Réplication optimiste pour les applications collaboratives asynchrones

Olivier Dedieu

Numéro d'identification : xxxx

Thèse

présentée à

L'Université de Marne la Vallée

en vue de l'obtention du titre de

Docteur de l'Université de Marne la Vallée

Spécialité :

Systèmes Informatiques

par

Olivier Dedieu

Sujet de la thèse :

Réplication optimiste pour les applications collaboratives asynchrones

Soutenue le 14 septembre 2000, devant le jury composé de :

Messieurs	Dominique	Perrin	Président
	Patrick	Valduriez	Rapporteurs
	Daniel	Hagimont	
	Vincent	Bouthors	Examinateurs
	Guy	Ferran	
	Mesaac	Makpangou	

Résumé

Dans cette thèse, nous présentons un système de réplication optimiste pour les applications collaboratives asynchrones. Dans notre modèle, les réplicas travaillent de façon autonome et se synchronisent deux à deux afin d'assurer la propagation épidémique des écritures et la cohérence globale à terme des données. Nous avons conçu pour cela un protocole qui garantit des échanges incrémentaux. Il optimise la consommation de bande passante et simplifie la détection des mises-à-jour à intégrer. Il repose sur un mécanisme de journalisation des écritures. Ce mode de persistance offre de bonnes performances, une grande robustesse et un support simple pour l'évolution de schéma. Lorsque deux réplicas se synchronisent, des conflits peuvent apparaître. Pour y faire face, nous proposons un mécanisme d'horloge logique qui définit un ordre total sur les écritures, cohérent avec l'ordre temporel. Il permet de résoudre les mises-à-jour conflictuelles par un ordonnancement conforme à l'intuition pour les utilisateurs. D'autres types de conflits propres à l'application et à l'intégrité des données peuvent survenir. Le développeur d'applications dispose d'une interface de programmation pour définir des procédures de résolution spécifiques. Par ailleurs, nous présentons deux extensions au protocole de propagation épidémique pour le rendre exploitable à large échelle : le contrôle de la vitesse de convergence et la réplication partielle. En complément de la réplication des données, nous proposons un « framework » pour le déploiement dynamique des mises-à-jour du code et des ressources de l'application. Enfin, l'ensemble du système de réplication est illustré sur une application collaborative asynchrone, Pharos, qui permet le partage de recommandations dans des communautés d'intérêt.

Je remercie très sincèrement :

Dominique Perrin, Président de l'université de Marne-la-Vallée, pour m'avoir permis d'entreprendre cette thèse et pour avoir porté une continuelle attention à mon travail. C'est un grand honneur qu'il me fait en présidant le jury de cette thèse.

Patrick Valduriez, Directeur de recherche à l'INRIA, et Daniel Hagimont, Chargé de recherche à l'INRIA, pour avoir accepter d'être les rapporteurs de cette thèse. Ainsi que Guy Ferran pour avoir accepté de faire partie du jury.

Vincent Bouthors, Directeur de l'action Webtools à Dyade, pour sa profonde investigation dans mon travail. Il a cru en moi en me proposant cette thèse. Il a su me donner une grande liberté tout en me soutenant dans mes choix. J'ai beaucoup appris en travaillant avec lui.

Mesaac Makpangou, Chargé de recherche à l'INRIA, responsable scientifique de l'action Système à Objets Répartis (SOR), qui a toujours été présent pour m'aider dans mes travaux. Il a lu et relu mon manuscrit et ses remarques ont été précieuses.

Marc Shapiro, Directeur de recherche à l'INRIA, ancien responsable scientifique du projet SOR, pour m'avoir accueilli dans son équipe et pour son intérêt soutenu pour mon travail.

Tous les membres du projet SOR avec qui il a toujours été agréable de travailler. Nombre d'entre-eux se sont investis dans les travaux de cette thèse en testant les premières implémentations. Je tiens en particulier à remercier Guillaume Pierre qui a été le premier et qui continue d'utiliser régulièrement Pharos. Ces remarques ont toujours été pertinentes et ses réflexion enrichissantes.

Valérie, mon épouse, pour avoir décellé les si nombreuses fautes d'orthographes du manuscrit et pour tous les efforts qu'elle a consentis durant ces trois dernières années.

Enfin, je remercie mes parents pour avoir continuellement soutenu mes choix et m'avoir toujours apporté leur aide pour les réaliser.

Table des matières

Table des figures

Première partie

Problématique

Chapitre 1

Internet : de la consultation à la collaboration asynchrone

1.1 Multiplication des points d'accès à l'information

L'accès à l'information sous forme numérique est devenue globale. Cela a été rendu possible grâce à la mise en réseaux de tous les appareils qui offrent des points d'accès à ces informations. Si les ordinateurs[1] représentaient encore la majorité des machines connectées en réseaux, des analyses prévoient un prochain renversement de tendance. Au courant de l'année 2000, la majorité des machines en réseau sera constituée de petits appareils nomades, tels que les PDA[2] et des téléphones portables.

Tout un chacun peut aujourd'hui consulter à distance des ressources multimédia. Cependant, l'infrastructure de communication existante limite les possibilités d'accès. En effet, si les solutions de connexion permanente à l'Internet se multiplient[3], celles-ci restent encore trop coûteuses, peu déployées et inadaptées à l'informatique nomade. La grande majorité des points d'accès n'est donc connectée que ponctuellement à l'Internet. Des solutions d'infrastructure sont proposées pour rendre permanente les connexions des machines fixes mais aussi mobiles. Cependant, leur diffusion reste encore confidentielle et on peut supposer que leur déploiement sera concentrée en priorité sur des îlots à forte densité urbaine. Avant un déploiement réellement global, l'accès à l'information restera donc ponctuel ou coûteux pour la majorité des utilisateurs.

Le développement d'outils complémentaires pour l'accès à l'information poussent les utilisateurs à multiplier leur équipement. Beaucoup d'entre eux ont maintenant plusieurs points d'accès à Internet : ordinateur sur le lieu de travail, à la maison, ordinateur portable, PDA, téléphone portable, ... Outre la consultation de ressources extérieures (p. ex. des pages HTML), l'utilisateur gère des données qu'il doit partager entre ces différentes machines. Il peut s'agir de documents

[1] Ordinateurs personnels, stations de travail, ordinateurs portables et serveurs.
[2] *Personal Digital Assitant*
[3] Liaisons spécialisées, RNIS, ADSL ou connexion par le câble.

personnels (carnet d'adresses, agenda, signets d'URL, ...) ou de documents professionnels (documents bureautiques, agenda de groupes, bases de données, ...)

Pour réaliser ce partage, l'utilisateur peut opter pour une solution centralisée. Dans ce cas, il place les données à partager sur une machine accessible par toutes les autres. Cette solution a l'avantage d'être simple à concevoir mais pose des problèmes de mise-en-œuvre. D'une part, le coût de mise en place d'une machine offrant un espace de stockage en lecture et écriture est élevée. De plus, bien souvent cette machine est hébergée dans un environnement sécurisé (p. ex. un pare-feu) ce qui en limite les accès. On ne peut donc généralement pas opté pour une solution de type système de fichiers partagés (NFS, Samba, ...) D'autre part, pour être exploitable, chaque point d'accès doit dialoguer avec cette machine ce qui est coûteux, peu performant avec des liaisons par modem et nécessite une connexion durant toute la durée de l'utilisation.

La solution duale consiste à disposer d'une copie, complète ou partielle, des données partagées sur chacun des points d'accès. Tant que les accès se font en lecture, les données sont accessibles à tout moment et à moindre coût. Par contre, dès que l'une des copies est modifiée, l'ensemble devient globalement incohérent. De plus, si des mises-à-jour ont lieu concurremment sur différentes copies, des conflits peuvent apparaître. Par exemple, deux modifications différentes d'un même enregistrement dans une base de données entraîneront un conflit lors des mises-à-jour des copies, si rien n'est prévu pour l'éviter. Pour faire face à ces problèmes, il est nécessaire de convenir d'un mécanisme assurant le maintien de la cohérence dans l'ensemble des copies. Son rôle est de contrôler les écritures et d'assurer des observations cohérentes sur chaque copies. Cependant, dans la pratique, faute de mécanismes adaptés, l'utilisateur opte pour des solutions *adhoc*, en fusionnant les mises-à-jour manuellement et en essayant d'éviter les conflits (mises-à-jour sur une seule machine). Ce mode de fonctionnement est peu pratique, délicat et long, ce qui amène l'utilisateur à ne l'employer que pour des données indispensables.

Des techniques de replication et de synchronisation de données sont déjà intégrées dans la plupart des PDA. Ils proposent de petites applications (carnet d'adresses, agenda, ...) dont les données peuvent être synchronisées avec celles des applications de bureau équivalentes. Cependant, il s'agit là, de cas particuliers d'application, gérés par des protocoles propriétaires. Si la synchronisation des données n'est pas plus généralisée dans les applications, c'est en partie parce que la gestion de la cohérence nécessite des algorithmes et des protocoles adaptés aux types de données gérés. Malgré ce besoin, les développeurs d'applications ont actuellement peu d'outils à leur disposition pour y répondre.

1.2 Émergence de la collaboration asynchrone

Le développement des réseaux favorisent la collaboration entre les utilisateurs. Les formes de collaboration sont très variées mais elles impliquent tou-

jours un groupe de personnes partageant un intérêt commun. Cet intérêt fédérateur peut être professionnel (p. ex. les membres d'une cellule de veille technologique) ou personnel (p. ex. les passionnés d'aquariophilie). Ces groupes de personnes forment des communautés qui partagent certaines données sous différentes formes. Les modes les plus rudimentaires de collaboration (forums, listes de diffusion, canaux IRC) se limitent à des échanges de simples messages textuels. Dans les modèles plus élaborés, les utilisateurs exploitent concurremment des données en écriture et en lecture (p. ex. base de données des stocks). Si, jusqu'à présent, les applications collaboratives définissaient un domaine applicatif propre (forums, tableaux partagés, jeux, ...), on peut supposer qu'avec l'avènement d'Internet et l'émergence des communautés, tous les domaines applicatifs vont devoir intégrer la dimension collaborative.

Selon le type d'applications collaboratives, les contraintes sur les données partagées sont différentes. Les caractéristiques d'une application collaborative sont donc déterminées par le mode de collaboration entre les utilisateurs et l'infrastructure de communication. Si l'application nécessite de voir rapidement des mises-à-jour, on privilégiera une *collaboration synchrone*. Si au contraire, on autorise la collaboration entre utilisateurs distants et potentiellement déconnectés, on choisira une *collaboration asynchrone*. A titre d'exemples, voici les caractéristiques de quelques types d'applications collaboratives :

- Forum : les données échangées sont faiblement structurées (auteur, date, sujet, corps du message) et plus ou moins transitoires. Ces communautés sont de tailles variables et peuvent exister aussi bien sur des LANs que sur des WANs. Le mode de collaboration peut être synchrone (p. ex. les canaux IRC) ou asynchrone (p. ex. les groupes de discussion USENET) ;

- Jeux multi-joueurs : dans la plupart des jeux, les interactions entre les acteurs sont fortes. Les données sont modifiées fréquemment et ce type d'application nécessite généralement une forte synchronisation des réplicas (ou tout au moins d'une partie d'entre eux) ;

- Communautés professionnelles : les applications sont variées : bases de données (des produits, des stocks, des clients, ...), agendas de groupe, ingénierie coopérative (CAO), édition collaborative de documents, ...

Les applications collaboratives dépendent aussi fortement de la nature des données échangées. Les données sont-elles faiblement ou fortement structurées ? Sont-elles modifiables et destructibles ? Quel volume d'informations représente l'ensemble des données partagées ? Existe-t-il des relations structurelles ou sémantiques entre les données ? Quel critère de cohèrence requierent-elles ? Les écritures conflictuelles doivent-elles être proscrites ou sont-elles résolvables *a posteriori*. Toutes ces questions, combinées au mode d'interaction (synchrone ou asynchrone) déterminent des choix techniques qui doivent être fait pour la réplication d'applications collaboratives.

Un utilisateur qui partage ses données entre plusieurs machines est aussi une forme particulière de collaboration. Bien que, dans ce cas précis, la communauté soit réduite à un seul membre, on rencontre une partie des problèmes liés à la collaboration. Toutes les machines de l'utilisateur n'étant pas joignables en permanence, le mode d'interaction est asynchrone. Les conflits sont à priori moins fréquents (l'utilisateur a une connaissance globale des opérations d'écritures) mais ils peuvent aussi apparaître si l'utilisateur modifie concurrement une même donnée sur deux machines différentes. La problèmatique de la réplication pour les applications collaboratives asynchrones peut donc aussi s'appliquer à réplication dans les applications mono-utilisateurs.

1.3 Le Web : solution d'accès universel

Le Web a unifié l'accès en lecture des ressources. Les URLs [BLMM94] fournissent le moyen de localiser l'information, le protocole HTTP [FGM+97] spécifie la façon de les obtenir, et les *mimetype* [BF93] permettent de connaître le type de la ressource afin de l'interpréter convenablement. Enfin, le format HTML [RHJ98] permet de mettre en page simplement des informations multimédias et d'établir des liens hypertextes entre les pages. Cependant, ces différentes spécifications ne définissent pas de quelle manière l'information est produite.

Si, au début du Web, la majorité des informations accessibles étaient stockées dans des fichiers, ce mode de stockage s'est rapidement avéré trop limitatif pour fournir un contenu dynamique. Différents mécanismes de traitement du côté du serveur (CGI[4], Servlet, ASP[5], JSP[6]) ont alors été conçus pour pallier ce besoin. Ils produisent des pages dynamiques en fonction des données de chaque requête. On peut ainsi, offrir des vues d'une base de données ou l'état d'avancement d'un calcul. En combinant ces mécanismes avec l'extension de HTML pour construire des formulaires, on dispose d'une solution simple pour la consultation et l'écriture de données distantes.

Le Web, combiné avec les traitements côté serveur, permet la mise en place de solutions pour le partage collaboratif de certaines données. Néanmoins, la centralisation des données pour une application collaborative pose, en plus des difficultés évoquées précédemment, les problèmes de passage à l'échelle. En effet, si la communauté est importante, des problèmes de charge et de disponibilité du service peuvent survenir. Là aussi, la réplication des données au plus près des utilisateurs augmente la rapidité des accès. Dans le cas extrême, chaque utilisateur possède une copie des données et de l'application (un réplica). Cependant, il faut considérer un cas plus général, où la communauté est fragmentée en îlots possédant chacun un réplica. Ainsi, l'utilisateur se connecte au réplica lui offrant

[4] *Common Gateway Interface*
[5] *Application Server Page*
[6] *Java Server Page*

la meilleur performance. Par ailleurs, si son réplica habituel n'est pas disponible il peut alors travailler sur un autre réplica.

1.4 Caractéristiques des données

Les données que nous cherchons à répliquer sont mutables. C'est-à-dire qu'en plus de pouvoir en créer de nouvelles, l'application peut modifier ou détruire celles qui existent. Cette caractéristique a des conséquences importantes pour le modèle de réplication. En effet, les écritures ne sont pas toutes commutatives et doivent donc être exécutées *dans le même ordre* sur tous les réplicas pour garantir un état globalement cohérent. Par exemple, considérons deux réplicas qui changent concurremment la valeur d'une donnée puis diffusent chacun leur modification. Pour garantir la cohérence à terme des observations, le système doit convenir d'un ordre pour l'exécution ces deux écritures tous les réplicas doivent le suivre. Ceci impose donc que le système de réplication dispose d'un mécanisme d'ordonnancement des écritures qui sera utilisé lors de la réception de nouvelles écritures (i.e. lors de la synchronisation de deux réplicas).

Les langages orientés objets se sont popularisés et de plus en plus d'applications sont construites avec cette approche. Le système de réplication doit tenir compte de ce modèle de programmation qui influe directement sur la nature des données. Les objets sont structurés, typés et organisés selon un graphe de référencement. Ceci a des conséquences directes sur la synchronisation des réplicas. D'une part, la structuration devrait permettre des mises-à-jour incrémentales (il est inutile de diffuser les attributs qui n'ont pas été modifiés). D'autre part, la réplication d'un graphe d'objets mutables est une source de conflits et de violations d'intégrité. Par exemple, il faut faire face à des situations telles que la destruction d'un objet sur un réplica R_1, référencé par un objet nouvellement créé sur un réplica R_2. De même, si on n'y prend pas garde, la synchronisation peut amener tous les réplicas dans un état globalement cohérent mais qui violent l'intégrité sémantique des données (p. ex. un arbre transformé en graphe). Ceci impose d'introduire la notion de référencement au sein du mécanisme de gestion des conflits. Il existe plusieurs modèles objets. Nous avons retenu celui de Java mais les mécanismes de réplication peuvent aisément être transposés à d'autres modèles objets (p. ex. C++ ou Objective CAML).

La taille des données gérées par une application collaborative est variable. Cependant, lorsque les données élémentaires sont représentées sous forme d'objets, leur taille est généralement peu importante (quelques Ko). Néanmoins, si la taille d'un objet est relativement petite, l'application peut gérer d'importantes collections d'objets. Les objets gérés dans les applications que nous visons occupent en moyenne 1 Ko d'espace mémoire. Les applications les plus actives peuvent gérer jusqu'à plusieurs dizaines de milliers d'objets de cette nature. À titre d'exemple, ces caractéristiques ont été constatées sur les applications que nous avons mis en œuvre (cf. section 2, p. 13) mais aussi sur des applications de

CAO collaboratives [FSB+98]. Pour offrir de bonnes performances, le système de réplication doit donc assurer l'incrémentalité des synchronisations afin de n'envoyer à un réplica que les nouveautés depuis la dernière synchronisation.

Les applications collaboratives asynchrones peuvent avoir des durées d'exploitation longue (p. ex. plusieurs années). Elles gèrent donc des données qui doivent perdurer d'une exécution à l'autre. Pour cela, l'application intègre un mécanisme assurant leur persistance. Ce mécanisme doit être compatible avec les caractéristiques que nous venons de présenter. En particulier, il doit fournir les informations sur les nouvelles écritures des objets. Une base de données traditionnelle ne peut donc pas suffire car elle ne connaît généralement que la dernière valeur des données. La persistance doit aussi gérer correctement le graphe des objets. Ce graphe est construit avec un système de référencement propre au langage de programmation (p. ex. les pointeurs ou les références). Pour minimiser la tâche du développeur, le graphe des objets doit pouvoir être enregistré sur disque de façon incrémentale au cours de l'exécution pour être reconstruit au prochain démarrage de l'application.

Enfin, une application collaborative évolue au cours du temps. Elle subit des corrections et la structure des données persistantes peut évoluer (des attributs disparaissent et d'autres apparaissent). Comme dans tout système distribué, la propagation des évolutions de version se fait généralement de façon asynchrone. Le système doit néanmoins continuer à fonctionner après des évolutions sur la structure des données et les réplicas ne doivent pas être partitionnés selon la version de l'application qu'ils possèdent.

1.5 Critère de cohérence

Le critère de cohérence définit ce à quoi peuvent s'attendre les utilisateurs d'une application répliquée. Il caractérise essentiellement la fraîcheur des données qu'ils perçoivent. On peut classer les protocoles de réplication en deux grandes catégories : ceux qui garantissent la cohérence des observations à tout moment et sur n'importe lequel des réplicas (cohérence forte), et ceux qui font des compromis entre cette garantie et les contraintes de communication (cohérence faible). Le modèle de cohérence forte nécessite beaucoup d'échanges de messages pour maintenir chaque donnée à la même valeur sur tous les réplicas. Cette garantie n'est généralement apportée que par des protocoles de verrouillage [BN97]. Ceux-ci ne sont donc pas compatibles avec notre modèle d'exploitation où les lectures et les écritures peuvent avoir lieu sur des réplicas déconnectés. On est donc obligé de relâcher le critère de cohérence et d'autoriser des divergences d'observations.

Dans un contexte de données mutables, les protocoles de réplication à cohérence faible peuvent eux aussi être scindés en deux grandes catégories : les protocoles pessimistes et les protocoles optimistes. Les premiers se prémunissent à l'avance des problèmes liés aux concurrences d'écritures qui pourraient sur-

venir sur les réplicas. Néanmoins, ils ne cherchent pas à assurer la cohérence immédiate des observations. Les incohérences constatées disparaîtront progressivement, avec l'intégration des nouvelles mises-à-jour. Cette garantie nécessite la mise en place d'un contrôle des écritures qui fixe des contraintes importantes sur la communication entre les réplicas.

Le protocole dit de *copie primaire*[7] [AD76] assure la cohérence des données en centralisant toutes les écritures sur un réplica particulier. Celles-ci sont ensuite propagées sur les réplicas secondaires[8]. Ce protocole a l'avantage d'être simple à implémenter et à administrer mais nécessite une connexion à la copie primaire à chaque écriture.

Les protocoles à base de quorum [HHB96] relâchent le principe de la copie primaire en permettant les écritures et les lectures sur n'importe quel réplica à condition qu'un certain nombre d'autres réplicas (le quorum) y participent. De même que pour la copie primaire, ce protocole n'est pas compatible avec les spécificités des applications collaboratives que nous adressons. D'une manière générale, les protocoles pessimistes doivent anticiper pour éviter l'apparition de divergences dans les écritures. Ils ne sont donc pas adaptés aux écritures concurrentes en environnement déconnecté.

Les protocoles de verrouillage à grain fin[9] peuvent aussi être utilisés dans l'approche pessimiste. Ils garantissent qu'une donnée ne sera accédée en écriture que par un seul réplica à la fois. Pour cela, ils intègrent un gestionnaire de verrous qui assure la pose et la libération des verrous sur les données et prévient l'apparition de verrous mortels[10]. L'utilisation des verrous à grain fin pour des applications collaboratives dans un environnement déconnecté impose que les utilisateurs puissent identifier à l'avance l'ensemble des données sur lesquelles porteront leurs écritures. Ce mécanisme est donc intéressant lorsque le rôle de chaque utilisateur est connu à l'avance. Par exemple, dans une application d'édition collaborative, les auteurs peuvent se répartir les sections sur lesquels ils travailleront entre chaque synchronisation.

Les protocoles pessimistes ont la propriété de pouvoir assurer des invariants forts sur les données. Néanmoins, il existe des applications pour lesquelles l'accessibilité en écriture est prioritaire. Par exemple, dans le cas d'un agenda de groupe répliqué, il faut autoriser la prise de rendez-vous par tous les participants, au risque de voir apparaître des conflits une fois que les agendas seront resynchronisés. Lorsque cela survient, des procédures de résolution *ad hoc* doivent être appliquées pour corriger les rendez-vous conflictuels. D'une manière générale, pour ce genre d'applications collaboratives, il faut envisager un autre type de protocole en acceptant *a priori* toutes les écritures et en repoussant le contrôle de concurrence *a posteriori*. Dans ce modèle, les nouvelles écritures sont

[7] *Primary copy*
[8] *Backup copy*
[9] *Fine grain lock*
[10] *Dead lock*

provisoirement instables et peuvent être remises en cause ultérieurement.

Les protocoles optimistes s'inscrivent dans cette approche en relâchant les contraintes sur les concurrences d'accès en écriture pour offrir de meilleures performances et augmenter la disponibilité des données. En contrepartie, ils acceptent consciemment des concurrences d'accès pouvant amener à des écritures conflictuelles. Deux écritures sont conflictuelles si leur intégration sur deux réplicas nécessitent un traitement particulier pour assurer que les réplicas convergeront vers un état globalement cohérent à terme. L'utilisation d'un protocole optimiste suppose que les conflits soient peu fréquents et qu'il soit moins coûteux de les traiter lorsqu'ils surviennent plutôt que de les éviter. Les protocoles optimistes doivent donc être accompagnés d'un mécanisme de détection de ces conflits lors des synchronisations et de procédures capables de corriger *a posteriori* les conflits, tout en préservant le critère de cohérence à terme.

Dans notre cas, le critère de cohérence le mieux adapté est celui qui assure la *cohérence à terme*[11]. Dans ce mode de fonctionnement, l'état globalement cohérent n'est atteint que lorsque toutes les écritures ont été reçues par tous les réplicas. En exploitation, ce cas est rarement atteint et le système est en perpétuelle instabilité. C'est néanmoins le prix à payer pour autoriser les écritures concurrentes sur des données mutables en déconnecté. Le protocole de réplication pour les contraintes que nous visons doit donc être construit sur un modèle optimiste et intégrer des traitements adaptés pour la gestion des conflits.

1.6 Conclusions

1.6.1 Réplication des applications collaboratives asynchrones

Dans cette thèse, nous nous intéressons à la conception d'un *framework* de réplication optimiste pour les applications collaboratives asynchrones. La réplication a pour principal objectif d'offrir une haute disponibilité en lecture et en écriture dans des environnements ponctuellement connectés. Cependant ce domaine est vaste. Aussi la conception d'un protocole de réplication universel, c'est-à-dire pouvant répondre à toutes les applications collaboratives, est une tâche extrêmement difficile, voire impossible. Nous avons donc fait le choix de n'adresser que les applications collaboratives asynchrones ayant les propriétés suivantes :

- L'application gère des collections de données persistantes ;
- Les données sont structurées, de petites tailles (quelques Ko par donnée), modifiables et destructibles ;
- Une cohérence à terme est suffisante ;
- La sémantique de l'application supporte que les écritures sur les données soient provisoirement instables ;
- L'application peut faire face à des divergences d'écritures qui seront résolues *a posteriori*.

[11] *Eventual consistency.*

1.6.2 Protocole de synchronisation adapté au déconnecté

La réplication des données mutables pour les applications collaboratives soulève de nombreux problèmes. Le principal d'entre-eux est d'assurer la cohérence des observations sur les différents réplicas. Pour cela, il faut définir un protocole de synchronisation adapté au fonctionnement en déconnecté. Ceci implique que les synchronisations engagent un nombre minimal de participants : le réplica à synchroniser et le réplica partenaire pour la synchronisation. Pour réduire le nombre de synchronisations, le protocole doit assurer la propagation épidémique des écritures. Cela nécessite que les réplicas échangent aussi les écritures qu'ils ont reçues des autres réplicas depuis leur dernière synchronisation. Cependant, les applications que nous visons ont des durées d'exploitation longues (plusieurs années), qui peuvent entraîner la constitution de données volumineuses. Le protocole de synchronisation doit prendre en compte cette dimension en assurant des échanges incrémentaux et en permettant la réplication partielle des données.

1.6.3 Gestion des conflits

Les données répliquées sont mutables. Les écritures parallèles sur les réplicas peuvent donc amener à des conflits lors des synchronisations. Il faut être en mesure de détecter ces conflits, de les caractériser et de trouver une procédure de résolution adaptée à chacun d'entre eux. La caractérisation des conflits et les procédures sont liées à la nature des données et au mode de fonctionnement de l'application. On doit donc fournir aux développeurs d'applications le moyen d'identifier les conflits qui peuvent survenir et les procédures à exécuter. Cependant, quelles que soient les applications, il existe des conflis récurrents (p. ex. deux mises-à-jour concurrentes de la même donnée) pour lesquels il est souhaitable de décharger le développeur de leur gestion en fournissant des mécanismes de détection et des procédures de résolution pré-définies pour les applications que nous visons.

1.6.4 Journalisation des écritures

Pour une gestion efficace des conflits lors des synchronisations, on a besoin de connaître les écritures qui ont fait diverger les deux réplicas. L'état courant des données ne peut pas suffire. Par exemple, si une donnée est absente d'un réplica, des informations supplémentaires sont nécessaires pour savoir si ce réplica ne l'avait encore jamais reçue ou s'il l'avait déjà reçue mais détruite. La réplication a donc des conséquences directes sur le mécanisme de persistance des données. Il faut être en mesure de journaliser toutes les écritures qui ont fait diverger les réplicas, pour les analyser et les fusionner lors des synchronisations.

1.6.5 Mise-à-jour de l'application

Enfin, la réplication des données pose le problème de la réplication des versions des applications. Comment garantir que tous les réplicas possèdent une version de l'application cohérente avec les données qu'ils répliquent ? Dans un système distribué, les mises-à-jour de l'application sont souvent asynchrones. Il faut donc permettre un fonctionnement avec des réplicas ayant chacun des versions différentes de l'application. Ceci n'est pas sans incidence sur la persistance des données (évolution de structure) ni sur leur cohérence (évolution des algorithmes). La diffusion des nouvelles versions de l'application doit aussi être prise en compte par un protocole approprié. Cependant, les contraintes sont différentes de celles liées à la synchronisation des données. En particulier, les mises-à-jour de l'application sont généralement moins fréquentes et centralisées. Il s'agit donc d'un protocole totalement différent de celui qui gère les données.

Chapitre 2

Choix d'applications collaboratives répliquées

L'élaboration d'un protocole de réplication pour les applications collaboratives doit être instancié sur des cas concrets. La confrontation avec des applications réelles permet de mettre en avant des problèmes fondamentaux et pratiques pour l'utilisation du protocole. D'autre part, ces applications nous permettront d'illustrer les mécanismes présents par des exemples concrets.

Dans cette section, nous présentons deux applications collaboratives asynchrones. La première est un carnet d'adresses qui illustre le cas d'un utilisateur partageant ce type d'information sur plusieurs machines. La seconde, Pharos, est une application beaucoup plus conséquente qui propose un service d'évaluation collaborative.

2.1 Réplication d'un carnet d'adresses

Le carnet d'adresses électronique est une application bureautique très utilisée. Le succès du courrier électronique a renforcé les besoins de gérer des informations sur les correspondants. La plupart des outils de courriers électroniques intègrent un carnet d'adresses. C'est aussi le cas des PDA. La figure ci-dessous est une capture d'écran de l'application de carnet d'adresses que nous avons développé pour valider les concepts.

Les données gérées dans un carnet d'adresses sont simples. Il s'agit d'enregistrements structurés contenant plusieurs attributs. La déclaration Java de la classes ci-dessous représente ce type de données :

```
public class Card {
  String name;
  String email;
  String homeTel;
  String workTel;
  String mobile;
  String fax;
```

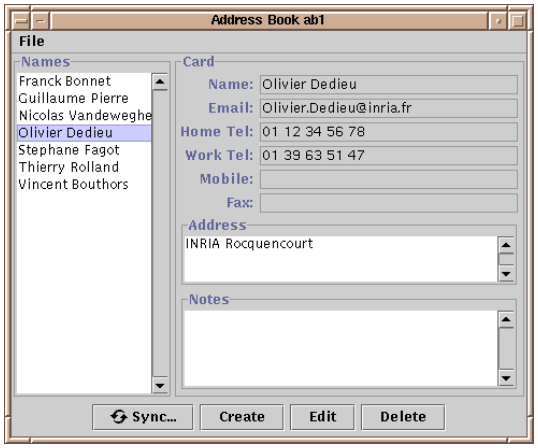


FIG. 2.1 – *Capture d'écran du carnet d'adresses*

```
  String address;
  String notes;
}
```

Les contraintes d'intégrité sur les données sont inexistantes. Les doublons d'adresses sont autorisés. Ceci simplifie donc la gestion des conflits lors des synchronisations d'adresses. Les seuls conflits pouvant donc survenir portent sur une même donnée et sont soit deux mises-à-jour concurrentes, soit une destruction et une mises-à-jour concurrentes.

2.2 Pharos

La quantité d'informations disponibles sur le Web est en constant accroissement. En janvier 2000, le volume du Web était estimé à un milliard de pages. Face à cette masse de données, la recherche d'informations précises et pertinentes est un problème auquel font face régulièrement les utilisateurs. Pharos[BD99] est un service développé pour aider les utilisateurs à organiser cette masse d'informations en partageant la connaissance qu'ils en ont. Pharos repose sur une infrastructure de collaboration qui permet à des groupes d'utilisateurs de cataloguer et d'évaluer des documents sur un sujet donné. Ces données, qui peuvent être subjectives, sont synthétisées afin de produire des recommandations personnalisées. Le passage à l'échelle du système est assuré par la distribution des serveurs et la réplication de leur bases de données.

2.2.1 Filtrage collaboratif

La multiplication des données disponibles par le Web tend à rendre inutilisables les moteurs de recherches par indexation automatique. Ceux-ci utilisent des agents qui parcourent le Web à la recherche de nouvelles pages et indexent leur contenu. Ces moteurs de recherche offrent des interfaces Web pour retrouver

des pages à partir de mots qu'elles contiennent. Il est donc difficile de formuler une requête de recherche pour obtenir des résultats précis. De plus, le classement des résultats qu'ils retournent n'est pas toujours pertinent. Il est donc bien souvent nécessaire de consulter chacun des documents proposés pour évaluer leur intérêt.

Les systèmes de filtrage collaboratif construisent des vues filtrées des informations disponibles pour une communauté. Le filtrage est personnalisé pour l'utilisateur. La personnalisation est calculée en déterminant des profils d'intérêt similaires dans la communauté. Appliqué au Web, le filtrage collaboratif permet donc de naviguer dans des informations thématiques, filtrées et personnalisées pour l'utilisateur.

2.2.2 Annotations, recommandations et canaux

Pharos est un système de filtrage collaboratif pour le Web. Il s'agit d'un service qui permet à des personnes de partager leurs connaissances sur un sujet donné. Pour cela, les utilisateurs sont invités à donner leur avis sur les documents qu'ils consultent en posant une *annotation* dans le canal approprié. Une annotation est une donnée structurée qui référence une URL. La structure est variable d'un canal à l'autre ; néanmoins on retrouve typiquement un titre, une note d'intérêt, un commentaire personnel et une liste de termes (cf. figure 2.2). Un canal est une base de données d'annotations concernant un sujet particulier (p. ex. la programmation Java, l'aquariophilie, l'enseignement des mathématiques, ...). Les annotations sont posées par les membres du canal. Plusieurs annotations peuvent concerner le même document mais un même membre ne peut poser qu'une seule annotation sur un document donné. Les annotations sont mutables : un membre peut revenir sur une précédente annotation pour la modifier ou la détruire.


4. Clifford Neuman, Scale in Distributed Systems ☆☆☆
Un excellent papier sur 3 techniques de passage a l'echelle: la distribution, la replication, la mise en cache. Le papier est un peu vieux (1994) et ne donne pas de details sur les travaux recents, en particulier sur la replication optim...
évalué par Olivier Dedieu – [Voir le détail des avis] – [Editez votre avis]
url : http://www.isi.edu/people/bcn/papers/pdf/94--_scale-dist-sys-neuman-readings-dcs.pdf
mots-clés : Groupe de recherche , Article / Rapport technique , Cohérence , Large Echelle , Cache , Replication

FIG. 2.2 – *Exemple d'annotation*

Les utilisateurs interagissent avec leur canal Pharos grâce à un *assistant de navigation*. Ce dernier leur permet de connaître les annotations des autres membres du canal sur les documents qu'il consulte. Pour chaque page consultée, l'assistant fait une requête au canal et affiche les annotations disponibles sur cette page (cf. figure 2.3). L'utilisateur peut ainsi rapidement déterminer le thème et l'intérêt du document consulté. L'assistant fournit aussi une interface de recherche. Celle-ci permet de rechercher des documents annotés selon

différents critères (note d'intérêt, membre, expression régulière, ...). Le canal se comporte donc aussi comme un moteur de recherche mais les résultats retournés sont tous liés à la thématique du canal et ils sont classés et évalués par d'autres utilisateurs.

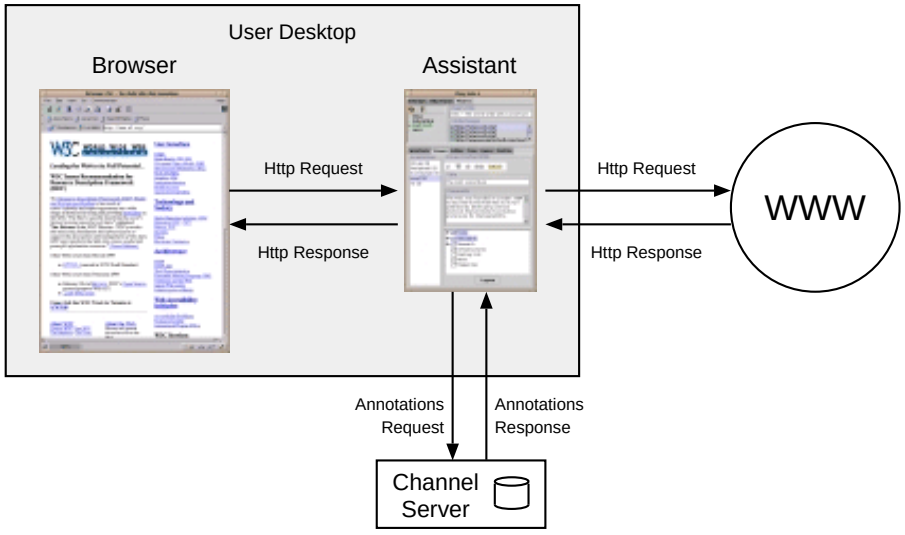


FIG. 2.3 – *Interactions entre l'assistant, le navigateur et les canaux Pharos*

Outre l'accès par l'assistant de navigation, les canaux offrent aussi un accès Web. Celui-ci a l'avantage d'être plus simple d'emploi pour les utilisateurs (il ne requiert qu'un navigateur). C'est aussi un moyen simple d'externaliser le travail de la communauté : des utilisateurs étrangers à la communauté peuvent consulter les recommandations (si l'accès est autorisé).

Pour classifier les documents annotés, les membres du canal gère un thésaurus. Il s'agit d'une structure arborescente de termes (cf. figure 2.4). Ce vocabulaire commun de classification est géré dynamiquement par les membres autorisés. Une annotation peut être classée sous plusieurs termes. La classification a un double rôle : elle permet de connaître rapidement le sujet d'un document et elle sert à rechercher les documents annotés selon un ou plusieurs termes.

Lorsque le nombre d'annotations devient important la consultation individuelle de chaque annotation devient coûteuse, en particulier lorsqu'il y a de nombreux avis sur un même document. Pour réduire le bruit, le canal calcule des synthèses d'annotations. L'algorithme de synthèse regroupe toutes les annotations disponibles sur un document et produit une *recommandation*. Comme tous les membres n'ont pas forcément le même avis sur l'intérêt de ce document, les recommandations sont personnalisées selon le profil de chacun.

FIG. 2.4 – *Exemple de thésaurus*

2.2.3 Niveau de collaboration

Au sein d'un canal Pharos, les membres collaborent sur trois types de données : les annotations, les termes du thésaurus et les informations sur les membres. Chaque membre possède des droits d'accès qui déterminent la portée de ses actions : éditer ses annotations et ses informations, éditer le thésaurus et administrer. Un membre ne peut pas éditer l'annotation d'un autre membre ni ses informations, sauf s'il a les droits d'administration.

Les termes du thésaurus sont les données les plus soumises à des écritures concurrentes. En effet, le système de droits réduit les écritures concurrentes sur les annotations et les informations personnelles (on suppose que peu de membres ont les droits d'administration d'un canal). Par contre, l'édition du thésaurus peut être offerte à un grand nombre de membres. Ceux-ci peuvent donc créer des nouveaux termes, en détruire certains, les déplacer et changer leur attributs. Il faut aussi assurer l'intégrité du thésaurus en interdisant l'apparition de cycles de référencement entre termes.

2.2.4 Architecture

Dans sa version centralisée, Pharos repose sur une architecture client/serveur. Un serveur Pharos peut héberger plusieurs canaux. Chaque canal est indépendant et possède sa propre base de données (annotations, termes et membres). L'agrégation de canaux dans un même serveur Pharos permet de mutualiser des ressources communes (serveur Web, service d'authentification, ressources pour l'interface Homme-machine, ...). Les canaux reposent eux aussi sur un modèle client/serveur : la partie serveur (le *backend*) se situe dans le serveur Pharos et la partie client (le *frontend*) est hébergée dans l'assistant de navigation.

Pour réduire les échanges réseaux entre le serveur et l'assistant, le client d'un canal met en cache les collections de données fréquemment utilisées : le thésaurus et la liste des membres. Ces données subissent relativement peu de

modifications en comparaison des annotations ; néanmoins lorsque cela arrive le serveur du canal doit les propager aux clients.

Plusieurs serveurs Pharos peuvent fonctionner en hébergeant chacun des canaux qui leur sont propres. Par exemple, un serveur Pharos peut être installé dans une entreprise et gérer les différents canaux de l'entreprise. Cependant, pour améliorer la qualité de service, un canal peut être répliqué dans différents serveur Pharos afin d'améliorer la disponibilité du service. En effet, les annotations constituent en soit une information à forte valeur ajoutées. Il est donc important de rendre accessibles ces données à tout moment. En les répliquant au plus proches des membres de la communauté, on peut garantir cette qualité de service. Les membres ne sont plus pénalisés par des accès distants peu performants (en particulier si ils sont connectés par modem) et l'arrêt momentané d'un serveur ne bloque plus totalement la communauté. La figure 2.5 illustre l'architecture de réplication utilisée dans Pharos.

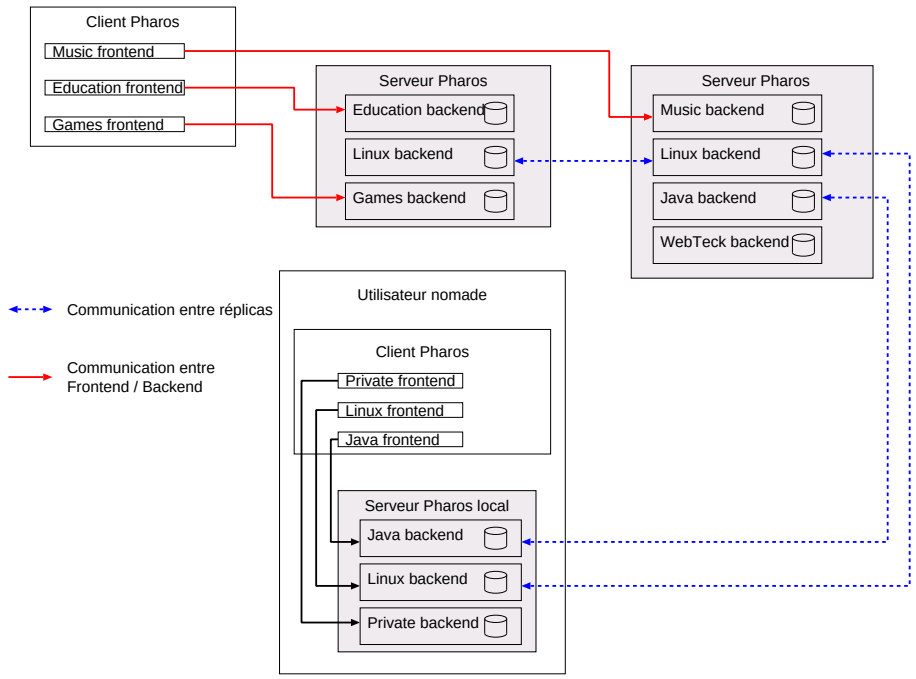


FIG. 2.5 – *Architecture de réplication dans Pharos*

La réplication d'un canal permet aussi d'augmenter la réactivité de l'assistant de navigation et de réduire la charge d'un serveur Pharos. Les assistants font une requête au canal courant à chaque fois que l'utilisateur change de page pour connaître les annotations disponibles. Si de nombreux assistants font des requêtes simultanément sur le même canal, la charge peut augmenter de façon notable. Il est donc pertinent de proposer la réplication du canal sur le poste du client (ou sur une machine proche) et ainsi de réduire la consommation de bande passante.

Enfin, la réplication a aussi un impact psychologique. Les utilisateurs peuvent être réticents à annoter dans un canal. Il s'agit d'un effort non négligeable (p. ex. en comparaison de la pose d'un signet dans le navigateur) et ce travail pour la communauté n'est pas toujours un acte immédiat. En permettant à chaque membre de répliquer sur sa machine tout ou partie de la base des annotations, on augmente le sentiment de possession et l'utilisateur est rassuré sur la pérennité des annotations qu'il enregistre : même si le service disparaît, son réplica continuera d'être opérationnel.

2.3 Conclusions

Les deux applications que nous avons présentées recouvrent des problèmes variés, liés à la réplication des applications collaboratives asynchrones. Le carnet d'adresses répliqué pose les questions sur la réplication d'une application mono-utilisateur sur plusieurs machines. Dans ce cas, on cherchera essentiellement à simplifier la synchronisation des réplicas mais on aura rarement à faire face à des conflits. Par contre, dans le cas de la seconde application, Pharos, les sources de conflits sont plus nombreuses. Cela tient à la nature des données, à la dimension de l'application et au mode de collaboration. Ces deux applications, nous servirons dans le reste de la thèse à illustrer les différents mécanismes mis en œuvre.

Chapitre 3

État de l'art

La conception d'un système de réplication de données dépend de plusieurs paramètres. Les choix que l'on fait sur ces paramètres influent fortement sur les fonctionnalités et les propriétés du système. Pour l'étude des systèmes existants, nous nous intéressons aux protocoles de réplication optimistes adaptés à la synchronisations de données mutables dans un environnement déconnecté. Ces protocoles sont généralement accompagnés d'un gestionnaire capable de détecter des écritures conflictuelles et de les résoudre. Chacun de ces gestionnaires se distingue par sa capacité à faire face à des types de conflits variés selon les applications. Enfin, pour ceux qui le permettent, nous examinons le support de réplication partielle et la gestion des évolutions de l'application.

3.1 USENET

USENET est l'architecture sur laquelle reposent les groupes de discussion (*newsgroup*). NNTP (*Network News Transfer Protocol*) [KL86] est le protocole de communication utilisé entre les serveurs USENET. C'est un protocole client/serveur dans lequel chaque serveur joue alternativement le rôle du client ou du serveur. La communication se fait par échange de messages USENET [HA87]. Les données manipulées sont les groupes et les articles. Un article est diffusé dans une liste de groupes. Les messages USENET manipulent ces deux types de données.

Le service USENET est organisé selon un graphe orienté où chaque nœud est une machine et chaque arc est un chemin de transmission des messages. Chaque arc est étiqueté par les groupes auxquels sont envoyés les messages qui y transitent. Généralement les arcs sont bidirectionnels et les deux directions sont étiquetées de la même façon. Grâce à cette topologie, un serveur peut recevoir le même message de plusieurs autres serveurs. Le protocole assure l'incrémentalité des envois entre les serveurs. Pour cela, il utilise un mécanisme d'historique qui garde trace des messages déjà reçus. Lors d'un nouvel échange, le client indique au serveur les messages qu'il a déjà reçus.

USENET est une architecture fortement répliquée avec un modèle de cohé-

rence à terme. Chaque serveur possède sa propre base de données des messages reçus. Les groupes et les articles ne sont pas mutables (ils sont cependant destructibles) et il n'existe pas de liens entre ces objets. Le protocole de synchronisation n'est donc ni optimiste ni perssimiste car il n'a jamais à faire face à des écritures conflictuelles.

L'architecture distribuée de USENET est très intéressante car il s'agit d'un service répliqué, opérationnel depuis longtemps et utilisé à très large échelle. La réplication a justement permis de faire face à la multiplication des groupes de discussion et des volumes de données échangées (en particulier avec l'échange d'images, de sons ou de programmes). Le modèle de réplication par envoi de message est pertinent car les messages permettent d'envoyer des *opérations* sur les données. Si dans USENET, ces opérations ne concernent que des créations, on peut généraliser le concept pour envoyer des opérations de destruction et de modification.

L'estampillage des messages est aussi une solution intéressante pour assurer l'incrémentalité des échanges. Le principal manque à ce modèle est de permettre les modifications de données et la gestion des référencements. C'est justement l'impossibilité de ce type d'opération qui permet au système de ne pas avoir à traiter de conflits. Cependant, pour une utilisation dans une application collaborative, il est nécessaire de fournir ces fonctionnalités.

3.2 Systèmes de fichiers répliqués

Une des premières applications de la réplication, en dehors des bases de données, a concerné les systèmes de fichiers. Comme pour les bases de données, les premiers protocoles proposés ont plutôt retenu l'approche pessimiste. Cependant, le développement de l'informatique nomade dans ce domaine a aussi fait apparaître le besoin d'assurer une continuité de service durant les périodes déconnectées.

La réplication d'un système de fichiers se fait à deux niveaux : la structure du système de fichiers et les fichiers eux-mêmes. Les fichiers étant de nature très variée (texte, texte formaté, code source, binaire...), leur synchronisation est généralement supportée par des extensions au protocole (une extension par type de fichier). La réplication de la structure est plus proche de nos préoccupations. En effet, elle contient la représentation de l'arborescence du système de fichier. Ce type de données correspond aux contraintes fixées sur les données à répliquer (cf. section 1.4, p. 7). Il est donc intéressant d'étudier comment les systèmes de fichiers pour l'informatique nomade gèrent la synchronisation de leur structure.

3.2.1 Coda

Coda [KS93] est un système de fichiers destiné principalement aux ordinateurs portables. Ce projet de recherche achevé est devenu un produit opération-

nel distribué en standard avec Linux. Coda apparaît pour l'utilisateur comme un système de fichiers Unix partagé traditionnel. Chaque client Coda dispose de son propre disque et est ponctuellement connecté à l'un des serveurs. La disponibilité du système est assuré par deux mécanismes : (i) les serveurs sont répliqués, (ii) les clients peuvent continuer à travailler une fois déconnectés. Dans Coda, les deux niveaux de réplication sont clairement séparés : les serveurs sont les réplicas de *première classe* et les clients, les réplicas de *seconde classe*.

Lorsqu'un client est connecté, la stratégie de réplication est un dérivé du modèle *read-one, write-all*. Il lit sur un serveur (le serveur préféré) mais fait ses écritures sur un groupe de serveurs disponibles (AVSG)[1]. Pour augmenter les performances, le client gère un cache. La cohérence des caches est assurée par un mécanisme d'invalidation par *callback*. La propagation des écritures sur un fichier ou sur un répertoire est laissée à la charge des clients. Pour faire face, à la latence de ces synchronisations, Coda propose un mécanisme de RPC[2] parallèle [SS90].

Lorsqu'un client est déconnecté, il devient autonome. Il ne peut agir que sur les fichiers présents dans son cache. Une fois reconnecté, il propage ses modifications au AVSG. Les modifications concernent à la fois les fichiers mais aussi leurs répertoires. Coda impose des limites sur l'utilisation des liens sur fichier pour garantir que les répertoires sont des arbres (et non des graphes acycliques). Toutes les écritures sont journalisées avec la RVM[3] [SMK+94]. Lorsqu'un client se reconnecte, Coda fusionne les écritures du client avec celles du serveur. Le protocole rétablit un état commun entre le serveur et le client, puis fusionne les écritures divergentes. Coda détecte 4 types de conflits sur l'arboresence :

- conflit de nommage : deux fichiers du même répertoire ont le même nom ;
- conflit de suppression/modification : un fichier/répertoire est supprimé d'un répertoire sur un réplica et mis-à-jour sur un autre réplica ;
- conflit de modifications : les données d'un répertoire (p. ex. la liste des fichiers) ont été modifiées par deux réplicas ;
- conflit de déplacement : un fichier/répertoire est déplacé (ou renommé) différemment sur deux réplicas.

Les conflits de mises-à-jour sont résolus par des traitements adaptés à la sémantique des fichiers. Lorsqu'un autre type de conflit apparaît, Coda le reporte sous forme d'une erreur qui doit être résolue manuellement.

Le modèle de réplication de Coda est très lié au domaine des systèmes de fichiers. Il y a deux niveaux de réplication et la topologie de synchronisation est imposée. Un client doit toujours se synchroniser avec un groupe de serveurs.

[1] *Available Volum Stoage Group*
[2] *Remote Procedure Call*
[3] *Recoverable Virtual Memory*

Le protocole n'autorise pas de synchronisation entre deux clients. La structure arborescente est un type de données assez générique. Cependant, tous les conflits ne sont pas détectés. Par exemple, Coda autorise les déplacements symétriques d'un répertoire D_1 dans D_2 (et D_2 dans D_1) qui introduisent des cycles et les rendent depuis la racine. D'autre part, Coda fait l'hypothèse que les conflits sur la structure sont rares (des mesures le confirment), ce qui lui permet de les traiter comme des erreurs.

3.2.2 Ficus

Ficus est un système de fichiers répliqués pour des machines ponctuellement connectées. Il se différencie de Coda sur plusieurs points. Tout d'abord, il ne fait pas de distinction entre les réplicas. Un réplica peut se synchroniser avec n'importe quel autre réplica. D'autre part, Ficus ne gère pas de journal des écritures. Il utilise, comme dans son prédécesseur Locus [WPE+83], un vecteur de version par fichier. La synchronisation consiste donc à comparer les vecteurs de version de chaque fichier. Comme dans tout système de fichiers Unix, les répertoires sont aussi des fichiers et possèdent donc, eux aussi, un vecteur de version.

Lors d'une synchronisation, Ficus examine l'arborescence de fichiers des deux réplicas. Les opérations possibles sur les répertoires sont ramenées à l'ajout et à la suppression de fichiers et de répertoires[4]. Ficus gère trois des quatre types de conflits présentés dans Coda. Cependant, il propose une résolution automatiquement :

- conflit de nommage : les fichiers concernés sont renommés avec un suffixe unique ;
- conflit de suppression/modification : le fichier détruit est déplacé dans une arborescence spéciale (l'orphelinat) ;
- conflit de modification : comme dans Coda, Ficus déclenche un traitement selon le type du fichier (déterminé par une expression régulière sur le nom du fichier).

Ficus ne journalisant pas les écritures, il doit faire face à l'ambiguïté d'insertion/destruction. Ce cas arrive lorsqu'un fichier est présent sur le réplica R_1 mais pas sur R_2 : le fichier a-t-il été détruit sur R_2 ou a-t-il été nouvellement créé sur R_1 ? Comme pour Oracle (cf. section 3.3, p. 25), Ficus marque le fichier comme détruit mais ne le détruira physiquement que plus tard. Pour cela, il met en place un mécanisme de ramasse-miettes pour connaître le moment opportun de cette destruction.

Comme pour Coda, les choix pour la gestion des conflits sont liés à son domaine applicatif. Il semble difficile de les généraliser à d'autres types de données. La synchronisation par échanges des états courants sur les données ne permet pas de déterminer les écritures qui ont fait diverger deux réplicas et ne permet

[4]Les déplacements sont géré comme un ajout suivit d'une suppression.

pas de résoudre l'ambiguïté d'insertion/destruction.

3.3 Réplication dans les SGBDR

Les SGBDR[5] ont été parmi les premières applications à intégrer des mécanismes de réplication. Cependant, l'emploi de la réplication avait d'autres objectifs que ceux que nous visons. Ces systèmes étaient essentiellement mis en place pour assurer la répartition de charge et la tolérance aux pannes. Ce type de réplication a suscité de très nombreux travaux de recherche et plusieurs protocoles sont apparus. Néanmoins, ils ont été conçus avec des propriétés qui ne sont pas en accord avec les hypothèses que nous avons faites ; principalement sur l'exploitation en déconnecté.

Depuis quelques temps, la plupart des éditeurs de SGBDR intègrent un support pour la réplication optimiste à leurs produits. Ce besoin est né avec le déploiement d'Internet et pour deux cas typiques d'utilisation. Le premier d'entre-eux est apparu avec l'essor de l'informatique nomade. Les employés des entreprises (p. ex. les forces de ventes) sont de plus en plus équipés d'ordinateurs portables et doivent pouvoir exploiter certaines données (produits, commandes, clients, stock, ...) lors de leurs déplacements chez les clients. Ils ne disposent donc que de connexions ponctuelles et les débits sont faibles et non garantis. Le second type d'utilisation concerne les extranets et l'exploitation multi-sites. Dans ce cas d'utilisation, une entreprise doit pouvoir répliquer ses données au plus près de ses agences. Ces cadres d'utilisation étant similaires aux notres, les protocoles utilisés sont donc intéressants à étudier.

L'étude des protocoles de réplication des principaux SGBDR [Ora97, IBM99, Syb99, Inf98] montre qu'ils employent des techniques similaires. Ils offrent généralement deux modèles de réplication. Le premier repose sur un protocole de copie primaire supportant éventuellement la déconnexion (dans ce cas les copies secondaires ne sont pas forcément cohérentes, il ne peut pas y avoir d'écriture en déconnecté et un mécanisme incrémental les remet à jour une fois reconnecté). Le second modèle est une approche optimiste autorisant les mises-à-jour asynchrones sur différents réplicas. Cependant, les éditeurs de SGBDR recommandent d'utiliser ce type de réplication avec parcimonie et préconise l'emploi d'un site de référence pour résoudre les écritures conflictuelles.

La gestion des conflits est assez pauvre. Seuls les conflits portant sur les données de la même table sont détectables et résolvables (en particulier il n'y a pas de support pour les conflits entre les tables liées par une relation). Les modèles de réplication les plus avancés prévoient trois types de conflits : conflits de mises-à-jour, conflits d'unicité d'enregistrement et conflits de modification/destruction. Pour ces derniers, certains SGBDR comme Oracle [Ora97], ne peuvent les gérer que si les destructions ne sont pas effectives. L'enregistrement est alors marqué comme détruit mais ne le sera qu'après une phase explicite de purge. Ceci est

[5]Systèmes de Gestion de Base de Données Relationnelles.

aussi indispensable pour lever l'ambiguïté de création/destruction : si un enregistrement n'est présent que sur l'un des réplicas il faut déterminer s'il s'agit d'une création sur l'un ou d'une destruction sur l'autre. Ces problèmes ne se posent pas pour les SGBDR comme Informix [Inf98], qui gèrent un journal des écritures. Pour les conflits de mises-à-jour, la résolution peut se faire champ par champ tout en préservant des champs sémantiquement liés (p. ex. les champs ville et code postal). En cas de conflit, des routines de résolution par défaut sont proposées (priorité selon la date de dernière modification, selon le réplica, ...) et le développeur peut écrire ses propres routines. L'utilisation des dates pose toutefois des problèmes car les horloges des machines sont rarement synchronisées.

3.4 Réplication dans Lotus Notes

Lotus Notes [Tam97] est le représentant le plus connu des outils de *Groupware*. La réplication optimiste est au cœur même de Notes. Elle est relativement complète et concerne aussi bien les données que les schémas et les masques.

Les données traitées par Notes sont des enregistrements dans des tables. Les schémas définissent la structure des tables et les masques matérialisent les formulaires de saisie. Les types de données sont assez pauvres (entiers, flottants, texte, date, listes). Il est donc mal adapté aux application devant gérer des relations entre les tables (p. ex. le thésaurus d'un canal Pharos, qui est une structure arborescente, n'est pas représentable facilement avec Notes).

Les synchronisations se font entre deux serveurs ou entre un client et un serveur. La topologie et la fréquence des synchronisations est laissée au choix de l'administrateur. Une synchronisation est soit monodirectionnelle (*push* ou *pull*), soit bidirectionnelle (*pull/push*). Dans le premier cas, seul l'un des deux réplicas est mis-à-jour à partir des données du second. Dans le deuxième cas, les deux réplicas sont mis-à-jour. Un réplica peut définir un filtre pour ne répliquer qu'une partie des données. Ce filtre peut changer d'une synchronisation à l'autre.

Notes détecte les conflits de mise-à-jour et les conflits de modification/destruction. Pour cela, il gère des *talons de destruction* qui permettent de connaître les données qui ont été détruites. Depuis la version 4.5, Notes est capable de détecter les conflits au niveau des attributs. Il peut ainsi réaliser la fusion de deux mises-à-jour sur la même donnée. Si les attributs modifiés sont les mêmes ou si la fusion des mises-à-jour n'est pas activée, il y a un conflit. Ce type de conflit apparaît comme un enregistrement special qui contient l'un des deux enregistrements conflictuels. Il n'existe pas de moyen simple pour les résoudre automatiquement. La résolution est laissée à l'administrateur. Dans le cas des conflits de modification/destruction, la résolution est automatique et ne peut pas être changée. C'est toujours la destruction qui l'emporte sur la modification. Enfin, aucun support n'est fourni pour détecter des conflits entre deux données distinctes (p. ex. des conflits de doublon).

Le mode de réplication de Notes est bien conçu pour les applications de *groupware* qu'il vise : carnet d'adresses, *workflow*, forum de discussion... Le modèle de synchronisation est simple et riche (topologie et fréquence de synchronisation flexible, filtres de sélection, synchronisation au niveau des attributs). Cependant, ce n'est pas une librairie de réplication généraliste. En particulier, l'absence de pointeurs dans les types de données lui ferme un certain nombre d'applications. De plus, c'est un système propriétaire et fermé : le protocole de synchronisation n'est pas diffusé et la gestion des conflits ne peut pas être spécialisée.

3.5 Protocoles de réplication généralistes

De nombreux travaux de recherche ont été menés au sein de la communauté CSCW[6] pour fournir des bibliothèques de base pour la construction d'applications collaboratives. Cependant, la majorité de ces travaux se sont concentrés sur les applications collaboratives centralisées (p. ex. [SKSH96, PHRM90]). Quelques uns se sont intéressés à la réplication mais la plupart avec une approche pessimiste (p. ex. [KP90, CBF+90]). Gina et Bayou sont parmi ceux qui ont opté pour l'approche optimiste.

3.5.1 Gina

Gina [BG93] était initialement une bibliothèque pour construire des applications multi-utilisateurs disposant d'interfaces graphiques. Elle se caractérisait par la possibilité de défaire et refaire les actions de l'utilisateur. C'est cette possibilité qui a été utilisée pour étendre la bibliothèque en un système de réplication.

Dans Gina, toutes les commandes de l'utilisateur sont enregistrées dans un historique. Ces commandes sont représentées par deux méthodes LISP (*doit* et *undoit*) qui permettent respectivement d'amener l'application dans l'état suivant ou précédant l'état actuel. L'historique permet donc de rétablir l'application dans n'importe lequel de ses états antérieurs.

L'intégration de la réplication se fait en diffusant les commandes aux autres membres du groupes. Chaque commande est estampillée par une horloge logique. Gina prévoit une diffusion par *broadcast* dans le groupe des réplicas. Lorsqu'une commande est reçue, elle est intégrée dans l'historique du receveur. Comme le modèle d'écriture est optimiste, des conflits peuvent apparaître. Lorsqu'un conflit de mutation sur un objet est détecté, le système crée une nouvelle branche dans l'historique. L'historique est donc en réalité un arbre de versions. Les conflits sont résolus par l'utilisateur en sélectionnant et en fusionnant les branches qu'il souhaite. Pour l'aider dans cette tâche, Gina fournit un mécanisme qui permet à l'utilisateur de ne défaire que ses commandes (*selective*

[6] *Computer-Supported Cooperative Work*

undo). Les autres branches restent néanmoins dans l'historique mais sont marquées comme annulée. Chaque commande est ensuite rejouée sélectivement (*selective redo*).

Gina est un système intéressant car il fournit les informations nécessaires sur les mutations grâce à la journalisation des commandes. La journalisation des commandes d'annulation permet aussi de revenir rapidement sur un état récent. Cette approche est pertinente pour la réplication avec cohérence à terme. D'une part, bien qu'il n'y soit pas fait allusion dans leur publication, l'historique peut servir de système de persistance. En rejouant tout l'historique au démarrage de l'application, on la rétablit dans son dernier état. D'autre part, au cours de l'exploitation, la partie commune de l'historique devient de plus en plus grande et la partie divergente (i.e les dernières commandes) ne représente qu'une faible proportion. Il est intéressant de pouvoir défaire rapidement les dernières actions pour les fusionner avec celles diffusées par les autres réplicas.

La gestion de la réplication dans Gina est néanmoins incomplète pour les applications collaboratives asynchrones. Tout d'abord, la diffusion des nouvelles commandes se fait par une diffusion totale au reste du groupe. Cela nécessite des connexions permanentes entre les réplicas. Si un réplica n'est pas joignable au moment de l'émission d'une nouvelle commande, Gina ne propose pas de mécanisme pour lui réémettre cette action plus tard. Par ailleurs, seuls les conflits structurels entre les commandes (modification/destruction) et les conflits de mises-à-jour sont détectés. Or, même dans le domaine des applications graphiques, il existe des conflits liés au référencement entre les objets (p. ex. les objets graphiques peuvent appartenir à un certain calque, un objet texte peut suivre un objet courbe, deux objets peuvent être liés par un connecteur, ...). D'autre part, la résolution des conflits est entièrement laissée à la charge de l'utilisateur. C'est une source potentielle d'incohérence : pour un même conflit, deux utilisateurs peuvent prendre des choix différents et donc ne pas amener le système dans un état cohérent.

3.5.2 Bayou

Bayou [PST+97] est un protocole généraliste de réplication optimiste de base de données. Il garantit la cohérence à terme de l'ensemble des réplicas grâce à une diffusion épidémique [DGH+87] des mises-à-jour. Les synchronisations se font deux-à-deux, sans contrainte sur le choix du partenaire ni sur leur fréquence.

Le protocole de réplication de Bayou est *anti-entropique* . Il garantit que tous les réplicas tendent vers un état globalement cohérent et stable. Pour cela, il gère la journalisation de toutes les opérations d'écriture faites sur les données répliquées. La journalisation n'étant pas transparente, c'est au programmeur de la déclencher. Lorsqu'une écriture est journalisée, elle est intégrée dans la base de données. Cependant, elle reste à l'état de *tentative*, tant qu'elle n'a pas été validée par le *réplica primaire*. Ce dernier assure que lorsqu'une opération

a été validée, la partie préfixe du journal contenant cette opération ne subira plus d'insertions de nouvelles opérations. Cet ensemble d'écritures peut alors être intégré à la base de données et supprimé du journal. Néanmoins, elles ne sont pas immédiatement supprimées après avoir été validées car certains réplicas peuvent ne pas les avoir encore reçu et devraient alors récupérer toute la base de données. Pour cela, Bayou gère un compteur d'effacement qui permet de savoir, sur un réplica donné, quelles opérations validées ont été supprimées.

Bayou garantit la diffusion épidémique des écritures tout en garantissant des échanges incrémentaux [PST+97]. Un réplica est assuré de ne recevoir une écriture qu'une seule fois, quel que soit les partenaires avec lesquels il se synchronise. Pour cela, toutes les écritures sont estampillées avec une horloge logique et chaque réplica gère un vecteur de versions [WPE+83]. Le vecteur de versions repose sur la *propriété préfixe* des journaux : Si un réplica possède l'écriture estampillée e sur le réplica R alors il possède aussi toutes les écritures de R ayant une estampille inférieure à e. Le vecteur de versions est envoyé lors d'une synchronisation pour permettre au partenaire de savoir les écritures qu'a déjà reçues ce réplica. Grâce à cela, les synchronisations ne nécessitent qu'un seul échange ce qui réduit la consommation de bande passante et autorise les échanges par médias amovibles (p. ex. disquette). En contre partie, la propriété préfixe empêche la réplication partielle des données.

En plus du mécanisme de réplication, Bayou offre des *garanties de session* [TDP+94]. D'une session à l'autre, un client d'une base de données répliquée peut changer de serveur répliqué. Bayou garantit à ce client qu'il percevra des vues cohérentes avec les écritures qu'il a déjà réalisées sur les autres réplicas. Il offre quatre garanties :

- Écritures monotones (*Monotonic Write*) : l'ordre des écriture du client est conservé ;
- Lectures monotones (*Monotonic Read*) : à chaque lecture, le client est assuré d'obtenir des valeurs équivalentes ou plus récentes ;
- Lecture de ses écritures (*Read Your Writes*) : le client est assuré d'obtenir la version d'une données reflétant au moins sa dernière écriture sur cette donnée ;
- Écriture succédant aux lectures (*Writes Follow Reads*) : les écritures qui suivent une lecture sont assurées de succéder aux écritures qui précèdent la lecture.

La gestion des conflits est basée sur deux mécanismes : la vérification des dépendances (*Check dependencies*) et les procédures de fusion (*Merge procedures*) [TTP+95]. À chaque écriture est associé une analyse de dépendance sous forme d'une requête SQL et de son résultat. Lors d'une synchronisation, le réplica qui reçoit les écritures se repositionne dans l'état précédant la plus petite écriture reçu[7]. Les écritures reçues et celles défaites sont ensuite réintroduites dans l'ordre des estampilles. Avant d'enregistrer chaque écriture, on vérifie les

[7] Bayou dispose en plus du journal des écritures un journal des opérations à défaire (*undo*), qui permet de rétablir le réplica dans un état antérieur.

dépendances en rejouant la requête SQL et en comparant le résultat avec celui inscrit dans l'écriture. En cas de divergence, on applique la procédure de fusion. Celle-ci prévoit des alternatives pour chaque écriture. Par exemple, sur un agenda de groupe, à chaque réservation de salle, on enregistre des réservations alternatives, au cas où la première aurait déjà été retenue. Ce mécanisme est à la fois simple et souple pour le programmeur. Néanmoins, il augmente notablement la taille des enregistrements du journal (requête SQL, résultat et code pour la procédure de fusion). D'autre part, l'ordre des estampilles a une grande importance sur la résolution : plus une écriture a une petite estampille, plus elle est jouée tôt et plus elle a de chance d'être acceptée. Or, les estampilles sont basées sur une horloge logique, ce qui ne permet pas un comportement prédictible pour les utilisateurs.

3.6 Conclusion

De nombreux travaux ont été menés sur la réplication. Si la majorité d'entre-eux concerne l'approche pessimiste, l'essor de l'informatique nomade a relancé l'intérêt pour la réplication optimiste. Cependant, peu de protocoles répondent à l'ensemble des critères que nous avons définis pour les applications collaboratives.

La principale difficulté de la réplication optimiste reste la synchronisation des réplicas ayant produit des écritures conflictuelles. Dès qu'il y a plusieurs types de données qui se référencent mutuellement, il faut assurer un ordonnancement des créations de données garantissant la séquentialité des dépendances. Si on admet en plus que les données sont mutables, il faut être capable de gérer les conflits de mutation sur ces données (modification et destruction) ainsi que sur les données qui en dépendent.

Les conflits doivent être détectés mais aussi résolus. La plupart des protocoles que nous venons d'étudier offrent une gestion de conflits insuffisante pour le modèle de données que nous visons. Seul Bayou intègre un réel support paramétrable de détection des conflits. Cependant, la mécanique de résolution est déchargée sur l'utilisateur (il doit prévoir les éventuels conflits et trouver des alternatives). Or, en exploitant la nature des objets (i.e. leurs attributs et leur référencement), le système a déjà des informations suffisantes pour détecter automatiquement certains conflits. De plus, pour bien des applications, il faut permettre au programmeur de mettre en place, lors de la conception de l'application, des résolutions adaptées aux conflits propres à la nature des données et au mode de collaboration de cette application.

La réplication augmente les performances entre un client et un serveur répliqué. De même, le système doit aussi assurer de bonnes performance lors de la synchronisation des serveurs. En particulier, il faut chercher, comme dans NNTP, à rendre les échanges de données incrémentaux pour éviter qu'un ré-

plica ne reçoive plusieurs fois les mêmes données. De même, l'utilisation de la réplication partielle est une solution intéressante pour optimiser les échanges et réduire la consommation de bande passante et d'espace de stockage.

La gestion de l'évolution de schéma des données répliquées est un critère important dans un système distribué à large échelle. En effet, les mises-à-jour de l'application se font généralement de façon asynchrone et le système doit continuer à fonctionner lorsque les réplicas n'ont pas tous les mêmes versions du logiciel et des structures de données.

Un protocole de réplication adapté aux applications collaboratives asynchrones devrait donc intégrer les mécanismes de journalisation utilisés dans Bayou et Gina pour permettre le retour sur un état particulier. La réplication partielle autoriserait la gestion de gros volumes de données tout en permettant à de petites machines d'intégrer le groupe des réplicas. Enfin, la détection et la résolution de conflits devrait être automatisable par le développeur pour réduire la tâche des utilisateurs et de l'administrateur.

Chapitre 4

Réplication optimiste de données structurées mutables

Dans cette thèse, nous présentons un système pour la construction d'applications collaboratives répliquées. Le système prend en charge la gestion de la persistance des données et leurs synchronisations sur les différents réplicas. Les réplicas sont autonomes et effectuent leurs écritures concurremment. Ils ne se connectent que ponctuellement pour se synchroniser. Le partitionnement des réplicas est l'état normal du système. Durant une période déconnectée, les lectures et les écritures peuvent porter sur n'importe quelle donnée. Des écritures concurrentes sur des données liées peuvent entraîner des conflits. Ceci implique que l'application doit intégrer la gestion des conflits au prix de la perte potentielle d'une partie des écritures conflictuelles. Le système est donc plutôt destiné à des applications produisant peu de conflits graves ou insolubles. Néanmoins, il ne s'agit pas d'un système totalement transparent. Le développeur peut intervenir à différents niveaux pour adapter les mécanismes aux spécificités de son application.

Le protocole de synchronisation garantit la cohérence à terme des réplicas. La synchronisation se fait deux à deux. Une fois synchronisés, deux réplicas sont cohérents. L'ensemble des réplicas converge vers un état globalement cohérent par propagation épidémique des écritures. Pour optimiser les échanges, d'une part les synchronisations entre deux réplicas sont incrémentales : deux réplicas ne s'échangent que des nouvelles écritures. D'autre part, le protocole offre une extension pour la réplication partielle des données en utilisant des filtres de synchronisation.

Pour faire face aux conflits d'écritures qui peuvent survenir entre deux synchronisations, nous proposons un gestionnaire de conflits et de contrôles d'intégrité semi-automatiques. Il automatise la détection et la résolution des conflits de mutation sur une même donnée. Il repose pour cela sur un nouveau modèle d'estampillage qui capture des informations temporelles dans l'estampillage des écritures tout en préservant leur ordonnancement causal. Pour les autres types de conflits, le gestionnaire peut être spécialisé (p. ex. conflits de doublon ou

de référencement). Enfin, il permet au développeur de garantir le respect des contraintes d'intégrités des données après une synchronisation.

La persistance des données est assurée par la journalisation des écritures. Un SGBDR ne gère que l'état courant des données. Or, on a besoin de connaître les écritures qui ont fait diverger les deux réplicas depuis leur dernière synchronisation. En effet, ceci est nécessaire pour garantir l'incrémentalité des synchronisations et la detection automatique des conflits. Par ailleurs, les SGBDR ne sont pas bien adaptés pour gérer la persistance de graphes d'objets. Nous proposons donc un modèle de persistance d'objets par journalisation de leurs mutations. Il est accompagné du mécanisme inverse capable de restituer le graphe d'objets correspondant au journal des mutations. La persistance et le chargement des objets est générique quelles que soient les classes gérées. Enfin, il offre un support simple pour la gestion des évolutions de schéma.

Des extensions au protocole de synchronisation sont présentées pour l'exploiter dans différentes situations. Dans le cadre d'applications collaboratives basées sur le modèle client/serveur (où les serveurs sont répliqués), nous décrivons une solution pour gérer un cache côté client tirant partie de la journalisation des écritures. Pour les utilisation à large echelle, nous proposons la mise en place de barrière de synchronisation permettant de faire face aux réplicas obsolètes. Enfin, nous étudions l'utilisation du protocole pour la synchronisation de bases de données relationnelles.

Pour maintenir une application collaborative répliquée, il est nécessaire de convenir d'un système de gestion des mises-à-jour. La réplication du code est complémentaire de la réplication des données. Cependant, l'application n'est généralement pas réduite à un seul programme exécutable. Nous proposons une infrastructure d'accueil de modules applicatifs, capable de gérer leur déploiement, leur mises-à-jour et leur dépendances.

Le reste de la thèse se décompose ainsi : la partie II (p. 37) présente les mécanismes de journalisation des écritures sur un graphe d'objets. Nous y décrivons la gestion du journal (chapitre 5 (p. 37), celle de l'état courant et le maintien de cohérence entre les deux (chapitre 6 (p. 47)). Dans la partie III (p. 63), nous étudions la gestion de la cohérence du systèmes. Nous y présentons le protocole de synchronisation (chapitre 7 (p. 63)), un modèle d'estampillage adapté à la réplication optimiste et la gestion des conflits (chapitre 8 (p. 79)), les extensions possibles du protocole (chapitre 9 (p. 91)) et la réplication des modules applicatifs (chapitre 10 (p. 101)). Enfin, dans la partie IV (p. 115), nous présentons l'implémentation de la bibliothèque de réplication et l'application à Pharos.

Deuxième partie

Modèle de persistance adapté à la réplication

Chapitre 5

Journalisation des écritures

Les applications collaboratives que nous visons ont des durées d'exploitation longues (plusieurs années). Elles produisent des données qui doivent être préservées d'une exécution à l'autre. Ces données sont dites persistantes. Toutes les données gérées par l'application ne sont pas forcément persistantes. C'est à la charge du développeur de définir celles qui doivent l'être et à quel moment. Pour cela, il doit choisir un mécanisme de persistance adapté à son application et à l'environnement d'exécution. Dans notre cas ce mécanisme doit en plus être compatible avec le système de réplication.

5.1 Nécessité d'un historique des versions

Notre système de réplication prévoit des successions de phases d'écritures asynchrones et de phases de resynchronisations. Durant une synchronisation, le système doit déterminer les nouveautés ; c'est-à-dire les données qui ont été créées, celles qui ont été modifiées et celles qui ont été détruites. Pour cela, il a besoin de savoir ce qui s'est passé sur chacun des réplicas. Les actions de consultation des données (les lectures) n'interviennent pas dans le mécanisme de synchronisation. Nous ne nous intéressons qu'aux écritures qui ont fait diverger les deux réplicas. Le modèle de persistance doit donc fournir les informations nécessaires au calcul des divergences.

La seule comparaison des valeurs courantes des données persistantes ne suffit pas à déterminer complètement les divergences. Elle ne fait pas apparaître les données détruites et il est difficile de savoir quels attributs d'une donnée ont été modifiés. Cela implique qu'en plus des valeurs des données applicatives, d'autres données doivent être rendues persistantes. Quelque soit la forme sous laquelle ces informations supplémentaires sont gérées, elles doivent fournir des renseignements sur l'*histoire des données*. Notre modèle de persistance doit donc intégrer un mécanisme permettant de connaître les versions successives des données.

5.2 Modèle de gestion des versions

La gestion des versions peut se faire selon plusieurs modèles. Ce modèle doit être choisi en fonction de l'utilisation que l'on fait des versions. Toutes les versions sont-elles utiles ? Quelles versions seront les plus utilisées ? Quel coût l'application est-elle prête à payer pour accèder à une version ?

Pour la synchronisation, nous avons besoin de pouvoir déterminer l'histoire commune de deux réplicas (i.e. avant la divergence) et de connaître les mutations des données (i.e. création, mises-à-jour ou destructions) qui ont eu lieu dans la phase de divergence. Ensuite, au fil des synchronisations, l'histoire commune des réplicas devient plus importante que leur divergences. Durant les synchronisations, on aura donc plutôt tendance à rétablir des versions récentes des données. Enfin, le système de gestion de versions doit permettre l'intégration des versions divergentes du réplica partenaire pour reconstruire une nouvelle histoire.

Trois principaux modèles de gestion de versions sont envisageables :

1. **Historique des valeurs** : Toutes les valeurs successives de la donnée sont gardées. Ce modèle a l'avantage de pouvoir fournir très rapidement n'importe quelle version. Par contre, il est nécessaire d'effectuer des calculs pour comparer deux histoires et connaître précisément les détails des divergences. Il faut aussi convenir d'un marquage spécial pour identifier une donnée détruite. D'autre part, à chaque nouvelle version, l'état complet de la donnée est réenregistrée. Cela peut s'avérer coûteux si les modifications sont fréquentes et si à chaque mise-à-jour la donnée varie peu ;

2. **Historique des mutations** : La valeur initiale de la donnée ainsi que toutes les mutations qu'elle a subit sont gardées. La version v est obtenues en jouant sur la valeur initiale les $(v-1)$ mutations. C'est le principe utilisé dans le gestionnaire de version SCCS [BG77].

   Ce modèle a l'intérêt d'être incrémental et donc moins coûteux en espace de stockage. Il a aussi l'avantage de ne produire que des ajouts ce qui augmente les performances des accès disques (les accès aléatoires sont plus coûteux que les accès séquentiels). En contre partie, la construction d'une version nécessite des calculs et plus elle est récente, plus le calcul est coûteux. La version courante est donc la plus coûteuse à calculer ;

3. **Historique des mutations inverses** : Ce modèle est une inversion du précédent. Seule la dernière valeur de la donnée est conservée. A chaque nouvelle version v, $(v-1)$ est remplacé par v et on enregistre l'opération de mutation qui permet de reconstruire $(v-1)$ à partir de v. Ce modèle est utilisé par le gestionnaire de version RCS [Tic85].

   La reconstruction d'une version nécessite toujours des calculs mais plus elle est récente et moins elle est coûteuse. De plus, on dispose, sans calculs,

de la dernière version. En revanche, les performances sont moins bonnes car les suppressions nécessitent des accès disques non-linéaires.

Le premier modèle n'est pas bien adapté à notre gestion des versions. En effet, les objets vont potentiellement subir de nombreuses mutations et le coût de l'analyse de toutes les versions pour déterminer les divergences de mutation et les intégrer peut être élevé. Le deuxième modèle est intéressant car il permet de calculer les divergences entre deux histoires plus facilement. Cependant, le rétablissement d'une version récente nécessite de rejouer toutes les mutations depuis le début. Il est donc plutôt recommandé lorsqu'il y a peu de mutations. Le troisième modèle est sur ce point plus performant car le rétablissement des versions se fait de la plus récente à la plus ancienne. Néanmoins, ce modèle n'est pas très adapté au calcul des divergences ni à leur intégration car chaque mutation dépend de la suivante. Le tableau suivant reprend les avantages et inconvénients de chacun de ces modèles.

Modèle	Calcul d'une version récente	Calcul des divergences	Intégration des versions
Historique des valeurs $(V_1, \ldots, V_n)$	immédiat	complexe	complexe
Historique des mutations $(V, \Delta_1, \ldots, \Delta_n)$	simple mais long	simple	simple
Historique des mutations inverses $(\Delta_1^{-1}, \ldots, \Delta_{n-1}^{-1}, V_n)$	rapide	complexe	complexe

Le modèle le mieux adapté pour la synchronisation est une combinaison des deux derniers. Ce modèle est proche de celui de Gina[BG93]. Toutes les mutations effectuées sur une donnée ainsi que les mutations inverses sont enregistrées. La valeur courante est gérée en mémoire et reconstruite, au démarrage, en exécutant toutes les mutations. Grâce à cela, on assure à la fois des accès immédiats à la version courante, rapides pour les versions passées et de bonnes performances d'entrées/sorties.

L'ensemble des opérations d'écriture sont gérées dans le *journal des opérations*. Chaque écriture est représentée par une nouvelle opération. Le journal intègre, pour chaque opération, une opération inverse, qui effectue la mutation opposée. Les opérations sont immutables et possèdent chacune un identificateur unique. Ceci est nécessaire pour permettre à un réplica de savoir s'il possède déjà une opération donnée mais aussi pour les ordonner. A chaque redémarrage, le système rejoue les opérations du journal et reconstruit le graphe des objets (entièrement ou partiellement en mémoire). Il doit aussi assurer la cohérence entre les deux représentations : la modification des objets en mémoire si le journal

évolue et l'ajout de nouvelles opérations lorsque ces objets sont modifiés. Enfin, il fournit un moyen de rétablir des versions cohérentes de l'ensemble des données.

Après une rapide analyse de quelques systèmes de persistance par journalisation, nous décrivons la gestion du journal des opérations et celle de l'état courant.

5.3 Persistance par journalisation des écritures

La persistance par journalisation des écritures existe depuis longtemps dans les système de bases de données. Ce mécanisme est en particulier utilisé dans la gestion des transactions [BN97]. Cependant, dans ce cas d'utilisation, l'objectif n'est pas la persistance complète des données de la base mais plus précisément celle des données mises en jeu lors d'une transaction. La journalisation permet de pouvoir rapidement valider ou annuler une transaction.

Des travaux ont aussi été menés sur la journalisation des écritures dans les systèmes de fichiers [RO91]. Mais là aussi, les objectifs étaient différents et plus orientés sur l'amélioration des performances. En ramenant les écritures disques à des écritures séquentielles dans le journal, on évite les coûteuses séries d'accès aléatoires. Bien que les différentes versions d'un fichier soit contenues dans le journal, cette possibilité n'a pas vraiment été exploitée dans ce système. Le système de fichiers Coda[KS93] possède lui aussi un journal des opérations. Celui-ci est assez proche de nos préoccupations puisqu'il sert aussi pour la synchronisation des réplicas. En fait, il se base sur un système de journalisation annexe, la RVM [SMK+94]. Elle est utilisé pour journaliser non pas le contenu des fichiers mais les opérations sur les répertoires.

LumberJack [HdH98], un entrepôt d'objets persistants, utilise aussi une persistance par journalisation. Dans ce système, ce ne sont pas les opérations qui sont journalisées mais l'état des objets (plus précisément les pages mémoires contenant les objets). Un ramasse-miettes nettoie régulièrement le journal des anciennes versions des objets. De même que pour le système de fichiers de Rosenblum et Ousterhout [RO91], l'accent est mis sur les performances. Le journal n'est pas conçu pour exploiter les versions successives des données mais pour minimiser les temps d'accès, de validation et de prise d'instantané[1].

5.4 Gestion du journal

Les données que nous devons rendre persistantes sont gérées sous forme d'objets Java. Ces objets sont mutables ; c'est-à-dire, que leur état peut changer au cours du temps et qu'ils peuvent être détruits. Chaque changement d'état d'un objet est représenté dans le journal par une *opération de mutation* sur cet objet. Le journal contient principalement trois types d'opérations de mutation :

[1] *Snapshots.*

des opération de création (`create`), de mise-a-jour (`update`) et de destruction (`delete`). Chaque opération de mutation ne concerne qu'un seul objet, désigné par un identificateur unique.

Le journal contient aussi d'autres types d'opérations (p. ex. des opérations de regroupement, des opérations liées aux synchronisations, ...) dont les rôles sont décrits dans les sections qui les concernent. Dans cette section nous présentons la gestion des opérations du journal, illustrée à travers les opérations de mutation. Par souci de concision, le terme *opération* désignera ces opérations.

5.4.1 Représentation des opérations

Chaque opération du journal est représentée par une arborescence de balises[2] XML [BPSM98]. XML a été choisi pour sa généricité, sa simplicité, sa représentation ASCII et la grande quantité d'outils disponibles (en particulier les *parsers* Java). Cependant, XML est un langage destiné à la description d'un état structuré. Le journal n'étant, par nature, jamais terminé, il ne respecte pas complètement la norme XML qui prévoit qu'un jeu de balises ouvrantes et fermantes encadrent le document. Dans notre cas, il s'agit donc plutôt d'une succession de balises que d'un véritable document XML.

Les opérations sont uniques et immutables. Chacune est identifiée par son estampille (représentée dans la balise par l'attribut `stamp`). Outre son rôle d'identification, celle-ci doit aussi permettre l'ordonnancement des opérations dans le journal. Dans une exploitation sans réplication, il suffirait d'un simple compteur qui s'incrémente à chaque enregistrement. Cependant, pour que l'application puisse supporter des écritures concurrentes asynchrones, les estampilles doivent être construites selon un mécanisme adapté. Celui-ci est détaillé plus loin dans ce chapitre.

Pour les opérations de mutation, la représentation XML que nous avons retenue suit un modèle objet : la classe de l'objet est représentée par le nom de la balise, l'instance par un attribut d'identification, la méthode par un attribut décrivant le type de mutation et les paramètres par le reste des attributs de la balise. Le tableau ci-dessous décrit les attributs communs à toutes ces opérations :

[2] Pour être précis, il faudrait parler, dans la terminologie XML, d'*éléments*. Cependant, ce terme est ambigü et peut entraîner certaines confusions. Aussi, nous lui préférons le terme de *balise*.

Attribut	Rôle
`stamp`	Estampille de l'opération.
`id`	Identificateur unique de l'objet sur lequel s'applique l'opération. Par défaut, l'identificateur est construit en prenant l'estampille de création.
`op`	Nature de l'opération (`create`, `update`, `delete`).

Les opérations de création ou de mise-à-jour contiennent la valeur des attributs de l'objet. Ils sont représentés dans la balise XML par une suite d'attributs XML de type *nom=valeur*. Pour les attributs correspondant à des références sur d'autres classes d'objets persistants (p. ex. dans un canal Pharos, une *annotation* qui référence un *term*), la valeur est calculée par le système de persistance. Les pointeurs sont matérialisés dans les attributs d'une balise par une référence sur identificateur (`IDREF` et `IDREFS`). Ce mécanisme ainsi que le mécanisme inverse, utilisé lors de l'interprétation des opérations, sont décrits à la section 6 (p. 47).

5.4.2 Modèle de mise-à-jour partielle

L'enregistrement d'une opération de création consiste à construire une balise XML identifiant l'objet et intégrant tous ses attributs. Une opération de destruction n'a besoin que de l'identifiant. Par contre, les opérations de mise-à-jour peuvent être enregistrées selon un modèle *total* (tous les attributs de l'objet sont réenregistrés) ou *partiel* (seuls les attributs modifiés sont enregistrés). Le modèle total est le plus simple à traiter puisque les attributs de l'opération sont construits à partir de l'état courant de l'objet modifié. Le modèle partiel est plus compliqué à mettre en œuvre car il faut être capable de déterminer les attributs concernés par la mise-à-jour. Cela nécessite soit de pouvoir calculer les différences entre deux états d'un même objet ; soit de contrôler le mécanisme de mise-à-jour des objets.

C'est cependant le modèle partiel que nous avons retenu car il réduit la taille des enregistrements dans le journal et il permet une gestion plus souple des différentes versions d'un même objet. Ce dernier point est particulièrement important pour la détection et la résolution des opérations conflictuelles lors d'une synchronisation.

L'exemple ci-dessous illustre ces principes sur une entrée du carnet d'adresses. Une entrée du carnet (`card`) est successivement créée, puis modifiée (l'email et le le numéro de fax) et enfin détruite. Le journal contient alors la séquence d'opérations suivante :

```
<card stamp="ab1:27341"
      id="ab1:27341"
      op="create"
      homeTel="01 23 45 67 89"
```

```
      workTel="01 39 63 51 47"
      email="dedieu@tif.inria.fr"
      name="Olivier Dedieu" >
</card>

<card stamp="ab1:27357"
      id="ab1:27341"
      op="update"
      email="olivier.dedieu@inria.fr" >
</card>

<card stamp="ab1:27386"
      id="ab1:27341"
      op="update"
      fax="01 39 63 53 72" >
</card>

<card stamp="ab1:27408"
     id="ab1:27341"
     op="delete" >
</card>
```

5.4.3 Opérations de compensation

Pour exploiter pleinement les différentes versions des données, il faut être en mesure de pouvoir rétablir un état des objets correspondant à un instant donné, défini par une estampille. Une solution consiste à annuler l'état courant de toutes les données, puis à rejouer le journal depuis le début jusqu'à la position recherchée, identifiée par une estampille. Cependant, lorsque la taille du journal est importante, cette technique est coûteuse. En particulier, si la partie à « défaire » est très petite par rapport à la taille du journal, il est plus efficace d'annuler l'action des dernières opérations enregistrées jusqu'à l'estampille donnée.

Comme nous avons choisi un modèle de mise-à-jour partielle, on ne peut pas rétablir un état simplement en rejouant les opérations du journal dans l'ordre inverse. Pour cela, nous associons à chaque opération une *opération de compensation*. Comme son nom l'indique, son rôle est d'annuler l'action de l'opération à laquelle elle est associée. Ainsi, en jouant les opérations de compensation en partant de la fin du journal, on peut rétablir un état courant correspondant à une position donnée.

L'exécution d'une opération de compensation peut être prise en compte de deux manières dans le journal [PK92] : soit on gère un pointeur sur le journal qui marque la dernière opération supprimée (*linear undo model*) ; soit on ajoute une nouvelle opération dans le journal qui représente l'opération de compensation (*history undo scheme*). Le second modèle a l'avantage d'être plus riche puisque l'on garde trace de toutes les opérations de compensation. Il est aussi plus performant car on ne produit que des ajouts dans le journal. En contre partie il a l'inconvénient de faire grossir le journal.

Dans notre cas, les opérations de compensation seront utilisées lors des synchronisations pour fusionner des suffixes de journaux. Notre modèle de compensation suit donc le modèle linéaire et ne modifie que l'état courant des données en mémoire. Il n'engendre pas directement de modifications dans le journal ; seul le pointeur de fichier est déplacé. Cependant, après l'exécution d'une série d'opérations de compensation, l'enregistrement d'une nouvelle opération doit être suivi de l'effacement de la fin du fichier à partir de cette opération (pour ne pas garder des opérations compensées).

Le calcul d'une opération de compensation nécessite de connaître la valeur précédente de la donnée. Il ne peut donc pas être réalisé efficacement au moment de la compensation. Les opérations de compensation sont calculées et enregistrées en même temps que les opérations auxquelles elles sont associées. Elles sont représentées par une balise `<undo>` contenue dans chaque balise représentant une opération. Cette balise contient le type de l'opération et la valeur des attributs associés. Le tableau ci-dessous répertorie pour chaque type d'opérations, l'opération de compensation associée et la nature de ses attributs.

Opération	Opération de compensation
`create`	`delete`
`update`	`update` rétablissant les valeurs précédentes pour les attributs modifiés.
`delete`	`create` avec les valeurs de tous les attributs du dernier état de la donnée.

L'exemple ci-dessous reprend la séquence d'opérations présentée précédemment en incluant les opérations de compensation :

```
<card stamp="ab1:27341"
      id="ab1:27341"
      op="create"
      homeTel="01 23 45 67 89"
      workTel="01 39 63 51 47"
      email="dedieu@tif.inria.fr"
      name="Olivier" >
  <undo op="delete" />
</card>

<card stamp="ab1:27357"
      id="ab1:27341"
      op="update"
      email="olivier.dedieu@inria.fr" >
<undo op="update"
      email="dedieu@tif.inria.fr" />
</card>

<card stamp="ab1:27386"
      id="ab1:27341"
      op="update"
```

```
      fax="01 39 63 53 72" >
<undo op="update"
      fax="" />
</card>

<card stamp="ab1:27408"
      id="ab1:27341"
      op="delete" >
<undo op="create"
      homeTel="01 23 45 67 89"
      workTel="01 39 63 51 47"
      email="olivier.dedieu@inria.fr"
      name="Olivier"
      fax="01 39 63 53 72" />
</card>
```

L'chapitre 12 (p. 121) détaille les différents attributs des opérations concernant les objets gérés dans Pharos.

5.4.4 Groupement d'opérations

Certaines applications peuvent produire plusieurs opérations simultanément. Par exemple, dans une application de CAO, l'utilisateur peut sélectionner un groupe d'objets et les déplacer. De même, dans le thésaurus d'un canal Pharos, le déplacement d'un nœud provoque une mise-a-jour de positionnement des nœuds voisins dans le nœud de départ et le nœud d'arrivée.

Si on ne prend aucune mesure particulière, ces séries d'opérations produites simultanément peuvent être compensées séparément. Or ceci n'est pas souhaitable. Les groupes opérations produites de façon atomique, doivent être compensées de la même façon. Lors des synchronisations il faudra aussi préserver l'atomicité des groupes d'opérations. Cela a pour conséquence de ne pouvoir introduire aucune opération extérieure au groupe à la suite d'une synchronisation. Il faut donc assurer l'isolement de ces opérations pour l'ordonnancement.

Un groupe d'opérations est représenté par une opération spéciale : l'*opération de regroupement*. Celle-ci est matérialisée dans le journal par une balise (`<group>`) qui encadre toutes les opérations concernées. L'opération de regroupement porte une estampille qui vaut pour toutes les opérations. Il s'agit d'une simple valeur entière qui assure l'ordre au sein du groupe. Les opérations sont ordonnées selon l'estampille de leur groupe puis selon leur propre estampille au sein du groupe. D'autre part, lorsque la dernière opération du groupe doit être compensée, le système assure que toutes les opérations de compensation du groupe seront jouées.

L'exemple ci-dessous illustre l'utilisation des opérations groupées avec le thésaurus de Pharos. Chaque terme possède un attribut qui définit sa position vis-à-vis de ses frères. Le terme `t-11` est déplacé du terme `t-1` sous le terme `lacoon:515`. En conséquence, son ex-frère, `t-12`, remonte d'une place et son

nouveau frère `t-21` descend d'une place.

```
<group stamp="t-40">

<!-- Le noeud est déplacé -->
<term stamp="0"
      id="t-11"
      op="update"
      position="1"
      parent="t-2" >
<undo op="update"
      position="0"
      parent="t-1" />
</term>

<!-- L'ex-frère remonte d'une place -->
<term stamp="1"
      id="t-12"
      op="update"
      position="0" >
<undo op="update"
      position="1" />
</term>

<!-- Le nouveau frère descend d'une place -->
<term stamp="2"
      id="t-21"
      op="update"
      position="2" >
<undo op="update" position="1" />
</term>
</group>
```

Chapitre 6

Gestion de l'état courant

Le journal des opérations assure la persistance de l'histoire des mutations des objets. Cependant pour qu'un objet soit réellement exploitable, il est nécessaire de connaître sa valeur courante, c'est-à-dire, celle qui reflète l'exécution de toutes les opérations qui le concernent. L'ensemble des valeurs courantes des objets gérés représente l'*état courant* du système.

Pour construire l'état courant, il faut convenir d'un modèle de stockage approprié. Nous proposons deux modèles complémentaires :

1. L'état courant est entièrement géré en mémoire centrale. Ce modèle est bien adapté soit pour des applications gérant des volumes limités de données, soit pour un serveur nécessitant de bonnes performances et possédant suffisamment de mémoire ;

2. L'état courant est géré de façon persistante. Seuls les index d'accès et une partie des données sont présents en mémoire. Ce modèle est destiné aux systèmes dont le volume de données ne peut pas être géré totalement en mémoire centrale. En contre partie, les performances sont réduites et il requiert une intervention plus importante de la part du développeur.

L'état courant de l'application est pris en charge par un composant dédié : l'*entrepôt*. Celui-ci offre une interface de haut niveau pour gérer les objets persistants de l'application et pour assurer la cohérence entre le journal et l'état courant.

6.1 Construction de l'état courant à partir du journal

Le journal ne contient que les opérations de mutation. Comme nous avons choisi un modèle de mise-à-jour relative, on ne possède pas la dernière version des objets ; l'entrepôt doit donc reconstruire les objets à chaque redémarrage. De même, lors d'une synchronisation, l'application reçoit de nouvelles opérations et peut en annuler (en cas de conflits). L'intégration de ces nouvelles opérations au journal doit être répercutée sur l'état courant des objets.

6.1.1 Reconstruction des objets

Une solution serait de gérer les valeurs courantes comme des n-uplets de chaînes de caractères contenant les dernières valeurs des attributs de l'objet de la même manière que dans le journal. Cette solution est simple à mettre en œuvre mais laisse à l'application tout le travail de conversion de représentation des attributs. Or, on veut permettre au programmeur de gérer ses données persistantes dans des objets Java, en minimisant les contraintes sur ces classes.

La construction d'un état courant sous forme d'objet est en grande partie automatisable grâce aux mécanismes de typage et d'introspection fournis par Java. En effet, l'introspection permet de connaître le nombre et la nature des attributs d'une classe Java. De plus, ces attributs sont bien souvent des types primitifs (entiers, réels, booléens, ...) ou des classes de base (chaînes de caractère, dates, ...) dont la conversion peut être prise en charge par l'entrepôt. Pour les autres types d'attributs, l'application a la charge de fournir à l'entrepôt des *convertisseurs* de représentation. Les détails sur la mise en place des convertisseurs sont donnés au chapitre 11 (p. 115).

6.1.2 Reconstruction du graphe de référencement

Parmi les problèmes de conversion de représentation, celui concernant les références entre objets requiert un traitement spécifique. En effet, certains types de données gérées dans le journal en référencent d'autres (p. ex. une `annotation` dans Pharos référence un `user`, et une liste de `term`). Une première solution consisterait à utiliser les identificateurs des données pour gérer leur référencement. Cependant, cette solution n'est pas performante et augmente le travail du programmeur : pour accéder à tout objet référencé, il faut passer par une table de correspondance. De nombreux travaux de recherche ont été mené pour éviter au programmeur cette tâche laborieuse. La technique proposée consiste à rétablir les pointeurs entre objets à partir des références persistantes (et l'opération inverse lors de l'enregistrement de l'objet). Cette technique est communément appelée le *swizzling de pointeurs*[1] [Mos92]. Plusieurs techniques de swizzling existent et ont des performances différentes en fonction de la nature et du volume de données à traiter. Le choix d'une d'entre elles est donc fortement lié au mode de représentation et de stockage de l'état courant (cf. sections 6.2 et 6.3).

L'entrepôt gère les objets dans une *table de référencement*, indéxés par leur identificateur. Cette table est construite à partir de la lecture des opérations du journal :

- Lors d'une création, on génére une nouvelle instance de la classe correspondant au type de données de l'opération. Cette instance est enregistrée

[1] L'étymologie du terme *swizzling* dans ce contexte est obscure. Nous utilisons cependant ce terme pour parler de cette technique car il n'existe aucune traduction française satisfaisante. On parlera d'*internalisation de référence* à partir d'un identificateur (*Persistant IDentifier* ou *Objet IDentifier* selon les terminologies) et d'*externalisation* pour l'opération l'inverse (appelée *unswizzling*).

dans la table de référencement. Son état est initialisé avec les attributs de l'opération ;

- Lors d'une mise-à-jour, on retrouve l'objet à partir de son identificateur et on modifie son état ;

- Lors d'une destruction, on retrouve l'objet à partir de son identificateur et on le retire des tables de référencement. L'objet n'est plus référencé par l'entrepôt mais peut néanmoins continuer d'exister si l'application le référence par ailleurs (p. ex. au sein d'un index).

6.1.3 Support pour l'évolution de schéma

Il est habituel qu'une application subisse, au cours de son exploitation, plusieurs mises-à-jour. Lorsque ces mises-à-jour concernent la structure des données persistantes, on parle d'évolution de schéma. On considère trois types d'évolution sur la structure : ajout d'attributs, suppression d'attributs et modification du type des attributs. Par exemple, dans le cas de Pharos, l'administrateur du système change la structure d'un canal en ajoutant un nouveau champs textuel pour les annotations.

Dans le cas d'une application centralisée, l'évolution de schéma est prise en compte par l'arrêt de l'application, la modification des données et du code puis un redémarrage. Cependant, dans le cas d'une application répliquée, les mises-à-jour peuvent rarement se faire simultanément sur tous les sites. Par exemple, une évolution peut avoir lieu, pendant qu'un réplica nomade est injoignable. Il faut donc prévoir une période de transition durant laquelle, l'application devra faire face à différentes versions de la structure des données.

L'entrepôt gère l'évolution de schéma dynamiquement sans modifier les opérations de mutation antérieures. Plusieurs versions des opérations de mutation peuvent ainsi cohabiter dans le même journal :

- **Ajout d'attributs** : à l'exécution d'une opération de création, les nouveaux attributs ne sont pas modifiés et ont comme valeur, la valeur par défaut définie par le développeur au sein de la nouvelle classe ;
- **Suppression d'attributs** : à l'exécution d'une opération de création ou de mise-à-jour, les valeurs persistantes des attributs n'existant plus au sein de la nouvelle classe sont ignorées ;
- **Modification du type d'un attribut** : Si l'interpretation de l'attribut est prise en charge par l'application, c'est à elle de supporter les différentes valeurs existantes pour l'attribut. Si l'interprétation est laissée à la charge de l'entrepôt, celui-ci tente une conversion automatique. Si celle-ci échoue, il utilise la valeur par défaut de l'attribut.

6.2 Gestion d'un état courant en mémoire

Dans ce modèle, l'état courant est reconstruit à chaque fois que l'application est redémarrée. L'entrepôt lit et exécute le journal pour construire la totalité des objets en mémoire et rétablit leurs référencements.

L'ensemble des données étant présent en mémoire, le modèle de swizzling le mieux adapté est le swizzling immédiat[2]. Celui-ci définit que, lors de l'internalisation d'un objet, toutes ses références, directes et indirectes, sont rétablies. Notre mécanisme d'externalisation des références présenté plus haut, nous assure que le graphe des objets peut être reconstruit par une lecture séquentielle du journal.

6.2.1 Justification de ce modèle

Le chargement de la totalité des données en mémoire peut apparaître déraisonnable. Cependant, pour les applications gérant un volume moyen de données, il se justifie pleinement :

1. Disposer de l'ensemble des données en mémoire assure des temps d'accès très rapides. La reconstruction des références entre objets lors de la lecture du journal et l'utilisation d'index adaptés amènent à des temps d'accès optimaux ;

2. Dans le cas de Pharos, les données gérées par un canal sont peu volumineuses. D'après une analyse des journaux de plusieurs canaux en exploitation (cf. section 7.4, p. 74), il ressort qu'une annotation consomme en moyenne 1.1 Ko en mémoire[3]. L'utilisation quotidienne d'un canal Pharos par un groupe de huit personnes chargées d'annoter à plein temps a généré une moyenne de 8.2 annotations par jour et par personne. Après un an d'utilisation à ce rythme, ce canal contiendrait environ 16 000 annotations soit un état courant de moins de 20 Mo ;

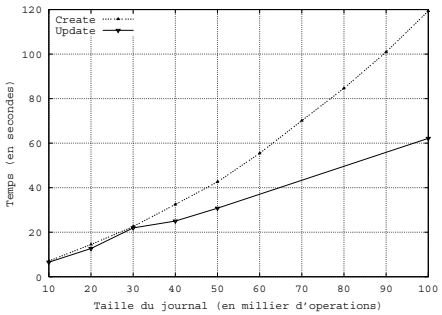
3. Le faible coût du stockage et des unités mémoires rendent ce modèle économiquement viable. Par exemple, un serveur Pharos hébergeant 100 000 annotations (quelle que soit la répartition, p. ex. 10 canaux de 10 000 annotations) occupera donc environ 50 Mo d'espace disque et consommera environ 110 Mo en mémoire. Pour assurer de bonnes performances, il est souhaitable que l'ensemble des données soit contenu en mémoire vive (i.e. non paginées sur le disque). Aux prix actuels des disques durs (500 F HT / 1 Go) et de la mémoire vive (1000 F HT / 128 Mo), une telle configuration coûte environ 1000 F.

[2] *Eager swizzling.*

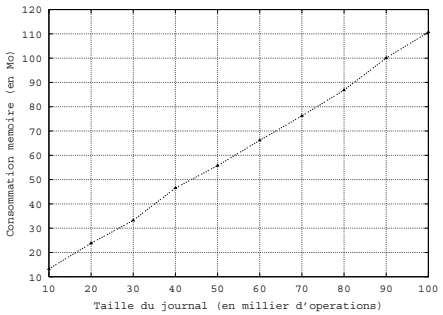
[3] Cette valeur a été calculée en divisant l'occupation mémoire après lecture du journal, par le nombre d'annotations (les autres objets étant en nombre négligeable). La mesure a été faite sur des architectures Intel 32 bits avec une JVM utilisant un compilateur *Just In-Time* (JIT). Sur une architecture 64 bits (DEC alpha), la consommation moyenne est de 2.4 Ko par annotation.

6.2.2 Limite de ce modèle

Pour de très gros volumes de données, la gestion complète de l'état courant en mémoire atteint ses limites. Si la consommation mémoire reste linéaire, le temps de chargement du journal, au-delà d'un certainss volume d'opérations devient très long (cf. figure 6.1). D'autre part, certains environnements d'exploitation peuvent ne pas disposer d'une quantité suffisante de mémoire ou ne pas souhaiter dédier une part importante de leur mémoire à l'application.



(a) Temps de chargement en fonction du nombre d'opérations.



(b) Consommation mémoire en fonction du nombre d'opérations.

FIG. 6.1 – *Courbes représentant le temps d'interprétation du journal et la consommation mémoire du canal en fonction du nombre d'opérations de création et de mise-à-jour. Mesures effectuées avec des canaux remplis artificiellement (100 termes sur 3 niveaux, 3 annotations par URL, 3 termes et 300 octets de commentaire par annotation) sur un Pentium II 450MHz/512Mo, sous Linux avec la JVM 1.1.6 d'IBM.*

6.3 Gestion d'un état courant persistant

Pour faire face au volume, il peut être souhaitable de décharger de la mémoire une partie des objets inutilisés. Il faut alors qu'un objet puisse être chargé rapidement et déchargé lorsqu'il n'est plus nécessaire. Pour assurer de bonnes performances au chargement, l'état ne peut pas être reconstruit par relecture du journal. Nous devons donc gérer, en plus de la persistance dans le journal, la persistance de la dernière version des objets. On doit pouvoir y rechercher la valeur d'un objet, mettre à jour cette valeur, créer de nouveaux objets et les détruire. L'utilisation d'un état courant persistant assure aussi de meilleures performances au démarrage car il n'est plus nécessaire de relire le journal.

L'utilisation d'un second modèle de persistance ne remet pas en cause la persistance par journalisation. Il ne s'agit que d'une optimisation du système. Ce modèle ne détient pas les informations nécessaires à la gestion des versions

(qui sont indispensables pour les synchronisations).

6.3.1 Choix d'un support de persistance

6.3.1.1 Persistance par sérialisation

Il existe de nombreuses techniques pour rendre persistant un graphe d'objets. La sérialisation d'objets [RWWB96] est la première d'entre-elles. Elle est fournie par l'API standard de Java. Ce mécanisme a été introduit en accompagnement de RMI [WRW96] pour permettre l'assemblage[4] et le désassemblage des objets échangés par copie. Le mécanisme n'est pas incrémental. Il faut donc parcourir et sauvegarder à chaque fois la totalité du graphe. De plus, l'implémentation actuelle de la sérialisation n'est pas très performante. Des mesures faites sur plusieurs canaux Pharos ont montré que la lecture d'un état sérialisé est environ six fois plus lente que l'interprétation du journal correspondant. Cette lenteur est essentiellement due à la gestion des types et au mode de représentation en tableau d'octets qui garantit la compatibilité avec les futures versions du *byte-code* [PH99].

6.3.1.2 Persistance par accessibilité

D'autres techniques plus ambitieuses, dites de *persistances par accessibilité* [AM95, ADJ+96], proposent de rendre persistant tout objet atteignable par l'une des racines de persistance définies par le programmeur. Cependant, leur utilisation entraîne des modifications profondes dans le mécanisme de gestion des objets (en particulier lors de l'accès aux attributs) et nécessite des JVM particulières. D'autre part, ces techniques posent des problèmes de déploiement. En effet, elles n'offrent généralement pas de support pour l'évolution de schéma des classes et ne peuvent donc pas relire l'état d'un objet lorsque sa classe a changé. Ceci est particulièrement gênant lorsque les données et les classes sont distribuées sur différents sites et que les mises-à-jour de l'application sont asynchrones.

6.3.1.3 Utilisation d'une base de données

Une autre solution consiste à utiliser une base de données. Au démarrage du canal, seules les données nécessaires au fonctionnement de la base (p. ex. les index) sont chargées en mémoire. La complexité d'intégration avec une base de données est proportionnelle à l'éloignement entre le modèle objet du langage (Java dans notre cas) et le modèle de gestion des données de la base :

1. l'intégration avec une base de données de type *entrepôt persistant d'objets*[5]

[4] *Marshaling.*

[5] Ces systèmes sont intermédiaires entre les bases de données traditionnelles et les systèmes de persistance par accessibilité. On peut, néanmoins, les classer dans la catégorie des bases de données car ils offrent les mêmes services : stockage, requêtes, transactions, ...

(p. ex. [LLB+97]) est relativement simple puisque ce type d'outil gère directement la persistance d'objets Java ; ils nécessitent généralement un post-traitement du *byte-code* des classes utilisées ;

2. l'intégration avec une base de données orientée objet [**?**] nécessite d'adapter le modèle objet de Java à celui de la base ; d'autre part, depuis l'apparition des entrepôts persistants d'objets, l'intérêt porté aux bases de données objets a diminué ;

3. l'utilisation d'une base de données relationnelle nécessite une transformation du modèle objet de Java vers le modèle table/relation. Dans ce cas, on peut utiliser certains outils effectuant automatiquement ces transformations [Sun98, XdSS97]. Généralement, ces outils n'offrent pas le rétablissement du graphe de références entre objets.

6.3.1.4 Utilisation d'un fichier séquentiel indexé

Tous ces outils sont relativement complexes à mettre en œuvre car ils offrent bien plus que le simple stockage des données. D'autre part, le mécanisme de stockage n'a pas besoin d'exploiter la structure des objets. Il est donc possible d'utiliser un mode de stockage plus simple, avec une base de données de type séquentiel indexé. Les index sont les identificateurs persistants des objets et les enregistrements sont la représentation externalisée en XML de l'objet. Un cache des objets fréquemment utilisés est géré pour augmenter les performances. Les détails d'implémentation sont décrits au chapitre 11 (p. 115).

La cohérence entre le journal et la base est maintenue en appliquant sur les données gérées dans la bases toutes les opérations enregistrées dans le journal. Comme pour la gestion de l'état courant en mémoire, les opérations de compensation (cf. section 5.4.3, p. 43) doivent être appliquées sur les données de la base pour rétablir un état correspondant à une version recherchée.

6.3.2 Internalisation des objets référencés

L'internalisation des références nécessite une attention particulière. La technique de swizzling immédiat utilisée dans la gestion de l'état courant en mémoire peut entraîner une consommation excessive de mémoire. En effet, si toutes les références d'un objet internalisé le sont aussi, on risque, par transitivité, d'internaliser beaucoup d'objets. Cela dépend fortement de la structure des référencements entre données. Dans le cas des canaux Pharos, la structure des objets ne peut pas conduire à une telle situation. Cependant, si, par exemple, l'arbre des termes avait été construit de façon descendante, l'accès à la racine entraînerait le chargement immédiat de tout les termes. Le *unswizzling à la demande* [6] permet d'éviter ce comportement. Dans ce cas, une référence n'est résolue que

[6] *Lazy swizzling*

lorsqu'on y accède.

L'implémentation de ce mécanisme de façon transparente nécessiterait de contrôler le mécanisme d'accès aux attributs des objets. Comme nous l'avons dit précédemment, l'intervention au sein de la JVM n'est pas souhaitable pour des raisons de portabilité. Les *meta-object protocol* [TC98, OB98, KG98] proposent ce genre de contrôle sous forme d'extension au langage. Cependant, ils requièrent des machines virtuelles (JVM) et des compilateurs Java particuliers. Depuis la version 1.3, Java intégre un mécanisme d'objets proxy[7] qui permet de prendre le contrôle des invocations sur un objet. Sous réserve que les modifications d'attributs se fassent via des méthodes d'accès[8], ce mécanisme est un bon candidat pour automatiser l'internalisation des références. Cependant, la pose d'un objet proxy sur chaque objet augmente la consommation mémoire et ralentit les performances de l'application.

Une autre approche consiste à demander l'intervention du programmeur pour effectuer l'internalisation au moment de l'accès à la donnée. C'est cette approche que nous avons retenue. Le programmeur définit les attributs qui doivent être résolus de façon immédiate ou retardée. Avant de retourner un objet d'une classe résolue de façon retardée, le programmeur doit déclencher l'internalisation de l'objet référencé. Pour cela, la solution la plus simple est d'introduire cet appel de méthode dans la méthode d'accès à l'objet référencé. Les détails d'utilisation sont décrits au chapitre 11 (p. 115)).

6.4 Gestion des index applicatifs

Les bases de données traditionnelles sont généralement accompagnées d'un langage de requête. Cependant, dans notre cas, il est possible de s'en passer. En effet, l'accès aux données peut-être fourni par des index. L'entrepôt fourni déjà un premier niveau d'indexation avec la table de référencement. Cependant d'autres index, propres à l'application, peuvent être gérés pour fournir les points d'accès particuliers aux données et obtenir de bonnes performances. Leur nombre, leur type et leur gestion sont laissés au programmeur de l'application. A titre d'exemple, le tableau ci-dessous décrit les index d'un canal Pharos.

[7] *Dynamic Proxy Class.*
[8] *Getter/Setter.*

Nom	Type	Rôle
`url_annotation`	Tableau associatif	Associe un ensemble d'*annotations* à un URL donné.
`term_annotation`	Tableau associatif	Associe un ensemble d'*annotations* à un *term* donné.
`user_annotation`	Tableau associatif	Associe un ensemble d'*annotations* à un *user* donné.
`re_annotation`	Automate	Permet de retrouver un ensemble d'*annotations* selon une expression régulière appliquée sur les attributs `comments` et `title`.

Le stockage des index dépend du modèle de stockage de l'état courant. Si celui-ci est entièrement reconstruit en mémoire au démarrage, les index peuvent aussi être retablit en même temps. Dans ce cas, les index sont des objets abonnés au service de notification des opérations de l'entrepot. Par contre si l'état courant n'est que partiellement chargé en mémoire, les index ne peuvent plus être construit par notification. L'application a donc la charge de construire et de gérer une représentation persistante *adhoc*.

6.5 Répercution des mutations de l'état courant sur le journal

L'existence de deux versions des données (journal et état courant) nécessite d'assurer leur cohérence mutuelle. Nous venons de voir comment les modifications du journal étaient reportées sur l'état courant. De même lorsque la valeur d'un objet est modifié, il faut répercuter cette modification dans le journal.

6.5.0.1 Détection des mutations

Ce processus est plus délicat que le processus inverse. En effet, l'application n'a, *a priori*, pas besoin de passer par l'entrepôt pour modifier l'état d'un objet. En Java, un objet peut être modifié, soit en invoquant l'une de ses méthodes soit, si la classe de l'objet le permet, en modifiant directement ses attributs. De plus, la destruction d'un objet ne requiert aucune opération particulière de la part du programmeur. Elle est prise en charge par le ramasse-miette de la JVM. Aussi, si l'on souhaitait offrir un mécanisme de cohérence transparent entre les objets et le journal, il serait nécessaire d'introduire des traitements spécifiques dans les mécanismes de création d'objet, d'affectation des attributs objets et dans le ramasse-miette. Pour cela, il faudrait soit intervenir au sein même de la JVM (directement comme par exemple [ADJ+96] ou par le biais d'un *meta-objet protocol* [TC98, OB98, KG98]) ; soit enrichir le *byte-code* des classes par un mécanisme de post-traitement (comme par exemple [LLB+97]).

Ces deux techniques sont contraignantes pour le programmeur et réduisent la portabilité de l'application. Par ailleurs, la transparence de la persistance n'est pas forcement ce qui est recherché par le programmeur.

Nous avons donc pris une approche différente, en imposant que les opérations de création, de mise-à-jour et de destruction des objets gérés par l'entrepôt se fassent par son intermédiaire. Ceci entraîne une programmation adaptée à ce modèle mais offre un contrôle plus fin sur l'enregistrement des opérations dans le journal. Le programmeur peut ainsi décider quelles opérations doivent être validées, à quel moment et si il s'agit d'une seule opération ou d'un regroupement.

6.5.0.2 Traitement des cycles de référencement

L'externalisation des références pose aussi des problèmes. Pour assurer que les références puissent être reconstruites par une lecture séquentielle du journal, il faut qu'une opération référençant un objet soit postérieure à l'opération de création de cet objet. Une solution consiste à produire les enregistrements dans le journal selon un parcours récursif sur les références de l'objet. Comme en Java tout attribut sur un objet est une référence, il faut être capable de distinguer les attributs sur des objets gérés par l'entrepôt (p. ex. l'attribut `user` dans la classe `Annotation`) des autres (p. ex. un objet de type `String`). Pour cela, la solution retenue consiste à typer les classes des objets enregistrés dans le journal. Tout objet géré par l'entrepôt doit posséder le type `Storable`. Le parcours des références ne concerne donc que les attributs de ce type.

Les références doivent être écrites dans l'ordre topologique inverse du graphe des références (cf. figure 6.2). Cependant, l'algorithme du tri topologique suppose qu'il n'y ait pas de cycle dans le graphe des références. Pour faire face à ce cas, nous proposons un algorithme en deux étapes (illustré à la figure 6.2) :

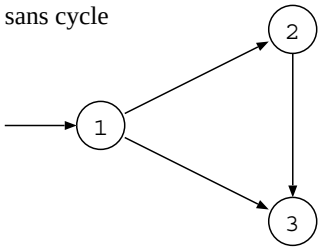
Première étape :
- Le graphe des références est parcouru en profondeur ;
- Les opérations sont écrites dans le journal lors de la remontée ;
- Lorsqu'on visite un objet référençant des objets en cours de visite (i.e. matérialisant des cycles), aucune des références de cet objet n'est enregistrée. Il est ajouté à la liste l des objets à traiter lors de la seconde étape.

Seconde étape :
- Une fois toutes les références visitées, on parcourt l et on enregistre des opérations de mises-à-jour sur ces objets pour leur attribuer les références qui n'ont pas été enregistrées lors de la première étape.

6.6 Conclusions

Dans cette partie, nous avons décrit notre modèle de persistance des données applicatives adapté à la réplication. Il se base sur un journal des opérations

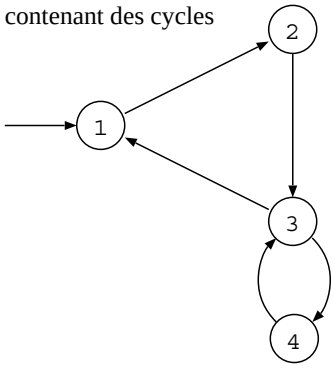
Graphe de références sans cycle



Opérations générées :

```
<node id='3' op='create' next=''>
<node id='2' op='create' next='3'>
<node id='1' op='create' next='2,3'>
```

Graphe de références contenant des cycles



Opérations générées :

```
<!-- first step -->
<node id='4' op='create' next=''>
<node id='3' op='create' next='4'>
<node id='2' op='create' next='3'>
<node id='1' op='create' next='2'>

<!-- second step -->
<node id='3' op='update' next='1'>
<node id='4' op='update' next='3'>
```

FIG. 6.2 – *Externalisation des références*

qui contient l'histoire des mutations des objets persistants de l'application. Le graphe d'objets est reconstitué en lisant ce journal. Des opérations de compensation permettent de rétablir une version antérieure de l'état courant de l'application. Enfin, nous présentons deux approches pour la gestion de l'état courant : soit complètement en mémoire, soi persistant sur disque et partiellement chargé en mémoire.

Le modèle de persistance que nous avons choisi (un journal des opérations et un état courant) est bien adapté pour la réplication mais a ses limites :

1. La lecture d'un long journal au démarrage est coûteuse. Le nettoyage permet d'augmenter les performances. Si l'application a des sessions d'exécution très courte, il peut alors être nécessaire de rendre persistant l'état courant ;

2. Notre modèle n'est pas bien adapté pour des volumes très importants d'objets. A titre d'exemple, sur Pharos chaque création d'annotation consomme environ 500 octets. Un ensemble de canaux Pharos gérant 200 000 annotations, consomme donc 100 Mo. Pour les applications gérant des millions d'objets volumineux (CAO, SIG, ...) la journalisation s'avére coûteuse en espace disque. Ceci devient alors crucial si l'application engendre beaucoup de mutation sur les objets créés ou si les objets ont une durée de

vie très courte. Dans ce cas, le déclenchement du nettoyage et surtout l'utilisation de techniques de compression des opérations peuvent réduire la consomation disque (la compression des opérations des canaux Pharos avec LZ77 a baissé la consommation moyenne d'un facteur cinq) ;

3. Nous avons fait le choix de ne pas être transparent. Le programmeur doit concevoir son application avec notre modèle de persistance. Il doit déclencher explicitement l'enregistrement des mutations dans le journal et gérer les notifications de l'entrepôt. En conséquence, l'utilisation de notre système pour des applications existantes nécessite des modifications dans la gestion des objets. Ceci d'autant plus, qu'en Java la présence du ramasse-miettes libére habituellement les programmeurs de la destruction des objets ;

4. Notre modèle ne propose pas de langage de requêtes. L'application doit donc prévoir tous les index dont elle a besoins pour assurer des accès rapides à ses données. En cas, de persistance de l'état courant, elle doit aussi assurer la persistance de ces index. De même, nous ne fournissons pas de support pour les transactions. Cependant, notre modèle de communication fortement distribué et asynchrone engendre d'autres problèmes pour la gestion des transactions.

Cependant, ce modèle se justifie aussi pour plusieurs raisons :

1. La journalisation des opérations permet de travailler sur les versions successives d'une donnée. C'est un point très important pour le mécanisme de synchronisation que nous avons retenu (cf. chapitre 7, p. 63). Une comparaison des états courants ne peut pas suffire pour une synchronisation efficace des réplicas. En effet, il est souvent nécessaire de connaître l'histoire d'une donnée pour savoir précisément quelles opérations l'ont amenées dans son état courant et ainsi détecter et résoudre les divergences entre deux réplicas ;

2. La persistance par journalisation est bien adapté aux applications serveurs. Elles sont généralement hébergées sur des machines dédiées. Elles peuvent être configurées de manière à gérer l'ensemble des données en mémoire ce qui améliore les performances des consultations. Leur durée d'exploitation est généralement longue (plusieurs jours sans arrêt) ; le coût de la lecture du journal au démarrage de l'application est donc moindre. De plus, en cours d'exploitation, les écritures sont rapides car il s'agit toujours d'ajout en fin de journal (accès séquentiels) ;

3. L'application est robuste. En cas de panne, le redémarrage est immédiat et seule la dernière opération est éventuellement perdue. D'autre part, si des mises-à-jour ou des destructions inopinées ont eu lieu sur certaines

données, elles sont facilement détectables et l'état courant peut être rétabli en annulant ces opérations ;

4. L'application est pérenne. Le choix d'une représentation XML nous rend indépendant d'un format bineaire lié à une architecture. L'application supporte l'évolution de schéma : les attributs supplémentaires sont ignorés à la lecture des opérations ; les attributs absents sont remplacés par leur valeur par défaut. La structuration XML du journal permet une maintenance plus simple : la consultation et la correction du journal peut être faite depuis n'importe quel éditeur de texte ;

5. Les écritures sont traçables. La journalisation des opérations permet de suivre l'évolution d'une donnée. Ces informations peuvent servir pour établir des mesures comportementales de l'application et des utilisateurs ;

6. L'application est autonome. Grâce à la structuration XML du journal, l'application peut l'exploiter tel quel, quelle que soit l'architecture matérielle et le système d'exploitation sous-jacent. Ce point est aussi important pour la réplication. En effet, on doit permettre la réplication des données en faisant le moins d'hypothèse sur l'environnement d'accueil des réplicas.

Troisième partie

Cohérence et intégrité du système

Chapitre 7

Protocole de synchronisation

Pour converger vers un état globalement cohérent, les réplicas échangent leurs nouvelles opérations en se synchronisant. Les échanges se font soit par le réseau soit par un médium transportable (p. ex. : une disquette). Certaines des opérations échangées peuvent être conflictuelles et doivent donc subir un traitement spécial. Le protocole de synchronisation assure la convergence à terme des réplicas vers un état globalement cohérent. La synchronisation d'écritures divergentes peut amener un réplica dans un état cohérent mais non intégre. Le protocole de synchronisation gère donc aussi le contrôle d'intégrité sur les données synchronisées.

7.1 Principes d'une synchronisation

L'objectif d'une synchronisation est de rendre cohérents deux réplicas ayant des états divergents. Ce processus nécessite d'identifier les opérations divergentes et de les intégrer de façon cohérente aux journaux des deux réplicas. Lorsque certaines de ces divergences sont conflictuelles, il faut assurer que les décisions prises pour la résolution des conflits amèneront à un état cohérent sur l'ensemble des réplicas.

7.1.1 Choix des partenaires

Une synchronisation s'effectue entre deux réplicas. Le choix d'un partenaire n'est pas contraint par le protocole de synchronisation. N'importe quel réplica peut se synchroniser avec n'importe quel autre réplica. D'une synchronisation à l'autre, un réplica peut changer de partenaire. L'administrateur du système peut donc choisir la topologie de réplication la mieux adaptée à son cas d'utilisation (en étoile, en arbre, en graphe, ...) Le protocole n'impose pas non plus de contrainte sur la fréquence des synchronisations mais des gardes-fou peuvent être mis en place. Si l'infrastructure de communication le permet, les synchronisations peuvent être fréquentes pour avoir des vues les plus cohérentes possible. Pour les groupes de réplicas intégrant des machines peu fréquemment connectés, on optera pour des fréquences de synchronisation plus relâchées avec la mise en

place de gardes-fous.

Une synchronisation est monodirectionnelle : l'un des deux réplica est l'*émetteur*, l'autre est le *récepteur*. Le récepteur collecte les opérations de l'émetteur et les intègre aux siennes. On dit que le récepteur effectue une *synchronisation entrante* et l'émetteur une *synchronisation sortante*. Les synchronisations bidirectionnelles sont réalisées par le protocole comme une succession de deux synchronisations monodirectionnelles. La synchronisation est un processus exclusif. Durant tout le traitement, le récepteur ne peut plus accepter de synchronisation entrante, ni de nouvelles opérations en dehors de celles collectées. Par contre, il peut effectuer plusieurs synchronisations sortantes simultanément.

Les synchronisations assurent la diffusion épidémique des mises-à-jour [DGH+87]. Durant une synchronisation, un réplica diffuse ses nouvelles opérations mais aussi celles qu'il a reçu des autres réplicas. De proche en proche, par le jeu des synchronisations, une opération est diffusée au reste des réplicas. Une même opération peut donc être collectée sur plusieurs réplicas. Pour réduire le volume des échanges, un réplica doit éviter de collecter des opérations qu'il a déjà reçues. Pour cela, le processus de collecte doit connaître l'état d'avancement du récepteur vis-à-vis des autres réplicas.

7.1.2 Quelques définitions

La synchronisation entraîne la lecture d'une partie du journal de l'émetteur et la modification d'une partie de celle du récepteur. Ces traitements ont lieu sur des zones particulières des journaux. Chacune de ces zones est définie ci-dessous. La figure 7.1 illustre ces définitions.

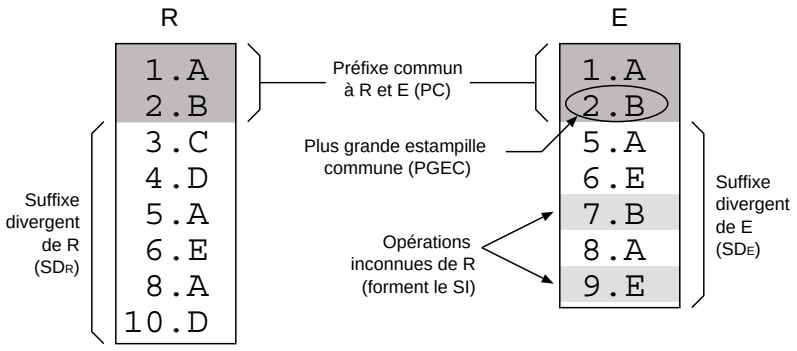


FIG. 7.1 – *Illustration des zones des journaux du récepteur (R) et de l'émetteur (E) entrant en jeu lors d'une synchronisation. Seules les estampilles des opérations (1.A, 2.B, ...) sont représentées.*

Définition 1 *Une **séquence** est un ensemble ordonné d'opérations. L'ordre des opérations est celui de leur estampille.*

Définition 2 *Un* ***segment*** *est un sous-ensemble d'opérations intégrant toutes les opérations d'une séquence (p. ex. un journal) comprises entre deux bornes (au sens des estampilles) .*

Définition 3 *Un* ***segment préfixe*** *d'une séquence est un segment qui contient la plus petite opération de cette séquence.*

Définition 4 *Un* ***segment suffixe*** *d'une séquence est un segment qui contient la plus grande opération de cette séquence.*

Définition 5 *Le* ***plus grand préfixe commun*** *de deux séquences (PC) est le plus long segment préfixe ne contenant que des opérations communes aux deux séquences.*

Définition 6 *La* ***plus grande estampille commune*** *(PGEC) est la plus grande estampille appartenant au préfixe commun (remarque : il se peut qu'il y ait d'autres estampilles plus grandes et communes aux deux séquence mais au-delà du préfixe commun).*

Définition 7 *Le* ***suffixe divergent*** *d'une séquence σ_1 vis-à-vis d'une séquence σ_2 (SD_{σ_1,σ_2}) est le plus grand suffixe de σ_1 ne contenant pas l'opération estampillée avec PGEC.*

Définition 8 *La* ***séquence des opérations de E inconnues de R*** *(SI) est la séquence du journal de E contenant toutes les opérations inconnues de R.*

7.1.3 Algorithme de synchronisation

Selon le mode d'échange choisi, l'un des réplicas est l'émetteur (E) et l'autre le récepteur (R). Dans ce qui suit, par souci de concisions, nous emploierons le nom du réplicas pour désigner son journal. La synchronisation de E vers R est réalisée en exécutant l'algorithme suivant :

1. On détermine le préfixe commun entre R et E puis R collecte les opérations du suffixe divergent de E ($SD_{E,R}$). Certaines opérations de $SD_{E,R}$ peuvent déjà être connues de R. La collecte est donc filtrée pour ne retenir que les opérations inconnues de R. Le résultat forme la *séquence des opérations de E inconnues de R* (SI). Selon le mode de collecte retenue, la collecte et le filtrage se passe sur R ou E ;

2. R détermine et mémorise son suffixe divergent ($SD_{R,E}$). Il restaure le dernier état commun avec E en compensant (*undo*) toutes les opérations de $SD_{R,E}$;

3. R détecte et résout les conflits entre $SD_{R,E}$ et SI ;

4. Une fois les conflits résolus, R intègre les opérations de $SD_{R,E}$ et de SI à son journal ainsi que les résolutions qui ont été prises par la gestion des conflits. Un contrôle d'intégrité est effectué avant l'enregistrement de chacune de ces opérations dans le journal de R. Toute opération enregistrée

est jouée.

5. Si il s'agit d'une synchronisation bidirectionnelle, on effectue le même traitement en inversant les rôles de E et R.

7.1.4 Traitement des rejets

Après une synchronisation bidirectionnelle, les deux réplicas ont atteint un état cohérent[1]. Toutes les opérations collectées sur E ont été intégrées à R, à l'exception des opérations rejetées. Une opération peut être rejetée lors de la résolution de conflits ou lors du contrôle d'intégrité.

La procédure de rejet d'une opération consiste à la retirer du journal et à ajouter une nouvelle opération marquant explicitement ce rejet. L'*opération de rejet* contient en attribut l'estampille de l'opération rejetée. Un rejet ne concerne que les opérations de mutation (création, mise-à-jour ou destruction) et les opérations groupées. Son rôle est d'assurer la suppression cohérente des opérations rejetées dans l'ensemble des réplicas. Une opération de rejet a les mêmes propriétés que les autres opérations du journal (i.e. elles possèdent une estampille) et elle est propagée de la même manière. Lorsqu'un réplica reçoit une opération de rejet, il l'intègre dans son journal et supprime l'opération rejetée.

Définition 9 *La **séquence des rejets** d'une séquence σ (SR_σ) est la séquence contenant toutes les opérations rejetées dans σ.*

7.2 Collecte des nouvelles opérations

L'algorithme utilisé pour la collecte des opérations dépend du type d'échange entre R et E. Si l'échange se fait par le réseau, R envoie une requête de collecte sur E et reçoit en réponse les opérations correspondantes. Si l'échange se fait par un support amovible, E copie l'ensemble de son journal sur le support, et R collecte les opérations en comparant ce journal avec le sien. Le composant chargé de la collecte s'appelle le *collecteur d'opérations* (CO). Selon le mode de collecte utilisé, il s'exécute soit sur E soit sur R.

7.2.1 Collecte par transfert du journal

La collecte la plus simple consiste à recopier entièrement le journal de E chez R. Dans ce cas, le CO opère uniquement sur R en faisant une lecture séquentielle du journal de E. Seules les opérations inconnues de R sont retenues[2]. La première opération de E n'appartenant pas à R définit $SD_{E,R}$. PGEC est

[1] On suppose qu'en cas de conflits ou de violation d'intégrité, les résolutions prises ont été les mêmes sur les deux réplicas

[2] On suppose que l'état courant est géré en mémoire (cf. section 6.2, p. 50). Grâce à ses tables de swizzling, R détermine rapidement si une opération est nouvelle ou pas. Dans le cas d'un état courant persistant (cf. section 6.3, p. 51), il faut alors comparer les deux journaux.

l'estampille de l'opération du journal de E précédent immédiatement cette opération. Le préfixe commun et $SD_{R,E}$ sont déduits à partir de PGEC.

Ce mode de collecte a l'avantage de demander très peu de traitement de la part de E. En particulier, il ne nécessite aucune information en provenance de R. Il convient bien pour des échanges de type *push* ; c'est-à-dire, quand un réplica prend l'initiative d'envoyer ses nouveautés. En contre partie, il consomme d'autant plus de bande passante que le journal est important. Il est donc plutôt adapté à des applications gérant de petits journaux (i.e. inférieur à quelques dizaine de Ko) ou à des échanges par supports amovibles (p. ex. disquettes, bandes magnétiques, ...) Ce dernier cas est intéressant car il ne nécessite aucune infrastructure réseau et supporte des gros volumes de données. Cependant, cette technique est peu efficace et doit être réservée à des synchronisations occasionnelles.

Pour les échanges via le réseau avec ce mode de collecte, une synchronisation se fait en un seul échange (une requête, une réponse) qui peut-être de type *push*, *pull* ou *push/pull*. La figure 7.2 illustre ces trois cas.

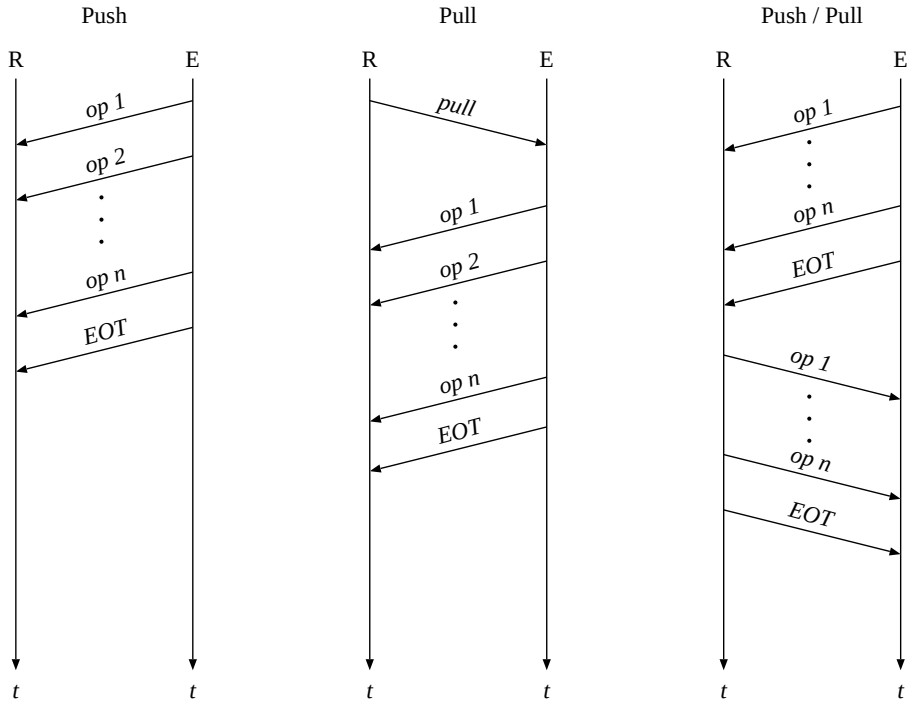


FIG. 7.2 – Type d'échanges lors d'une collecte par échange des journaux

7.2.2 Collecte par indice d'arrivée

Pour réduire la consommation de bande passante, il faut éviter la collecte d'opérations que le récepteur connaît déjà. Pour cela, le CO est exécuté sur

E et dispose d'informations sur le contenu du journal de R. Ces informations sont nécessaires pour optimiser la collecte et réduire le volume des données à transmettre. Pour que le gain soit vraiment positif, il faut que la bande passante consommée pour l'envoi de ces informations au CO soit nettement inférieure à la consommation due à l'envoi des opérations redondantes.

La topologie des réplicas n'est pas imposée. Néanmoins, dans notre cadre applicatif, on peut supposer qu'une partie des réplicas se synchronisera toujours avec le ou les mêmes partenaires. Dans ce type de configuration, l'échange complet des journaux parait encore plus coûteux puisque d'une synchronisation à l'autre les opérations déjà connues seront de plus en plus nombreuses.

La collecte par indices d'arrivée tient compte de ce type de configuration. Chaque réplica connaît l'état d'avancement de chacun de ses partenaires. Ainsi, lors d'une nouvelle collecte, le CO reçoit du récepteur sa connaissance sur l'état d'avancement de E. Il se sert de ces informations pour ne retenir que les nouvelles opérations enregistrées chez E. Ceci change donc le type d'échange. On peut toujours faire du *push* et du *pull* mais pour une synchronisation bidirectionnel on préférera les échanges *pull/push*. La figure 7.3 illustre ces trois cas.

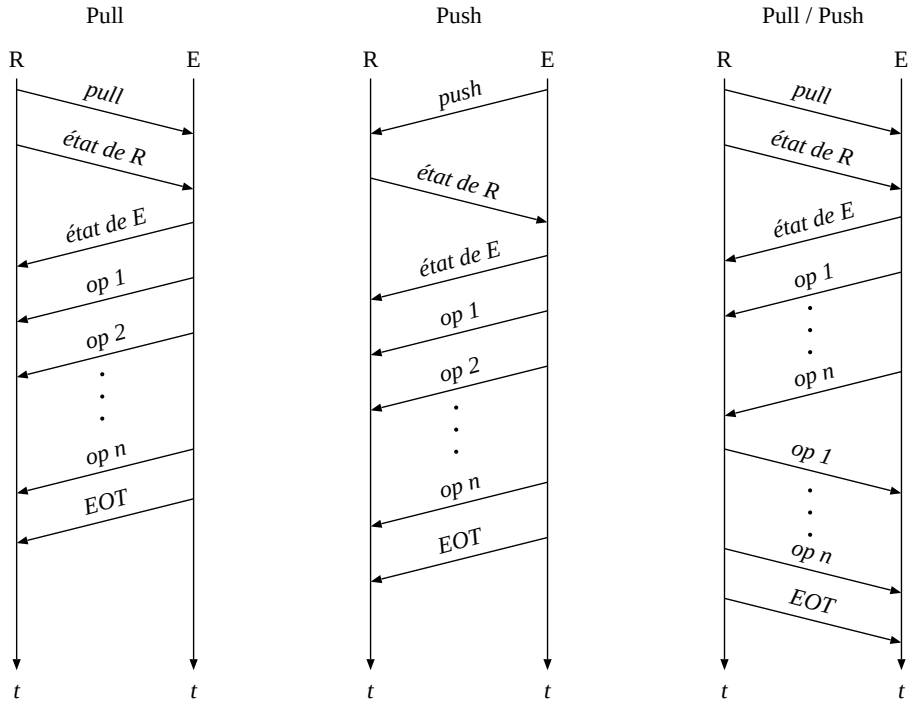


FIG. 7.3 – Type d'échanges lors d'une collecte par indices d'arrivée

L'état d'avancement est calculé en utilisant les indices d'arrivée des opérations. Chaque réplica gère un compteur, l'indice d'arrivée, qui s'incrémente à chaque enregistrement d'une nouvelle opération dans le journal. Chaque opération est indicée par cette valeur. L'indice d'arrivée n'a pas le mêmes rôle que

l'estampille. Il ne sert pas a ordonner totalement les opérations mais seulement à les ordonner selon leur arrivée dans le journal d'un réplica. Chaque réplica maintient une *table des arrivées* (TA). La TA associe à chaque réplica partenaire le plus grand indice d'arrivée déjà collecté.

À chaque synchronisation, R envoie à E le plus grand indice d'arrivée (I_{max}) qu'il connaît de E. L'algorithme du CO est le suivant :

1. Le CO recherche toutes les opérations ayant un indice d'arrivée supérieur à I_{max}. Cet ensemble d'opérations forme *la séquence des opérations de E inconnues de R* (SI) ;

2. L'opération de SI ayant la plus petite estampille, marque le début de $SD_{E,R}$. L'opération précédant immédiatement cette opération dans le journal est la PGEC. En effet, par construction c'est la plus grande estampille connue de E et R et elle précède immédiatement la première opération de $SD_{E,R}$. La PGEC marque la fin du préfixe commun ;

3. Le CO applique un filtre sur SI pour retirer les opérations déjà connues de R ;

4. Le CO envoie à R la PGEC et SI dans l'ordre des indices d'arrivée.

7.2.3 Optimisation de la collecte

Les indices d'arrivées seuls ne permettent pas une collecte optimum (c'est-à-dire exempte d'opérations déjà reçues). Lors d'une synchronisation bidirectionnelle, le récepteur recollecte les opérations qu'il vient d'envoyer à l'émetteur. En effet, ces nouvelles opérations ont été enregistrées sur l'émetteur avec des indices d'arrivée supérieurs à celui connu du récepteur. D'autre part, si depuis sa dernière synchronisation avec E, R s'est synchronisé avec d'autres réplicas, il risque de collecter des opérations nouvelles sur E mais déjà collectées sur les autres réplicas. Enfin, lors de la première synchronisation entrante avec un nouveau partenaire, le récepteur ne connaît aucun indice d'arrivée ce qui provoque la collecte complète du journal de E. On retombe, dans ce cas, sur le mode de collecte précédant.

Pour réduire le volume des données échangé, le collecteur d'opérations intègre un mécanisme de filtrage qui évite l'envoi d'opérations déjà connues à R. Pour cela, il doit disposer d'information supplémentaires sur l'état d'avancement de R vis-à-vis des autres réplicas.

La table des arrivées (TA) ne contient que des informations propre à un réplica. Or, les réplicas sont libre de se synchroniser avec de nouveaux partenaires au cours du temps. L'optimisation de la collecte nécessite donc d'avoir

des informations globales sur les opérations. Considérons l'exemple suivant. Un réplicas A diffuse une opérations O aux réplicas B et C. Un réplicas D se synchronise avec B, il intègre donc O. Lorsqu'il se synchronise avec D, l'optimiseur de collecte doit être pouvoir déterminer que O a déjà été collecté. Or, la TA de D ne contient des informations relatives au réplica B mais pas sur O.

Les *vecteurs de version* [WPE+83] permettent de construire une vue globale de l'état d'avancement des autres réplicas. Chaque réplica gère un vecteur qui contient pour chacun des autres réplicas la plus grande valeur d'horloge d'estampillage qu'il a reçu. Ce vecteur est alimenté au cours des synchronisations, en analysant les estampilles des opérations collectées. En effet, chaque estampilles se compose d'un identifiant de réplica et de la valeur de son horloge d'estampillage (cf. section 8.2, p. 80). L'estampille fournit l'information nécessaire pour savoir qui a produit une opération et à quel moment. Dans l'exemple de la figure 7.1 (p. 64), le vecteur de version du récepteur, V_R, est égal à :

$$V_R \left| \begin{array}{l} A = 8 \\ B = 2 \\ C = 3 \\ D = 10 \\ E = 6 \end{array} \right.$$

Grâce à cette information, on sait que R à déjà reçue l'opération 8 créée par A, l'opération 2 créée par B, ...Il offre donc une vue globale de R vis-à-vis des opérations qu'il a déjà collectée. Cependant pour pouvoir utiliser les vecteurs de version, il faut respecter la propriété préfixe de Bayou [PST+97]. Cette propriété peut s'exprimer ainsi :

$$\forall\, e_i \in R_1 \cap R_2,\ e_j \in R_2\ :\ e_j < e_i \Longrightarrow e_j \in R_1 \tag{7.1}$$

Elle assure que si un réplica R_1 possède une opération estampillée e_i qui a été initialement retenue par un réplica R_2, alors R_1 possède aussi toutes les opérations de R_2 antérieures à e_i. L'information contenu dans le vecteur de version est donc suffisante pour déterminer si une opération doit être collectée ou non.

Dans Bayou, la collecte ne se fait que par les vecteurs de version. Ce mode de collecte est plus simple mais impose de parcourir le journal depuis le début lors de chaque synchronisation, pour repérer les nouvelles opérations. Ceci est pénalisant dans notre cas, où les journaux sont amenés à contenir de nombreuses opérations ne divergeant que vers la fin. D'autre part, comme nous le verrons plus loin, cette technique n'est pas compatible avec le support de la réplication partielle. Notre utilisation des vecteurs de version étant limité à l'optimisation de la collecte, leur gestion peut être adaptée pour continuer à respecter la propriété préfixe lors des réplications partielles.

Le principe du filtrage par les vecteurs de version consiste à ne sélectionner que les opérations ayant une estampille supérieure aux entrées du vecteur de

version de R. Chaque entrée du vecteur est ensuite positionnée à la plus grande estampille reçue. La figure 7.4 illustre ce mécanisme.

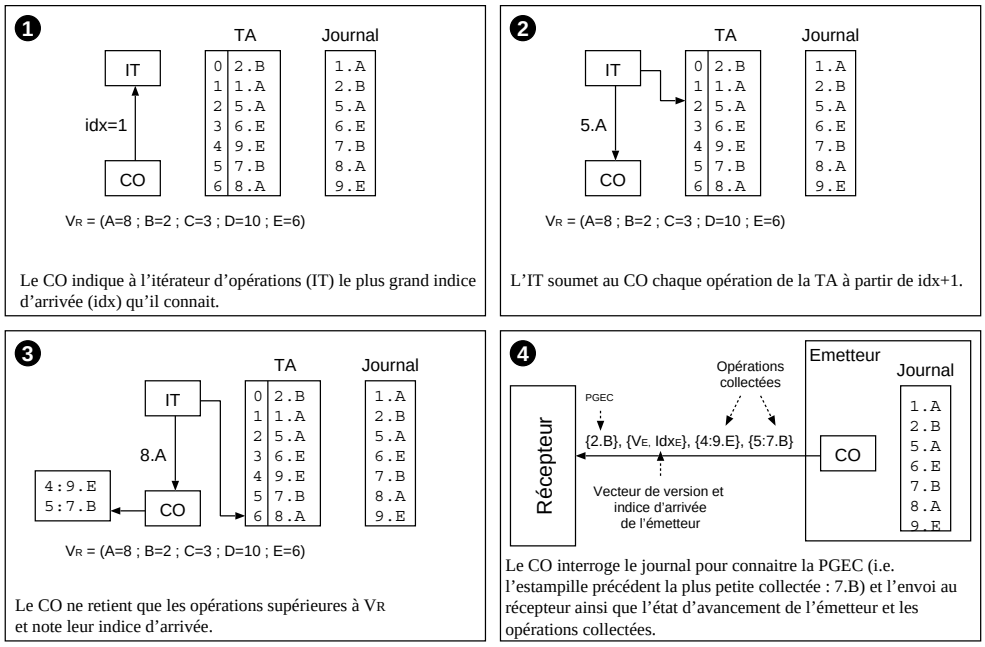


FIG. 7.4 – *Collecte des nouvelles opérations chez l'émetteur.*

En procédant de cette façon, on assure l'incrémentalité des diffusions : même si la connexion est rompue durant la transmission, le récepteur peut intégrer les opérations qu'il a reçu et faire progresser sa TA. Lors de la prochaine synchronisation, le récepteur ne collectera que les opérations non collectées ou en transit dans le réseau au moment de la panne. Il est par contre important de noter que le récepteur ne doit incrémenter son vecteur de version que si toutes les opérations collectées ont été reçues. En effet, les opérations ne sont pas envoyées dans l'ordre de leur estampilles mais dans celui de leur arrivée dans le journal de l'émetteur. Si V_R n'est incrémenté que partiellement, la propriété préfixe peut être rompue ce qui empêcherait la collecte de certaines opérations lors des synchronisations suivantes.

7.2.4 Implémentation extensible

Pour assurer l'extensibilité du mécanisme de collecte, le collecteur d'opération (CO) est implémenté sous forme d'un objet migrant, construit par le récepteur et envoyé à l'émetteur. Il possède toutes les données nécessaires à l'accomplissement de sa tâche. Après avoir réceptionné le CO, l'émetteur le démarre pour qu'il commence sa collecte. La collecte des opérations repose sur le *pattern* Visiteur [GHJV95]. Les entrées de la TA sont successivement présentées au CO qui les traite comme il le souhaite. Cela permet de découpler fortement le CO de la gestion interne de la TA. Une fois la collecte terminée, le CO se

connecte au récepteur pour les lui envoyer. La figure 7.5 illustre ce mécanisme.

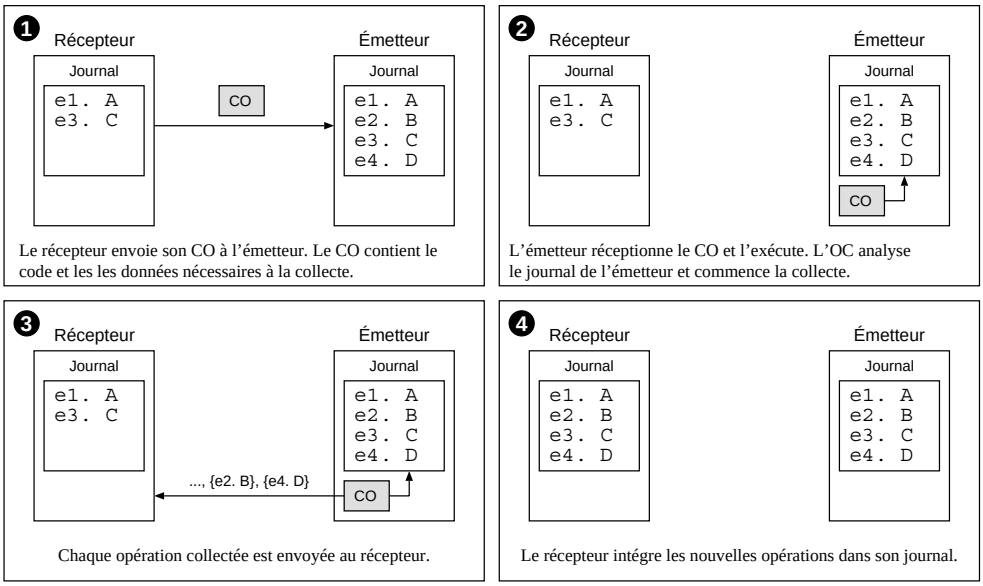


FIG. 7.5 – *Utilisation d'un objet migrant pour la synchronisation entre deux réplicas.*

La présence d'un objet du récepteur chez l'émetteur permet au récepteur d'effectuer, en plus de la collecte, des traitements spécifiques sur les opérations collectées. On peut ainsi envisager le chiffrement des envois pour sécuriser les échanges ou la compression les opérations pour augmenter les performances. Dans les deux cas, le récepteur peut utiliser les algorithmes et les données qu'il souhaite, sans que l'émetteur ait à les connaître.

L'utilisation de code mobile impose de délimiter le domaine d'actions des objets migrants en plaçant des barrières de sécurité. Notre implémentation étant en Java, nous utilisons les mécanismes de sécurité fournies par cette plate-forme. Un `SecurityManager` est mis en place pour limiter l'accès aux ressources du système. Il est aussi possible d'utiliser les techniques de signature de code pour vérifier la provenance de l'objet migrant.

7.3 Intégration de la collecte

Une fois les opérations collectées, R doit les intégrer à son journal et à son état courant. L'intégration ne peut pas se faire simplement en enregistrant les opérations dans le journal et en les jouant sur l'état courant. En effet, les opérations ne sont pas forcement commutatives et elles doivent être jouées dans leur ordre d'estampillage. De plus, pour la gestion des conflits on doit pouvoir comparer les divergences entre R et E. Il est donc indispensable de ramener R dans un état courant correspondant à l'état précédent le début des divergences

avec E ; puis d'intégrer les opérations de E à celles de R et de les jouer.

7.3.1 Principes

On repositionne R à l'état correspondant au préfixe commun qu'il partage avec E. Pour ce faire, on compense l'état courant de R jusqu'à l'opération estampillée par PGEC. R résoud les conflits entre les opérations du suffixe divergent de R, $SD_{R,E}$, et celles du suffixe inconnu, SI (cf. section 8, p. 79). Il intègre toutes les opérations de $SD_{R,E}$, de SI[3] et celles produites par la résolution des conflits dans une nouvelle séquence, NS_R. $SD_{R,E}$ est supprimé du journal de R et remplacé par NS_R. La figure 7.6 représente l'état du journal du récepteur de la figure 7.1 (p. 64) après l'intégration des nouvelles opérations de E.

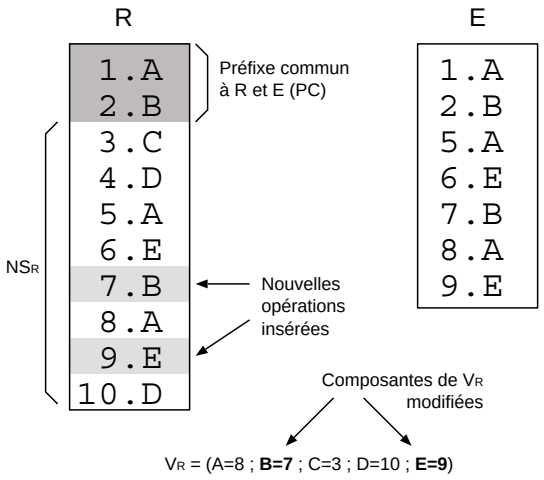


FIG. 7.6 – *Intégration des mises à jour.*

7.3.2 Robustesse du système

L'intégration des nouvelles opérations est une étape sensible pour le récepteur. Si un problème survient, le journal et l'état courant peuvent ne plus être intègres. Il faut donc prévoir qu'à tout moment la synchronisation puisse être interrompue.

Pour gérer la robustesse du système, des procédures de traitement sont mises en œuvre pour rétablir l'intégrité de l'entrepôt du récepteur après la levée d'une exception. Le $SD_{R,E}$ est copié sur un support fiable (i.e. un fichier sur disque) avant d'être modifié (i.e. avant la gestion des conflits). On considère alors trois périodes durant la synchronisation, où une exception peut apparaître :

[3] Certaines opérations de $SD_{R,E}$ et de SI ont pu être rejetées par l'un des algorithmes de résolution des conflits.

1. Si elle a lieu avant que les opérations du $SD_{R,E}$ aient été compensées, le journal et l'état courant n'ont pas été modifiés. L'application peut donc continuer son exécution ;

2. Si elle a lieu après la compensation du $SD_{R,E}$ mais avant la gestion des conflits, l'état courant doit être reconstruit. Comme le journal n'a pas été modifié, toutes les opérations du $SD_{R,E}$ sont rejouées ;

3. Si elle a lieu après la gestion des conflits, le $SD_{R,E}$ a été potentiellement modifié. On remplace toutes les opérations au-delà de la $PGEC$ par la copie du $SD_{R,E}$ et on rejoue ce segment.

Le journal des opérations assure que l'on garde trace de toutes les opérations d'écriture. Cependant, lors de la résolution des conflits certaines opérations peuvent être rejetées. Un rejet est une opération sensible puisqu'il supprime définitivement une opération du journal. Aussi, afin de faire face à d'éventuelles dysfonctionnements ou attaques entraînant des rejets non-justifiés, les opérations rejetées sont supprimées du journal et transférées dans un *journal des opérations rejetées*. Ce journal n'est pas destinée à l'application mais a son administrateur. En cas d'erreur, il peut ainsi réintégrer une opération en la transférant du journal des rejets au journal des opérations. Cette tâche est néanmoins délicate car l'introduction manuelle d'opération dans le journal peut violer l'intégrité des données ou provoquer des conflits.

7.4 Nettoyage du journal

Les objets peuvent être mis à jour de nombreuses fois. Le journal contient chacune de ces mutations. Le rôle du *nettoyage interne* est d'épurer le journal de ces opérations transitoires. Une application peut aussi avoir ses propres critères de nettoyage. Ce *nettoyage applicatif* revient à enregistrer des nouvelles opérations dans le journal (mise-à-jour ou destruction) qui seront prises en compte par le nettoyage interne. Par exemple, dans Pharos, certaines annotations ont une date de péremption. Lorsqu'elle est atteinte, la note d'intéret est diminuée (mise-à-jour de l'annotation) puis lorsque cette note a atteind le minimum, l'annotation est détruite.

7.4.1 Algorithme général

La politique de déclenchement du nettoyage est propre à l'application. Il peut avoir lieu au démarrage de l'application, ou en cours d'exploitation (par exemple, juste avant une synchronisation). L'algorithme de nettoyage effectue une passe de lecture puis une passe d'écriture sur le journal. Lors de la première passe, on détermine, pour chaque objet, la séquence d'opérations qui le concerne. Dans la seconde phase, on simplifie ces séquences. Il existe deux règles de remplacement :

1. Une séquence composée d'une opération de création suivie d'une série de mises-à-jour est remplacée par une séquence composée de deux opéra-

tionss : une création et une mise-à-jour reflétant les différentes mises-à-jour ;

2. Si cette séquence se termine par une destruction, toutes les opérations de la séquence sont retirées du journal.

Il est à noter que dans la première règle, la séquence simplifiée ne peut pas être réduite à la seule opération de création. En effet, il est important de préserver l'ordre des opérations de création pour ne pas briser l'intégrité des référencements entre objets. Considérons la séquence d'opérations suivante :

```
<node stamp="0" id="0" op="create" parent="" />
<node stamp="1" id="1" op="create" parent="0" />
<node stamp="2" id="2" op="create" parent="0" />
<node stamp="3" id="1" op="update" parent="2" />
```

Si on nettoie les opérations 1 et 3 en une seule opération de création, l'opération 1 référence un objet (2) qui n'existe pas encore :

```
<node stamp="0" id="0" op="create" parent="" />
<node stamp="1" id="1" op="create" parent="2" />  <-  référencement non-sequentiel
<node stamp="2" id="2" op="create" parent="0" />
```

La quantité d'opérations à nettoyer est liée au comportement des utilisateurs de l'application. A titre d'exemple, l'analyse des journaux de trois canaux Pharos utilisés quotidiennement pendant 5 mois, comportant respectivement 1100, 400 et 300 annotations et n'ayant pas de nettoyage applicatif, à fait apparaître une certaine régularité sur la répartition des opérations quel que soient les canaux et les types de données : 75-82 % de `create`, 11-20 % de `update` et 4-6 % de `delete`. Chacun de ces journaux contient donc entre 19 et 32 % d'opérations transitoires[4].

Le nettoyage réduit la consommation d'espace disque et diminue les temps de chargement. Il faut cependant faire attention aux conséquences qu'il engendre. Un nettoyage complet du journal appauvrit les informations disponibles et ne permet plus de faire de traitement sur les versions successives d'une donnée. Une solution intermédiaire, consiste à garder toutes les mutations sauf celles sur les objets détruits. Dans ce cas, il n'y a alors plus qu'une seule règle de remplacement : toute séquence se terminant par une opération de destruction est retirée du journal.

7.4.2 Adaptation pour la synchronisation

L'intégration de l'algorithme de nettoyage avec la synchronisation des journaux entraîne certaines modifications. D'une part, l'algorithme doit tenir compte

[4] Une opération de destruction compte double puisque son effacement entraîne l'effacement d'une opération de création.

des données déjà diffusées aux autres réplicas afin de ne pas introduire d'incohérence dans les journaux. D'autre part, il faut assurer qu'une partie des données déjà nettoyées ne soit pas réinsérées dans le journal lors de la réception de nouvelles opérations.

Le nettoyage ne pose pas de problèmes particulier lorsqu'il s'agit de séquences d'opérations qui n'ont encore jamais été diffusées par synchronisation. Ces séquences concernent les opérations enregistrées après la dernière synchronisation. Des opérations spéciales sont introduites dans le journal pour marquer chaque synchronisation entrante et sortante.

Les formes de séquences à nettoyer sont plus nombreuses que dans l'algorithme précédent. Certaines séquences peuvent contenir des opérations de création qui ont été enregistrées avant la dernière synchronisation. Les séquences à nettoyer et leur opération de remplacement sont présentées dans le tableau 7.1.

TAB. 7.1 – Nettoyage des séquences d'opérations non diffusées

Séquence	Séquence de remplacement
`create update+`	Le `create` et le dernier `update` intégrant les différentes mises-à-jour
`create update* delete`	Suppression de cette séquence
`update+`	Le dernier `update` intégrant les différentes mises-à-jour.
`update+ delete`	`delete`

Le nettoyage ne peut pas être réalisé tel quel pour les séquences s'étalant sur plusieurs synchronisations. Si une partie de la séquence a déjà été diffusée, son nettoyage entraînerait des inconsistances. Ainsi, le nettoyage ne peut pas concerner les séquences {`create, update*, delete`} car le modèle de cohérence à terme serait brisé. Par exemple, si un objet est créé sur un réplica R_1, diffusé à un réplica R_2 puis détruit sur R_1 et si la séquence est nettoyée par R_1, R_2 ne recevra jamais l'opération de destruction. Soit l'objet ne sera pas détruit, ce qui empêchera la convergence d'état entre R_1 et R_2 ; soit R_2 détruira ce terme, ce qui entraînera, après diffusion, des incohérences dans le journal de R_1 (destruction d'un objet inexistant). Cet exemple montre que l'on ne peut pas supprimer une séquence de création/destruction complètement ou partiellement diffusée. Ces séquences ne sont donc pas nettoyées. Pour réduire la taille de ces séquences, avant chaque synchronisation, les journaux de l'émetteur et du récepteur sont systématiquement nettoyés.

7.5 Gestion des réplicas

7.5.1 Création d'un réplica

Un réplica est créé à partir d'un autre réplica par clônage. Le fils récupère l'ensemble des données du réplica père. Le réplica père attribue un identificateur à son fils. Cet identificateur est enregistré dans le journal sous forme d'une opération spéciale. Le réplica est alors prêt et peut recevoir des clients.

7.5.2 Identification des réplicas

Chaque réplica possède un numéro d'identification unique (URID). Les seules contraintes imposées sur la construction d'un URID sont d'assurer son unicité dans le temps et l'espace. L'URID sert à identifier les réplicas mais pas à les localiser. Il entre dans la construction des estampilles qui servent à la création des identificateurs des opérations et des données.

Nous proposons la méthode de construction suivante :

```
replica-id = addr : clk
```

L'unicité spatiale est assurée par `addr` qui représente l'adresse de la machine du père. L'unicité temporelle est assurée par `clk` qui représente l'horloge logique du réplica père au moment du clônage (cf. section **??**, p. **??**). L'unicité de `addr` peut-être obtenue, par exemple en prenant le couple {adresse IP, port}[5] (6 octets).

La création d'URID nécessitent des représentations sur plusieurs octets. Chaque estampille contient un URID plus la valeur de l'horloge logique (8 octets). Pour réduire la taille de la représentation des estampilles dans le journal, des identificateurs locaux à un réplica sont attribués à chaque réplica. Grâce à cette technique la taille des journaux est réduite (si la numérotation locale occupe 1 octet, le gain est de $((8+6)-1) \times n_{operation}$ octets ; soit pour 100000 opérations, environs 1,3 Mo). Avant tout échange avec d'autres réplicas, il est nécessaire de reconstruire les estampilles en remplaçant les identificateurs locaux par leurs URIDs.

7.5.3 Localisation des réplicas

Un réplica est localisable par son URL. Un réplica peut changer d'URL au cours du temps. Le format des URLs des réplicas est de la forme :

```
jrep://host:port/<urid>
```

Les URLs des réplicas sont gérés dans le journal. Les changements de localisation sont enregistrés sous forme d'opérations dans le journal. Ainsi, c'est le mécanisme de diffusion épidémique des mises-à-jour qui assure la diffusion de la

[5] On suppose que les réplicas sont hébergés sur des serveurs ayant une adresse IP statique. L'attribution dynamique d'adresse IP (p. ex. le protocole DHCP [Dro93]), ne permet pas d'éviter que les adresses de deux réplicas soit échangées au cours du temps.

nouvelle localisation aux autres réplicas.

Déclaration XML :

```
<!ELEMENT jrep:rurl EMPTY>
<!ATTLIST jrep:rurl
          jrep:stamp      CDATA                  #REQUIRED
          jrep:index      CDATA                  #REQUIRED
          jrep:filtered   (true|false)           false
          jrep:id         ID                     #REQUIRED
          jrep:op         (create|update|delete) #REQUIRED
          jrep:user-id    IDREF                  #REQUIRED
          jrep:urid       ID                     #REQUIRED
          jrep:url        IDREF                  #REQUIRED
>
```

Exemple de représentation dans le journal :

```
<jrep:rurl jrep:stamp="35829:0"
           jrep:index="676"
           jrep:filtered=false
           jrep:id=123
           jrep:op=create
           jrep:urid="45"
           jrep:url="jrep://tif.inria.fr:7878/45"
/>
```

7.6 Conclusions

Dans ce chapitre, nous avons présenté le protocole de synchronisation des réplicas. La synchronisation s'opère entre deux réplicas et a pour but de les amener à un état cohérent. Le choix du partenaire et la fréquence des mises-à-jour ne sont pas contraints par le protocole.

La synchronisation consiste à intégrer les nouvelles opérations de mutations dans chacun des journaux des deux réplicas. L'échange peut se faire par fichier ou par un protocole réseau. Dans ce dernier cas, le volume des échanges est réduit, grâce a l'emploi conjugué des indices d'arrivée et des vecteurs de version.

La cohérence du système en cas de conflits ou de violation d'intégrité est assurer grâce aux opérations de rejet. Grâce à ce mécanisme, on assure la diffusion épidémique des rejets et donc la convergence à terme vers un état cohérent.

Chapitre 8

Gestion des conflits

Entre deux synchronisations, les réplicas peuvent produire des écritures conflictuelles. Par exemple, un terme du thésaurus peut être mis-à-jour différemment sur plusieurs réplicas tandis que sur un autre réplica, il est détruit. Lorsque ces réplicas se synchronisent deux à deux, le récepteur doit détecter que l'une des écritures de l'émetteur entre en conflit avec l'une de ses écritures. Une fois ces opérations identifiées, le récepteur doit prendre une décision de résolution. Celle-ci dépend de plusieurs paramètres comme le type de conflit, les données en jeu, les auteurs des opérations, etc. Néanmoins, quelle que soit l'application il existe des conflits récurrents. Pour ceux-la, nous proposons des mécanismes automatiques de détection et de résolution. Pour les conflits propres à l'application, le système permet au développeur de définir des politiques adaptées au contexte applicatif.

Dans ce chapitre, après avoir abordé les principes généraux de détection et résolution des conflits, nous présentons une technique d'estampillage permettant d'offrir une résolution automatique et « naturelle » des conflits de mises-à-jour concurrentes. Nous étudions ensuite les deux principaux types de conflits et leur gestions, ainsi que la procédure de contrôle d'intégrité des données. Enfin, nous analysons les propriétés globales du systèmes vis-à-vis des politiques de gestion d'intégrité et de leur évolution.

8.1 Principes

La gestion des conflits consiste à détecter les opérations conflictuelles entre le suffixe divergent de R ($SD_{R,E}$) et le suffixe inconnu (SI), puis à les résoudre (cf. figure 7.1, p. 64). Les conflits peuvent porter sur des opérations concernant le même objet ou des objets distincts. Les opérations étant immutables, aucune opération ne peut être modifiée. La résolution se fait en rejetant certaines des opérations et en produisant de nouvelles opérations.

Plusieurs opérations peuvent porter sur le même objet dans les ensembles $SD_{R,E}$ et SI. Il faut éviter de gérer des conflits sur ces opérations transitoires. Par exemple, si un objet a été mis-à-jour plusieurs fois puis détruit, on doit consi-

dérer qu'il s'agit d'une destruction. Pour chacun des ensembles ($SD_{R,E}$ et SI), on construit une *liste des séquences de mutations*. Chacune de ces séquences contient toutes les opérations portant sur un même objet. La détection des conflits se fait en confrontant deux-à-deux les séquences de $SD_{R,E}$ avec celles de SI. Si les deux séquences concernent le même objet on parle d'un *conflit de mutation*, sinon on parle d'un *conflit inter-objets*. Pour ce dernier type, les conflits sont potentiels car seules des opérations concurrentes sur des objets *liés* peuvent amener à un conflit. Dans les conflits inter-objets, on trouve les conflits de référencement (p. ex. dans un canal Pharos, le lien entre une annotation et les termes qu'elle référence ou les relation père-fils entre les termes) et les conflits sémantiques (p. ex. doublon de termes).

Les résolutions se font en éliminant les opérations au sein de $SD_{R,E}$ et de SI et en ajoutant les opérations de rejet associées dans NS_R. Les algorithmes employés peuvent aussi insérer d'autres opérations dans NS_R. Ces nouvelles opérations ne seront donc pas prises en considération dans l'étude des séquences conflictuelles mais elles seront soumises au contrôleur d'intégrité.

8.2 Estampillage logique à incrément temporel

Si on considére une application collaborative qui n'est pas répliquée mais qui utilise la persistance par journalisation, son journal progresse de façon monotone. Les nouvelles opérations y sont ajoutées successivement. Chaque opération peut donc être simplement estampillée avec un simple compteur qui s'incrémente. Ce comportement doit être adapté pour fonctionner dans un environnement réparti. Lors des phases de synchronisation, de nouvelles opérations sont reçues par le réplica et doivent être intégrées au journal. La réplication introduit donc le problème de l'ordonnancement des opérations.

Certaines opérations parallèles peuvent être conflictuelles (p. ex. si un même terme a été mis à jour sur deux réplicas). Il faut alors convenir d'un mécanisme capable de détecter ces conflits. Parmi, les différents mode de résolution, la résolution temporelle (p. ex. retenir la plus récente des deux opérations) est aisément compréhensible des utilisateurs. Ce mode de résolution est liée a l'ordonnancement des opération. Cependant, sa mise en œuvre dans un environnement réparti soulève des difficultés.

8.2.1 Limite de l'estampillage physique

La solution qui consiste à ajouter les opérations reçues directement à la fin du journal peut générer des incohérences. Prenons l'exemple où un réplica R_1 enregistre successivement dans son journal les opérations A et B alors qu'un réplica R_2 enregistre B puis A. Les opérations A et B correspondent à deux mises-à-jour concurrentes et non commutatives du même objet. R_1 et R_2 n'auront pas le même état pour cet objet bien qu'ayant tous les deux reçu les mêmes

opérations. D'une manière générale, ce problème apparaît lorsqu'il a des mutations concurrentes non-commutatives sur un même objet. L'absence d'ordre global sur les opérations non-commutatives entraîne des incohérences sur les réplicas.

L'ordonnancement des opérations ne peut pas directement reposer sur les horloges physiques des machines sur lesquelles sont hébergés les réplicas. En effet, ces horloges ne sont généralement pas synchronisées. Aussi, l'ordonnancement des opérations selon leur date physique d'exécution peut engendrer des incohérences. Considérons la situation suivante : un objet est créé à un temps t_1 sur un réplica R_1, qui enregistre cette opération dans son journal. Lors d'une synchronisation avec le réplica R_2, il lui envoie cette opération. Supposons que l'horloge de R_2 soit en retard par rapport à celle de R_1. Si cet objet est détruit sur R_2 à un temps t_2, avec $t_2 < t_1$, l'opération de destruction sera placé dans le journal de R_2 avant l'opération de création. Si R_2 se synchronise avec un réplica R_3, celui-ci exécutera d'abord la destruction d'un objet inexistant (qui sera ignorée), puis sa création. Les trois réplicas auront des états incohérents bien qu'ayant tous reçus les mêmes opérations.

Plusieurs travaux ont été menés sur l'utilisation des horloges physiques dans les systèmes distribués [AL97, GL93]. Cependant, ces protocoles reposent sur l'hypothèse que les horloges restent approximativement synchrones et, en cas de forte dérives, les propriétés d'ordre ne sont plus assurées. Pour éviter cela, ces travaux supposent la présence d'un protocole de synchronisation des horloges système tel que NTP [Mil92]. Cependant, ce protocole nécessite que les machines soient connectées en permanence pour recevoir les messages de synchronisation. Or, ceci est contradictoire avec notre utilisation de la réplication, dont le rôle est de permettre, entre autre, un fonctionnement en déconnecté.

8.2.2 Limite de l'estampillage logique

Toutes les opérations ne nécessitent pas d'être globalement ordonnées. Par exemple, dans Pharos, les opérations sur les annotations distinctes sont commutatives. Ainsi, deux opérations de création d'annotation pourraient être enregistrées dans n'importe quel ordre sans que cela provoque des incohérences sur l'état global des réplicas. En fait, seules les opérations sur des objets « liés » ne sont potentiellement pas commutatives et doivent donc être ordonnées globalement. Cela concerne des opérations portant soit sur le même objet, soit sur des objets liés par un référencement (p. ex. un terme et une annotation qui le référence), soit enfin sur des objets liés sémantiquement (p. ex. les termes ayant le même nom).

L'ordre causal défini par Lamport [Lam78] est la première relation d'ordre qu'il est nécessaire d'assurer. Deux évènements sont causalement liés, si le second dépend du premier. Appliquée aux opérations du journal, cette relation permet d'assurer que deux opérations portant sur des données « liées » seront

correctement ordonnées dans les journaux de tous les réplicas. Par exemple, si une opération de création d'annotation référence un terme alors l'opération de création de ce terme lui sera antérieure dans le journal.

Lamport définit un moyen simple de construire un ordre causal. Chaque opération est estampillée avec l'état d'une *horloge logique*. Il s'agit d'un simple compteur dirigé par les règles suivantes :

- R1 : Chaque opération enregistrée par le processus P_i est estampillée avec $h_i = h_i + d$ (avec $d > 0$) où h_i représente la valeur de l'horloge logique de P_i.
- R2 : Lors d'une phase de synchronisation du processus P_j vers le processus P_i : $h_i = max(h_i + d,\ h_j)$.

Ces deux règles permettent d'établir une relation d'ordre partiel qui capture la relation de causalité entre les opérations. En classant les opérations dans le journal par ordre croissant d'estampilles, on assure que l'ordre causal sera respecté. Si une opération op_j a été enregistrée après la réception d'une opération op_i, elle sera toujours classée après op_i dans les journaux de tous les réplicas.

L'ordre causal ne permet d'établir des relations entre les opérations qu'en fonction de leurs dépendances. Pour les *opérations parallèles*, c'est-à-dire ayant été produites entre deux phases de synchronisations dans notre cas, Lamport propose d'assurer un ordonnancement total arbitraire basé sur les identificateurs des réplicas. Ainsi, la relation d'ordre total entre deux opérations op_i et op_j est donnée par :

$$\begin{aligned} & op_i < op_j \\ \iff \quad & ((op_i.h < op_j.h) \vee \\ & ((op_i.h = op_j.h) \wedge (op_i.id < op_j.id))) \end{aligned} \tag{8.1}$$

L'ordre causal n'offre aucune prévision sur l'ordre de classement dans le journal des opérations parallèles. Si d est une valeur statique (p. ex. $d = 1$), l'ordonnancement des opérations dans le journal se fera selon la vitesse de production des réplicas. Ainsi, un réplica qui progresse lentement et qui se synchronise que peut fréquement, aura ses opérations classées avant celles de ceux qui en produisent beaucoup. Un tel réplica entraînera, à chaque synchronisation, des repositionnements arrière (ou *rollback*) important dans les journaux de ceux avec qui il se synchronise. La figure 8.1 illustre des synchronisations entre deux réplicas utilisant des horloges logiques à incrément statique. Par ailleurs, l'ordonnancement par horloge logique peut entraîner, lors des synchronisations, des *rollback* profonds dans le journal, qui augmentent les temps de synchronisation et la consommation de ressources systèmes. Une solution à ces deux problèmes serait de pouvoir tenir compte de la réalité temporelle des opérations parallèles.

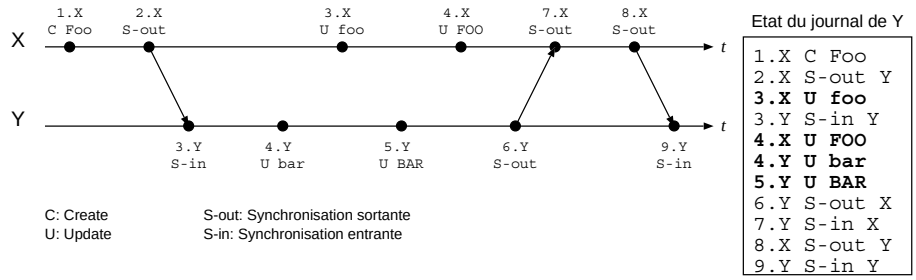


FIG. 8.1 – *Exemple d'estampillage par horloge logique à incrément statique*

8.2.3 Horloge logique à incrément temporel

Nous proposons une nouvelle méthode d'estampillage qui capture l'ordre réel d'occurence des événements (les écritures dans notre cas). Elle permet ainsi d'assurer l'ordonnancement des opérations parallèles proche du temps physique tout en assurant la causalité. Ce modèle se base sur le *temps relatif des opérations*. Comme nous l'avons vu précédemment, les horloges physiques des machines ne sont pas toujours synchrones. Cependant, si l'on ne s'intéresse plus aux valeurs absolues des horloges mais à leur dérive, on constate qu'elles sont généralement assez peu importante. Nous avons mené une petite étude pendant 72 heures sur cinq machines[1] n'ayant pas de support NTP. Il en ressort que les dérives varient d'une machine à l'autre mais elle sont régulières et restent, sur la période d'étude, inférieures à la minute.

Notre méthode d'estampillage se base sur cette propriété et introduit une extension temporelle aux horloges logiques scalaires. Les règles R1 et R2 restent les mêmes, seule la valeur de d est spécifiée par une nouvelle règle :

- R3 : $d = max(1, \Delta_t)$, avec Δ_t représentant l'intervalle de temps physique qui s'est écoulé depuis l'estampillage de la dernière opération.

Quelque soit les variations des horloges physiques, la règle R3 continue d'assurer la progression monotonique des estampilles et donc la causalité des opérations. Pour assurer un ordonnancement temporelle correcte des opérations parallèles, il n'est pas nécessaire que les horloges physiques soient à la même heure. Seules leurs *dérives relatives* entre deux synchronisations doivent rester faibles car la règle R2 assure la resynchronisation des horloges logiques. Ainsi, même si une forte dérive fausse l'ordonnancement temporelle de certaines opérations parallèles, on est assuré que cette défaillance sera bornée par la prochaine synchronisation. En effet, l'horloge logique de chacun des deux réplicas se calera sur celle la plus en avance et l'ordonnancement des nouvelles opérations redeviendra temporellement cohérent[2]. La figure 8.2 reprend l'exemple de syn-

[1] Deux portables sous Windows 95 et 98. Deux stations sous Windows NT et une station sous Linux.

[2] Ceci sous-entend qu'il s'agit d'une synchronisation bidirectionnelle ; sinon seule l'horloge de celui recevant les nouvelles opérations est mise à jour.

chronisation de la figure 8.1 (p. 83) mais avec une horloge à incrément temporel. On peut constater que l'ordre des opérations dans le journal reflète la distribution temporelle des opérations entre les deux réplicas.

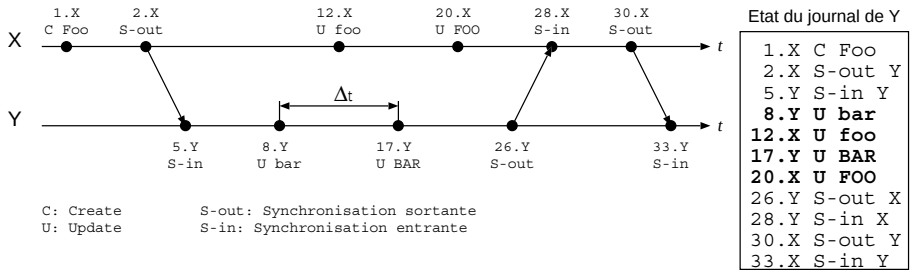


FIG. 8.2 – *Exemple d'estampillage par horloge logique à incrément temporel.*

La résolution avec laquelle sont représentés les intervalles de temps a une importance. En effet, si deux opérations sont enregistrées avec des intervalles de temps inférieurs à cette résolution, l'horloge redevient une pure horloge logique. Dans les systèmes collaboratifs, les opérations étant essentiellements produites par des êtres humains, on peut considérer qu'une résolution de l'ordre de la seconde, voire de la minute, suffit à les distinguer.

8.2.4 Contrainte sur la mise en œuvre

L'inconvénient majeur de cette méthode d'estampillage est d'accélérer la progression de l'horloge logique. Cela n'a pas de conséquence sur les propriétés d'ordonnancement mais pose le problème du stockage des estampilles. En augmentant, de façon raisonnable, la résolution temporelle on peut réduire l'amplitude du phénomène. Par exemple, en prenant une résolution de 5 minutes au lieu d'une seconde, on divise par 300 la vitesse de progression. Cependant, si l'horloge d'un serveur a été avancée de façon importantes, tous les serveurs convergeront à terme vers cette valeur. En particulier, si les horloges sont stockées dans des représentations mémoire de tailles fixes (p. ex. un entier long sur 64 bits), il faut veiller à ce qu'un serveur qui aurait mis son horloge à $2^{64} - \varepsilon$ ne rende impossible la génération d'estampilles, au bout d'un temps ε. Pour faire face à ce problème, on peut soit envisager une représentation extensible des estampilles[3] soit, de façon plus pragmatique, mettre en place des garde-fous afin de refuser les synchronisations avec des réplicas trop en avance (p. ex. plus d'un mois de décalage).

Un autre inconvénient mineur de cette méthode d'estampillage est de produire, potentiellement, des *incohérences externes*[4] les dates applicatives. Lors-

[3] En utilisant une bibliothèque d'entier à précision arbitraire (p. ex. java.math.BigInteger).

[4] Une incohérence externe apparaît lorsque l'ordre des opérations dans un système n'est pas en accord avec ce que l'utilisateur s'attend à obtenir.

qu'un objet gère des informations applicatives de datation[5], il peut y avoir des incohérences entre l'ordre des opérations dans le journal (défini par l'horloge logique à incrément temporel) et les dates applicatives de ces opérations (données par l'horloge système). Si, entre deux phases de synchronisations, l'horloge physique d'un des réplicas est avancée ou reculée, les dates applicatives des opérations enregistrées ne refléterons pas forcement l'ordre de ces opérations dans le journal. Cependant, les utilisateurs n'ont pas accès au détail des opérations et ne perçoivent que l'état courant des données. De plus, une fois les deux réplicas synchronisés, les incohérences externes disparaissent pour les nouvelles opérations.

8.3 Conflits de mutation

Les conflits de mutation concernent des écritures divergentes sur un même objets. Ils sont résolus en premier. En effet, il faut d'abord déterminer quelle sera la séquence définitive d'opérations pour un objet avant de la confronter aux autres objets. Les deux séquences étudiées (celle de $SD_{R,E}$ et celle de SI) portent sur le même objet. Elles peuvent contenir les trois types de mutations. Si on est sûr que l'opération de création ne fait pas partie de $SD_{E,R}$ (car elle est forcement connue de R), rien ne l'empêche d'être présente dans $SD_{R,E}$ et donc potentiellement dans SI (si le filtrage n'a pas été optimum). Cependant, l'opération de création n'intervient pas dans la gestion des conflits de mutation puisqu'il n'y a pas de conflit à son sujet. Tout ce passe donc, comme si les séquences de R ne contenaient que des mises-à-jour (U) ou des destructions (D). On considère alors trois types de conflits de mutation :

1. Conflit de mise-à-jour (U/U) : les deux séquences correspondent chacune à une mise-à-jour ;

2. Conflit de mise-à-jour / destruction (U/D) : l'une des séquences correspond à une mise-à-jour et l'autre à une destruction ;

3. Conflit de destructions (D/D) : il ne s'agit pas véritablement d'un conflit puisque les deux séquences sont équivalentes. Cependant, il est nécessaire de prévoir ce cas pour éviter la présence d'opérations incohérentes dans le journal.

Du point de vue du programmeur, la résolution consiste à choisir l'une des deux *opérations équivalentes* à chacune des séquences (une mise-à-jour ou une destruction). Ce choix peut être guidé par différents critères :

- Le type des opérations ;
- Les dates relatives des opérations ;
- Le rôle de l'auteur de l'opération (p. ex. celle de l'administrateur peuvent passer devant les autres) ;

[5] C'est le cas, dans Pharos, des types de données qui gèrent des dates de création (`cdate`) et de dernière modification (`mdate`).

– Les données des deux opérations.

Nous proposons d'exprimer la résolution des conflits de mutation avec des *matrices de mutation*. Ces matrices sont inspirées des matrices de fusion[6] proposées par Munson et Dewan [MD94]. Il existe une matrice de mutation par classe d'objet. Les lignes sont composées des opérations de R et les colonnes sont composées des opérations de E. Chaque cellule des matrices contient le traitements de résolution. Il est important de noter que la matrice doit être symétrique vis-à-vis du traitement. Ceci est indispensable pour assurer que les résolution seront les mêmes sur chaque réplicas. À titre d'exemple, voici la matrice de résolution par défaut :

Récepteur \ Émetteur	`update`	`delete`
`update`	$U_R + U_E$	D_E
`delete`	D_R	$min(D_R, D_E)$

Cette matrice prévoit qu'en cas de conflit U/U, les opérations des deux séquences soit gardées ($U_R + U_E$). En cas de conflits U/D, on privilégie les destructions (D_R et D_E). Enfin en cas de conflit D/D, on retient la plus petite des deux opérations.

A partir des informations exprimées par le programmeur, le système supprime les opérations correspondant dans $SD_{R,E}$ et SI et ajoute une opération de rejet dans NS_R. On construit une séquence, S, représentant l'union des deux séquences étudiées. On considère trois formes possibles pour S :

1. Conflit de mise-à-jour (U/U).
   Forme de S : U^*

   Résolution par défaut : Aucune opération n'est rejetée. Le conflit sera réglé par l'ordonnancement des opérations. Si les opérations ne concernent pas le même attribut, le résultat intégrera chacune des mises-a-jours. Si il s'agit du même attribut, le résultat reflétera la mise-à-jour ayant la plus grande estampille, c'est-à-dire, à priori, la plus récente.

2. Conflit de mise-à-jour / destruction (U/D).
   Forme de S : U^*DU^*
   Résolution par défaut : si on privilégie les destructions, on rejette les opérations de mise-à-jour postérieures à D. Si on souhaite privilégier les mises-à-jour, seule D est rejetée.

3. Conflit de destruction (D/D) Forme de S : $U * D_1U * D_2$
   Résolution : Il faut éviter la présence d'opération de mutation après le premier D. Aussi, les opérations supérieures à D_1 sont rejetées.

[6] Merge matrix.

Pour les conflits U/D, selon que l'on privilégie les mises-à-jour ou les destructions, les propriétés du système varient. Par exemple, si on privilégie les mises-à-jour sur les destructions, une destruction ne sera stable que lorsque tous les réplicas l'auront reçue et intégrée (cette propriété est élargie dans la section sur les barrières de synchronisation). D'autre part, une mise-à-jour qui a précédée de peu la destruction, peut, selon l'ordonnancement, réapparaître. Le problème existe aussi si on privilégie les destructions mais ce comportement est plus compréhensible pour les utilisateurs. En effet, contrairement à une destruction, une mise-à-jour peut toujours être suivie d'autres opérations de mutation.

8.4 Conflits inter-objets

Une fois les conflits de mutations résolus, on gère les conflits inter-objets. Les conflits inter-objets concernent des écritures divergentes sur deux objets distincts. Ils sont traités en comparant deux à deux toutes les séquences de $SD_{R,E}$ et de SI à l'exception des séquences portant sur un même objet. La méthode de résolution est propre aux deux classes d'objets traités. Le programmeur fournit une méthode de résolution par couple de classes (C_1, C_2) qu'il considère comme potentiellement conflictuelles. L'ordre du couple définit le sens du conflit. La méthode de résolution pour (C_1, C_2) est donc différente de celle de (C_2, C_1)[7]. Si, pour un couple donné, il n'existe pas de méthode de résolution, on suppose que les classes ne sont pas conflictuelles et on ne fait aucune résolution.

De même que pour les conflits de mutation, les méthodes de résolution des conflits inter-objets peuvent être représentées par des *matrices inter-objets*. Les lignes représentent par les opérations sur la classe C_1 et les colonnes les opérations sur la classe C_2. La résolution du conflit consiste alors à retenir l'une des deux séquences ou les deux. De même que pour les conflits de mutation, on compare les deux séquences, au travers de leur opération équivalente. Le tableau ci-dessous défini pour chaque séquence, l'opération équivalente associée :

Séquence	Opération équivalente
`create update+`	`create` intégrant les différents `update`
`create update* delete`	Séquence à ignorer
`update+`	`update` intégrant les différents `update`
`update+ delete`	`delete`

Les matrices inter-objets n'ont pas les mêmes propriétés que les matrices de mutation. Elles ne doivent pas être symétriques (sauf dans le cas, où le conflit porte sur deux objets de la même classe). Dans le cas d'une matrice inter-objets gérant un conflit de référencement, il ne peut pas y avoir de conflit dans la colonne de création. En effet, les conflits de référencement concernent les mises-à-jour et les destruction sur des objets existants.

[7] Sauf dans le cas ou $C_1 = C_2$.

A titre d'exemple, voici la matrice inter-objet utilisé dans un canal Pharos, pour le couple de classes (Annotation, Term) :

		Terme		
		`create`	`update`	`delete`
Annotation	`create`	A+T	A+T	(1)
	`update`	A+T	A+T	(1)
	`delete`	A+T	A+T	A+T

1. Retenir les deux opérations ainsi qu'une opération supplémentaire de mise-à-jour de l'annotation ne tenant plus compte de ce terme.

8.5 Gestion des doublons

Les doublons sont un cas typique de conflits inter-objets. Un doublon apparaît lorsque deux objets du même type ont des valeurs égales. La notion d'égalité entre valeurs est propre à chaque classe d'objets. Par exemple, dans Pharos, deux annotations sont considérées comme égales si elles ont le même `url` et le même `user-id` ; tandis que pour les termes, l'égalité porte sur les attributs `name` et `parent-id`.

La difficulté dans la gestion des doublons vient des références entre objets et de la distribution des traitements. Si les objets doublons ne sont pas référencés par d'autres objets, il suffit de rejeter l'une des deux séquences d'opérations amenant au doublon (p. ex. celle ayant la plus grande estampille). Dans le cas contraire, on ne peut pas supprimer l'un des deux doublons car d'autres objets peuvent en dépendre (c'est le cas, par exemple, d'un terme référencé par une annotation). D'autre part, les objets étant mutables, des doublons peuvent apparaître sur n'importe quel réplica. Il faut assurer que les doublons convergeront, sur l'ensemble des réplicas, vers un identificateur unique.

La solution proposée consiste à gérer des listes d'*identificateurs synonymes*. Chacune de ces listes contient un *identificateur de référence* (Id_{ref}). L'Id_{ref} doit être choisi de façon déterministe. Pour cela, nous prenons le plus petit des identificateurs de la liste. Ces listes sont gérées par l'entrepôt comme les autres objets.

Lorsqu'un conflit de doublons survient, on ajoute une nouvelle entrée dans la table des synonymes. L'opération de création portant sur le synonyme de l'Id_{ref} est rejetées. Les autres opérations de mutation ne sont pas changées. Avant de les jouer, l'entrepôt, vérifie si il existe un identificateur synonyme. Si c'est le cas, l'opération est appliquée sur l'objet de référence. La figure 8.3 (p. 89) illustre ce mécanisme au travers d'un doublon diffusé entre trois réplicas.

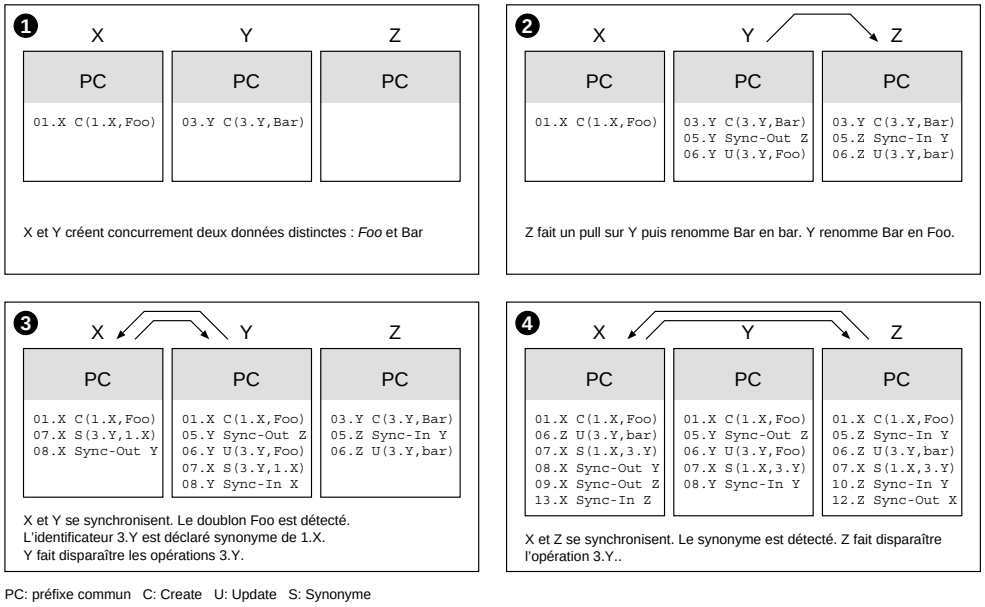


FIG. 8.3 – *Résolution des conflits de type doublon.*

8.6 Contraintes sur les choix de résolution

Les conflits étant résolus par des rejets explicites d'opérations, à terme, tous les réplicas auront reçu les opérations de rejet et atteindrons un état globalement cohérent. Pour deux séquences conflictuelles données, l'état global intégrera donc tous les rejets produits par ces deux séquences sur chaque réplica. Si tous les réplicas traitent ce conflit avec le même algorithme globalement déterministe, tous produiront la même séquence de rejet. Cependant, si l'algorithme de résolution employé n'est pas déterministe, les opérations de rejet seront différentes d'un réplica à l'autre. C'est le cas, par exemple, si le récepteur privilégie ses opérations, ou si l'application demande l'avis de l'utilisateur pour résoudre le conflit. Ceci ne remet pas en cause la propriété de cohérence à terme mais peut perturber le fonctionnement de l'application et sa compréhension par les utilisateurs (potentiellement toutes les opérations des séquences conflictuelles seront rejetées).

8.7 Contrôle d'intégrité

Toutes les opérations de NS_R sont insérées dans le journal de R en remplacement de $SD_{R,E}$. Cependant, certaines opérations peuvent ne pas avoir été détectées comme conflictuelles et, une fois réordonnées dans NS_R, provoquer des violations d'intégrité. Un contrôle d'intégrité est donc effectué avant l'enregistrement de chaque opération. Le contrôleur agit séquentiellement sur les opérations de NS_R en les intégrant ou en les rejetant. Le programmeur peut y intégrer les règles de contrôle d'intégrité propre à la sémantique de ses objets.

Par défaut, le contrôleur d'intégrité ne rejette que les opérations concernant la création d'un objet déjà existant (même identificateur) ou les mises-a-jour et les destructions d'un objet inexistant (ou déjà détruit).

8.8 Dynamicité du système

Le déploiement d'un système à large échelle pose immédiatement les problèmes de sa maintenance et de son administration. Parmi ces problèmes, se pose celui de l'évolution des algorithmes employés et de leur paramétrage. Généralement, ces évolutions ne peuvent pas se faire de façon synchrones car cela exigerait l'arrêt complet de toutes les écritures sur tous les réplicas. Si on accepte des évolutions asynchrones du système, pendant la phase de diffusion des évolutions, chaque réplicas aura perçu une partie des évolutions. Il est donc intéressant de connaître le comportement du système durant cette période où, potentiellement, chaque réplica se comporte différemment.

Lorsque l'un des algorithmes de gestion de conflit ou de contrôle d'intégrité est mis-à-jour, son comportement diffère. Pour les mêmes séquences d'opérations, il produira des rejets différents. À terme, ces rejets étant véhiculés par des opérations explicites, ils seront tous appliqués. Durant toute la période de transition, la cohérence à terme reste garantie mais au prix d'une augmentation du nombre de rejet. Il convient donc de s'assurer que cette période soit la plus courte possible.

8.9 Conclusions

Nous avons opté pour modèle de réplication optimiste. Aussi, les mutations étant réalisées concurremment, des conflits peuvent apparaître lors de la synchronisation. Dans ce chapitre nous avons présenter les techniques pour détecter et résoudre ces conflits et préserver l'intégrité des données lors des synchronisations.

Les conflits sont classés en deux catégories : les conflits de mutation et les conflits inter-objets. Les premiers concernent des mutations divergentes sur le même objet. Leur détection est automatique et un algorithme de résolution par défaut est proposé. Grâce aux propriétés temporelles d'une nouvelle technique d'estampillage qui assure un ordonnancement globale cohérent avec le temps physique, les conflits de mises-à-jour peuvent être résolus « naturellement » par un simple ordonnancement. Les confits inter-objets concernent des mutations divergentes de deux objets liés par référencement ou sémantiquement. Leur détection est automatique mais leur résolution est laissée à l'application. Une fois les conflits traités, un contrôle d'intégrité est appliqué juste avant l'enregistrement des opérations dans les journaux.

Chapitre 9

Extensions au système

Le protocole de synchronisation que nous venons d'étudier, couplé avec la gestion des conflits et le contrôle d'intégrité, fournit un système homogène pour assurer la convergence à terme des réplicas. Dans le cadre normal d'utilisation, l'application et ses données sont répliqués en totalité chez chaque participant et ceux-ci se synchronisent régulièrement. Nous présentons dans ce chapitre quatres extensions qui permette d'adapter le système pour des cas particulier d'utilisation.

9.1 Gestion des réplicas obsolètes

Le protocole de synchronisation n'impose aucune contrainte sur le choix des partenaires ni la fréquence de leur synchronisation. Il peut donc arriver que dans un groupe de réplicas, l'un d'entre-eux se synchronise après une longue absence (par exemple, pendant plusieurs mois). Si, durant cette période, il a enregistré de nouvelles opérations, à sa prochaine synchronisation le nombre d'opérations à compenser sera important. D'autre part, ces opérations peuvent provoquer des conflits avec des opérations que les utilisateur pouvait considérer comme stables car bien antérieures à la période moyenne des synchronisations.

Le rôle des *barrières de synchronisation* est d'assurer la stabilité d'une partie régulièrement croissante des journaux, en refusant d'intégrer de trop vieilles opérations. Pour cela, elles se basent sur la dimension temporelle des estampilles pour estimer la date des opérations. Elles assurent que les opérations antérieures à une certaine valeur ne seront plus rejetables. Les barrières de synchronisation peuvent donc s'apparenter à une validation temporelle des opérations.

9.1.1 Barrières de synchronisation

Les barrières de synchronisation agissent en refusant les synchronisations avec de trop vieux réplicas. Un réplica R refuse une synchronisation entrante avec un réplica E, si la plus petite opération ou si l'une des opérations rejetées collectée sur E a une estampille inférieure à sa barrière de synchronisation (BS).

BS est une valeur temporelle relative à l'horloge d'estampillage (H). La collecte est donc refusée si l'une des opérations du suffixe inconnu (SI) ou du suffixe divergent de E ($SD_{E,R}$) à une estampille inférieure à ($H - BS$).

Si E veut néanmoins se synchroniser avec R, il doit effectuer une *procédure de réintégration*. Durant cette procédure, E abandonne son journal au profit de celui de R puis tente de réintroduire ses opérations perdues (avec de nouvelles estampilles). Il y a donc un risque que certaines de ses opérations soit perdues. Ce risque est d'autant plus important que les objets concernés ont subit des mutations ou qu'ils possèdent des liens avec d'autres objets.

9.1.2 Procédure de réintégration

La procédure de réintégration se fait en quatre étapes :

1. E fait une demande de réintégration vers R. Il reçoit en retour le journal de R (J_R) ;

2. E détermine le préfixe commun (PC) des deux journaux et calcule le SI. Il compense toutes les opérations postérieures au PC. Parmi les opérations du SI, il insère dans un ensemble trié, B_R toutes celles qu'il a produit. Il génére dans l'ensemble R_R des opérations de rejet pour toutes les opérations de B_R qui ont été diffusées[1] ;

3. E intègre toutes les opérations de $SD_{R,E}$ ainsi que tous les rejets de R_R ;

4. E tente une récupération des opérations placées dans B_R. Chacune des opérations est réestampillée puis soumise au contrôleur d'intégrité avant d'être insérée dans le journal.

Une fois réintégré E peut se synchroniser avec R. La procédure de réintégration est coûteuse et potentiellement destructrice. Elle doit n'être utilisée qu'exceptionnellement. La barrière de synchronisation doit donc être choisie avec soin.

9.1.3 Choix de la barrière de synchronisation

La valeur de la barrière de synchronisation (BS) va déterminer le comportement du système. Une BS trop large (p. ex. 6 mois) diminue son intérêt. Au contraire une BS trop courte (p. ex. 48 heures), laissera peu de chance aux réplicas nomades (p. ex un utilisateur parti en déplacement avec un réplica). Choisir des valeurs de BS différentes sur chaque réplica est, a priori, un mauvais choix. Le réplica ayant la BS la plus petite l'imposera au reste du groupe (en rejetant toutes les opérations que les autres BS avaient laissées passer). De plus, les réplicas avec des BS plus large devront potentiellement effectuer une procédure

[1] Il est inutile de rejeter des opérations qui n'ont été vues que par E.

de réintégration pour se synchroniser avec les réplicas ayant de plus petites BS. Il est donc préférable, pour une meilleure compréhension du système global que tous les réplicas aient la même BS.

La topologie joue un rôle important dans le choix de la BS. Par exemple, considérons une topologie qui se compose d'un groupe de réplicas qui se synchronisent régulièrement (p. ex. toutes les 24 h) et de réplicas satellites (des portables) qui se synchronisent plus ou moins fréquemment avec le groupe (au maximum tous les 15 jours). La BS est positionnée à 30 jours sur les réplicas du groupe pour éviter qu'un réplica satellite, trop longtemps déconnecté, introduise des opérations trop anciennes. Le réplicas satellites n'ont pas de BS. Le *temps maximum de propagation* (TMP) dans le groupe est de 48 h. Il s'agit du temps maximum que met une opération introduite dans le groupe pour être reçue par tous les autres membres du groupe. Si un réplica satellite diffuse à un réplica du groupe une opération vieille de 29 jours elle sera acceptée et pourra continuer à se propager pendant 24 h. Si au bout de ce temps, certains réplicas ne l'ont pas reçu, le groupe va être scinder en deux. Tous les réplicas qui l'auront intégrée vont devenir obsolètes vis-à-vis des autres et devrons donc tous subir une procédure de réintégration. Pour éviter cela, l'administrateur du système doit intégrer le TMP dans la gestion des BS. Une solution consiste à gérer deux BS sur chaque réplicas du groupe : une pour les membres du groupe (BS_G) et une pour les réplicas satellites (BS_S). BS_S reste positionnée à 30 jours mais BS_G doit être positionnée en-dessous de $30 - TMP$ jours. Ainsi, même si une opération rentre dans le groupe à la limite d'acceptation, on est assurée qu'elle ne provoquera pas de scission. Dans le cas extrême, les réplicas du groupes peuvent se faire entièrement confiance, et ne pas utilisé de BS entre eux.

9.1.4 Calcul du temps maximum de propagation

Le calcul du temps maximal de propagation (TMP) est fortement lié à la topologie des synchronisations et à leur fréquence. TMP ne peut être déterminé que si l'on connaît la *période maximum de synchronisation* (PMS) de chaque réplica avec ses partenaires. Pour simplifier, on considère que les synchronisations sont bidirectionnelles et que λ_s est identique pour tous les partenaires sur tous les réplicas. La complexité du calcul de TMP est alors liée à la complexité de la topologie. TMP est le maximum des TMP de chaque réplica concerné (i.e. dans l'exemple précédent on ne considérerait pas les réplicas satellites). Pour chaque réplica R on calcul son TMP. Soit λ_s la plus grande PMS du groupe :

1. Topologie en étoile : $TMP_R = 2 \times \lambda_s$

2. Topologie en arbre : soit h la hauteur de l'arbre : $TMP_R = 2 \times h \times \lambda_s$

3. Topologie en graphe : Soit p_i le plus court chemin pour se synchroniser avec le réplica i (exprimé en nombre de nœud à traverser). Soit P le plus long des p_i. $TMP_R = P \times \lambda_s$

9.1.5 Propriétés

L'élargissement d'une BS ne prendra effet que lorsque tous les réplicas auront intégré cette valeur. Durant toute la période de transition, c'est la plus petite BS qui continuera à prévaloir (car ce sera celle qui sera en mesure de provoquer le plus de refus de synchronisation). À l'inverse, une réduction de BS, prend effet immédiatement et peut provoquer des refus de synchronisation avec les réplicas qui ont encore l'ancienne valeur. La réduction doit donc être inférieure au temps de mises-à-jour de tous les réplicas. Néanmoins, comme ces procédures ont pour conséquence des rejets, là aussi, la cohérence à terme sera préservée mais aux dépens des opérations enregistrées durant la période de transition.

9.2 Réplication partielle

Lorsque le volume d'information devient très important, il peut-être coûteux de répliquer la base de données en totalité. C'est par exemple le cas si le réplica se trouve sur une machine ayant peu d'espace de stockage (p. ex. un PDA). La réplication partielle consiste à ne répliquer qu'une partie des données selon certains critères (p. ex. date, auteur, valeur, ...) Ces critères sont exprimés au travers d'un *filtre de réplication* qui est utilisé lors des synchronisations. D'une synchronisation à l'autre, les critères de filtrage peuvent changer.

Les filtres de réplication doivent agir lors de la collecte. En effectuant le filtrage du côte de l'émetteur, seules les opérations réellement utiles pour le récepteur sont transmises. La réplication partielle réduit donc, en plus de l'espace disque, la consommation de bande passante et les temps de traitement du côté du récepteur. Du point de vue de l'architecture, le filtre peut être intégré au collecteur d'opérations avant son départ vers l'émetteur (cf. section 7.2.4, p. 71).

9.2.1 Contraintes

L'introduction de la réplication partielle influence le modèle de cohérence. L'état globalement cohérent de l'ensemble des réplicas $\mathcal{R}$ est atteint si et seulement si :

$$\forall\, i, j \in [0, |\mathcal{R}|[:\ f_j(R_i) = f_i(R_j) \tag{9.1}$$

où, R_i représente l'état du réplica i et f_i la composition de tous les filtres de réplication qu'il a appliqué.

La réplication partielle remet aussi directement en cause la propriété préfixe (cf. équation 7.1, p. 70) qui s'exprime alors ainsi :

$$\forall\, e_i \in R_1 \cap R_2,\ e_j \in f_1(R_2)\ et\ e_j < e_i \Rightarrow e_j \in R_1 \tag{9.2}$$

où la fonction f_1 représente la composition des filtres qui ont été appliqués par le réplica R_1 lors des synchronisations avec R_2.

La remise en cause de la propriété préfixe nécessite d'adapter l'utilisation des vecteurs de version. En effet, si un réplica R_1 se synchronise avec un réplica R_2 ayant répliqué partiellement les opérations d'un réplicas R_3 alors R_1 ne recevra qu'une partie des opérations de R_3 même si il a un filtre de réplication plus large que celui de R_2. Ceci brise le critère de cohérence car certaines opérations de R_3 ne pourront plus jamais être collectées par R_1 du fait que celui-ci aura mis à jour sont vecteur de version lors de sa synchronisation avec R_2. Toute les opérations non-collectées et antérieures aux opérations collectées par R_2 ne seront pas collectées par R_1 lors d'un synchronisation avec R_3. La figure 9.1 (p. 95) illustre cette situation.

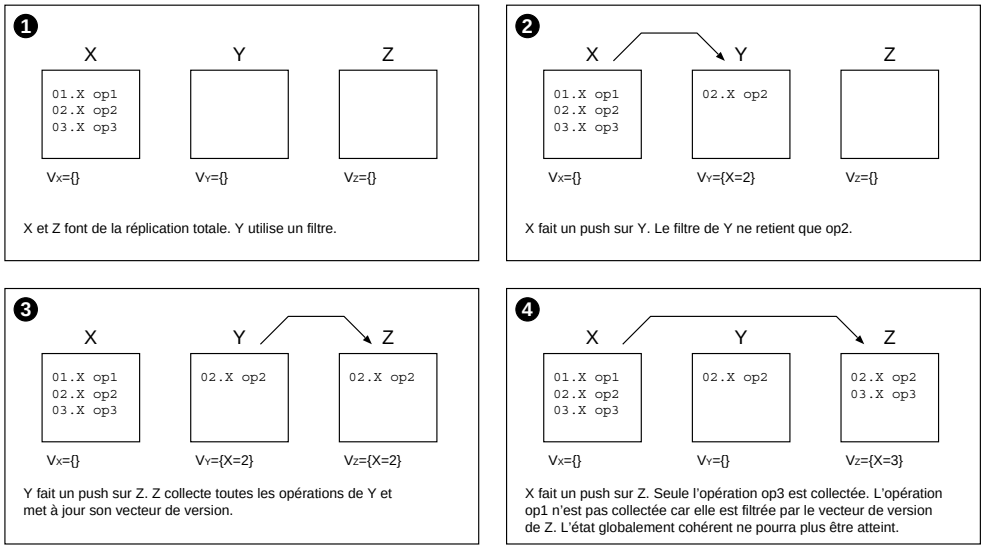


FIG. 9.1 – *Exemple d'incohérence due à l'utilisation des vecteurs de version sur des opérations filtrées.*

9.2.2 Filtrage de la collecte

Pour faire face à ce problème, nous proposons de modifier la gestion des vecteurs de version en ne faisant confiance qu'aux opérations qui n'ont pas été filtrées. Toute opération est enregistrée dans le journal avec l'attribut `filtered` qui indique si, directement ou indirectement, elle a été collectée avec filtrage. Lors de la réception des nouvelles opérations d'un réplica, seule les opérations non-filtrées sont prises en compte pour la mise-à-jour du vecteur de version. On est assuré, par transitivité, que pour une opération non-filtrée, le réplica possède toutes les opérations antérieures en provenance du réplica sur lequel a été créée l'opération.

La synchronisation avec réplication partielle n'est possible que parce que la collecte des opérations ne se base pas uniquement sur les vecteurs de version mais aussi sur la table des arrivées d'un réplica (cf. section 7.2.2, p. 67). Il en résulte que c'est seulement lors des changements de partenaires que le CO risque de collecter des opérations déjà reçues. Ceci ne remet pas en cause la cohérence à terme (ces opérations seront simplement éliminées par le récepteur à leur arrivée) mais augmente la consommation de bande passante. Cette augmentation est néanmoins limitée par le filtrage lui-même. Enfin, l'avantage de ce mécanisme est de supporter des changements de filtre de réplication sans que ne soit remis en cause les principes de synchronisation.

9.2.3 Propriétés

L'ajout de filtres peut perturber l'utilisation de l'application. Lorsqu'une machine est utilisée par de nombreux réplicas pour se synchroniser, la mise en place de filtre de réplication sur cette machine pivot peut entraîner un retard de convergence pour les autres réplicas. Lorsqu'un réplicas élargit son filtrage, il doit abandonner l'utilisation des barrières de synchronisation. En effet, à la prochaine synchronisation, le réplicas collectera d'anciennes opérations qui étaient jusqu'à présent filtrées. La mise en place d'une BS entraînerait des procédures de réintégration abusives.

9.3 Synchronisation d'autres sources de données

Le protocole que nous venons de décrire permet de synchroniser des applications collaboratives gérant leur données sous forme d'objets Java. Cependant, le protocole de synchronisation n'est pas lié à un langage objet en particulier ni même à un modèle de données. Il peut même être utilisé pour synchroniser des sources de données hétérogènes. Dans cette section, nous étudions comment il pourrait être adapté pour la synchronisation d'un SGBDR.

9.3.1 Contraintes d'utilisation du protocole

Le protocole de synchronisation ne gère que des opérations. Les opérations de mutation et de groupement (cf. section 5.4.4, p. 45) sont les seules qui sont à la charge de la source de données à synchroniser. Celle-ci doit donc être en mesure de gérer un journal des opérations ayant les même propriétés que celui décrit à la section 5 (p. 37). La gestion du journal entraîne des contraintes sur le type de données et sur leur gestion.

Les données à synchroniser doivent pouvoir subir les trois types d'opérations de mutation (création, mises-à-jour et destruction). Chaque opération doit respecter le modèle d'ordonnancement défini à la section 8.2 (p. 80). La construction des opération de mises-a-jour nécessite de calculer des différences d'état pour produire des mises-à-jour partielles (cf. section 5.4.2 (p. 42)). Enfin, chaque opération doit être accompagnée d'une opération de compensation. Ce

type de contrainte est donc plutôt adapté à des données structurées.

La source de données doit être en mesure de détecter et de produire les opérations de mutation. Elle doit ensuite être capable d'appliquer une séquence d'opérations sur ses données. Cela nécessite de pouvoir déterminer à partir de l'opération la donnée concernée et d'appliquer la mutation sur cette donnée.

9.3.2 Synchronisation d'un SGBDR

Comme nous l'avons vu à la section section 3.3 (p. 25), certains SGBDR possèdent leur propre mécanisme de réplication. Cependant, ce n'est pas le cas de tous et pour ceux offrant un tel support, les techniques mises en œuvre peuvent ne pas convenir. D'autre part, il peut être intéressant de découpler le mécanisme de synchronisation du SGBDR afin de pouvoir synchroniser des SGBDR de différents vendeurs.

Le type de données géré par un SGBDR correspondant tout à fait aux contraintes que l'on vient de définir. Le calcul des mises-à-jour partielles peut être réalisé par des procédures génériques utilisant les meta-données sur les tables gérées. Le calcul des compensations ne pose pas de difficulté et peut se faire avec la procédure décrite à la section 5.4.3 (p. 43).

La capture des mutations se fait à l'aide de *trigger*. Il s'agit de procédures qui sont associées à une table pour réagir à certains évènements. Dans notre cas, il faut écouter les évènements d'écriture (`insert`, `update`, `delete`) sur les tables contenant les données à synchroniser. Les triggers construisent et enregistrent l'opération de mutation correspondant au type d'écriture. La représentation des opérations suit le même schéma que pour les objets Java (cf. section 5.4.1 (p. 41)). Le nom de balise XML est construit d'après le nom de la table. Par contre, on utilisera l'identificateur du tuple plutôt qu'un nouvel identificateur calculé.

Grâce au mode de représentation des opérations, l'exécution d'une opération ne pose pas de problème majeur. A chaque opération, une requête SQL est exécutée. Elle est construite à partir des constituants de l'opération (nom de la table, identificateur, type d'opération et attribut de l'opération).

9.4 Réplication chez les clients

Certaines applications collaboratives reposent sur un modèle client/serveur. Le protocole de synchronisation assure la propagation des nouvelles opérations entre les serveurs. Cependant, lorsque les données du serveur sont fréquemment utilisées par les clients, il est pertinent de vouloir aussi les répliquer côté client. Si les client sont toujours connectés au même serveur via un réseau local, notre protocole de synchronisation entre serveurs est lourd, coûteux et inadapté. Dans cette section, nous présentons un second modèle de réplication pour les client

d'un serveur répliqué.

9.4.1 Synchronisation par mise-à-jour différées

Les données répliquées sont transitoires. À chaque connexion, les clients fournissent un filtre de réplication et le serveur leur envoie une copie. Le serveur à ensuite la responsabilité de maintenir cohérentes les copies des clients.

Pour supporter un grand nombre de clients, il est important de réduire la consommation de ressource et la charge du serveur. La diffusion des mises-à-jour à l'initiative du serveur nécessiterait soit d'utiliser une infrastructure de diffusion *multicast* soit de contacter successivement tous les clients pour les informer. Dans le premier cas, les protocoles de multicast ne permettent pas d'assurer la fiabilité de la diffusion pour un nombre important de participants [2]. Dans le second cas, le serveur doit maintenir ou établir des connexions avec chaque client. Enfin, les deux techniques ne sont pas facilement déployable à large échelle où la présence de *firewall* bloque les diffusions en direction des clients.

La solution retenue consiste à différer les mises-à-jour et à laisser l'initiative de la diffusion au client. Lors de sa connexion, le client reçoit le plus grand indice d'arrivée produit par leur serveur (`pgia`). A chaque requête au serveur, le client envoie en plus des données de la requête le `pgia`. Cette information permet au serveur de connaître la position du client et de lui retourner, en plus des données de la réponse, les opérations qu'il n'avait pas encore reçues. Le client intègre ces opérations à son état courant et met à jour son `pgia`.

Lorsque le client fait une écriture, elle n'est pas appliqué en local. L'opération est transmise au serveur et n'est appliquée localement qu'au retour de la réponse. Si cette opération entre en conflit avec une autre opération déjà reçue par le serveur mais pas encore perçue par ce client, le serveur refuse l'opération, informe le client et lui renvoie les nouvelles opérations.

9.4.2 Gestion des synchronisation

Entre deux requêtes d'un client, le serveur peut effectuer une synchronisation entrante. Dans ce cas, à la prochaine requête du client, il ne suffit plus de lui envoyer les opérations supérieures à `pgia`. La synchronisation a pu entraîner des réordonnancements qui doivent être répercutés sur le client. Le serveur doit donc invalider les opérations déjà reçues et concerné par cette synchronisation. Puis les lui renvoyer avec les nouvelles opérations dans l'ordre de leur estampille.

A chaque requête, le serveur détermine, à partir du `pgia`, si il y a eu une synchronisation entrante ; c'est-à-dire si il existe une opération marquant cette synchronisation et ayant un indice d'arrivée supérieur à `pgia`. Si c'est le cas, il

[2] Typiquement les protocoles multicast à large échelle reposent sur UDP.

détermine son estampille, e_s. Il envoie au client les opérations de compensation des opérations inférieures à `pgia` et ayant une estampille supérieure à e_s ; puis toutes les opérations supérieures à e_s.

9.5 Conclusions

Dans ce chapitre, nous avons présenté des extensions au protocole de synchronisation. Pour éviter qu'un réplica, resté trop longtemps désynchronisé, provoque de nombreux conflits, des barrières de synchronisation peuvent être mises en place. Elles permettent de refuser une synchronisation avec un réplica contenant des opérations trop anciennes. Dans ce cas, une procédure est proposée pour réintégrer ce réplica et récupérer ses opérations.

Les environnements disposant de peu d'espace de stockage ou d'une connexion à faible bande passante peuvent effectuer des réplications partielles. Dans ce cas, lors d'une synchronisation, les opérations échangées sont filtrées pour ne garder que celles qui sont utiles. Deux réplicas avec des filtres de réplication peuvent néanmoins se synchroniser. Ce mode de fonctionnement est possible grâce au mécanisme de collecte par indice d'arrivée.

Nous avons montré que le protocole de synchronisation pouvait être utiliser indépendamment de tout langage. Il permet de synchroniser toutes sources de données capables de produire d'interpréter un journal des opérations. On peut ainsi envisager des synchronisation entre des entrepôts d'objets de différents langages (Java, C++, SmallTalk) et des bases de données relationnelles hétérogènes.

Enfin, pour la réplication des applications client/serveur, on peut augmenter la performance en répliquant une partie des données chez les client. Pour ce second niveau de réplication, nous proposons un protocole tirant partie de la réplication des serveurs. Le modèle de cohérence proposé se base sur la notification différée à l'initiative des clients. En déchargeant le serveur et en renversant les flux de notification, on assure un meilleur passage à l'échelle du système.

Chapitre 10

Réplication des modules applicatifs

Le déploiement d'une application répliquée pose rapidement le problème de sa maintenance. Comme pour tout système distribué, il faut pouvoir intégrer des correctifs et des évolutions fonctionnelles de l'application tout en assurant un certain niveau de fonctionnement du système. De plus, dans le cas d'une application répliquée, les réplicas ne sont pas toujours joignable au moment du changement de version de l'application. Le système doit donc pouvoir déployer de façon asynchrones les mises-à-jour sur les réplicas qui le compose.

10.1 Maintenance des applications répliquées

Le problème de la maintenance des applications distribuées s'est posé avec l'avènement des application client/serveur. Outre les problèmes de rupture de fonctionnement durant la mise-à-jour du serveur, il faut surtout arriver à déployer les mises-à-jour sur les postes clients. Dans une architecture large échelle (au-delà d'une centaines de client), ce déploiement devient très coûteux. C'est pour cette raison, que l'arrivée de protocoles standards comme HTTP et HTML ont fait émerger un *client universel* : le navigateur. Le problème du changement de version est alors reporté aux évolutions de HTTP et HTML (qui se font à priori moins fréquemment que les évolutions des fonctionnalités de l'application). Cependant, dans ce modèle, toute les traitements restent localisés sur le serveur ; le client ne gère que l'interface Homme-machine.

Dans le cadre d'une application client/serveur répliquée, il faut non seulement assurer la maintenance des clients mais aussi celles des serveurs. Or, la réplication servant à rendre les serveurs autonomes, on ne peut pas se satisfaire d'un déploiement synchrones d'une nouvelle version. Le système de mise-à-jour de l'application doit être en accord avec le mode de réplication : les mises-à-jour doivent se faire de façon asynchrones.

La réplication du code de l'application n'est pas la seule difficulté. En effet, une application est rarement réduite à un simple fichier exécutable. Bien souvent, elle est accompagnée de fichiers de configuration, de données persistantes,

de fichiers pour l'IHM (page HTML, images, ...), etc. La mise-à-jour complète de l'application est coûteuses, d'autant plus si la majorités de ses ressources n'évoluent pas d'une version à l'autre. Seules les parties de l'application ayant subit des mises-à-jour devraient être modifiées. Cela nécessite donc de pouvoir identifier ces parties.

Le passage d'une version à l'autre peut soit se faire à l'initiative de l'application en retard (modèle *pull*) soit à celle de l'application possédant la nouvelle version (modèle *push*). Dans notre cas, comme les réplicas sont autonomes, ils ne peuvent être mis-à-jour que lorsqu'ils se reconnectent. C'est entre autre le cas lorsqu'ils synchronisent leur données. La mise-à-jour des versions est donc réalisée par une phase de négociation juste avant la synchronisation de deux réplicas.

La mise-à-jour d'un serveur pose le problème de la disponibilité durant la durée du changement de version. Si on est capable d'identifier et de mettre à jour uniquement les services concernés par le changement de version, les autres services peuvent continuer à s'exécuter. Par exemple, un serveur Pharos héberge plusieurs canaux. Lorsque l'un des canaux subit un changement de version (p. ex. parce que la structure des annotations a été changée), les autres canaux doivent continuer à fonctionner. Cela nécessite donc à la fois de pouvoir distinguer les services du serveurs mais aussi de permettre le changement de version à chaud d'un service (i.e. sans redémarrer l'application).

La modularisation du serveur permet de matérialiser les services et les gérer indépendamment. Le démarrage et l'arrêt dynamique de chacun des modules fournissent les fonctionnalités de base pour la mise-à-jour *à chaud* d'un module. L'application peut intégrer des modules effectuant des traitements intéressants pour d'autres modules. A titre d'exemple, les canaux Pharos dépendent d'un module commun d'identification des membres. Dans ce cas, il devient indispensable de gérer les relations de dépendances qui existent entre les modules. En effet, certaines modifications sur un module peuvent entraîner le redémmarage voire la mise-à-jour des modules qui en dépendent.

10.2 Outils pour la gestion d'applications modulaires

Les projets de recherche et les produits industriels pour la construction d'applications modulaires sont nombreux. Cependant, le sujet est vaste et chacun d'entre-eux l'aborde sur un point particulier. Ils se distinguent par les fonctionnalités qu'ils offrent pour la gestion des modules, le niveau de granularité des modules, et la gestion de leur dépendances.

10.2.1 Composition d'application

Dans ce cadre applicatif, la modularisation permet de construire une application par assemblage de constituant élémentaire réutilisable. L'assemblage peut se faire soit par des langages de description soit par des outils visuels. On

peut citer par exemple JavaBeans[Jav96] qui est un *framework* pour définir et assembler des composants Java. Les JavaBeans communiquent par envoi d'événements. Un JavaBean est défini par ses propriétés et les événements qu'il gère. L'assemblage consiste à relier un JavaBean producteur d'événements e avec un JavaBean consommateur d'événement e. Bien qu'ils ne soient pas restreints aux application graphiques, la grande majorité des JavaBeans sont des composants graphiques destinés à être assemblés dans des IDE[1] visuels.

De nombreux travaux ont aussi été menés sur la composition d'application à base de composants hétérogènes reliés entre-eux par un langage d'interconnexion (MIL[2]). Parmi eux, le projet Aster[IB96] propose un système de configuration destiné à faciliter la construction d'applications distribuées ayant différentes exigences vis-à-vis de l'environnement d'exécution. L'application est construite à partir de composants ayant la meilleure adéquation entre les contraintes données et les comportements recherchés. Avec des objectifs assez proches, le projet Olan[BAKR96] vise à fournir des outils répondant à deux besoins : (i) construire des applications réparties en combinant des techniques de programmation à base d'objets et des techniques d'intégration de composants ; (ii) faciliter l'administration, la configuration et l'évolution de ces applications. Comme pour Aster il propose un MIL pour décrire les composants et leur relations. Ceux-ci dialoguent par un bus logiciel construit au-dessus de CORBA.

10.2.2 Inter-opérabilité entre modules

Plusieurs protocoles existent pour faire inter-opérer des composants. Ceux-ci sont donc souvent rattachés à la littératures des systèmes modulaires mais n'apportent cependant peu de solutions au déploiement et la composition dynamique d'application. Corba est une norme de l'OMG pour permettre l'inter-opérabilité entre composant écrit dans des langages différents. Pour cela, il fournit un langage de description d'interface des composants indépendant du langage (IDL). RMI[WRW96] est le protocole standard pour faire communiquer des objets Java. Il ne permet donc pas l'inter-opérabilité avec d'autres langages mais est relativement simple à déployer. Par contre, il offre peu de support pour la gestion des évolutions des objets distribués.

10.2.3 Déploiement de modules

Pour faire face aux problèmes d'évolution du code client, plusieurs solutions ont été proposées. Les applets Java permettent d'utiliser le navigateur Web comme client universel. Le code du client est téléchargé à le demande dans le navigateur. Il n'y a donc plus de problème d'évolution des versions car le code client est géré sur le serveur. Cette approche est intéressante mais ne résout pas tous les problèmes. En particulier, le code étant toujours géré par le serveur, à

[1] *Integrated Development Environment*
[2] *Module Interconnection Language.*

chaque fois que la page est accédé, il faut télécharger l'applet, celle-ci n'étant mise en cache que durant la session du navigateur. D'autre part, une applet s'exécute en mode graphique et ne repose pas sur les standards d'interaction et de navigation communément utilisés sur le Web. Enfin, les applets n'ont pas été conçues pour inter-opérer ; une application ne peut donc pas être composée de plusieurs applets.

Les systèmes à agents peuvent aussi être considérés comme des applications modulaires. Ces systèmes se caractérisent par des *places* dans lesquelles s'exécutent des *agents*. Les agents ont la particularité de pouvoir migrer de place en place. Les fonctionnalités des systèmes à agent sont variés. Dans les implémentations telles que les Aglets[LO98] ou Mole[SBH96], le système assure l'accueil de l'agent, son exécution et le renvoie vers une autre place. L'envoi de place en place se fait en ramenant l'agent dans un état stable (en invocant une méthode avant son envoi) et en transmettant sa classe et son état. Des systèmes plus élaborés, tels que Telescript[Whi94] ou JoCAML[Fes99], sont capables de transférer l'agent en cours d'exécution (migration du contexte d'exécution) et offre une plus grande autonomie des agents (déplacement à la demande de l'agent, rendez-vous entre agents, ...). Un agent peut être considéré comme un module que l'on charge, décharge et transfert d'un serveur à l'autre. Cependant dans notre cas, les modules doivent être persistants d'une session à l'autre. Or, dans les systèmes à agent, la durée de vie d'un agent dans une place est généralement courte. Les agents ne sont donc pas persistant. De même, ils ne laissent généralement pas de trace dès qu'ils quittent une place. Les places ne leur offre aucun support pour la persistance de leur données. Enfin, les agents sont généralement de petit objet autonomes. Seul les systèmes offrant un mécanisme de rendez-vous permettent d'établir des relations entre agents. Cependant, même dans ce cas, ils supposent que le code de ces agents n'évolue pas (les interfaces restent toujours compatibles).

Les systèmes de mise-à-jour automatiques d'application entrent aussi dans le cadre des applications modulaire. Le format OSD[3] permet de décrire une version d'un logiciel ou d'un de ces composant ainsi que le graphe de ses dépendances. Ce format peut être utilisé pour mettre à jour automatiquement un logiciel soit à la demande (mode *pull*), soit à l'initiative du distributeur du logiciel (mode *push*). Le produit Marimba Castanet[Mar98] repose sur le modèle *push* en proposant un système de déploiement sur la métaphore des émissions radio. Le système se compose de deux entités : les canaux et les récepteurs[4]. Une application est placée par son distributeur dans un canal. Un client s'abonne à cette application en séléctionnant ce canal sur son récepteur. À chaque nouvelle version de l'application, le distributeur la dépose sur le canal. Les récepteurs sont régulièrement informés par les canaux qu'ils écoutent des nouvelles versions des applications. Pour économiser la consommation de bande passante, le système

[3] *Open Software Description*
[4] *tuner*

assure des mises-à-jour incrémentale avec le protocole DRP[5] [vHGH+97] (il calcul les différences entre deux version et n'envoie que le *patch* correspondant). Le système est intéressant car il permet de centralisé le déploiement d'une nouvelle version d'une application. En contre partie, la diffusion des informations sur les nouvelles version par *multicast* est pénalisant lorsqu'il y a de nombreux clients et que ceux-ci ne sont connectés que ponctuellement.

10.2.4 Gestionnaires de modules

Bien que les services présentés ci-dessus offrent tous un mécanisme plus ou moins élaboré de gestion de module, ce n'est pas leur fonction première. Il existe des systèmes dont c'est le rôle. EJB[6] [Jav] est un *framework* pour construire de serveurs modulaires. Bien que le nom soit lexicalement proche des *JavaBeans* ces deux systèmes ont peu de chose en commun si ce n'est le langage Java. Les JavaBeans servent à composer visuellement une application, les EJB permettent de construire, de déployer et d'administrer des application de type serveur. Les modules EJB sont représentés par des composants Java et sont hébergés dans un conteneur. Celui-ci offre différents services tels que la localisation d'un autre EJB, la gestion de session client, le support des transactions ACID[7], ou la persistance des données. Chaque EJB est accompagnée d'un descriptif de déploiement qui définit son environnement d'exécution (les autres EJB dont il dépend, son modèle de transaction, de persistance, de sécurité, ...) Cependant, les conteneurs d'EJB n'offrent pas de support pour la mise-à-jour *à chaud* d'EJB ni de gestion des versions.

Jini[AWO+99] est une initiative de Sun visant à simplifier l'administration de périphériques sur un réseau local (machines, imprimantes, systèmes de stockage, scanners, ...) L'objectif est de fournir une certaine intelligence à chacun de ces appareils pour les rendre aisément inter-opérables. Par exemple, le simple raccordement d'une imprimante au réseau doit permettent à tous les autres appareils d'être informés de la présence de cette nouvelle ressource et de pouvoir immédiatement l'utiliser. Pour cela, chaque appareil possède un composant (écrit en Java) qui lui permet de dialoguer avec les autres. Jini offre l'infrastructure pour rechercher un service (i.e. appareil) particulier, enregistrer un nouveau service, être notifié de l'arrivée ou du départ d'un service, etc. Le modèle de gestion des services offre de nombreuses fonctionnalités mais n'est pas conçu pour être intégré au sein d'une application. Les services représentent essentiellement des composants physiques. Le mode d'interaction avec ces composants n'est pas le même qu'avec les composants logiciels. Par exemple, Jini peut détecter l'arrêt d'un service mais il n'est pas à l'origine de cette action (c'est le débranchement physique d'une imprimante qui marque la disparition de ce service d'impression).

[5] *Distribution and Replication Protocol*
[6] *Enterprise JavaBeans.*
[7] *Atomicity, Consistency, Isolation and Durability*

10.3 JPlug : plate-forme de composition dynamique

Les systèmes que nous venons de voir offrent chacun une partie des besoins de définition et de gestion de modules. Cependant, leur utilisations pour la modularisation des applications répliquées n'est pas forcement adaptés. Dans cette section, nous décrivons la plate-forme de composition modulaire conçu pour ce type d'environnement.

10.3.1 Structure d'un module

La granularité d'un module est propre à l'application. D'une manière générale, il s'agit d'un ensemble de ressources réalisant une fonction particulière. Un module peut lui-même être composé de plusieurs sous-modules ou coopérer avec d'autres modules. D'un point de vue pratique, un module est écrit en Java[8] et regroupe plusieurs classes.

Un module se compose d'un fichier de description et d'un ensemble de ressources. Le fichier de description permet de connaître les caractéristiques du module :

- Nom et version du module ;
- URL du site de distribution de ce module ;
- Description du module ;
- Nom de la classe de démarrage ;
- Nom et version des modules dont il dépend ;
- Nom du module générique pour les modules utilisant l'héritage de module ;

Un module peut être distribué par différents protocole (p. ex. HTTP ou FTP). Une URL identifie le site de distribution du module pour toutes ses version. À cette URL ce trouve le fichier d'historique du module. Il indique les versions et leurs URL de téléchargement.

10.3.2 Cycle de vie d'un module

Le cycle de vie d'un module au sein de l'application se décompose en sept étapes. La figure 10.1 illustre ces étapes et leur enchaînement. Après avoir localisé le module dont elle a besoin, l'application le téléchage et l'installe.

Un module se présente sous forme d'une archive de fichiers qui contient toutes les ressources nécessaires à son exécution. Cette archive est installé dans un répertoire dédié de l'arborescence de fichier de l'application. D'une version à l'autre le contenu sera remplacé par les ressources de la nouvelle archive. Pour permettre aux modules de préserver des données d'une version à l'autre, un autre répertoire dédié à ce module est créé dans une autre branche de l'arborescence. Le module peut y enregistrer toutes les données qu'il souhaite. Contrairement

[8] Une classe Java est nécessaire pour la gestion du module. Le reste du module peut être écrit dans n'importe quel langage intégrable à Java : soit sous forme d'une librairie C, soit sous forme d'un script interprété (Python, TCL, Perl, Scheme, ...).

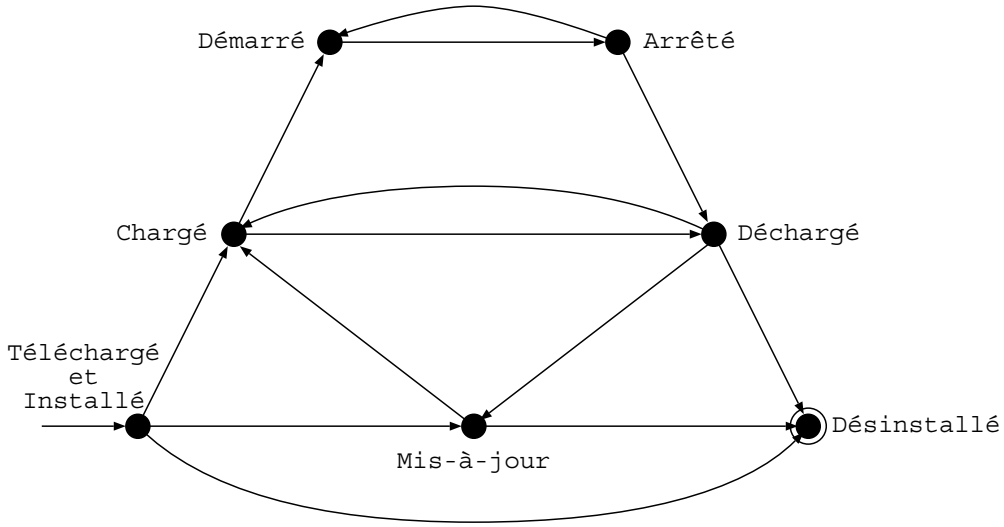


FIG. 10.1 – *Cycle de vie d'un module.*

au précédent, ce répertoire n'est pas effacé lors d'une mise-à-jour du module. Le système garantit au module que l'accès à un fichier, se fait d'abord dans ce répertoire puis, si il n'a pas été trouvé, dans le répertoire d'installation. Ce mécanisme permet de fournir des fichiers par défaut spécialisables par le module. La figure 10.2 illustre cette organisation à travers l'application Pharos. Le répertoire `channels` contient les archives installé des canaux ; le répertoire `channels_data` contient les fichiers produits par les canaux (la structure des annotations, les données du canal et les préférences si elles ont été modifiées). Lors de la mise-à-jour d'un des canaux, seul le répertoire contenu dans `channel` sera effacé.

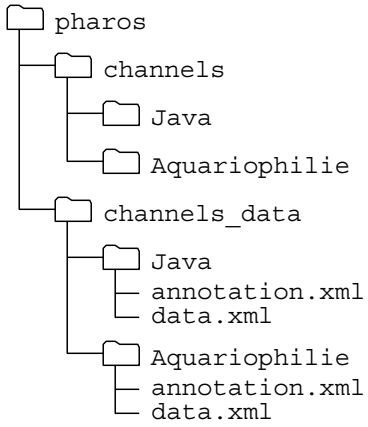


FIG. 10.2 – *Exemple d'arborescence de modules.*

Une fois installé, le module est disponible pour l'application. Pour l'utiliser l'application doit commencer par le charger. Durant la phase de chargement, les dépendances sont résolues ce qui provoque le chargement (éventuellement

précédé de l'installation) des modules concernés. Une fois chargé, le module est enregistré dans la liste des modules actifs et la méthode d'initialisation du module est invoquée. Le module est alors actif. A tout moment le module peut être arrêté puis redémarré. Cela se traduit par un invocation aux méthodes `start` ou `stop` du module. Cependant, aucun mécanisme préemptif n'est déclenché, c'est au développeur du module d'implémenter un comportement correct dans ces deux méthodes.

Lorsqu'un module est arrêté, il peut être déchargé. Dans ce cas l'instance du module est retiré des modules actifs et ses classes sont déchargées. Avant de décharger un module l'application doit vérifié qu'aucun autre module présent n'en dépend. Lorsque le module est à nouveau rechargé, ses classes le sont aussi et une nouvelle instance est créée. Ce mécanisme est intéressant car, en phase de mise au point, il permet de tester facilement des modifications sur un module sans avoir à relancé toutes l'application et tous les autres modules.

Pour la mise-à-jour d'un module, le système consulte le fichier d'historique du module. Si la version installée correspond à la dernière version de ce module, aucune mise-à-jour n'est nécessaire. Par contre, si une nouvelle version existe, le module actuel est desinstallé, la nouvelle version est téléchargée, installée puis chargée. La nouvelle version peut accéder aux fichiers enregistrés par l'ancienne version. Si des changements de structure de ces fichiers ont eu lieu entre ces deux versions, les adaptations sont laissées à la charge du module.

10.3.3 Gestion des versions

Un module est identifié par son nom et son numéro de version. Ce dernier permet d'identifier précisément une implémentation particulière du module. Le numéro de version est incrémenté à chaque nouvelle distribution d'un module. Il se compose de deux nombres entiers : le majeur et le mineur. Le mineur est incrémenté à chaque nouvelle version. Le majeur n'est incrémenté que lorsque l'interface ou la structure des données du module change. Cette distinction permet à un module de continuer à coopérer avec autre un module même si celui a reçu des correctifs.

10.3.4 Chargeur de modules

Le chargeur de module a pour rôle de charger les classes d'un module, d'instancier la classe de démarrage et d'invoquer la méthode d'initialisation. L'instance est ensuite enregistrée dans l'annuaire des modules chargés par le système. Le module est prêt, il est démarré en invoquant la méthode `start` sur la classe de démarrage.

Le langage Java[AG98] offre un mécanisme spécialisable pour le chargement

des classes. Une application Java peut utiliser plusieurs chargeurs[9] pour charger différents groupes de classes. Chaque classe Java est représenté par un fichier particulier qui contient le byte-code de cette classe. Dans Java, le chargement des classes est dynamique : une classe n'est chargée que lors de sa première instanciation. Si la classe chargée créé dans son constructeur des instances de classes absentes, celles-ci seront d'abord chargées. Cependant, le chargement implicite de toutes ces classes se fera avec le chargeur de la classe qui a déclenché ces chargements. Il en va de même pour toutes les classes utilisées par les instances de cette classe. Java cloisonne les classes selon leur chargeur. Ainsi, les classes chargées par un chargeur C_1 ne peuvent pas accéder aux classes chargées par un chargeur C_2 si elles ne disposent pas d'une référence sur C_2.

Une classe ne peut être chargée qu'une seule fois par le même chargeur. D'autre part, Java n'offre aucun support explicite pour le déchargement de classe. Les déchargements se font toujours à l'initiative de la machine virtuelle, selon une règle très stricte. Une classe (ou une interface) peut être déchargé si et seulement si son chargeur est inaccessible (i.e. toutes les instances du chargeur, ont été collectées par le ramasse-miettes). Les classes chargés par le chargeur initial ne peuvent pas être déchargées.

Pour le chargement des classes d'un module, le système instancie un nouveau chargeur dédié à ce module. Cependant, comme les modules peuvent interagir entre-eux, il est nécessaire de permettre l'accès aux classes et aux interfaces gérés par d'autres chargeurs. Pour cela, le système gère l'ensemble des chargeurs en étoile. Lorsque le chargement d'une classe par le chargeur du module échoue, on interroge le chargeur central qui tente successivement de la charger avec les chargeurs des autres modules. Au déchargement d'un module, son chargeur est retiré du système, pour permettre le déchargement des classes. Si le module est à nouveau chargé (p. ex. après une mise-à-jour), le système instancie un nouveau chargeur et charge les classes à travers celui-ci. Grâce à cette organisation des chargeurs, on peut préserver l'indépendance des modules tout en permettant la coopération.

10.3.5 Dépendances entre modules

Certains modules réalisent des traitements dont peuvent bénéficier les autres modules. Par exemple, dans Pharos, l'identification des utilisateurs est partagée par tous les canaux. Lorsqu'un canal doit identifier un utilisateur (p. ex. pour vérifier ses droits), il interroge le module d'identification. Un module interagit avec un autre par invocation de méthodes sur son instance. Il obtient cette instance en interrogeant l'annuaire des modules avec le nom et le numéro majeur de version du module recherché.

Le système garantit à un module que les modules avec lesquels il interagit

[9] *ClassLoader.*

sont présents. Pour cela, le module exprime ses dépendance dans son fichier de configuration en lisant les modules dont il dépend. Au chargement d'un module, le système calcul le graphe des dépendances de ce module en analysant les fichier de description des modules concernés. Si un cycle de dépendance est détecté, le chargement est interrompu. Le système fait un tri topologique sur le graphe et charge les modules dans cet ordre.

10.3.6 Héritage de modules

Dans certains cas, les comportement des modules peuvent être similaires. C'est par exemple le cas dans Pharos. Un module représente le canal d'une communauté. Les fonctionnalités étant les mêmes d'un canal à l'autre, le canal ne varie que par les données qu'il produit : structure des annotations, propriétés du canal, données persistante du canal, pages Web, ... Il serait donc inutilement coûteux de recopier dans la branche d'installation (le répertoire `channels` dans la figure 10.2) les mêmes ressources statique. Pour éviter cela, nous proposons l'héritage de ressources entre modules.

Il existe trois catégories de modules : les autonomes, les génériques et les spécifiques. Les autonomes sont fournis avec toutes les ressources nécessaires à leur exécution[10]. Les génériques fournissent un ensemble plus ou moins complet de ressources génériques mais ne peuvent pas être directement chargés dans l'application. Ils doivent être spécialisés au travers d'un ou plusieurs modules spécifiques qui héritent de leur ressources. Le modèle que nous proposons ne permet de ne faire que de l'héritage simple[11] à un seul niveau. Un module spécifique ne peut pas servir de module générique.

Un module générique est caractérisé par la présence de l'attribut `generic` dans son fichier de configuration. Ce typage explicite des modules générique permet au système de les distinguer et de faire des vérifications (p. ex. le chargement d'un module générique ou l'héritage d'un module non générique sont refusés). Un module spécifique est caractérisé par la présence de l'attribut `is-a` positionné avec le nom et la version d'un module générique dont il hérite.

Au chargement d'un module spécifique, le système lit son fichier de description, vérifie si les classes du module générique ont déjà été chargés, sinon il déclenche le chargement du module générique. Ensuite, le module spécifique est instancié. S'il ne possède aucune classe, elles sont trouvées par le chargeur au sein du module générique. S'il a redéfini certaines classes, elles seront chargées en priorité par le chargeur de classe. De même pour l'accès aux fichiers, le système recherche d'abord dans le répertoire d'exploitation du module spécifique, puis dans son répertoire d'installation et enfin dans le répertoire d'installation du module générique.

[10] Ils peuvent cependant faire appel à d'autre modules durant leur exécution
[11] Par opposition à l'héritage multiple.

10.3.7 Sécurité

Le chargement de code extérieur à une application est toujours une opération sensible car, potentiellement, l'intégrité de l'application peut être remise en cause. Le système doit donc pouvoir contrôler les actions des modules. Pour cela, on utilise le mécanisme de gestionnaire de sécurité[12] fournit par Java. Il s'agit de la couche supérieure du modèle de sécurité de Java (les couches inférieurs étant représentées par les chargeurs de classes et le vérificateur de *byte-code*). La mise en place d'un gestionnaire de sécurité sur un module permet de contrôler totalement ses accès aux ressources du système (fichier, réseau, arrêt du programme, ...). Avant l'invocation de toute méthodes sensibles, une demande d'autorisation est fait au gestionnaire. Dans notre cas, le gestionnaire par défaut ne contrôle que l'accès aux fichiers et vérifie qu'ils sont bien cloisonnés dans les répertoires autorisés au module.

10.3.8 Mise-à-jour des modules

Lorsque deux modules synchronisent leur données, ils commencent par vérifier leur versions respectives ainsi que les versions des modules dont ils dépendent. Une fois la synchronisation des données effectuée, si les versions sont différentes, on tente une mise à jour des modules en retard dans l'ordre topologique des dépendances. Si cela ne peut pas être fait immédiatement (p. ex. parce que le réplica est déconnecté ou parce que le site de distribution du module n'est pas accessible), ce module est enregistré dans la liste des modules à mettre à jour. Lorsqu'il sera de nouveau connecté, le réplica proposera la mise-à-jour de ces modules.

10.4 Conclusion

L'évolution d'une application distribuée doit être prévue avant d'être déployer. La plate-forme de composition que nous avons conçu permet, en modularisant l'application, de gérer dynamiquement les modules qui la constituent. Ils peuvent être ajoutés, retirés sans bloquer le fonctionnement des modules non concernés. La modularisation permet la mise-à-jour incrémentale de l'application. Pour faciliter la modularisation, la plate-forme gère les relations entre modules en vérifiant les dépendances des modules et en offrant la spécialisation de modules génériques.

L'intégration de cette plate-forme dans le modèle de réplication permet une propagation épidémique des informations sur les versions des modules. La distribution des versions d'un même module est centralisée mais chaque module peut avoir son propre serveur de distribution. Cette approche de centralisation et de répartition des serveurs de distribution garantit la cohérence des versions

[12] *Security Manager.*

tout en assurant le passage à l'échelle du système.

La distribution des versions est assurée par des protocoles standards (p. ex. HTTP ou FTP) ce qui permet d'exploiter l'infrastructure existante. En particulier, les modules seront répliqués dans les proxy-caches déployés pour ces protocole. On peut aussi envisager des réplications complètes, en utilisant les protocoles existants pour ce types de serveurs. Comme les versions ne sont pas des objets mutables et qu'elles peuvent être produites sur le même site, le modèle à copie primaire est parfaitement adapté à ce type de données. Les systèmes de réplication par copie miroir des données d'un serveur peuvent donc être utilisés pour répliquer les sites de distribution[13].

[13]Pour rendre totalement transparent l'accès aux versions des modules, il faut combiner la réplication par miroir à un mécanisme de redirection d'URL sur le miroir le plus approprié.

Quatrième partie

Implémentation et application

Chapitre 11

Implémentation de JRep

Dans ce chapitre nous décrivons l'API Java de réplication des données, *JRep* et le protocole de synchronisation *ODSP*.

11.1 JRep

Le rôle de JRep est de fournir une interface simple, homogène et générique pour le développement d'applications tirant partie des mécanismes de persistance et de réplication décrits dans les précédents chapitres. Elle intégre la gestion du journal des opérations, la représentation en mémoire des données (cf. chapitre 5, p. 37) et le protocole de synchronisation entre réplicas(cf. chapitre 7, p. 63). Le protocole entre clients et serveurs (cf. chapitre 9.4, p. 97) n'est pas intégré à l'API mais est construit à partir des méthodes qu'elle fournit.

L'API a été conçue pour être utilisable tel quel dans les cas simples mais elle fournit un certains nombre de points d'interaction pour les applications devant interagir plus finement avec les mécanismes internes. L'objectif est rendre le travail du programmeur proportionnel à la complexité du modèle de données et de réplication de son application.

11.1.1 Store

L'API se compose d'un ensemble de modules encapsulés dans un objet *façade* [GHJV95] : le *Store*. Le Store ne fait aucun traitement en particulier mais délégate les invocations sur ces différents modules. La figure 11.1 représente les principales méthodes du Store, les modules qui le composent et leur relation.

11.1.2 LogManager

Le `LogManager` est chargé de la lecture et de l'écriture du journal des opérations. Cependant, il n'est pas responsable de la construction des opérations à partir des objets (c'est le rôle de l'`ObjectManager`). Il ne gère que des opérations qui sont représentées en mémoire par des objets de la classe `LogEntry`.

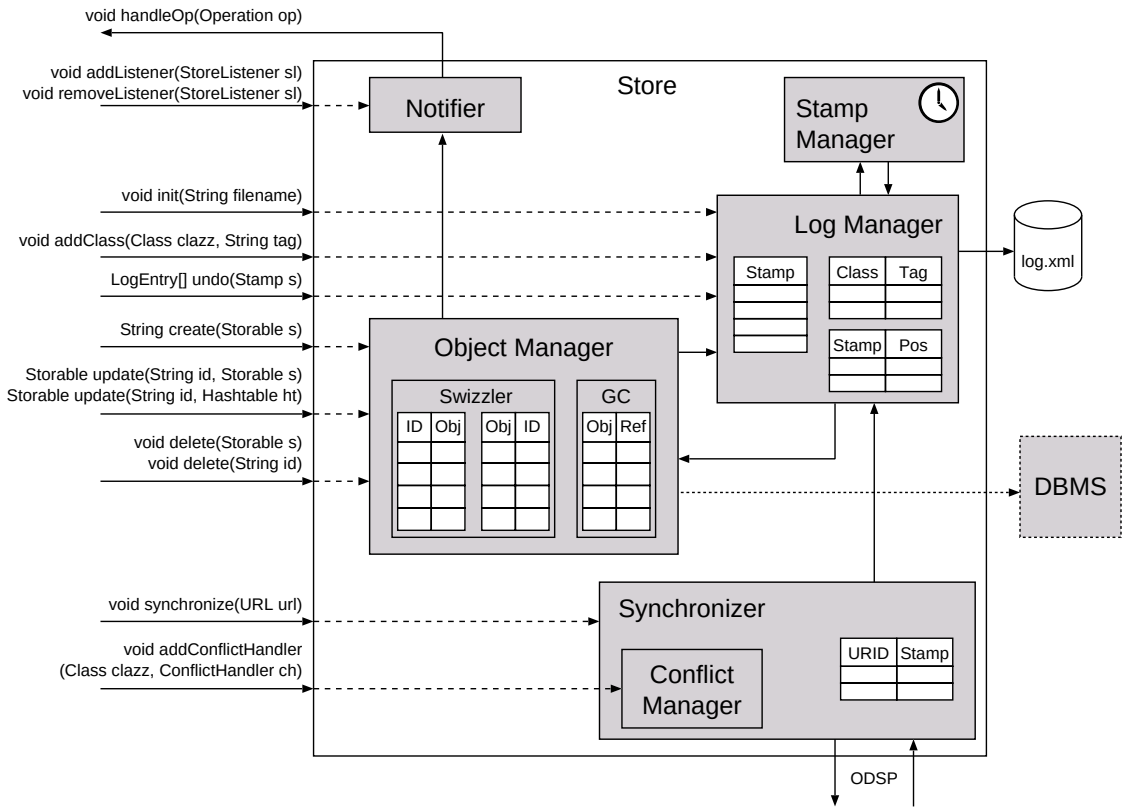


FIG. 11.1 – *Architecture des composants de l'API de réplication*

À chaque insertion dans le journal, l'estampille de l'opération est ajoutée à la *table des arrivées* (cf. section 7.2.2, p. 67). Pour permettre des accès rapides sur le journal, un index des opérations est construit. Il s'agit d'une table qui associe à chaque estampille la position de l'opération correspondante dans le journal.

La méthode `init()` déclenche la lecture du journal. A chaque opération, un objet `LogEntry` qui contient tous les attributs de l'opération est produit et soumis à l'`ObjectManager`. Le `LogManager` est chargé de la compensation des opérations (methode `undo()`). On lui précise jusqu'à quelle estampille il faut compenser. Il soumet à l'`ObjectManager` les opération de compensation en relisant le journal depuis la fin jusqu'à l'opération recherchée.

11.1.3 StampManager

Ce module est chargé de la génération des estampilles pour les opérations du journal. Il utilise le mécanisme de génération décrit dans la section 8.1 (p. 79). La méthode `getStamp()` renvoit une estampille correspondant à la date d'invocation de la méthode. L'estampille est une classe composé de deux attributs : `clock` qui représente la valeur de l'horloge logique et `urid` qui représente l'identificateur du réplica. La méthode `updateClock()` met à jour la valeur de l'horloge logique. Cette méthode est appelée par le `Synchronizer` lorsque l'horloge

du réplicas émetteur est plus en avance.

11.1.4 ObjectManager

L'`ObjectManager` gére la représentation en mémoire des objets. Toutes les opérations qui ont lieu sur les objets doivent passer par ce modules pour être ensuite enregistrées par le `LogManager`. Ce module est composé du `Swizzler` qui gère les références entre objets et d'un ramasse-miette qui élimine de l'`ObjectManager` les objets inutilisés. Celui-ci fonctionne avec un simple comptage des références.

L'`ObjectManager` dispose d'une table de correspondance entre les classes qu'ils gère et leur nom de balise XML. Par défaut, le nom de la balise correspond au nom de la classe Java. Cependant, du fait de leur construction, les noms de classes Java peuvent être très long. Ce mécanisme évite des consommations d'espace disque inutiles et rend le journal plus lisible.

11.1.5 Internalisation des opérations

L'`ObjectManager` reçoit des objets `LogEntry` fournit par le `LogManager`. Le nom de la classe Java est retrouvée à partir du nom de la balise. Si il s'agit d'une création, on instancie cette classe, on remplit les champs de l'objets et on enregistre l'objet dans l'`ObjectManager`. Si il s'agit d'une modification, on retrouve l'objet par l'intermédiaire de l'`ObjectManager` et on modifie les attributs concernés. Si il s'agit d'une destruction, l'`ObjectManager` est prévenu. Dans tous les cas, l'exécution de opération est notifié au `Notifier`.

Pour internaliser la représentation XML des attributs, nous proposons de méthode :

- interne à la classe : ce mode est bien adapté pour les classes écrite par le développeur (p. ex. la classe `annotation`) ;

- externe à la classe : ce mode permet une ouverture pour des classes dont on ne contrôle pas l'implémentation (p. ex. la classe `java.uti.Date`).

11.1.6 Swizzler

Le `Swizzler` gère la conversion des références entre objets et indirectement leur représentation mémoire. Pour cela, il tient à jour deux tables permettant d'accèder à un `Storable` à partir de son PID (*Persistent IDentifier*) et vice-versa. Lors de la lecture du journal, il est notifié des opérations par le `Log Manager`. Un objet est construit à la réception de chaque opération de création (soit de l'extérieur par l'invocation de la méthode `create()` soit en provenance du `LogManager`). Son état est modifié pour chaque opération de mise-à-jour. Il est détruit (i.e. plus référencé par le `Swizzler`) lorsqu'une opération de destruction est lue. À l'inverse, le `Swizzler` est invoqué par le `LogManager` pour

convertir les références entre objet en référence sur les identificateurs de ces objets (attributs de type `IDREF` en XML).

Lors de la création ou de la mise-à-jour d'un objet à partir des opérations reçues, le `Swizzler` informe le `Notifier` de cette opération..

11.1.7 Notifier

Le `Notifier` permet à des objets externe au Store d'être informé des mutations des objets. C'est grâce à cela que l'on peut construire des index propre à l'application. Typiquement les index sont représentés par des tables de hachage associant une clé applicative à un des objets persistants.

Le modèle évènementiel utilisé reprend celui généralement utilisé dans les API Java. Les évènements sont représenté par des objets de classes caractèrisant le type d'événements. Les objets à notifié implémentent une interface (`StoreListener`) qui possède une méthode par type d'événement. Les objets implémentant cette interface sont appelé des *listeners*. Lorsqu'un événement survient, le `Notifier` itére sur tous les *listeners* qui se sont enregistrés et invoque la méthode correspondant au type d'évenement. L'interface `StoreListener` possède quatre méthodes :

- `handleCreate(Storable storable, String id)` est appelée pour notifier la création de l'objet `storable` identifié par `id` ;
- `handlePrepareUpdate(Storable storable, String id, Map attributes)` est appelée pour notifier la mise-à-jour de l'objet `storable` *avant* qu'il ne soit modifié ;
- `handleCommitUpdate(Storable storable, String id, Storable oldStorable)` est appelée pour notifier la mise-à-jour effective de l'objet `storable` ;
- `handleDelete(Storable storable, String id)` est appelée pour notifier la destruction de l'objet `storable`.

11.1.8 Synchronizer

Le `Synchronizer` gère les synchronisations entre deux réplicas. Une synchronisation est déclenchée en invoquant la méthode `synchronize()`. Celle-ci reçoit en paramètre l'URL du réplica avec lequel il faut établir la synchronisation ainsi que le type de synchronisation (pull, push, push/pull). Si l'URL n'est pas précisée, la synchronisation à lieu avec le réplica père.

La synchonisation se fait via le protocole ODSP (cf. section 11.2, p. 119). Le `Synchronizer` implémente à la fois un serveur et un client ODSP. Lorsqu'il reçoit une synchronisation entrante, il applique l'algorithme décrit à la section 7.1.3 (p. 65). Les conflits sont gérés par le `ConflictManager`.

11.1.9 ConflictManager

Le `ConflictManager` gère les conflits qui surviennent durant une synchronisation. Les politiques de gestion des conflits sont spécifique à une classe donnée. Pour les conflits de mutation (cf. section 8.3, p. 85), en cas de conflit `update/update`, par défaut le gestionnaire ne fait aucune intervention (résolution par l'estampillage). Pour les conflits `delete/update`, le développeur doit indique le mode de résolution (par défaut `delete`). Par défaut, le gestionnaire suppose qu'il n'y a pas de conflits inter-objets. Si c'est le cas, le développeur indique pour chaque couple de classe, la procédure de résolution à appliquer. Celle-ci est une interface (`ClassConflictResolver` ou `InterClassConflictResolver`) dont la méthode `resolveConflict` reçoit en parametre la ou les classes concernées, les identificateurs des objets et les opérations conflictuelles. Elle retourne une liste d'opérations.

11.2 Le protocole ODSP

Le protocole ODSP (*Optmistic Data Synchronization Protocol*) défini le modèle de communication pour la synchronisation de deux réplicas. Il s'agit d'un protocole client/serveur sans session. La structure des messages (requêtes et réponse) est très proche de celle de HTTP[BLFN96]. Nous utilisons les même conventions que celles décrites dans ce protocole. La version décrites permet des synchronisations par l'envoi complet des journaux ou par l'envoi incrémental en utilisant les indices d'arrivés.

11.2.1 Messages ODSP

Les messages ODSP correspondent aux requêtes provenant des clients et aux réponse provenant des serveurs. Chaque réplica est tour à tour client ou serveur.

```
ODSP-message = ODSP-Request
               | ODSP-Response

ODSP-request   = Request-Line
                 Request-Header
                 CRLF
                 [ Entity-Body ]

ODSP-response   = Status-Line
                  Response-Header
                  CRLF
                  [ Entity-Body ]
```

11.2.2 Requêtes

Chaque requête contient sur la première ligne du message le type de synchronisation à effectuer.

```
Request-Line = Method SP ODSP-Version CRLF
Method       = PUSH
               | PULL
               | PUSH-PULL

Request-Header = URID
               | Exchange-Type
               | Incoming-Indice
               | Content-Type
               | Content-Encoding
               | Content-Length

Exchange-Type = "Full"
                | "Incremental"
```

Le champ d'entête `Exchange-Type` permet de caractérisé si il s'agit d'un envoi complet du journal ou d'un envoi incrémentale. Dans ce dernier cas, si les indices d'arrivés sont utilisés, le champ `Incoming-indice` doit être positionné.

11.2.3 Réponses

La première ligne d'une réponse contient la `Status-Line` :

```
Status-Line =ODSP-Version SP Status-Code SP Reason-Phrase CRLF

Status-Code = "200" ; OK
              | "201" ; BACK_SYNC
              | "400" ; BAD_REQUEST
              | "401" ; UNAUTHORIZED
              | "403" ; FORBIDDEN
              | "500" ; INTERNAL_SERVER_ERROR
              | "503" ; SERVICE_UNAVAILABLE
```

Lors d'une requête avec la méthode `PUSH` le status-code de la réponse est positionné à `OK`. `BACK_SYNC` est possitionné pour les réponses aux requêtes utilisant les méthodes `PULL` ou `PUSH_PULL`.

Chapitre 12

Application à Pharos

Ce chapitre illustre les concepts et les techniques présentés dans cette thèse à travers Pharos.

12.1 Journalisation des opérations

Les canaux Pharos matérialisent les communautés thématiques d'utilisateurs. Un canal se compose d'une partie serveur et d'une partie client. La première gère un ensemble de données typées produites par les utilisateurs. Chacune de ces données est une structure regroupant un ensemble d'attributs. Les attributs sont soit des données simples (p. ex. une chaîne de caractère, un entier, ...) soit des références sur d'autres types de données gérées par le canal. Un canal se compose de trois types de données (dont les relations sont illustrées à la figure 12.1) :

- Le type `user` décrit un membre du canal (nom, mot de passe, description, droits, ...).
- Le type `term` décrit un terme du thésaurus[1] (nom, référence sur le nœud parent, note d'application, ...)
- Le type `annotation` décrit une annotation. La structure des annotations est propre à chaque canal. On retrouve néanmoins les attributs `url` et `user` en commun. À titre d'exemple, nous considérons des annotations composées d'un titre, d'une note pour évaluer l'intérêt du document, d'un commentaire et d'une liste de termes.

Les types de données *user*, *term* et *annotation* ont trois autres attributs en commun :

- `cdate` : date de création ;
- `mdate` : date de dernière modification ;
- `user-id` : identificateur de l'auteur de l'opération.

[1] Le terme de thésaurus est un peu abusif puisqu'il s'agit simplement d'un arbre de termes. Pour gérer un véritable thésaurus, il faudrait introduire en plus de la relation de hiérarchie d'autres liens tels que la relation de synonymie ou de connexité.

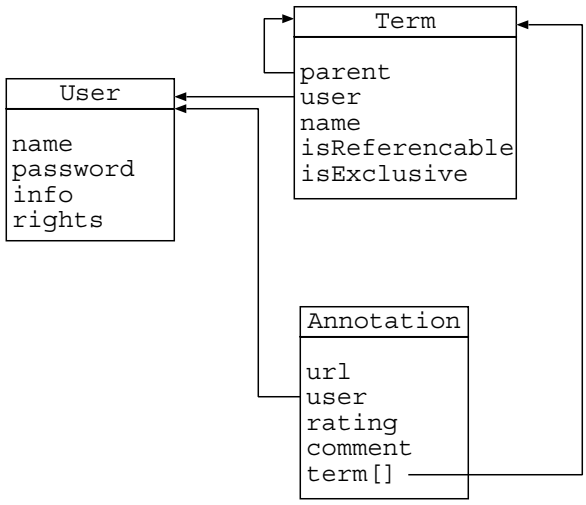


FIG. 12.1 – *Modèle relationnel des données dans un canal Pharos*

12.1.1 Opérations portant sur le type *annotation*

Déclaration XML :

```
<!ELEMENT annotation EMPTY>
<!ATTLIST annotation
          stamp     CDATA                  #REQUIRED
          index     CDATA                  #REQUIRED
          filtered  (true|false)           false
          id        ID                     #REQUIRED
          op        (create|update|delete) #REQUIRED
          cdate     CDATA                  #IMPLIED
          mdate     CDATA                  #IMPLIED
          user-id   IDREF                  #REQUIRED
          url       CDATA                  #IMPLIED
          rating    CDATA                  #IMPLIED
          title     CDATA                  #IMPLIED
          term-ids  IDREFS                 #IMPLIED
          comments  CDATA                  #IMPLIED
          pdate     CDATA                  #IMPLIED>
```

Signification des attributs spécifiques :
- `url` : URL du document annoté ;
- `title` : titre de l'annotation ;
- `rating` : note d'intérêt ;
- `terms-ids` : liste des identificateurs des termes ;
- `comments` : commentaire ;
- `pdate` : date de péremption de l'annotation (en seconde). Absent si pas de date de péremption prévue.

Exemple de représentation dans le journal :

1. Création d'une annotation :

```
<annotation stamp="3252"
```

```
            index="34"
            filtered=false
            id="3252"
            op=create
            cdate="924791842697"
            user-id="4"
            url=""http://www.alphaworks.ibm.com/formula/jikes/"
            title="Jikes"
            rating="1"
            terms-ids="115"
            comments="The faster and most rigorous Java compiler."
>
   <undo op=delete/>
</annotation>
```

2. Mise-à-jour de cette annotation (la note d'intérêt est modifiée et un terme est ajouté) :

```
<annotation stamp="3324"
            index="38"
            filtered=false
            id="3252"
            op=update
            mdate="924791823457"
            user-id="4"
            rating="3"
            terms-ids="115 124"
>
   <undo op=update
         mdate=""
         rating="1"
         terms-ids="115"
   />
</annotation>
```

3. Destruction de cette annotation :

```
<annotation stamp="3415"
            index="50"
            filtered=false
            id="3252"
            op=delete
            mdate="924791831451"
            user-id="5"
/>
   <undo op=create
         cdate="924791842697"
         mdate=""
         user-id="4"
         url=""http://www.alphaworks.ibm.com/formula/jikes/"
         title="Jikes"
         rating="3"
         terms-ids="115 124"
         comments="The faster and most rigorous Java compiler."
   />
</annotation>
```

12.1.2 Opérations portant sur le type *user*

Déclaration XML :

```
<!ELEMENT user EMPTY>
<!ATTLIST user
          stamp     CDATA                  #REQUIRED
          index     CDATA                  #REQUIRED
          filtered  (true|false)           false
          id        ID                     #REQUIRED
          op        (create|update|delete) #REQUIRED
          cdate     CDATA                  #IMPLIED
          mdate     CDATA                  #IMPLIED
          user-id   IDREF                  #REQUIRED
          login     CDATA                  #IMPLIED
          password  CDATA                  #IMPLIED
          email     CDATA                  #IMPLIED
          info      CDATA                  #IMPLIED
          rights    CDATA                  #IMPLIED>
```

Signification des attributs spécifiques :

- `login` : login de l'utilisateur ;
- `password` : mot de passe de l'utilisateur ;
- `info` : attribut textuel de description de l'utilisateur ;
- `rights` : droit d'accès de utilisateur.

Remarque : l'attribut `user-id` contient l'identité de l'utilisateur l'ayant créé. Il s'agit typiquement de l'administrateur du canal. Cependant, il peut être égal à lui-même s'il s'agit du premier utilisateur du canal ou si le canal autorise la création automatique d'utilisateurs.

Exemple de représentation dans le journal :

```
<user stamp="1817"
      index="2"
      filtered=false
      id="2"
      op=create
      cdate="924791832465"
      user-id="0"
      login="Olivier"
      password="BdZak7cLdHHnM"
      email="Olivier.Dedieu@inria.fr"
      info="Channel Administrator"
      rights="3"
>
   <undo op=delete/>
</user>
```

12.1.3 Opérations portant sur le type *term*

Déclaration XML :

```
<!ELEMENT term EMPTY>
<!ATTLIST term
          stamp     CDATA                  #REQUIRED
          index     CDATA                  #REQUIRED
```

```
          filtered  (true|false)           false
          id        ID                     #REQUIRED
          op        (create|update|delete) #REQUIRED
          cdate          CDATA                  #IMPLIED
          mdate          CDATA                  #IMPLIED
          user-id        IDREF                  #REQUIRED
          name           CDATA                  #IMPLIED
          parent-id      IDREF                  #IMPLIED
          scope-note     CDATA                  #IMPLIED>
          referenceable  (true|false)           true
          exclusive      (true|false)           false>
```

Signification des attributs spécifiques :

- `name` : nom du terme ;
- `parent-id` : identificateur du nœud parent. Le nœud racine est son propre nœud parent ;
- `scope-note` : note sur l'utilisation du terme ;
- `referenceable` : indique si le terme est référençable[2] ;
- `exclusive` : indique si les nœuds fils sont exclusivement référençables.

Exemple de représentation dans le journal :

```
<term stamp="115"
      index="20"
      filtered=false
      id="115"
      op=create
      cdate="924791836697"
      user-id="2"
      name="Compilers"
      parent-id="90"
      scope-note="Any tool producing a .class file from a .java file"
      referencable=true
      exclusive=false
>
   <undo op=delete/>
</term>
```

12.2 Synchronisation des réplicas

Les canaux peuvent être répliqués sur les différents serveurs Pharos. Cependant, tous les canaux ne sont pas forcément répliqués sur tous les serveurs. La décision de répliquer un canal sur un serveur est laissée à l'administrateur de ce serveur.

La topologie des serveurs n'est pas contrainte par l'application. La synchronisation des données des canaux se fait directement entre les deux parties serveur des canaux. Les serveurs concernés n'interviennent que pour établir la communication entre les deux canaux.

[2]Certains termes du thésaurus peuvent ne servir qu'au regroupement de termes sémantiquement liés (p. ex. *Nature du document*, *Sujet*, ...). Ces termes n'ont aucune valeur de classement et ne doivent donc pas être référençables.

12.2.1 Gestion des conflits

12.2.1.1 Le type *user*

Les objets de type *users* utilisent la matrice de mutation par défaut, en privilégiant les destructions[3]. Il n'y a pas de matrice inter-objet car les objets *user* n'ont pas de références sur d'autres types et les doublons sont autorisés[4].

12.2.1.2 Le type *term*

Les objets de type *term* représentent l'arbre des termes du thésaurus. Il s'agit d'un type récursif. Ils utilisent la matrice de mutation par défaut en privilégiant les destructions. Il n'y a pas de matrice inter-objets avec le type *user*. L'attribut *user-id* sert surtout à avoir une trace des personnes ayant modifié le thésaurus. La matrice inter-objet pour le type *term* tient compte des relations père-fils entre terme. Elle intègre aussi le mécanisme de gestion des doublons décrit à la section 8.5 (p. 88). Un doublon est identifié lorsqu'il y a égalité sur le nom du terme et le nœud père.

Opérations portant sur deux termes distincts :

Récepteur (R) \ Émetteur (E)	`create`	`update`	`delete`
`create`	(1)	(1)	(2)
`update`	(1)	(1)	(2)
`delete`	(3)	(3)	E+R

1. Si les deux opérations ne font pas apparaître de doublon, elles sont toutes les deux enregistrées. Sinon on applique le mécanisme décrit à la section 8.5 (p. 88).
2. Si l'opération de E ne concerne pas le père de l'opération de R, alors il n'y a pas de conflit (E+R). Sinon on retient la destruction.
3. Si l'opération de R ne concerne pas le père de l'opération de E, alors il n'y a pas de conflit (E+R). Sinon on retient la destruction.

12.2.1.3 Le type *annotation*

Les données de type *annotation* utilisent la matrice de mutation par défaut. On évite les doublons en rejetant de façon déterministe la mutation faisant apparaître le doublon (celle ayant la plus grande estampille). Un doublon est identifié lorsqu'il y a égalité sur l'identificateur de l'utilisateur et l'identificateur de l'URL. Il n'y a pas besoin de mettre en œuvre le mécanisme décrit à la section 8.5 (p. 88) car, contrairement aux termes, les annotations ne sont pas

[3] On privilégie les destructions provenant toujours de l'administrateur plutôt que les mises-à-jour pouvant provenir de l'utilisateur.

[4] En cas de doublons (même nom), l'utilisateur avec le plus grand identificateur est prévenu lors de la connexion suivante et doit changer de nom.

référencées par d'autres objets. Deux matrices inter-objets sont définies pour les relations avec les types *user* et *term* :

Opérations portant sur une annotation et un utilisateur :

Annotation \ User	`create`	`update`	`delete`
`create`	∅	A+U	(1)
`update`	∅	A+U	(1)
`delete`	∅	A+U	A+U

1. Détruire l'utilisateur et son annotation.

Opérations portant sur une annotation et un terme :

Annotation \ Terme	`create`	`update`	`delete`
`create`	A+T	A+T	(1)
`update`	A+T	A+T	(1)
`delete`	A+T	A+T	A+T

1. On retient les deux opérations ainsi qu'une opération supplémentaire de mise-à-jour de l'annotation, en ne tenant plus compte de ce terme.

12.2.2 Gestion du contrôle d'intégrité

Pharos utilise le contrôle d'intégrité par défaut (cf. section 8.7, p. 89) qu'il enrichit du contrôle d'intégrité sur le thésaurus. En effet, une synchronisation de deux mises-à-jour indépendantes peut introduire des cycles dans le thésaurus. Soient t_1 et t_2 deux termes. Sur le réplica R_1, t_1 devient fils de t_2 ; sur le réplica R_2, t_2 devient fils de t_1. Lorsque R_1 et R_2 se synchronisent, les deux mises-à-jour sont acceptées car il ne s'agit ni d'un conflit de mutation ni d'un conflit entre deux objets liés par un référencement[5].

La violation est détectée en vérifiant qu'à chaque enregistrement d'une mise-à-jour sur un terme celui-ci n'existe pas dans la liste de ses parents. Si c'est le cas, la mutation est rejetée (l'autre aura été préalablement acceptée).

[5]Il s'agit du cas le plus simple; si le cycle se forme sur un grand nombre de *term*, la détection par comparaison des mutations deux à deux devient très difficile.

Cinquième partie

Conclusion

Chapitre 13

Conclusions et perspectives

Dans cette thèse nous avons décrit un système de réplication optimiste d'applications collaboratives asynchrones. L'objectif principal est d'offrir une haute disponibilité des données en lecture et en écriture. Aussi, la cohérence permanente du système n'est pas le but recherché. On offre un mode de fonctionnement dans lequel chaque réplica est autonome. Les réplicas se synchronisent deux à deux et de façon opportuniste pour tendre vers un état globalement cohérent. Le protocole n'impose aucune contrainte sur le choix des partenaires ni sur les fréquences de synchronisation. Sur ce dernier point, des barrières, extérieures au protocole, peuvent néanmoins être mises en place pour faire face à des réplicas obsolètes. Pour augmenter la performance du système, le protocole de synchronisation assure des échanges incrémentaux. Ainsi, un réplica est assuré de ne recevoir et de n'envoyer que des nouvelles opérations. De même, il peut filtrer une partie des données à répliquer pour ne recevoir, durant les synchronisations, que les nouvelles écritures qui lui sont destinées.

La gestion des écritures concurrentes est la principale difficulté apportée par la réplication optimiste. Les données étant mutables et les réplicas effectuant leurs écritures de façon autonome, chaque écriture est potentiellement conflictuelle avec celles des autres réplicas. L'application doit donc être prête à accepter une résolution postérieure des conflits, au risque de perdre certaines écritures. Le protocole de synchronisation assure néanmoins, qu'une écriture n'entrera en conflit qu'avec de nouvelles écritures des partenaires. Lors d'une synchronisation le protocole a la charge de détecter et de résoudre les conflits. Nous considérons deux principales catégories de conflits : les conflits de mutation et les conflits inter-objets. Les premiers sont gérés de façon automatique grâce à l'estampillage logique à incrément temporel. Les seconds sont traités par le programmeur d'applications.

Le système a été conçu de manière à être déployable et à subir des évolutions. Ainsi, le mécanisme de persistance offre un support simple pour les évolutions de schéma et autorise, au sein du même journal, des écritures sur différentes versions des classes d'objets. Le protocole de synchronisation est, lui aussi, souple vis-à-vis des évolutions : il ne fait aucune supposition sur la struc-

ture des données sur lesquelles portent les écritures qu'il synchronise. La gestion des conflits se fait par la production explicite d'opérations de rejet. Ainsi, les changements de politique de conflits ne remettent pas en cause la cohérence à terme des réplicas. Pour que le protocole puisse être opérationnel dans un environnement ouvert, nous proposons des barrières de synchronisation qui assurent qu'un groupe de réplicas bien synchronisés ne sera pas destabilisé par un réplica obsolète. Enfin, en plus de la synchronisation des données, il faut aussi assurer le déploiement du code applicatif. Ceci est pris en charge par un second protocole basé sur JPlug, un *framework* de composition modulaire d'applications.

L'application Pharos nous a servi de fil conducteur pour illustrer sur un cas concret les mécanismes et les algorithmes proposés. Cette application est actuellement en exploitation. La persistance des données par journalisation a été intégrée sur la version opérationnelle. Le protocole de réplication a été implémenté partiellement sur un prototype. L'objectif était de confronter le système à une véritable application. Ceci nous a permis de faire apparaître des problèmes qui n'avaient pas été initialement envisagés comme par exemple, le contrôle d'intégrité après une synchronisation.

13.1 Extensions possibles

13.1.1 Contrôle des écritures

Le système actuel est fonctionnel mais des améliorations sont cependant envisageables. Parmi les extensions que nous pourrions apporter, il serait intéressant de pouvoir gérer plus finement le contrôle des écritures. Par exemple, dans le cas de Pharos, ce sont principalement les termes du thésaurus qui risquent de provoquer des conflits. De plus, du fait de la structure arborescente du thésaurus, leur résolution n'est pas simple. Par ailleurs, la manipulation du thésaurus est réservée à certains membres et en pratique il change assez peu. Aussi, il serait pragmatique d'opter pour une approche hybride, combinant une approche optimiste pour les écritures sur les annotations et membres avec une approche pessimiste pour les écritures sur le thésaurus. En permettant, d'adapter la politique des écritures (centralisées ou réparties) pour chacune des classes d'objets gérés, on déchargerait le développeur de la réalisation d'une part importante des procédures de résolution qui peuvent s'avérer inutiles en pratique.

13.1.2 Prédiction des conflits

Lorsqu'un réplica produit de nombreuses écritures sans se resynchroniser, il augmente les probabilités de conflits. Il n'existe pas une fréquence de synchronisation idéale. Elle dépend des données modifiées, du type de mutation et du mode de connection de l'utilisateur. Par exemple, dans Pharos, la création d'une nouvelle annotation est moins sensible que la destruction d'un terme. Pour l'utilisateur, ce genre d'information s'acquiert avec l'expérience. Aussi, il

serait intéressant de pouvoir fournir des recommandations aux utilisateurs. Ces dernières pourraient être soit statiques (p. ex. selon la nature des objets et des mutations) soit dynamiques, à partir de l'analyse des conflits passés.

13.2 Perspectives et enjeux de la réplication optimiste

L'émergence d'Internet et de l'informatique nomade a mis en évidence la dimension fonctionnelle de la réplication. Jusque là, les systèmes intégraient la réplication des données pour augmenter leur performances ou la tolérance aux pannes. C'est dans cette approches que la plupart des systèmes de réplication pessimiste ont été conçus. La réplication optimiste apparaît davantage alors comme une nouvelle brique dans l'élaboration des systèmes d'information, au même niveau que les systèmes de stockage ou de communication. La multiplication des points d'accès à l'information (ordinateur de bureau, portable, PDA, téléphone cellulaire, ...) renforce ce besoin de disposer en permanence d'un accès aux données. Tant que l'on ne disposera pas d'infrastructures de communication efficaces pour tous, la réplication optimiste restera une solution pertinente pour offrir un service de haute disponibilité.

Avec l'essor du commerce électronique, les sites marchands doivent faire face aux nouvelles formes de banditisme. Les attaques par refus de services peuvent occasionner des pertes importantes pour ces sites. La réplication optimiste permet d'y faire face en partie. D'une part, toutes les écritures sont tracées. On rétablit ainsi facilement et rapidement une version des données telles qu'elles étaient avant l'attaque. D'autre part, en analysant les écritures illégitimes, on peut comprendre la cible des attaques et en déduire des pistes pour retrouver leurs auteurs. D'autre part, la distribution des points d'accès rend plus difficile les attaques par refus de services. En effet, les attaquants ne peuvent plus concentrer leurs assauts sur un point central mais doivent eux aussi les distribuer. Enfin, notre modèle de réplication a été conçu pour une exploitation en déconnecté. ce qui représente une solution radicale pour éviter les attaques. La réplication optimiste permet de concevoir des architectures où les informations sensibles peuvent n'être physiquement accessibles sur Internet qu'au travers de copies. Pour réduire les contacts physiques entre les deux réseaux, on utilisera la synchronisation par support amovible pour garantir la cohérence (retardé) des copies, tout en continuant à assurer la protection des données sensibles.

La disponibilité permanente des données augmente l'intérêt des utilisateurs à les exploiter. En les intégrant physiquement à leur système d'information, on leur apporte en plus le sentiment d'appropriation. Cette dimension psychologique est importante pour les systèmes collaboratifs, car il n'est pas toujours aisé de faire participer activement des utilisateurs. Or, ce n'est qu'à cette condition que de tels systèmes prennent véritablement leur intérêt. Il n'est cependant pas nécessaire que tous les participants produisent des données (la majorité peut se contenter de lecture). Cependant, il faut encourager ceux qui souhaitent

participer activement à le faire. À travers les données, la réplication optimiste développe le sentiment d'appropriation du système et peut augmenter la participation des utilisateurs.

Bibliographie

[AD76] Peter A. Alsberg et John D. Day. A principle for resilient sharing of distributed resources. Dans *Proc. 2nd Int. Conf. on Software Engineering*, pages 562–570, San Francisco, CA (USA), octobre 1976.

[ADJ+96] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, et S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4) :68–75, décembre 1996.

[AG98] Ken Arnold et James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[AL97] Atul Adya et Barbara Liskov. Lazy consistency using loosely synchronized clocks. Dans *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 73–82, Santa Barbara, California, août 1997. `http://www.pmg.lcs.mit.edu/papers/podc97/`.

[AM95] M. P. Atkinson et R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3) :319–401, 1995.

[AWO+99] Ken Arnold, Ann Wollrath, Bryan O'Sullivan, Robert Scheifler, et Jim Waldo. *The Jini specification*. Addison-Wesley, Reading, MA, USA, 1999.

[BAKR96] L. Bellissard, S. B. Atallah, A. Kerbrat, et M. Riveill. Component-based programming and application management with Olan. *Lecture Notes in Computer Science*, 1107 :290– ??, 1996.

[BD99] Vincent Bouthors et Olivier Dedieu. Pharos, a collaborative infrastructure for web knowledge sharing. Sous la direction de Serge Abiteboul et Anne-Maire Vercoustre, *Research and Advanced Technology for Digital Libraries. Third European Conference, ECDL '99, Paris, France, September 22–24, 1999 : Proceedings*, Lecture Notes in Computer Science, pages 215–233. Springer-Verlag Inc., 1999.

[BF93] N. Borenstein et N. Freed. RFC 1521 : Mime (multipurpose internet mail extensions) part one : Mechanisms for specifying and describing the format of internet message bodies, septembre 1993. `ftp://ftp.inria.fr/rfc/rfc15xx/rfc1521.Z`.

[BG77] L. E. Bonanni et A. L. Glasser. SCCS/PWB user's manual. Sous la direction de T. A. Dolotta, R. C. Haight, et E. M. Piskorik,

PWB/UNIX User's Manual–Edition 1.0, pages 1–22. Bell Laboratories, 1977.

[BG93] T. Berlage et A. Genau. A framework for shared applications with a replicated architecture. Dans *Proceedings of ACM Symposium on User Interface Software and Technology*, Atlanta, Georgia, novembre 1993. `http://zeus.gmd.de/~berlage/uist93.pdf`.

[BLFN96] Tim Berners-Lee, Roy T. Fielding, et Henrik Frystyk Nielsen. Informal RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0, mai 1996. `ftp://ftp.inria.fr/rfc/rfc19xx/rfc1945.Z`.

[BLMM94] T. Berners-Lee, L. Masinter, et M. McCahill. RFC 1738 : Uniform resource locators (url), décembre 1994. `ftp://ftp.inria.fr/rfc/rfc17xx/rfc1738.Z`.

[BN97] P. A. Bernstein et E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, San Francisco, 1997.

[BPSM98] Tim Bray, Jean Paoli, et C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. W3C recommendation, W3C, feb 1998. `http://www.w3.org/TR/1998/REC-xml-19980210.html`.

[CBF+90] T. Crowley, E. Baker, H. Forsdick, P. Milazzo, et R. Tomlinson. MMConf : An infrastructure for building shared applications. Dans *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, Los Angeles, California, 1990. ACM Press.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, et D. Terry. Epidemic algorithms for replicated database maintenance. Dans *Sixth Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, Canada, août 1987.

[Dro93] R. Droms. RFC 1541 : Dynamic host configuration protocol, octobre 1993. `ftp://ftp.inria.fr/rfc/rfc15xx/rfc1541.Z`.

[Fes99] Fabrice Le Fessant. Jocaml : programmation distribuée et agents mobiles. Dans *Conférence Française sur les Systèmes d'Exploitation*, Rennes, France, 1999.

[FGM+97] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, et Tim Berners-Lee. RFC 2068 - Hypertext Transfer Protocol – HTTP/1.1, janvier 1997. `ftp://ftp.inria.fr/rfc/rfc20xx/rfc2068.Z`.

[FSB+98] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Joâo Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, et Sacha Krakowiak. PerDiS : design, implementation, and use of a PERsistent DIstributed Store. Technical Report QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, QMW, CSTB, INRIA and INESC, octobre 1998. `http://www-sor.inria.fr/publi/PDIUPDS_rr3525.html`.

[GHJV95] Gamma, Helm, Johnson, et Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 1995. `http://hillside.net/patterns/DPBook/DPBook.html`.

[GL93] Richard Golding et Darrell D. E. Long. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. ucsc-crl-93-03, University of California, Santa Cruz, février 1993. `ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-93-09.ps.Z`.

[HA87] M. Horton et R. Adams. RFC 1036 : Standard for interchange of usenet messages, décembre 1987. `ftp://ftp.inria.fr/rfc/rfc10xx/rfc1036.Z`.

[HdH98] David Hulse, Alan dearle, et Alasdair Howells. Lumberjack : A log-structured persistent object store. Dans *Proceedings of the Eighth International Workshop on Persistent Object Systems : Design, Implementation and Use.*, Tiburon, California, USA, août 1998.

[HHB96] Abdelsalam Helal, Abdelsalam Heddaya, et Bharat Bhar. *Replication Techniques in Distributed Systems.* Kluwer Academic Publishers, august 1996. `http://www.cise.ufl.edu/~helal/books/repl.html`.

[IB96] V. Issarny et C. Bidan. Aster : A corba-based software interconnection system supporting distributed system customization. Dans *Proceedings of the International Conference on Configurable Distributed Systems*, 1996.

[IBM99] IBM. DB2 replication guide and reference, mars 1999. `http://publib.boulder.ibm.com:80/cgi-bin/bookmgr/BOOKS/ASNU5002/CCONTEN%TS`.

[Inf98] Informix. Informix enterprise replication, 1998. `http://www.informix.com/informix/products/technologies/extensibility/re%plication/`.

[Jav] JavaSoft. Enterprise JavaBeans.

[Jav96] JavaSoft. JavaBeans. http ://java.sun.com/beans, décembre 1996. Version 1.00-A.

[KG98] J. Kleinoeder et M. Golm. MetaJava — A platform for adaptable operating-system mechanisms. *Lecture Notes in Computer Science*, 1357, 1998. `http://www4.informatik.uni-erlangen.de/TR/ps/TR-I4-97-10.ps.gz`.

[KL86] Brian Kantor et Phil Lapsley. RFC 977 : Network news transfer protocol, février 1986. `ftp://ftp.inria.fr/rfc/rfc9xx/rfc977.Z`.

[KP90] Michael J. Knister et Atul Prakash. Distedit : A distributed toolkit for supporting multiple group editors. Dans *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work*, Systems Infrastructure for CSCW, pages 343–355, 1990.

[KS93] Puneet Kumar et M. Satyanarayanan. Log-based directory resolution in the Coda file system. Dans *Second International Conference on Parallel and Distributed Information Systems*, pages 202–213, 1993.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, juillet 1978.

[LLB+97] Gordon Landis, Charles Lamb, Tim Blackman, Sam Haradhvala, Mark Noyes, et Dan Weinreb. ObjectStore PSE : a persistent storage engine for java. Dans *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, août 1997.

[LO98] Danny B. Lange et Mitsuru Oshima. Mobile agents with java : The aglet API. *World Wide Web Journal*, 1998.

[Mar98] Marimba. Castanet infrastructure suite, 1998. `http://www.marimba.com/`.

[MD94] Jon Munson et Prasun Dewan. A flexible object merging framework. Dans *ACM Conference on Computer Supported Cooperative Work*, pages 231–242, octobre 1994. `http://www.cs.unc.edu/~dewan/abstracts/merge.html`.

[Mil92] David L. Mills. RFC 1305 : Network time protocol (version 3) specification, implementation, mars 1992. `ftp://ftp.inria.fr/rfc/rfc13xx/rfc1305.Z`.

[Mos92] J. Eliot B. Moss. Working with persistent objects : To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8) :657–673, août 1992.

[OB98] Alexandre Oliva et Luiz Eduardo Buzato. Composition of metaobjects in guarana. Dans *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, Vancouver, Canada, octobre 1998.

[Ora97] Oracle. Oracle server replication, juin 1997. `http://oradoc.photo.net/ora8doc/DOC/server803/A54651_01/toc.htm`.

[PH99] M. Philippsen et B. Haumacher. More efficient object serialization. *Lecture Notes in Computer Science*, 1586, 1999.

[PHRM90] John F. Patterson, Ralph D. Hill, Steven L. Rohall, et W. Scott Meeks. Rendezvous : An architecture for synchronous multi-user applications. Dans *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work*, Systems Infrastructure for CSCW, pages 317–328, 1990.

[PK92] A. Prakash et M. J. Knister. Undoing actions in collaborative work. Dans *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Consistency in collaborative systems, pages 273–280, Toronto, Ontario, 1992. ACM Press.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, , et Alan J. Demers. Flexible update propagation for weakly consistent replication. Dans *16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, octobre 1997. `http://www.parc.xerox.com/csl/projects/bayou/`.

[RHJ98] Dave Raggett, Arnaud Le Hors, et Ian Jacobs. Html 4.0 specification, avril 1998. `http://www.w3.org/TR/REC-html40`.

[RO91] Mendel Rosenblum et John K. Ousterhout. The design and implementation of a log-structured file system. Dans *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, octobre 1991.

[RWWB96] Roger Riggs, Jim Waldo, Ann Wollrath, et Krishna Bharat. Pickling state in the Java system. Sous la direction de USENIX Association, *2nd Conference on Object-Oriented Technologies & Systems (COOTS), June 17–21, 1996. Toronto, Canada*, Berkeley, CA, USA, juin 1996. USENIX. `http://www.usenix.org/publications/library/proceedings/coots96/riggs.ht%ml`.

[SBH96] M. Straßer, J. Baumann, et F. Hohl. Mole – A Java Based Mobile Agent System. Dans *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, University of Linz, Austria, juillet 1996.

[SKSH96] Christian Schuckmann, Lutz Kirchner, Jan Schummer, et Jorg M. Haake. Designing object-oriented synchronous groupware with COAST. Dans *Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, Language Support for Groupware, pages 30–38, 1996.

[SMK+94] M. Satyanarayanan, H. H. Mashburn, Puneet Kumar, David Steere, et J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1) :33–57, février 1994. http ://http ://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html.

[SS90] M. Satyanarayanan et E. H. Siegel. Parallel communication in a large distributed environment. *IEEE Transactions on Computers*, 39(3), mars 1990. http ://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html.

[Sun98] Sun Microsystems. Java blend. Technical report, 1998. `http://www.sun.com/software/javablend/index.html`.

[Syb99] Sybase. Sybase Replication Server : A practical architecture for distributing and sharing corporate information, 1999. `http://www.sybase.com/products/datamove/repserver_wpaper.html`.

[Tam97] Randy Tamura. *Lotus Notes and Domino 4.6 Unleashed.* SAMS Publishing, 1997.

[TC98] Michiaki Tatsubori et Shigeru Chiba. Programming support of design patterns with compile-time reflection. Dans *Proceedings of*

OOPSLA'98 Workshop on Reflective Programming in C++ and Java, pages 56–60, Vancouver, Canada, octobre 1998.

[TDP+94] Douglas Terry, Alan Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, et Brent Welch. Session guarantees for weakly consistent replicated data. Dans *International Conference on Parallel and Distributed Inormation Systems*, Austin, TX, US, septembre 1994. http ://www.parc.xerox.com/csl/projects/bayou/.

[Tic85] Walter F. Tichy. RCS : A system for version control. *Software - Practice and Experience*, 15(7) :637–654, juillet 1985.

[TTP+95] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, et Carl H. Hauser. Managing update conflict in bayou, a weakly connected replicated storage system. Dans *15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, US, décembre 1995. `http://www.parc.xerox.com/csl/projects/bayou/`,.

[vHGH+97] Arthur van Hoff, John Giannandrea, Mark Hapner, Steve Carter, et Milo Medin. The http distribution and replication protocol, août 1997. `http://www.w3.org/TR/NOTE-drp`.

[Whi94] J. E. White. Telescript technology : The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.

[WPE+83] Bruce J. Walker, Gerald J. Popek, R. English, C. Kline, et G. Thiel. The LOCUS distributed operating system. pages 49–70, octobre 1983.

[WRW96] Ann Wollrath, Roger Riggs, et Jim Waldo. A distributed object model for the Java system. Dans *Proceeding of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)*, Toronto, juin 1996. USENIX. `http://www.usenix.org/publications/library/proceedings/coots96/wollrath%.html`.

[XdSS97] Florian Xhumari, Cassio Souza dos Santos, et Marcin Skubiszewski. Java Universal Binding : Storing java objects in relational and object-oriented databases. Dans *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, août 1997.

— 17 octobre 2000 —