapprocher du débit maximal théorique. Ce facteur deux est révélateur d'un défaut dans notre code. En effet, c'est le même *thread* qui gère la réception d'un message, puis son traitement via l'algorithme 12. Notamment, quand un processus membre du circuit des trains reçoit un train, il le fait via un *thread* qui se charge de la réception de ce train (ligne 12.1) et de l'émission du train résultant (ligne 12.8). De ce fait, au sein du processus, il n'y a aucun parallélisme entre la réception et l'émission des messages. En mettant le double de trains, quand un processus p envoie un train t à son successeur p_s, même si p_s envoie à cet instant un train à son propre successeur, le train t arrive quand même au niveau de la machine de p_s, mais dans un tampon système de cette machine. De ce fait, quand par la suite p_s se met en attente d'un nouveau message, il peut immédiatement lire t qui est dans un tampon système. Notez que ce *thread* unique pour la réception et l'émission d'un message conduit à un interblocage entre les processus, si les tampons système ne sont pas assez grands pour qu'au moins un processus réussisse à exécuter entièrement son écriture. Nous mettons en évidence cet interblocage avec un test d'intégration dans lequel 6 participants font circuler 12 trains en parallèle et uot-diffusent autant de messages qu'ils le peuvent. Alors, comme chaque wagon stocké dans le train est plein, chaque train a une taille de $10 + 5 \times 32 \times 1\,024 = 163\,850$ octets (alors qu'il a une taille de $10 + 4 \times 32 \times 1\,024 = 131\,082$ octets dans le cadre de 5 participants). Or, sur nos machines, nous avons mesuré que la taille de chaque tampon système est d'environ 141 900 octets. De ce fait, avec 6 participants , un train entier ne peut pas être stocké dans le tampon système (alors qu'il peut l'être avec 5 participants). Par conséquent, chaque processus p se retrouve bloqué à envoyer un train vers son successeur p_s, car ce train ne peut pas rentrer entièrement dans le tampon système de p_s. Or, p_s ne vide pas ce tampon puisque p_s est lui-même bloqué en essayant d'envoyer un train vers son propre successeur. L'une des perspectives de cette thèse est de mettre en place deux *threads*, l'un pour la lecture des messages, l'autre pour leur traitement et donc l'envoi des messages résultant. Ces deux *threads* communiqueront via une queue FIFO [6] de messages. Il n'y aura alors plus aucun risque d'interblocage. De plus, nous paralléliserons l'activité de lecture de messages dans le tampon système et l'activité d'écriture sur le réseau : les performances devraient être améliorées.

La qualité des débits expérimentaux par rapport aux débits théoriques ne doit pas nous faire oublier que le débit théorique du protocole des trains peut être très inférieur au débit maximum de diffusion, c'est-à-dire $H_{point-à-point} \times 5/4 = 117,5$ Mbps. En effet, si $H_{trains_{théorique}}$ vaut 117,4 Mbps (soit -0,1% par rapport au débit maximum de diffusions) pour des messages de 10 000 octets, $H_{trains_{théorique}}$ se réduit à 78,3 Mbps (soit -33,4%) pour des messages de 10 octets. Cette réduction s'explique avec notre métrique coût (cf. définition 7 page 110). Lorsque la taille des messages est de 10 octets, $c_{train} = 33,4\%$. Si une application en éprouve le besoin, nous pouvons réduire ce coût. En effet, tous les champs `lng`

6. La propriété FIFO garantit que les propriétés d'uot-diffusion restent respectées.

transportés dans le train (longueur totale du train, longueur de chaque wagon, et longueur de chaque message) sont actuellement codés sur 4 octets. Or, la taille de `wagonAEnvoyer` est limitée à 32 Kio. Nous pouvons donc stocker le champ `lng` des wagons et des trains sur 2 octets. En adaptant la formule démontrée au lemme 20 page 111, c_{train} passe à 23,1% (soit une réduction de coût de 31,2% par rapport au coût initial). Si une grande majorité des messages fait moins de 127 octets, nous pouvons aller encore plus loin en reprenant la méthode employée dans l'intergiciel *ZeroMQ* [Hin08] : la longueur du message est stockée sur un octet, avec un des bits de cet octet qui permet d'indiquer si le message fait plus de 127 octets. Avec cette dernière méthode, c_{train} passe à 16,7% (soit une réduction de coût de 50,4% par rapport au coût initial). Notez que d'autres réductions de coût sont possibles en codant certains champs (par exemple, l'horloge logique et l'identifiant du train) sur un seul et même octet.

En ce qui concerne la consommation CPU, elle est inférieure à 9%. La petitesse de cette consommation confirme que la méthode[7] que nous avons employée pour limiter le nombre de recopies mémoire lors du traitement d'un train (cf. algorithme 13) assure correctement son rôle. Un *profiling* de notre code explique le pic à environ 9% quand la longueur des messages est de 10 octets. En effet, à chaque fois qu'un processus appelle la primitive d'uot-diffusion (cf. algorithme 2 page 59), l'instruction `append(wagonAEnvoyer.msgs,unMessage)` (correspondant à la ligne 2.1) effectue un `realloc` de la zone mémoire correspondant à `wagonAEnvoyer`. Quand les messages sont de petite taille, ces `realloc` deviennent nombreux et pèsent sur la CPU.

La table 6.3 a pour objectif de comparer le protocole des trains au protocole LCR. Les données de la colonne « H_{LCR} pratique » et donc les données de la colonne « écart trains pratique » qui les exploitent doivent être considérées avec une extrême prudence. Nous ne les fournissons qu'à titre indicatif, dans la mesure où elles ne correspondent pas à des tests que nous avons effectués sur les mêmes machines et sur le même réseau que pour nos tests du protocole des trains. Nous nous sommes contentés de reprendre les résultats présentés en figure 12 de [Guerraoui *et al.*, 2010], résultats que avons multipliés par 94/93 pour tenir compte du fait que le réseau présenté dans [Guerraoui *et al.*, 2010] offre un débit de 93 Mbps alors que notre réseau offre un débit de 94 Mbps. Donc, les données de la colonne « H_{LCR} pratique » ne tiennent pas compte de la différence de puissance des machines. Une perspective de cette thèse est mesurer les performances de LCR sur nos propres machines. Toutefois, nous disposons d'un autre moyen pour comparer, sans vigilance particulière, les performances pratiques du train et celle de LCR. En effet, nous pouvons calculer les valeurs de la colonne « H_{LCR} théorique » en appliquant le lemme 24 page 114. Comme les valeurs théoriques de débit sont toujours supérieures aux valeurs expérimentales[8], comparer $H_{trains_{pratique}}$ et $H_{LCR_{théorique}}$ nous fournit des informations sur les performances respectives de ces protocoles. En nous concentrant sur les mesures faites avec 10 trains circulant en parallèle, nous constatons que, pour des messages de 10 octets, l'écart entre $H_{trains_{pratique}}$ et $H_{LCR_{théorique}}$ est de 249,8%. Pour des messages de 100 (respectivement 1 000) octets, l'écart est de 33,2% (respectivement 1,0%). Par conséquent, dans 3 cas sur 4, nos mesures faites avec le protocole des trains sont meilleures que ce que nous pourrions espérer théoriquement (et donc pratiquement) avec le protocole LCR. Ainsi,

7. La description de cette méthode est en dehors du cadre de cette thèse.
8. La figure 5.3 page 115 permet de vérifier que $H_{LCR_{théorique}}$ est toujours supérieur à $H_{LCR_{pratique}}$.

TABLE 6.3 – Comparaison entre le protocole des trains (version multitrain) et LCR (les données des colonnes « H_{LCR} pratique » et « écart trains pratique » doivent être employées avec précaution, cf. texte)

nb proc.	nb trains	taille msg (octets)	H_{trains} pratique (Mbps)	H_{LCR} pratique (Mbps)	écart trains pratique	H_{LCR} théorique (Mbps)	écart trains théorique
5	5	10	73,8	-	-	21,8	239,3%
5	5	100	105,5	32,3	226,1%	81,6	29,3%
5	5	1000	110,3	87,9	25,4%	112,5	-2,0%
5	5	10000	110,5	112,2	-1,5%	117,0	-5,5%
5	10	10	76,1	-	-	21,8	249,8%
5	10	100	108,7	32,3	236,1%	81,6	33,2%
5	10	1000	113,6	87,9	29,2%	112,5	1,0%
5	10	10000	113,9	112,2	1,5%	117,0	-2,6%

nous confirmons expérimentalement le lemme 25 (page 115) qui démontre que le protocole des trains est plus performant en termes de débit que LCR. Il reste toutefois la contre-performance dans le cas de 10 trains circulant en parallèle et transportant des messages de 10 000 octets : l'écart est de $-2,6\%$. Pour l'instant, nous n'avons pas d'explication satisfaisante par rapport à cette exception.

Nos tests ont besoin d'être poursuivis afin de répondre aux questions suivantes : quel est le comportement du protocole avec deux *threads*, l'un pour la réception des messages et l'autre pour l'émission ? Quelle est la latence observée avec notre protocole ? Comment évoluent le débit et la latence en fonction du nombre de participants ? Par ailleurs, le lemme 21 page 111 montre que, théoriquement, plus un wagon contient de messages, meilleur est le débit. Dans la pratique, quelle est la taille optimale pour `wagonAEnvoyer` en fonction du nombre de participants ? De même, quel est le nombre optimal de trains en fonction du nombre de participants ?

6.1.6 Améliorations envisageables

Cette section présente des améliorations envisageables (mais non implantées) pour le protocole des trains, selon deux axes : 1) améliorations pour tenir compte du développement durable, et 2) améliorations pour exploiter les processeurs multicœurs.

Axe « prise en compte du développement durable ». Réduire les besoins en ressources matérielles des logiciels est aujourd'hui incontournable pour réduire leur consommation électrique. De ce fait, [Green Code Lab, 2012] propose des « patrons de conception verts » (*Green Patterns* en anglais) pour éco-concevoir les logiciels. Jusqu'à présent, pour le protocole des trains, nous nous sommes focalisés sur l'optimisation du débit. Mais, avec tous ces trains qui circulent en permanence, si le protocole transporte peu de messages, de nombreux trains qui circulent sont vides. Par conséquent, le protocole utilise inutilement des ressources CPU et réseau, donc de l'énergie.

Pour répondre à ce problème, une première amélioration consiste à ralentir la circulation

des trains. Quand un train a fait le tour du circuit, il reste un certain temps (la durée d'attente est un paramètre du protocole) au niveau d'un processus avant d'être renvoyé vers le successeur de ce processus. Aussi, pour éviter la mise en attente d'un train tandis que des messages s'accumulent sur les autres processus, nous proposons d'ajouter à l'entête de chaque train, un champ `urgentId` de type entier codé sur 1 octet. Alors, un processus p ne fait attendre un train que si le champ `urgentId` de ce train est égal au champ `urgentId` du train de même identifiant que p avait envoyé. p incrémente `urgentId` d'un train s'il constate, au moment où il envoie ce train, que sa couche applicative attend que `wagonAEnvoyer` soit vide pour confier à la couche intergicielle une nouvelle uot-diffusion (cf. section 6.1.3).

Dans le cas où, malgré cette première amélioration, des trains continuent à circuler à vide, une seconde amélioration est envisageable : réduire dynamiquement le nombre de trains sur le circuit. Pour ce faire, nous proposons d'ajouter un champ `nombreTrains` dans l'entête du train. Quand le processus p_0 qui est premier sur le circuit des trains constate que des trains circulent à vide, il décrémente ce champ `nombreTrains` au niveau du train d'identifiant 0, puis au niveau des trains ayant d'autres identifiants. Quand p_0 reçoit à nouveau le train d'identifiant 0, il sait que tous les autres processus ont été informés de sa volonté de réduire le nombre de trains. p_0 retire le train de plus grand identifiant de la circulation. Avec le même mécanisme, p_0 peut injecter de nouveaux trains.

Intéressons-nous maintenant à l'autre axe.

Axe « exploitation des processeurs multicœurs ». Cet axe est justifié dans le cas où le réseau offre un débit suffisamment élevé pour que les processus n'arrivent pas à construire assez vite des trains pour que le réseau transporte des trains à chaque intervalle de temps. Dans ce cas, s'il y a n processus sur le circuit des trains, il faut faire tourner $k \times n$ trains sur le circuit, où k est le plus grand entier tel que : 1) k est inférieur ou égal au nombre de cœurs sur les processeurs des machines hébergeant chaque processus, et 2) le réseau arrive à écouler les $k \times n$ trains. Quand un processus reçoit un train, il le fait traiter par un des cœurs disponibles dans le processeur. De ce fait, sur un même processeur (et pour le compte d'un même processus), k cœurs exécutent en parallèle les algorithmes 12 et 13, mais sur des trains différents. Donc, il faut faire évoluer ces algorithmes pour prendre en compte les problèmes de synchronisation dus notamment au fait que les trains doivent quitter chaque processus dans l'ordre dans lequel ils sont arrivés. Cette évolution est une autre perspective de cette thèse.

6.2 Gestion de groupes applicatifs

Le protocole des trains permet à un participant au protocole d'envoyer des uot-diffusions à l'ensemble des participants au protocole. Par conséquent, notre protocole fait partie des protocoles à groupe (de participants) fermé [Chockler *et al.*, 2001, Défago *et al.*, 2004] : les processus qui ne participent pas au protocole ne peuvent pas faire d'uot-diffusions aux participants au protocole. Cette sémantique convient aux besoins de réplication d'un système de contrôle-commande et de supervision comme le *P3200* (cf. chapitre 2). Par ailleurs, dans le cas du protocole des trains, la gestion du circuit des trains garantit la connaissance des participants au protocole et l'intégration des messages d'arrivée ou départ

de processus dans le flot des uot-livraisons. Toutefois, ce niveau de service est insuffisant pour permettre son intégration dans le *P3200*. En effet, quand une réplique de *P3200* démarre, cette réplique doit récupérer l'état d'une autre réplique de manière synchronisée avec le flux des uot-livraisons. Cela nous conduit à gérer des groupes applicatifs, de manière à disposer de la notion de transfert d'état. Pour que le *P3200* puisse utiliser indifféremment l'intergiciel Isis [Birman, 2010] ou bien l'intergiciel basé sur le protocole des trains, nous avons décidé que l'interface de programmation (API) fournie au *P3200* soit similaire à l'API d'Isis [Birman, 1993].

La section 6.2.1 détaille cette interface. Puis, la section 6.2.2 décrit la mise en œuvre de la notion de groupes de processus. Ensuite, la section 6.2.3 présente la réalisation du transfert d'état. Enfin, la section 6.2.4 présente l'API plus simple qui a été implantée pour les tests de performance du protocole des trains.

6.2.1 Interface de programmation du protocole des trains

Pour le système de contrôle-commande et de supervision *P3200*, les primitives suivantes (inspirées de celles d'Isis [Birman, 1993]) sont implantées.

`uto_remote_init` permet d'initialiser l'intergiciel basé sur le protocole des trains. Cette primitive permet notamment d'initialiser le protocole des trains (en déclenchant l'insertion du processus courant dans le circuit des trains).

`uto_entry` permet d'associer un numéro applicatif à une fonction f_{entry}. Ainsi, quand un message m ayant ce numéro applicatif est uot-livré par le protocole des trains, f_{entry} est appelée avec m en paramètre.

`uto_pg_join` permet de rejoindre un groupe applicatif dont le nom est fourni en paramètre. Cette fonction permet de préciser 4 paramètres optionnels utilisés par le module de gestion de groupes (cf. section 6.2.2) :

- une fonction $f_{monitor}$ que le module de gestion de groupes appelle à chaque fois qu'il y a un changement dans le nombre de membres du groupe ;
- une fonction f_{init} que le module de gestion de groupes appelle s'il constate que le processus qui exécute `uto_pg_join` est le premier à rejoindre le groupe ;
- une fonction $f_{xferout}$ appelée quand un processus p rejoint le groupe et que le module de gestion de groupes souhaite que ce soit le processus courant qui envoie les données caractéristiques de l'état du groupe ;
- une fonction f_{xferin} appelée quand un processus p rejoint le groupe et que le module de gestion de groupes souhaite qu'il reçoive les données caractéristiques de l'état du groupe.

`uto_pg_leave` permet de quitter un groupe applicatif dont le processus est membre.

`uto_xfer_out` est une fonction appelée à partir de $f_{xferout}$ pour envoyer une (ou plusieurs) donnée(s) caractéristique(s) de l'état du groupe.

`uto_abcast_l` permet d'uot-diffuser un message m sur un groupe applicatif donné en lui associant un numéro applicatif qui permettra de déterminer la fonction f_{entry} à appeler pour uot-livrer m.

6.2.2 Mise en œuvre de la notion de groupes

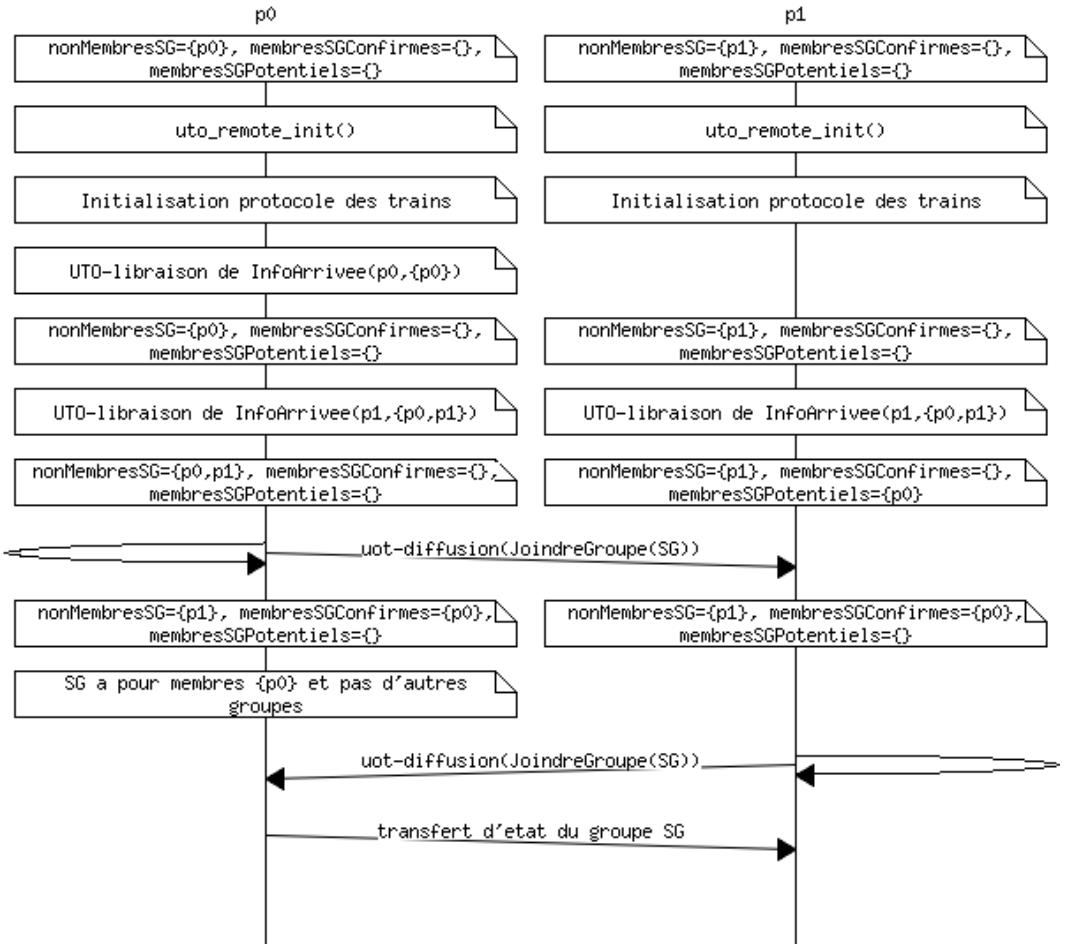
Cette section présente comment la notion de groupe a été implantée au dessus du protocole des trains. Compte tenu du nombre limité de processus participant au protocole des trains, nous pouvons mettre en œuvre cette notion à l'aide de l'approche présentée par [Défago *et al.*, 2004] :

1. former un super-groupe (noté SG dans la suite) qui est l'union de tous les groupes sur lesquels au moins un processus a exécuté `uto_pg_join` ;
2. quand un processus exécute `uto_abcast_l`, faire une uot-diffusion sur SG ;
3. quand un message est uot-livré, ne pas appeler la fonction f_{entry} associée si le processus n'est pas membre du groupe destinataire de ce message (le message est donc ignoré).

Si cette approche simplifie les développements, elle ne nous semble pas aussi triviale que le laisse entendre le titre de la section « *Minimality and Trivial Solutions* » de [Défago *et al.*, 2004] dans laquelle est présentée cette approche. En effet, quand un processus p démarre, il a besoin de récupérer la structure de données, notée `LG` dans la suite, contenant la liste des groupes qui ont déjà été créés et, pour chaque groupe créé, la liste de ses membres. Ainsi, par la suite, lorsque p invoque `uto_pg_join()` sur un groupe g donné, il lui suffit d'uot-diffuser un message `JoindreGroupe` ayant g et p en paramètres. Sur uot-livraison de ce message `JoindreGroupe`, un processus p' cherche g dans sa liste de groupes. Étudions le cas où $membres(g) \neq [\ ]$. Si p' est différent de p, que p' appartient à g et que p' est le premier membre de g, p' appelle $f_{xferout}$ pour envoyer l'état de g à p. Par ailleurs, si p' est p, p attend de recevoir l'état du groupe d'un autre processus via des appels à f_{xferin}. Intéressons-nous maintenant au cas $membres(g) = [\ ]$. Si p' est différent de p, p' ajoute g à sa liste de groupes et note que p en est l'unique membre. Par ailleurs, si p' est p, p ajoute g à sa liste de groupes, note que p en est l'unique membre et appelle f_{init}. Tout dépend donc de la récupération de `LG` par p. La méthode la plus simple pour ce faire consiste à : 1) définir des fonctions $f_{init_{SG}}$, $f_{xferout_{SG}}$ et $f_{xferin_{SG}}$ et 2) faire uot-diffuser par p un message `JoindreGroupe(`SG`,`p`)`. En effet, si p connaît $membres(SG)$ au moment où la couche intergicielle lui livre `JoindreGroupe(`SG`,`p`)`, il peut appliquer les algorithmes décrits précédemment. Dans le cas du protocole des trains, le calcul de $membres(SG)$ résulte de l'uot-livraison des messages `InfoArrivée` contenant l'identifiant du processus qui vient d'arriver et le circuit des trains au moment de son arrivée (cf. lignes 5 et 43 de l'algorithme 10). En effet, tout processus qui exécute `uto_remote_init()` exploite cette information selon l'algorithme 17 qui garantit qu'il n'y a pas d'interblocages entre 2 processus qui chercheraient à rejoindre SG simultanément. Nous illustrons le fonctionnement de cet algorithme à travers l'exemple de la la figure 6.6. Deux processus p_0 et p_1 démarrent simultanément, alors que le circuit des trains est vide. Supposons que c'est p_0 qui passe par l'état `DansCircuitSeul` (cf. algorithme 11). Le protocole des trains uot-livre le message `InfoArrivée(`$p_0, [p_0]$`)` à p_0. Ce dernier ajoute donc son adresse à `nonMembresSG` (lignes 17.11–12). Avant que p_0 n'ait eu le temps d'uot-diffuser son `JoindreGroupe(`SG`,`p_0`)` (lignes 17.16–18), p_1 rejoint le circuit des trains. Le message `InfoArrivée(`$p_1, [p_0, p_1]$`)` est donc uot-livré à p_0 et p_1. Grâce aux lignes 17.11–12, la variable `membresSGPotentiels` vaut $\varnothing$ au niveau de p_0, alors qu'elle est égale à $[p_0]$ au niveau de p_1. Supposons qu'avant que p_1 n'ait eu le temps d'uot-diffuser son `JoindreGroupe(`SG`,`p_1`)` (lignes 17.16–18), le

message `JoindreGroupe(`SG`,`p_0`)` est uot-livré à p_0 et p_1. Alors, p_0 considère qu'il est le seul membre de SG tandis que p_1 transfère p_0 de `membresSGPotentiels` à `membresSGConfirmés` (lignes 17.23–28). Par conséquent, p_0 quitte la boucle des lignes 17.8–32. De plus, comme p_0 est le seul membre de `membresSGConfirmés`, p_0 invoque la fonction $f_{init_{SG}}()$ (lignes 17.33–34). Quand le message `JoindreGroupe(`SG`,`p_1`)` est uot-livré à p_0 et p_1, p_1 transfère p_1 de `membresSGPotentiels` à `membresSGConfirmés` (lignes 17.23–28). Par conséquent, p_1 quitte la boucle des lignes 17.8–32. De plus, comme p_1 n'est pas le seul membre de `membresSGConfirmés` (lignes 17.33 et 17.35), p_1 attend qu'un autre membre du groupe lui envoie l'état du groupe en se connectant à p_1 et en lui communiquant des messages de transfert d'état (ligne 17.36 et cf. section 6.2.3 pour des détails sur le transfert d'état). Quand à p_0, la livraison du message `JoindreGroupe(`SG`,`p_1`)` déclenche le transfert d'état du groupe SG (lignes d'un algorithme non présenté dans cette thèse, mais évoqué dans la section 6.2.3). Notez que, malgré le démarrage simultané de p_0 et p_1, il n'y a aucun interblocage entre p_0 et p_1. De plus, ce raisonnement est généralisable à n processus.

FIGURE 6.6 – Diagramme de séquence de la gestion du super-groupe en cas de démarrage simultané de deux processus



Algorithme 17 Procédure `uto_remote_init()`

```
1: // Liste des participants dont le processus sait qu'ils n'appartiennent pas à SG
2: variable locale: nonMembresSG ← [myAddress]
3: // Liste des participants dont le processus sait qu'ils appartiennent à SG
4: variable locale: membresSGConfirmés ← [ ]
5: // Liste des participants dont le processus pense qu'ils pourraient appartenir à SG
6: variable locale: membresSGPotentiels ← [ ]
7: lancerThreadChargéExécuter_algorithmes_9_puis_10()
8: repeat // Le processus attend d'être dans membresSGConfirmés
9:   msg ← attenteUOTLivraison()
10:  switch (msg.type)
11:  case InfoArrivée:
12:    append(nonMembresSG,msg.InfoArrivée.procArrivé)
13:    for all p ∈ msg.InfoArrivée.circuit \ membresSGConfirmés \
       nonMembresSG \ {msg.InfoArrivée.procArrivé} do
14:      append(membresSGPotentiels,p)
15:    end for
16:    if msg.InfoArrivée.procArrivé == myAddress then
17:      utoDiffusion(JoindreGroupe(SG,myAddress))
18:    end if
19:  case InfoDépart:
20:    remove(nonMembresSG,msg.JoindreGroupe.procParti)
21:    remove(membresSGConfirmés,msg.JoindreGroupe.procParti)
22:    remove(membresSGPotentiels,msg.JoindreGroupe.procParti)
23:  case JoindreGroupe:
24:    if msg.JoindreGroupe.groupe == SG then
25:      remove(nonMembresSG,msg.JoindreGroupe.procJoint)
26:      remove(membresSGPotentiels,msg.JoindreGroupe.procJoint)
27:      append(membresSGConfirmés,msg.JoindreGroupe.procJoint)
28:    end if
29:  case Simpleuot-livraison:
30:    // Ce message est ignoré
31:  end switch
32: until myAddress ∈ membresSGConfirmés
33: if membresSGConfirmés possède un seul membre then
34:   f_init_SG()
35: else
36:   Attendre que membres(SG) ait été reçu via l'appel à f_xferin_SG
37: end if
```

6.2.3 Mise en œuvre du transfert d'état

Cette section détaille la mise en œuvre du transfert d'état. Nous avons vu à la section précédente que, quand un processus p souhaite rejoindre un groupe, il uot-diffuse un message `JoindreGroupe(nomGroupe,`p`)`. Sur uot-livraison de ce message, l'un des processus membres (noté p_m dans la suite) envoie l'état du groupe à p. Pour ce faire, p_m établit une connexion TCP dédiée entre p_m et p. Sur cette connexion, p_m envoie un message `DébutXFER(nomGroupe)`, puis autant de messages `SuiteXFER` que la fonction $f_{xferout}$ ne fait d'appels à `uto_xfer_out()`, avant d'envoyer le message `FinXFER`.

Deux stratégies sont possibles concernant les autres processus membres du groupe. La première stratégie consiste à ce que ces autres processus (notés p_a dans la suite) attendent que le transfert d'état entre p_m et p soit terminé avant de traiter l'uot-livraison qui suit le message `JoindreGroupe(nomGroupe,`p`)`. Ainsi, dans le cas où p_m s'arrête (volontairement ou non) avant d'avoir transmis tout l'état du groupe à p, p peut solliciter un des p_a pour qu'il lui transfère l'état. Cette stratégie présente l'inconvénient de bloquer l'ensemble des processus membres du groupe pendant le transfert d'état. Dans le cas du système de contrôle-commande et de supervision P3200, le transfert d'état peut nécessiter jusqu'à 1 minute. Or, il n'est pas acceptable de bloquer l'ensemble des répliques du P3200 pendant 1 minute. C'est pourquoi nous avons choisi la seconde stratégie.

La seconde stratégie consiste à ce que les processus p_a traitent les autres uot-livraisons sans attendre la fin du transfert d'état. L'ensemble des répliques n'est donc plus gelé pendant un transfert d'état. En revanche, les p_a considèrent que p est membre du groupe alors que le transfert d'état n'est pas terminé. De ce fait, si p_m s'arrête avant la fin du transfert, p doit uot-diffuser un nouveau message `JoindreGroupe(nomGroupe,`p`)` pour que l'un des p_a envoie à p l'état du groupe. Or, entre les deux messages `JoindreGroupe(nomGroupe,`p`)` de p, les processus p_a considèrent que p est membre du groupe. Par conséquent, si nous utilisons ce groupe pour répartir une charge de traitement entre les membres du groupes, une certaine charge a été envoyée à p qui l'a ignorée, vu qu'il ne se considérait pas encore membre du groupe. Dans le cas du P3200, nous n'avons pas ce problème puisque nous n'utilisons les groupes que pour faire de la réplication. La seconde stratégie reste donc préférable à la première.

6.2.4 Interface de programmation pour les tests de performance

Pour les tests de performance du protocole des trains, nous avons mis en œuvre une API plus simple (et donc plus performante). Voici les primitives implantées.

`tr_init` initialise l'intergiciel basé sur le protocole des trains. Elle prend en paramètre deux fonctions :

- $f_{callbackCircuitChange}$ qui est appelée à chaque fois qu'il y a un changement dans le circuit des trains (arrivée ou départ de processus),
- $f_{callbackUtoDeliver}$ qui est appelée à chaque fois qu'un message est uot-livré par l'intergiciel.

`uto_broadcast` permet de demander l'uot-diffusion d'un message.

`tr_terminate` permet d'arrêter l'intergiciel.

6.3 Mémoire répartie partagée basée sur de l'uot-diffusion

Dans cette section, nous détaillons la mémoire répartie partagée implantée sur un protocole d'uot-diffusion (qui n'est pas forcément le protocole des trains). Après une présentation générale (cf. section 6.3.1) et une présentation des contraintes liées à la cohérence (cf. section 6.3.2), nous décrivons l'intégration de diffusions fiables et ordonnées provenant d'entités ne participant pas au protocole (cf. section 6.3.3). Nous commentons ensuite les mécanismes présentés (cf. section 6.3.4) avant de conclure (cf. section 6.3.5).

6.3.1 Présentation générale

Tout au long de cette section 6.3, nous considérons une application codée à l'aide d'un langage orienté objet. La mémoire de l'application est donc constituée d'objets (cf. ronds et double-ronds dans la figure 6.7). Ce langage, Objective-C dans le cas du *P3200*, offre la propriété d'attachement dynamique (*dynamic binding* en anglais) : lorsqu'un objet B invoque une méthode m sur un objet C (noté m_{BC} dans la figure 6.7), le code à exécuter est déterminé par une procédure, notée `Messagerie` dans la suite. Pour alléger la figure 6.7, seul le traitement de m_{CE} par `Messagerie` est représenté. Mais, en fait, toutes les invocations de méthodes sont traitées par `Messagerie`. L'invocation d'une méthode sur un objet (par exemple, l'invocation de m_{BC} par B sur C dans la figure 6.7) peut entraîner l'invocation de méthode(s) sur un ou plusieurs objets (par exemple, m_{CD} et m_{CE} dans la figure 6.7). En cas d'arrêt volontaire ou non du processus qui exécute l'application et donc héberge tous ces objets, l'état de chacun de ces objets, défini par les valeurs stockées dans ses variables d'instance, est perdu. Comme la perte de données de l'état de certains objets est préjudiciable à l'application, ces objets sont répliqués sur d'autres processus (cf. double-ronds dans la figure 6.7). Par conséquent, nous pouvons considérer qu'ils font partie d'une même mémoire répartie partagée.

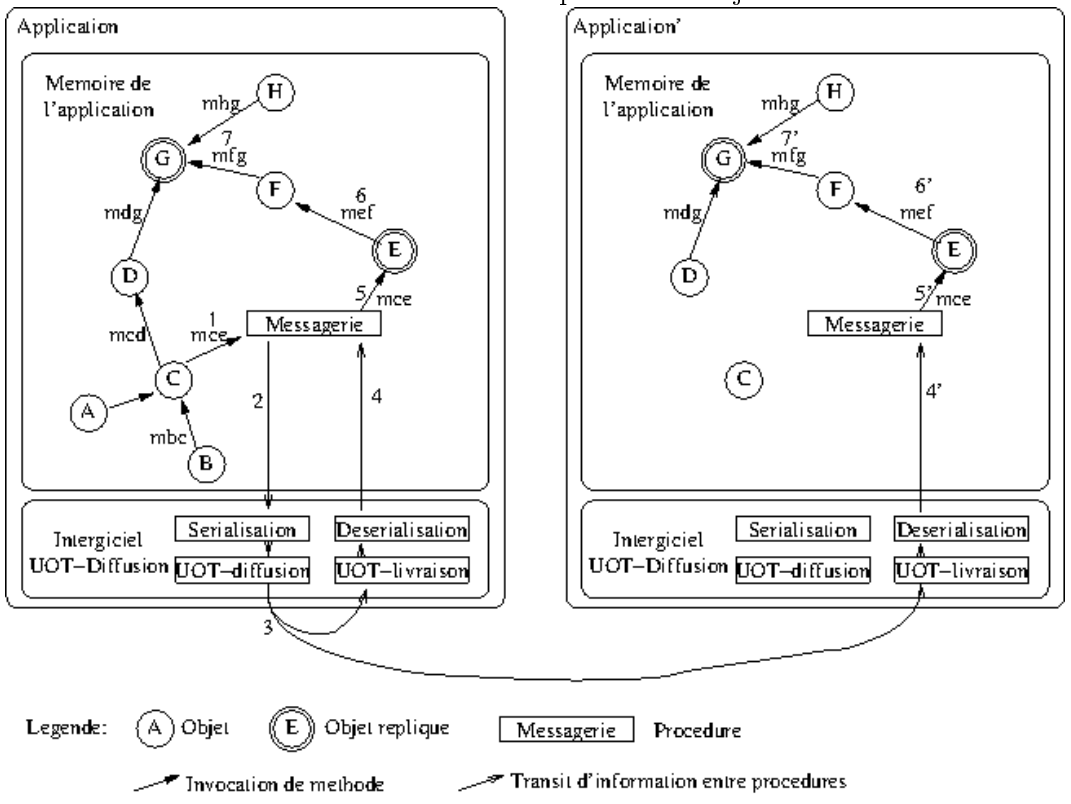
Dans cette mémoire, toute lecture de l'état d'un objet répliqué est faite localement sur le processus qui a besoin de faire cette lecture. Toute mise à jour sur un objet répliqué est propagée sur l'ensemble des processus à l'aide d'une uot-diffusion de cette mise à jour.

Pour réaliser cette propagation, nous exploitons la propriété d'attachement dynamique du langage[9] [Eychenne *et al.*, 1992, Baradel *et al.*, 1996] : quand un objet C invoque une méthode m_{CE} sur un objet répliqué E (cf. étape 1 dans la figure 6.7), `Messagerie` est invoquée afin de déterminer la méthode correspondant à m_{CE} pour E. Au niveau de `Messagerie`, nous mettons en place un intercepteur. Ce dernier vérifie que l'objet sur lequel est invoqué la méthode est un objet répliqué. Pour ce faire, l'intercepteur teste un bit dédié dans les variables d'instance de l'objet. Si l'objet est répliqué, l'intercepteur vérifie que la méthode est une méthode qui modifie l'état de E. Pour ce faire, il consulte une structure de données `selectorTable` qui comprend l'ensemble des méthodes modifiantes de chaque classe.

Si ces deux conditions sont vérifiées, alors l'intercepteur prépare l'uot-diffusion de l'invocation de méthode (étape 2 de la figure 6.7). Il consulte une structure de données

9. Mes collègues de travail ont conçu la solution. Je suis le contributeur principal concernant son implantation.

FIGURE 6.7 – Mise en œuvre de la réplication d'objet avec de l'UTDN



`objectTable` pour déterminer l'identifiant global de cet objet. Puis, il récupère dans `selectorTable` l'identifiant global de la méthode et les instructions de sérialisation des paramètres de cette méthode. L'intercepteur construit alors un message contenant les deux identifiants globaux et les paramètres sérialisés, et uot-diffuse ce message. `Messagerie` redonne le contrôle à C comme si elle avait effectivement traité l'appel à m_{CE}.

In fine (étape 3 de la figure 6.7), le message est uot-livré sur l'ensemble des processus.

Sur chaque processus (étape 4 et 4' de la figure 6.7), le *thread* th chargé de traiter les messages uot-livrés lit dans ce message les deux identifiants globaux. Avec l'identifiant global d'objet, en consultant `objectTable`, th retrouve ou non le pointeur sur l'objet. S'il ne le retrouve pas, cela signifie simplement que ce processus n'est pas une copie conforme du processus qui a uot-diffusé le message [10] : th se contente d'ignorer le message. S'il le retrouve, avec l'identifiant global de la méthode et en consultant `selectorTable`, th cherche le pointeur sur le code de la méthode et les instructions de désérialisation des paramètres de cette méthode. Il les désérialise, positionne à vrai le drapeau (*Flag* en anglais) de Réentrance `FR` (`FR` est une variable booléenne globale au processus dont le rôle est précisé ci-dessous), puis invoque la `Messagerie` Objective-C avec le pointeur sur l'objet, le pointeur sur la méthode, et les paramètres de la méthode.

Sur chaque processus (étape 5 et 5' de la figure 6.7), comme `FR` vaut vrai, `Messagerie` ne teste pas à nouveau si l'objet destinataire est un objet répliqué et si sa méthode est modifiante : `Messagerie` invoque effectivement la méthode m_{CE} sur E.

Ensuite (étape 6 et 6' de la figure 6.7), E (respectivement E') invoque m_{EF} sur F (respectivement F'). `FR` valant vrai, la `Messagerie` Objective-C invoque effectivement cette méthode.

Enfin (étape 7 et 7' de la figure 6.7), F (respectivement F') invoque m_{FG} sur G (respectivement G'). `FR` valant vrai, la `Messagerie` Objective-C invoque effectivement cette méthode. Le traitement de l'uot-livraison correspondant à m_{CE} est terminé : th repositionne `FR` à faux.

Remarquez que le booléen `FR` nous a permis d'économiser une uot-diffusion. En effet, si nous supposons que G est un objet répliqué et que m_{FG} est une méthode modifiante, la `Messagerie` aurait dû intercepter cette invocation de méthode et l'uot-diffuser. Mais, comme `FR` valait vrai, la `Messagerie` ne l'a pas fait, ce qui est correct puisque chaque processus était en train d'appliquer la méthode m_{FG} sur sa propre réplique de G.

6.3.2 Contraintes liées à la cohérence

Le mécanisme présenté à la section 6.3.1 est orthogonal à l'application. Et les conséquences sur le code de cette dernière semblent limitées à la modification de la `Messagerie` et l'ajout d'un bit au niveau de la classe mère de toutes les classes. Toutefois, l'application doit prendre en compte trois contraintes afin que la cohérence des mises à jour de la mémoire répartie partagée soit préservée.

La première contrainte est qu'un seul *thread* doit s'occuper du traitement des uot-

10. Dans le cas du *P3200*, cela peut arriver quand le système de contrôle-commande et de supervision s'exécute en présence du configurateur du système : le configurateur n'est pas une copie conforme.

livraisons d'invocations de méthode. En particulier, ce *thread* ne doit pas créer un *thread* par uot-livraison en vue de paralléliser le traitement des uot-livraisons (et profiter, par exemple, de la présence de plusieurs cœurs sur le processeur exécutant l'application). En effet, si au moins 2 *threads* traitent les uot-livraisons, il se pourrait que, sur un processus, 2 méthodes uot-livrées dans un certain ordre soient traitées dans cet ordre, alors qu'elles sont traitées dans l'ordre inverse sur d'autres processus.

La deuxième contrainte est que le *thread* th chargé de gérer les uot-livraisons d'invocations de méthodes peut être interrompu par un *thread* th' plus prioritaire ou bien par une routine d'interruption ri. Or, cette interruption peut avoir lieu après que th a positionné `FR` à vrai. Dans le cas où th' ou ri invoque une méthode modifiante sur un objet répliqué (par exemple, H invoque m_{HG} sur G dans la figure 6.7). Comme `FR` vaut vrai, l'intercepteur de la `Messagerie` ne détourne l'invocation de cette méthode modifiante : G change d'état, sans que les autres processus en soient avertis. Dans le cas de *threads*, nous pouvons résoudre ce problème en définissant `FR` comme une variable globale spécifique à chaque *thread* (en utilisant par exemple les primitives `pthread_key_create`, `pthread_setspecific`, etc. dans le cas de la librairie *pthread*). Quand th' prend la main, la librairie de gestion de *threads* stocke dans `FR` une valeur spécifique à th'. Ainsi, le fait que, pour th, `FR` vaut vrai n'a pas d'influence sur th'. En ce qui concerne les interruptions, le code de la fonction appelée par chaque interruption doit commencer par sauvegarder la valeur de `FR`, positionner `FR` à faux, traiter l'interruption, et enfin restaurer la valeur de `FR`. Les répercussions sur le code applicatif sont un peu plus importantes qu'avec la première contrainte, mais restent localisées.

La troisième contrainte est liée au mélange possible d'opérations de consultation et de mises à jour. Considérons la figure 6.7 et supposons que B invoque m_{BC} sur C. C n'étant pas un objet répliqué, l'intercepteur de la `Messagerie` ne détourne pas cette invocation. La méthode m_{BC} s'exécute. Elle invoque tout d'abord m_{CE} sur E, invocation uot-diffusée, et ensuite, m_{CD} sur D. Cette dernière invocation résulte en l'appel de m_{DG} sur G. Supposons que G est un objet répliqué et que m_{DG} le met à jour. Alors, l'intercepteur de la `Messagerie` uot-diffuse m_{DG}. En version non répliquée de l'application, l'invocation de m_{BC} sur C et ses conséquences (m_{CE} sur E et m_{DG} sur G) sont traitées au sein d'un même *thread* th. De ce fait, si une méthode de consultation est faite sur G (par exemple, m_{HG}) par un autre *thread* de même priorité que th, nous avons la garantie que, soit m_{HG} est exécuté avant m_{BC}, soit après m_{BC} et tous les appels de méthodes qui en ont résulté (m_{CD}, m_{DG}, m_{CE}, etc). Avec l'interception et l'uot-diffusion de méthodes modifiantes, nous n'avons plus cette garantie : G peut recevoir m_{HG} en n'ayant reçu que m_{DG} ou bien m_{FG} (conséquence de m_{CE}). Dit autrement, H peut avoir une vision incohérente de G. Pour répondre à cette contrainte, il faut analyser l'arborescence du code pour analyser s'il y a besoin que : 1) des objets deviennent répliqués (C dans l'exemple), et 2) des méthodes soient considérées comme des mises à jour même si elles n'en sont pas (m_{BC} dans l'exemple).

6.3.3 Intégration de diffusions fiables et ordonnées provenant d'entités ne participant pas à l'uot-diffusion

Dans le cas de l'application *P3200*, nous avons une autre contrainte (généralisable à d'autres applications). En effet, en tant que système de contrôle-commande et de super-

vision, le *P3200* reçoit des informations d'automates industriels qui lui envoient l'état de capteurs auxquels ils sont reliés. Pour ce faire, ces automates diffusent, de manière fiable et ordonnée, leurs informations aux n répliques R_i de *P3200*. « Fiable » signifie que, pour chaque automate, chaque R_i reçoit tous les messages de l'automate (il n'y a pas de perte). « Ordonné » signifie que, pour chaque automate, chaque R_i reçoit tous les messages de cet automate dans le même ordre.

Ces diffusions fiables et ordonnées introduisent un problème pour notre mémoire répartie partagée. En effet, supposons que la réception d'un message msg d'un certain automate A entraîne l'appel d'une méthode m sur un objet obj. De plus, supposons que obj est un objet répliqué sur tous les R_i. Ces hypothèses sont illustrées dans la figure 6.8 qui intègre le contenu de la figure 2.1 page 28 et celui de la figure 6.7 page 148. L'automate A (nommé PLC A dans la figure selon l'acronyme de l'anglais *Programmable Logic Controller*) reçoit une certaine information d'un des capteurs qui lui est connecté, information issue d'un certain événement au niveau du procédé industriel. L'automate A traduit cette information en le message msg qu'il diffuse de manière fiable et ordonnée (à travers le bus de terrain, la passerelle et le réseau Ethernet) aux deux répliques de *P3200* R_0 et R_1. Sur chacune de ces répliques R_i, msg est reçu par l'objet $I/FPLC$ chargé de faire l'interface avec l'automate A. Cet objet invoque alors la méthode m sur obj. Décrivons maintenant la suite du traitement qui n'est pas représentée sur la figure afin de ne pas l'alourdir. Quand chaque R_i reçoit msg, R_i invoque m sur obj. Donc, sur chaque R_i, `Messagerie` déclenche l'uot-diffusion de cette invocation. De ce fait, m est uot-diffusée n fois : cela induit un problème de surconsommation de ressources réseau. De plus, au niveau de chaque R_i, m est invoquée n fois sur obj (alors que msg n'a été reçu qu'une fois). Cela peut introduire des erreurs dans l'état d'obj. Par exemple, considérons qu'obj est destiné à mémoriser les 10 dernières valeurs reçues dans les messages de l'automate et que m est la méthode qui permet d'ajouter une nouvelle valeur. Comme m est invoqué n fois, obj considère à tort qu'il y a eu n messages contenant une même valeur envoyée par l'automate.

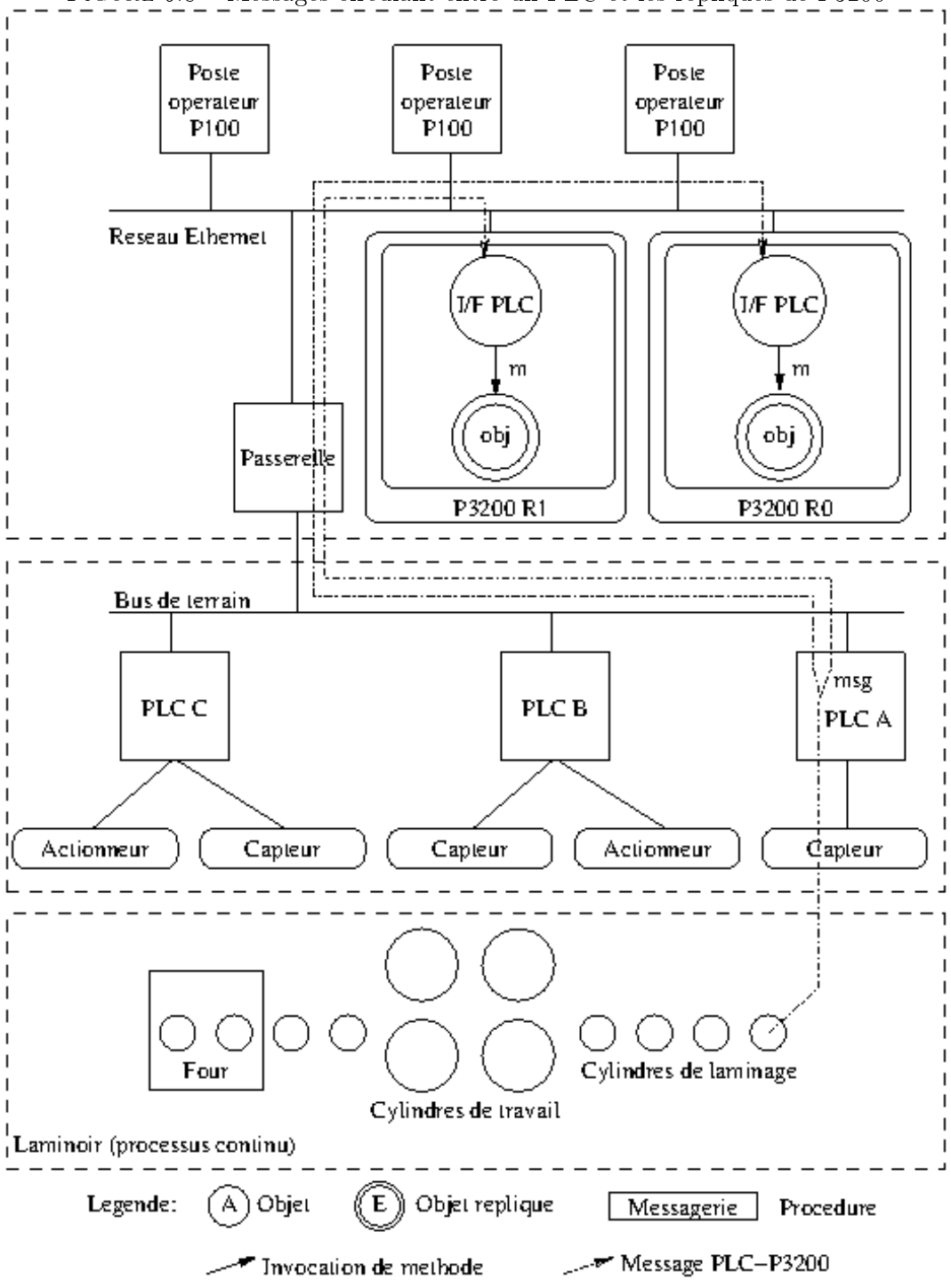
La solution la plus simple serait que les automates uot-diffusent leurs mises à jour. Elle n'est pas réaliste, car : 1) elle supposerait d'ajouter du code à des automates industriels déjà très contraints en place mémoire ; 2) elle ajouterait plusieurs dizaines, voire centaines, de participants au protocole d'uot-diffusion, ce qui est incompatible avec les hypothèses de notre système.

Une solution plus réaliste consisterait à connecter les automates à une seule des répliques du *P3200*. Cette dernière, notée R_0 se chargerait d'uot-diffuser les mises à jour en provenance des automates. Cette solution fonctionne, mais présente deux inconvénients en cas de défaillance de R_0. Tout d'abord, les messages venant des automates que R_0 n'a pas eu le temps d'uot-diffuser avant sa défaillance sont perdus. De plus, pendant le temps qu'une des autres répliques prenne la relève de R_0 dans la connexion aux automates, les messages que les automates auraient transmis à R_0 s'il n'avait pas défailli ne sont transmis à aucune réplique. Les données que contenaient ces messages sont perdues.

C'est pourquoi nous avons adopté la méthode suivante [11] [Simatic *et al.*, 1998] : tous les automates diffusent de manière fiable et ordonnée [12] leurs messages aux processus qui

11. Cette méthode s'appuie sur une stratégie de réplication semi-active.
12. La diffusion fiable et ordonnée des automates présentée en début de cette section ne doit pas être

FIGURE 6.8 – Messages circulant entre un PLC et les répliques de P3200



se sont abonnés. Sur réception d'un message msg d'un automate A, toutes les répliques de *P3200* enregistrent ce message dans une liste L_A spécifique à chaque automate, sauf une réplique R_{r_A} « relais » de cet automate. En effet, R_{r_A} uot-diffuse cette donnée. Notez qu'une réplique peut être relais de plusieurs automates, mais qu'une seule réplique peut être le relais d'un automate donné. Sur réception de l'uot-livraison associée, toutes les répliques hormis R_{r_A} retire msg de L_A. En cas d'arrêt de R_{r_A} (notifié par l'invocation de la *callback* $f_{monitor}$, cf. section 6.2.1), la nouvelle réplique relais se charge d'uot-diffuser tous les messages qui sont encore dans L_A : aucune donnée de l'automate n'est perdue. Nous devons gérer les cas où un processus non-relais uot-livre msg avant d'avoir reçu msg de l'automate. Une première solution consiste à associer à L_A une liste L'_A (spécifique à l'automate A) des msg uot-livrés alors que L_A était vide. Ainsi, lorsque le processus p reçoit par la suite des messages directement de l'automate, p ne stocke pas à tort ces messages dans L_A (cf. algorithme 18). Cette solution n'est pas optimale, puisque L'_A ne sert qu'à compter l'avance prise par les uot-livraisons sur les réceptions directes de messages venant de l'automate. C'est pourquoi nous proposons l'utilisation d'un compteur $\texttt{nbUOTLivraisonsAttendues}_A$ au lieu de L'_A (cf. algorithme 19). Ainsi, un message n'est stocké dans L_A que si nécessaire et il n'y a plus besoin d'ajouter/retirer inutilement des messages dans L'_A. De plus, s'il y a besoin de retirer un message msg de L_A, il n'y a pas besoin de chercher msg dans L_A. En effet, msg est forcément le premier message de L_A à partir du moment où $\texttt{nbUOTLivraisonsAttendues}_A$ et L_A ont été correctement initialisés. Le principe de cette initialisation (pour laquelle nous ne présentons aucun algorithme) est le suivant. Quand une réplique R_i démarre et qu'elle reçoit un message automate en provenance directe de A (respectivement trouvé dans une uot-diffusion), R_i l'ajoute à L_A (respectivement L'_A, une liste qui ne sert désormais qu'au moment de l'initialisation). Quand R_i trouve le même message dans L_A et L'_A, R_i calcule $\texttt{nbUOTLivraisonsAttendues}_A$, supprime tous les messages de L_A qui sont dans L'_A, exécute `Messagerie(msg)` pour tout $\texttt{msg} \in L'_A$, et enfin supprime L'_A.

6.3.4 Discussion

Le mécanisme de réplication d'objets que nous avons implanté peut être assimilé à un « *lazy cache algorithm* » [Afek *et al.*, 1993] : quand un processus p a besoin d'accéder en lecture au contenu d'un objet o, il lui suffit de consulter directement l'état de o (car p détient sa valeur dans sa mémoire qui peut être qualifiée de « cache » de p). En revanche, quand p a besoin d'invoquer une méthode sur o et que cette méthode modifie l'état de p, p diffuse cette mise à jour pour que toutes les répliques mettent à jour la valeur de o dans leur propre cache. Selon la terminologie d'autres auteurs, notre travail peut être assimilé à de la diffusion d'écriture (*write broadcast*) [Dubois *et al.*, 1986], sauf que nous diffusons des modification d'objets et non des pages mémoire après modification.

confondue avec l'uot-diffusion de la solution la plus simple évoquée précédemment. En effet, si deux processus p_0 et p_1 sont abonnés aux diffusions fiables et ordonnées de deux automates A_0 et A_1, le caractère fiable et ordonné des diffusions garantit que les informations issues de A_0 (respectivement A_1) sont reçues dans le même ordre au niveau de p_0 et p_1. En revanche, parce que les automates n'uot-diffusent pas leurs messages, les flux de diffusions issus de A_0 et A_1 peuvent se mélanger différemment au niveau de p_0 et p_1. Par exemple, supposons qu'A_0 diffuse les messages a_0 et a'_0 tandis qu'A_1 diffuse a_1 et a'_1. Alors, p_0 peut recevoir dans l'ordre a_0, a'_0, a_1 et a'_1, tandis que p_1 peut recevoir dans l'ordre a_0, a_1, a'_1 et a'_0.

Algorithme 18 Traitement d'un message automate en provenance directe de l'automate A ou bien trouvé dans une uot-diffusion faite par la réplique relais de A (utilisation de L'_A)

```
1: switch (Origine de msgReçu)
2: case provenance directe de l'automate A:
3:    if Processus courant n'est pas relais pour A then
4:       if L'_A != [ ] then
5:          removeFirst(L'_A)
6:       else
7:          extend(L_A,msgReçu)
8:       end if
9:    else
10:      utoDiffusion(msgRecu)
11:   end if
12: case uot-livraison de l'uot-diffusion faite par le processus relais de l'automate A:
13:   if Processus courant n'est pas relais pour A then
14:      if L_A != [ ] then
15:         removeFirst(L_A)
16:      else
17:         extend(L'_A,msgReçu)
18:      end if
19:   end if
20:   FR ← true
21:   Messagerie(msgRecu)
22:   FR ← false
23: end switch
```

Algorithme 19 Traitement d'un message automate en provenance directe de l'automate A ou bien trouvé dans une uot-diffusion faite par la réplique relais de A (sans utilisation de L'_A)

```
1: switch (Origine de msgReçu)
2: case provenance directe de l'automate A:
3:    if Processus courant n'est pas relais pour A then
4:       if nbUOTLivraisonsAttendues_A ⩾ 0 then
5:          extend(L_A,msgReçu)
6:       end if
7:       nbUOTLivraisonsAttendues_A ← nbUOTLivraisonsAttendues_A + 1
8:    else
9:       utoDiffusion(msgRecu)
10:   end if
11: case uot-livraison de l'uot-diffusion faite par le processus relais de l'automate A:
12:   if Processus courant n'est pas relais pour A then
13:      if nbUOTLivraisonsAttendues_A > 0 then
14:         removeFirst(L_A)
15:      end if
16:      nbUOTLivraisonsAttendues_A ← nbUOTLivraisonsAttendues_A - 1
17:   end if
18:   FR ← true
19:   Messagerie(msgRecu)
20:   FR ← false
21: end switch
```

Grâce à la dynamicité du langage Objective-C, notre mémoire répartie partagée à base d'uot-diffusion n'entraîne que des répercussions réduites sur un code existant : au niveau de la classe mère de toutes les classes, ajout d'un bit (objet répliqué ou non) ; au niveau de chaque classe, mise à jour du constructeur pour que l'objet s'inscrive à la table des objets répliqués `objectTable`, détermination des méthodes modifiantes pour chaque classe, instructions de sérialisation/désérialisation des paramètres pour chacune de ces méthodes, codage des méthodes d'envoi (respectivement réception) d'état. Ces répercussions ont un effet limité sur les performances. Pour la plupart des invocations de méthodes, l'intercepteur requiert juste un test du bit « objet répliqué ? ».

La programmation orientée aspects nous permettrait d'obtenir le même résultat. Il faudrait alors appliquer un aspect gérant l'uot-diffusion d'invocation de méthode au début de chaque méthode modifiant l'état d'un objet.

En ce qui concerne la méthode d'intégration des diffusions fiables et ordonnées provenant d'entités ne participant pas à l'uot-diffusion, notre solution revient à attribuer un rôle de séquenceur (cf. section 3.2.1) à la réplique relais. Donc, au final, notre méthode mixe le protocole des trains avec un protocole à séquenceur fixe. Nous perdons probablement en efficacité en n'utilisant pas le même protocole.

6.3.5 Conclusion

Dans cette section, nous avons détaillé une mémoire répartie partagée implantée sur un protocole d'uot-diffusion. Cette mémoire [13] a été mise en œuvre dans des installations de *P3200* tolérantes aux fautes sur des sites industriels [Baradel *et al.*, 1995]. Ainsi, au métro du Caire, 2 répliques de P3200 sont chargées du contrôle-commande et de la supervision de la ventilation des tunnels. De plus, 5 répliques supervisent le trafic des rames de métro.

6.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'utilisation du protocole des trains pour une application industrielle. Cela nous a conduit à étudier la mise en œuvre du protocole des trains et ses conséquences sur les algorithmes proposés dans le chapitre 4. Entre autres, nous avons proposé une méthode pour gérer le dépassement de capacité de l'horloge logique du train, une autre pour coder le circuit des trains dans l'entête de chaque train, et une dernière pour gérer la régulation du débit d'émission.

Puis, nous avons défini les services additionnels nécessaires pour que l'intergiciel à base de trains puisse être exploité par des applications : gestion de groupes et prise en compte de contraintes liées à la cohérence.

Enfin, nous avons présenté la mise en œuvre d'une mémoire répartie partagée sur un intergiciel d'uot-diffusion. Dans la terminologie de la programmation orientée aspects, notre solution consiste à appliquer un aspect au début de chaque méthode modifiante du code du *P3200*. Ainsi, les mises à jour sont uot-diffusées à toutes les répliques de *P3200*. Ce mécanisme d'aspect doit être complété par un autre mécanisme pour tenir compte de

13. Le protocole d'uot-diffusion utilisé était la version monotrain du protocole des trains.

l'architecture système dans laquelle s'inscrit un *P3200*. En effet, les répliques de *P3200* reçoivent des diffusions fiables et ordonnées provenant d'automates programmables et ce flux de messages doit être intégré au sein du flux d'uot-diffusions.

Ce chapitre clôt la présentation détaillée de nos travaux autour de la mémoire répartie partagée à base d'uot-diffusion. Le prochain chapitre fait la synthèse de ces travaux et les met en perspective.

Chapitre 7

Conclusions et perspectives

Ce chapitre conclut cette première partie portant sur la communication de données dans un système réparti. Il synthétise nos contributions qui ont permis d'aboutir à une mémoire répartie partagée à base d'uot-diffusion. Il propose ensuite des perspectives de nos travaux.

Dans un premier temps, nous nous sommes focalisés sur l'uot-diffusion et avons présenté un protocole original, le « Protocole des trains ». Nous avons démontré qu'il vérifiait toutes les propriétés d'un protocole d'uot-diffusion. Par ailleurs, nous avons montré que, théoriquement, ce protocole surclasse en termes de débit LCR, le plus performant des protocoles existants. De plus, nous avons proposé des solutions originales aux problèmes de mise en œuvre (notamment la gestion des dépassements de capacité des compteurs). Nous avons également présenté les services à ajouter à ce protocole des trains pour qu'il devienne un intergiciel aisément utilisable par une application. Une implantation a été réalisée. Les premiers résultats des tests confirment la théorie par l'expérience.

Dans un second temps, nous avons présenté comment implanter une mémoire répartie partagée à l'aide d'un protocole d'uot-diffusion. Nous avons décrit le cœur de la solution : un intercepteur intégré à la messagerie du langage objet utilisé. Puis, nous avons traité les contraintes induites par une application à base d'objets Objective-C. Notamment, nous avons proposé une solution pour que les participants au protocole d'uot-diffusion intègrent, sans perte d'informations, même en cas de défaillance, un flux de diffusions fiables et ordonnées envoyés par des entités ne participant pas à l'uot-diffusion.

Une version monotrain du protocole des trains et la mémoire répartie partagée basée sur l'uot-diffusion ont été implantées dans le système de contrôle-commande et de supervision *P3200* pour en faire une version tolérante aux fautes par réplication. Cette version a été déployée sur plusieurs dizaines de sites industriels, le plus gros site comptant 5 répliques de *P3200*. Par ailleurs, nous avons codé la version multitrain.

Les perspectives de ces travaux sont multiples.

En ce qui concerne la facette « uot-diffusion » de notre contribution, il nous semble tout d'abord intéressant de travailler à l'amélioration des performances du protocole des trains. Trois pistes valent la peine d'être explorées : 1) faire évoluer les algorithmes pour qu'ils soient adaptés à un système où les ordinateurs sont multi-cœurs, en vue de gagner

encore en performances dans le cas où ce sont les machines qui sont les éléments limitant le débit du protocole ; 2) intégrer la gestion dynamique du nombre de trains de sorte que ce protocole utilise moins de ressources, ce qui permettrait d'envisager son utilisation dans un environnement contraint en énergie (par exemple, les centres de calculs correspondant au *cloud*) ; 3) réfléchir aux conséquences (sur les algorithmes, sur les propriétés et leur démonstration, sur les performances) du remplacement de TCP par des canaux offrant des performances meilleures au détriment d'une moindre qualité de service. Pour cette dernière piste, la librairie *JGroups* [Ban02] est une candidate intéressante puisqu'elle permet de spécifier précisément les propriétés des canaux qu'on utilise. La librairie *eNet* [ENe03] pourrait être également une alternative intéressante. En effet, dans le domaine du jeu vidéo multijoueur, elle est souvent préférée à TCP du fait de ses performances.

Vu les débits élevés qu'offre le protocole des trains, il serait aussi intéressant d'expérimenter son intégration à un intergiciel pour la tolérance aux fautes comme *JGroups* [Ban02] ou *Spread* [Spr98]. Cela permettrait la diffusion des idées présentées dans cette thèse. De plus, en ce qui concerne *JGroups* qui ne garantit qu'une livraison avec ordre total *non-uniforme* [Guerraoui *et al.*, 2010], cela lui apporterait l'uniformité.

L'intégration de l'UTDN à des intergiciels de communication classiques mérite également d'être investiguée, sur la base de l'expérience de Ken Birman avec *Isis*, *Live Objects* et *Quicksilver* [Birman, 2010] : expérimenter l'intégration de l'uot-diffusion à des intergiciels comme *ZeroMQ* [Hc11] ou *RabbitMQ* [Videla et Williams, 2011] nous fournirait des éléments pour contribuer pertinemment aux spécifications d'*AMQP* [AMQ12].

Par rapport à la facette « Mémoire répartie partagée à base d'uot-diffusion » de notre contribution, nous souhaitons l'approfondir en expérimentant la mise en place d'un intercepteur au sein du langage GNU-Objective-C. Cela pourrait déboucher sur des applications industrielles actuelles intéressantes, vu que GNU-Objective-C est le compilateur utilisé par Apple pour la plupart des applications Mac, iPod, iPhone et iPad. Nous souhaitons également expérimenter l'intégration de l'uot-diffusion à l'application à l'aide d'aspects de la programmation orientée aspects. Nous serons ainsi en mesure de comparer concrètement les deux approches.

Un domaine d'expérimentation de tous ces travaux est l'industrie du jeu vidéo sur mobile. Nous contribuons à des intergiciels de communication pour les jeux sur mobile avec *uGASP* [PAD$^+$07, Pellerin *et al.*, 2005a] et le *Communication Middleware TOTEM* (qui s'appuie sur l'intergiciel *RabbitMQ*) [TOT09, Conan *et al.*, 2011]. Nous envisageons d'y intégrer l'uot-diffusion (éventuellement enfouie au sein d'une mémoire répartie partagée) pour faciliter la programmation de jeux multijoueurs grâce à des objets répliqués. Cela nécessitera des APIs dédiées, comme nous avons commencé à l'expérimenter avec un mini-« jeu » basé sur *JGroups* et *Android* [Cheynet et Sionneau, 2011]. Certes, des intergiciels de communication pour le jeu comme *Marauroa* [Mar99], *Raknet* [Jen05] ou encore *ReplicaNet* [Rep01] proposent déjà des APIs pour gérer des objets répliqués. Mais, aucun ne garantit toutes les propriétés des protocoles d'uot-diffusion. Or, certains jeux en ont besoin (cf. *Age of Empire* [Bettner et Terrano, 2001] évoqué en section 3.2.4, et *Darwinia* [Kno09] évoqué en section 3.2.1) et développent donc des solutions *ad hoc*.

Cette perspective conclut ce chapitre et cette première partie. La seconde partie concerne un système réparti caractérisé par l'absence d'un réseau informatique global.

Ce que nous avons étudié jusqu'à présent n'est plus adapté : nous devons nous concentrer sur le partage de données.

Deuxième partie

Partage de données dans un système réparti

Chapitre 8

Introduction

Cette seconde partie présente nos travaux liés au partage de données dans un système réparti. Effectués dans le cadre du projet ANR PLUG (*PLay Ubiquitous Games and play more*) de janvier 2008 à décembre 2009, ces travaux concernaient la *mémoire répartie partagée à base de* RFID (abrégée en RDSM pour RFID-*based Distributed Shared Memory*) qui a été mise en œuvre au sein du jeu ubiquitaire (/démonstrateur) PLUG *: les Secrets du Musée* (*PSM*) [daem09b, Sim09]. Ce jeu a été expérimenté dans le *Musée des Arts et Métiers* (Paris).

Ce jeu avait les contraintes suivantes :

1. les utilisateurs peuvent lire et modifier (à l'aide de téléphones mobiles NFC) des valeurs stockées dans des étiquettes NFC ;
2. à l'aide de ces mêmes téléphones mobiles, les utilisateurs peuvent avoir une estimation des valeurs stockées dans des étiquettes NFC distantes. Ils peuvent ainsi décider s'il est souhaitable qu'ils aillent jusqu'à une certaine étiquette ou non ;
3. du fait de contraintes projet (impossibilité d'acheter des forfaits 3G pour tous les téléphones utilisés) et techniques (les téléphones utilisés, des Nokia 6131 NFC, ne peuvent pas se connecter à un réseau Wi-Fi), les téléphones ne doivent jamais avoir besoin d'accéder à un réseau informatique global de type 3G ou Wi-Fi.

Le chapitre 9 introduit la technologie RFID (dont fait partie la technologie NFC) et présente les différentes architectures RFID de la littérature. Il montre qu'aucune ne permet de répondre simultanément aux contraintes 1, 2 et 3. Puis, dans le chapitre 10, nous introduisons des vecteurs d'horloges dans des étiquettes RFID de manière à obtenir la RDSM. L'architecture originale qui en résulte répond à l'ensemble des contraintes 1, 2 et 3. Nous présentons les données expérimentales issues de l'exécution du jeu qui s'appuie sur cette RDSM. Ensuite, le chapitre 11 positionne la RDSM par rapport aux autres architectures RFID. Nous présentons des critères de comparaison des différentes architectures RFID, évaluons chacune d'entre elles, puis proposons un guide pour choisir l'architecture la plus adaptée à une application donnée. Enfin, le chapitre 12 conclut cette partie en présentant les perspectives de nos travaux.

Chapitre 9

État de l'art

Ce chapitre présente l'état de l'art du domaine RFID. Après une brève introduction à la technologie RFID (cf. section 9.1), nous synthétisons les trois architectures présentes dans la littérature académique ou dans l'industrie pour retrouver les données informatiques associées à une entité physique munie d'une étiquette RFID. Nous les avons baptisées : architecture centralisée (les données sont stockées dans une base de données centralisées, cf. section 9.2), architecture semi-répartie (elles sont stockées dans une base de données locale au terminal utilisé par la personne qui lit l'étiquette, cf. section 9.3), et architecture répartie (elles sont stockées dans l'étiquette, cf. section 9.4). Nous détaillons chacune de ces architectures en présentant des exemples d'application et en expliquant leur inadéquation aux contraintes du projet PLUG énoncées au chapitre 8.

9.1 Introduction à la technologie RFID

Cette section introduit brièvement la technologie RFID (pour une présentation plus approfondie, se reporter à [Roussos et Kostakos, 2009]). Cette technologie met en jeu deux types d'entités [Roussos et Kostakos, 2009]. D'une part, les *étiquettes* (ou *transpondeurs*) RFID contiennent au minimum un numéro de série et éventuellement de la mémoire. Cette mémoire peut être de type *Write Once Read Many* (WORM). Elle a alors une capacité de quelques octets. Cette mémoire peut être aussi de type *Non-Volatile RAM* (NVRAM). Dans ce dernier cas, sa capacité peut aller de quelques octets à 4, voire 8 Kio. Pour des raisons de sécurité, l'accès à cette mémoire (en lecture et/ou en écriture) peut nécessiter une clé d'accès. D'autre part, les *lecteurs* RFID sont des terminaux capables de lire des informations, via un protocole radio dédié, avec les étiquettes RFID. Selon le type d'étiquette et de lecteur, ce dernier peut également modifier le contenu de l'étiquette. Notez que : 1) certaines étiquettes sont capables de chiffrer les communications entre elles et le lecteur RFID ; 2) les lecteurs RFID peuvent aussi interagir entre eux via le protocole utilisé avec les étiquettes. *Pair-à-pair* est le nom donné à cette interaction.

Les étiquettes RFID peuvent être *actives* ou *passives*. Les étiquettes *actives* disposent de leur propre source d'énergie. Cela permet une meilleure portée et une meilleure fiabilité pour les communications avec les lecteurs. En revanche, ces étiquettes cessent de fonctionner quand leur source d'énergie est tarie. Cela explique le grand intérêt pour les étiquettes

passives (qui, selon [Roussos et Kostakos, 2009], explique en grande partie l'engouement pour la technologie RFID) : elles convertissent le champ magnétique généré par le lecteur en un courant électrique qui leur permet de s'activer et de communiquer avec le lecteur. Elles n'ont donc pas besoin d'avoir leurs batteries rechargées ou remplacées pour pouvoir être toujours opérationnelles face à un lecteur.

Selon les étiquettes utilisées (et donc les fréquences de travail qu'elles acceptent), un lecteur peut être à plusieurs mètres de distance pour lire une étiquette ou bien doit être à quelques centimètres. Par ailleurs, la lecture peut se faire ou non à travers un liquide (ce qui fait que l'étiquette peut être injectée ou non dans le corps d'un être vivant).

Avec le prix des étiquettes qui ne cesse de diminuer, la technologie RFID est de plus en plus utilisée dans le milieu industriel, notamment pour la gestion de la chaîne globale d'approvisionnement des entreprises [Gonzalez, 2008]. Des standards comme l'EPC [Armenio *et al.*, 2007] ont été mis en place.

En parallèle, un second standard a émergé de l'industrie des télécommunications : le *Near Field Communication* (NFC) désigne une technologie qui vise l'intégration de fonctionnalités RFID dans les appareils personnels (type téléphone mobile, smartphone...) pour qu'ils se comportent comme des étiquettes RFID ou des lecteurs RFID [Wiechert *et al.*, 2008]. Travaillant en fréquence HF, les étiquettes NFC ne peuvent être lues qu'à quelques centimètres (dizaines de centimètres pour l'ISO 15693) de distance. De ce fait, le standard NFC est (ou plutôt sera, vu que beaucoup de pays, dont la France, en sont encore au stade des expérimentations) majoritairement utilisé pour du paiement avec mobile, de la gestion de cartes de fidélité... tandis que la RFID/EPC est dédié au traçage d'objets, à l'amélioration de la gestion des stocks... [Wiechert *et al.*, 2008].

Différentes études prévoient à plus ou moins court-terme beaucoup plus de lecteurs NFC que de lecteurs RFID/EPC. Certes, le marché est loin des 700 millions de terminaux NFC en 2013 prédits en 2008 par certaines études [Chr08]. Mais 10% des 445 millions de téléphones vendus en 2011 incluaient la fonctionnalité NFC ; et les vendeurs de circuits NFC prévoient des chiffres encore meilleurs en 2012 [Dav12]. Par ailleurs, dans le chapitre 10, nous verrons que nos expérimentations se sont appuyées sur des téléphones mobiles NFC. Toutefois, l'ensemble de notre recherche est applicable aussi bien au domaine RFID/EPC que NFC. C'est pourquoi, dans la suite de cette thèse, nous utiliserons systématiquement le terme générique RFID.

Beaucoup d'auteurs perçoivent la RFID comme un moyen de communiquer avec les objets, voire d'avoir des objets communicants [Ranque *et al.*, 2011, Gonzalez, 2008]. En fait, c'est seulement un moyen d'accéder aisément à des données informatiques associées à un objet [Roussos et Kostakos, 2009]. En effet, en munissant une entité physique (qui peut être un endroit, un objet, une plante, un animal...) d'une étiquette RFID, il est possible d'accéder aisément [1] à ses données informatiques : soit elles sont stockées physiquement sur l'étiquette, soit l'étiquette permet de retrouver la base de données où sont rangées ces données.

Les sections suivantes détaillent les trois architectures utilisées dans la littérature ou dans l'industrie pour stocker ces données.

1. Un code-barre ou un QR-Code permettent également d'associer des données informatiques à une entité physique, mais ils doivent être visibles du lecteur (ce qui n'est pas nécessaire avec la RFID).

9.2 Architecture centralisée

Cette architecture, très souvent utilisée dans la gestion de la chaîne globale d'approvisionnement des entreprises, a été standardisée par EPCglobal [Armenio *et al.*, 2007]. Il en existe des versions simplifiées par rapport à ce standard.

Quand un lecteur RFID est proche d'une étiquette, il lit l'identifiant de l'étiquette ou bien l'identifiant stocké dans la zone de données de l'étiquette (son *Electronic Product Code* dans le cas de l'EPCglobal). Le lecteur RFID demande alors à un serveur informatique (*ONS lookup service* dans le cas de l'EPCglobal) quelle est la machine (*EPC Manager* dans le cas de l'EPCglobal) qui gère l'identifiant lu. Fort de la réponse de ce serveur, le lecteur contacte cette machine avec l'identifiant lu, afin que cette machine consulte sa base de données et renvoie les données associées à cet identifiant.

9.2.1 Exemples d'applications

Aspire RFID est un intergiciel *Open Source* conforme aux spécifications de l'EPCglobal [Asp11]. Il propose plusieurs exemples d'applications industrielles de suivi de produits.

Dans la suite de cette section, nous présentons des prototypes ou des produits développés selon cette architecture centralisée, mais hors EPCglobal.

PAC-LAN est un prototype de jeu dans lequel les joueurs sont équipés de téléphones mobiles NFC (sans fonctionnalité GPS) [Rashid *et al.*, 2006]. Les joueurs doivent notamment interagir avec des étiquettes NFC qui ont été disséminées dans un quartier. Dans une base de données centrale, l'identifiant de chaque étiquette est associé à ses coordonnées géographiques dans le quartier. Quand un joueur lit une étiquette, son téléphone mobile contacte le serveur avec cet identifiant, via le réseau 3G. Le serveur retrouve ainsi les coordonnées géographiques du joueur, coordonnées qu'il retransmet au téléphone mobile de chaque autre joueur.

[Haberman *et al.*, 2009] propose une application pour que les visiteurs d'une exposition d'art puissent découvrir des peintures autrement. Des étiquettes NFC sont disposées au dos de peintures exposées. Muni d'un téléphone NFC, le visiteur pose son téléphone à des endroits de la peinture qui l'intriguent. Le téléphone lit le numéro codé dans l'étiquette, contacte le serveur uGASP [PAD+07, Pellerin *et al.*, 2005a, Pellerin *et al.*, 2005b]. Ce dernier indique au téléphone ce qu'il doit faire : afficher un texte, une image, ou bien faire écouter un commentaire audio. L'auteur de l'œuvre peut ainsi communiquer avec le visiteur.

Via Mineralia est un jeu pervasif sérieux (*pervasive serious game* en anglais) qui permet d'enrichir la visite d'un musée de Freiberg (Allemagne) [Heumer *et al.*, 2007]. Dans ce jeu, le visiteur utilise un PDA équipé d'un lecteur RFID. Des étiquettes RFID (avec un identifiant unique) sont disposées au niveau des vitrines que le musée souhaite mettre en valeur. Quand le PDA scanne une étiquette, il envoie une requête HTTP (avec cet identifiant en paramètre) à un serveur Web, via un réseau Wi-Fi qui couvre l'ensemble du musée. Ce serveur renvoie toutes les informations multimédia au PDA qui les affiche dans un navigateur.

La société *Touchatag* (ex-*Tikitag*) vend des lecteurs NFC connectables à des ordina-

teurs personnels (sous Windows ou Mac-OS) et des étiquettes NFC dédiées à *Touchatag* [Tou10]. L'acheteur peut ensuite se connecter au site Web `http://www.touchatag.com` et définir la réaction associée à la lecture d'une (ou plusieurs) de ces étiquettes. Quand le lecteur NFC lit une étiquette NFC, il en informe l'*application Touchatag* qui tourne en permanence sur l'ordinateur personnel. Cette application contacte alors, via le réseau Internet auquel est connecté cet ordinateur, un service *Touchatag* appelé l'*Application Correlation Service* (ACS). L'*application Touchatag* donne l'identifiant de l'étiquette à l'ACS. Ce dernier retrouve alors la réaction associée à cette étiquette et en informe l'*application Touchatag*. Cette dernière réagit de manière appropriée. Par exemple, supposons que l'utilisateur ait défini sur le site Web de *Touchatag* que, quand l'étiquette r d'identifiant i est posée sur le lecteur, il souhaite accéder, avec un navigateur, à l'*Uniform Resource Locator* (URL) d'un site Web W. Quand l'utilisateur pose son étiquette r sur le lecteur, l'*application Touchatag* contacte l'ACS avec l'identifiant i. L'ACS renvoie l'URL de W. L'*application Touchatag* ouvre alors un navigateur avec cette URL.

Skylanders est un jeu vidéo développé par Activision [Act11]. Jeu pour enfant le plus vendu en 2011 selon NPD, GfK et Charttrack [Age12], il suppose l'utilisation de figurines en plastique. Ces figurines contiennent une étiquette NFC. Quand le joueur pose une figurine sur le *portail du jeu* (en fait, un lecteur d'étiquettes NFC), le jeu vidéo récupère l'identifiant NFC stocké dans l'étiquette, contacte un serveur et récupère ainsi les informations concernant la figurine virtuelle et animée qui doit apparaître dans le jeu vidéo : la figurine en plastique devient vivante à l'écran. Notez que, selon [Pla12], des informations sont également stockées au niveau de l'étiquette : le jeu peut donc s'affranchir de l'utilisation d'un réseau global pour contacter un serveur. *Skylanders* utilise donc non seulement une architecture centralisée, mais également répartie.

9.2.2 Inadéquation aux contraintes du projet PLUG

L'architecture centralisée permet à des utilisateurs de lire et de modifier (à l'aide de téléphones mobiles NFC) des valeurs stockées dans des étiquettes NFC (contrainte 1). En effet, avec l'identifiant récupéré de l'étiquette, le téléphone peut envoyer des requêtes (de lecture ou de mise à jour) vers le serveur qui gère la base de données contenant les données associées à l'identifiant.

De plus, cette architecture permet aux utilisateurs de consulter les valeurs stockées dans des étiquettes NFC distantes (contrainte 2). En effet, si le téléphone connaît l'identifiant de l'étiquette distante, il peut envoyer des requêtes vers le serveur.

Toutefois, l'architecture centralisée ne convient pas. En effet, elle requiert un réseau informatique global, ce qui est contraire à la contrainte 3.

L'architecture semi-répartie cherche à s'affranchir de ce besoin de réseau global.

9.3 Architecture semi-répartie

Cette architecture n'est évoquée que via l'application présentée dans [ITR06]. Chaque terminal mobile contient une base de données contenant toutes les données associées aux identifiants des étiquettes RFID utilisées. Chaque base de données est synchronisée pério-

diquement avec une base de données centrale. Lorsqu'un utilisateur lit l'identifiant d'une étiquette RFID avec son terminal mobile, ce dernier lit et/ou met à jour les données associées qu'il trouve dans sa base de données locale. Les éventuelles mises à jour sont remontées à la base de données centrale lors de la prochaine synchonisation de ce terminal. De plus, elles sont répercutées sur les autres terminaux mobiles lors de la prochaine synchronisation de ces derniers avec la base de données centrale.

9.3.1 Exemple d'application

L'unique exemple de mise en œuvre de cette architecture est une application de gestion des arbres de la ville de Paris [ITR06]. Chacun des quatre-vingt-quinze mille arbres des avenues de Paris est équipé avec une étiquette RFID. Chaque jardinier synchronise sa tablette PC avec la base de données centrale avant une nouvelle journée de travail. Ensuite, pendant sa journée de travail, à chaque fois qu'il effectue une opération sur un arbre, le jardinier lit l'étiquette RFID de l'arbre : son terminal met à jour les données associées qu'il trouve dans sa base de données locale. Le soir, le jardinier synchronise sa tablette PC avec la base de données centrale. Il remonte ainsi ses propres mises à jour et récupère celles faites par les autres jardiniers.

9.3.2 Inadéquation aux contraintes du projet PLUG

Cette architecture semble répondre pleinement à l'ensemble des contraintes du projet PLUG. Notamment, cette architecture s'affranchit du réseau global dont a besoin l'architecture centralisée (cf. contrainte 3).

Toutefois, cette architecture est confrontée à un problème concernant les mises à jour des données associées à une étiquette. En effet, si un utilisateur u_1 modifie les données associées à l'étiquette r, u_1 modifie le contenu de la base de données locale à son terminal. Si, quelques instants plus tard, un utilisateur u_2 vient consulter les données associées à r, u_2 consulte la base de données locale à son propre terminal. Les données associées ne reflètent donc pas les mises à jour faites par u_1, à moins que u_1, puis u_2 aient synchronisé leur terminal avec la base de données centrale, après la mise à jour faite par u_1.

Dans le cas de la gestion des arbres de la ville de Paris, chaque jardinier travaille sur une zone géographique différente. Donc, la probabilité que deux jardiniers consultent dans la même journée l'état d'une même étiquette r (donc entre deux synchronisations avec la base de données centrale) est nulle (sauf si les jardiniers travaillent en équipe et que chacun dispose de sa propre tablette). En revanche, dans le cas du jeu envisagé pour le projet PLUG, il est irréaliste de demander aux joueurs qu'à chaque fois qu'un joueur modifie les données associées à une étiquette, tous les joueurs reviennent à un point spécifique du musée pour synchroniser, via Bluetooth, leur mobile à une base de données centrale.

L'architecture répartie a pour objectif de s'affranchir de ce besoin de synchronisation à une base de données centrale.

9.4 Architecture répartie

Le principe de cette architecture est de stocker toutes les données associées à une étiquette RFID au sein même de cette étiquette.

9.4.1 Exemples d'application

Les téléphones *Nokia 6131* NFC sont vendus avec trois étiquettes NFC. Chacune permet de déclencher une fonction différente au niveau du téléphone : l'une d'elles permet d'activer la fonction réveil du téléphone ; une autre permet de faire jouer une certaine musique au téléphone ; la dernière permet d'afficher une présentation sur le NFC. Pour accomplir cela, le téléphone lit le contenu de ces étiquettes qui est codé sous la forme d'un *Uniform Resource Identifier* (URI) selon les spécifications des *Smarts Posters* du NFC *Forum* [NFC Forum, 2006a, NFC Forum, 2006b, NFC Forum, 2006c]. Ces URI permettent à l'étiquette de « demander » au téléphone (si ce dernier est programmé pour comprendre le contenu de l'étiquette selon ces spécifications) d'accomplir une certaine fonction comme envoyer un SMS, appeler une certaine personne, ouvrir une page à une adresse Web donnée, etc.

C'est d'ailleurs cette spécification *Smart Posters* qui permet à n'importe quel téléphone NFC d'exploiter les étiquettes *Touchatag* évoquées en section 9.2.1. En effet, les étiquettes *Touchatag* contiennent non seulement un identifiant i exploité par l'*application Touchatag* évoquée en section 9.2.1, mais aussi un URI : cet URI correspond à l'URL d'un serveur Web *Touchatag*, avec un paramètre correspondant à l'identifiant i. Ainsi, quand un utilisateur lit une étiquette *Touchatag* avec un téléphone NFC, le mobile lit l'URL et ouvre donc un navigateur avec cette URL. Le serveur Web *Touchatag* est donc contacté, via 3G ou Wi-Fi, avec l'identifiant i en paramètre. Ce serveur Web adresse alors une requête à l'ACS (cf. section 9.2.1) avec l'identifiant i. Dans le cas où i est associé à un site Web W, l'URL de W est renvoyée au serveur Web *Touchatag*. Ce dernier retourne au navigateur du téléphone mobile une page HTML contenant une redirection vers W. Le navigateur affiche donc ensuite W. Ainsi, dans le cas d'une étiquette *Touchatag* lue par un téléphone mobile, le téléphone s'appuie sur une architecture répartie pour retrouver le serveur Web *Touchatag* à contacter ; le serveur Web *Touchatag* s'appuie sur une architecture centralisée pour interpréter l'identifiant.

C'est encore cette spécification *Smart Posters* qui est utilisée par la société *Connecthings* pour rendre « intelligentes » des boîtes aux lettres de *La Poste* [Con11]. Quand l'utilisateur scanne une boîte aux lettres équipée d'une étiquette NFC, son téléphone lit l'URL stockée dans cette étiquette (elle contient en paramètre un identifiant qui permet notamment de retrouver l'emplacement géographique de cette boîte aux lettres) et ouvre un navigateur vers cette URL. Cette page Web permet à l'utilisateur de connaître la localisation de boîtes aux lettres environnantes et leur heure de levée, le code postal d'une ville donnée, etc.

Le passe *Navigo* constitue un mode de validation du billet pour les Franciliens empruntant les transports en commun (métro, bus, train transilien, RER et tramway) [Levallois-Barth, 2009]. Les 4,5 millions d'utilisateurs de ce passe n'ont pas de lecteurs RFID, mais ce passe qui contient une étiquette NFC. Au niveau d'une borne d'achat, chaque usager initia-

lise, moyennant paiement, son tag avec les droits d'utilisation des transports en commun [2]. Ensuite, à chaque fois qu'il veut emprunter un transport en commun, l'usager présente son passe devant le lecteur NFC qui est installé dans chaque portillon. L'ordinateur connecté au lecteur vérifie les droits stockés sur la carte et ouvre le portillon si jamais l'usager a effectivement le droit d'emprunter ce transport en commun.

Ubi-Check est une application (académique) permettant aux voyageurs qui vont prendre l'avion de ne pas perdre de bagages à main lors des contrôles de sécurité [Couderc et Banâtre, 2009]. Une étiquette RFID est attachée à chaque bagage à main du voyageur. Au début du voyage, chaque étiquette est initialisée avec une valeur spécifique au voyageur. Ensuite, après le contrôle de sécurité où le voyageur a dû quitter ses bagages avant de les reprendre, le voyageur passe sous un portillon qui lit toutes les étiquettes RFID. Si une incohérence est découverte parmi les différentes valeurs lues, cela signifie que le voyageur a échangé son bagage à main avec le bagage d'un autre voyageur. Une alarme est donc déclenchée pour prévenir le voyageur qu'il détient un bagage qui ne lui appartient pas (et que donc il lui manque un bagage).

[Mamei et Zambonelli, 2007] propose un système (académique) basé sur des phéromones digitales pour retrouver des objets égarés dans une maison. Pour ce faire, tous les sols de la maison sont couverts d'étiquettes RFID. Un lecteur RFID est associé à chaque objet de la maison. Quand l'utilisateur déplace un objet d'un point A à un point B, le lecteur RFID associé à cet objet se comporte comme une fourmi qui dépose des phéromones sur le chemin qu'elle emprunte : ce lecteur dépose une phéromone digitale (constitué d'un identifiant de l'objet et une date/heure de passage) dans la NVRAM de chaque étiquette RFID au dessus de laquelle il passe. Notez que, de même qu'une phéromone s'évapore avec le temps, si jamais le lecteur RFID ne trouve pas de place dans la NVRAM d'une étiquette (il y a trop de phéromones stockées dessus), le lecteur retire la plus ancienne phéromone présente sur l'étiquette. En cas de perte d'un objet, l'utilisateur prend un lecteur RFID dédié et déambule dans la maison pour trouver la phéromone digitale de l'objet. Une fois qu'il l'a trouvée, il suit sa trace et arrive à l'endroit où il a déposé l'objet.

Roboswarn est une application (académique) qui permet à des robots (équipés de lecteurs NFC) de se positionner dans un espace physique en vue d'accomplir une certaine tâche [Zecca *et al.*, 2009]. Des étiquettes NFC sont disposées à des endroits bien précis d'une pièce (par exemple, au niveau d'un lit d'hôpital que les robots devront pousser afin qu'un robot laveur ne soit pas gêné pendant son ménage). Chaque étiquette est initialisée avec la localisation géographique des autres étiquettes de la pièce et une date/heure de dernier nettoyage. Quand les robots entrent dans la pièce, ils errent à la recherche d'une étiquette NFC. Dès qu'un robot en trouve une, il lit sur l'étiquette NFC les coordonnées géographiques des autres étiquettes et les transmet aux autres robots. Ces derniers rejoignent donc les autres étiquettes. Si la date/heure du dernier nettoyage est trop ancienne, alors les robots déplacent le lit d'hôpital, puis inscrivent la date/heure actuelle comme date/heure de dernier nettoyage. Sinon, les robots ne font rien.

La société *SALTO Systems* vend des serrures de porte électroniques, les clés étant des cartes NFC. Pour faciliter la gestion de l'ensemble des serrures et des cartes, cette société

2. L'usager peut également initialiser son passe chez lui en utilisant un lecteur NFC (dédié) connecté à son ordinateur.

a développé le *SALTO Virtual Network* (SVN [3]) [SAL07]. Ainsi, l'opérateur de l'aéroport de Heathrow peut gérer facilement les 1 000 clés NFC, les 1 000 serrures électroniques normales (des serrures avec un pêne contrôlé par un lecteur NFC) et les 37 *hot spots*, des serrures spéciales chargées de : 1) déverrouiller un accès en entrée sur le site, 2) initialiser la clé NFC avec le fait que cette clé peut ouvrir des serrures banalisées pendant la journée, donner des permissions exceptionnelles (par exemple, pour accéder à une salle de réunion où l'utilisateur de la clé doit se rendre dans la journée) et signaler la mise sur liste noire de certaines clés, 3) déverrouiller un accès en sortie du site, 4) récupérer toutes les données collectées sur la clé pendant la journée de travail de la personne. En effet, à chaque fois qu'une personne cherche à déverrouiller une serrure avec sa clé NFC, la serrure y lit des informations pour vérifier les droits stockés sur la clé et récupérer les clés mises sur liste noire. Mais, elle écrit aussi des informations en stockant notamment sur la clé des informations quant à l'accès de la personne ou des informations concernant l'éventuelle faiblesse de la batterie animant la serrure. Ainsi, grâce à SVN, même si les serrures normales n'ont pas d'accès à un réseau global, elles peuvent quand même recevoir des informations (la liste des clés mises sur liste noire) et peuvent en émettre (éventuelle faiblesse de la batterie) : les serrures normales se servent du réseau constitué par les utilisateurs des clés NFC.

9.4.2 Inadéquation aux contraintes du projet PLUG

L'architecture répartie permet à des utilisateurs de lire et de modifier (à l'aide de téléphones mobiles NFC) des valeurs stockées dans des étiquettes NFC (contrainte 1). De plus, il n'y a pas de possibilité de lire un contenu obsolète comme avec l'architecture semi-répartie.

En outre, cette architecture ne nécessite aucun réseau global (contrainte 3).

En revanche, cette architecture ne permet pas aux utilisateurs de consulter les valeurs stockées dans des étiquettes NFC distantes (contrainte 2).

Elle ne répond donc pas non plus aux contraintes du projet PLUG.

9.5 Conclusion

Ce chapitre a introduit la technologie RFID. Puis, il a présenté les architectures RFID qu'on peut trouver dans la littérature académique et dans l'industrie : architecture centralisée (les données sont stockées dans une base de données centralisées), architecture semi-répartie (elles sont stockées dans une base de données locale au terminal utilisé par la personne qui lit l'étiquette), et architecture répartie (elles sont stockées dans l'étiquette).

Aucune de ces architectures ne satisfait simultanément l'ensemble des contraintes du projet PLUG. C'est pourquoi nous devons envisager une nouvelle architecture qui peut s'affranchir d'un réseau informatique global et mixe les qualités des architectures semi-répartie (possibilité de connaître le contenu d'une étiquette distantes) et répartie (aucun risque de lire un contenu obsolète au niveau d'une étiquette).

3. Ce sigle SVN ne doit pas être confondu avec le sigle du système de contrôle de version de fichiers *Subversion*.

Ce mélange suppose la dissémination d'informations de mises à jour de contenu entre les étiquettes et les mobiles, idée utilisée dans le réseau SVN (cf. section 9.4.1). Dans le chapitre suivant, nous présentons la RDSM que nous avons mise au point pour cette dissémination et l'expérimentation de cette nouvelle architecture.

Chapitre 10

Mémoire répartie partagée à base d'étiquettes RFID

Ce chapitre présente la mémoire répartie partagée à base d'étiquettes RFID (RDSM) que nous avons mise au point dans le cadre du projet PLUG. Cette mémoire s'inscrit dans la lignée des mémoire répartie partagée à cohérence causale [Ahamad *et al.*, 1991]. Toutefois, une première spécificité de notre proposition est que nous n'invalidons pas les données périmées. une autre spécificité est que notre mémoire s'affranchit d'un réseau informatique global, en exploitant le réseau formé par les personnes qui utilisent cette mémoire via l'application stockée dans leur lecteur RFID. En effet, nous stockons des informations de mise à jour de la RDSM dans les terminaux. Ainsi, en se déplaçant, les utilisateurs transportent ces informations d'une étiquette RFID à l'autre. L'utilisation de ce réseau humain introduit une dernière spécificité : pour que notre mémoire soit la plus à jour possible, il est nécessaire de stimuler ce réseau de manière à pousser les utilisateurs à se déplacer rapidement d'une étiquette à l'autre ou bien interagir, via lecteur RFID en pair-à-pair, pour s'échanger des informations de mise-à-jour. De ce fait, certaines fonctionnalités de l'application doivent être pensées en fonction de ce besoin de stimulation. La section 10.1 présente le modèle du système que nous considérons. Puis, la section 10.2 décrit les principes de la RDSM. Ensuite, la section 10.3 présente un exemple d'application utilisant la RDSM. Elle décrit notamment les mécanismes que nous avons mis en œuvre pour stimuler le réseau humain des utilisateurs. Enfin, la section 10.4 présente les résultats expérimentaux sur cette RDSM, issus de l'utilisation de cette application.

10.1 Modèle du système

Le système que nous considérons comprend deux types d'éléments : des étiquettes RFID et des téléphones mobiles capables d'interagir avec ces étiquettes (cf. figure 10.1).

L'ensemble de ces éléments offrent une mémoire répartie (entre les différents éléments) DM (pour *Distributed Memory*). De ce fait, chaque élément e du système détient une partie de DM : quand e est un téléphone mobile (respectivement une étiquette RFID), cette partie n'est modifiable que par le téléphone mobile e (respectivement un téléphone

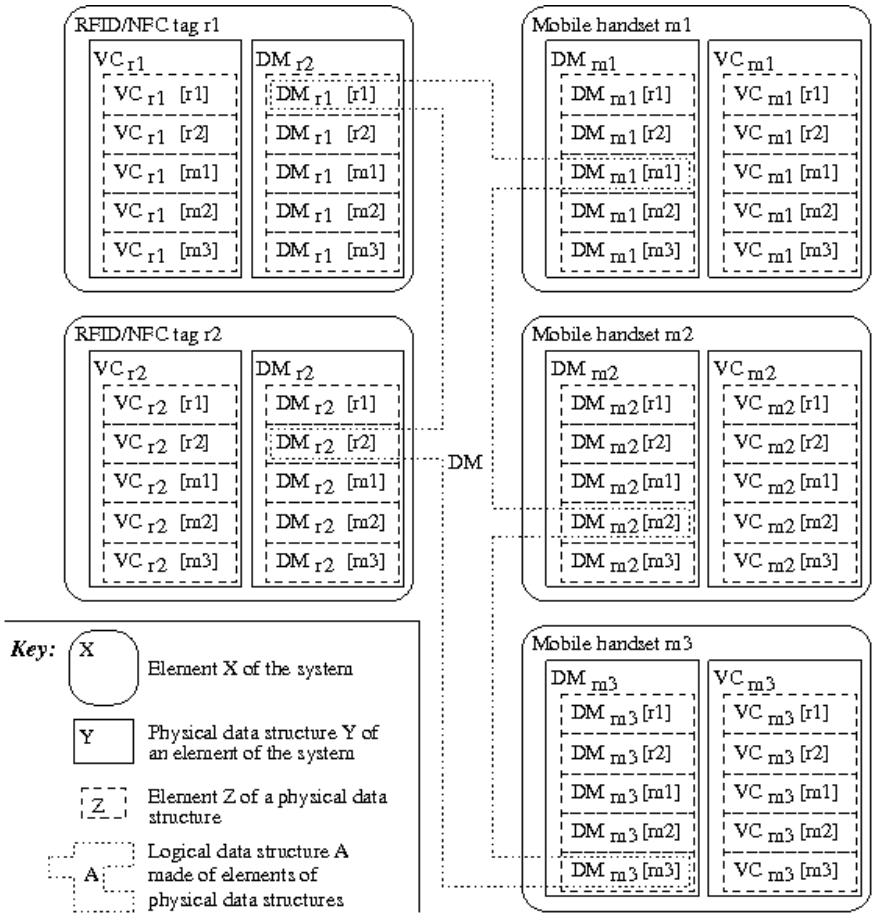


FIGURE 10.1 – Données présentes dans un système constitué de 2 étiquettes RFID et 3 téléphones mobiles

mobile qui est en train d'interagir avec l'étiquette e).

DM est non seulement répartie, mais partagée entre les différents éléments du système (d'où son nom de RDSM) de la manière suivante.

Chaque élément e contient aussi une vue locale DM_e de la RDSM. Nous notons $DM_e[e']$ la vue que l'élement e a du contenu de DM détenu par l'élément e'.

Chaque élément e contient également un vecteur d'horloges VC_e [Mattern, 1988, Fidge, 1988] qui permet de répercuter, sur les différents DM_e, les mises à jour qui ont été faites sur DM. $VC_{e'}[e']$ contient l'estampille de la dernière requête de lecture ou de mise à jour faite sur $DM_{e'}[e']$ (qui est la partie de la mémoire répartie DM détenue par l'élément e'). Par ailleurs, $VC_{e,e\neq e'}[e']$ contient l'estampille de la dernière requête de lecture ou de mise à jour de $DM_{e'}[e']$ dont e a connaissance.

10.2 Principes de cette mémoire

Voici les principales idées de notre solution [Simatic, 2009a]. Chaque étiquette et chaque téléphone mobile détient une vue locale de DM. Nous associons un vecteur d'horloges par vue locale. À chaque fois qu'un mobile entre en contact avec une étiquette ou un autre mobile, ces deux entités échangent leur vue de DM en comparant leurs vecteurs d'horloges. Ainsi, chaque entité utilise la connaissance de l'autre entité pour récupérer des informations plus récentes concernant les évolutions de DM. Elles récupèrent ainsi une vue plus récente de l'état de DM et participent à un protocole de cohérence de données réparties. Notez que, même si un utilisateur pense seulement consulter le contenu d'un étiquette, de manière transparente, son lecteur RFID met éventuellement à jour le contenu de cette étiquette pour y inscrire des informations plus fraiches concernant DM. Les paragraphes suivants détaillent notre solution.

Lors de l'initialisation de l'application, pour chaque élément e du système, DM_e est initialisé avec la valeur initiale de la mémoire répartie. Cette valeur initiale dépend de l'application qui utilisera la mémoire répartie. En revanche, VC_e est initialisé à une valeur indépendante de l'application. Nous supposerons pour l'instant que cette valeur est $(0, \ldots, 0)$ (dans la suite de cette section, nous verrons que cette valeur initiale peut être différente).

Ensuite, pendant le reste de l'exécution de l'application, à chaque fois qu'un téléphone mobile m change la valeur stockée dans $DM_e[e]$ (e étant m ou bien une étiquette r avec laquelle m interagit), m applique l'algorithme 20.

Algorithme 20 Mise à jour de $DM_e[e]$ (e étant un mobile m ou bien une étiquette r avec laquelle m interagit)

1: $DM_e[e] \leftarrow$ mise à jour de $DM_e[e]$
2: $VC_e[e] \leftarrow$ mise à jour de $VC_e[e]$
3: $DM_m[e] \leftarrow DM_e[e]$
4: $VC_m[e] \leftarrow VC_e[e]$

Habituellement $VC_e[e]$ est une horloge logique (cf. , par exemple, [Saito et Shapiro, 2005]) : la ligne 2 de l'algorithme 20 est dans ce cas $VC_e[e] \leftarrow VC_e[e] + 1$. Mais, comme dans [Golding, 1992], nous pouvons tirer parti de deux conditions pour économiser de la mémoire sur chaque étiquette (et ainsi améliorer l'extensibilité). La première condition est que l'estampille temporelle de la dernière mise à jour de l'élément e est une des données de $DM_e[e]$. La seconde condition est que deux mises à jour consécutives sur la même étiquette ont toujours des estampilles temporelles consécutives et différentes. Si ces deux conditions sont satisfaites, alors $VC_e[e]$ peut contenir l'estampille temporelle de la mise à jour en lieu et place de l'horloge logique habituellement utilisée : la ligne 2 de l'algorithme 20 devient : $VC_e[e] \leftarrow$ Estampille temporelle de la mise à jour de $DM_e[e]$. Nous gagnons ainsi la place mémoire d'une horloge logique.

Une autre optimisation peut s'ajouter à la précédente. Elle peut être mise en œuvre si l'estampille temporelle de la dernière requête de lecture sur $DM_e[e]$ est une des données de $DM_e[e]$. Dans le cas où il n'y a pas besoin de distinguer l'estampille temporelle de la dernière mise à jour et l'estampille temporelle de la dernière requête de lecture, $VC_e[e]$

peut contenir l'estampille temporelle de la dernière requête de mise à jour ou de lecture : la ligne 2 de l'algorithme 20 devient : $VC_e[e] \leftarrow$ Estampille temporelle de la dernière requête de lecture ou mise à jour sur $DM_e[e]$. Nous gagnons ainsi l'espace d'une estampille temporelle.

En plus de l'algorithme 20, en appliquant l'algorithme 21, DM_e et VC_e peuvent être mises à jour à chaque fois qu'un élément e est en mesure d'échanger des informations avec une autre élément e'. Dans le cas d'une application basée sur de la RFID, cela peut advenir dans deux cas : un téléphone mobile est proche d'une étiquette RFID (respectivement un autre téléphone mobile) et est en mesure d'interagir via le protocole RFID (respectivement le protocole pair-à-pair NFC).

Algorithme 21 Mise en cohérence de $DM_{e'}$ et $DM_{e''}$

1: **for all** $e \in \{elements\_dans\_le\_systeme\}$ **do**
2: **if** $VC_{e''}[e] < VC_{e'}[e]$ **then** // L'élément e' a une vue plus à jour de cet élément de DM
3: $DM_{e''}[e] \leftarrow DM_{e'}[e]$
4: $VC_{e''}[e] \leftarrow VC_{e'}[e]$
5: **else if** $VC_{e''}[e] > VC_{e'}[e]$ **then** // L'élément e'' a une vue plus à jour de cet élément de DM
6: $DM_{e'}[e] \leftarrow DM_{e''}[e]$
7: $VC_{e'}[e] \leftarrow VC_{e''}[e]$
8: **end if**
9: **end for**

En appliquant les algorithmes 20 et 21, chaque élément e du système a la vue de la mémoire répartie la plus à jour qu'il puisse avoir. Mais, il n'a aucune garantie que c'est la vue effectivement la plus à jour. En effet, DM évolue tout au long de la vie de l'application. De ce fait, il peut exister un élément $e', e' \neq e$ telle que $DM_e[e']$ ne corresponde pas à la valeur $DM_{e'}[e']$ actuellement stockée dans l'élément e'. La seule certitude de e est que l'élément e' a effectivement contenu la valeur $DM_e[e']$ à un moment de son histoire : soit c'est sa valeur initiale, soit c'est une valeur ultérieure qui a été induite par la modification de $VC_{e'}[e']$ (après application de l'algorithme 20) et donc la propagation de cette nouvelle valeur au niveau de $DM_e[e']$ (après application de l'algorithme 21).

La disponibilité de DM_m au niveau de chaque mobile m a un effet de bord intéressant : s'il y a besoin de réinitialiser toutes les valeurs de DM, on peut se contenter de réinitialiser toutes les valeurs de $DM_{m,m\in\{mobiles\}}$ (il est inutile de réinitialiser $DM_{r,r\in\{etiquettes\}}$). Pour ce faire, nous réinitialisons tous les $DM_{m,m\in\{mobiles\}}$ à une valeur initiale DM_{init}. De plus, nous réinitialisons $VC_{m,m\in\{mobiles\}}$ à une valeur initiale VC_{init} plus grande que la plus grande des valeurs actuellement stockée dans $VC_{m,m\in\{mobiles\}}$. Alors $\forall m \in \{mobiles\}, \forall r \in \{etiquettes\}, \forall e \in \{elements\_dans\_le\_systeme\}, VC_m[e] > VC_r[e]$ (puisqu'il y a toujours au moins un mobile qui a connaissance du vecteur d'horloges stocké dans une étiquette) [1]. De ce fait, à chaque fois qu'une étiquette r est en interaction avec un mobile m pour la première fois après la réinitialisation, r (et en particulier $DM_r[r]$) est naturellement

1. Il faut toutefois qu'au tout début de la vie du système, tous les éléments de $VC_{r,r\in\{etiquettes\}}$ soient initialisés à la valeur minimale que peut avoir un élément de vecteur d'horloges.

réinitialisée par m.

Grâce à ces différents principes de la RDSM, une application s'exécutant sur le téléphone mobile m est en mesure de faire des requêtes sur DM_m : elle a une vision de l'ensemble de la mémoire répartie DM sans qu'il y ait besoin que son utilisateur se déplace physiquement au niveau d'une étiquette ou bien communique, via un réseau global informatique, avec un serveur. Cette vision de DM peut contenir des données périmées (cf. section 10.5.2.1). L'application peut contribuer à minimiser ce problème de péremption : comme dans les protocoles de bavardage, si elle stimule l'échange d'informations entre les éléments du système, les mises à jour se propageront plus rapidement dans le système. De ce fait, l'écart entre DM_e et DM sera plus réduit. Nous retrouvons ici la contrainte de bon fonctionnement d'une mémoire répartie partagée à cohérence causale caractérisée par la propriété de vivacité [Ahamad *et al.*, 1991].

Dans cette section, nous avons présenté les principes de la RDSM. La section suivante présente un exemple d'application, puis comment sa conception stimule la dissémination d'informations dans cette mémoire.

10.3 Exemple d'application : un jeu pervasif

La RDSM présentée à la section précédente a été implantée dans le contexte d'un jeu pervasif « PLUG : les Secrets du Musée » (PSM) [daem09b, Sim09]. Cette section présente brièvement PSM, fait le lien entre PSM et les structures de données présentées en section 10.2, et décrit comment le jeu est conçu pour stimuler la dissémination des données dans la RDSM.

PSM a été développé dans le contexte du projet PLUG [CLM+09]. Il est conçu pour permettre aux joueurs de découvrir, d'une manière différente, les objets importants d'un musée [Simatic *et al.*, 2009]. Ce jeu permet notamment de développer les interactions entre les visiteurs/joueurs, alors qu'une visite de musée est en général un loisir individuel [Gentes *et al.*, 2009b, Gentes *et al.*, 2009a]. Dans PSM, 8 équipes de joueurs sont équipées de téléphones NFC (des Nokia 6131 NFC). Leur objectif principal est de collectionner des familles d'objets pendant une session de jeu qui dure au plus 85 minutes. Pour cela, utilisant une *midlet* J2ME que nous avons développée, les joueurs échangent des cartes virtuelles (représentant des objets du musée) soit avec une des 16 étiquettes NFC (ISO 14443, Mifare-NFC, 13.56 MHz, intégrant 1 Kio de RAM) placées dans différents endroits du musée, soit avec une des autres équipes (via des communications pair-à-pair NFC). Pour réduire les risques de verrous mortels entre les joueurs (qui pourraient advenir si deux équipes font la même collection d'objets), chaque carte existe en trois exemplaires. Au final, il y a dans le jeu 4 familles de 4 cartes, chacune existant en 3 exemplaires, soit $4 \times 4 \times 3 = 48$ cartes au total. Au début d'une session de jeu, les cartes sont mélangées et distribuées entre les 8 téléphones mobiles (4 cartes par mobile) et les 16 étiquettes NFC (1 carte par étiquette).

PSM spécialise les structures de données présentées en section 10.2 de la manière suivante (cf. figure 10.2). La mémoire répartie DM n'est constituée que d'un octet par

étiquette (la valeur de cet octet correspond à la carte stockée dans l'étiquette)[2]. $DM_e[r]$ (e étant un élément du système et r étant une étiquette) est un octet stockant une valeur entre 0 et 15 (chaque valeur correspondant à l'une des 16 cartes). $VC_e[r]$ est une valeur de type `short` (deux octets). Elle contient l'estampille temporelle de la dernière requête de lecture ou de mise à jour dont e a connaissance à propos de $DM_r[r]$. En effet, nous pouvons utiliser les optimisations présentées en section 10.2 du fait que notre jeu présente les deux caractéristiques suivantes. La première est que les horloges des téléphones mobiles que nous utilisons sont synchronisées manuellement au début de chaque journée d'utilisation de l'application[3] ; nous avons vérifié expérimentalement que les horloges des différents téléphones mobiles ne dérivent pas de plus d'une seconde pendant la journée. La seconde caractéristique est qu'il faut au moins 10 secondes pour que deux joueurs échangent une carte virtuelle avec la même étiquette ou qu'ils fassent une requête à la même étiquette. Comme 10 secondes est une période beaucoup plus grande qu'une période d'une seule seconde : 1) nous pouvons appliquer ces optimisations ; 2) cette estampille temporelle peut être stockée sous la forme du nombre de secondes écoulées depuis le début du jeu[4]. Cette estampille temporelle *spécifique* à chaque élément de VC_e est complétée par une estampille temporelle *commune* à tous les élements de VC_e. Cette dernière contient la date/heure de début de la session de jeu (8 octets codant le nombre de millisecondes écoulées depuis le 1^{er} janvier 1970). Ainsi, pour réinitialiser les données en vue d'une nouvelle session, il suffit de fixer $DM_{m,m\in\{mobiles\}}$ aux valeurs initiales des 16 cartes, fixer la date/heure commune des mobiles à l'heure de début de session, et les différents dates/heures spécifiques des mobiles à 0.

En phase de jeu, quand une équipe souhaite échanger une des cartes de son téléphone mobile avec la carte présente sur une étiquette, elle doit se déplacer physiquement jusqu'à l'étiquette. L'application sur le téléphone mobile applique alors l'algorithme 20. De plus, quand une équipe souhaite connaître la carte virtuelle qui est réellement stockée sur l'étiquette, elle doit aussi se déplacer physiquement jusqu'à l'étiquette. Nous profitons de cette opération de lecture pour appliquer l'algorithme 21 et rendre ainsi cohérent DM_{mobile} et $DM_{étiquette}$. De ce fait, alors que l'équipe pense qu'il y a seulement une opération de *lecture* (pour afficher la carte virtuelle stockée sur l'étiquette), il y aussi une opération d'*écriture* qui modifie éventuellement $DM_{étiquette}$ et $VC_{étiquette}$. L'algorithme 21 est aussi appliqué dans le cas où deux équipes échangent des cartes virtuelles (via protocole pair-à-pair NFC) : cela met en cohérence DM_{mobile_1} et DM_{mobile_2}.

Pour aider les joueurs dans leur recherche de cartes virtuelles adaptées, la conception du jeu (*game design* en anglais) introduit la fonction *indice* : toute équipe peut demander à son téléphone mobile une indication d'une étiquette NFC qui contient une carte virtuelle adaptée à sa collection. La fonction *indice* est mise en œuvre en analysant DM_{mobile} et

2. Nous présentons ici la mémoire répartie dont l'expérimentation est décrite en section 10.4. Dans la version actuelle de l'application, DM inclut aussi 4 octets par téléphone mobile (chaque octet représentant l'une des cartes stockée dans le mobile). La fonction *indice* (évoquée dans la suite de cette section) permet alors non seulement de chercher un indice parmi les étiquettes du système, mais également parmi les téléphones mobiles.

3. Les Nokia 6131 NFC que nous utilisons sont incapables de se synchroniser avec le réseau de l'opérateur qui détient la carte SIM éventuellement utilisée.

4. Comme une session de jeu dure au plus 85 minutes (5100 secondes), il n'y a aucun risque de dépassement de capacité du `short`.

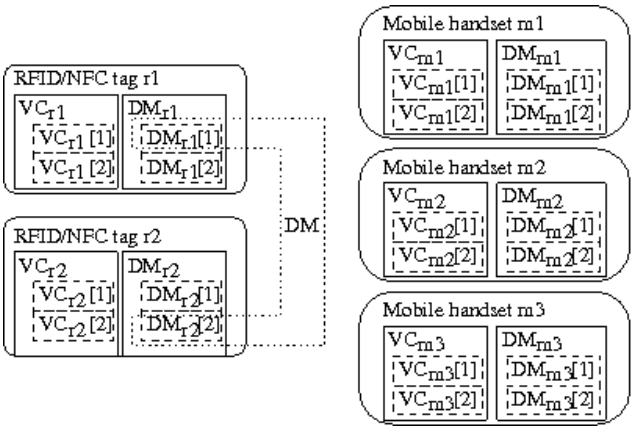


FIGURE 10.2 – Données présente dans le jeu PSM (pour simplifier la figure, nous n'avons représenté que 2 étiquettes RFID et 3 téléphones mobiles)

VC_{mobile}. Elle considère les cartes virtuelles de DM_{mobile} qui correspondent à la famille collectionnée par l'équipe. Parmi celles-ci, la fonction sélectionne la carte que l'équipe ne possède pas déjà et pour laquelle l'information dans VC_{mobile} est la plus récente (cette heuristique est basée sur l'intuition que, plus cette information est récente, plus il est probable que la carte virtuelle est toujours dans l'étiquette NFC). En fait, cette fonction *indice* fournit 2 types de résultat. Le premier type est l'indication d'une étiquette r et de la carte $DM_{mobile}[r]$ que r contenait, il y a $dateHeureCourante - VC_{mobile}[r]$ unités de temps. Le second type de résultat est le message « Pour le moment, aucune étiquette ne contient une carte qui vous intéresse. » Ce message est affiché quand aucune des cartes de DM_{mobile} ne satisfait les besoins de l'équipe.

Les paragraphes suivants décrivent les spécificités du *game design* qui favorise la dissémination des données liées à la RDSM.

Nous l'avons vu précédemment : à chaque fois qu'un téléphone mobile est en contact avec un autre élément du système (que ce soit une étiquette ou un autre téléphone), nous appliquons l'algorithme 21 pour rendre cohérents DM_{mobile} et $DM_{etiquette\_ou\_mobile}$. Comme demander un indice coûte des points, les équipes sont poussées à plutôt se rapprocher des étiquettes pour consulter leur contenu. Nous stimulons donc la dissémination de données via les étiquettes. Par ailleurs, une équipe marque des points à chaque fois qu'elle échange des cartes avec une autre équipe : l'équipe est donc motivée pour faire de tels échanges. Cela permet non seulement de favoriser les interactions entre les équipes [Gentes *et al.*, 2009b], mais cela stimule la dissémination de données via les mobiles.

À propos des échanges entre les équipes, une règle empêche deux équipes de passer leur session PSM à s'échanger des cartes entre elles (et donc récupérer, à chaque échange, les points correspondants) : une équipe n'a le droit d'échanger avec une autre équipe qu'au plus 2 cartes dans une fenêtre glissante de 10 minutes. Cette règle limite le nombre d'échanges faits par une équipe pendant une session. Ainsi, dans le cas d'une session de 85 minutes, une équipe peut faire au plus $(85/10+1)\times 2 = 18$ échanges. Bien que cette règle limite le nombre d'échanges, elle a en fait un effet positif sur la dissémination de données. En effet, en son absence, au moins deux équipes pourraient être tentées de passer leur temps à s'échanger des

cartes entre leurs téléphones (puisque chaque échange entre mobiles rapporte des points). De ce fait, au moins deux téléphones mobiles ne contribuerait pas à la propagation de données vers d'autres mobiles ou d'autres étiquettes. Dit autrement, au moins 2 mobiles sur 8 ne serviraient plus à la tâche de dissémination d'information : nous perdrions au moins $2/8 = 25\%$ de notre capacité « réseau ».

Cette section a présenté le jeu que nous avons développé comme application exemple de la RDSM. La prochaine section présente les résultats expérimentaux que nous avons observés avec ce jeu.

10.4 Expérimentation de cette mémoire

En Juin 2009, au cours de la manifestation *Futur en Seine*, cinq sessions publiques de PSM ont été organisées au *Musée des arts et métiers* (Paris). Chacune a été jouée avec 6 équipes. Dans cette section, nous commençons par présenter la méthodologie employée pour obtenir des données concernant ces sessions. Puis, nous analysons certaines de ces données. Nous montrons notamment le bien-fondé des mécanismes mis en place dans l'application pour stimuler le réseau humain des utilisateurs qui permet à notre RDSM de fonctionner.

Pendant chaque session, chaque téléphone mobile enregistre tous les événements générés par l'équipe l'utilisant et les horodate. Le type d'événement et l'estampille temporelle sont les seules informations communes à tous les événements. Les autres informations enregistrées dépendent du type de l'événement. Voici quelques détails sur les événements concernant la suite de cette section :

- pour l'événement `Application start` (début d'application), nous enregistrons l'identifiant de session de jeu, la famille collectionnée par l'équipe, les cartes virtuelles qu'a reçues le téléphone, DM_{mobile} et VC_{mobile} ;
- pour l'événement `Consult a tag` (consultation d'une étiquette), nous enregistrons l'identifiant de l'étiquette, la carte virtuelle stockée dans l'étiquette, DM_{mobile} et VC_{mobile} après application de l'algorithme 21 pour mettre en phase DM_{mobile} et $DM_{étiquette}$;
- pour l'événement `Exchange with a tag` (échange avec une étiquette), nous enregistrons l'identifiant de l'étiquette, la carte virtuelle donnée et celle reçue ;
- pour l'événement `Exchange with another team` (échange avec une autre équipe), nous enregistrons l'identifiant de l'autre téléphone mobile, la carte virtuelle donnée et celle reçue, DM_{mobile} et VC_{mobile} après application de l'algorithme 21 pour mettre en phase DM_{mobile} et $DM_{mobile\ autre\ equipe}$;
- pour l'événement `Hint request` (demande d'indice), nous enregistrons la famille collectionnée par l'équipe, le type d'indice fourni, et l'information transmise à l'équipe (dans le cas d'un résultat du premier type).

À la fin d'une journée de tests, tous les fichiers contenant ces événements sont transférés des téléphones mobiles vers un ordinateur. Sur ce dernier, nous exécutons une application Java. Elle ordonne tous les événements selon leur estampille temporelle. Ensuite, elle compte les différents événements. Enfin, elle construit un historique de l'état du système. Cet état est constitué des cartes virtuelles stockées dans les différents téléphones mobiles et dans les différentes étiquettes. L'état concernant les cartes contenues dans chaque té-

léphone mobile est initialisé en analysant les événements `Application start`. Il évolue à chaque fois qu'il y a un événement `Exchange with a tag` or `Exchange with another team`. L'état concernant les étiquettes est initialisé quand on analyse le premier événement `Consult a tag` concernant l'étiquette. Il évolue à chaque fois qu'il y a un événement `Exchange with a tag`.

Du fait que nous observons un système réparti, il est quasiment certain que l'historique que nous obtenons ne correspond pas à ce qui s'est passé dans la réalité. La section 10.3 a présenté les propriétés de PSM concernant le temps. Elles induisent qu'avec l'historique des événements que nous obtenons, nous avons la garantie que la suite d'événements advenue sur chaque étiquette et chaque mobile est correcte. C'est l'entrelacement de ces événements entre les différents éléments du système qui peut ne pas avoir existé dans la réalité [Chandy et Lamport, 1985].

Pendant les 5 sessions de jeu, 30 téléphones mobiles ont été utilisés pour des sessions de jeu qui ont duré entre 70 et 85 minutes. 3 889 événements ont été enregistrés. Ils ont permis de recenser 590 consultations simples d'une étiquette, 279 échanges entre un téléphone mobile et une étiquette, et 142 échanges entre deux téléphones mobiles.

14 demandes d'indices ont été recensées[5]. Comme notre application d'analyse d'événements construit un historique de l'état de l'ensemble du système, à chaque fois qu'il y a un événement `Hint request`, nous pouvons évaluer le bien-fondé de l'indice. 8 sur 9 indices de premier type (il y a u unités de temps, une étiquette r contenait une carte c) ont été corrects. De plus, 3 sur 5 indices du second type (aucune étiquette ne contient une carte intéressant le joueur) ont été corrects. Cela signifie que $(8 + 3)/(9 + 5) = 79\%$ des indices étaient corrects. Mais ce pourcentage est calculé seulement sur 14 données : nous ne pouvons qu'être soupçonneux par rapport à sa représentativité.

Pour avoir un échantillon plus grand, nous avons modifié l'application d'analyse d'événements : à chaque fois qu'elle analyse un événement `Consult a tag`, l'application génère un événement artificiel `Hint request`. Pour ce faire, elle applique l'algorithme que le mobile aurait appliqué si l'équipe avait effectivement déclenché la demande d'indice à ce moment-là. Avec ces événements générés, nous observons maintenant que 490 (respectivement 30) sur 540 (respectivement 50) des indices de premier (respectivement second) type sont corrects. Cela signifie que $(490 + 30)/(540 + 50) = 88\%$ des indices sont corrects ! Bien sûr, 88% est beaucoup moins que les 100% d'indices corrects que nous aurions obtenus avec une architecture centralisée. Mais nous obtenons 88% sans installation, ni coût opérationnel d'un réseau global. Notez que l'acceptabilité de ce taux d'indices corrects est dépendant de l'application. Dans le cas de notre jeu, la fonction d'indice a été présentée comme étant une sorte de pari : un taux de 88% est acceptable.

Intuitivement, on pourrait penser qu'un indice récent a plus de chances d'être correct qu'un indice ancien. Le début de la table 10.1 (jusqu'à la ligne « $[30, 34]$ ») confirme cette conjecture. Mais nous n'avons pas d'explication satisfaisante pour les forts pourcentages observés dans les dernières lignes de cette table.

5. Ce nombre de demandes d'indice est faible probablement à cause de la conception du jeu : demander un indice coûte des points à une équipe. De plus, lors de la formation des joueurs, nous indiquions qu'un indice n'était pas sûr à 100 %. C'est pour toutes ces raisons que nous soupçonnons les joueurs de ne pas avoir demandé souvent un indice.

TABLE 10.1 – Relation entre le taux d'indice correct de premier type et l'âge de l'indice

Âge de l'indice (en minutes)	Nombre de demandes d'indices	Nombre d'indices corrects	% indices corrects
[0, 4]	176	178	99%
[5, 9]	59	65	91%
[10, 14]	50	61	82%
[15, 19]	55	64	86%
[20, 24]	32	35	91%
[25, 29]	7	8	88%
[30, 34]	14	18	78%
[35, 39]	15	15	100%
[40, 44]	12	12	100%
[45, 49]	7	8	88%
[50, 54]	7	8	88%
[55, 59]	10	10	100%
[60, 64]	6	7	86%
[65, 69]	10	17	59%
[70, 74]	11	13	85%
[75, 79]	10	12	83%
[80, 84]	5	5	100%
[85, 85]	4	4	100%

Un autre facteur de l'exactitude d'un indice est le nombre d'étiquettes qui sont averties d'un changement dans une certaine étiquette (cf. figure 10.3) et le temps nécessaire pour disséminer cette information (cf. figure 10.4). Il est relativement fréquent qu'au moins 3 étiquettes soient notifiées d'un changement dans une étiquette (cela arrive avec une fréquence de 39%, cf. figure 10.3). En revanche, la fréquence de notification de 12 étiquettes n'est plus que de 9%. Cela est dû au fait que la notification de changement dans une étiquette met du temps à se répandre dans les autres étiquettes.

La figure 10.4 montre, par exemple, que, pour avoir 3 étiquettes notifiées d'un changement, il faut au plus 5 minutes dans 50% des cas. De ce fait, pendant que l'information concernant un changement dans une étiquette r se répand, la probabilité que r reçoive une nouvelle carte virtuelle augmente. En cas de nouveau changement, cette nouvelle information se répand aussi, remplaçant l'ancienne information. Cela explique pourquoi il est très rare (fréquence de 3%, cf. figure 10.3) que l'ensemble des 16 étiquettes soient notifiées d'une information de changement d'une étiquette.

La dissémination d'une nouvelle valeur prend du temps. Il est possible qu'à un moment donné, la nouvelle valeur d'une étiquette se répande parmi les éléments du système, pendant que l'ancienne valeur continue à se répandre, mais auprès d'autres éléments. C'est pourquoi il peut arriver qu'une étiquette soit notifiée d'une information périmée. Pour évaluer ce dernier phénomène, à chaque fois qu'un élément e est notifié d'un changement de valeur dans l'étiquette r, nous comparons l'estampille temporelle t_{notif} de notification avec l'estampille temporelle t_{chang} du prochain changement de l'étiquette r.

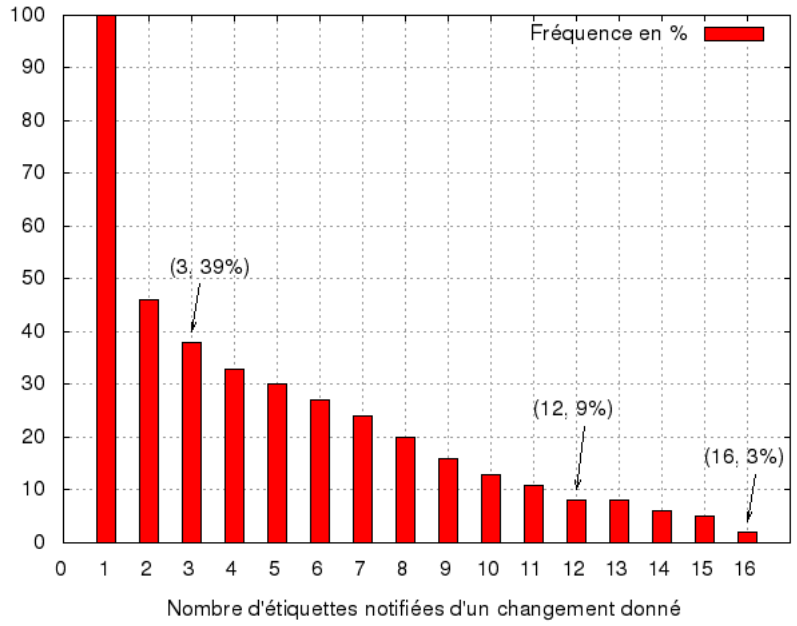


FIGURE 10.3 – Fréquence du nombre d'étiquettes (y compris l'étiquette concernée) notifiées d'un changement

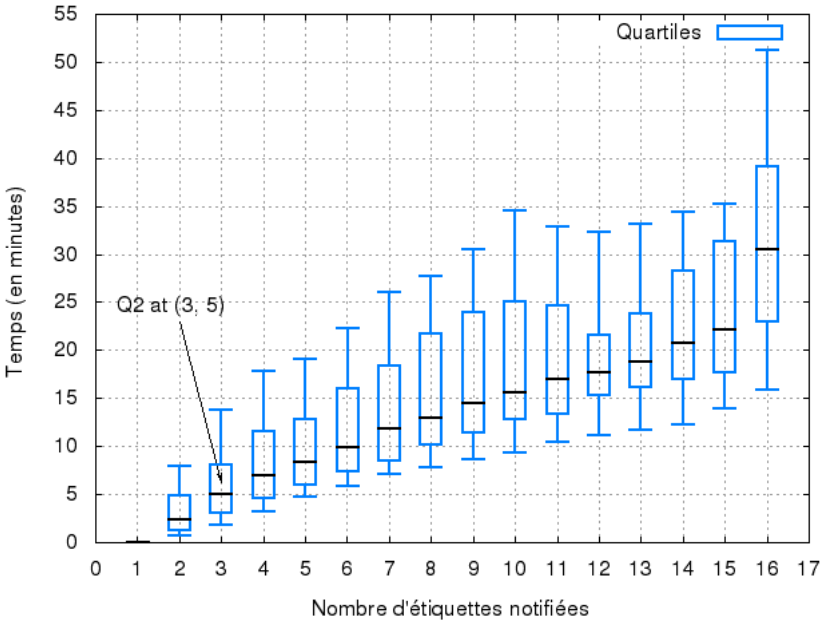


FIGURE 10.4 – Boîte à moustache du temps (en minutes) requis pour qu'un certain nombre d'étiquettes soient notifiées d'un changement (décile D1, quartiles Q1, Q2, Q3, et décile D9)

Définition 8. *Nous appelons* période de validité *le nombre :* $T_{valid} = t_{chang} - t_{notif}$.

Si $T_{valid} \leqslant 0$, le changement de valeur de r qui vient d'être notifié à e est périmé. Si $T_{valid} > 0$, le changement de valeur de r signalé est une information récente.

Par exemple, dans la figure 10.5, au temps t_1, le mobile bleu écrit EFG sur l'étiquette r_1. Au temps t_2 (respectivement t_4), il écrit sur r_2 la valeur RST et l'information « r_1 contient EFG » (respectivement sur r_3 la valeur MNO et les informations « r_1 contient EFG » et « r_2 contient EFG »). Quant au mobile orange, en t_3 (respectivement t_5), il écrit la valeur HIJ (respectivement UVW et l'information « r_1 contient HIJ ») sur l'étiquette r_1 (respectivement r_2). En ce qui concerne les informations écrites par le mobile bleu en t_4 :

- pour l'information concernant r_1, $t_{notif} = t_4$ et $t_{chang} = t_3$. Donc $T_{valid} \leqslant 0$: l'information « r_1 contient EFG » est périmée ;
- pour l'information concernant r_2, $t_{notif} = t_4$ et $t_{chang} = t_5$). Donc $T_{valid} > 0$: l'information « r_2 contient RST » est à jour.

Notez que la période de *validité* ne permet pas de capturer la durée d'*invalidité* d'une information[6]. Ainsi, dans la figure 10.5, l'information « r_1 contient EFG » est valide pendant $t_3 - t_2$ (ce qui est bien capturé par la notion de période de validité que nous avons définie), mais invalide pendant la période $t_5 - t_3$.

Les résultats sur les périodes de validité sont résumés dans la figure 10.6[7]. Dans 29% des cas, $T_{valid} \leqslant -1$. Dit autrement, dans 29% des cas, nous disséminons une information périmée. Dans une mémoire répartie partagée utilisée pour une application massivement parallèle, ce pourcentage serait inacceptable, ne serait-ce que parce qu'il est non nul. Dans le cas de notre jeu, comme l'utilisateur sait qu'il fait une sorte de pari en demandant un indice, ce pourcentage devient acceptable, même s'il atteint 29%.

La figure 10.7 évalue l'impact de la dissémination d'informations périmées sur la perception de l'état global par un mobile. Pour l'obtenir, pour chaque événement `Consult a tag`, nous comptons le nombre de différences (c'est-à-dire le nombre d'étiquettes périmées) entre DM_{mobile} et l'état réel. La figure 10.7 montre que DM_{mobile} ne contient aucune étiquette périmée dans 6% des cas. Dit autrement, DM_{mobile} contient au moins une étiquette périmée dans au moins $100 - 6 = 94\%$ des cas. D'un autre côté, DM_{mobile} contient au moins 5 (respectivement 7) étiquettes correctes dans 100% (respectivement 99.8%) des cas. Dans le contexte de notre jeu, ces pourcentages permettent de mieux comprendre la différence de taux d'indices corrects entre les deux types de résultat. En effet, le taux du premier (respectivement second) type vaut $490/540 = 91\%$ (respectivement $30/50 = 60\%$). C'est parce que, pour le premier type, nous considérons seulement la valeur d'une carte parmi 16. Comme DM_{mobile} contient au moins 12 étiquettes correctes dans 71% des cas, la probabilité que la carte retenue soit correcte est élevée. En revanche, pour le second type, la décision est prise sur la base de l'ensemble des cartes de DM_{mobile}. La probabilité d'une réponse correcte ne peut être que plus faible.

6. Dans la littérature, le mot « fraîcheur » (respectivement « péremption ») est parfois utilisé en lieu et place de « validité » (respectivement « invalidité »).

7. La période de validité +85 a une signification spéciale : la carte virtuelle stockée dans l'étiquette r au moment t_{chang} n'a plus jamais changé jusqu'à la fin de la session de jeu (qui peut durer au plus 85 minutes).

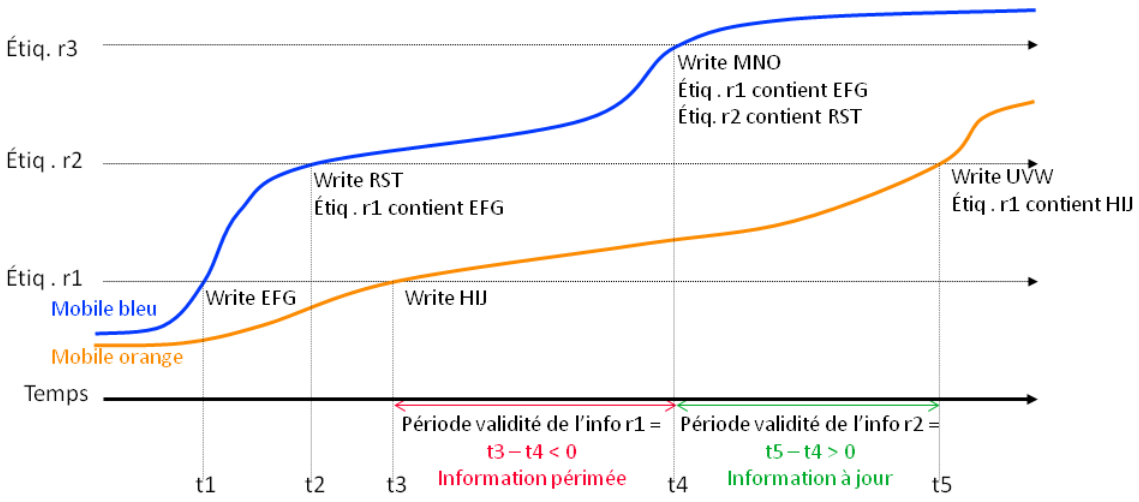


FIGURE 10.5 – Notion de période de validité

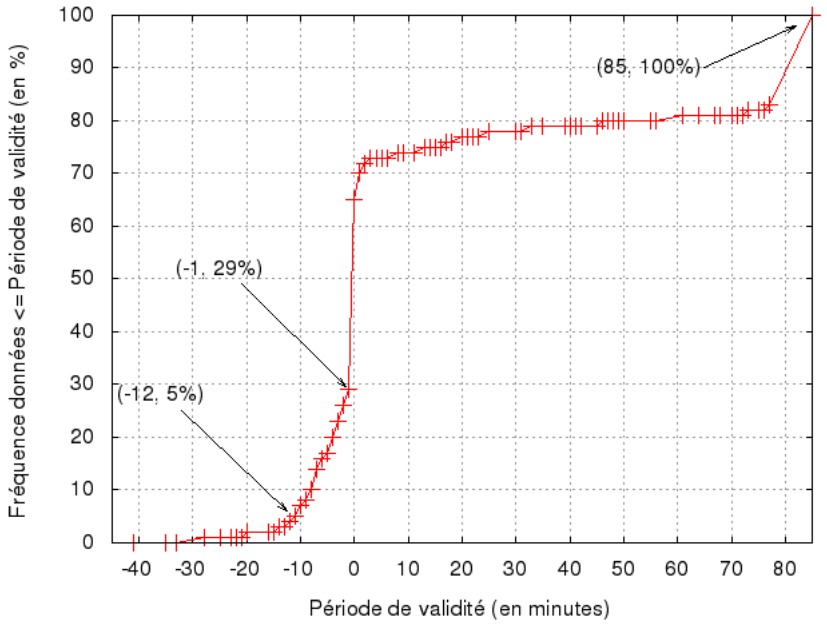


FIGURE 10.6 – Fréquence des périodes de validité (T_{valid})

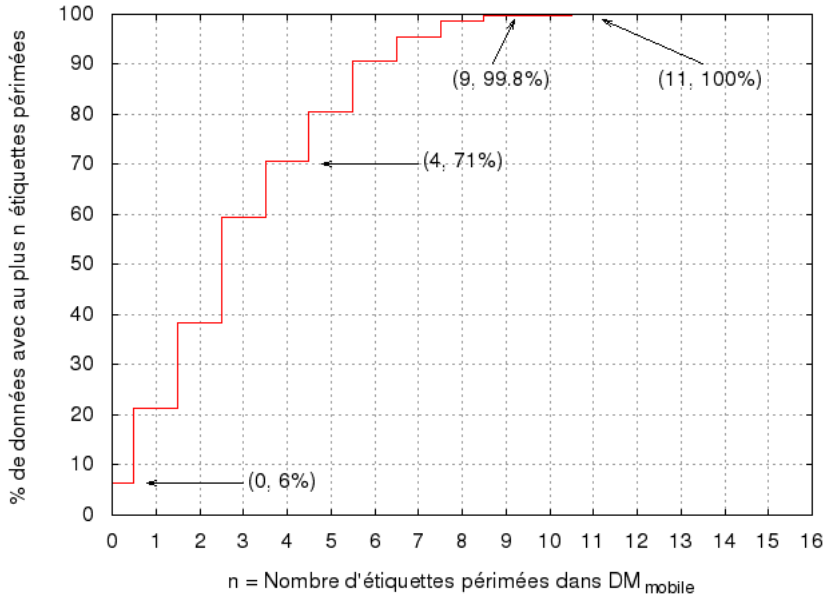


FIGURE 10.7 – Fréquence (en %) d'observation d'au plus n étiquettes périmées dans DM_{mobile}

Cette section a présenté les résultats expérimentaux issus de cinq sessions de jeu publiques. En résumé, au moins 3 étiquettes ont été notifiées d'un changement dans 39% des cas. La dissémination de cette notification a pris au plus 5 minutes dans 50% des cas. De plus, dans 29% des cas, les notifications reçues par une étiquette étaient périmées. Malgré ce pourcentage, la vue détenue par chaque mobile contenait au plus 4 étiquettes périmées dans 71% des cas. Outre ces chiffres « système », nous avons observé un taux « applicatif » d'indices corrects de 88%. Intuitivement, ce taux npous semble acceptable par les utilisateurs de l'application dans le contexte de notre jeu (vu que la fonction d'indice était présentée comme étant une sorte de pari). Mais nous n'avons mené aucune enquête sociologique pour confirmer ou infirmer cette intuition.

La prochaine section analyse la RDSM avec un point de vue plus théorique.

10.5 Discussion

Cette section décrit les travaux apparentés à notre proposition. Puis elle s'intéresse aux limitations de la RDSM.

10.5.1 Travaux apparentés

Comme mentionné précédemment, la RDSM s'appuie sur la théorie des vecteurs d'horloges [Mattern, 1988, Fidge, 1988]. De plus, notre utilisation des vecteurs d'horloges peut être assimilée à un protocole de bavardage (*gossip protocol* en anglais). En effet, nos algorithmes satisfont la plupart des conditions définies dans [Birman, 2007] pour considérer qu'un protocole est un protocole de bavardage. Par exemple, quand un mobile interagit avec une étiquette ou un autre mobile, l'état de chacun change d'une manière qui reflète l'état de l'autre. Le type de protocole de bavardage mis en œuvre dans la RDSM est un protocole anti-entropique pour réparer des données répliquées [Demers *et al.*, 1987]. Les données sont contenues dans la mémoire répartie DM. Elle est répliquée sur tous les éléments du système. Les utilisateurs de l'application forment le réseau : ce sont eux qui transportent, à l'aide de leur téléphone mobile, l'information d'un élément à l'autre.

Quittons la communauté *Systèmes répartis* pour la communauté *Réseaux*. La RDSM s'appuie sur des mécanismes proches de ceux des *Delay-Tolerant Networks* (DTN) [Cerf *et al.*, 2007]. En effet, ce domaine de recherche étudie les communications où il n'existe pas de liaison de bout en bout pour une session de communication.

Le réseau *ZebraNet* est un premier exemple de DTN [8] [Juang *et al.*, 2002]. *ZebraNet* est un réseau de capteurs mobiles pour collecter des données sur des zèbres vivant dans une réserve. Pour ce faire, un capteur est attaché à chaque animal. Il collecte des données localement et les transmet dès qu'un autre capteur passe à portée. Ainsi des zoologistes peuvent récupérer des données sur un grand nombre d'animaux, alors qu'ils ne sont à portée de communication que de quelques zèbres. Notre contribution est d'intégrer des étiquettes passives à une telle architecture pair-à-pair. Certes, ces étiquettes n'ont pas de capacité de transmission, mais elles ont une capacité de stockage d'information. Et, si elles

8. [Juang *et al.*, 2002] qualifie *ZebraNet* d'« inondation opportuniste » (*opportunistic flooding*). Mais en fait, selon la terminologie d'aujourd'hui, c'est bien un DTN.

se déplacent (cas possible avec les zèbres) ou bien si différentes personnes viennent les voir (cas des bornes du jeu PSM), elles peuvent faire suivre ces informations à un autre lecteur RFID.

Cette idée rejoint celle de [Simner, 2007] qui propose d'étudier un DTN à base d'étiquettes NFC correspondant aux badges Mifare portés par les étudiants de l'université de Cambridge. [Simner, 2007] implémente un simulateur destiné à calculer la bande passante, la latence et le taux de perte de données dans un tel réseau.

La RDSM semble n'être qu'un cas particulier de DTN puisqu'elle met en œuvre un ensemble d'entités qui passent leur temps à s'échanger des messages indiquant leur vision globale du système. Toutefois, ne voir la RDSM que comme un DTN nous semble réducteur. En effet, un telle vision fait oublier que la RDSM offre des propriétés d'architecture système qui en font, pour certaines applications, l'unique architecture alternative aux architectures centralisée, semi-répartie et répartie classiquement utilisées dans le monde de la RFID (cf. chapitre 11).

Cette section a étudié les travaux apparentés à la RDSM. Intéressons-nous maintenant à ses limites

10.5.2 Analyse des limites

Cette section étudie les limites de la RDSM en termes de péremption et d'extensibilité.

10.5.2.1 Limites liées à la péremption

Pour tout élément e du système, DM_e peut contenir des données périmées de trois types différents.

Premièrement, il peut y avoir un élément $e', e' \neq e$ pour lequel $DM_e[e']$ ne correspond pas à la valeur $DM_{e'}[e']$ actuellement stockée au niveau de e'. Cela est arrivé dans 94% des cas de notre expérience PSM (cf. figure 10.7).

Deuxièmement, la RDSM subit le problème inhérent à l'utilisation de vecteurs d'horloges dans un système réparti [Mattern, 1988, Fidge, 1988] : l'élément e n'a aucun garantie que DM_e corresponde à une valeur de DM qui a existé pendant la vie du système [Chandy et Lamport, 1985].

Troisièmement, l'élément e n'a aucune garantie de voir toute l'histoire des changements de $DM_{e'}[e']$ (pour tout élément e' du système). Par exemple, supposons qu'au temps t_1, un mobile m_1 affecte la valeur v_{t_1} à $DM_{e'}[e']$: l'algorithme 20 conduit à $VC_{m_1} \leftarrow t_1$. Puis, au temps t_2, le mobile m_2 affecte la valeur v_{t_2} à $DM_{e'}[e']$: $VC_{m_2} \leftarrow t_2$. Ensuite, si m_2 interagit avec l'élément e'' avant m_1, quand m_2 applique l'algorithme 21, on a : $DM_{e''}[e'] \leftarrow v_{t_2}$ et $VC_{e''}[e'] \leftarrow t_2$. Quand m_1 interagit avec e'', vu que $VC_{m_1}[e'] < VC_{e''}[e']$, $DM_{e''}[e']$ ne reçoit jamais la valeur v_{t_1}.

La suite de cette section se concentre sur les données périmées du premier type, pour lesquelles des actions peuvent être prises.

Le taux de données périmées et la période durant laquelle une donnée reste périmée est dépendante de l'application. À quelle fréquence est-ce que les données évoluent ? Quelle

est la fréquence de visite des différentes étiquettes (en particulier, les étiquettes qui sont excentrées par rapport aux autres) ? À quelle fréquence des mobiles se rencontrent-ils ? Combien de temps met un utilisateur pour aller d'une étiquette à une autre ? Etc. Dans le contexte de PSM, le plus fort taux de données périmées que nous avons observé est 11/16 = 69% (cf. figure 10.7). Concernant la période, 5% des données sont restées périmées pendant plus de 12 minutes (cf. figure 10.6).

Si le taux et/ou la période sont trop grands pour l'application, trois méthodes peuvent être envisagées pour les réduire. Les deux premières consistent à stimuler le « réseau ». La troisième s'appuie sur un réseau global informatique.

La première méthode consiste à avoir un utilisateur dédié qui passe périodiquement par toutes les étiquettes pour lire leur contenu et surtout mettre à jour leur $DM_{étiquette}$ et $VC_{étiquette}$. Ainsi, nous réduisons le risque d'avoir des étiquettes qui ne sont pas rafraîchies pendant une longue période. Dans le cas de PSM, cette méthode ne serait pas fructueuse. En effet, vu la taille du *Musée des arts et métiers*, cet utilisateur dédié aurait besoin de 12 minutes pour passer par toutes les étiquettes. Cette méthode n'enlèverait donc que $Frequence\ Periodes\ Validite\{T_{valid} \leqslant -12\} = 5\%$ de dissémination de données périmées (cf. figure 10.6). Cela réduirait le nombre total de disséminations négatives de seulement 5%/29% = 17%. Ce taux est trop réduit pour justifier un utilisateur dédié (surtout pour un jeu).

Une autre méthode consiste à demander périodiquement à tous les utilisateurs de se réunir pour synchroniser leur DM_{mobile}. Cette méthode (qui rappelle le fonctionnement de l'architecture semi-répartie) induit des contraintes sur les utilisateurs qui risquent de ne pas l'accepter. Dans le contexte de PSM, elle est inapplicable : elle casserait l'ambiance et la dynamique du jeu.

La dernière méthode est une approche hybride. Chaque mobile utilise les algorithmes 20 et 21. Mais parfois, dans des circonstances à définir et probablement dépendantes de l'application, chaque mobile se connecte à un serveur via un réseau informatique global. Par application de l'algorithme 21, il synchronise DM_{mobile} avec $DM_{serveur}$. Cette approche introduit un réseau informatique global et les coûts associés. Mais, elle garantit un usage modéré de ce réseau informatique global (ses coûts seraient donc réduits). Il n'empêche : pour des applications (comme PSM), elle est inapplicable.

Cette section a étudié les limites liées à la péremption des données de la RDSM. L'architecture semi-répartie a aussi des limites de péremption que nous étudions au chapitre 11. Auparavant, nous étudions les limites d'extensibilité de la RDSM.

10.5.2.2 Limites d'extensibilité

Le nombre de données que peut gérer la RDSM dépend du matériel utilisé.

Lemme 28. *Soit TH la borne supérieure du temps acceptable pour qu'un mobile m lise DM_r et VC_r d'une étiquette r, exécute l'algorithme 21, et écrive la nouvelle valeur de DM_r et VC_r sur r. Ce temps est une donnée applicative. Ainsi, dans le contexte de* PSM, *pendant nos expériences, nous avons observé que les joueurs devenaient stressés vers la fin des sessions. À ce moment-là, ils passaient leur temps à aller d'une étiquette à l'autre, posant leur téléphone sur l'étiquette pour voir son contenu et l'enlevant en moins d'une*

demi-seconde. Dans ce laps de temps, le mobile doit avoir effectué toutes les opérations récessaires à la mise en phasde entre DM_m et DM_r ;

soit TR le débit de transmission de données entre l'étiquette et le mobile ;

soit L la longueur totale d'un élément de DM_r et de VC_r ;

soit S, la taille de la RAM (en octets) disponible sur l'étiquette ;

soit N le nombre d'éléments dans DM et donc dans DM_r ;

alors : $N \leqslant \min(\frac{TH \cdot TR}{2 \cdot L}, \frac{S}{L})$

Démonstration. Quand un mobile m interagit avec une étiquette r, m lit DM_r et VC_r en $L \cdot N/TR$ secondes. Puis, m exécute l'algorithme 21 en un temps que nous négligeons devant le temps de lecture et écriture de l'étiquette. Enfin, m écrit la nouvelle valeur de DM_r et VC_r sur r en $L \cdot N/TR$ secondes. Le temps total de l'opération doit être inférieur à TH. Donc, $2 \cdot L \cdot N/TR \leqslant TH$. Par conséquent, N est contraint par l'inégalité suivante : $N \leqslant \frac{TH \cdot TR}{2 \cdot L}$.

Par ailleurs, le temps de mise à jour de DM_r et VC_r est beaucoup plus grand que le temps requis pour que deux mobiles s'échangent leur DM_{mobile} et VC_{mobile}. L'échange d'informations entre mobiles par pair-à-pair NFC n'apporte donc pas de nouvelles contraintes.

En revanche, le matériel est limité en capacité. N est donc aussi contraint par : $N \leqslant \frac{S}{L}$.

On conclut :$N \leqslant \min(\frac{TH \cdot TR}{2 \cdot L}, \frac{S}{L})$ □

Le lemme 28 quantifie l'influence des optimisations présentées dans la section 10.2 sur la borne supérieure de N.

Dans le contexte de PSM, on a $TH = 0.5$ secondes, $TR = 106$ kbps (nous utilisons des tags ISO 14443), $L = 3$ octets [9], and $S = 752$ octets. En effet, si PSM utilise des étiquettes Myfare de 1 Kio, une partie de cette NVRAM est réservée. Ces 1 Kio sont répartis en 16 secteurs de 4 blocs, chacun faisant 16 octets. Pour le secteur 0, le bloc 0 est réservé. Pour tous les secteurs, le bloc 3 n'a qu'un octet de disponible, si on souhaite utiliser les 2 clés d'accès. Or, comme l'explique [Simner, 2007], y accéder n'est pas rentable d'un point de vue performances. Au final, seuls 752 des 1024 octets de l'étiquette sont exploitables. Appliquons le lemme 28 : $N \leqslant \min(\frac{0{,}5 \cdot \frac{106 \cdot 10^3}{8}}{2 \cdot 3}, \frac{752}{3}) = \min(1\,104, 250) = 250$.

10.6 Conclusion

Ce chapitre a présenté la mémoire répartie partagée à base de RFID que nous avons conçue et expérimentée. L'étude des limites en termes d'extensibilité a montré que notre RDSM n'est pas applicable au delà de quelques centaines d'éléments. Cette extensibilité est acceptable pour des applications comme PSM. Mais, elle ne convient pas à nombre d'applications et, en particulier, aux applications classiquement gérées par l'architecture centralisée (qui est capable de gérer des millions, voire des milliards d'étiquettes).

Le chapitre suivant compare plus avant les différentes architectures RFID pour établir un guide de leur(s) champ(s) d'application.

9. On néglige ici les 8 octets consommés par la date/heure commune de PSM.

Chapitre 11

Comparaison des différentes architectures RFID

11.1 Introduction

La technologie RFID permet de faire le lien entre les *entités physiques* du monde réel et leur *avatar* dans le monde virtuel. L'entité physique peut être une localisation, un objet, une plante, un animal ou un être humain. Une étiquette RFID lui est physiquement attachée. L'avatar correspond à l'ensemble des données informatiques associées à l'entité physique. Par exemple, considérons le cas d'une étiquette RFID attachée à un arbre. L'arbre est l'entité physique. Son avatar contient la variété de l'arbre, la circonférence de son tronc, la liste des interventions qui ont été menées sur cet arbre, etc.

Quand il conçoit une application RFID, un architecte système dispose de trois endroits pour stocker l'avatar : une base de données centrale, une base de données locale au lecteur RFID, l'étiquette RFID elle-même. Chacun de ces endroits s'intègre dans un patron d'architecture [1] basée sur de la RFID. Mais comment choisir la bonne architecture ? Quels sont les attributs applicatifs qui doivent être pris en compte pour faire le bon choix ?

L'état de l'art du chapitre 9 ne fournit pas de réponses satisfaisantes. En effet, quand un article décrit une architecture basée sur de la RFID, il n'évoque pas les attributs applicatifs qui ont conduit à choisir cette architecture. Par ailleurs, des livres comme [Bass *et al.*, 2003] présentent les qualités de l'architecture. Mais, ils ne prennent pas en compte les spécificités de la RFID.

C'est pourquoi nous avons analysé plusieurs applications industrielles ou expérimentales basées sur de la RFID. Par ailleurs, nous avons développé des applications basées sur de la RFID. Ainsi, nous identifions les attributs applicatifs qu'il est important de prendre en compte pour comparer des architectures basées sur de la RFID (cf. section 11.2). Au moyen de ces différents attributs, les sections 11.3 à 11.6 analysent les quatre architectures étudiées aux chapitres 9 et 10 : architecture centralisée, architecture semi-répartie, architecture répartie, et RDSM. Ainsi, en section 11.7, nous sommes en mesure de donner

1. Un patron d'architecture est une description des types d'éléments et de relations (entre ses éléments), accompagnés de l'ensemble des contraintes sur la manière de les utiliser [Bass *et al.*, 2003].

des directives pour choisir la bonne architecture basée sur de la RFID. Dans cette même section, nous présentons l'utilisation de ces directives dans le contexte d'une application réelle. La section 11.8 conclut ce chapitre et présente les perspectives de ces travaux.

11.2 Critères de comparaison

En nous appuyant sur l'expérience acquise en analysant des applications utilisant la RFID et en développant des applications RFID, nous distinguons 3 classes d'attributs parmi les classes présentées dans [Bass *et al.*, 2003] : 1) attributs fonctionnels, 2) attributs d'extensibilité, et 3) attributs de coût. Pour chacune de ces classes, nous détaillons ses différents attributs.

11.2.1 Attributs fonctionnels

La fonctionnalité d'un système est la capacité de ce système à effectuer la tâche qu'il est sensé faire.

Toutes les architectures basées sur de la RFID permettent de lire/écrire les données associées à une étiquette lue.

Le premier attribut fonctionnel a pour but de vérifier le comportement de l'application quand elle lit l'avatar d'une étiquette lue. A-t-on la garantie que l'avatar fourni par l'architecture correspond effectivement aux dernières données écrites ? En d'autres termes, y a-t-il un problème potentiel d'obsolescence de l'avatar d'une étiquette lue ?

Le deuxième attribut fonctionnel concerne la possibilité de connaître la valeur (ou bien d'avoir un ordre d'idée de cette valeur) de l'avatar d'une étiquette géographiquement distante. « Géographiquement distante » signifie que l'utilisateur n'est pas physiquement proche de l'étiquette : il ne peut pas poser son lecteur sur l'étiquette. Tout ce dont il dispose est l'identifiant de l'étiquette distante.

Le troisième attribut fonctionnel est le problème potentiel d'obsolescence de l'avatar d'une étiquette distante. Si l'utilisateur est capable de connaître l'avatar d'une étiquette géographiquement distante, a-t-on la garantie que la valeur de l'avatar renvoyée par l'architecture correspond effectivement aux dernières données écrites ?

11.2.2 Attributs d'extensibilité

Les attributs d'extensibilité évaluent le comportement d'une architecture quand il y a de nombreuses étiquettes ou de nombreux lecteurs.

Le premier attribut est le nombre maximum d'étiquettes qui peuvent être gérées par l'architecture.

Le second attribut évalue la sensibilité de l'architecture au nombre de lectures simultanées d'étiquettes RFID.

11.2.3 Attributs de coût

Les attributs de coûts caractérisent tout ce qui influence les coûts d'installation et les coûts d'exploitation d'une application basée sur de la RFID.

Le premier attribut de coût concerne le besoin d'un réseau informatique global : est-ce que les lecteurs RFID ont besoin d'accéder à une certaine machine (par exemple, un serveur dans le cas de l'architecture centralisée) quel que soit le moment et quelle que soit leur localisation géographique ? Pour répondre à ce besoin, les lecteurs peuvent être équipés d'une connexion filaire à un réseau informatique. Mais leur mobilité est alors limitée. Les lecteurs peuvent également s'appuyer sur des passerelles Bluetooth® ou Wi-Fi. Ces passerelles peuvent induire des coûts d'installation. De plus, certains lecteurs peuvent ne pas disposer de capacités Wi-Fi. Par exemple, le *Nokia 6212* est capable de lire/écrire les étiquettes NFC, mais ne peut pas accéder à un réseau Wi-Fi. Enfin, le lecteur peut utiliser un réseau téléphonique de données (UMTS, HSDPA...). Utiliser un tel réseau induit alors des coûts d'exploitation pour le forfait d'accès à ce réseau.

Le deuxième attribut de coût est lié au besoin de RAM sur chaque étiquette : plus il y a besoin de RAM, plus chère est l'étiquette. Remarquez que certaines étiquettes peuvent ne pas contenir de RAM pour des raisons techniques et non pour des raisons de coût. En effet, l'application peut avoir besoin d'utiliser des étiquettes à basses fréquences, par exemple 125 kHz, de sorte que le lecteur puisse lire ces étiquettes quand elles sont stockées dans un matériau comme de l'eau ou du sang, imperméable à certaines fréquences : le débit est alors trop faible pour que l'étiquette puisse héberger l'avatar en plus de l'identifiant.

Le troisième attribut de coût concerne l'introduction d'une nouvelle étiquette dans le système. Pour chaque architecture, nous déterminons la séquence d'opérations qui est requise pour introduire une nouvelle étiquette dans le système. Ainsi, nous pouvons déterminer le temps d'exécution de cette séquence. Comme cette procédure d'initialisation est exécutée par un être humain ou bien un robot, le coût de cette procédure est proportionnel au temps passé.

Le dernier attribut de coût est lié à la réinitialisation de toutes les étiquettes. Cet attribut ne concerne que les applications qui, durant leur exécution, ont plus ou moins régulièrement besoin que chacune des étiquettes du système reçoive une nouvelle valeur. C'est le cas, par exemple, du système *Navigo* qui est un mode de validation de billet pour les Franciliens empruntant leurs transports en commun. Ces usagers sont munis d'un passe qui contient une étiquette NFC. Au début de chaque mois (ou de chaque nouvelle année), l'usager doit recharger son passe (pour rafraîchir ses droits d'accès) : dit autrement, son passe a besoin d'être réinitialisé. Certains jeux basés sur de la RFID ont également besoin d'une telle réinitialisation. Dans le cas de jeux non permanents, les joueurs jouent pendant des sessions de jeu. De ce fait, au début de chaque session de jeux, les étiquettes peuvent avoir besoin d'être réinitialisées.

Dans cette section, nous avons présenté différents attributs applicatifs qui vont nous permettre d'analyser des architecture à base de RFID. Ces attributs ont été regroupés en trois classes : 1) attributs fonctionnels, 2) attributs d'extensibilité, 3) attributs de coûts. La section suivante analyse l'architecture centralisée grâce à ces attributs.

11.3 Évaluation de l'architecture centralisée

La section 9.2 présente l'architecture centralisée. La section 9.2.1 en donne des exemples d'utilisation.

Analysons les attributs fonctionnels. Quand un lecteur souhaite modifier l'avatar d'une étiquette, il envoie un message de modification au serveur qui héberge l'avatar dans une base de données. Par conséquent, ce serveur est toujours au courant des dernières mises à jour qui ont été faites sur les différents avatars qu'il héberge. Comme un lecteur interroge systématiquement ce serveur pour connaître l'avatar d'une étiquette, il est impossible que la valeur renvoyée par ce serveur soit obsolète. De plus, connaissant un identifiant d'étiquette, un lecteur peut retrouver le serveur qui héberge l'avatar de cette étiquette. Il peut alors interroger ce serveur et ainsi récupérer la valeur de l'avatar de cette étiquette (qui peut être éventuellement géographiquement distante). Comme cette valeur vient du serveur qui détient l'avatar, elle ne peut pas être obsolète.

Intéressons-nous maintenant aux attributs d'extensibilité. Le nombre maximum d'étiquettes qui peut être géré par cette architecture est limité par le nombre d'avatars qui peuvent être sauvegardés en base de données. Soit s la taille moyenne en octets d'un avatar et $S_{central}$ la taille maximum en octets de la base de données. Nous négligeons l'espace requis pour stocker le lien entre les identifiants d'étiquette et leur avatar. Alors, le nombre maximum d'étiquettes est borné par $S_{central}/s$. En ce qui concerne le nombre de lectures simultanées, l'architecture centralisée est contrainte par sa nature centralisée. Le serveur qui fournit la fonction *ONS lookup service* peut devenir un nœud de congestion. De plus, les différents serveurs d'avatars peuvent ne pas renvoyer les valeurs d'avatars suffisamment rapidement. Certes, pour pallier ces deux problèmes, il est possible d'augmenter le nombre de serveurs. Mais cela rend l'architecture plus complexe et plus chère (en coût d'installation et d'exploitation). Cette solution peut ne pas être applicable.

Étudions maintenant les attributs de coût. Le lecteur doit toujours être en contact avec le serveur *ONS lookup service* et les serveurs d'avatars : un réseau informatique global est requis. En revanche, comme cette architecture a seulement besoin de lire un identifiant sur chaque étiquette, cet identifiant peut être stocké en ROM une bonne fois pour toutes : aucune RAM n'est requise sur l'étiquette. Quand une nouvelle étiquette est introduite dans le système, trois opérations sont requises : 1) l'étiquette est physiquement attachée à l'entité physique ; 2) l'avatar est créé et initialisé dans la base de données ; 3) dans la base de données, un lien entre l'identifiant de l'étiquette et son avatar est créé. Quand toutes les étiquettes ont besoin d'être réinitialisées, un programme est exécuté sur le(s) serveur(s) qui héberge(nt) la base de données. Il fixe la nouvelle valeur de chaque avatar.

Cette section a analysé l'architecture centralisée selon les attributs présentés en section 11.2. Elle répond au mieux à tous les besoins fonctionnels. Mais, elle requiert un réseau global. Un autre désavantage est sa sensibilité au nombre de lectures simultanées.

La section suivante analyse l'architecture semi-répartie qui pallie le besoin de réseau global et réduit la sensibilité au nombre de lectures simultanées.

11.4 Évaluation de l'architecture semi-répartie

Dans cette architecture, chaque terminal mobile contient une base de données contenant toutes les données associées aux identifiants des étiquettes RFID utilisées. Chaque base de données est synchronisée périodiquement avec une base de données centrale. Lorsqu'un utilisateur lit l'identifiant d'une étiquette RFID avec son terminal mobile, ce dernier lit et/ou met à jour les données associées qu'il trouve dans sa base de données locale. Les éventuelles mises à jour sont remontées à la base de données centrale lors de la prochaine synchonisation de ce terminal. De plus, elles sont répercutées sur les autres terminaux mobiles lors de la prochaine synchronisation de ces derniers avec la base de données centrale.

La section 9.3 présente l'architecture semi-répartie et son unique exemple d'utilisation.

Analysons les attributs fonctionnels. L'avatar d'une étiquette peut être obsolète. Supposons que les utilisateurs u_1 et u_2 synchronisent leur lecteur avec la base de données centrale. Puis, u_1 modifie l'avatar d'une étiquette r : u_1 modifie sa copie locale de la base de données. Quand u_2 arrive au niveau de l'étiquette r, u_2 consulte sa copie locale de la base de données. Donc, u_2 récupère la valeur de l'avatar de r avant la mise à jour faite par u_1 : la valeur est obsolète. Remarquez qu'il est possible de limiter le risque d'observer ce type d'obsolescence en assignant un ensemble d'étiquettes, donc d'entités, à chaque lecteur. Par exemple, dans le cas de l'application de gestion des arbres de la ville de Paris présentée en section 9.3.1, un responsable peut assigner à chaque jardinier la liste des arbres dont il doit s'occuper pendant la journée. Si chaque liste est disjointe de celle des autres jardiniers, aucune obsolescence de l'avatar ne peut être observée.

En lisant sa base de données locale, chaque lecteur est en mesure de récupérer l'avatar d'une étiquette distante. Mais, la valeur lue peut être obsolète. Elle ne redeviendra correcte qu'une fois que les différentes bases de données locales auront été resynchronisées avec la base de données centrale.

Intéressons-nous maintenant aux attributs d'extensibilité. Le nombre maximum d'étiquettes que peut gérer cette architecture est limité par le nombre d'avatars que peut stocker la base de données centrale et chaque base de données locale. Soit S_{local} le minimum de toutes les capacités des bases de données locales. Le nombre maximum d'étiquettes est borné par $\min(S_{central}/s, S_{local}/s)$, le minimum étant probablement S_{local}/s comme les terminaux mobiles n'ont en général pas autant de mémoire que les serveurs. Remarquez que cette borne peut être remontée à $S_{central}/s$ en ne transférant dans chaque base de données locale qu'un sous-ensemble de la base de données globale. Par exemple, dans le cas de l'application de gestion des arbres de la ville de Paris, le terminal mobile d'un jardinier pourrait ne recevoir que les avatars des arbres dont le jardinier s'occupera pendant la journée.

En ce qui concerne la sensibilité au nombre de lectures simultanées, cette architecture est moins sensible à ce nombre que l'architecture centralisée. En effet, l'architecture semi-répartie n'a pas besoin d'interroger un serveur lors d'une lecture d'étiquette RFID. Toutefois, tous les lecteurs doivent périodiquement se synchroniser avec la base de données centrale. Comme le temps de synchronisation est proportionnel au nombre de lecteurs, ce temps peut atteindre des propositions inacceptables pour les utilisateurs. Ce problème peut être limité en réduisant le nombre d'avatars transférés sur chaque terminal mobile, ce qui réduit le volume de données qui a besoin d'être transféré ultérieurement entre le lecteur et

la base de données centrale.

Étudions maintenant les attributs de coût. Les lecteurs RFID ont seulement besoin d'un réseau global lorsqu'ils se synchronisent avec la base de données globale. Aux autres moments, ils n'ont besoin d'aucun réseau global. Par exemple, pour l'application de gestion des arbres de la ville de Paris, seul un réseau Wi-Fi est requis dans les locaux techniques des jardiniers en vue de synchroniser leurs terminaux avec la base de données centrale. En revanche, aucun forfait mobile n'est requis.

De plus, comme dans l'architecture centralisée, il n'y a aucun besoin de RAM sur les étiquettes. Quand une nouvelle étiquette est introduite dans le système, la procédure à appliquer est identique à la procédure centralisée. Quand les étiquettes ont besoin d'être réinitialisées, un programme est exécuté sur le serveur hébergeant la base de données centrale. Il fixe la nouvelle valeur de chaque avatar. Toutefois, la réinitialisation des étiquettes ne devient effective que quand tous les terminaux mobiles se sont synchronisés avec la base de données centrale.

Cette section a analysé l'architecture semi-répartie. Cette architecture ne requiert aucun réseau globale. Sa sensibilité au nombre de lectures simultanées est moyenne. Toutefois, elle est confrontée à l'obsolescence de la valeur lue pour l'avatar d'une étiquette, qu'elle soit locale ou distante.

La section suivante analyse l'architecture répartie qui pallie la sensibilité au nombre de lecture simultanées et le problème de l'obsolescence de l'avatar d'une étiquette locale.

11.5 Évaluation de l'architecture répartie

Le principe de cette architecture est de stocker toutes les données associées à une étiquette RFID au sein même de cette étiquette. La section 9.4.1 présente des exemples d'utilisation.

Analysons les attributs fonctionnels. L'avatar d'une étiquette est écrit et lu dans la RAM de l'étiquette : il n'y a aucun problème d'obsolescence. En revanche, il est impossible de connaître la valeur de l'avatar d'une étiquette distante.

Intéressons-nous maintenant aux attributs d'extensibilité. Le nombre d'étiquettes déployable dans l'environnement est illimité. De plus, vu que toutes les lectures/écritures d'étiquettes se font au niveau du lecteur et de l'étiquette concernée, cette architecture n'a aucune sensibilité au nombre de lectures simultanées.

Étudions maintenant les attributs de coût. Le lecteur n'a aucun besoin d'accéder à un réseau informatique global. En revanche, de la RAM est requise sur chaque étiquette. Sa taille doit être au minimum celle de l'avatar. Un effet de bord est que l'avatar ne doit pas être trop grand. En effet, des étiquettes NFC comme les Mifare offrent au maximum 4 Kio de RAM avec seulement 3 440 octets réellement disponibles pour l'application.

Quand une nouvelle étiquette est introduite dans le système, seules deux opérations sont requises : 1) l'étiquette est physiquement attachée à l'entité physique ; 2) l'avatar est créé et initialisé dans l'étiquette.

En ce qui concerne la réinitialisation des étiquettes, elle est dépendante de l'application. Certaines applications requièrent qu'un utilisateur dédié passe par chaque étiquette

pour la réinitialiser. Le passe *Navigo* adopte une autre démarche : chaque utilisateur a la responsabilité d'amener son passe, donc son étiquette, au niveau d'une machine chargée de la réinitialisation. En début de mois, cela crée des temps d'attente importants, donc des coûts induits pour les utilisateurs, à cause des files d'attente qui se créent au niveau de ces machines. Pour éviter les coûts de réinitialisation, certaines applications mettent en place des mécanismes spéciaux. Ces mécanismes s'appuient sur le temps écoulé pour réinitialiser automatiquement les données. Dans l'application académique *Roboswarm*, des robots sont programmés pour pousser des lits d'hôpitaux de sorte que le nettoyage de la chambre puisse se faire [Zecca *et al.*, 2008]. Des opérateurs humains ou robots (dénommés *path finders*) collent des étiquettes RFID sur le sol pour indiquer aux robots pousseurs (qui interviendront par la suite) les endroits où se placer pour pousser les lits. Quand un robot pousseur trouve une telle étiquette, il consulte la date du dernier nettoyage stockée dans l'étiquette. Il la compare à la date actuelle moins un *niveau de détérioration*. Si la date est trop ancienne, il pousse le lit et met à jour la date. Quant à l'application destinée à retrouver des objets perdus à l'aide de phéromones numériques [Mamei et Zambonelli, 2007], elle évite une réinitialisation périodique des étiquettes, même si leur RAM devient saturée de phéromones numériques : chaque phéromone est inscrit dans l'étiquette avec une estampille temporelle. Ainsi, si un lecteur ne trouve pas de place sur une étiquette RFID, il écrit le phéromone qu'il souhaite écrire en lieu et place du phéromone le plus âgé.

Cette section a analysé l'architecture répartie. Cette architecture n'a pas besoin de réseau informatique global. Et, elle est totalement insensible au nombre de lectures simultanées d'étiquettes. Toutefois, elle est incapable de lire la valeur de l'avatar d'une étiquette distante.

La section suivante analyse la RDSM qui pallie cet inconvénient.

11.6 Évaluation de la mémoire répartie partagée

La RDSM combine les avantages de l'architecture semi-répartie et de l'architecture répartie (cf. chapitre 10). La section 10.3 en présente l'unique exemple d'utilisation.

Analysons ses attributs fonctionnels. Quand un terminal mobile arrive au niveau d'une étiquette, deux cas peuvent se présenter. Soit l'étiquette a déjà été initialisée : l'avatar est stocké dans l'étiquette et la valeur lue dans la RAM de l'étiquette est la valeur la plus à jour. Soit l'étiquette n'a pas été initialisée : la première tâche du terminal mobile est d'initialiser l'étiquette avec la valeur de son avatar. La valeur désormais présente dans la RAM de l'étiquette est la valeur la plus à jour. Dans les deux cas, il n'y aucun risque d'obsolescence. Par ailleurs, chaque terminal mobile détient une copie de tous les homologues numériques du système. En interrogeant cette copie locale, chaque terminal peut donc retrouver la valeur de l'avatar d'une étiquette distante. Toutefois, cette copie locale peut ne pas être à jour : il y a risque d'obsolescence dans la valeur de l'avatar d'une étiquette distante.

Intéressons-nous maintenant aux attributs d'extensibilité. Le nombre maximum d'étiquettes est limité par la taille de la RAM de chaque étiquette. En effet, cette architecture requiert une copie de tous les avatars et un vecteur d'horloges dans chaque étiquette. Soit S_{tag}, le minimum des capacités en RAM (en octets) des étiquettes du système. Soit L la taille en octets d'un élément du vecteur d'horloges et s la taille de l'avatar. Alors, le

nombre maximum d'étiquettes est borné par $S_{tag}/(s+L)$. Donnons un ordre de grandeur de cette limite. Dans le cas de l'architecture semi-répartie, la taille du numérateur est exprimée en Gio. Dans la RDSM, la taille du numérateur est exprimée en Kio. RDSM gère au minimum un million de fois moins d'étiquettes que l'architecture semi-répartie.

En ce qui concerne la sensibilité au nombre de lectures simultanées, chaque lecteur terminal mobile passe son temps à se synchroniser avec l'étiquette qu'il lit ou le terminal mobile qu'il rencontre : RDSM est aussi insensible au nombre de lectures simultanées que l'architecture répartie.

Étudions maintenant les attributs de coût. Le lecteur n'a aucun besoin d'accéder à un réseau informatique global. En revanche, de la RAM est requise sur chaque étiquette. Sa taille doit être au minimum la taille d'un avatar plus L, le tout multiplié par le nombre d'étiquettes dans le système. Cela signifie que RDSM gère des avatars moins grands que dans le cas de l'architecture répartie.

Quand une nouvelle étiquette est introduite dans le système : 1) l'étiquette est physiquement attachée à l'entité physique ; 2) l'avatar est créé et initialisé dans DM_{init}, la valeur utilisée pour (ré)initialiser $DM_{m,m\in\{mobiles\}}$ et hébergée sur une machine spécifique (ce peut être l'un des terminaux mobiles) ; 3) un lien entre l'identifiant de l'étiquette et l'avatar est créé dans la base de données d'initialisation ; 4) tous les éléments du vecteur d'horloges stocké dans l'étiquette sont initialisés à 0.

Quand il y a besoin de faire une nouvelle initialisation des étiquettes, un programme est exécuté sur la machine spécifique hébergeant DM_{init}. Il calcule VC_{init} de sorte que chacun de ses éléments soit plus grand, donc plus « récent », que tous les éléments de vecteurs d'horloges des terminaux mobiles. Deux méthodes sont disponibles. Dans la première méthode, la machine spécifique récupère tous les vecteurs d'horloges des terminaux mobiles et calcule le maximum : il n'y a aucun besoin de place mémoire additionnelle sur les étiquettes. En revanche, il y a besoin de communications supplémentaires entre les terminaux mobiles et cette machine spécifique. Remarquez que cette méthode ne fonctionne que parce que le vecteur d'horloges d'une étiquette n'évolue que si l'étiquette est en contact avec un mobile. De ce fait, il y a toujours au moins un mobile qui est au courant du vecteur d'horloges stocké dans une étiquette donnée. La seconde méthode suppose que chaque élément du vecteur d'horloges est constitué de deux champs : le champ `identifiantDeSession` et le champ `versionDansSession`. Ainsi la machine spécifique n'a besoin que d'incrémenter le champ `identifiantDeSession` et de remettre à zéro tous les champs `versionDansSession`. Cette méthode requiert de la mémoire additionnelle sur chaque étiquette, mais ne nécessite aucune communication supplémentaire entre la machine spécifique et les terminaux mobiles. Le choix de la méthode dépend de l'application. Une fois que l'une de ces deux méthodes a été appliquée, la machine spécifique initialise chaque terminal mobile avec DM_{init} et VC_{init}. Après coup, quand un terminal mobile m est au contact d'une étiquette RFID r qui n'a pas encore été initialisée pour cette session, comme chaque élément de VC_m est plus grand que son *alter ego* dans VC_r, DM_r (respectivement VC_r) est initialisé avec DM_m (respectivement VC_m) : la RDSM tire parti du fait que les utilisateurs se déplacent jusqu'aux étiquettes, pour initialiser les dites étiquettes.

Cette section a analysé la RDSM. Cette architecture n'a pas besoin de réseau global. Elle n'a pas de problème d'obsolescence de la valeur de l'avatar d'une étiquette locale. De

TABLE 11.1 – Comparaison des architectures à base de RFID (les valeurs en italique correspondent aux attributs qui représentent une limite de cette architecture)

	Centr.	Semi.	Dist.	RDSM
Obsolescence d'une étiquette lue localement	Non	*Oui*	Non	Non
avatar d'une étiquette distante ?	Oui	Oui	*Non*	Oui
Obsolescence d'une étiquette géographiquement distante	Non	*Oui*	*n.a.*	*Oui*
Nombre maximum d'étiquettes	$S_{central}/s$	S_{local}/s	∞	$S_{tag}/(s+L)$
Sensibilité au nombre de lecture simultanées	*Forte*	Moyenne	Aucune	Moyenne
Réseau informatique global	*Oui*	Non	Non	Non
RAM requise sur les étiquettes	Non	Non	*Oui*	*Oui*
Coût d'introduction étiquette (opération la plus coûteuse)	Lier étiq. à entité physique	Lier étiq. à entité physique	Lier étiq. à entité physique	Lier étiq. à entité physique
Coût réinitialisation étiquettes (opération la plus coûteuse)	Réinit. BD	Synchro. BD	*Aller à chaque entité physique*	Synchro. BD

plus, elle peut fournir une valeur pour l'avatar d'une étiquette distante. Toutefois, cette valeur peut être obsolète. En outre, cette architecture est limitée en termes de nombre maximum d'étiquettes qu'elle peut gérer.

Jusqu'à présent, nous avons analysé les différentes architectures à base de RFID. En synthétisant les conclusions obtenues, la section suivante fournit des directives pour choisir l'architecture la plus adaptée à une application donnée.

11.7 Directives pour le choix d'une architecture

La table 11.1 synthétise l'analyse des attributs des différentes architectures à base de RFID.

Si l'application requiert le mieux pour tous les attributs fonctionnels, l'architecture centralisée doit être retenue. En effet, c'est la seule architecture qui n'a aucun problème avec les différentes attributs fonctionnels. Mais, cette architecture nécessite un réseau global et ses coûts associés. De plus, elle est sensible au nombre de lectures simultanées d'étiquettes.

Si l'un de ces deux problèmes est bloquant, l'architecte système doit étudier les trois autres architectures à base de RFID. L'architecture semi-répartie doit être choisie si les étiquettes RFID ne peuvent pas contenir de RAM. Cette contrainte peut être motivée par les coûts, mais aussi par des considérations techniques.

S'il peut y avoir de la RAM sur les étiquettes, le nombre maximum d'étiquettes envisagées dans le système doit être estimé. S'il est compatible avec la RDSM, c'est cette dernière architecture qui doit être retenue. En effet, c'est la RDSM qui est la moins limitée en termes de fonctionnalités. S'il y a incompatibilité, l'architecte système doit se tourner vers l'architecture répartie (s'il faut éviter le problème d'obsolescence de la valeur de l'avatar d'une étiquette locale) ou l'architecture semi-répartie (si le coût de réinitialisation des étiquettes est un facteur important). Remarquez que le mélange de l'architecture répartie et de la RDSM peut être une alternative intéressante : l'idée est de stocker sur chaque étiquette son avatar et l'élément de vecteur d'horloges correspondant à cet avatar. Chaque terminal mobile détient une copie de tous les avatars et un vecteur d'horloges complet. En appliquant les algorithmes présentés au chapitre 10, nous obtenons une solution pour dépasser la limite du nombre maximum d'étiquettes RFID dans le système ; en même temps, nous résolvons le problème de l'architecture répartie : un utilisateur peut demander à son mobile de lui fournir la valeur de l'avatar d'une étiquette distante.

Pour illustrer ces directives, étudions le choix de l'architecture la plus adaptée au jeu PSM présenté en section 10.3. Chaque étiquette coûte 0,10 euro (respectivement 1,50 euro) si elle offre 0 Kio (respectivement 1 Kio avec 752 octets réellement disponibles, $S_{tag} = 752$ octets) de RAM. L'avatar d'une étiquette est le carte virtuelle « contenue » dans l'étiquette. Il y a au maximum 16 cartes dans le jeu. Donc, l'avatar est codé sous la forme d'un nombre compris entre 0 et 15 : $s = 1$ octet. Pour le vecteur d'horloges, le projet utilise la méthode d'initialisation avec un champ `identifiantDeSession` et un champ `instantDansSession`. `identifiantDeSession` stocke l'heure d'initialisation sous la forme du nombre de millisecondes écoulées depuis le premier janvier 1970 : 8 octets sont nécessaires. `versionDansSession` est un `short`, $L = 2$ octets, qui représente l'estampille temporelle sous forme de nombre de secondes depuis l'heure stockée dans `identifiantDeSession`.

Il faut environ 20 minutes pour attacher chacune des étiquettes à sa localisation correcte, soit une moyenne de 75 secondes par étiquettes. Lier l'étiquette à son avatar requiert environ 5 secondes par étiquettes. Le programme d'initialisation a besoin de quelques millisecondes pour initialiser un avatar. Pour réinitialiser les étiquettes, synchroniser les 8 mobiles avec une machine spécifique prend environ une minute, soit une moyenne de 4 secondes par étiquette.

Si le jeu opte pour l'architecture centralisée ou semi-répartie, il utilisera une machine dédiée pour héberger la base de données centrale. Cette machine sera équipée d'un disque de 500 Go : $S_{central} = 500$ Go. Dans le cas de l'architecture semi-répartie, le jeu utilisera la moitié de la capacité de la carte micro-SD du téléphone : $S_{local} = 1$ Gio.

Si le jeu opte pour l'architecture centralisée, chaque mobile aura une carte SIM lui donnant accès à un forfait données 3G. Chaque forfait coûtera 15 euros par mois.

Si le jeu utilise l'architecture répartie, il faudra 13 minutes pour aller au niveau de chaque étiquette pour la réinitialiser, soit une moyenne de 49 secondes par étiquette.

Nous appliquons ces valeurs numériques à la table 11.1. La table 11.2 synthétise les résultats.

L'architecture centralisée requiert un réseau qui coûte 120 euros mensuels. Le musée qui accueille ce jeu considère que ces frais sont trop élevés. Nous devons donc envisager l'une des trois autres architectures. Comme le jeu gère 16 étiquettes et que la RDSM

TABLE 11.2 – Comparaison des architectures à base de RFID dans le cas du jeu PSM (les valeurs en italique correspondent aux attributs qui représentent une limite de cette architecture)

	Centr.	Semi.	Dist.	RDSM
Nombre maximum d'étiquettes si $s = 1$ *byte* (si $s = 250$ *bytes*)	500×10^9 (2×10^9)	10^9 (4×10^6)	∞ (∞)	*248* (*2*)
Coût réseau informatique (par mois)	*120 euros*	0 euro	0 euro	0 euro
Coût étiquette (par étiquette)	0,10 euro	0,10 euro	*1,50 euro*	*1,50 euro*
Coût d'introduction nouvelle étiq. (en sec./étiq.)	80 s	80 s	80 s	80 s
Coût réinit. une étiq. (en sec./étiq.)	0 s	4 s	*49 s*	4 s

gère au maximum 248 étiquettes dans le cas où $s = 1$ octet, nous pouvons opter pour la RDSM. La conclusion aurait été différente si s avait valu 250 octets. En effet, dans ce cas, la RDSM n'aurait pu gérer que 2 étiquettes : elle n'aurait pas convenu. Il ne doit pas y avoir de problème d'obsolescence de la valeur de l'avatar d'une étiquette locale, sinon le jeu ne serait pas amusant. Nous aurions donc dû opter pour l'architecture répartie ou bien une combinaison d'architecture répartie et semi-répartie, pour réduire le coût de réinitialisation des étiquettes.

11.8 Conclusion

Ce chapitre analyse les 4 architectures à base de RFID : architecture centralisée, architecture semi-répartie, architecture répartie, et RDSM. Il les compare selon 9 attributs répartis en 3 catégories : les attributs fonctionnels, les attributs d'extensibilité, et les attributs de coût.

Ce chapitre propose des directives pour choisir l'architecture à base de RFID la plus adaptée à une application donnée. Ces directives sont testées dans le cas du jeu PSM.

La principale perspective de ces travaux concerne l'intégration de l'attribut extrafonctionnel sécurité. La sécurité est la mesure de la capacité d'un système à résister à un usage non autorisé tout en fournissant le service attendu aux utilisateurs légitimes [Bass *et al.*, 2003]. Or, la mise en place d'une politique de sécurité pourrait changer les directives présentées à la section 11.7. Il en va de même pour l'intégration de l'attribut extrafonctionnel tolérance aux fautes d'un des éléments du système.

Ce chapitre clôt la présentation détaillée de nos travaux autour de la RDSM. Le prochain chapitre fait la synthèse de ces travaux et les met en perspective.

Chapitre 12

Conclusions et perspectives

Ce chapitre conclut cette seconde partie portant sur le partage de données dans un système réparti. Il synthétise nos contributions qui ont permis d'aboutir à la mémoire répartie partagée à base d'étiquettes RFID. Il propose ensuite des perspectives de nos travaux.

Dans un premier temps, nous avons présenté cette mémoire, baptisée RDSM. Grâce à l'utilisation de vecteurs d'horloges, la RDSM garantit que chaque élément du système a la vision la plus à jour qu'il puisse avoir, compte tenu des interactions qu'il a eues avec les autres éléments du système. L'étude des limites en termes d'extensibilité a montré que la RDSM n'est pas applicable au delà de quelques centaines d'éléments. Cette extensibilité est acceptable pour des applications comme PSM, le jeu qui nous a permis d'expérimenter la RDSM dans des conditions réelles.

Dans un second temps, nous avons analysé les 4 architectures à base de RFID : architecture centralisée, architecture semi-répartie, architecture répartie, et RDSM. Nous les avons comparées selon 9 attributs répartis en 3 catégories : les attributs fonctionnels, les attributs d'extensibilité, et les attributs de coût. Nous avons proposé des directives pour choisir l'architecture à base de RFID la plus adaptée à une application donnée. Il en résulte que l'architecture centralisée est celle qui répond au mieux aux différents critères. Mais elle requiert un réseau global, ce qui peut générer des coûts inacceptables pour une application. Si le nombre d'étiquettes est de l'ordre de quelques centaines, la meilleure alternative selon nos critères est alors la RDSM. Sinon, il faut se tourner vers l'architecture répartie (s'il ne faut pas de problème d'obsolescence de la valeur de l'avatar d'une étiquette locale) ou l'architecture semi-répartie (si le coût de réinitialisation des étiquettes est un facteur important).

La RDSM a été implantée dans le jeu pervasif « PLUG : les Secrets du Musée », déployé au *Musée des arts et métiers* (Paris) [daem09b, Sim09]. Au cours de diverses manifestations comme la *Fête de la Science* 2008 ou Futur en Seine 2009, il a été utilisé par environ un millier de joueurs.

Les perspectives de ces travaux sont doubles.

De nouveaux critères de comparaison d'architectures à base de RFID devraient être intégrés à notre étude. C'est le cas notamment de l'attribut extrafonctionnel sécurité (res-

pectivement tolérance aux fautes d'un élément du système). En effet, la mise en place d'une politique de sécurité (respectivement tolérance aux fautes) pourrait changer les directives présentées à la section 11.7.

Une autre perspective est d'intégrer la RDSM dans un intergiciel de gestion de données. En effet, nous pensons que cette intégration faciliterait l'utilisation de la RDSM par un développeur d'applications. Pour réaliser cette intégration, l'utilisation de *LIME* ou sa déclinaison *TinyLIME* pour réseau de capteurs passifs nous semble une piste prometteuse [Mur07]. En effet, ces intergiciels offrent la notion d'espace de n-uplets partagés [Murphy *et al.*, 2006] aux fonctionnalités génériques (insertion de n-uplets, recherche, etc.), tout en étant adaptés aux capacités réduites des étiquettes RFID. De plus, des expérimentations de réplication de n-uplets au sein de ces intergiciels ont déjà été menées [Murphy et Picco, 2006]. L'intégration de la RDSM à *LIME* ou *TinyLIME* devrait donc représenter un investissement raisonnable et éviterait aux développeurs une implantation *ad hoc* de la RDSM.

Cette perspective conclut ce chapitre et cette seconde partie. Le prochain chapitre conclut ce manuscrit de thèse.

Conclusion et perspectives

Chapitre 13

Conclusion et perspectives

13.1 Synthèse

Cette section met en avant les apports de ce manuscrit. Elle résume les différents points abordés dans cette thèse.

Tout au long de cette thèse, nous avons contribué au domaine des systèmes répartis avec une perspective d'implantation et d'application. Dans un premier temps, nous nous concentrons sur la communication de données dans un système, en vue de le rendre tolérant aux fautes. Par conséquent, nous développons le protocole des trains. Ce protocole d'uot-diffusion fait circuler, en parallèle, des trains d'uot-diffusions entre les participants répartis sur un anneau virtuel, le circuit des trains. L'utilisation d'une horloge logique au sein de chaque train nous permet d'intégrer la gestion des participants au protocole d'uot-diffusion proprement dite. Les temps de récupération de défaillances de processus s'en trouvent réduits. Nous prouvons la correction de ce protocole. Par ailleurs, nous proposons un algorithme pour gérer le dépassement de capacité de l'horloge logique. Nous nous intéressons également aux performances théoriques de ce protocole. Il agrège naturellement les messages dans un même train. De ce fait, les métriques présentées dans la littérature ne permettent pas d'évaluer correctement l'efficacité de notre protocole. Aussi, nous proposons une nouvelle métrique : le rendement en termes de débit, c'est-à-dire le rapport entre le nombre d'octets uot-livrés par période et le nombre maximum d'octets qui peuvent être diffusés à tous les processus par période. Cette métrique nous permet de montrer que le protocole des trains a un rendement théorique supérieur au meilleur, en termes de débit, des protocoles présentés dans la littérature (LCR [Guerraoui *et al.*, 2010]). De plus, le rendement en termes de débit du protocole des trains nous permet de calculer le débit maximum théorique que peut atteindre notre protocole. Nous proposons une implantation de notre protocole. Nos premiers tests montrent que l'écart entre le débit maximum théorique et le débit expérimental est de 3%. D'une part, ce pourcentage confirme le bien-fondé de notre théorie. D'autre part, il nous indique que notre code ne contient pas de défaut majeur entraînant une anomalie de performances. Enfin, nous présentons comment réaliser avec notre protocole une mémoire répartie partagée, utilisée par une application constituée d'objets Objective-C. Ainsi, nous rendons cette application tolérante aux fautes.

Dans un second temps, nous nous concentrons sur le partage de données dans un sys-

tème utilisant des étiquettes RFID et n'ayant pas de réseau informatique global. Or, nous souhaitons connaître, à distance, le contenu des différentes étiquettes. Notre état de l'art montre qu'aucune des architectures existantes à base d'étiquettes RFID (centralisée, semi-répartie, et répartie) ne répond à notre besoin. De ce fait, nous proposons une mémoire répartie partagée à base de RFID (RDSM). S'appuyant sur des vecteurs d'horloges, cette mémoire s'inscrit dans la lignée des mémoire répartie partagée à cohérence causale [Ahamad *et al.*, 1991], mais avec trois spécificités : 1) nous n'invalidons pas les données périmées, ce qui est acceptable pour les applications que nous visons ; 2) notre mémoire s'affranchit d'un réseau informatique global, en exploitant le réseau formé par les personnes qui utilisent cette mémoire via l'application stockée dans leur lecteur RFID ; 3) ce réseau humain nécessite d'être stimulé, ce qui requiert des fonctionnalités idoines au niveau de l'application Nous expérimentons cette architecture au sein d'un jeu ubiquitaire. Puis, nous étudions théoriquement les limites en termes d'extensibilité de notre RDSM : elle n'est pas applicable au delà de quelques centaines d'éléments, ce qui est acceptable pour des applications comme notre jeu. Nous comparons ensuite la RDSM aux trois autres architectures à base de RFID, selon 9 attributs répartis en 3 catégories : les attributs fonctionnels, les attributs d'extensibilité, et les attributs de coût. Nous sommes ainsi en mesure de proposer des directives pour choisir l'architecture à base de RFID la plus adaptée à une application donnée.

Intéressons-nous maintenant aux perspectives de nos travaux de thèse.

13.2 Perspectives

Dans cette ultime section, nous reprenons et complétons les perspectives introduites dans les conclusions intermédiaires, dédiées à chaque partie.

Pespectives liées à la communication de données : par rapport à la facette « uot-diffusion » de notre contribution, nous voulons approfondir nos premiers tests de performance et, plus particulièrement, déterminer les valeurs optimales des paramètres du protocole (par exemple, la taille optimale de ses wagons). Par ailleurs, nous souhaitons approfondir les preuves que nous avons proposées. Ainsi, nous envisageons de développer plusieurs tests de vérification. De plus, nous considérons le codage de notre protocole en *TLA+* et/ou en *B* en vue de preuves automatisées.

Vu les débits remarquables qu'offre le protocole des trains notamment pour les messages de petite taille, nous souhaitons étudier les possibilités de liens avec les intergiciels pour la réplication *JGroups* [Ban02], *Spread* [Spr98] et *Hazelcast* [Haz10]. Les briques logicielles utilisant de la réplication comme, par exemple, *Hadoop Distributed File System* [Apa07], ou bien certaines bases de données répliquées méritent également d'être considérées.

En ce qui concerne notre métrique rendement en termes de débit, la différence de 3% entre les performances attendues grâce à cette métrique et les performances observées avec nos premiers tests nous montre la précision de cet outil. Toutefois, nous sommes convaincus que cette métrique a besoin d'être affinée. En effet, tout d'abord, nous souhaitons comprendre d'où proviennent ces 3% de différence, afin de réduire les erreurs de cette métrique. De plus, nous avons montré dans ce manuscrit que notre métrique ne tenait

pas compte de toutes les limites du système considéré : quelles autres données prendre en compte dans cette métrique pour la rendre encore plus précise ? L'analyse d'autres protocoles d'uot-diffusion semble une piste de recherche intéressante par rapport à cette question. Des échanges avec des chercheurs du domaine des réseaux et peut-être de l'évaluation des performances pourraient également se révéler fructueux.

Un domaine d'expérimentation de tous ces travaux est l'industrie du jeu vidéo sur mobile. Nous contribuons à des intergiciels de communication pour le jeu sur mobile avec *uGASP* [PAD+07, Pellerin *et al.*, 2005a] et le *Communication Middleware TOTEM* (qui s'appuie sur l'intergiciel *RabbitMQ*) [TOT09, Conan *et al.*, 2011]. Nous envisageons d'y intégrer l'uot-diffusion (éventuellement enfouie au sein d'une mémoire répartie partagée) pour faciliter la programmation de jeux multijoueurs grâce à des objets répliqués. Cela nécessitera des APIs dédiées, comme nous avons commencé à l'expérimenter avec le concept de jeu *Alterland* basé sur *uGASP* [SA11] ou bien un mini-« jeu » basé sur *JGroups* et *Android* [Cheynet et Sionneau, 2011].

Toutes ces perspectives s'inscrivent dans une optique de contribution à la cohérence de données dans le *cloud*. En effet, depuis une dizaine d'années, les architectes ont développé des solutions pour le *cloud* qui ignoraient l'uot-diffusion. Cette dernière étaient considérée comme inadéquate à cause du théorème CAP [Brewer, 2012]. Selon ce théorème, un système à données partagées par le réseau ne peut offrir que 2 des 3 propriétés souhaitables : 1) la cohérence (C) permettant de considérer que l'application a un seul exemplaire des données, 2) la haute disponibilité (A) des données pour les mises à jour, 3) la tolérance aux partitions réseau (P). Or, cette année, des travaux ont montré que, du fait de l'évolution des technologies et de leurs usages, les limitations inhérentes au théorème CAP pouvaient être surmontées [Brewer, 2012, Birman *et al.*, 2012]. Nous souhaitons contribuer à ce domaine de recherche.

En ce qui concerne la facette « Mémoire répartie partagée à base d'uot-diffusion » de notre contribution, son principe est de diffuser des opérations de mises à jour sur des objets du système et non l'état des objets après application de la mise à jour. La compacité de cette diffusion couplée aux débits importants que peut atteindre le protocole des trains pourrait être une piste de recherche pour la tolérance aux fautes dans les plates-formes post-petascales. De fait, la communauté du calcul à haute-performances s'intéresse à la réplication pour l'amélioration de la résilience de ces applications [Bougeret *et al.*, 2012].

Pespectives liées au partage de données : de nouveaux critères de comparaison d'architectures à base de RFID doivent être intégrés à notre étude actuelle. C'est le cas notamment des attribut extrafonctionnels sécurité et tolérance aux fautes d'un élément du système qui pourrait changer les directives que nous avons présentées.

Une autre perspective est d'intégrer la RDSM dans un intergiciel de gestion de données. En effet, nous pensons que cette intégration faciliterait l'utilisation de la RDSM par un développeur d'applications. Pour réaliser cette intégration, l'utilisation de *LIME* ou sa déclinaison *TinyLIME* pour réseau de capteurs passifs nous semble une piste prometteuse [Mur07].

La RDSM requiert que l'application qui l'utilise mette en place des fonctionnalités qui stimule le réseau humain sur lequel s'appuie la RDSM. À notre connaissance, c'est le

premier exemple d'un intergiciel qui requiert des fonctionnalités au niveau de l'application qui l'utilise. Il nous semble intéressant d'étudier si cet exemple est révélateur d'une tendance de fond concernant le lien entre les intergiciels et les applications.

Pespective générale : tout au long de cette thèse, nous avons cherché à exploiter au mieux les ressources du système, quitte à réduire les fonctionnalités ou la qualité de service offerte à l'application de l'utilisateur. Ainsi, le protocole des trains cherche à optimiser le débit des uot-diffusions au détriment de la latence qui est en général privilégiée. De plus, en l'absence d'un réseau global, la RDSM permet d'interroger l'avatar d'une étiquette RFID distante, avec des risques de péremption des données. En cela, nous considérons que notre démarche est en phase avec une tendance de fond de l'informatique d'aujourd'hui : la maîtrise de l'énergie consommée par les systèmes. Ainsi, dans les supercalculateurs, « quand on parvient à réaliser une addition en dépensant 7 picojoules, il faut en dépenser 50 pour transporter le résultat sur un millimètre » [Vandeginste, 2010]. De plus, l'éco-conception des logiciels devient un enjeu pour les développeurs d'application [Philippot, 2012, GCL10]. Nos travaux sont un premier apport à ce domaine émergent. Nous souhaitons poursuivre dans cette direction.

Bibliographie et Webographie

Bibliographie

[Afek *et al.*, 1993] Afek, Y., Brown, G. et Merritt, M. (1993). Lazy caching. *ACM Trans. Program. Lang. Syst.*, 15:182–205.

[Ahamad *et al.*, 1991] Ahamad, M., Hutto, P. et John, R. (1991). Implementing and programming causal distributed shared memory. *In Distributed Computing Systems, 1991., 11th International Conference on*, pages 274 –281.

[Amir *et al.*, 1995] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A. et Ciarfella, P. (1995). The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13:311–342.

[Armenio *et al.*, 2007] Armenio, F., Barthel, H., Burstein, L., Dietrich, P., Duker, J., Garrett, J., Hogan, B., Ryaboy, O., Sarma, S., Schmidt, J., Suen, K., Traub, K. et Williams, J. (2007). The EPCglobal architecture framework. Rapport technique Version 1.2, GS1 EPCglobal.

[Attiya et Welch, 2004] Attiya, H. et Welch, J. (2004). *Distributed Computing : Fundamentals, Simulations and Advanced Topics*. Wiley–Interscience, second édition.

[Baldoni, 1998] Baldoni, R. (1998). A positive acknowledgment protocol for causal broadcasting. *Computers, IEEE Transactions on*, 47(12):1341 –1350.

[Baldoni et Raynal, 2002] Baldoni, R. et Raynal, M. (2002). Fundamentals of distributed computing : A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2).

[Baradel *et al.*, 1995] Baradel, Ch., Eychenne, Y., Junot, L., Kohen, B. et Simatic, M. (1995). Fault-tolerance and on-line maintainability in a process control supervision system. *Distributed Systems Engineering*, 2(2):65.

[Baradel *et al.*, 1996] Baradel, Ch., Eychenne, Y. et Kohen, B. (1996). Software system having replicated objects and using dynamic messaging, in particular for a monitoring/control installation of redundant architecture. American patent number 5488723.

[Bass *et al.*, 2003] Bass, L., Clements, P. et Kazman, R. (2003). *Software Architecture in Practice, 2nd Edition*. Addison-Wesley Professional. ISBN-13 : 978-0-321-15495-8.

[Bauer *et al.*, 2002] Bauer, D., Rooney, S. et Scotton, P. (2002). Network infrastructure for massively distributed games. *In Network and System Support for Games 2002 (NetGames)*, pages 36–43. Braunschweig, Germany, ACM.

[Bettner et Terrano, 2001] Bettner, P. et Terrano, M. (2001). 1500 Archers on a 28.8 : Network Programming in Age of Empire and Beyond. *In Proceedings of the 2001 Game Developer Conference*. San Jose, California, USA.

[Birman, 2007] Birman, K. (2007). The promise, and limitations, of gossip protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):8–13.

[Birman, 2010] Birman, K. (2010). A history of the virtual synchrony replication model. *In* Charron-Bost, B., Pedone, F. et Schiper, A., éditeurs : *Replication*, volume 5959 de *Lecture Notes in Computer Science*, pages 91–120. Springer Berlin / Heidelberg. 10.1007/978-3-642-11294-2_6.

[Birman, 1993] Birman, K. P. (1993). The process group approach to reliable distributed computing. *Commununications of the ACM*, 36:37–53.

[Birman *et al.*, 2012] Birman, K. P., Freedman, D. A., Huang, Q. et Dowell, P. (2012). Overcoming cap with consistent soft-state replication. *Computer*, 45:50–58.

[Birman et Joseph, 1987b] Birman, K. P. et Joseph, T. A. (1987b). Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5:47–76.

[Blagojevic, 1998] Blagojevic, V. (1998). Implementing Totems' total ordering protocol in JavaGroups reliable group communication toolkit. Rapport technique, Dept. of Computer Science York University. Available at `http://www.jgroups.org/papers/totaltoken.ps.gz`.

[Bougeret *et al.*, 2012] Bougeret, M., Casanova, H., Robert, Y., Vivien, F. et Zaidouni, D. (2012). Using group replication for resilience on exascale systems. Rapport technique 7876, Inria, Research Centre Grenoble, Rhônes-Alpes.

[Brewer, 2012] Brewer, E. (2012). Cap twelve years later : How the "rules" have changed. *Computer*, 45:23–29.

[Cerf *et al.*, 2007] Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K. et Weiss, H. (2007). Delay-Tolerant Networking Architecture (RFC4838). Rapport technique, The IETF Trust.

[Chandra et Toueg, 1996] Chandra, T. D. et Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43:225–267.

[Chandy et Lamport, 1985] Chandy, K. M. et Lamport, L. (1985). Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75.

[Chang et Maxemchuk, 1984] Chang, J.-M. et Maxemchuk, N. F. (1984). Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273.

[Cheynet et Sionneau, 2011] Cheynet, J. et Sionneau, Y. (2011). Intergiciel de communication pair-à-pair utilisant de la synchronie virtuelle. Rapport technique, Télécom SudParis.

[Chockler *et al.*, 2001] Chockler, G. V., Keidar, I. et Vitenberg, R. (2001). Group communication specifications : a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469.

[Conan *et al.*, 2011] Conan, D., Simatic, M. et Adgeg, G. (2011). Presentation of the TOTEM Communication Infrastructure. Rapport technique, Télécom SudParis.

[Couderc et Banâtre, 2009] Couderc, P. et Banâtre, M. (2009). Beyond RFID : The Ubiquitous Near-Field Distributed Memory. *ERCIM news*, (76):35–36.

[Cristian, 1991] Cristian, F. (1991). Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116.

[Cristian *et al.*, 1985] Cristian, F., Aghili, H., Strong, R. et Dolev, D. (1985). Atomic broadcast : From simple message diffusion to byzantine agreement. *In Information and Computation*, pages 200–206.

[Cristian et Mishra, 1993] Cristian, F. et Mishra, S. (1993). The pinwheel asynchronous atomic broadcast protocols. Rapport technique CSE93-331, University of California, San Diego.

[Cristian et Mishra, 1995] Cristian, F. et Mishra, S. (1995). The pinwheel asynchronous atomic broadcast protocols. *In Autonomous Decentralized Systems, 1995. Proceedings. ISADS 95., Second International Symposium on*, pages 215 –221.

[Défago *et al.*, 2003] Défago, X., Schiper, A. et Urbán, P. (2003). Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. Inf. Syst. E86-D*, pages 2698–2709.

[Défago *et al.*, 2004] Défago, X., Schiper, A. et Urbán, P. (2004). Total order broadcast and multicast algorithms : Taxonomy and survey. *ACM Comput. Surv.*, 36:372–421.

[Demers *et al.*, 1987] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D. et Terry, D. (1987). Epidemic algorithms for replicated database maintenance. *In PODC '87 : Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA. ACM.

[Dubois *et al.*, 1986] Dubois, M., Scheurich, C. et Briggs, F. (1986). Memory access buffering in multiprocessors. *In Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 434–442, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Dunagan *et al.*, 2004] Dunagan, J., Harvey, N. J. A., Jones, M. B., Kostic', D., Theimer, M. et Wolman, A. (2004). Fuse : Lightweight guaranteed distributed failure notification. *In In OSDI*, pages 151–166.

[Eychenne et Simatic, 1996] Eychenne, Y. et Simatic, M. (1996). Methods of broadcasting data by means of a data train. American patent number 5483520.

[Eychenne *et al.*, 1992] Eychenne, Y., Simatic, M., Baradel, C. et Kohen, B. (1992). Exploiting late binding in object messaging for implementing object replication. *In Proceedings of the 5th workshop on ACM SIGOPS European workshop : Models and paradigms for distributed systems structuring*, EW 5, pages 1–7, New York, NY, USA. ACM.

[Fidge, 1988] Fidge, C. J. (1988). Timestamps in message-passing systems that preserve the partial ordering. *In Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66. K. Raymond.

[Florin et Toinard, 1992] Florin, G. et Toinard, Ch. (1992). A new way to design causally and totally ordered multicast protocols. *SIGOPS Oper. Syst. Rev.*, 26:77–83.

[Friedman et van Renesse, 1997] Friedman, R. et van Renesse, R. (1997). Packing messages as a tool for boosting the performance of total ordering protocols. *In High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*, pages 233–242.

[GauthierDickey *et al.*, 2004] GauthierDickey, C., Zappala, D., Lo, V. et Marr, J. (2004). Low latency and cheat-proof event ordering for peer-to-peer games. *In Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '04, pages 134–139, New York, NY, USA. ACM.

[Gentes *et al.*, 2009a] Gentes, A., Jutant, C., Guyot, A. et Simatic, M. (2009a). Designing mobility : pervasiveness as the enchanting tool of mobility. *In Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on*, pages 1–4.

[Gentes *et al.*, 2009b] Gentes, A., Jutant, C., Guyot, A. et Simatic, M. (2009b). RFID technology : Fostering human interactions. *In* Blashki, K., éditeur : *Proceedings of IADIS International Conference Game and Entertainment Technologies 2009*, pages 67–74. International Association for Development of the Information Society (iadis), IADIS Press.

[Golding, 1992] Golding, R. A. (1992). *Weak-consistency group communication and membership*. Thèse de doctorat, University of California Santa Cruz.

[Gonzalez, 2008] Gonzalez, L. (2008). *RFID : Les enjeux pour l'entreprise (RFID : Stakes for the enterprise !, in French)*. Afnor Editions, `http://www.boutique-livres.afnor.org`. ISBN-13 978-2-12-465153-5.

[Green Code Lab, 2012] Green Code Lab (2012). *Green patterns - Manual for eco-design of software (in french)*. Lulu, first édition.

[Guerraoui *et al.*, 2010] Guerraoui, R., Levy, R. R., Pochon, B. et Quéma, V. (2010). Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, 28:5 :1–5 :32.

[Haberman *et al.*, 2009] Haberman, O., Pellerin, R., Gressier-Soudan, E. et Haberman, U. (2009). Rfid painting demonstration. *In* Natkin, S. et Dupire, J., éditeurs : *Entertainment Computing - ICEC 2009*, volume 5709 de *Lecture Notes in Computer Science*, pages 286–287. Springer Berlin / Heidelberg.

[Hadzilacos et Toueg, 1994] Hadzilacos, V. et Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Rapport technique, Cornell University, Ithaca, NY, USA.

[Herlihy et Mohan, 2003] Herlihy, M. et Mohan, A. (2003). Peer-to-peer multiplayer gaming using arrow multicast : Peer-to-peer quake. *In The 23rd International Conference on Distributed Computing Systems (ICDCS-2003)*. Providence, Rhode Island USA.

[Heumer *et al.*, 2007] Heumer, G., Gommlich, F., Jung, B. et Müller, A. (2007). Via Mineralia - a pervasive museum exploration game. *In Proc. of Pergames 2007, Salzburg, AT*.

[Juang *et al.*, 2002] Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L. S. et Rubenstein, D. (2002). Energy-efficient computing for wildlife tracking : design tradeoffs

and early experiences with zebranet. *In ASPLOS-X : Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107, New York, NY, USA. ACM.

[Kaashoek *et al.*, 1989] Kaashoek, M. F., Tanenbaum, A. S., Hummel, S. F. et Bal, H. E. (1989). An efficient reliable broadcast protocol. *SIGOPS Oper. Syst. Rev.*, 23:5–19.

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

[Levallois-Barth, 2009] Levallois-Barth, C. (2009). *L'évolution des cultures numériques (The evolution of digital cultures, in French)*, chapitre 5 – La sécurité et la protection des données / Navigo : simplification ou tra cabilité absolue (Security and data protection / Navigo : simplification or absolute traceability, in French), pages 173–181. FYP éditions. ISBN-13 978-2-91-657113-3.

[Mamei et Zambonelli, 2007] Mamei, M. et Zambonelli, F. (2007). Pervasive pheromone-based interaction with rfid tags. *ACM Trans. Auton. Adapt. Syst.*, 2(2):4.

[Mattern, 1988] Mattern, F. (1988). Virtual time and global states of distributed systems. *In Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France*, pages 215–226. Elsevier.

[Murphy et Picco, 2006] Murphy, A. L. et Picco, G. (2006). Using LIME to Support Replication for Availability in Mobile Ad Hoc Networks. *In Proceedings of the 8th International Conference on Coordination Models and Languages (COORD06)*, volume 4038, pages 194–211. Bologna, Italy, Springer Lecture Notes on Computer Science.

[Murphy *et al.*, 2006] Murphy, A. L., Picco, G. P. et Roman, G.-C. (2006). LIME : A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328.

[NFC Forum, 2006a] NFC Forum (2006a). NFC Data Exchange Format (NDEF) - Technical Specification 1.0. Rapport technique, NFC Forum.

[NFC Forum, 2006b] NFC Forum (2006b). Smart Poster Record Type Definition - Technical Specification 1.0. Rapport technique, NFC Forum.

[NFC Forum, 2006c] NFC Forum (2006c). URI Record Type Definition - Technical Specification 1.0. Rapport technique, NFC Forum.

[Oestreicher, 1991] Oestreicher, D. (1991). A simple reliable globally-ordered broadcast service. *SIGOPS Oper. Syst. Rev.*, 25:66–76.

[Parker *et al.*, 1983] Parker, D. S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S. et Kline, C. (1983). Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247.

[Pellerin *et al.*, 2005a] Pellerin, R., Delpiano, F., Duclos, F., Gressier-Soudan, E. et Simatic, M. (2005a). Gasp : an open source gaming service middleware dedicated to multiplayer games for J2ME based mobile phones. *In 7th Int. Conference on Computer Games CGAMES'05 Proceedings*, pages 75–82.

[Pellerin *et al.*, 2005b] Pellerin, R., Delpiano, F., Gressier, E. et Simatic, M. (2005b). GASP : Un intergiciel pour les jeux en réseaux multijoueurs sur téléphones

mobiles. *In UBIMOB 05 : Deuxièmes journées Ubiquité et Mobilité*, pages 61–64. Grenoble, France, ACM.

[Peterson *et al.*, 1989] Peterson, L. L., Buchholz, N. C. et Schlichting, R. D. (1989). Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7:217–246.

[Philippot, 2012] Philippot, O. (2012). Eco-designs of softwares and green patterns (in french). *Programmez !*, pages 34–41.

[Preguica *et al.*, 2009] Preguica, N., Marques, J. M., Shapiro, M. et Letia, M. (2009). A commutative replicated data type for cooperative editing. *In Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 395–403.

[Ranque *et al.*, 2011] Ranque, D., Rousseau, L., Stephan, R., Chevet, P.-F. et Appert, O. (2011). Technologies clés 2015 (Key technologies 2015, in French). Rapport technique, Ministère de l'Industrie, de l'Énergie et de l'Économie numérique.

[Rashid *et al.*, 2006] Rashid, O., Bamford, W., Coulton, P., Edwards, R. et Scheible, J. (2006). PAC-LAN : mixed-reality gaming with RFID-enabled mobile phones. *Computers in Entertainment*, 4(4):4–20.

[Raynal et Singhal, 1995] Raynal, M. et Singhal, M. (1995). Logical time : A way to capture causality in distributed systems. Rapport technique 2472, IRISA.

[Robert, 2000] Robert, Ph. (2000). *Réseaux et files d'attente : méthodes probabilistes*. Springer, first édition. ISBN 978–3–540–67872–4 (in French).

[Roussos et Kostakos, 2009] Roussos, G. et Kostakos, V. (2009). RFID in pervasive computing : State-of-the-art and outlook. *Pervasive Mob. Comput.*, 5(1):110–131.

[Saito et Shapiro, 2005] Saito, Y. et Shapiro, M. (2005). Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81.

[Schiper *et al.*, 1991] Schiper, A., Birman, K. et Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9:272–314.

[Schneider, 1990] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach : a tutorial. *ACM Comput. Surv.*, 22:299–319.

[Simatic, 2009a] Simatic, M. (2009a). RFID-based replicated distributed memory for mobile applications. *In Proceedings of the 1st International Conference on Mobile Computing, Applications, and Services (Mobicase 2009)*. San Diego, USA, ICST.

[Simatic *et al.*, 2009] Simatic, M., Astic, I., Aunis, C., Gentes, A., Guyot-Mbodji, A., Jutant, C. et Zaza, E. (2009). "Plug : Secrets of the Museum" : A pervasive game taking place in a museum. *In Entertainment Computing - ICEC 2009, Eighth International Conference, Paris, France, September 3-5, 2009, Proceedings*, Lecture Notes in Computer Science, pages 302–303. Springer.

[Simatic *et al.*, 1998] Simatic, M., Hurst-Frost, E., Junot, L., Kohen, B. et Orban, O. (1998). Method of ordering a plurality of messages from a plurality of sources and system for implementing the method. American patent number 5848228.

[Simner, 2007] Simner, D. (2007). Networks using MiFARE cards as the transport medium. Rapport technique, Jesus College.

[Smed *et al.*, 2001] Smed, J., Kaukoranta, T. et Hakonen, H. (2001). Aspects of networking in multiplayer computer games. *In Proceedings of The International Conference on Application and Development of Computer Games in the 21st Century.*

[Stevenson, 1964] Stevenson, R. (1964). Mary Poppins. 45^{th} anniversary DVD edited by Walt Disney. In France, movie was released on 15/09/1965.

[Toinard *et al.*, 1999] Toinard, Ch., Florin, G. et Carrez, Ch. (1999). A formal method to prove ordering properties of multicast systems. *SIGOPS Oper. Syst. Rev.*, 33:75–89.

[Urbán *et al.*, 2000a] Urbán, P., Défago, X. et Schiper, A. (2000a). Contention-Aware Metrics : Analysis of Distributed Algorithms. Rapport technique DSC/2000/012, École Polytechnique Fédérale de Lausanne (Switzerland).

[Urbán *et al.*, 2000b] Urbán, P., Défago, X. et Schiper, A. (2000b). Contention-aware metrics for distributed algorithms : comparison of atomic broadcast algorithms. *In Computer Communications and Networks, 2000. Proceedings. Ninth International Conference on*, pages 582 –589.

[Urban *et al.*, 2003] Urban, P., Shnayderman, I. et Schiper, A. (2003). Comparison of failure detectors and group membership : performance study of two atomic broadcast algorithms. *In Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 645 – 654.

[Vandeginste, 2010] Vandeginste, P. (2010). Billion of billions of operations per second (in french). *La Recherche*, pages 62–65.

[Videla et Williams, 2011] Videla, A. et Williams, J. J. (2011). *RabbitMQ in action.* Manning Publications, zero édition. Only available through Manning Early Access Program.

[Wiechert *et al.*, 2008] Wiechert, T. J. P., Thiesse, F., Michahelles, F., Schmitt, P. et Fleisch, E. (2008). Connecting Mobile Phones to The Internet Of Things, A discussion of compatibility issues between EPC technology and NFC technology. Rapport technique, Auto-ID Lab Switzerland.

[Yen et lu Huang, 1997] Yen, L.-H. et lu Huang, T. (1997). Resetting vector clocks in distributed systems. Rapport technique, Computer Science Department of Michigan State University. His.

[Zecca *et al.*, 2008] Zecca, G., Couderc, P., Banatre, M. et Beraldi, R. (2008). Cooperation in a swarm of robots using rfid landmarks. *In Robotic and Sensors Environments, 2008. ROSE 2008. International Workshop on*, pages 1–6.

[Zecca *et al.*, 2009] Zecca, G., Couderc, P., Banatre, M. et Beraldi, R. (2009). Swarm robot synchronization using rfid tags. *In Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on*, pages 1–4.

Webographie

[Act11] Activision : Skylanders : Spyro's adventure. Available at `http://www.skylanders.com/)` (last access in February 2012), 2011.

[Age12] Agence Française pour le Jeu Vidéo (AFJV) : Skylanders Spyro's Adventure au salon international du jouet (Skylanders Spyro's Adventure at toy international fare, in French). Available at http://www.afjv.com/news.php?id=655&title=skylanders_spyros_adventure) (last access in February 2012), February 2012.

[AMQ12] AMQP : Advanced Messaging Queuing Protocol. http://www.amqp.org/, February 2012.

[Apa07] Apache : Hadoop Distributed File System (last access in June 2012). http://hadoop.apache.org/hdfs/, September 2007.

[Asp11] Aspire RFID : OW2 Aspire RFID : an RFID suite for SMEs. Available at http://wiki.aspire.ow2.org) (last access in February 2012), 2011.

[Ban02] Bela Ban : JGroups — A Toolkit for Reliable Multicast Communication. http://www.jgroups.org, January 2002.

[Chr08] Christian D. : 700 million of users of NFC mobiles in 5 years (in French). http://www.generation-nt.com/juniper-etude-technologie-nfc-mobile-utilisateurs-actualite-151831.html, September 2008.

[CLM+09] CÉDRIC, L3i, Musée des arts et métiers, NET Innovations, Orange, Institut Télécom—Télécom & Management SudParis, Institut Télécom—Télécom ParisTech et Tetraedge : PLUG : PLay Ubiquitous Games and play more. http://cedric.cnam.fr/PLUG/, January 2009.

[Con11] Connecthings : 08/12/2011 - BAL intelligente @ mobulles Paris @ Lab Postal 2011 (Intelligent mailbox, in French). http://www.connecthings.com/node/123, December 2011.

[daem09b] Musée des arts et métiers : Plug Secrets of the Museum - TRAILER (last access in May 2012). http://www.dailymotion.com/video/xalc8t_plug-secrets-of-the-museum-trailer_creation, September 2009.

[Dav12] Joe Davies : NXP : One in ten smartphones sold in 2011 included NFC. Available at http://www.nfcworld.com/2012/02/10/313115/nxp-one-in-ten-smartphones-sold-in-2011-included-nfc/ (last access in March 2012), February 2012.

[ENe03] ENet : ENet. http://enet.bespin.org/ (last access in December 2011), February 2003.

[GCL10] GCL : Green Code Lab. Available at http://greencodelab.fr/ (last access in May 2012), 2010.

[Haz10] Hazelcast : Hazelcast, a clustering and highly scalable data distribution platform for Java (last access in June 2012). https://github.com/hazelcast/hazelcast/, October 2010.

[Hc11] Pieter Hintjens et 40+ contributors : 0mq - the guide. http://zguide.zeromq.org/page:all, August 2011.

[Hin08] Pieter Hintjens : What is wrong with AMQP (and how to fix it). http://www.imatix.com/articles:whats-wrong-with-amqp, August 2008.

[ITR06] ITR Manager.com : La Ville de Paris gère ses arbres avec des puces RFID (City of Paris is taking care of its trees with RFID tags, in French) (last access in March 2012). http://www.itrmanager.com/articles/59758/59758.html, December 2006.

[Jen05] Jenkins Software LLC : RakNet - Multiplayer game network engine. http://www.jenkinssoftware.com/ (last access in February 2012), 2005.

[Kno09] John Knottenbelt : Networks With Letters. http://www.gamasutra.com/view/feature/4052/networks_with_letters.php, August 2009.

[Mar99] Marauroa : Marauroa General. http://stendhalgame.org/wiki/Navigation_for_Marauroa_General (last access in February 2012), July 1999.

[Mur07] Amy L. Murphy : LIME, TinyLIME, TeenyLIME. Available at http://lime.sourceforge.net/ (last access in May 2012), 2007.

[PAD+07] Romain Pellerin, Gabriel Adgeg, Fabien Delpiano, Eric Gressier-Soudan et Michel Simatic : Gasp – a middleware for mobile multiplayer games. http://gasp.ow2.org, July 2007.

[Pla12] Seth Planck : Kids going crazy for Activision Skylanders NFC game. Available at http://www.nfcrumors.com/01-17-2012/kids-crazy-activision-skylanders-nfc/) (last access in March 2012), January 2012.

[Rep01] Replica Software : ReplicaNet. http://www.replicanet.com/ (last access in February 2012), 2001.

[SA11] Michel Simatic et Gabriel Adgeg : Alterland : jeu multijoueur transmédia sur le thème de l'écologie (Alterland : transmedia multiplayer game based on ecology theme, in French, last access in June 2012). http://www.youtube.com/watch?v=XTndb9LNbPU, June 2011.

[SAL07] SALTO Systems : SALTO Networked Locking System - SVN. Available at http://www.saltosystems.com/index.php?option=com_content&task=view&id=62&Itemid=57) (last access in March 2012), 2007.

[Sim09] Michel Simatic : Plug : secrets of the museum (last access in May 2012). http://plug-futur-en-seine.it-sudparis.eu/en/, August 2009.

[Sim12] Michel Simatic : About structures alignment (in french). Exercise 3 of "Memory management" exercises in lecture "CSC4508/M2 : Concepts of operating systems and implementation under Unix" (available at http://www-inf.it-sudparis.eu/modules/CS21/Current/Documents/WebCoursSys/index.html), 2012.

[Spr98] Spread : The Spread Toolkit. http://www.spread.org/, October 1998.

[TOT09] TOTEM : TOTEM - Theories and Tools for Distributed Authoring of Mobile Mixed Reality Games. http://www.totem-games.org/?q=overview (last access in February 2012), September 2009.

[Tou10] Touchatag : Using the Advanced HTTP Application. Available at http://www.touchatag.com/developer/docs/applications/advanced-HTTP (last access in February 2012), 2010.

Annexes

Annexe A

Preuves des théorèmes 2, 3 et 4

Dans cette annexe, nous prouvons les théorèmes 2, 3 et 4.

Rappel de l'énoncé du théorème 2. Nous considérons le cas où le circuit des trains est vide. Soient deux processus p_1 et p_2 qui cherchent simultanément à rejoindre le circuit des trains. Notez qu'il n'y a pas d'autres processus qui cherchent à rejoindre le circuit pendant cette phase. Si les processus p_1 et p_2 effectuent leur $k^e, k \in \mathbb{N}^*$, attente(s) dans l'état `AttenteAvantNouvelleTentativeInsertion`, la probabilité qu'ils se retrouvent en conflit après cette k^e attente (et rebasculent donc dans l'état `AttenteAvantNouvelle-TentativeInsertion`) est de l'ordre de : $(2D-1)/(2^{k-1}T)$.

Démonstration. Nous supposons que les tirages aléatoires faits par un processus sont indépendants entre eux et que les tirages aléatoires de p_1 sont indépendants de ceux de p_2.

Notons τ, l'unité de temps utilisée. Supposons que T et D sont des multiples de τ. Alors :

$$
\begin{aligned}
Proba(\text{conflit } k^e \text{ attente}) &= \sum_{i=0}^{2^{k-1}T-1} Proba(\text{conflit quand } p_1 \text{ attend } i\cdot\tau) \\
&= \sum_{i=0}^{D-1} Proba(\text{conflit quand } p_1 \text{ attend } i\cdot\tau) \\
&\quad + \sum_{i=D}^{2^{k-1}T-D-1} Proba(\text{conflit quand } p_1 \text{ attend } i\cdot\tau) \\
&\quad + \sum_{i=2^{k-1}T-D}^{2^{k-1}T-1} Proba(\text{conflit quand } p_1 \text{ attend } i\cdot\tau) \\
&= \sum_{i=0}^{D-1} Proba(p_1 \text{ attend } i\cdot\tau) \left(\sum_{j=0}^{i+D-1} Proba(p_2 \text{ attend } j\cdot\tau) \right) \\
&\quad + \sum_{i=D}^{2^{k-1}T-D-1} Proba(p_1 \text{ attend } i\cdot\tau) \left(\sum_{j=i-D+1}^{i+D-1} Proba(p_2 \text{ attend } j\cdot\tau) \right) \\
&\quad + \sum_{i=2^{k-1}T-D}^{2^{k-1}T-1} Proba(p_1 \text{ attend } i\cdot\tau) \left(\sum_{j=i-D+1}^{2^{k-1}T-1} Proba(p_2 \text{ attend } j\cdot\tau) \right) \\
&= \sum_{i=0}^{D-1} \frac{1}{2^{k-1}T} \left(\sum_{j=0}^{i+D-1} \frac{1}{2^{k-1}T} \right) \\
&\quad + \sum_{i=D}^{2^{k-1}T-D-1} \frac{1}{2^{k-1}T} \left(\sum_{j=i-D+1}^{i+D-1} \frac{1}{2^{k-1}T} \right) \\
&\quad + \sum_{i=2^{k-1}T-D}^{2^{k-1}T-1} \frac{1}{2^{k-1}T} \left(\sum_{j=i-D+1}^{2^{k-1}T-1} \frac{1}{2^{k-1}T} \right)
\end{aligned}
$$

$$
\begin{aligned}
Proba(\text{conflit } k^e \text{ attente}) &= \frac{1}{(2^{k-1}T)^2} \left[\sum_{i=0}^{D-1} (i+D) + \sum_{i=D}^{2^{k-1}T-D-1} (2D-1) + \sum_{i=0}^{D-1} (i+D) \right] \\
&= \frac{1}{(2^{k-1}T)^2} \left[2(\frac{D(D-1)}{2} + D^2) + (2^{k-1}T - 2D)(2D-1) \right]
\end{aligned}
$$

Comme $D \ll T$:

$$\begin{aligned} Proba(\text{conflit } k^e \text{ attente}) &\simeq \frac{1}{(2^{k-1}T)^2}[2^{k-1}T(2D-1)] \\ &\simeq \frac{2D-1}{2^{k-1}T} \end{aligned}$$

□

Rappel de l'énoncé du théorème 3. Considérons le cas où le circuit des trains est vide. Soient n processus $p_1,\ldots,p_{n,n\in[\![2,+\infty[\![}$ qui cherchent simultanément à rejoindre le circuit des trains. La probabilité qu'au moins 2 processus parmi $p_1,\ldots,p_n$ ne réussissent pas à s'insérer dans ce circuit au bout de ν tentatives est de l'ordre de : $[n(n-1)(2D-1)]^\nu/(2^{\frac{\nu(\nu+1)}{2}}T^\nu)$.

Démonstration. Si au moins 2 processus n'ont pas réussi à s'insérer au bout de ν tentatives, cela signifie qu'à chacune des tentatives, il y a eu au moins 2 processus qui ont été en conflit.

De ce fait :

$$\begin{aligned} Proba(\text{insert au moins 2 échoue}) &= Proba(\text{conflit } 1^{re} \text{ attente } \wedge \cdots \wedge \text{ conflit } \nu^e \text{ attente}) \\ &= \prod_{i=1}^{\nu} Proba(\text{conflit } i^e \text{ attente}) \\ &= \prod_{i=1}^{\nu} \left[C_2^n \cdot Proba(\text{conflit } i^e \text{ attente entre } p_{j,j\in[\![1,n]\!]} \text{ et } p_{k,k\in[\![1,n]\!]})\right] \end{aligned}$$

Donc, d'après le théorème 2 :

$$\begin{aligned} Proba(\text{insert au moins 2 échoue}) &\simeq \prod_{i=1}^{\nu}\left[C_2^n \cdot \frac{2D-1}{2^{i-1}T}\right] \\ &\simeq [C_2^n]^\nu \cdot \prod_{i=1}^{\nu}\frac{2D-1}{2^{i-1}T} \\ &\simeq \left[\frac{n(n-1)}{2}\right]^\nu \cdot \frac{(2D-1)^\nu}{2^{0+\cdots+(\nu-1)}T^\nu} \\ &\simeq \frac{[n(n-1)(2D-1)]^\nu}{2^{\frac{\nu(\nu+1)}{2}}T^\nu} \end{aligned}$$

□

Rappel de l'énoncé du théorème 4. Si nous supposons que T est suffisamment grand pour que $\forall i \in [\![2,+\infty[\![, \frac{1}{T^i} \ll \frac{1}{T}$, alors : $E(X_n) \simeq \frac{T}{2} + \frac{n(n-1)(2D-1)}{4}$

Démonstration.

$E(X_n) = x_{n_1} \times Proba(x_{n_1}) + \cdots + x_{n_\nu} \times Proba(x_{n_\nu})$

Appliquons le théorème 3 pour estimer Proba$(x_{n_{i,i\in[\![1,\nu]\!]}})$:

$$\begin{aligned} E(X_n) &= \frac{T}{2} \times (1 - \frac{n(n-1)(2D-1)}{2T}) + \frac{2T}{2} \times \frac{n(n-1)(2D-1)}{2T}(1 - \frac{[n(n-1)(2D-1)]^2}{8T^2}) + \dots \\ &= \frac{T}{2}\left[(1 - \frac{n(n-1)(2D-1)}{2T}) + 2\frac{n(n-1)(2D-1)}{2T}(1 - \frac{[n(n-1)(2D-1)]^2}{8T^2}) + \dots\right] \end{aligned}$$

Nous supposons que T est suffisamment grand pour que $\forall i \geqslant 2, \frac{1}{T^i} \ll \frac{1}{T}$. Donc :

$$\begin{aligned} E(X_n) &\simeq \frac{T}{2}\left[1 - \frac{n(n-1)(2D-1)}{2T} + 2\frac{n(n-1)(2D-1)}{2T}\right] \\ &\simeq \frac{T}{2}\left[1 + \frac{n(n-1)(2D-1)}{2T}\right] \\ &\simeq \frac{T}{2} + \frac{n(n-1)(2D-1)}{4} \end{aligned}$$

□

Annexe B

Curriculum Vitæ

B.1 État civil

Michel SIMATIC
Né le 15/09/1965 à Beyrouth (Liban)
Nationalité : Française
Situation de famille : Marié (4 enfants)
Lieu de travail actuel : Télécom SudParis

B.2 Diplômes

1990	Ingénieur IIE (Institut d'Informatique d'Entreprise) Enseignants : Gérard Berthelot, Gérard Florin, etc.
1990	D.E.A. Systèmes répartis (Université Paris VI) Enseignants : Clause Girault, Claude Kaiser, Stéphane Natkin, Marc Shapiro, etc.

B.3 Parcours professionnel

2003–...	Télécom SudParis Directeur d'Études
2001–2002	Alcatel Responsable d'une équipe de 15 personnes, chargée de l'outillage logiciel et des méthodes pour le support d'une équipe de développement de 150 personnes
1997–2000	Alcatel Architecte système
1992–1996	Alcatel Alsthom Recherche Chef de projets recherche
1990–1991	Alcatel Développeur logiciel

B.4 Recherche

B.4.1 Thématiques de recherche

Ma thématique de recherche principale concerne la Réplication de données dans un environnement réparti.

De 1992 à 1996, j'ai étudié la réplication de données dans une optique de tolérance aux fautes. Cela m'a conduit à l'étude des protocoles d'uot-diffusion. Compte tenu des contraintes du système cible (le système de contrôle-commande et de supervision *P3200*), j'ai développé un intergiciel original utilisant la version monotrain du protocole des trains. Puis, j'ai contribué à son intégration dans le système cible.

En parallèle, de 1994 à 2000, je me suis intéressé à un autre volet de ma thématique : la gestion des données de configuration dans un environnement réparti. Cela m'a amené à l'étude de modèles XML (eXtensible Markup Language) permettant de représenter l'ensemble des données de configuration d'un système et leur projection, à l'aide de feuilles XSL (eXtensible Stylesheet Language), vers les différents configurateurs du système. Le changement de données de configuration à l'exécution a été étudié.

De 1997 à 2000, je me suis également intéressé à la problématique de l'interfaçage de systèmes d'information. Des données existent dans un système d'information principal (par exemple, le système d'information commercial d'un opérateur téléphonique). Un sous-ensemble de ces données est également présent dans un système d'information annexe (par exemple, le système d'information des machines du réseau téléphonique de cet opérateur). Comment mettre en cohérence ces données répliquées dans les deux systèmes d'information ? Après études, j'ai promu l'utilisation d'intergiciels de communication CORBA.

De 2003 à aujourd'hui, je me suis intéressé à la réplication de données dans le domaine des jeux vidéo sur téléphone mobile. Cela m'a amené à contribuer au développement de l'intergiciel de communication *GASP*, de l'intergiciel de communication *ZebroGaMQ*, et à étudier une mémoire distribuée partagée basée sur des étiquettes RFID.

Enfin, de 2010 à aujourd'hui, j'ai intensifié mon activité de recherche autour des protocoles d'uot-diffusion.

B.4.2 Publications

B.4.2.1 Chapitres de livres [2 chapitres]

- I. Astic, C. Aunis, J. Dupire, V. Gal, E. Gressier-Soudan, Ch. Pitrey, M. Roy, F. Sailhan, M. Simatic, A. Topol, E. Zaza. "Chapitre 12 ? Entre jeux pervasifs et applications critiques". Editeur : HERMES Science Publishing Ltd. À paraître en 2012.
- E. Gressier-Soudan, R. Pellerin, M. Simatic. "Chapter 10. Using RFID/NFC for pervasive serious games : the PLUG experience". Editors : Syed Ahson, Microsoft, Redmond, Washington, Mohammad Ilyas, College of Engineering & Computer Science, Florida Atlantic University. CRC Press. Taylor and Francis. Sep 23th 2011. pp279-303. ISBN 9781420088144

B.4.2.2 Revues internationales [1 revue]

- Fault-tolerance and on-line maintainability in a process control supervision system , C. Baradel, Y. Eychenne, L. Junot, B. Kohen and M. Simatic, Distributed System Engineering, Vol. 2, n° 2, pp. 65-73, June 1995.

B.4.2.3 Revues nationales [1 revue]

- Placement et migration de processus dans les systèmes répartis faiblement couplés, G. Bernard, D. Stève and M. Simatic, Technique et Science Informatiques, vol. 10, n°5, 1991.

B.4.2.4 Conférences internationales [9 articles longs, 2 démonstrations]

- M. Simatic. RFID-based replicated distributed memory for mobile applications. In Proceedings of the 1st International Conference on Mobile Computing, Applications, and Services (Mobicase 2009). San Diego, USA, ICST, October 2009. [article long]
- M. Simatic, I. Astic, C. Aunis, A. Gentes, A. Guyot-Mbodji, C. Jutant, and E. Zaza. "Plug : Secrets of the Museum" : A pervasive game taking place in a museum. In Entertainment Computing - ICEC 2009, Eighth International Conference, Paris, France, September 3-5, 2009, Proceedings, Lecture Notes in Computer Science. Springer, September 2009. [démonstration]
- Gentes, C. Jutant, A. Guyot, and M. Simatic. RFID technology : Fostering human interactions. In K. Blashki, editor, Proceedings of IADIS International Conference Game and Entertainment Technologies 2009, pages 67-74. International Association for Development of the Information Society (IADIS), IADIS Press, June 2009. [article long]
- ShareX3D, a scientific collaborative 3D viewer over HTTP, S. Jourdain, J. Forest, Ch. Mouton, B. Nouailhas, G. Moniot, F. Kolb, S. Chabridon, M. Simatic, A. Zied and L. Mallet, Web3D 2008 : 13th International Symposium on 3D Web Technology, August 9-10, Los Angeles, California, USA, New York, NY : ACM, pp. 35-41, 2008. [article long]
- uGASP (ubiquitous Gaming Services Platform) : an OSGi based midlleware , R. Pellerin, E. Gressier-Soudan and M. Simatic, Demonstration at International Conference on Pervasive Systems (ICPS), Sorrento, Italy, July 2008. [demonstration]
- Issues for multiplayer mobile game engines, A. Rawat and M. Simatic, Proceedings of Game-On 2007, 8th International Conference on Intelligent Games and Simulation, Bologna, Italy, pp. 76-82, November 2007. [article long]
- GASP : an Open Source Gaming Service Middleware Dedicated to Multiplayer Games for J2ME Based Mobile Phones, R. Pellerin, F. Delpiano, F. Duclos, E. Gressier-Soudan and M. Simatic, CGAMES'2005 7th International Conference on Computer Games : AI, Animation, Mobile, Educational and Serious Games. [article long]
- Some technical and usage issues for mobile multiplayer games, M. Simatic, S. Craipeau, A. Beugnard, S. Chabridon, M-C Legout and E. Gressier, Proceedings of International Conference on Computer Games : Artificial Intelligence, Design and Education, Reading, UK, 2004. [article long]

- Service management : The forgotten side of IN service development, M. Simatic and B. Vilain , Proceedings of 5th International Conference on Intelligence of Networks, Bordeaux, 13-15 May 1998. [article long]
- The use of object groups to implement dependability in a process control supervision system, Y. Eychenne, M. Simatic, C. Baradel, L. Junot and B. Kohen, 23rd International Symposium on Fault-Tolerant Computing, Toulouse, France, 22-24 June 1994. [article long]
- A decentralized and efficient algorithm for load sharing in networks of workstations, G. Bernard and M. Simatic, EurOpen '91, Tromsö, 20-24 May 1991. [article long]

B.4.2.5 Workshops internationaux [1 article long]

- Designing mobility : pervasiveness as the enchanting tool of mobility, A. Gentes, C. Jutant, A. Guyot, and M. Simatic, Proceedings of the 1st international ICST Workshop on Innovative Mobile User Interactivity (IMUI 2009). San Diego, USA, ICST, 29th October 2009. [article long]

B.4.2.6 Conférences nationales [1 article court]

- GASP : un intergiciel pour les jeux en réseaux multijoueurs sur téléphones mobiles, R. Pellerin, F. Delpiano, E. Gressier, M. Simatic, UBIMOB 05 : Deuxièmes journées Ubiquité et Mobilité, pages 61-64, 2005. [article court]

B.4.2.7 Rapports [5 rapports]

- Jeux sur Mobiles : Technologies et Usages (JEMTU) / Délivrable D0.2 / Rapport de synthèse, M. Simatic, N. Auray, A.Beugnard, S. Chabridon, S. Craipeau, F. Dagnat, G. Dubey, J-C Moissinac, M. Preda et B. Seys, 2007.
- Architecture pour jeux multijoueurs en ligne sur mobile (MEGA) / Rapport de synthèse, M. Simatic, A. Beugnard, S. Chabridon, S. Craipeau, B. Defude, M-C Legout et E. Gressier, 2005.
- Document de Conception Préliminaire de Sego version 3, M. Simatic, Rapport de recherche interne Alcatel, Janvier 1997.
- Performances de Sego révision 1.2, M. Simatic, Rapport de recherche interne Alcatel, Décembre 1993.
- Diffusion fiable et ordonnée, M. Simatic, Rapport de recherche interne Alcatel, Mai 1993.

B.4.2.8 Brevets [2 brevets]

- Method of ordering a plurality of messages from a plurality of sources and system for implementing the method, L. Junot, E. Hurst-Frost, B. Kohen, O. Orban and M. Simatic, Brevet européen n°784270, 1998.
- Y. Eychenne and M. Simatic, Procédé de diffusion à train de données, Brevet européen n°650280, 1996.

B.4.3 Encadrement de jeunes chercheurs

2008 M. Chebira (CNAM)
Environnement de développement de jeux pervasifs et ubiquitaires
stage Master Recherche, encadrement à 100%

2006 S. Kassabian (CNAM)
Plateformes pour le développement de jeux sur téléphones mobiles
stage de Master Recherche, encadrement à 100%

2004 R. Pellerin (Paris 6)
Évaluation de plates-formes de développement de jeux en réseau pour téléphones mobiles
stage de D.E.A., encadrement à 50%, les 50% restants étant assurés par
E. Gressier-Soudan (CNAM-Cédric)

2004 E. Hetmanski (Évry)
Architectures pour jeux sur mobile adaptées aux temps de latence des réseaux mobiles 2G/3G
stage de DEA, encadrement à 100%

2004 S. Falempin (Paris 13)
Algorithmes de réconciliation dans le domaine applicatif des jeux sur mobile
stage de DEA, encadrement à 100%

B.4.4 Contrats de recherche

- 2012–2013 : Projet Eurostars ICEbreak (Interactive Cinema Experience through advanced storyboarding and mobile devices to BREAK the fun frontier), 1,5 an
  - Contribution à la tâche T3.2 (Communication Middleware)
- 2009–2012 : Projet Inter Carnot-Fraunhofer TOTEM (Theories and Tools for Distributed Authoring of Mobile Mixed Reality Games), 3 ans
  - Leader du WP5 (Infrastructure for MMRG)
  - Contribution à la tâche T5.3 (Infrastructure for MMRG / Iterative development of Game Server)
  - Contribution au Work package WP6 (Game Design and Development)
- 2009–2011 : Projet DGE PlayOnLine, 2 ans
  - Contribution à la tâche T4.8 (Inter-plateformes / Conseils, spécifications & transfert de compétences)
- 2008–2009 : Projet ANR PLUG (Play Ubiquitous Game and play more), 2 ans
  - Co-responsabilité du projet
  - Contribution à la tâche T4.2 (Conception et réalisation du jeu pervasif)
  - Contribution à la tâche T4.3 (Déploiement du jeu in situ)
  - Contribution à la tâche T5.3 (Middleware pour les jeux pervasifs)
- 2006–2008 : Projet financé en interne par l'Institut Mines-Télécom (ex-GET) JEMTU (Jeux sur Mobiles : Technologies et Usages), 3 ans
  - Responsabilité du projet
  - Contribution à la tâche T4.2 (Brique technologique Intergiciel de communication)
  - Contribution à la tâche T4.5 (Plateforme pour le développement de jeux multijoueurs sur mobile)
  - Contribution au sous-projet SP1 (Conception et réalisation des jeux)

- 2004 : Projet financé en interne par l'Institut Mines-Télécom (ex-GET) MEGA (Architecture pour jeux multijoueurs en ligne sur mobile), 1 an
  - Responsabilité du projet
  - Contribution à la tâche T1 (Exigences dans le domaine des jeux)
  - Contribution à la tâche T2a (Étude d'intergiciels disponibles)
  - Contribution à la tâche T3 (Architectures adaptées au temps de latence des réseaux mobiles)
  - Contribution à la tâche T6 (Démonstration)
- 1996 : Projet financé en interne par Alcatel Interface entre 2 systèmes de configuration, 1 an
  - Responsabilité du projet
  - Contribution à la tâche de synthèse des modèles de configuration
  - Contribution à la tâche de spécification de l'interface
  - Contribution à la tâche de développement de l'outil d'interface
- 1994–1995 : Projet financé en interne par Alcatel Interconnexion logiciel à base d'objets / SGBD relationnel, 2 ans
  - Responsabilité du projet
  - Contribution à la tâche d'état de l'art
  - Contribution à la tâche de spécification de l'interconnexion
  - Contribution à la tâche de démonstration
- 1992–1995 : Projet financé en interne par Alcatel Tolérance aux fautes de processus Unix ou VMS, 4 ans
  - Responsabilité du projet à partir de 1994
  - Contribution à la tâche d'état de l'art
  - Contribution à la tâche de spécification de l'intergiciel support du mécanisme de tolérance
  - Contribution à la tâche de réalisation de cet intergiciel
  - Contribution à la tâche d'intégration de cet iontergiciel dans l'application cible

B.5 Enseignement

B.5.1 Synthèse

L'essentiel de mon activité d'enseignement concerne le domaine informatique (initiation à l'informatique, initiation au développement, programmation système, etc.). Ces cours se répartissent en cours magistraux (15% de l'activité), TPs (35% de l'activité, et cours intégrés (cours avec application directe sur machine, 40% de l'activité).

En cours système, j'ai créé un polycopié de cours système (`http://www-inf.it-sudparis.eu/modules/CS21/Current/Poly/poly.pdf`), un ensemble d'exercices (`http://www-inf.it-sudparis.eu/modules/CS21/Current/Documents/WebCoursSys/Contenu/index.html`) et 13 sujets d'examen (`http://www-inf.it-sudparis.eu/modules/CS21/Current/Documents/WebCoursSys/Evaluation/index.html`). Des vidéos de la plupart de ces cours système ont été enregistrées pour faciliter le travail de reprise du cours fait par les étudiants (`http://tmsp.ubicast.eu/`, accès réservé aux membres de Télécom SudParis).

En cours d'initiation à l'informatique, j'ai créé une épreuve d'évaluation du niveau des étudiants inscrits. Par ailleurs, j'ai changé le format d'examen. Auparavant, l'examen était sur table. Il se déroule désormais sur machine (les 192 étudiants passant l'épreuve en 2 vagues de 96 étudiants).

De plus, pour un master recherche, j'ai créé les cours « Jeux multijoueurs en réseau sur terminaux mobiles » (3 heures) et « Issues for pervasive games » (3 heures).

Par ailleurs, chaque année, je coordonne 1 option (constituée de 7 U.V.), 2 cours et j'encadre une moyenne de 4 projets d'élève-ingénieur.

Enfin, je coordonne actuellement une équipe constituée d'enseignants-chercheurs de l'ENSIIE, de Télécom SudParis et de l'Université d'Évry-Val de Seine pour créer une option « Jeux vidéo » qui sera commune à l'ENSIIE et l'ensemble des écoles de l'Institut Mines-Télécom (ouverture de cette option prévue en septembre 2013).

B.5.2 Détail des cours

- Cours « Initiation au système Unix » (1 groupe de 24 étudiants de 1ère année, 15h), 2004–2005, 2005–2006, 2006–2007
- Cours « Architecture Matérielle des ordinateurs » (1 groupe de 24 étudiants de 1ère année de Télécom SudParis, 21 h), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2003–2004
- Cours « Concepts des Systèmes d'Exploitation » (20 étudiants de 2ème année de Télécom SudParis, 21 h), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2003–2004, 2003
- Cours « Programmation orientée objet » (1 groupe de 24 étudiants de 2ème année de Télécom SudParis, 51h)
- Cours « Jeux multijoueurs en réseau sur terminaux mobiles » (10 étudiants de Mastère MOPS (Evry) + 10 étudiants de Summer School IT + 10 étudiants de MS Marketing + 2 x 10 étudiants de 3ème année de Télécom SudParis), 2008–2009, 2007–2008, 2006–2007, 2005–2006 o Cours « Programmation Système Linux » (20 étudiants de formation continue Alcatel, 18h), 2003–2004
- Cours « Initiation au système Unix » (1 groupe de 24 étudiants de 1ère année de Télécom École de Management, 15h), 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2003–2004
- Cours d'introduction à l'« Introduction à la programmation » (1h30, 180 étudiants de 1ère année de Télécom École de Management), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005
- Cours « Introduction à la programmation » (21h, 1 groupe de 24 étudiants de 1ère année de Télécom École de Management), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2004–2005, 2003

B.5.3 Coordinations de cours

- Coordination de la Voie d'Approfondissement (option ou filière) de 3ème année de l'ENSIIE et des écoles de l'Institut Mines-Télécom (ouverture prévue en septembre 2013)

- Coordination de la Voie d'Approfondissement (option) de 3ème année ASR (Architecte de Services en Réseau), 2008–2009, 2007–2008. Cette VAP concerne 15 étudiants de 3ème année de Télécom SudParis. Elle est constituée de 7 U.V.
- Coordination de l'UV « Conception et programmation des systèmes centralisés » (concerne 24 étudiants de 2ème année de Télécom SudParis), 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2003–2004, 2002–2003
- Coordination du cours « Initiation au système Unix » (concerne 192 étudiants de 1ère année de Télécom École de Management), 2006–2007, 2005–2006

B.5.4 Encadrements de projets

- Encadrement de projets de 3ème année de Télécom SudParis (groupe de 2 étudiants travaillant 190h chacun), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007
- Encadrement de projets de 2ème année de Télécom SudParis (groupe de 4 étudiants travaillant 150h chacun), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2003–2004
- Encadrement de projets de 1ère année de Télécom SudParis (groupe de 4 étudiants travaillant 50h chacun), 2011–1012, 2010–2011, 2009–2010, 2008–2009, 2007–2008, 2006–2007, 2005–2006, 2004–2005, 2003–2004

B.6 Responsabilités

B.6.1 Encadrement

- 2001–2002 : Au sein d'Alcatel, encadrement d'une équipe de 15 personnes (2 techniciens, 13 ingénieurs, âge de 32 à 57 ans) réparties sur 3 sites (Lannion, Shangaï et Vélizy). Cette équipe assurait le support des équipes de développement du Radio Control Point (RCP) et du Home Location Register (HLR) Alcatel (150 personnes).
- 1993–1996 : Au sein d'Alcatel Alsthom Recherche, encadrement de deux équipes de 2 personnes (2 ingénieurs, chacune). Ces équipes réalisaient des études de recherche et développement pour le compte de la société Cegelec.

B.6.2 Gestion de contrats

- 2008–2009 : Au sein de Télécom SudParis, co-gestion du projet ANR PLUG (`http://cedric.cnam.fr/PLUG/`) avec E. Gressier-Soudan (CNAM-Cedric). Subvention ANR de 402 kEuros dont 116 kEuros gérés pour le compte de l'Institut Télécom.
- 2006–2008 : Au sein de Télécom SudParis, gestion de JEMTU (`http://projetsrecherche.int-edu.eu/JEMTU/`), projet de recherche financé en interne par l'Institut Télécom (ex-GET). Budget de 450 kEuros.
- 2004 : Au sein de Télécom SudParis, gestion de MEGA (`http://bscw.enst-bretagne.fr/pub/bscw.cgi/d3043906-1/*/MEGA_Presentation_VF.html`), projet de recherche financé en interne par l'Institut Télécom (ex-GET). Budget de 10 kEuros

– 1992–1996 : Au sein d'Alcatel Alsthom Recherche, gestion de contrats de recherche au sein d'Alcatel. Budgets de 1,5 à 4,5 MFrancs.

Glossaire

- *circuit des trains* : Anneau virtuel sur lequel sont répartis les processus participant au protocole des trains.

- *Mbps* : Le Mbps est le symbole du mégabit par seconde, un mégabit correspondant à un million de bits.

- *RDSM* : Mémoire répartie partagée à base de RFID (abrégée en RDSM pour RFID-*based Distributed Shared Memory*)

- *Train (de messages)* : Jeton qui circule sur le circuit de train et qui transporte les messages à uot-diffuser.

- *Train obsolète* : Voir *Train périmé*.

- *Train périmé* : Un train t est *périmé* (ou *obsolète*) pour un processus p quand t ne contient que des messages que p a déjà reçus.

- *Train récent* : Quand un processus p reçoit un train t, ce train est *récent* pour p quand t contient des messages que p n'a jamais reçus.

- *uot-diffuser* : Appeler la primitive `uot-diffusion(m)` sur un message m donné.

- *uot-diffusion* : Type de protocole vérifiant les propriétés spécifiées en section 3.1, ou bien invocation par un processus p de la primitive `uot-diffusion(m)` sur un message m donné (voir section 3.1 pour plus d'informations).

- *uot-livrer* : Faire une uot-livraison.

- *uot-livraison* : Invocation par un processus p de la primitive `uot-livraison(m)` sur un message m donné (voir section 3.1 pour plus d'informations).

Résumé :

L'objectif de cette thèse est d'exploiter deux outils fondamentaux des systèmes répartis asynchrones : l'horloge logique et le vecteur d'horloges. Dans une première partie, nous nous concentrons sur la communication de données et contribuons au domaine de la diffusion uniforme à ordre total. Nous proposons le protocole des trains qui utilise des horloges logiques. À l'aide de notre métrique Rendement en termes de débit, nous prouvons que notre protocole a un rendement supérieur au meilleur, en termes de débit, des protocoles présentés dans la littérature. Les performances en termes de débit du protocole des trains, notamment pour les messages de petites tailles, en font un candidat remarquable pour le partage de données entre cœurs d'un même processeur. Dans une seconde partie, nous nous concentrons sur le partage de données et contribuons au domaine de la RFID. Nous proposons une mémoire répartie partagée basée sur des étiquettes RFID. Cette mémoire s'appuie sur des vecteurs d'horloges et exploite le réseau formé par les utilisateurs mobiles de l'application répartie. Ainsi, ces derniers peuvent lire le contenu d'étiquettes RFID distantes. Notre mémoire répartie partagée à base de RFID apporte une alternative aux trois architectures à base de RFID disponibles dans la littérature.

Mots clés :

Systèmes répartis, Horloge logique, Vecteur d'horloges, Mémoire distribuée partagée, RFID, NFC, Diffusion uniforme à ordre total

Abstract :

The goal of this PhD thesis is to exploit two core tools asynchronous distributed systems : logical clock and vector clocks. In the first part of this thesis, we focus on data communication and contribute to the total order broadcast domain. We propose trains protocol which uses logical clocks. Thanks to our metric Throughput efficiency, we prove that, from a throughput point of view, trains protocol performs better than protocols presented in literature. Thanks to its throughput performances, in particular for small messages, trains protocol is a remarkable candidate for data sharing between the cores of a processor. In the second part of this thesis, we focus on data sharing and contribute to RFID domain. We propose a distributed shared memory based on RFID tags. This memory uses vector clocks and relies on the network made by the mobile users of the distributed application. Thus, the users are able to read the contents of remote RFID tags. Our RFID-based distributed shared memory is an alternative to the three RFID-based architectures available in the literature.

Keywords :

Distributed systems, Logical clock, Vector clocks, Distributed shared memory, RFID, NFC, Totally ordered broadcast