SOMMAIRE

Chapitre 3 : Conception et Définition du Modèle de Gestion de Version

Chapitre 4 : Implémentation du modèle

SOMMAIRE DES FIGURES

Chapitre 1 : Les Systèmes de Gestion de Versions & Les Web Services

Partie 1 : Les Systèmes de Gestion de Versions

Partie 2 : Les Web Services

Chapitre 2 : Les Systèmes Actifs & Les Règles ECA

Chapitre 3 : Conception et Définition du Modèle de Gestion de Version

Chapitre 4 : Implémentation du Modèle

La gestion de versions consiste à maintenir la traçabilité de l'ensemble des versions d'un ou de plusieurs fichiers, elle est utilisée essentiellement dans le domaine de l'implémentation des logiciels et s'est limitée surtout à la gestion des codes sources. Le besoin de la gestion de versions est apparu, quand les applications nécessitaient pour leurs fonctionnements des versions antérieures de leurs codes source, alors qu'ils disposaient uniquement de la dernière version [BOR 00].

Ensuite, la gestion de versions devenait de plus en plus essentielle dans les environnements où plusieurs utilisateurs manipulent une même base de ressources (outil de travail en groupe). Elle a été utilisée aussi dans ce type d'environnement afin de permettre à plusieurs développeurs de travailler sur un même projet en même temps et d'effectue la fusion des modifications non conflictuelles.

Actuellement, la gestion de versions est employée dans des secteurs divers vu son importance [HEI 04] [CHA 11]. Ces systèmes permettent l'accès concurrent de divers développeurs, le suivi de l'historique, la visualisation des différences entre les différentes versions et le retour à des versions antérieures [CAD 04] [WHI 04].

Les web services sont aussi des applications qui relient les programmes, les objets, les bases de données ou les processus d'affaires à l'aide des documents XML et des protocoles standards de web, ils sont utilisés afin d'envoyer des données destinées à être lues par des machines. Cependant, ils ne contiennent pas des spécifications ou des technologies standards qui gèrent l'évaluation dynamique des web services en terme de gestion de versions.

L'idée fondamentale derrière les web services est de morceler les applications et les processus d'affaires en morceaux réutilisables appelés « *Service* » de sorte que chacun de ces segments effectue une tâche distincte suivant un programme. Ces services peuvent alors servir à l'intérieur et à l'extérieur de l'entreprise via l'échange des données entre eux, facilitant l'interopérabilité entre tous ces services. Les web services offrent donc aux entreprises la flexibilité de réponse et d'anticipation des besoins changeants des consommateurs, la rationalisation des infrastructures logicielles et la flexibilité d'interaction et de configuration des alliances externes avec les partenaires et fournisseurs.

Face à ces situations, les consommateurs et les fournisseurs du service web ont besoin d'un système de gestion de versions qui leur permet de maintenir la trace de leurs anciens fichiers, de les informer des dernières versions disponibles au niveau du web service, de contrôler les mises à jour effectuées, de leur permettre l'accès à toutes les versions des fichiers du service via le web. Le système doit être utilisé sur n'importe quelle plateforme suivant les caractéristiques des web services. Les caractéristiques de développement et l'impact des web services imposent aux chercheurs de se focaliser sur ce manque de contrôle de versions dans les web services.

A ce jour, les serveurs web n'offrent pas la possibilité de garder les traces ou les modifications apportées à un document et tous les travaux réalisés par les chercheurs qui existent, restent non adapté au web services actuels. Les web services devront offrir un plus grand degré d'automatisation. Il est incontestable que la gestion de versions est un atout indéniable pour l'utilisateur, pour qu'il puisse accéder à l'historique de sa base et obtenir pour chaque document la traçabilité de tous les mouvements effectués au long de son cycle de vie.

Dans ce contexte, les objectifs de notre travail sont doubles :

- Nous proposons une nouvelle approche de gestion de versions de documents dans les web services. Cette approche utilise les deux stratégies « *Verrouiller-Modifier-Déverrouiller* » et « *Copier-Modifier-Fusionner* » définies dans les systèmes de gestion de versions existants.

- Afin de rendre actif notre modèle, certaines fonctionnalités sont dotées d'un système de règles actives, plus précisément les règles ECA (Evénement-Condition-Action). Les règle actives ECA nous fournissent une sémantique déclarative claire et offrent une réalisation immédiate des opérations sans l'intervention humaine.

Cette nouvelle approche est destinée aux échanges des informations entre les web services et leurs consommateurs qui sont des documents codés en XML, dans un système centralisé contenant un serveur et deux utilisateurs. Ce système centralisé est l'un des composants d'un système décentralisé global.

Le travail présenté dans cette thèse consiste en une conception et une implémentation d'un système de gestion de versions de documents permettant de gérer l'évaluation des documents entre ses utilisateurs, le contrôle d'accès aux documents par les utilisateurs, et le stockage de l'historique de l'ensemble des opérations effectuées sur les documents.

Ainsi, le travail présenté dans cette thèse s'articule autour de quatre chapitres suivants :

Chapitre 1: Les Systèmes de Gestion de Versions & Les Web Services

Ce premier chapitre comporte deux parties :

Partie 1 : Les Systèmes de Gestion de Versions

Ce chapitre est une présentation générale des systèmes de gestion de versions existants. Nous mettons en valeur les différentes stratégies utilisées pour la gestion de versions, et une brève description des différents systèmes de gestion de versions existants (locaux, centralisés, et décentralisés).

Partie 2 : Les Web Services

Cette partie décrit les caractéristiques des web services, les protocoles et les langages utilisés; ainsi que la nature des données échangées entre les différents services et leurs consommateurs de services, ensuite les caractéristiques de la gestion de versions dans les web services focalisée surtout sur l'interface du service et le contrat de service.

Chapitre 2: Les systèmes Actifs & Les Règles ECA

Ce chapitre est un état de l'art sur les systèmes actifs. Tout d'abord, nous introduisons les principaux constituants qui permettent de définir un système actif basé sur la règle ECA : le modèle de connaissance, le modèle de détection, et le modèle d'exécution. Puis, nous présentons une brève description de trois systèmes actifs existants : le système SAMOS, le système NAOS, et le système HIPAC.

Chapitre 3: Conception et Définition du Modèle de Gestion de Versions

Dans ce chapitre, nous présentons et nous définissons la solution proposée pour la gestion de versions de documents pour les échanges dans les web services de type XML via une architecture qui englobe les différents rôles fournis par cette proposition. Ensuite, nous décrivons chaque composant et module de l'architecture de l'approche proposée, ainsi que les outils dont dispose chacun d'eux.

Chapitre 4: Implémentation du Modèle

Ce chapitre, décrit l'environnement du travail et les différents outils nécessaires au développement du système de gestion de versions de documents tout en justifiant leurs utilisations.

Finalement, nous tirons les conclusions de notre travail et nous discutons des perspectives de travaux futurs.

Trois annexes complètent le travail de cette thèse notamment :

Annexe 1 : La liste des documents utilisés dans le système en codage XML (4 documents).
Annexe 2 : La liste des événements de modification des documents (24 exemples).
Annexe 3 : Le code source et du module *Contrôle du Verrouillage.*

1. Introduction

Les logiciels contiennent un ensemble de programmes [MEY 00] et évoluent constamment selon les besoins des utilisateurs dans le but de les améliorer. Le résultat après chaque amélioration du programme est appelé version ou révision. Les différentes versions sont nécessairement liées à travers des modifications (une modification est un ensemble d'ajouts, de suppressions, ou de modifications de données). Une modification constitue l'évolution entre deux versions. On peut donc aussi bien parler de la différence entre deux versions que de modifications ayant amené à une nouvelle version, et le passage d'une version à une version antérieure est établi en supprimant les dernières modifications appliquées au code source du programme [ALT 08] [GUE 11]. Mais avec un nombre important de versions, la récupération du code source d'un programme déterminé devenait de plus en plus difficile et l'existence de plusieurs versions du code source d'un programme obligeait les développeurs à implémenter un système qui permet la gestion de ces versions : c'est le « ***Système de Gestion de Versions »*** [HOC 10].

2. Définition des Systèmes de Gestion de Versions

Un Système de Gestion de Versions, en anglais Version Control System (VCS) ou Source Code Management (SCM) appelé aussi Versionnage (*Versionning*) est un logiciel contenant un ensemble d'outils qui permettent de stocker et de manipuler un fichier ainsi que toutes les révisions qu'il a subies depuis sa création [BRE 06].

Par exemple, lorsqu'un développeur modifie son code source et par prudence il enregistre régulièrement le code source, ensuite, il quitte l'éditeur du code pour compiler et tester la nouvelle version, puis il se rend compte que ça ne fonctionne pas comme il le désirait et qu'il faut retourner à la version précédente. S'il a ajouté des lignes de code il serait facile de retourner à la version précédente du code en supprimant les lignes ajoutées mais s'il a supprimé des lignes du code ça devient difficile de retrouver la version précédente du code. Pour ce faire, l'utilisation d'un VCS permet au développeur de revenir à un état stable s'il se trompe ou perd des fichiers [PET 10].

Le système de gestion de versions permet de récupérer un fichier ou un projet complet à des états précédents afin de pouvoir les corriger [WHI 04], de comparer les modifications au fil du temps, de détecter le responsable d'une modification erronée causant un problème au niveau d'un fichier, et de détecter la date de la modification. Il permet ainsi de conserver la chronologie historique de toutes les modifications qui ont été effectuées sur un fichier informatique, lorsqu'il est accompagné d'un commentaire contenant toutes les informations nécessaires de la création de la version.

Ces systèmes donnent aux utilisateurs la possibilité de travailler en parallèle, tout en gérant les conflits dus au travail en équipe sur les mêmes documents qui contiennent le code source du programme, afin d'éviter le blocage, et ils contrôlent l'accès et la disponibilité des documents tout en respectant la confidentialité du contenu.

Un commentaire dans les systèmes de gestion de versions est généralement rédigé par le responsable de la modification, où il explique les raisons de cette modification. Il est fortement conseillé d'écrire un commentaire ce qui facilitera la relecture future des modifications passées.

Ce type de système permet de gérer un code à la fois à travers :

- Le temps : Possibilité d'archiver l'ensemble des versions du code source et d'inverser (reversion) les modifications pour revenir à une version particulière,
- L'espace : Possibilité de faire partir le code dans plusieurs directions séparées (branching), et éventuellement de les rassembler.

Le système de gestion de versions s'applique principalement au code source, mais également à la documentation, aux bases de données, à tout fichier texte et parfois aussi aux logiciels. Les systèmes de gestion de versions sont utilisés notamment en ingénierie du logiciel pour conserver leur code source relatif aux différentes versions d'un logiciel. De ce fait, il est courant de parler de modifications pour un seul document (contenant le code source du programme) et d'ensemble de modifications lorsqu'il s'agit d'un logiciel (un ensemble de programmes), et les deux n'évoluent pas au même rythme. Chaque modification faite constitue une révision.

La gestion de versions est indispensable dans les environnements où plusieurs utilisateurs coexistent; elle leur permet d'accéder facilement et de manière concurrente dans de bonnes conditions à l'ensemble des versions des fichiers pour manipuler et modifier cette même base de ressources. Et un système de gestion de versions permet à un utilisateur de travailler sur une version d'un code et de développer par la suite une nouvelle version indépendamment des autres utilisateurs du système, ou de la rassembler avec une version d'un autre utilisateur tout en gérant les conflits. Un tel système permet à un groupe de développeurs de travailler autour d'un même développement et de stocker toute évolution du code source [WHI 04].

Pour garder l'historique des changements, les VCS introduisent la notion de branche, de tag, et d'arbre de version.

- ***La branche***

La branche est en général créée lorsqu'un développement secondaire est mis en route, que ce soit pour ajouter une nouvelle fonctionnalité ou parce que certains développeurs souhaitent essayer de prendre une autre direction pour certains aspects du développement. Le concept de branches offre la possibilité de dupliquer le code de base dans un point particulier dans le temps (voir figure 1) et permet de maintenir ces deux usuelles branches de développement [JGE 05].

Par exemple, lorsqu'une version d'un logiciel est distribuée, il est possible de créer une branche dans laquelle seront faites les corrections des bugs de cette version. Les corrections peuvent ensuite être répercutées dans le tronc si elles s'appliquent aussi aux développements en cours. Ceci permet de fournir une version corrective de la version distribuée.

Remarque : le tronc (trunk) est la version originale du fichier code source d'un programme.

- ***Le tag***

Les systèmes de gestion de versions marquent les événements importants dans le cycle de vie d'un code du projet par des étiquettes connues sous le nom de tag [DON 09].

- ***L'arbre de version***

Afin de suivre l'évolution d'un fichier donné, ces systèmes produisent un arbre de version. Ils génèrent la hiérarchie de l'arborescence des versions d'une manière automatique pour chaque fichier qu'ils contiennent. Le numéro de version (*renommage*) identifie la progression des différents nœuds de l'arbre de version au fil du temps. Plus le nombre est grand, plus la version est récente, et les chiffres apparaissent comme une extension pour montrer l'évolution [BAR 03] [AIX 10].

La numérotation proposée dans l'arbre de version aide les utilisateurs à identifier l'évolution du logiciel. Également, dans les systèmes Unix, elle les aide à maintenir les différences entre les versions relatives des fichiers.

Par exemple, l'arbre de version de la figure 1 est tel que les nœuds de l'arbre représentent les modifications appliquées au fichier et la racine de l'arbre est la version initiale du fichier.

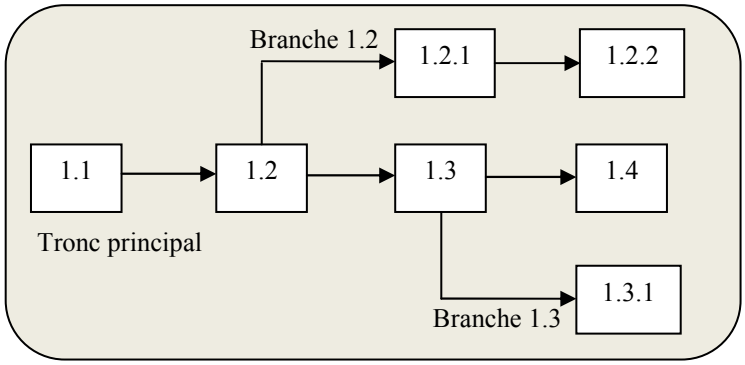


Figure 1. L'Arbre de version

3. L'Historique des Systèmes de Gestion de Versions

Pendant les années 70 les systèmes de gestion de versions ont été utilisés pour stocker et gérer les différentes versions des fichiers en particulier les codes sources. La réalisation du premier système de gestion de versions était le SCCS (Source Code Control System) en 1972.

Le SCCS [ROC 75] a été développé pour gérer le code source, la configuration des fichiers, ou des documentations. Il s'est focalisé sur la gestion des fichiers et des utilisateurs. Le SCCS a apporté des mécanismes qui permettent aux développeurs de garder la trace de l'historique d'un fichier, en utilisant le contrôle « *checksums* » pour détecter la corruption de données [BUR 98].

Le RCS (Revision Control System) [NGU 11] [TIC 85] est un autre système de gestion de versions développé dans les années 1980. Le RCS comme le SCCS supporte un seul utilisateur qui stocke simplement les différentes versions des fichiers dans un fichier système. Mais contrairement à SCCS, le RCS ne conserve aucune trace de l'historique d'un fichier. Cet outil fonctionne en conservant des ensembles de patch (la différence entre les versions d'un fichier) dans un format spécial sur le disque; il peut alors restituer l'état de n'importe quel fichier à n'importe quel instant en ajoutant toutes les différences.

En 1989 le célèbre CVS (Concurrent Version System) [BAR 03] [CED 93] a été développé afin d'améliorer le RCS. Ce système apporte plusieurs ajouts, étant donné qu'il supporte de

multiples fichiers et de multiples utilisateurs, mais l'avantage majeur de CVS était le support de fusions concurrent des fichiers au lieu d'utiliser un mécanisme de verrouillage; ceci permettait aux développeurs de collaborer dans leurs projets. Cependant le CVS avait certaines limitations concernant la manipulation des répertoires et des fichiers binaires. Avec l'extension de l'Internet, les utilisateurs de CVS critiquaient le manque de support à l'accès à distance; pour cela un groupe de développeurs a mis en œuvre une extension en 1994, afin d'utiliser le CVS sur TCP, en lui permettant d'être utilisé plus facilement sur Internet.

Ensuite dans l'année 2000, la société CollabNet a décidé de développer un système remplaçant le CVS. Ainsi, Ben Collins-Sussman, Brian Behlendorf et Jason Robbins de CollabNet ont débuté le projet de Subversion (SVN) [COL 08] [COL 11]. Le système Subversion est considéré comme un standard de l'industrie. C'est un logiciel Open-Source en conformité avec le Debian Free Software Guidelines (DFSG), qui a apporté de nombreux avantages pour les développeurs par une meilleure gestion des fichiers, y compris une structure de répertoire, mettant l'accent sur le projet au lieu que sur les fichiers seulement. Le SVN fournit une interface abstraite du dépôt pour une élaboration facile de différentes méthodes d'accès, tout en garantissant la cohérence [MAS 05].

Avec le développement de Bitkeeper [WOL 04] en 1997 il est apparu une nouvelle approche de gestion de versions, c'est les systèmes de gestion de versions décentralisés. Ces systèmes sont devenus de plus en plus populaires avec le développement de Git [LOE 09] [ZAC 12], Mercurial [SUL 09], Bazaar [BAZ 10] et Darcs [HOC 10].

Le système Git, est développé depuis 2005 comme un projet open-source initié par Linus Torvalds. Le projet Git a été lancé dans le but de remplacer le système propriétaire BitKeeper, utilisé pour maintenir le noyau linux et utilisable pour les projets open-source [HOC 10].

Certains de ces systèmes de gestion de versions seront détaillés dans la section 5.

4. Les Stratégies des Systèmes de Gestion de Versions

Les systèmes de gestion de versions récents fonctionnent selon deux différentes stratégies pour gérer l'accès concurrent aux fichiers communs et pour autoriser l'édition collaborative et le partage de ces fichiers : le modèle « Verrouiller-Modifier-Déverrouiller » et le modèle « Copier-Modifier-Fusionner ».

4.1 Le Modèle « Verrouiller-Modifier-Déverrouiller »

Dans le modèle « Verrouiller-Modifier-Déverrouiller » (*Lock-Modify-Unlock*), le système autorise seulement un utilisateur à la fois de travailler sur un document particulier. Le principe de ce modèle consiste à bloquer l'accès au document lors de sa modification, puis d'autoriser par la suite l'accès à ce document une fois la modification terminée.

Grâce à ce modèle, les développeurs ne pouvaient pas travailler sur un fichier « verrouillé » et ne pouvaient pas apporter leurs modifications parce qu'un document verrouillé était accessible par l'ensemble des utilisateurs du système uniquement en lecture. Cependant si un développeur souhaite modifier un fichier déverrouillé il devra le verrouiller et apporter ces modifications. Une fois ces dernières terminées, il doit procéder au déverrouillage du fichier modifié (voir figure 2). Ensuite si un développeur essaie de verrouiller un fichier déjà verrouillé, il devra attendre jusqu'à ce qu'il soit déverrouillé. Dans

ce modèle, il est nécessaire de négocier avec d'autres développeurs, avant de travailler sur les mêmes fichiers, même s'ils travaillent dans des différents domaines du code (les développeurs parfois oublient de déverrouiller les fichiers quand ils terminent leur travail) [BAR 03].

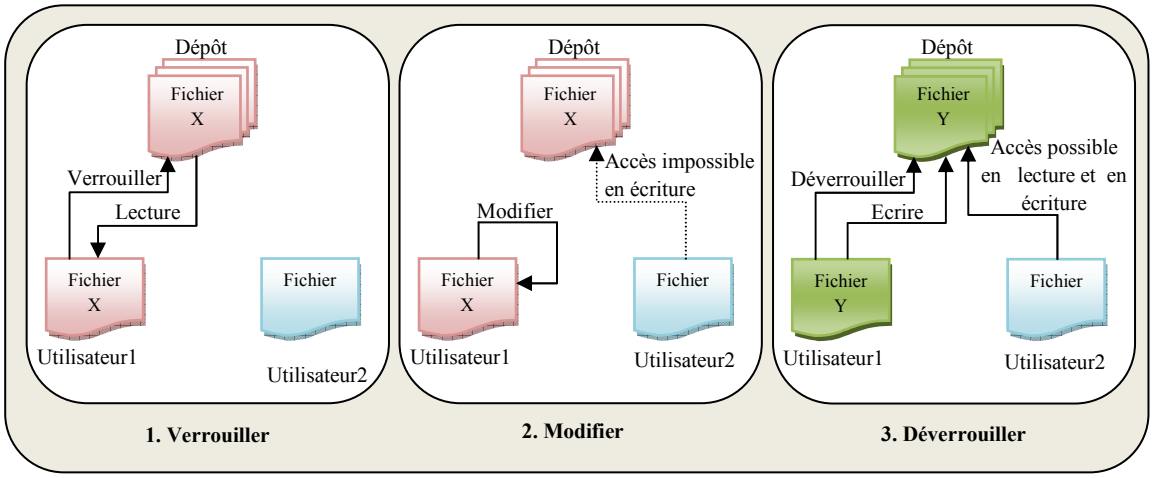


Figure 2. Le Modèle « Verrouiller-Modifier-Déverrouiller »

Le verrouillage est un peu restrictif, car il peut poser des problèmes administratifs pour les utilisateurs et pour la productivité du système, il interdit ainsi le travail en parallèle. Lorsqu'un utilisateur oublie accidentellement de déverrouiller un fichier alors qu'il a terminé avec, la situation finit par causer un retard et une perte de temps inutile. Aussi, le verrouillage peut causer une sérialisation inutile, quand un utilisateur édite dans une partie du fichier indépendante de celle que le deuxième utilisateur concurrent désire effectuer ses modifications.

Le verrouillage peut causer un faux sentiment de sécurité dans le cas où deux utilisateurs verrouillent et travaillent sur deux fichiers différents et que ces deux fichiers dépendent l'un de l'autre et le résultat de la modification des fichiers engendre des problèmes de compatibilité. Face à ce type de cas, le système reste incompétent aux différents problèmes qui surviennent après. Alors que les utilisateurs le pensaient sécurisé et propre parce qu'ils ont verrouillés leurs fichiers.

4.2 Le Modèle « Copier-Modifier-Fusionner »

Le modèle « Copier-Modifier-Fusionner » (*Copy-Modify-Merge*) est l'alternative du modèle précédent.

Dans ce modèle (figure 3), les utilisateurs désirant modifier un fichier doivent tout d'abord contacter le dépôt et extraire une copie du fichier du dépôt. Les utilisateurs ensuite peuvent travailler sur leurs copies simultanément et indépendamment les uns des autres et établir leurs modifications. Le premier utilisateur qui termine son traitement sur la copie envoie le résultat au dépôt, ce dernier renvoie la copie reçue au deuxième utilisateur qui n'a pas encore envoyé sa copie et dans ce cas, c'est à cet utilisateur de fusionner les deux copies au sein d'une nouvelle version finale puis de remettre le résultat au dépôt, qui à son tour, distribue cette copie aux autres utilisateurs du système [GUE 11].

Ce modèle est basé sur l'hypothèse que tous les fichiers sont contextuellement fusionnables. De même, les systèmes de gestion de versions qui utilisent ce modèle fournissent de l'aide aux utilisateurs afin de réaliser cette fusion; mais au final la responsabilité de s'assurer que tout se passe bien retombe sur les utilisateurs eux mêmes.

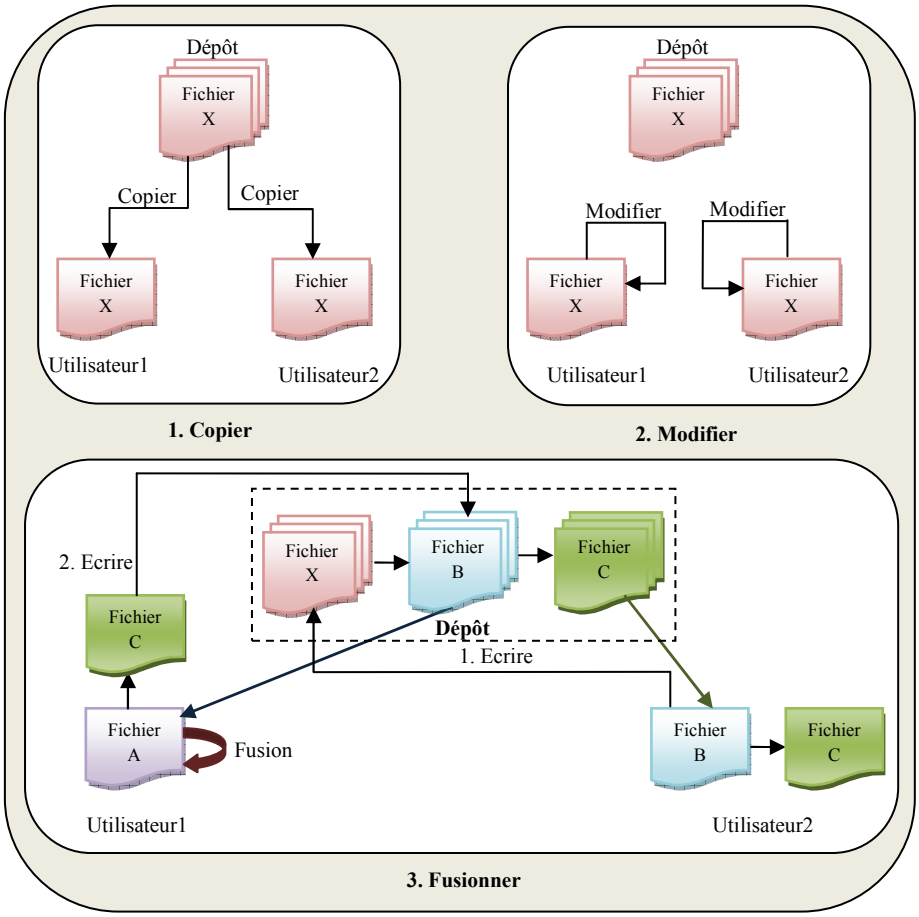


Figure 3. Le Modèle «Copier-Modifier-Fusionner»

Le modèle « Copier-Modifier-Fusionner » peut sembler un peu chaotique mais, en pratique, il fonctionne de façon très fluide. Quand les utilisateurs travaillent sur le même fichier, il est bien rare que leurs modifications concernent les mêmes parties du fichier et par conséquent les conflits sont rares. Et le temps nécessaire à la résolution des conflits est en général bien inférieur au temps gaspillé par un système de verrouillage.

Cependant, ce modèle engendre aux systèmes qui les utilisent, des problèmes de conflit. Pendant la fusion, si les modifications de l'un des utilisateurs recouvrent les modifications de l'autre, cela engendre un conflit chez l'autre utilisateur parce qu'il reçoit une version fusionnée différemment à celle fusionnée à son niveau pour les mêmes fichiers. Dans le cas où le système de gestion de versions ne peut pas résoudre automatiquement les conflits, les utilisateurs doivent résoudre le conflit manuellement entre eux.

Quand les utilisateurs communiquent mal, les erreurs syntaxiques et sémantiques peuvent survenir et les conflits vont augmenter. Et dans ce cas il est préférable de travailler avec le modèle « Verrouiller-Modifier-Déverrouiller » qui contrôle le conflit surtout qu'aucun système ne peut forcer les utilisateurs à communiquer. Mais dans la pratique, le verrouillage limite la productivité des conflits [COL 08].

Même si le modèle « Verrouiller-Modifier-Déverrouiller » est en général considéré comme pénalisant pour la collaboration, il y a quand même des cas où le verrouillage est approprié.

5. Modèles de Système de Gestion de Versions

Les systèmes de gestion de versions existants fonctionnent selon trois modèles : local, centralisé et décentralisé [HER 10].

5.1 Les Systèmes de Gestion de Versions Locaux

Le principe de la gestion de versions était de recopier les fichiers dans un autre répertoire. C'est une méthode simple mais moins fiable; étant donné qu'il est facile d'oublier le site du répertoire de travail et d'écrire accidentellement dans le mauvais fichier ou d'écraser des fichiers destinés à être conservés.

Afin de remédier à ce problème, les programmeurs ont développé des systèmes de gestion de versions VCS locaux qui utilisaient une base de données simple pour conserver les modifications d'un fichier. Les utilisateurs des VCS locaux disposent d'une base de données des versions contenant l'ensemble de toutes les versions du fichier et les utilisateurs effectuent une extraction (Checkout) afin d'accéder au fichier (voir figure 4) [CHA 11].

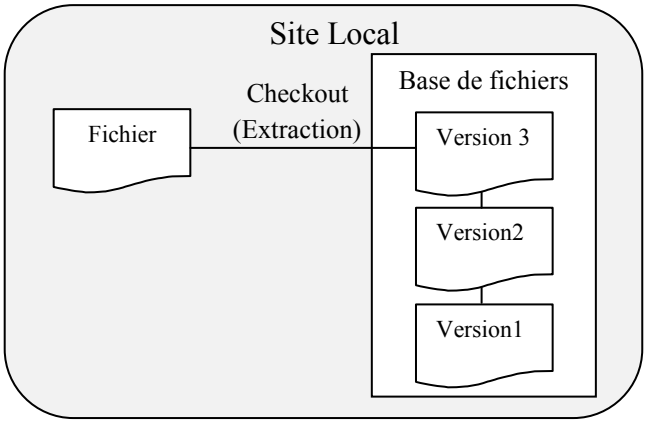


Figure 4. Diagramme des Systèmes de Gestion de Versions Locaux

Les systèmes de gestion de versions locaux comme le SCCS et le RCS sont conçus au dessus du modèle de fichiers du système d'exploitation (SCCS1 [ROC 75], RCS2 [TIC 85]). Ces outils permettent la distinction entre l'espace de travail et l'espace de stockage (figure 5).

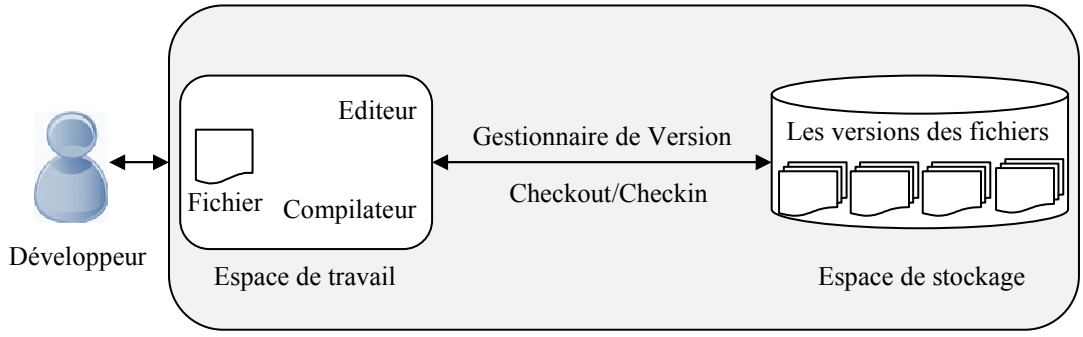


Figure 5. L'espace de Travail et L'espace de Stockage

Ces outils ne sont pas une partie intégrée dans le système de gestion de fichiers, pour cela il a été nécessaire d'ajouter des commandes spécifiques pour extraire (checkout) les fichiers de l'espace de stockage et les récupérer vers l'espace de travail de l'utilisateur, et inversement, pour les remettre (checkin) dans l'espace de stockage. Ce modèle appelé checkin/checkout est la base de la plupart des modèles des logiciels de gestion de versions [CAS 96].

Dans Unix, le SCCS et le RCS sont les deux packages les plus utilisés dans la gestion de versions. Ils ont une version abstraite de l'historique qui contient toutes les révisions d'un document particulier [WHI 04] [BRO 02].

- ***Le Système SCCS***

Le SCCS (Source Code Control System) est un système de gestion de versions de code source d'Unix développé par MJ Rochkind qui n'est pas un Open Source. Le SCCS était disponible en 1972 sur n'importe quel système Unix [TEL 84] [LOE 09]. C'est le premier VCS qui a défini les concepts de base des VCS. Ce système répond aux problèmes que l'on rencontre lorsqu'on doit maintenir des produits en versions multiples. Il garde une trace des modifications effectuées sur un programme en langage C, et permet de revenir en arrière à des états bien déterminés comme tous les VCS.

Le SCCS peut permettre à plusieurs versions d'un même fichier d'exister simultanément si les utilisateurs le souhaitent, ce qui peut être utile lors de l'élaboration d'un projet nécessitant de nombreuses versions de fichiers volumineux.

Le système dispose d'un fichier spécial appelé fichier SCCS ayant le préfixe « *s* » pour le distinguer des fichiers de texte ordinaire. Il dispose ainsi d'un répertoire particulier appelé « *référentiel* » où il garde une trace de toutes les mises à jour. Le site central où le système stocke ses données s'appelle « *dépôt* ».

Le SCCS utilise le verrouillage et ne dispose pas de technique de fusion. Le modèle de verrouillage fourni par le système est simple et propre à lui pour sérialiser le développement. Ce modèle n'est pas identique à celui du modèle « Verrouiller-Modifier-Déverrouiller » défini dans la section 4.1. Selon le SCCS, si un utilisateur a besoin d'un fichier pour l'exécuter ou pour tester un programme, il doit vérifier tout d'abord que ce fichier est déverrouillé. Cependant il doit le verrouiller afin d'éviter l'accès à un autre utilisateur. Lorsqu'il termine son traitement il doit déverrouiller le fichier. Cette méthode de travail par verrouillage a été améliorée et utilisée dans le système CVS (modèle « Verrouiller-Modifier-Déverrouiller ») [LOE 09] [PET 10].

Afin de ne pas créer un fichier distinct pour chaque version d'un fichier, le SCCS ne stocke que les modifications de chaque version d'un fichier. Ces modifications sont appelées des « *deltas* ». Les modifications sont suivies d'une table delta dans chaque fichier SCCS. Chaque delta possède un SID (SCCS IDentification number), qui est un numéro d'identification de quatre chiffres maximum. Le premier chiffre indique la version finale (*release*), le deuxième chiffre indique le niveau, le troisième chiffre indique la branche, et le quatrième indique la séquence. Aucun chiffre SID ne peut être égal à 0, donc il ne peut pas y avoir un SID de 2.0 ou de 2.1.2.0, par exemple.

Exemple d'un nombre SID qui spécifie la version finale 1, niveau 2, la branche 2, séquence 4: SID = 1.2.2.4.

Les principes de base du fonctionnement interne de SCCS est de tout stocker dans le fichier SCCS qui contient l'historique, d'archiver toutes les modifications et non pas les versions complètes, et de traduire la ligne modifiée dans le fichier historique par une ligne effacée ou insérée. De ce fait, le SCCS dispose des commandes placées dans l'annuaire /usr/ccs/bin pour créer les liens dans l'arbre de version.

Remarque : Le système SCCS contient uniquement la dernière version, et les versions antérieurs sont conçues à partir des deltas du fichier SCCS.

Le schéma suivant, illustre un exemple de l'utilisation du système SCCS.

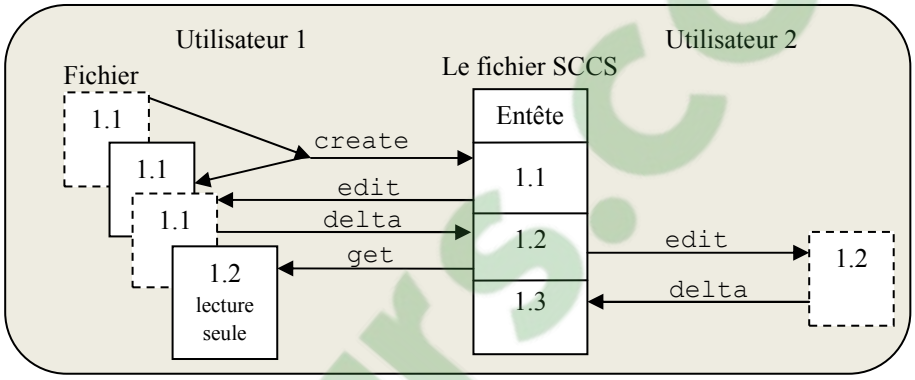


Figure 6. Utilisation du Système SCCS

Les commandes `sccs create <pgm>`, `sccs edit <pgm>`, `sccs delta <pgm>`, `sccs get <pgm>` permettent respectivement de placer le fichier sous le contrôle de SCCS en créant un nouveau fichier historique tout en utilisant le texte complet du fichier comme version initiale, d'extraire une version pour l'édition, de marquer les modifications (une contribution aux opérations d'édits de SCCS), et de récupérer une copie en lecture seule du fichier à partir de s.fichier (par défaut c'est la version la plus récente).

Le SCCS comprend des limites d'ouverture puisqu'il est pratiquement impossible à un non-initié de reconstituer un texte géré par le SCCS, à part quelques grands projets spécifiques à une équipe permanente ou à un constructeur, comme le noyau SunOS, ceux-ci l'ont donc spontanément abandonné [BOR 00]. Le SCCS stocke les versions de deltas dans plusieurs fichiers différents et reconstruit une version complète d'un fichier source à partir d'un ground up (écrasement) à chaque fois que le fichier est accessible ce qui rend le traitement long [BRO 02]. Aucune amélioration de SCCS, n'a été effectuée. Et ce système est devenu un outil lourd et ne convient pas à un grand nombre d'utilisateurs qui travaillent sur un projet. Par conséquent, le SCCS n'est pas recommandé pour les nouveaux projets, mais il est toujours présent pour soutenir la base ancienne du code [VAS 03].

- ***Le Système GNU RCS***

Le système GNU RCS (Revision Control System) est le deuxième VCS développé en 1982 par Walter F. Tichy au sein de l'université Purdue [TIC 85]. Ce logiciel représente à l'époque une alternative libre au système SCCS, et une évolution technique, notamment par son interface qui est plus facile pour les utilisateurs apprentis (figure 7), et la rapidité dans la récupération des données par l'amélioration du stockage des différentes versions. Ce gain de

performance provient de l'algorithme « *reverse differences* » (deltas) qui consiste à stocker la copie complète des versions les plus récentes et de conserver uniquement les modifications réalisées. Le RCS a présenté des concepts de stockage efficace pour les différents fichiers de révisions [LOE 09] [ZAC 12].

Le logiciel RCS était écrit en langage assembleur pour les systèmes UNIX. Il peut être appliqué à toute sorte de situations de développement, y compris la création de documents, images, formes, articles, et bien sûr, le code source.

Le RCS est aujourd'hui distribué avec de nombreux systèmes d'exploitation tels que le système Mac OS X qu'il inclut pour l'installation des outils de développement logiciel. Cet outil fonctionne en conservant des ensembles de *« patch »* (la différence entre les versions d'un fichier) dans un format spécial sur le disque; il peut alors restituer l'état de n'importe quel fichier à n'importe quel instant en ajoutant toutes les différences.

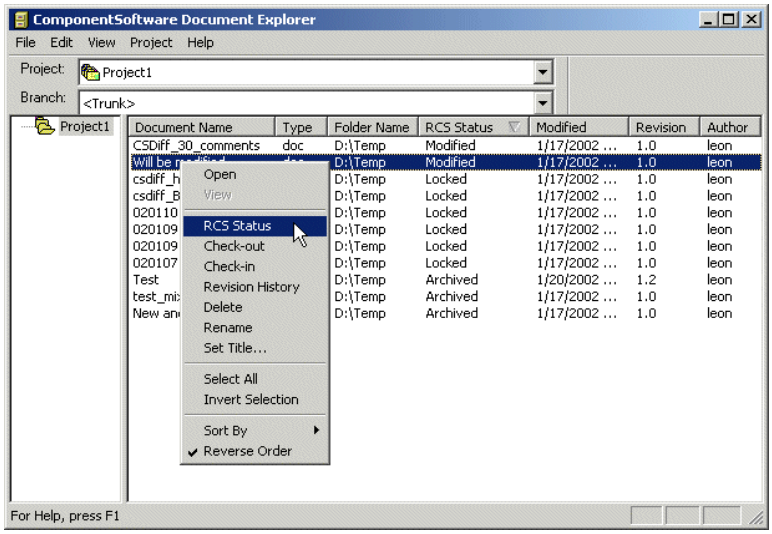


Figure 7. L'Interface du Système RCS en Cours d'Exécution

Le RCS gère les multiples révisions de fichiers en automatisant le stockage, la récupération, le log (journalisation), l'identification, et la fusion des révisions [JAY 11] [SCH 95]. Il stocke les dernières révisions d'un fichier dans des archives avec l'extension ",v", par exemple pour un fichier appelé "doc.sas", les archives de RCS s'appelleront "doc.sas, v". Ces archives contiennent une copie de la dernière version du fichier, le contenu des versions précédentes, et les informations sur les diverses versions. Le RCS traite les projets d'une manière centrée sur un fichier. Pour cela il utilise le modèle de développement « Verrouiller-Modifier-Déverrouiller » [BAR 03].

Généralement les révisions RCS sont numérotées de façon séquentielle, par exemple, 1.1, 1.2, et ainsi de suite [HUN 92].

Dans le RCS, toutes les modifications apportées à un fichier sont automatiquement stockées dans un fichier journal dans le sous-répertoire RCS, où il sauvegarde le texte de chaque révision, l'identification d'auteur, la date et l'heure des inscriptions, et le message de notation qui récapitule les modifications effectuées. Ensuite, le système met à jour ces informations automatiquement. C'est l'un des meilleurs dispositifs du RCS.

Le système RCS apporte les avantages suivants :

- Grâce à la bibliothèque RCS, il augmente considérablement la productivité des développeurs. Comme il donne la possibilité de les organiser de manière centralisée et décentralisée.
- Permet le développement concurrent par de multiples développeurs.
- Utilise les logs pour trouver des bugs et gérer la procédure de développement.
- Identifie automatiquement chaque révision avec le nom, le numéro de révision, la date de création, auteur, etc
- Marque les conflits de codage : si deux lignes ou plus de développement modifient la même section du code, le RCS peut alerter les programmeurs sur les modifications.
- Résout les conflits d'accès : lorsque deux ou plusieurs programmeurs souhaitent modifier la même révision, le RCS avertit les programmeurs et permet de s'assurer qu'une modification n'éliminera pas l'autre.
- Fournit une version finale et un contrôle de configuration : les révisions peuvent être marquées comme version finale, stables, ou routinières.
- Fournit une visibilité de haut niveau pour la gestion : il facilite le suivi de l'état d'un logiciel.
- Compatibilité avec les outils existants de développement de logiciel.
- Le RCS est mis en œuvre avec les deltas inverses. Cela signifie que la dernière révision, qui est la plus accessible, est stockée intacte. Toutes les autres révisions peuvent être régénérées à partir de la plus récente en appliquant les deltas inverses (différences en arrière).

Comme tout système de gestion de versions, le système RCS dispose d'un ensemble de commandes [NGU 11] [PET 10]. Il est à noter que, les commandes RCS sont plus délicates que celles de SCCS. Cependant, il y a moins de commandes que dans SCCS; le système est donc plus cohérent tout en ayant une plus grande variété d'options.

La fusion introduite dans le système RCS et la gestion des branches sont estimées très lourdes. Ainsi les principales limites de ce système se résument sur l'utilisation d'un seul utilisateur à la fois et une seule copie de travail [VAU 07]. Il repose sur le système de fichiers natifs comme dépôt de données (même disposition pour SCCS mais ce dépôt est subdivisé en sous-parties nommées *modules*, correspondant généralement à des projets distincts). L'utilisation du système de fichiers natifs fournit une stabilité essentielle, mais expose également des informations potentiellement sensibles ou des informations de la propriété de toute personne ayant accès au système de fichiers [BRO 02]. Néanmoins le RCS reste intéressant pour certains cas simples.

5.2 Les Systèmes de Gestion de Versions Centralisés

Le problème majeur rencontré avec les systèmes locaux est le besoin de collaborer avec des développeurs sur d'autres ordinateurs. Pour traiter cela, les systèmes de gestion de versions centralisés CVCS (Centralized Version Control System) furent développés. Ces systèmes tels que CVS, Subversion, et Perforce [PER 12], mettent en place un serveur central qui contient tous les fichiers sous une gestion de versions, et les clients peuvent extraire les fichiers de ce dépôt central. Pendant de nombreuses années, cela a été le standard pour la gestion de versions (figure 8) [CHA 11].

Les systèmes centralisés apportent de nombreux avantages par rapport à la gestion de versions locale; étant donné qu'un utilisateur sait jusqu'à un certain niveau ce que tous les autres sont en train de faire sur le projet. Les administrateurs ont un contrôle sur l'accès, comme il est beaucoup plus facile d'administrer un système de gestion de versions centralisé que de gérer des bases de données locales.

Avec les logiciels de CVCS (figure 8), il n'existe qu'un seul dépôt des versions qui fait référence. Cela simplifie la gestion de versions mais il reste difficile pour certains usages comme le travail sans connexion réseau, dans la mesure où quand le serveur est en panne pendant une durée, aucun utilisateur ne peut collaborer ou enregistrer les modifications issues de son travail. Quelque soit la durée de la panne, l'interruption influencera sur l'impact du travail, ou tout simplement lorsqu'on travaille sur des branches.

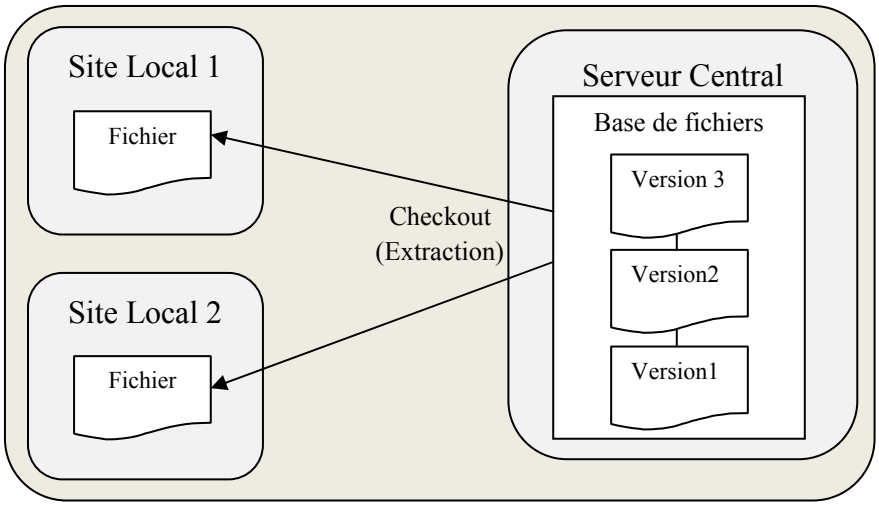


Figure 8. Diagramme de la Gestion de Versions Centralisée

- ***Le Système CVS***

Le CVS (Concurrent Version System) est un système écrit en langage Pascal par Dick Grune en 1986 puis complété par Brian Berliner en 1989, distribué sous la licence GPL (GNU Public License) et destiné à des groupes de développeurs travaillant sur les mêmes fichiers. Il permet aux développeurs de conserver leurs projets sur le dépôt accessible via une URL locale ou distante. Les développeurs comme dans les autres VCS, peuvent demander des versions antérieures d'un fichier particulier, regarder l'historique des modifications, et travailler à plusieurs sur les mêmes objets en même temps, et autres actions utiles [VAS 03].

Avec le CVS, chaque utilisateur travaille sur une copie de l'original et toutes les modifications des originaux passent par le système. Les fichiers originaux sont modifiés quand un utilisateur envoie ses nouvelles versions. Les anciens fichiers ne sont jamais détruits (ce qui évite l'écrasement accidentel d'un projet par un utilisateur); seules les différences sont enregistrées. Afin de synchroniser la copie de travail avec le dépôt il faut mettre à jour la copie locale de l'utilisateur et corriger manuellement les éventuels conflits qui peuvent apparaître, avant de pouvoir valider. Et la synchronisation se fait soit :

- En répercutant dans sa copie locale les modifications faites dans le dépôt par d'autres utilisateurs : c'est la mise à jour (*update*). La mise à jour consiste à synchroniser la

copie de travail locale avec le dépôt en récupérant la dernière version des fichiers du dépôt. C'est à cette occasion que des conflits de versions peuvent apparaître.
- En répercutant ses propres modifications dans le dépôt : c'est la validation du commit (Le commit est l'opération inverse d'un update, et consiste à mettre à jour le dépôt à partir de la copie de travail locale).

Un développeur peut être informé aussi des modifications des autres développeurs, soit par la configuration du système CVS, soit par la notification par mail lorsque l'un d'entre eux effectue un commit [BOR 00].

La validation ne peut avoir lieu que si les modifications sont intervenues dans le dépôt depuis la dernière mise à jour locale. Il faut alors mettre à jour sa copie locale et corriger manuellement les éventuels conflits qui peuvent apparaître, avant de pouvoir valider [GUE 11].

Ainsi dans le CVS, chaque fichier possède son propre historique, comme dans les VCS précédents, et il est possible de garder un état symbolique du développement à l'aide du tag [MOL 06] d'une version finale d'un projet à un moment donné et de pouvoir le rappeler une seule fois quand cela est nécessaire [GUE 11].

Ce système exige aussi d'utiliser les branches uniquement quand une version du logiciel est distribuée, lorsqu'un grand développement implique des modifications importantes qui influent sur la logique du code, et enfin dans le cas de développement expérimental qui ne conduit pas à une intégration dans le tronc principal de l'arbre de version [GUE 11].

Concernant les révisions de fichiers, chacune doit être accompagnée d'un commentaire de l'utilisateur selon les modifications effectuées. Ce commentaire est très utile pour la relecture future des modifications passées, par lui-même comme par les autres.

Le travail coopératif entre ses différents utilisateurs est basé sur la technique « Copier-Modifier-Fusionner ».

L'ensemble de commandes offertes pas CVS permettent aux utilisateurs de récupérer, mettre à jour des fichiers, et exécuter d'autres fonctionnalités. Par exemple, la commande `cvs checkout <projet>` : permet de récupérer un projet.

Dans le système CVS, c'est à l'administrateur de créer le dépôt (repository) et de créer les modules. L'administrateur peut personnaliser le comportement de CVS grâce à un système de triggers définis dans les fichiers de configuration *loginfo*. Ce fichier contient deux colonnes, une expression régulière destinée à être confrontée à des chemins contenus dans le dépôt et une commande à exécuter dans ce cas.

Le CVS supporte quelques faiblesses surtout pour les utilisateurs qui passent d'un développement anarchique à un développement coopératif [GUE 11] [BOR 00] [MOL 06] :

- La gestion des fichiers binaires est approximative.
- Le CVS ne supporte pas le renommage d'un fichier, par conséquent si on change le nom d'un fichier, on perd tout l'historique de ce fichier.
- Le travail sur les branches est lent et difficile à utiliser.
- Les répertoires et les méta-données ne sont pas versionnés.

Le CVS dispose d'un système de contrôle de versions fiable, bien que limité mais gère comme même les développements concurrents. La communication entre le répertoire unique et les espaces de travail clients est effectué via le TCP/IP par les protocoles RSH (remote shell) et SSH (Secure Shell) [VER 06].

Malgré ces points faibles, le CVS est utilisé avec succès pour des projets de taille raisonnable en associant à l'outil une méthode de travail et une rigueur dans l'utilisation. Cependant, c'est certainement à cause de ces points faibles que de nombreux projets alternatifs l'ont évité récemment.

- ***Le Système Subversion (SVN)***

Le système Subversion (SVN) aussi surnommé « CVS++ » est un logiciel gratuit et un open source, distribué sous licence Apache et BSD et fonctionnant sous Linux, Windows et Mac OS. Il a été conçu pour remplacer le CVS. Ce système a été écrit afin de combler certains manques de CVS. Les utilisateurs peuvent passer de CVS à Subversion en quelques minutes, cependant c'est plus difficile pour l'administrateur car le changement est plus profond et le format du dépôt est complètement différent.

Le choix de Subversion par les développeurs est justifié par la facilité dans l'administration par rapport à celle de CVS, en plus il supporte plusieurs modes d'accès distants, dont SSH et WebDAV via Apache [COL 08].

Le système Subversion reprend le modèle de CVS en comblant le manque du renommage automatique et du déplacement de fichiers sans perte de l'historique, la gestion des répertoires, les commits atomiques (un ensemble de modifications sera soit entièrement validé dans le dépôt soit pas du tout), la gestion des métadonnées (ex : permissions), la possibilité de migrer de CVS vers SVN sans perte de l'historique (cvs2svn), et l'introduction des protocoles réseaux sécurisés (HTTPS).

Dans le Subversion, chaque patch a un numéro de révision unique. Il devient simple de se souvenir d'une version particulière d'un projet, en ne retenant qu'un seul numéro (le renommage du système CVS n'est pas supporté par Subversion parce qu'il est manuel et peut casser l'historique). Les numéros de révisions sont globaux contrairement aux autres VCS, et les révisions sont gérées sur toute la hiérarchie des fichiers alors que dans le CVS ils sont gérés fichier par fichier.

Une autre différence est que les tags des autres systèmes sont des points dans le temps, alors que Subversion recommande de définir les tags comme des points dans l'espace du système de fichiers. Un "tag" Subversion n'est qu'une copie. Cette absence de tag au sens habituel rend certaines opérations un peu moins pratiques dans Subversion. Pour pallier à ces manques, l'ajout de labels « *alias* » a été proposé sur les listes de discussions de SVN. Ces labels seraient équivalents aux tags d'autres systèmes comme CVS, git ou autre [AGO 09] [GUE 11].

Le système Subversion ne fait aucune distinction entre un label, une branche et un répertoire. C'est une simple convention de nommage pour ses utilisateurs. Il devient ainsi très facile de comparer un label et une branche ou autre croisement.

De même, ce système donne à ses utilisateurs la possibilité de travailler selon le modèle « Verrouiller-Modifier-Déverrouiller » ou le modèle « Copier-Modifier-Fusionner » selon leur exigence [PET 10].

L'installation de Subversion permet l'utilisation de plusieurs programmes tels que:

- svn, le programme client en ligne de commande,
- svnversion, le programme qui permet de connaître l'état d'une copie de travail,
- svnlook, un outil d'interrogation directe du dépôt,
- svnadmin un outil permettant de créer, de paramétrer et de réparer le dépôt,
- svndumpfilter, un programme de filtrage des dumps (modifier l'historique) [COL 08] de repository svn,
- mod_dav_svn, un plug-in pour le serveur apache, afin de rendre le dépôt disponible sur le réseau,
- svnserve, un serveur svn, lançable comme un démon ou invocable par SSH et,
- svnsync, un programme de recopie incrémentale de dépôt.

Le Subversion est sensible aux erreurs réseaux, notamment lors de la création des commits volumineux et la rectification de ces commits est difficile. Étant donné qu'il n'existe pas de commit sans publication, d'autres utilisateurs peuvent utiliser ou reprendre les commits défectueux avant leur correction par d'autres commit. Aussi, les utilisateurs de Subversion trouvent des difficultés dans la fusion des branches, c'est un domaine où le subversion reste fragile. Un autre point faible de Subversion serait la gestion nécessairement manuelle du système de fichiers. Il est nécessaire de passer par les commandes de Subversion pour les déplacer, copier ou supprimer pour ne pas rompre l'historique des modifications où il est possible que les utilisateurs placent ces fichiers là où ils ne devraient pas l'être [DEV 11].

5.3 Les Systèmes de Gestion de Versions Décentralisés

Les systèmes de gestion de versions décentralisés (Distributed Version Control Systems DVCS) [CHA 11] [KUH 10] comportent plusieurs dépôts pour un même logiciel (figure 9). Ils permettent à chaque utilisateur de travailler à son rythme afin de se désynchroniser des autres, puis d'offrir un moyen à ses développeurs de s'échanger leurs travaux respectifs. Ce système est très utilisé par les logiciels libres.

Dans un DVCS les utilisateurs n'extraient plus la dernière version d'un fichier, mais ils dupliquent le dépôt au complet. Ainsi, si le serveur s'interrompt, n'importe quel dépôt d'un des utilisateurs peut être copié sur un nouveau serveur pour restaurer le système, et l'extraction devient une sauvegarde complète de toutes les données.

De plus, un grand nombre de ces systèmes gère particulièrement bien le fait d'avoir plusieurs dépôts avec lesquels les développeurs travaillent. Ces derniers peuvent collaborer avec différents groupes de développeurs de manière différente simultanément dans le même projet. Cela permet la mise en place de différentes chaînes de traitement qui ne sont pas réalisables avec les systèmes centralisés, tels que les modèles hiérarchiques.

Les utilisateurs de DVCS pourront suivre le système en amont, en ajoutant quelques patchs qui ne seront pas forcément intégrés. Ils utilisent ainsi la notion de fork (voir le fonctionnement du système Monotone). Les DVCS permettent de ne pas être dépendant d'une

seule machine, et permettent le travail privé pour réaliser des brouillons sans avoir besoin de publier ses modifications et gêner les autres utilisateurs.

Cependant les systèmes décentralisés ne supportent pas de « verrou »; ce qui peut causer des problèmes pour les données binaires qui ne se fusionnent pas. Concernant l'opération « copier », elle est plus longue que récupérer une version dans la gestion de versions décentralisée car tout l'historique est copié (ce qui est un avantage) [CHA 10].

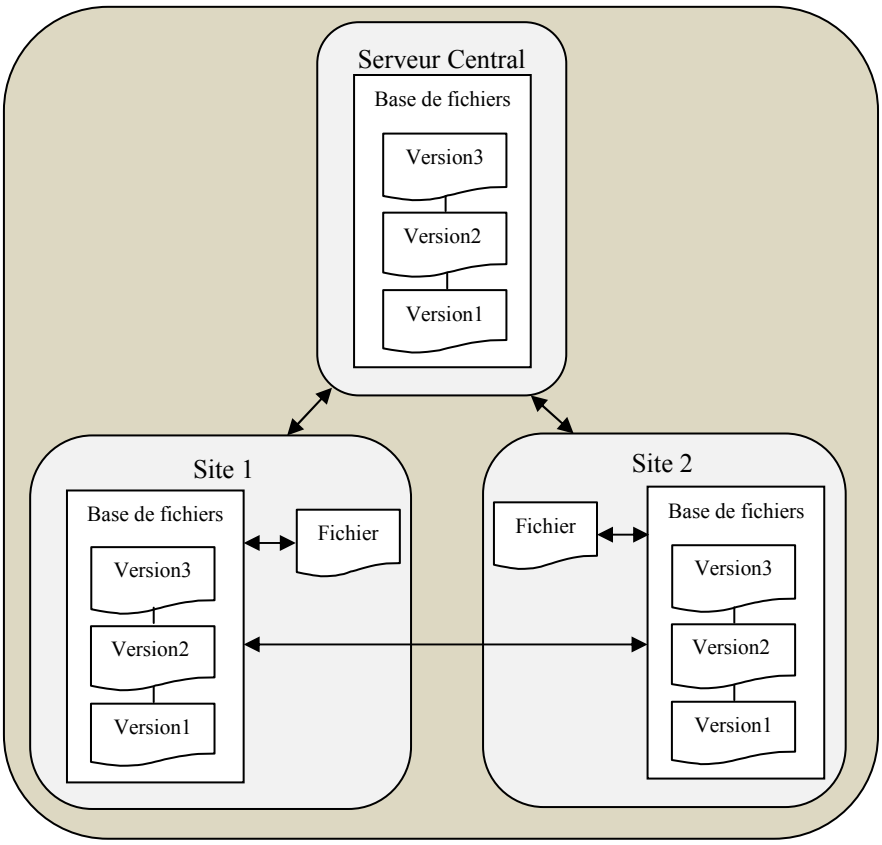


Figure 9. Diagramme de la Gestion de Versions Décentralisée

Les recherches dans le domaine, ont abouti à l'émergence de plusieurs DVCS : le système Bitkeeper (Le premier DVCS développé par Larry McVoy pour le noyau Linux) [JGE 05] [BAR 03] [WOL 04], Mericurial [BOR 05] [LYN 07], Bazzar, Darcs [CHA 11], Git [LOE 09] [ZAC 12], Monotone [HOA 11] [JGE 05], etc…

Pour la suite nous décrivons, brièvement, le système Monotone et le système Git.

- ***Le Système Monotone***

Le Monotone est un DVCS libre écrit en C++ en 2003 [BOR 05]. Cet outil fournit une protection des données accrue grâce à la cryptographie, notamment via *SHA*-1 (le *SHA*-1 « Secure Hash Algorithm – 1 » est une fonction de hachage cryptographique et le résultat produit est appelé « hash ») [EAS 01], et *GPG* (le *GPG* « GNU Privacy Guard » permet la transmission des messages signés ou chiffrés, et il garantit l'authenticité et la confidentialité des message) [GNU 99]. La communication réseau se fait via un protocole propre, appelé *Netsync* [CAR 00].

Le Monotone propose un système simple pour le stockage de versions, utilisant un fichier unique et des transactions. Les opérations s'effectuent hors ligne et le système utilise un protocole de synchronisation pair-à-pair efficace. Il supporte la fusion de versions basée sur l'historique, un système simplifié de branches, la vérification de code intégrée et les tests externes. Monotone fonctionne sous Linux, Solaris, OSX et Windows, comme il est placé sous licence GNU GPL [FOG 11] [JGE 05].

Ce système a été le premier à demander systématiquement la signature cryptographique de chaque patch. En effet, un DVCS est à priorité sensible aux *Chevaux de Troie* puisqu'il va recevoir des patchs d'origine très diverses. La confiance n'étant pas transitive car si un utilisateur fait confiance aux patchs du dépôt B et que le dépôt B faisait confiance aux patchs du dépôt C, l'utilisateur se retrouve avec les patchs de C qu'il ne connait pas. La signature cryptographique est pénible en pratique mais peut apporter une utile traçabilité et garantit la fiabilité de l'origine et du contenu des fichiers. De même, ce système est handicapé par son serveur spécifique qui rend difficile la traversée des pare-feu (les autres VCS utilisent en général HTTP et SSH) [BOR 05].

Monotone possède sa propre interface graphique « *le MonotoneViz »* et un algorithme de fusion amélioré. En revanche ce système n'attribue pas de numéros de versions aux données, mais se base sur leur hash code, ce qui est plus sûr, mais confus pour l'utilisateur. Les versions des fichiers binaires sont différenciées et archivées de manière incrémentale, grâce à la technologie xdelta (également un projet libre) [JGE 05].

Monotone déplace les informations dans différents types de stockage :

- Un keystore (stockage de clé) dans le répertoire local.
- Un workspace (espace de travail) dans le système de fichier local.
- Une base de données locale dans le système de fichier local.
- Une base de données distante accessible par l'Internet.

Toutes les informations passent à travers la base de données locale (voir figure 10). Par exemple, lorsque des modifications sont apportées dans un espace de travail, il est possible d'enregistrer ces modifications à la base de données correspondante, et plus tard il sera possible de synchroniser la base de données avec un autre utilisateur. Le Monotone ne déplacera pas les informations entre l'espace de travail et la base de données distante, ou entre les espaces de travail, mais il utilise la base de données locale comme point de commutation pour la communication entre tous les utilisateurs.

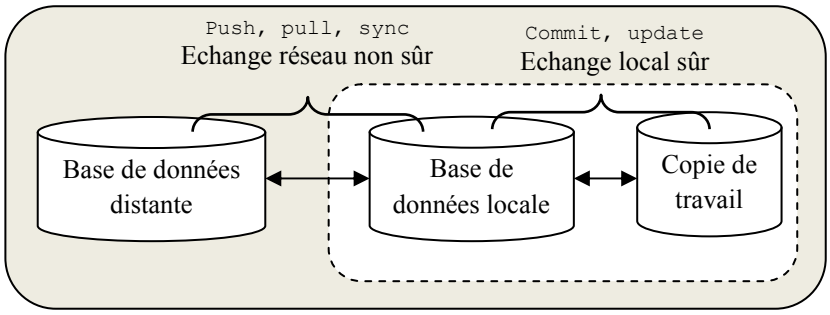


Figure 10. Les Echanges des Données entre les Différents Répertoires

L'espace de travail est l'arbre des fichiers qui se trouve dans le système de fichier. Le « _MTN » est un répertoire spécial qui existe dans la racine de tout espace de travail. Le système conserve certains fichiers spéciaux dans le répertoire «_MTN», afin de suivre les modifications apportées par l'utilisateur sur son espace de travail.

Le système donne la possibilité de modifier directement les fichiers dans un espace de travail en utilisant un éditeur de texte ou un autre programme, ensuite il prend note chaque fois que l'utilisateur effectue ce type de modification et les inclut dans le prochain commit. Si l'utilisateur ajoute, supprime, ou déplace des fichiers dans sa zone de travail, il doit alors expliquer ces opérations parce que ces actions ne peuvent pas être déduites. Le Monotone stocke ensuite ces modifications dans « *_MTN/revision* » qui feront partie du prochain commit. La validation et la mise à jour auront lieu entre la base de données locale et l'espace de travail sans l'implication du réseau (figure 11).

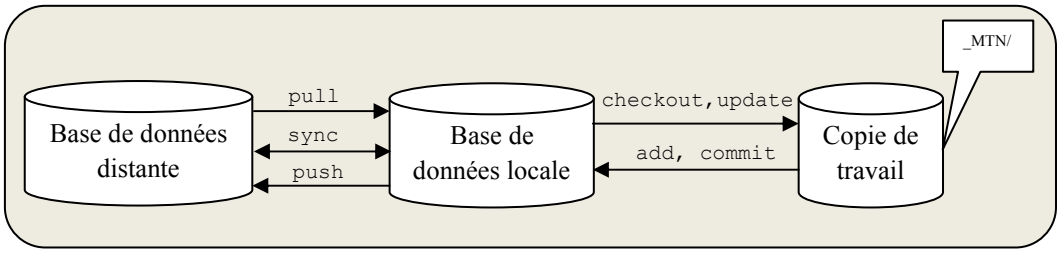


Figure 11. Rapport entre la Base de Données Locale et la Copie de Travail et entre la Base de Données Locale et la Base de Données Distante

La commande `mtn push` établit les modifications à une base de données distante, la commande `mtn pull` rapporte la nouvelle version d'une base de données distante, `mtn add` ajoute un répertoire ou un fichier à la copie de travail, `mtn sync` combine les deux opérations le pull et le push afin de se synchroniser, et la commande `mtn serve` rend la base de données disponible pour les autres utilisateurs afin de pouvoir la synchroniser avec les autres.

Si un utilisateur donné réalise une nouvelle révision, cette dernière est appelée « *révision fils* » (figure 12). Et les autres utilisateurs peuvent réaliser simultanément de nouvelles révisions, qui pourraient donc être dérivées du même parent (fichier) que celui de l'utilisateur donné en contenant des modifications différentes.

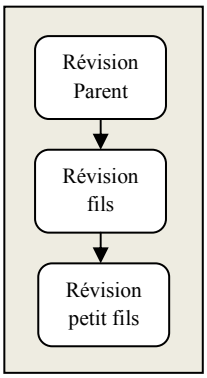


Figure 12. Le Versionnement d'un Fichier sous Monotone

Dans le cas d'édition simultanée qui produit deux révisions à la fois, le raisonnement établi par le système Monotone est la production de deux fichiers fils, gauche et droit : c'est le principe du fork (figure 13). Souvent l'ensemble de modifications entre le fichier parent (fichier racine) et le fils gauche n'est pas lié à l'ensemble de modifications entre le fichier parent et le fils droit [HOA 11]. Pour cela le système envisage la fusion, ce qui conduit à la production d'un petit fils de révision commun contenant les modifications des deux utilisateurs, et la fusion est déclenchée par l'un de ces utilisateurs.

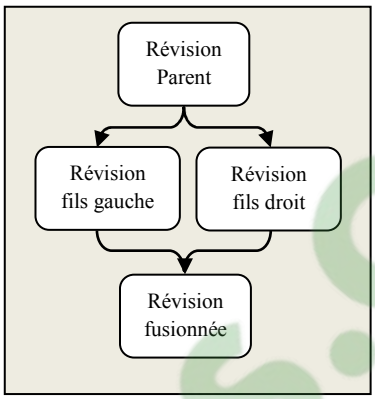


Figure 13. Principe du Fork

Les utilisateurs peuvent aussi produire les branches pour les révisions qui ne sont pas censées être fusionnées; afin de pouvoir travailler de façon autonome pendant un certain temps, ou quand ils voudront changer leurs fichiers de manière à ce que leurs nouvelles révisions ne soient pas logiquement compatibles avec les autres.

Les avantages de l'utilisation du système Monotone sont aperçus dans le contrôle complet des versions et des développements concurrents, l'architecture distribuée P2P, l'accent sur la sécurité des données (cryptographie, réplication), la différentiation des fichiers binaires, l'interface graphique, et le projet actif [JGE 05].

Le logiciel Monotone est techniquement ambitieux et novateur, mais il se trouve sur un stade de développement prématuré et ne propose qu'une version bêta du logiciel qui est peu répandue. Ainsi le protocole réseau utilisé dans ce système fonctionne par propriété, et ne dispose d'aucune approche de gestion de processus, et d'identification des versions de fichiers puisqu'ils sont basés sur leurs codes et difficilement interprétables par l'utilisateur [JGE 05] [MAL 06].

- ***Le Système Git***

Ce système est, à l'origine, développé par Linus Torvalds pour proposer une alternative libre au logiciel propriétaire de BitKeeper qui était utilisé pour le développement de Linux et la première version était publiée en 2005 [SWI 10] [CHA 10]. Le Git n'était pas au départ un logiciel de gestion de versions. Linus Torvalds expliquait que, « par bien des aspects, vous pouvez considérer Git comme un système de fichiers » : il permet un adressage associatif, et possède la notion de versionnage, mais surtout, a été conçu pour résoudre le problème du point de vue d'un spécialiste des systèmes de fichiers. Il n'y avait donc aucun intérêt à créer un système de gestion de versions traditionnel. Il a aujourd'hui évolué pour intégrer toutes les fonctionnalités d'un gestionnaire de versions. Le système Git est considéré comme performant, au point que certains logiciels de gestion de versions (Darcs, Arch), qui n'utilisent

pas de base de données, se sont montrés intéressés par le système de stockage des fichiers de Git pour leur propre fonctionnement. Ils continuent toutefois de proposer des fonctionnalités plus évoluées [BOR 05].

Le Git est un ensemble de programmes censé gérer les objets de bas niveau, compacter les objets en paquets, envoyer et recevoir les objets selon plusieurs manières, créer et gérer des patchs, interroger les données par les opérations diff, et log; chercher les bugs, et importer les projets des autres gestionnaires de versions. Pour assurer ces fonctionnalités, le système stocke l'historique des fichiers sous forme de quatre types d'objets : les Blobs « *Binary Large Object* », les Arbres « *Trees* », les Commits, et les Tags (figure 14).

- Le blob est une donnée binaire non structurée utilisée pour stocker un fichier, il représente le contenu d'un fichier. Chaque révision du fichier à son propre blob et il n y'a pas de relation entre le nom ou l'emplacement du fichier et le blob, mais si un fichier est renommé il n'y aura pas de changement dans le blob.

- Les arbres sont des listes d'objets de type blob et des informations associées à chaque blob, tel que le nom du fichier et les permissions (lecture, écriture et exécution). Les objets décrivent l'arborescence des fichiers à un instant donné.

- Le commit est le résultat de l'opération du même nom et qui donne accès à l'historique d'une arborescence du code source. Il contient un message log, un objet arbre, et pointe vers un ou plusieurs commit parents.

- Le tag est une manière de représenter un commit spécifique. Il pointe vers un des objets précédents (les Blobs, les Arbres, et les Commits). Le tag porte un nom et une signature.

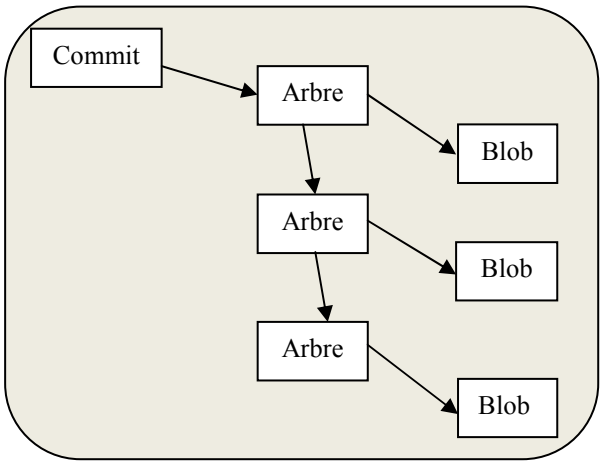


Figure 14. Le Commit, l'Arbre, et le Blob dans le Git

Le système Git permet aux développeurs de gérer leurs codes source. Il fournit un dépôt centralisé comme celle du CVS, et du SVN tout en conservant les avantages de la conservation du dépôt local. Chaque développeur dispose d'un dépôt devant être synchronisé avec les autres. De même, le Git indexe les fichiers d'après leur somme de contrôle (appelée cheksum ou empreinte, la somme de contrôle est un nombre calculée avec la fonction SHA-1 : c'est un nombre qu'on ajoute à un message à transmettre pour permettre au récepteur de vérifier que le message reçu est bien celui qui a été envoyé). Quand un fichier n'est pas

modifié, la somme de contrôle ne change pas et le fichier n'est stocké qu'une seule fois. Par contre, si le fichier est modifié, les deux versions sont stockées sur le disque.

Le Git dispose d'une base des objets qui peut contenir n'importe quel type d'objets, et utilise des indexes (les sommes de contrôle) pour établir un lien entre les objets de la base et l'arborescence des fichiers. Chaque objet est identifié par une somme de contrôle SHA-1 de son contenu. Le Git calcule la somme de contrôle et utilise cette valeur pour déterminer le nom de fichier de l'objet. L'objet est placé dans un répertoire dont le nom correspond aux deux premières lettres de la somme de contrôle, et le reste de la somme de contrôle constitue alors le nom du fichier de cet objet.

Le Git enregistre chaque révision dans un fichier en tant qu'un objet blob unique et les relations entre les objets blobs sont déterminées en examinant les objets commits.

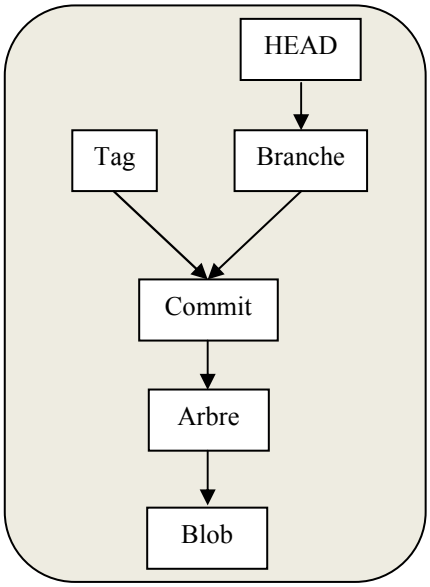


Figure 15. La Gestion de Versions Selon le Système Git

Le système Git propose une façon de développement très différente de celle des systèmes CVS et Subversion. La création de la branche doit être triviale, et les commits doivent être sauvegardés afin de pouvoir y revenir plus tard, par contre il n'y a pas besoin de sauvegarder des patchs à la main; et un seul checkout suffit pour tous.

Un commit est identifié lui aussi comme tous les objets par un hash (SHA-1) pour cela les numéros de révisions à la SVN auraient peu de sens dans le système décentralisé. Le hash dépend du contenu, du message et du prédécesseur et un même commit dans deux branches aura nécessairement deux hashs différents.

Les branches peuvent être fusionnées afin de les maintenir à jour comme ils peuvent ne pas l'être. En cas de fusion, le système combine les historiques de ces branches en un seul historique, le commit de la fusion contient les informations de l'opération `merge`, et il crée dans l'arbre de version approprié, un nouveau nœud fils de la fusion dans la même source de la branche courante.

En cas de conflits, le commit de fusion n'est pas exécuté et des chevrons seront insérés aux endroits problématiques, et les fichiers seront considérés comme des fichiers à conflits. Afin de résoudre ces conflits, le système imposera aux utilisateurs de les corriger manuellement et d'enlever les chevrons [BOR 05].

La gestion de versions proposée par le système Git s'appuie sur le principe de comparaison de fichiers ligne par ligne « *diff* » et sur l'application des différences dans un fichier pour passer d'une version à un autre « *patch* » [BOR 05]. Le processus de la production d'un patch sous le Git est le suivant :

1. Créer une nouvelle branche de travail correspondante à une tâche effectuée,
2. Une fois la tâche réalisée, il faut valider les modifications et les décrire et,
3. Créer le fichier « *.patch* », qui sera ensuite attaché à la tâche sur un fichier nommé « *tracker* ».

Les avantages du système Git se marquent par son interface console qui est plus évoluée que celle de Subversion (permet de paginer automatiquement lorsqu'on affiche le log des commits), ainsi que ses algorithmes de fusion « *merge* », qu'il utilise quand un fichier est modifié par plusieurs personnes en même temps. Dans le cas rare mais possible, où deux utilisateurs modifient en même temps la même ligne, le Git marque un conflit et invite les développeurs à décider ce qu'il faut garder. Le Git est rapide, lorsque les utilisateurs mettent à jour les données même si les modifications sont nombreuses. Contrairement à SVN, Git ne contrôle pas les fichiers mais leur contenu. Cela permet de faire des opérations qui avaient été impossibles dans les systèmes de gestion de versions précédents.

Le Git est plutôt fait pour être utilisé sous Linux; étant donné que sous Windows le partage est plutôt faible et les fusions sont sensiblement plus lentes sous Windows.

Le Git dispose de très grand nombre de commandes qui ne sont pas forcément très faciles ni très sûres à utiliser.

6. Conclusion

Les principes de la gestion de versions et des modifications sont connus depuis des dizaines d'années et aujourd'hui les systèmes logiciels les implémentent de manière optimale vu leurs impacts sur l'aptitude lors la programmation.

Les systèmes de gestion de versions ont permis d'archiver les anciennes versions, de savoir qui a fait une modification et à quelle date, d'empêcher les modifications parallèles d'un même fichier, et de reconstituer un fichier à partir de différentes versions. Ainsi ils ont permis de contrôler l'accès en écriture aux fichiers source, de gérer les modifications apportées à ces fichiers, et de gérer l'accès concurrent aux fichiers via le modèle « Verrouiller-Modifier-Déverrouiller » et le modèle « Copier-Modifié-Fusionner ».

Les VCS ont accompagné les développeurs depuis de nombreuses années. Les plus anciens étaient très contraignants par leur modèle de verrouillage préalable. Le CVS a libéré les développeurs et a mis à leur disposition un modèle où on édite d'abord et on fusionne (éventuellement) après que ça soit stable, alors que beaucoup d'experts prédisaient que cela mènerait au désordre.

Aujourd'hui, il y a le VCS décentralisé qui paraissait inquiétant car il s'agit d'un modèle nouveau et qui semble moins contrôlé. Mais le développement du noyau Linux, où ce modèle est utilisé depuis plusieurs années, montre que le développement décentralisé est possible. Si les outils sont encore primitifs (par exemple, les mécanismes de publication sur le réseau sont sommaires), et trop nombreux (une sélection va être nécessaire) le paradigme des VCS décentralisés est déjà bien installé et représente certainement l'avenir.

Le choix des solutions logicielles spécialisées dans le domaine est très large, mais le nombre croissant des projets de logiciels libres qui y sont présents témoigne de l'intérêt des développeurs pour la gestion de versions et ramène le principe du travail en groupe. Bien que ce domaine ne soit pas nouveau, les outils actuels, en particulier CVS et les outils les plus récents contribuent à faire de la gestion de versions une pratique courante et incontournable. Afin qu'elle soit efficace, les développeurs et les responsables des projets doivent planifier et choisir les méthodes et les outils les mieux adaptés à leurs besoins.

1. Introduction

Actuellement, le web n'est plus simplement un énorme entrepôt de textes et d'images, son évolution a fait qu'il est aussi un fournisseur de service. La notion de «Web Service» désigne essentiellement une application mise à disposition sur Internet par un service fournisseur, et accessible par les clients à travers des protocoles Internet standards. Les web services font partie de ce qu'on appelle les SOA (Service Oriented Architecture), c'est-à-dire qu'il s'agit d'une technologie qui privilège la logique métier et qui n'utilise la technologie que pour mettre en œuvre l'architecture.

Les web services ont été créés et normalisés en janvier 2002 par le groupe W3C (Word Wide Web Consortium) [W3C 04]. Cette technologie est initiée par IBM et Microsoft et acceptée maintenant par l'ensemble des acteurs de l'industrie informatique sans exception. C'est surtout ce point qui fait des web services une technologie révolutionnaire et qui les rend aussi populaires.

Grâce aux web services, les applications peuvent être vues comme un ensemble de services métiers, structurés et correctement décrits, dialoguant selon un standard international plutôt qu'un ensemble d'objets et de méthodes entremêlés.

Les web services facilitent non seulement les échanges entre les applications de l'entreprise mais permettent aussi une ouverture vers les autres entreprises. Cependant, ils ne disposent d'aucune procédure de gestion de versions des documents utilisés dans ces standards ou échangés entre les web services.

2. Définition des Web services

Le paradigme des web services repose sur une architecture par composants qui utilise des protocoles Internet comme infrastructure pour gérer la communication entre les composants. Il offre un modèle à base de composants (les web services) en ligne encapsulant une application logique derrière une interface interactive uniforme et standardisée [SEG 04].

Selon [KAD 03], « Les web services sont des applications auto-descriptives, modulaires et faiblement couplées qui fournissent un modèle de programmation et de déploiement d'applications, basé sur des normes, et s'exécutant au travers de l'infrastructure web ».

Ainsi, un web service est une application conçue pour assurer une interopérabilité entre machines au travers d'un réseau. C'est une interface qui décrit un ensemble d'opérations accessibles via un réseau par des messages XML standards.

En d'autres termes, les web services sont des compléments aux programmes et applications existantes, développées dans différents langages de programmation, et servent de pont pour que ces programmes communiquent entre eux via l'emploi de protocoles standards. Ainsi, ils permettent d'interfacer des systèmes d'information hétérogènes, pour fournir [BEN 09] :

- Un faible couplage avec les technologies employées en interne,
- Une grande flexibilité de mise à jour des systèmes employés de part et d'autre,
- L'emploi de protocoles réseau simples, répandus et bénéficiant d'implémentations dans toutes les technologies majeures.

3. L'Architecture et le Fonctionnement des Web services

L'architecture des web services comme l'architecture du web se base sur des instances d'architecture orientées services (SOA : Service Oriented Architecture). Elle propose une perspective globale sur le développement, la gestion et le fonctionnent des web services [DOU 03].

Les web services s'articulent autour du protocole d'échange de données SOAP (Simple Object Access Protocol) basé sur XML. Ce protocole se situe dans une couche supérieure des protocoles de niveaux applicatifs de l'internet comme HTTP (Hypertext Transfer Protocol), FTP (File Transport Protocol), SMTP (Simple Mail Transport Protocol), etc. Ces derniers encapsulent donc les messages XML issus du protocole SOAP dans leurs propres messages.

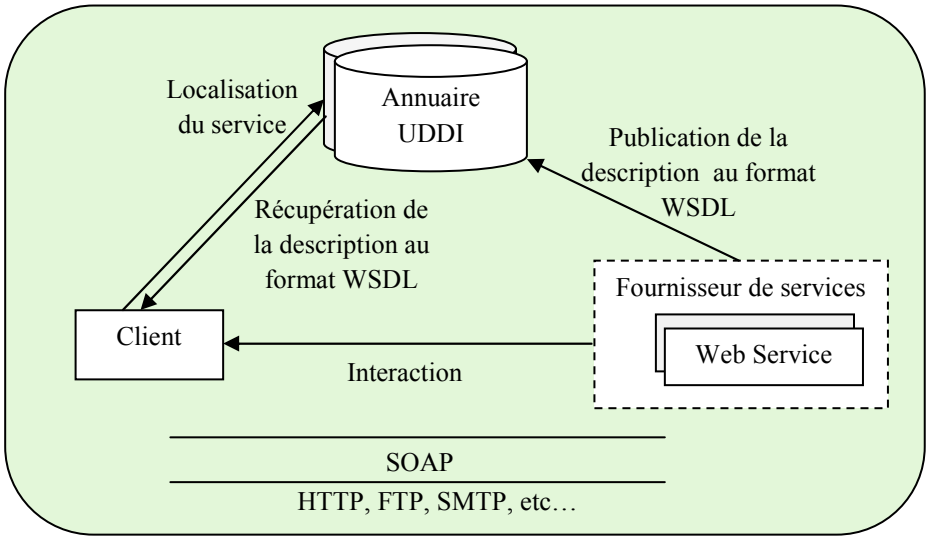


Figure 16. Modèle fonctionnel de l'architecture de publication et d'invocation d'un Web Service

Comme il est possible de le voir sur le modèle fonctionnel de la figure 16, le fournisseur de services est un serveur exécutant des applications ou composants assimilables à des web services. Leur interface est alors décrite en WSDL (Web Services Description Language). Pour se faire, le fournisseur de services publie la description WSDL des services qu'il fournit sur un ou plusieurs annuaires de services UDDI (Universal Description, Discovery and Integration). Cet annuaire peut être local à une application, à un réseau d'entreprise ou à l'échelle d'Internet.

Ainsi lorsque le demandeur de services, c'est-à-dire le client, veut savoir quels sont les serveurs fournissant un service correspondant à une certaine description, il fait appel à un annuaire UDDI dont il a connaissance. Ce dernier lui renvoie alors le ou les fichiers WSDL des web services correspondant à la demande. Le client fait alors son choix, s'adresse au fournisseur et invoque le web service, dont il n'avait pas nécessairement connaissance auparavant [RAM 06].

4. Les Protocoles et Langages Utilisés dans les Web Services

La description fonctionnelle précédente montre l'utilisation de nombreux langages et protocoles durant le déploiement et l'invocation du web service. Ce sont ces principaux langages et protocoles qui vont être abordés dans les sections suivantes.

4.1 Le Protocole SOAP

La communication par message constitue un point crucial dans toute architecture SOA. Le standard actuel qui assure la messagerie est le protocole SOAP.

Le SOAP est un protocole de communication inter web services, basé sur XML et permettant l'échange de données structurées indépendamment des langages de programmation ou des systèmes d'exploitation. Le protocole SOAP est une spécification XML qui définit un protocole léger d'échange de données structurées entre un réseau d'applications dans un environnement totalement distribué et hétérogène. Il est indépendant du contenu du message, et laisse la responsabilité de l'interprétation aux couches de communication supérieures. Il se contente d'offrir la possibilité de structurer des messages destinés à des objectifs particuliers allant d'un simple échange de données jusqu'à l'appel de procédure à distance. Il peut être employé dans tous les styles de communication : synchrones ou asynchrones, point à point ou multi-point.

Les données dans SOAP, sont encodées en XML et échangées par des appels de procédures à distance RPC (Remote Procedure Call) en utilisant les protocoles de communication. Concernant les messages SOAP, ils ne sont pas perturbés par les pare-feu et sont indépendants de la plateforme et du langage [ZUI 03].

Le protocole applicatif le plus utilisé pour transmettre les messages SOAP est HTTP, mais il est également possible d'utiliser les protocoles SMTP ou FTP: la norme n'impose pas de choix. Le choix de l'utilisation de protocoles applicatifs, comme par exemple HTTP, est lié aux problèmes d'interconnexion connus des réseaux.

4.1.1 La Structure d'un message SOAP

Un message SOAP est composé de deux parties indépendantes (figure 17) [RAM 06]:

- ***L'en-tête du protocole de transport :*** Cette partie dépend du protocole de transport utilisé. Par exemple, si le protocole HTTP est utilisé, cette en-tête contient :
  - La version de HTTP utilisée,
  - La date de génération de la page qui est ici le message SOAP lui même,
  - Le type d'encodage du contenu qui est généralement de type XML ou texte.

- ***Les messages SOAP (Enveloppe)*** : Le principal élément d'un message SOAP est l'enveloppe symbolisée par la balise enveloppe, c'est un élément obligatoire. L'enveloppe permet de spécifier la version de SOAP utilisée en utilisant un espace de nom `http://www.w3.org/2001/06/soap-envelope`. Elle permet aussi de spécifier les règles d'encodage (sérialisation et désérialisation) mises en œuvre dans le message `encoding`, (voire l'exemple de la section 4.1.2). Cette enveloppe est subdivisée en deux sous-parties, la partie en-tête et la partie corps du message.

- ***L'en-tête du message SOAP (SOAP Header)*** : Cette partie est optionnelle et extensible. Elle permet de spécifier certaines directives pour le traitement du message. Les balises XML permettant de symboliser cette partie sont `<env:Header>` et `</env:Header>`. Ces balises peuvent être complétées par des attributs permettant de définir le domaine de noms du web service.

- ***Le corps du message SOAP (SOAP Body)***: contient les données spécifiques à l'application. Les balises XML symbolisant cette partie sont `<env:Body>` et`</env:Body>`.

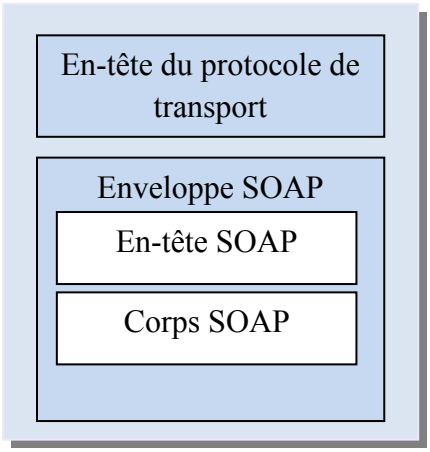


Figure 17. Structure d'un message SOAP

4.1.2 Exemple de message SOAP

```
<env:Envelope xmlns env=:"http://www.w3.org/2001/06/soap-envelope">
  <env:Body>
    <m:ValidatePostcode
     env:encodingStyle=:http://www.w3.org/2001/06/soap- encoding
     xmlns:m="http://www.somesite.com/Postcode">
      <Postcode>XXXXX</Postcode>
      <Country>ALGERIA</Country>
   </m:ValidatePostcode>
  </env:Body>
</env:Envelope>
```

`ValidatePostcode` est le nom du web service invoqué. Le web service contient deux paramètres, `Postcode` et `Country`.

Une réponse à cette requête pourrait ressembler à :

```
<env:Envelope xmlns env=:"http://www.w3.org/2001/06/soap-envelope">
  <env:Body>
    <m:ValidatePostcode
     env:encodingStyle=:http://www.w3.org/2001/06/soap- encoding
     xmlns:m="http://www.somesite.com/Postcode">
      <Valid>Yes</Valid>
    </m:ValidatePostcode>
  </env:Body>
</env:Envelope>
```

L'élément `ValidatePostcode` dans la requête a eu comme réponse l'élément `ValidatePostcodeResponse` qui contient un seul élément `Valid` indiquant si le code postal est Ok ou Non. L'en-tête contient diverses informations, par exemple, les autorisations et les transactions. Le corps transporte les informations sur l'appel et la réponse ainsi que sur les erreurs et les attachements éventuels.

4.2 Le Langage WSDL

Le langage de description WSDL a été créé dans le but de fournir une description unifiée des web services sous forme XML. Il se présente comme un standard actuel dans ce domaine [RAM 06]. Il décrit de manière abstraite et indépendante du langage de programmation, l'ensemble des fonctionnalités offertes par un service. Il permet de connaître les protocoles, les serveurs, les ports, le format des messages, les entrées, les sorties, les exceptions possibles et les opérations réalisées par un web service. Les définitions de services WSDL fournissent de la documentation pour les systèmes répartis et servent de procédure pour l'automatisation des détails impliqués dans la transmission des applications. Le WSDL décrit via le langage XML :

- le contrat de service : l'ensemble des opérations disponibles, ainsi que la structure des messages XML échangés (exprimée par le schéma XML),
- Comment transporter les messages XML sur le protocole SOAP et,
- la localisation du service web.

Le langage de description WSDL se comporte donc comme un langage permettant de décrire l'interface visible (ou publiée) du web service. Il décrit à l'aide du langage de balises XML les différents éléments du service [MSD 00] :

- Les messages intervenant lors les échanges et leurs types associés pour gérer l'interopérabilité entre les différents intervenants de l'échange. Un message représente une description abstraite des données transmises.
- Les opérations composées d'un message ou plus, elles contiennent une description abstraite des actions supportées par le service.
- Les liaisons et les ports de communication permettant de lier les opérations à un protocole de transport sous-jacent. La liaison définit les détails relatifs au protocole et au format de message pour les opérations dans l'attribut appelé binding, et le port spécifie une adresse pour la liaison.
- La description du service lui-même.
- Le type fournit les définitions de types de données utilisées pour décrire les messages échangés.
- Le port type, est un ensemble abstrait d'opérations.

Les messages sont composés de plusieurs parties. Ces parties correspondent, par exemple dans le paradigme objet, aux différents champs d'une structure. Ils sont décrits en XML et un type leur est associé. Les types de base XML peuvent être utilisés (entier, chaîne de caractère, etc.), mais également des types complexes définis dans un fichier WSDL (fichier identique à celui de la description du service ou externe). La partie opération associe les messages aux opérations; une opération peut être vue comme une méthode (les différents messages interprètent les différents paramètres de cette méthode). Un mot clé permet de distinguer le mode et de spécifier le format abstrait de message pour la requête et la réponse :

- *input* : mode entrée ou paramètre de données (la requête).
- *output* : mode sortie ou paramètre de résultat (la réponse).

Exemple sur la structure de WSDL [MSD 00]:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
 <import namespace="uri" location="uri"/>*
 <wsdl:documentation…/> ?
 <wsdl:types> ?
   <wsdl:documentation…/>?
   <xsd:schema…/>*
   <-- élément d'extension --> *
 </wsdl:types>
 <wsdl:message name="nmtoken> *
   <wsdl:documentation…/>?
   <part name="nmtoken" element="qname"? type="qname"?/> *
 </wsdl:message>
 <wsdl:portType name="nmtoken">*
   <wsdl:documentation…/>?
   <wsdl:operation name="nmtoken">*
     <wsdl:documentation…/> ?
     <wsdl:input name="nmtoken"? message="qname">?
       <wsdl:documentation…/> ?
     </wsdl:input>
     <wsdl:output name="nmtoken"? message="qname">?
       <wsdl:documentation…/> ?
     </wsdl:output>
     <wsdl:fault name="nmtoken" message="qname"> *
       <wsdl:documentation…/> ?
     </wsdl:fault>
   </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="nmtoken" type="qname">*
   <wsdl:documentation…/>?
   <-- élément d'extension --> *
   <wsdl:operation name="nmtoken">*
     <wsdl:documentation…/> ?
     <-- élément d'extension --> *
     <wsdl:input name="nmtoken"?> ?
       <wsdl:documentation…/> ?
       <-- élément d'extension -->
     </wsdl:input>
     <wsdl:output name="nmtoken"?> ?
       <wsdl:documentation…/> ?
       <-- élément d'extension --> *
     </wsdl:output>
     <wsdl:fault name="nmtoken"> *
       <wsdl:documentation…/> ?
       <-- élément d'extension --> *
     </wsdl:fault>
   </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="nmtoken"> *
   <wsdl:documentation…/>?
   <wsdl:port name="nmtoken" binding="qname"> *
     <wsdl:documentation…/> ?
     <-- élément d'extension -->
   </wsdl:port>
   <-- élément d'extension -->
 </wsdl:service>
 <-- élément d'extension --> *
</wsdl:definitions>
```

On peut trouver quelques implémentations permettant d'utiliser les informations des fichiers WSDL, dont WSDL4J qui est une implémentation Java utilisée dans un grand nombre d'outils, exemple Axis [VIA 06].

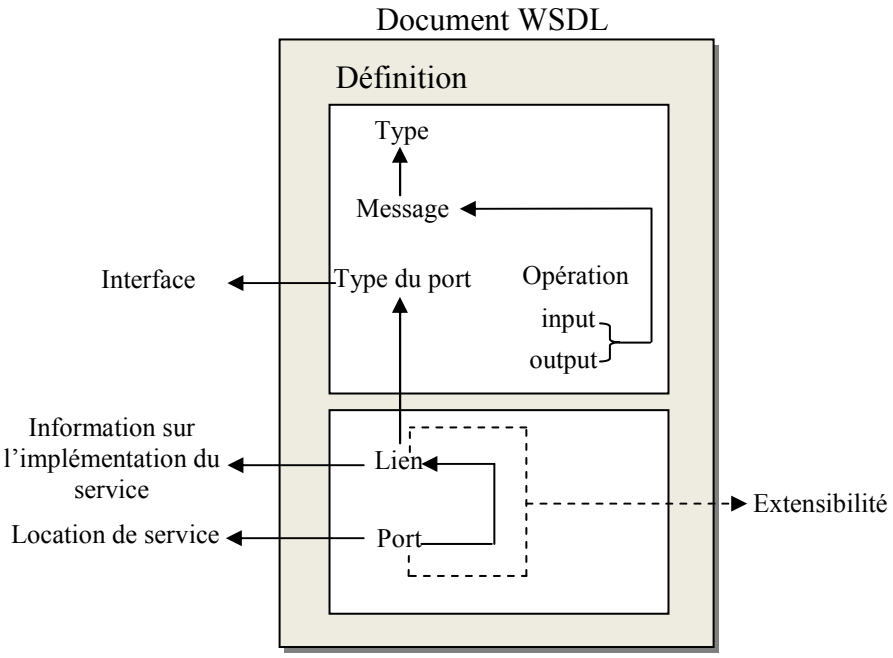


Figure 18. Le document WSDL selon J2EE

Le contrat de service :

Le contrat de service [MAE 03] définit un accord entre le fournisseur et le consommateur du service. Il est composé d'un contrat syntaxique, d'un contrat sémantique, et d'un contrat de niveau de service.

- *Contrat syntaxique* : il propose une représentation technique de l'utilisation du service (son interface). Il présente le nom du traitement, ses paramètres d'entrée et de sortie et les contraintes structurelles (format et contrainte sur les données) qui s'y appliquent.
- *Contrat sémantique* : il fournit une description informelle du traitement en précisant les règles et les contraintes d'utilisation du service ainsi que la valorisation des messages de réponse (résultat d'un calcul, d'une erreur, ou d'une interruption de service), les exceptions, les pré et post conditions techniques (le volume des données échangées), ou métiers (ex. : écriture comptable équilibrée).
- *Contrat de niveau de service* : *(QoS « Quality of Service » & SLA « Service Level Agreement »)* il précise les engagements du service, tels que : le temps de réponse maximum attendu, les plages horaires d'accessibilité, le temps de reprise après interruption, les procédures mises en œuvre en cas de panne, ou les procédures de prise en charge du support, …

4.3 Le Protocole UDDI

L'UDDI est plus qu'un simple protocole. Il fournit un protocole, une API et une plateforme permettant aux utilisateurs du web service, depuis n'importe quel système, de

localiser dynamiquement à travers l'Internet les web services qu'ils désirent utiliser. Ceci passe par le biais des annuaires qui peuvent être maintenus dans plusieurs optiques :

- Par une seule et même personne pour son usage internet,
- Par un groupe d'utilisateurs regroupant des web services répondant à certains critères,
- Par un organisme comme base de données mondiale de web services et,
- UDDI, dans une optique d'interopérabilité, est basé sur les standards tels que HTTP, XML, et SOAP.

L'UDDI est considéré comme un méta-service de localisation de web services.

4.3.1 Les Différents Rôles d'UDDI

L'UDDI fournit trois services de base :

Publish : Ce service gère comment le fournisseur de web service s'enregistre lui-même ainsi que ses services.
Find : Ce service gère comment un client peut localiser le web service désiré. Cela peut passer par des invocations de web services pour une utilisation automatique par un programme ou par une consultation d'annuaire des mots clés.
Bind : Ce service gère également comment un client peut se connecter et utiliser le web service une fois celui-ci localisé.

Les entreprises qui fournissent un service et hébergent un annuaire globale UDDI sont appelées opérateurs UDDI responsables de la synchronisation de l'information des annuaires. Cette synchronisation dénote la réplication.

Un des enjeux d'UDDI est d'éviter le monopole d'une entreprise qui, par un annuaire quelconque, donnerait comme réponse systématiquement ses web services plutôt que ceux des autres. Ce genre de pratique étant fortement limité car les annuaires UDDI se doivent d'avoir un contenu identique aux autres annuaires. Ainsi, il permet de donner des solutions industriellement fiables pour la localisation de web services.

4.3.2 Les Structures de Données UDDI

L'UDDI est un modèle d'information composé de structures de données persistantes appelées entités [ZUI 03]. Ces entités doivent être décrites en XML et stockées dans les différents nœuds UDDI. Les différentes informations sont divisées en trois catégories (figure 19) :

- ***Pages Blanches (White Pages) :*** Elles regroupent les informations sur les noms des entreprises publiant leurs web services, les moyens de les contacter etc,
- ***Pages Jaunes (Yellow Pages) :*** Elles regroupent les informations à propos de la classification des entreprises et,
- ***Pages Vertes (Green Pages) :*** Elles contiennent des informations techniques comme les services offerts par une entreprise, leur spécification, etc.

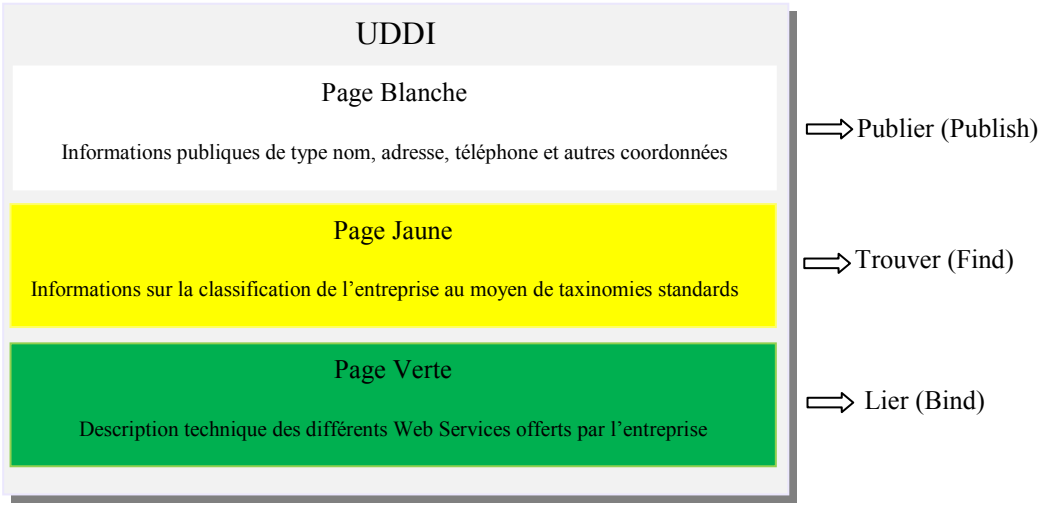


Figure 19. Représentation des différentes pages UDDI

5. L'utilisation des Web Services

La figure 20 illustre le cycle de vie d'utilisation d'un web service.

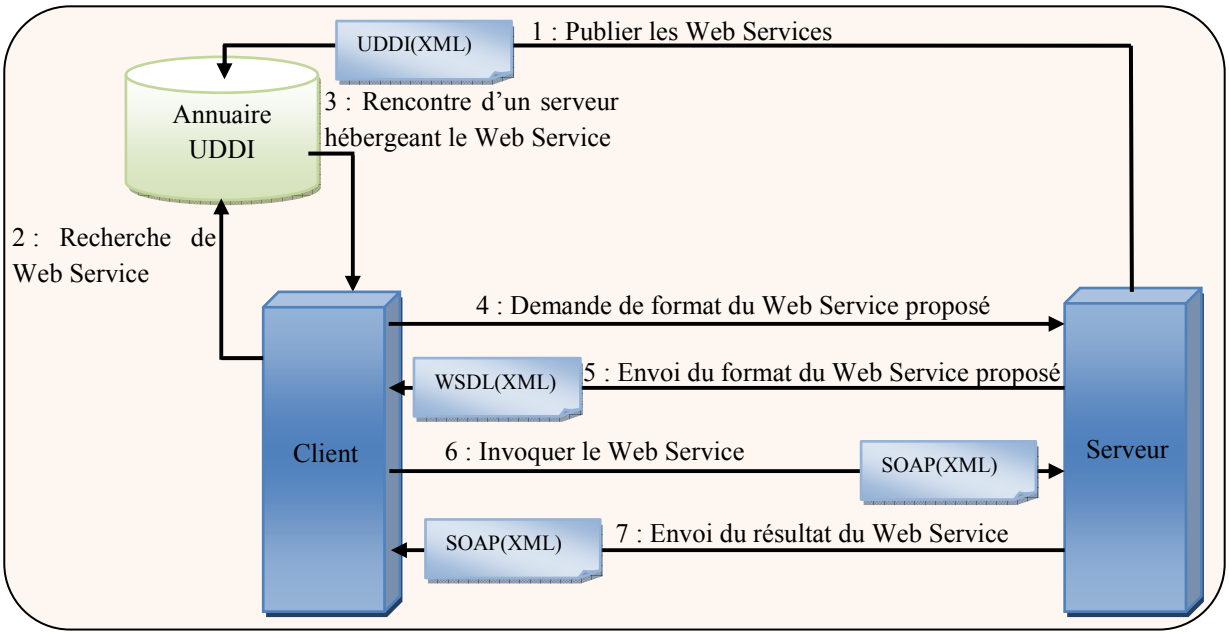


Figure 20. Déploiement, recherche et invocation des Web Services

L'utilisation du web service s'effectue en quatre étapes :

Etape 1 : Déploiement du web service dépendant de la plateforme (Apache)

Au niveau de cette étape, il faut définir une politique dictant comment déployer les web services de manière physique pour baliser les efforts futurs de développement, confiner les applications existantes, et consommer les web services internes et externes.

Etape 2 : Enregistrement du web service

La publication par une entreprise d'un web service requière que celle-ci s'authentifie auprès du site de l'opérateur UDDI. L'entreprise doit s'enregistrer chez l'opérateur si cela n'est pas déjà le cas.

Etape 3 : Découverte du web service

Cette tâche est assurée grâce à un moteur de recherche intégré au site de l'opérateur UDDI choisi. Ce moteur de recherche permettra d'affiner la recherche.

Etape 4 : Invocation du web service par client

Le client qui appelle un web service peut être soit un navigateur web, soit une application cliente. L'appel peut être fait explicitement par l'utilisateur ou peut être automatisé.

6. L'impact du Langage XML dans les Web Services

Le langage XML est devenu un standard incontournable dans les échanges de données sur le web (figure 21). Il est supporté par toute l'industrie informatique, et a pris une place majeure dans les applications web, les systèmes d'information, l'intégration de données et d'applications, le commerce électronique B2B, etc. Le langage XML fournit tout un ensemble de langages pour la définition et la manipulation des données, mais aussi un cadre pour des architectures distribuées à base de web services.

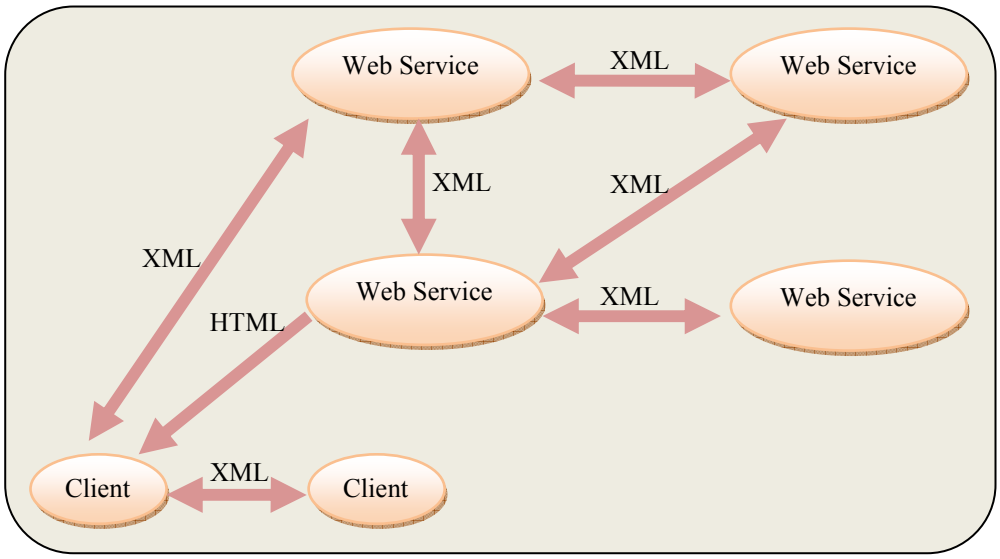


Figure 21. Les échanges entre les Web Services

L'utilisation de XML dans les web services permet la séparation du contenu d'un document, de sa structure et de sa représentation, facilite l'échange d'informations entre les différents partenaires, permet également la composition de services plus complexes à travers la définition des enchaînements entre les services, et favorise l'émergence de standards d'industrie dans la mesure où les structures de données sont réutilisées et réutilisables [SCH 02].

7. Les Avantages des Web Services

L'utilisation des web services apporte de nombreux avantages :

- Les web services fournissent l'interopérabilité entre divers logiciels fonctionnant sur diverses plateformes.
- Les web services utilisent des standards et des protocoles ouverts.
- Les protocoles et les données sont au format texte dans la mesure du possible, facilitant ainsi la compréhension du fonctionnement global des échanges.
- Basés sur le protocole HTTP, les web services peuvent fonctionner au travers de nombreux pare-feux (firewalls) sans nécessiter des changements sur les règles de filtrage. Ces derniers permettent le passage sans problème à travers les différents réseaux des messages générés par l'utilisation du protocole SOAP (XML).
- Utilisables à distance via n'importe quel type de plateforme, ils peuvent servir au développement des applications distribuées et ils sont accessibles depuis n'importe quel type de client.
- Les web services appartiennent à des applications capables de collaborer entre elles de manière transparente pour l'utilisateur.

Les technologies apportées par les web services peuvent être appliquées à toute sorte d'applications auxquelles elles offrent de considérables avantages en comparaison aux anciennes API propriétaires, aux implémentations spécifiques à une plateforme et à quelques autres restrictions classiques que l'on peut rencontrer (multi-plateforme, multi-langage, disponible sur Internet avec une information actualisée disponible en temps réel).

8. Les Web Services et la Gestion de Versions

Les web services sont une nouvelle combinaison de technologies qui permettent la communication entre les systèmes susceptibles et l'utilisation des différentes plateformes sur différents sites [VAU 07]. L'évolution de ces web services est une opération fréquente pour leur permettre de progresser, de s'adapter, et de s'améliorer par rapport aux besoins de ses utilisateurs au fil du temps [WIL 04].

Dans la procédure de son évolution, les fournisseurs et les consommateurs des web services peuvent avoir recours seulement aux versions les plus récentes qui sont disponibles à leurs niveaux. Cependant, les fournisseurs de services peuvent avoir souvent besoin d'accéder aux anciennes versions de leurs services qui sont en ligne pour un certain temps, afin de garder les considérations déjà prises avec les consommateurs de service existants [LEI 08] [ERL 09]. Ainsi, les fournisseurs de services ne disposent actuellement d'aucun moyen qui leur permet de récupérer et d'accéder aux anciennes versions de leurs services car la gestion de versions est absente dans le fonctionnement des web services, et aucun des organismes de normalisation des services d'hébergement web ne supporte la gestion de versions.

Face à ces situations et comme les systèmes informatiques sont toujours en évolution, il est devenu de plus en plus nécessaire pour les applications informatiques de communiquer les unes avec les autres et de partager les données. En conséquence, garder la trace des modifications apportées aux documents est devenu alors essentiel dans de nombreux scénarios différents. Des recherches ont été faites dans ce contexte, et ont abouti à la conception des systèmes de gestion de versions limités par leur dépendance de plateforme.

Comme dans une organisation les utilisateurs peuvent être répartis sur plusieurs différents systèmes d'exploitation, et le contrôle de versions devient accablant.

La gestion de versions est donc très importante dans les architectures logicielles et en particulier dans les services de développement web tels que les web services [VAU 07]. La gestion de versions est aussi un aspect important dans le développement des web services puisque en général, ils évoluent fréquemment au fil du temps [MAT 09].

Pour cela, plusieurs efforts de normalisation de gestion de versions sur les web services sont en cours, mais aucun d'entre eux n'est inclus actuellement dans le fonctionnement des web services [WIL 04] [KAM 06] [PAR 07] [VAU 07] [W3C 07] [MAT 09] [PAL 10] [BEE 11].

Mais avant de citer les différentes solutions proposées par ces chercheurs pour la gestion de versions des web services, nous allons commencer par citer les concepts et terminologie de base pour la gestion de version.

La Compatibilité

Les modifications apportées aux web services sont classées selon leurs impacts sur ces services en deux grands types : la compatibilité en arrière « *backward compatible* » et la compatibilité non en arrière « *non-backward compatible* » [PAR 07] [ERL 09] [W3C 07].

La compatibilité en arrière désigne qu'une version plus récente du service peut interpréter les demandes des consommateurs d'une version ancienne sans rompre le service. Les modifications de la compatibilité en arrière ne rompent pas le contrat de service avec le consommateur; cela signifie que le consommateur peut utiliser la nouvelle version de l'interface du service sans la modifier. Les modifications de la compatibilité en arrière peuvent être une:

- Correction d'une implémentation du service sans modifier le contrat de service, par exemple correction d'un bug,
- Modification de l'interface due à l'ajout ou la suppression des paramètres pour modifier l'implémentation ou pour améliorer le service. Cette opération n'implique pas une modification sémantique de l'ancienne interface, puisque le paramètre modifié est facultatif,
- Modification de l'implémentation du service, pour changer les règles business sans modifier l'interface du service.

La compatibilité non en arrière signifie que la version la plus récente du service ne peut pas interpréter les formats de données des anciennes versions. Les modifications de ce type peuvent être une :

- Suppression d'une opération,
- Modification du nom d'une opération,
- Modification des paramètres (dans le type de données ou dans l'ordre) d'une opération,
- Modification de la structure des données de type complexe.

Il y a aussi ceux qui parlent de la compatibilité en avant « *forward compatibility* ». La compatibilité en avant désigne la capacité d'un système d'accepter des entrées destinées à des versions ultérieures (les nouvelles versions peuvent être déployées sans casser les versions

existantes côté consommateurs). La compatibilité en avant peut être réalisée par l'adoption des standards qui ignorent les éléments non identifiables pouvant apparaître dans le futur. Le but de la compatibilité en avant est de s'assurer que les consommateurs des anciennes versions d'un service peuvent continuer à fonctionner avec la nouvelle version.

Remarque : Une version d'un service est dite compatible si elle est compatible avec les versions antérieures de ce service, surtout si ces versions antérieures devraient être obsolètes.

Les modifications peuvent survenir à différentes étapes du cycle de vie des services, telles que l'identification de service, la conception, le développement, le déploiement et l'exécution selon [PAR 07]. Ces modifications sont classées comme suit :

1. *Les modifications dans les fonctionnalités Business*

Les modifications dans le service Business sont traitées par les modifications des fonctionnalités telles que la modification du comportement de base, la modification sémantique, la modification du niveau d'orchestration. Dans la plupart de ces cas de modifications, il est logique d'en arriver à un nouveau service plutôt que d'une version.

2. *Les modifications dans le contrat de service*

- *Modification de l'interface WSDL* telle que :

  - *Modification dans l'élément type des données*
  - *Modification dans l'élément message*
  - *Modification dans l'élément opération*
  - *Modification dans l'élément Binding*

Les modifications de WSDL vis-à-vis la compatibilité en arrière peuvent être un ajout d'une opération ou un ajout d'une nouvelle structure d'une donnée de type option pour le message d'entrée. Les modifications de WSDL vis-à-vis la compatibilité non en arrière peuvent être une suppression de l'opération, une modification de la cardinalité du message de sortie, un ajout de structures d'une donnée supplémentaire (paramètres ou attributs), un ajout dans le message de sortie ou une modification de la définition des types de données.

- *Modification des politiques*

Les modifications des politiques qui sont compatibles en arrière peuvent être une modification dans la qualité de service (QoS) ou une modification des valeurs de domaine accepté dans le message de sortie. Cependant, la modification compatible non en arrière peut être une modification dans la messagerie de sécurité et de fiabilité de WS-Policy.

3. *Modification dans l'implémentation*

La figure 22 ci-dessous montre la compatibilité en arrière et la compatibilité en avant selon W3C [W3C 07].

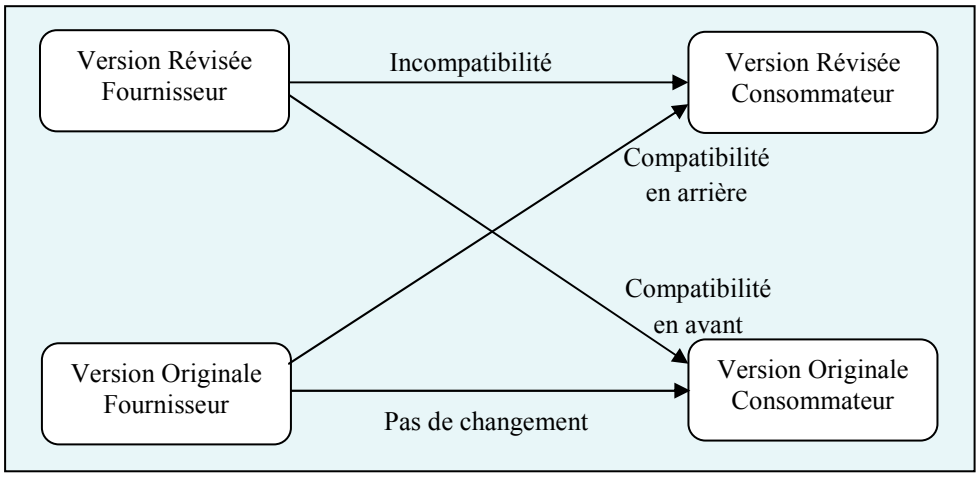


Figure 22. La compatibilité en arrière et la compatibilité en avant

Les stratégies du Versioning

La gestion de version des contrats de services (WSDL, XML Schema, et WS-Policy) selon [ERL 09] est classée en trois stratégies suivant les politiques, les priorités et les besoins : « *The Strict Strategy* », « *The Flexible Strategy* », et « *The Loose Strategy* ».

La stratégie « Strict Strategy »

Dans cette stratégie toute modification compatible ou incompatible établie se traduit par une création d'une nouvelle version du contrat de service. Cette approche ne prend pas en charge la compatibilité en avant ou en arrière.

Cette stratégie offre un contrôle complet sur l'évolution du contrat de service. En revanche, l'ensemble des consommateurs du service existants peuvent ne plus être compatibles avec toutes les nouvelles versions du contrat. Par conséquent, ils ne pourront plus continuer à communiquer avec le web service tandis que l'ancienne et la nouvelle version du contrat est disponible. Les consommateurs doivent se mettre à jour pour être en conforme avec le nouveau contrat.

La stratégie « Flexible Strategy »

La stratégie flexible désigne que toute modification incompatible se traduisant par une nouvelle version du contrat de service et que le contrat est créé pour supporter la compatibilité en arrière mais pas la compatibilité en avant. Cette stratégie permet aux modifications compatibles de se produire sans forcer la création d'une nouvelle version du contrat.

La stratégie « Loose Strategy »

Dans cette stratégie, toute modification incompatible produit une nouvelle version du contrat de service, et le contrat est conçu pour supporter la compatibilité en arrière et la compatibilité en avant.

Remarque : Un nouveau service est créé quand il propose une fonctionnalité différente et distincte par rapport à la fonctionnalité existante ou bien quand sa sémantique est différente.

Remarque : Les systèmes de gestion de versions existants n'ont pas pu être utilisés dans les web services parce qu'ils dépendent de leurs plateformes.

Pour résoudre le problème de la gestion de versions des web services, le travail de [LEI 08] vise à identifier les versions et les relier les unes aux autres. Ce travail se base sur le principe d'un graphe de version où les versions sont représentées par les sommets et les arêtes sont les transitions à une nouvelle version. Cela permet de voir toutes les versions d'un web service en perspective et de les relier les unes aux autres. Les auteurs de ce travail font une classification du WSDL (une classification des différents types de modification de service) pour la gestion de version des web services afin de faciliter aux consommateurs la mise à niveau des services vers une version plus récente. Ils gèrent l'intégration de nouvelles fonctionnalités (l'évolution des web services en terme fonctionnement) dans les web services pour permettre aux consommateurs d'accéder aux différentes versions d'un service par rapport à la base UDDI.

Dans le travail de [MAT 09], les auteurs proposent des extensions pour l'interface WSDL et pour la base UDDI afin de soutenir la gestion de versions des interfaces des web services au moment du développement pour que l'intégration de ces extensions dans les environnements de développement soit facile. Ainsi ces auteurs visent à améliorer le taux de réutilisation des services tout en concevant une solution compatible avec les consommateurs de service (qui ne sont pas au courant). Ils proposent d'ajouter l'élément <wsdletx:versions>, et l'élément <wsdlx:version> dans la description WSDL. L'élément <wsdlx:versions> décrit les versions et spécifie leurs dépendances, et l'élément <wsdlx:version> déclare l'historique complet de la version. Les dépendances entre les versions sont utiles pour assurer la traçabilité des modifications entre les versions. Selon [MAT 09], ces extensions auront été suffisantes dans des environnements où aucun annuaire de service n'est nécessaire. Dans les environnements complexes, les annuaires de services UDDI ne supportent pas les versions. Pour cela, dans [MAT 09] ils proposent des extensions UDDI pour permettre l'enregistrement des informations de version d'un service et pour la notification des consommateurs de services sur les modifications de version.

Dans l'article [BEE 11], les auteurs proposent une approche légère que celle proposée dans [MAT 09] pour enrichir WSDL avec les informations de version. Cela permet aux fournisseurs de service d'informer les consommateurs des modifications établies sur le service et pour que les consommateurs passent rapidement à la nouvelle version. L'article présente un ensemble de minimes extensions destinées à être insérées dans les documents WSDL existants, en ajoutant une information sur le versioning. La principale différence par rapport à la solution proposée dans [MAT 09], est que cette approche vise à être facile à utiliser l'extension sans aucun outillage spécifique. L'approche de [MAT 09] rassemble toutes les descriptions WSDL et les regroupe dans un seul élément, alors que l'approche de [BEE 11] vise à insérer les informations de version à la fin de l'arbre XML tout en maintenant la structure originale des fichiers. Les auteurs de l'article [BEE 11] proposent d'ajouter une balise de version comme un enfant à n'importe quelle balise d'interface dans le fichier WSDL. Par exemple, la balise nextVersion est utilisée pour indiquer qu'une nouvelle version du service a été introduite et qu'elle succède à la version actuelle. La balise nextVersion possède deux attributs obligatoires name et location. L'attribut name est utilisé pour identifier la nouvelle version et l'attribut location est utilisé pour localiser la nouvelle version.

Dans un autre travail [KAM 06], les auteurs proposent une technique de conception qui facilite la gestion de l'évolution des services web appelée chaîne des adaptateurs pour la mise en œuvre des versions précédentes avec un petit service qui appelle la prochaine version du web service. Lors de l'introduction d'une nouvelle version, l'adaptateur doit remplacer la

version précédente par la nouvelle. Cette technique permet aux consommateurs d'accéder aux différentes versions d'un service et permet l'évolution de l'interface et la mise en œuvre d'un service toute en restant compatible avec les versions précédentes que disposent les consommateurs. Pour expliquer le fonctionnement de cette technique les auteurs donnent l'exemple suivant : Supposons qu'une première version d'un service vient d'être achevée. Afin de permettre l'évolution du service à travers la chaîne d'adaptateurs, le développeur doit:

1. Dupliquer l'interface du web service dans un espace de noms différent. La copie qui résulte aura alors les mêmes membres que la structure des données de l'originale, tout en n'ayant aucune relation formelle avec la version originale. Cette interface est dénotée v1.

2. Créer une implémentation de l'interface v1 qui suit tous les appels à l'extrémité de l'interface d'origine, en suite traduire l'espace de noms de toutes les structures de données qui sont nécessaires.

3. Publier et afficher le critère de l'interface v1 comme la première version stable du web service.

Une fois la version v1 déployée et en cours d'utilisation, le développement du service passe à une nouvelle version v2. Chaque fois que le web service est modifié, une modification de la compensation est réalisée dans l'adaptateur pour maintenir le contrat de l'interface v1. Par exemple:

- Si l'interface en cours est modifiée par l'ajout d'un paramètre d'une opération qui existe, l'adaptateur doit être modifié pour fournir une valeur par défaut de ce paramètre lors de la transmission de l'appel.
- Si la définition d'une structure de données est modifiée, l'adaptateur doit traduire la structure de données de l'ancienne vers la nouvelle version (pour les paramètres) ou de la nouvelle à l'ancienne version (pour les paramètres et les valeurs de retour).
- Si une opération est retirée de l'interface, elle doit être remise en place dans l'adaptateur parmi les autres opérations disponibles dans l'interface actuelle.
- Si le contrat d'une opération est modifié, l'adaptateur doit soit compenser la différence ou ré-implémenter l'opération conformément à son contrat v1 comme si elle avait été supprimée.

L'adaptateur n'a pas besoin d'être modifié lorsqu'une nouvelle opération est ajoutée à l'interface, ni lorsque de nouveaux membres facultatifs sont ajoutés à une structure de données.

Selon [KAM 06], la compatibilité en arrière est préservée qu'en théorie, en publiant uniquement les versions de l'interface du service qui ne sont pas utilisées (dont chacun a sa propre adresse du point de terminaison « *endpoint* »). Et dans la pratique, c'est au développeur d'assurer que les adaptateurs compensent correctement les modifications qui ne sont pas de type compatibilité en arrière. Les incompatibilités sémantiques seront plus difficiles à trouver et les chances de réussite peuvent être augmentées lorsque l'adaptateur est développé en même temps que le code principal. Néanmoins, le risque que des changements au web service auront un impact sur les anciennes versions est nécessaire à l'approche proposée, et peut le rendre inutilisable dans certains contextes. L'évolution des services selon cette conception est sans contraintes. L'interface et la mise en œuvre peuvent être modifiées de manière arbitraire, à condition qu'il existe un moyen de mettre en œuvre

le contrat de l'interface précédente en termes de la nouvelle, ce qui n'est pas une limitation très coûteuse, car les opérations obsolètes peuvent simplement être déplacées dans l'adaptateur.

Dans l'article [WIL 04], un schéma sémantiquement extensible pour l'évolution du web service est présenté. Il est basé sur diverses technologies liées au langage XML. L'idée de base est de construire un framework qui augmente le support quasi inexistant pour la gestion de versions des web services pour le protocole SOAP et le langage WSDL. Le framework proposé couvre non seulement les moyens contrôlés pour traiter une version différente du vocabulaire d'un service, mais c'est aussi pour décrire sémantiquement l'extension, pour que les versions plus anciennes d'un service puissent "comprendre" le vocabulaire de la nouvelle version du service.

Enfin, la gestion de version pour le web service e-learning est proposée dans [PAL 10]. Cette gestion permet de faciliter la recherche, l'extraction, l'importation et l'évaluation. Ce modèle de gestion de versions de services proposé répond à certaines exigences telles que l'accessibilité, l'interopérabilité, l'adaptabilité, la durabilité et la réutilisation dans un environnement d'apprentissage. La compatibilité en arrière est focalisée sur la préservation de contrats de services existants tout en étendant le service par l'ajout de nouvelles fonctionnalités. La compatibilité en avant est beaucoup plus difficile à réaliser puisqu'il faut que le service interagisse avec un certain nombre de caractéristiques inconnues ou inattendues. Le modèle de gestion de versions proposé doit prendre en charge plusieurs scénarios d'évolution : les changements de mise en œuvre, la modification de l'interface et la modification de l'annuaire, le versioning de l'objet d'apprentissage avec des métadonnées dans le système e-learning, la recherche, l'extraction, et la livraison rapide des données aux apprenants.

Ainsi, l'évolution des langages utilisés par les web services tel que XML ou XML schéma conduit à la réalisation de nouvelle version; cette opération est faite en ajoutant, supprimant, ou en modifiant la syntaxe ou la sémantique du langage, comme ce qui a été fait pour le langage XML dans les web services afin de passer de la version 1.0 à la version 1.1 [W3C 06b] et ce qui a été fait pour le XML schéma [W3C 06a]. La gestion de version de ces langages a été réalisée indépendamment du web service.

9. Conclusion

L'importance du web service a été reconnue et largement acceptée par l'industrie et la recherche universitaire. Tous les deux ont proposé des solutions qui évoluent dans des directions différentes. La recherche universitaire a été surtout concernée par le fait d'être expressive dans la description de service, tandis que l'industrie s'est concentrée sur la modularisation de couches de service pour la rentabilité à court terme.

Le web service est une application mise en œuvre autour d'un ensemble de protocoles et de normes utilisés pour échanger les données exprimées en XML entre les applications. Les technologies utilisées dans le web service sont le XML, le SOAP, le WSDL et l'UDDI. Le protocole SOAP définit sous XML un langage d'échange permettant à une application d'envoyer et de recevoir des messages, de type requêtes-réponses, via l'Internet. Et le WSDL décrit l'interface publique d'accès à un web service et indique comment communiquer pour utiliser le service par le biais des messages au format SOAP. Tandis que l'UDDI représente une interface d'un référentiel commun d'entreprises pour la description des web services

accessibles au moyen de SOAP. Il décrit et localise un service ou une entreprise et localise également la façon d'accéder aux web services.

L'approvisionnement de la technologie du web service est en plein potentiel pour le calcul orienté vers le web service, qui exige toujours un effort important de recherche et de développement. Parmi les publications ouvertes, nous pouvons mentionner le problème de création automatique et celui de la validité des spécifications des web services de haut niveau incluant la politique de sécurité, de la gestion de versions, des contraintes temporelles et transactionnelles, et la politique de la qualité de service.

Chacune de ces publications implique beaucoup de technologies complémentaires et un défi principal sera pour l'avenir l'échange et l'intégration d'outils théoriques et pratiques et des solutions produites par des communautés différentes de recherche et de développement.

Malgré son impact sur l'Internet, les web services ne disposent actuellement d'aucun système de gestion de versions qui veille à contrôler l'historique des données échangées entre les web services. La proposition d'un modèle de gestion de versions pour le web service est l'objet du troisième chapitre.

1. Introduction

Les nouveaux systèmes de base de données à objets, temporels, multimédias, distribués, …offrent un large éventail de modèles de données et de services qui permettent de résoudre des problèmes spécifiques. Ils se sont révélés souvent limités pour gérer explicitement la dynamique des bases de données, et en particulier, pour réagir aux changements d'état de ces bases. En effet, dans ces systèmes, toute manipulation de données est effectuée sur la demande explicite d'un utilisateur ou d'une application.

Ces systèmes sont considérés comme des systèmes passifs par opposition, aux systèmes actifs, qui permettent d'effectuer des actions prédéfinies, en réponse à des événements spécifiques, quand certaines conditions sont satisfaites afin de réagir aux changements d'états des bases de données.

Ces systèmes actifs, permettent de créer des systèmes d'information plus performants, plus rapides, et à meilleur coût que dans les systèmes passifs. En plus, un serveur actif peut incorporer sa propre logique de contrôle, opérer à partir de ses utilisateurs d'autres serveurs, gérer ses événements internes.

2. Les Systèmes Actifs

Les systèmes actifs sont caractérisés par une capacité de réagir automatiquement à certaines situations en exécutant des opérations prédéfinies [DAY 89] [COU 98] [COU 96].

Cette capacité de réaction consiste à enrichir un système de bases de données « classique » par des règles actives. Les règles actives décrivent de façon déclarative, les opérations qui doivent être exécutées en réaction à un fait significatif et non à la demande explicite d'une application ou d'un utilisateur.

Les règles actives généralisent la notion de déclencheurs (Triggers) proposés dans les années soixante dix, dans le cadre des bases de données relationnelles [BAI 04]. Ces triggers étaient utilisés pour garantir l'intégrité et la cohérence des bases de données [DAY 95] [COL 96], où l'action consistait généralement à rejeter l'opération de mise à jour en cas de violation de contraintes.

3. Les Règles ECA

La notion de réagir à des situations dans les systèmes actifs est supportée par des règles actives de la forme Evénement-Condition-Action (les règles ECA).

La sémantique d'une règle ECA est :

Lorsqu'un événement E se produit
Si la condition C est satisfaite
Alors exécuter l'action A

- La partie *Evénement* caractérise un fait significatif pour le déclenchement de la règle.
- La partie *condition* examine le contexte d'occurrence de l'événement.

- La partie *Action* spécifie la tâche à réaliser si l'événement a eu et si la condition a été validée.

Cette interprétation est adoptée par la plupart des systèmes actifs. Néanmoins il existe deux autres combinaisons pour regrouper les composantes d'une règle :

- La première consiste à définir une règle par le couple (E, (C, A)) où la condition est codée comme une partie de l'action, elle est adoptée par SYBASE [SYB 99].
- La deuxième combinaison définie par le couple ((E, C), A) considère la condition comme partie de l'événement, elle est adoptée par le système ODE [GEH 91].

Exemple : La règle qui spécifie et vérifie la contrainte d'intégrité «l'âge d'une personne ne peut évoluer que par incrément de valeur 1», est exprimée selon *NAOS* [COU 96] par :

```
Create Rule Modif-age
On after update.personne age with P
Coupling immediate
If current(p)age – old(p)age =! 1
Do {display('MAJ refusée:transaction courante annulée'; abort ;}
```

L'intégration des règles actives dans un SGBD permet d'offrir un mécanisme général et puissant pouvant être utilisé pour remplir diverses tâches. Les domaines d'application qui peuvent adopter les règles actives sont classés en trois grandes catégories selon [PAT 99] :

- Une extension de la base de données, par la gestion de contraintes d'intégrité (ex. il n'y a pas plus de 15 employés dans chaque département), la dérivation de vues, etc…
- Des applications "fermées", c'est-à-dire spécifiant le comportement du système en interne, sans faire référence à l'extérieur
- Des applications "ouvertes", c'est-à-dire répondant à des situations survenant en dehors de la base de données.

Un SGBD qui supporte des règles actives dispose d'un nouveau mode de programmation. Certains traitements qui étaient habituellement noyés dans le code d'une application, peuvent être réécrits sous forme déclarative et modulaire au moyen de règles actives améliorant ainsi la qualité des applications et la productivité du programmeur. Ainsi les règles actives offrent un moyen simple d'interrompre les programmes utilisateurs, de prendre le contrôle de l'application, et d'exécuter de nouveaux programmes spécifiés dans l'action des règles.

4. Structuration en Modèles

Un SGBD actif doit permettre à l'utilisateur de décrire les règles qui seront exécutées selon un modèle d'exécution précis [PAT 99]. Les caractéristiques de la description, la manipulation et le comportement des règles ont été structurés selon trois modèles (figure 23) :

- Modèle de connaissances
- Modèle de détection et de production des événements
- Modèle d'exécution

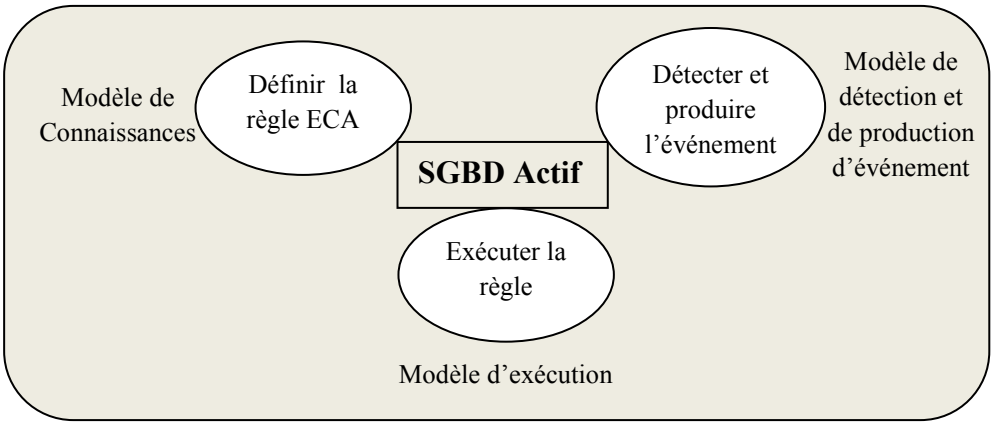


Figure 23. Structuration des systèmes actifs en modèles

4.1 Modèle de Connaissances

Le modèle de connaissances d'un système actif décrit la façon dont sont définies, représentées et manipulées les données et les règles. Il dispose des deux éléments constitutifs suivants : le *modèle de Données* et le *modèle de Règles.*

4.1.1 Modèle de Données

Le modèle de données sert à définir la structure des données et les opérations applicables à ces données. Le modèle dépend du modèle adopté par le SGBD cible du système (Modèle Relationnel étendu, modèle Objet…).

4.1.2 Modèle de Règles

Le modèle de règle permet de définir la structure des règles et les opérations applicables sur ces règles (création, suppression, activation, désactivation). Une règle est composée de trois parties : la partie événement, la partie condition et la partie action.

4.1.2.1 Evénement

La notion d'événement est liée aux notions de types d'événement et des occurrences d'un type d'événement [COL 96].

- ***Type d'événement***

Le type d'événement est une expression qui caractérise une classe de faits significatifs pour le déclenchement d'une règle qui se produit dans un intervalle de validité. Ce dernier détermine l'intervalle de temps où les occurrences du type peuvent se produire. Le type d'événement d'une règle est décrit par son nom et une suite de paramètres formels du type d'événement. Par exemple, le type d'événement d'insertion de données dans une relation « employés » peut être décrit par `insert_emp(emp :type_emp)`. Le nom du type est `insert_emp`, qui a un paramètre formel décrit par son type, `type_emp`, et par l'identificateur `emp`.

- ***Evénement***

L'événement est une occurrence d'un type d'événement auquel on peut toujours associer un point de temps appartenant à l'intervalle de validité donné par le type [BOU 08].

Exemple :

Type d'événement : MAJ du salaire d'un employé
Un événement de ce type : MAJ du salaire de l'employé Mohamed

- ***Contexte d'un Evénement***

A chaque événement est associé un contexte, qui est un ensemble d'informations permettant d'expliciter la production de cet événement : son type et son instant de production.

Le contexte peut être simple comme dans le cas de l'événement « avant l'insertion d'un n-uplet dans une relation » et le contexte contient alors le n-uplet à insérer, et il peut être complexe comme pour les événements liés aux appels de méthode et dans ce cas, les paramètres de l'appel feront partie du contexte de l'événement.

Afin de faire référence aux informations du contexte de l'événement traité, la majorité des langages de règles proposent un mécanisme pour les utiliser dans les parties « condition » et « action » de la règle déclenchée par l'événement.

- ***Classification des Evénements***

Le modèle d'événements spécifie l'ensemble des types d'événements responsables dans le système, ils sont classés selon deux types *Primitifs* ou *Composés* (*Composites*).

- ✓ *Evénements Primitifs*

Les événements primitifs sont produits par des opérations atomiques; ils décrivent les occurrences élémentaires qui sont produites par l'application ou transmises au système par des processus extérieurs. Ces événements peuvent être internes, temporels ou externes.

Evénement Interne: L'événement émane directement du SGBD. Il est lié à la manipulation des données et des transactions (par exemple, dans le modèle objet l'appel de méthode est un événement).

Evénement Temporel: C'est un événement qui fait référence au temps. Il peut être *Absolu*, *Relatif*, ou *Périodique*.

Absolu: associé à une valeur absolue dans le temps. Exemple : le 25/12/2010.
Relatif: fait référence à une valeur de temps relative à un point de repère. Exemple : 3 Mn après login.
Périodique: associé à une valeur de temps périodique. Exemple : Tous les mois.

Evénement Externe: L'événement ne dépend d'aucun phénomène opératoire du SGBD, mais plutôt de phénomène externe au système, Exemple : alarme, contrôleur de température.

- ✓ *Evénements Composés*

Un événement composé se produit lorsque des événements composants se sont produits dans un certain ordre ou en respectant certaines conditions données décrites par la sémantique des opérateurs utilisés [COL 96]. Ils représentent une combinaison d'événements primitifs ou composés grâce aux opérateurs de composition (constructeurs) tels que: la conjonction, la disjonction, la séquence, la négation…

La sémantique des opérateurs de composition est comme suit:

Conjonction ($e=e_1, e_2$): L'événement e a lieu quand les deux événements e_1 et e_2 ont lieu sans considération du temps de production.
Disjonction ($e=e_1| e_2$) : L'événement e a lieu quand e_1 ou e_2 a lieu.
Séquence ($e=e_1; e_2$) : e se produit quand e_2 survient après que e_1 se soit produit.
Négation (e=Not e_1 in I) : l'événement e est signalé si aucune occurrence de e_1 ne survient pendant l'intervalle I.
L'historique (Times (n, e) in I) : l'événement a lieu, toutes les n occurrences de e dans l'intervalle I.
Constructeur *(e=* e' in I) : l'événement e a lieu après la première occurrence de e' dans l'intervalle I.
L'opérateur fa (fa (e_1, e_2, e_3)) : est défini comme la première occurrence de l'événement e_2 par rapport à l'événement e_1, sans qu'il y'ait l'occurrence de e_3 entre e_1 et e_2.
Choose (E= Choose (n, e_i)) : L'événement composé E est détecté si la nième occurrence de l'événement e_i est détecté.

4.1.2.2 Condition

La condition est une formule qui permet de préciser les situations pour lesquelles il faut exécuter l'action de la règle. L'action n'est exécutée que si la condition est satisfaite.

Pour certains systèmes actifs, la partie condition d'une règle est optionnelle. L'absence de la condition signifie que la condition est toujours vraie, ou elle est intégrée dans la partie action ou la partie événement (comme dans ODE [GEH 91] où la condition est insérée dans la spécification de l'événement).

4.1.2.3 Action

L'action correspond en général à un ensemble d'actions qui sont exécutées après que la condition a été évaluée à vrai ou après l'événement déclenchant dans le cas où la condition n'existe pas. Elle est spécifiée au moyen du langage de requêtes ou d'un langage de programmation supporté par le SGBD (par exemple, les commandes SQL dans le relationnel).

Exemples de Règles Actives : Soit une application active gérant les salaires du personnel d'une entreprise. Le schéma de la base comporte [LLI 97]:

Employé (nemp, nom, salaire, statut, grade, ndpt)
Droit-ss (nemp, niveau) niveau de prestation sociale
Departement (ndpt, description, chef, pays)

L'application supervise les modifications de salaire des employés au moyen de 2 règles gestion se déclenchant quand il y'a des modifications de salaire des employés. La règle *«Prestation_salaire »* et la règle «*Moyen_Salaires»* que nous présentons ci-dessous.

Règle1 « Prestation_salaire » : Indique qu'il faut recalculer le niveau de prestation sociale d'un employé en CDI, chaque fois que l'augmentation de son salaire est supérieure à 600. Pour déterminer le contexte de l'événement, il faut déterminer quelles informations la règle a besoin pour être évaluée et exécutée correctement.

Le contexte de la règle « *Prestation_salaire* » : On doit connaître l'identification de l'employé concerné par le changement de salaire mais aussi de la variation de salaire dont il a bénéficié.

Le contexte de l'événement déclenchant est défini par le tuple < nemp,variation>.

Evénement : ps(nemp)
Contexte : nom=contexte; structure=tuple
of (nemp,variation)
Condition: exists (select * from Employé
where nemp=contexte.nemp
and statut=CDI)
and (contexte.variation>600)
Action : calculprest(nemp)

Règle2 « Moyen_Salaires » : Indique qu'il faut prévenir le directeur financier lorsque la moyenne des augmentations faites dans une transaction est ≥ 800.

Le contexte de l'événement déclenchant est défini par le tuple <emp,variation>.

Evénement : mds
Contexte : nom=contexte; structure= table of
<emp,variation>
Condition : AVG(Select variation from contexte
where variation>0) ≥ 800
Action : Envoyer_mail('Chef, ça revient cher!')

*Remarque***:** Les parties événement, condition et action d'une règle ne sont pas cloisonnées, et les données peuvent-être transmises de l'événement vers la condition et vers l'action ou de la condition vers l'action.

4.2 Modèle de Détection et de Production d'Evénements

4.2.1 Introduction

Les événements (primitifs ou composés) déclenchants ne sont produits que dans la mesure où les événements primitifs ont été détectés [COU 96]. Le modèle de détection spécifie quels événements sont détectés et quand ces événements sont détectés. Une fois détectés et produits, ils sont signalés au moteur d'exécution des règles. L'événement fournit la précision de *la Granularité* de la source de production et *le Mode de production* des événements composés.

La Granularité représente une unité dans laquelle l'événement est produit. En général une transaction ou une opération de base de données (commande SQL dans SGBD Relationnel).

Concernant *le Mode de production,* il permet de préciser la sémantique des types d'événements composés en fonction de l'application. Il existe 4 modes de production: récent, chronologique, continu et cumulatif.

Exemple: Soit le type d'événement $E=E_1;E_2$. Considérons l'historique d'événements e_{21} e_{12} e_{13} e_{24} e_{15} e_{26} e_{27} où e_{ij} est un événement de type E_i qui s'est produit à l'instant t_j dans l'unité de production considérée.

Il est possible de produire différents événements composés selon le mode de production considéré.

- Récent: seules les occurrences du type E_i les plus récentes sont utilisées pour produire un événement composé.
  (e_{13}, e_{24}) , (e_{15}, e_{26}) et (e_{15}, e_{27})
- Chronologique: les occurrences du type E_i sont considérées dans leur ordre d'apparition chronologique.
  (e_{12}, e_{24}) , (e_{13}, e_{26}) et (e_{15}, e_{27})
- Continu: toutes les occurrences du type E_i qui marquent le début d'un intervalle d'un type d'événements composés sont potentiellement considérées comme des événements initiateurs d'événements composés.
  (e_{12}, e_{24}) , (e_{13}, e_{24}) , (e_{12}, e_{26}) , (e_{13}, e_{26}) , (e_{15}, e_{26}) , (e_{12}, e_{27}) , (e_{13}, e_{27}) et (e_{15}, e_{27}).
- Cumulatif: quand une occurrence de E est reconnue, le contexte associé inclut (cumule) les paramètres de toutes les occurrences de E_i intervenant dans la définition du type de E. Une occurrence de E_i participe au contexte d'une seule occurrence de E. Une occurrence de type E avec e_{24} qui inclut (e_{12}, e_{13}) dans son contexte et une autre avec e_{26} qui inclut (e_{15}).

4.2.2 Les Techniques de Détection des Evénements Composites

Le SGBD Actif doit être en mesure de détecter ou de signaler les événements liés aux opérations sur la base. Les événements primitifs sont détectés directement, tandis que les événements composites ont donné lieu à de nombreuses recherches. Elles ont abouti à la mise au point de trois approches; l'utilisation des automates finis comme dans ODE [GEH 91], des RdP Colorés dans SAMOS [GAT 94a], ou des graphes d'événements comme dans NAOS [COU 96] et SENTINEL [CHA 94].

Remarque : seuls les événements dont les types sont connus du SGBD actif seront détectés.

- ***Détection par Automates***

La détection des événements composites par automate est une approche proposée par le système actif de base de données ODE, vu que les expressions d'automate d'un événement composé sont semblables aux expressions régulières lorsqu'elles ne sont pas paramétrées. Cette approche fournit un automate fini de la détection des événements composés (figure 24).

La détection des événements se fait de la façon suivante : à chaque fois qu'un événement primitif est signalé, le système vérifie toutes les règles actives pour déterminer si un événement logique a eu lieu afin de passer l'automate à son état suivant. Si un événement déclenche plusieurs règles, alors l'ordre d'exécution dépend de l'implémentation, c'est-à-dire le programmeur doit être prudent.

Les automates doivent être suivis d'une structure de données pour stocker des informations additionnelles sur le temps d'occurrence et le temps de la détection de l'événement composé. Pour chaque définition de règle, la table de transition de l'automate est stockée une fois pour

toute la classe, et l'état de l'automate est stocké avec chaque objet pour qui la règle est activée.

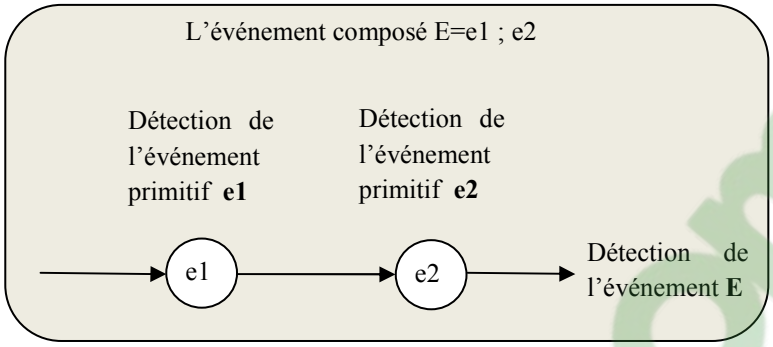


Figure 24. Exemple de détection d'un événement composite (Séquence) par Automate

- ***Détection par les Réseaux de Petri***

La détection des événements par les *RdP* a pour but de modéliser et détecter les événements composés, ce mécanisme est utilisé dans le système SAMOS [GAT 94b] [MOT 95].

- *Les RdP Classiques*

En 1993, Gatziu et son équipe ont utilisé pour la première fois les RdP classiques pour la détection des événements composés [GAT 93]. Selon le contexte SAMOS, le *RdP* est défini comme un tuple (*N*, *Mo*) où *N* représente la structure du *RdP*, et *Mo* est le marquage initial du RdP. *N* est un 5 uplet, *N*= (P, T, A, Ps, Pe) où Pe représente l'ensemble des places en entrée et Ps les places en sortie. Les places en entrée représentent les événements primitifs et les places en sortie, les événements composés.

Chaque fois qu'un utilisateur définit un événement composé, le système crée le *RdP* approprié suivant sa sémantique.

Comme exemple, nous proposons le constructeur séquence qui est modélisé par le RdP ci-dessous (figure 25).

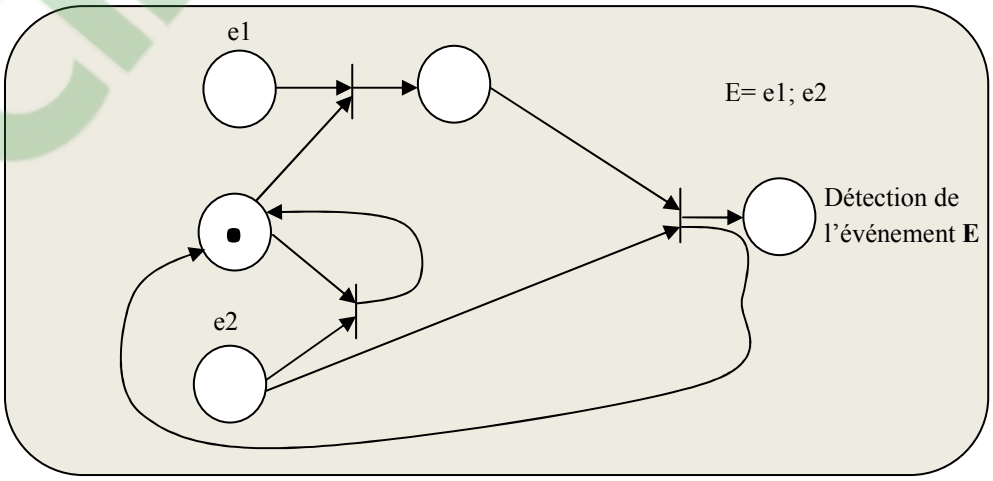


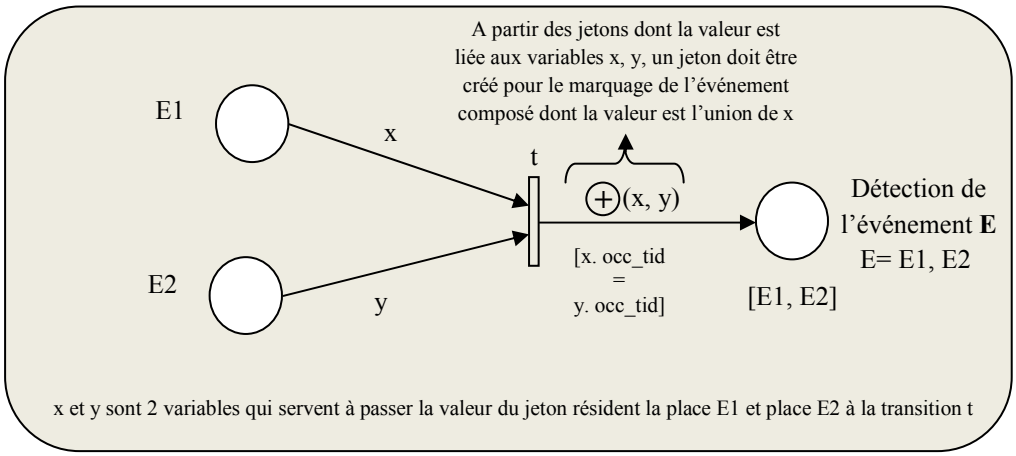
Figure 25. Exemple de détection d'un événement composite (Séquence) par RdP classique

A chaque occurrence d'un événement, la place en entrée modélisant l'événement approprié est marquée, la transition peut-être franchie et les places en sortie seront marquées. Si une place en sortie correspond à un événement composé alors ce dernier est détecté.

- *Les RdP colorés*

Afin de prendre en charge les informations complexes à travers le RdP et prendre en considération les dépendances entre les événements, une nouvelle approche basée sur le mécanisme des RdP colorés (S-PN) [GAT 94b] a été proposée par l'équipe de SAMOS. Les informations (paramètres de l'événement, identificateur de l'objet, l'utilisateur de l'occurrence,..) sont prises en charge par le jeton assigné à une place.

Soit l'événement composé E=E1, E2 same transaction (où E=Update-tr, E1= Update-value1, E2= Update-value2). Le type S-PN pour conjonction avec « same » est modélisé comme suit :



Une instance de S-PN est définie ci-dessus :

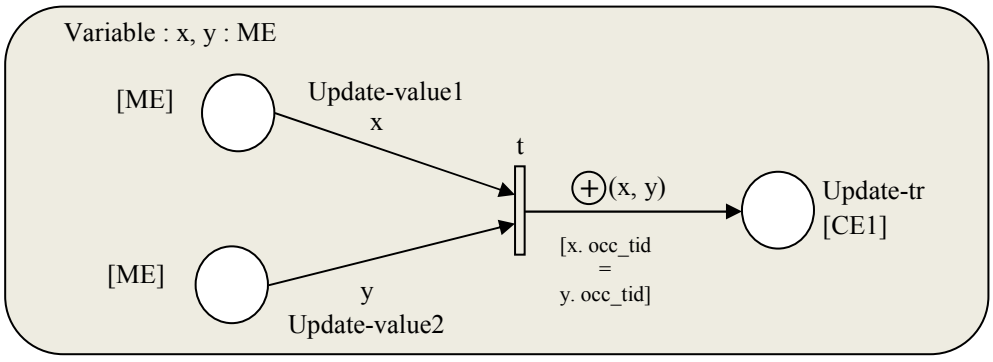


Figure 26. Exemple de détection d'un événement composite (Conjonction) par RdP coloré

Les jetons ME et CE1 modélisant respectivement un événement méthode et un événement composé ont la structure suivante :

ME= Record (event-id, occ-point, occ-tid, user-tid, object-tid, parameter)

CE1= Record (event-id, occ-point, list of Record, comp-event, parameter)

- ***Détection par les Graphes DAG***

La détection des événements primitifs est basée sur le principe d'abonnement de chaque type d'événement (Abonner indique que l'opération doit-être signalée comme événement). Pour chaque abonnement, l'adresse d'une fonction de manipulation est fournie (liée au type d'événement et delta structure). Elle sera appelée quand l'événement correspondant est reconnu. Le détecteur d'événements signale au gestionnaire d'événements, uniquement les événements abonnés, c'est-à-dire dont les types sont représentés dans le graphe d'événement à partir de la détection.

La détection est basée sur le mécanisme des graphes d'événements (figure 27). Le graphe d'événements est un graphe dirigé acyclique DAG « Direct Acyclic Graph ». Les graphes sont construits récursivement selon la sémantique de l'expression des événements. Les événements primitifs sont injectés aux feuilles carrés du graphe puis transmis vers le sommet en suivant les arêtes du graphe et en traversant des nœuds internes. Un nœud correspond à une expression qui représente soit un ou plusieurs type d'événements composites soit tout simplement une sous expression.

Le processus du détecteur est le suivant : il commence à signaler l'événement primitif détecté avec son environnement par le détecteur, il construit une structure de données appelée jeton primitif (identificateur de type d'événement, temps d'occurrence, son environnement), puis, injecte ce jeton dans la feuille (carré) correspondante dans le graphe événement, ensuite il transmet le jeton primitif vers le sommet en suivant les arcs du graphe en traversant les nœuds internes. A la fin il signale l'événement composite à l'exécutif quand le nœud déclenchant, racine du graphe, correspondant est atteint.

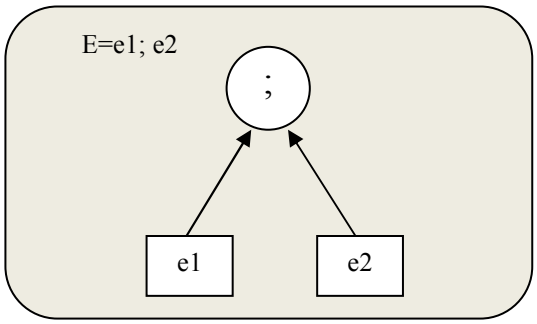


Figure 27. Exemple de détection d'un événement composite (Séquence) par Graphe

4.3 Modèle d'Exécution

Ce modèle spécifie quand et comment sont déclenchées, évaluées et exécutées les règles au cours du déroulement d'une application.

Dans le modèle d'exécution des règles, on parle du mode de couplage, de la granularité de la transaction, de la politique en matière d'effet net, de la politique de cycle, des priorités, de l'ordre d'exécution, et du traitement des erreurs [PAT 99].

- Le mode de couplage indique où et quand s'exécutent les parties « Condition » et « Action » d'une règle ECA relativement à la production de l'événement déclenchant d'une part, et relativement l'une à l'autre d'autre part.
- La granularité de la transaction indique si une règle est déclenchée pour chaque instance d'un événement ou pour un ensemble d'instances.

- La politique en matière d'effet net indique si le système prend en compte l'effet net global des modifications ou bien l'effet net de chaque occurrence d'événement.
- La politique de cycle indique : la réaction du système si l'évaluation de la condition ou l'exécution de l'action signale des événements et par conséquent déclenche d'autres règle.

Les règles qui doivent être déclenchées ainsi que le temps de déclenchement doivent prendre en considération :

- Comment calculer les événements déclenchants ? (mode de production).
- Quand calculer les événements déclenchants ? Doit-on les calculer à chaque fois qu'un nouvel événement primitif est détecté ? (impact sur les contextes associés).
- Que faire si plusieurs événements déclenchants sont produits au même moment ? Doit-on déclencher toutes les règles correspondantes ?

Les règles qui doivent être exécutées ainsi que le temps d'exécution doivent prendre en considération :

- Quand exécuter une règle par rapport à l'événement qui l'a déclenchée ? Doit-on évaluer la condition et exécuter immédiatement ? Doit-on différer ?
- Que faire quand plusieurs règles doivent être exécutées au même moment ? Doit-on les exécuter en parallèle ? En séquence ?
- Que faire quand une règle produit des événements déclenchant d'autres règles ? Doit-on les exécuter toutes ? Dans quel ordre ?

4.3.1 Processus d'Exécution

- **Processus d'Exécution d'une Règle**

Le processus d'exécution d'une règle passe généralement par trois principales phases : le déclenchement, l'évaluation, et l'exécution.

Le déclenchement est le processus par lequel une règle passe d'un état déclenchable à un état déclenchée, suite à l'occurrence d'un ou plusieurs événements dont au moins un est un événement déclenchant. L'évaluation correspond à l'évaluation de la partie condition de la règle. Si la condition est satisfaite, la règle passe dans l'état évalué, sinon elle revient à l'état déclenchable. L'exécution correspond à l'exécution de la partie action de la règle. Après l'exécution , la règle revient à l'état déclenchable (figure 28).

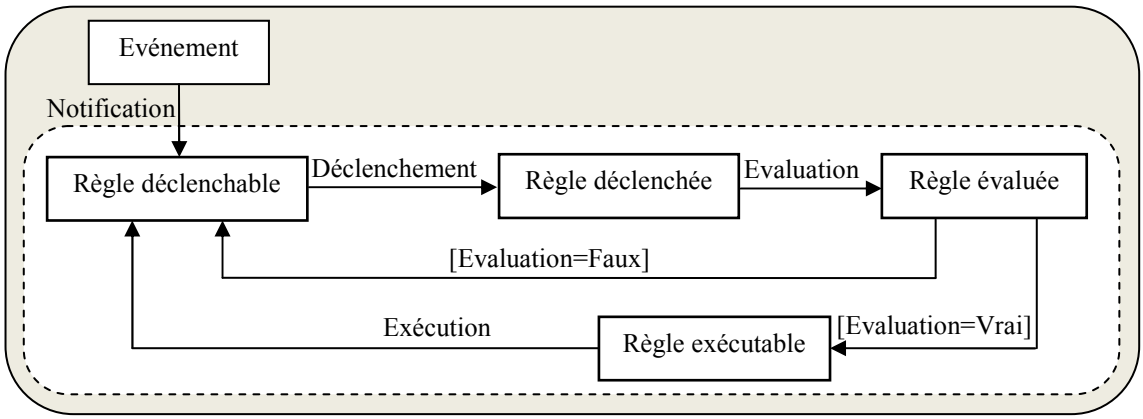


Figure 28. Processus d'exécution d'une règle

- **Processus d'Exécution d'un Ensemble de Règles**

Le processus d'exécution d'un ensemble de règles a pour but de déterminer l'ordre et la manière dont les règles de cet ensemble vont être exécutées.

Dans ce processus, plusieurs règles sont déclenchables, soit parce qu'un événement a déclenché plusieurs règles, soit parce qu'il existait déjà un ensemble de règles déclenchées auquel il faut intégrer de nouvelles règles déclenchées (figure 29). Les règles passent à l'état déclenchées après l'occurrence de l'événement déclenchant, en suite le processus d'ordonnancement détermine parmi ces règles celles qui vont être exécutées afin de sélectionner une règle. La règle sélectionnée passe à l'état évaluable, si cette dernière est évaluée à faux elle retourne à l'état déclenchable, sinon elle passe à l'exécution. Lorsque l'exécution est terminée la règle repasse à l'état déclenchable.

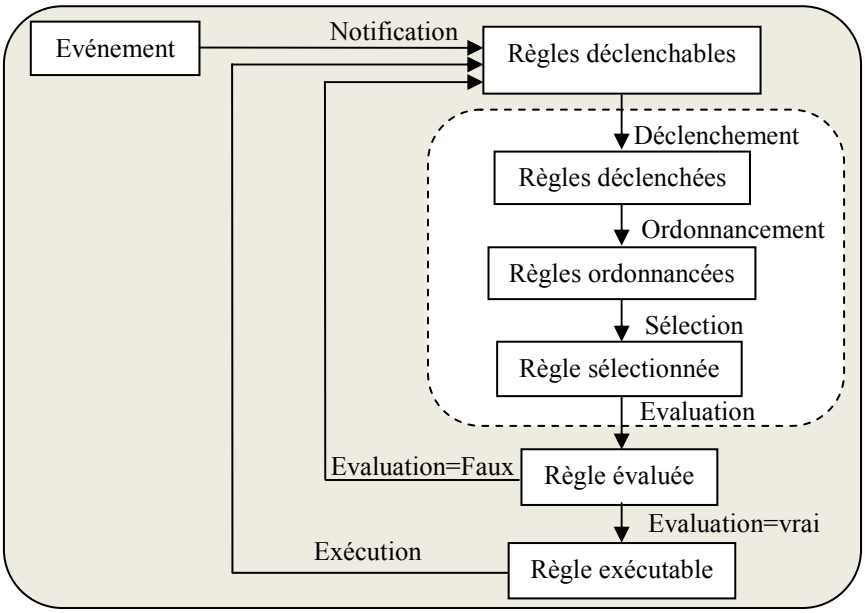


Figure 29. Processus d'exécution d'un ensemble de règles

4.3.2 Caractérisation du Comportement d'une Règle

- ***Granularité du Déclenchement***

Quand plusieurs occurrences d'un type d'événement sont produites dans une même unité de production, il existe deux types de stratégies de prise en compte de ces événements qui influent sur le déclenchement de la règle R, selon une sémantique par instance ou selon une sémantique ensembliste.

La sémantique par instance consiste à traiter les événements instance par instance, et exécuter donc R autant de fois qu'il y'a des événements déclenchants. Alors que la sémantique ensembliste consiste à exécuter R qu'une seule fois pour l'ensemble des événements.

- ***Mode de Consommation des Evénements***

Une règle est susceptible d'être déclenchée et exécutée plusieurs fois dans la même transaction par les occurrences d'un même type d'événement. Deux politiques de traitement de l'événement sont adoptées :

Consommation : L'événement est consommé pour la première exécution de la règle qu'il a déclenchée et ignorée dans les exécutions ultérieures.

Préservation : Après la première prise en compte de l'événement, l'événement est préservé et pris en compte dans les exécutions ultérieures de la règle (possible si on a pour les règles déclenchées, une sémantique ensembliste).

- ***Instant de Consommation des Evénements***

La consommation peut avoir lieu soit à l'évaluation de la règle, soit de manière restrictive, à l'exécution de la règle (l'événement est réellement consommé uniquement si la condition est satisfaite).

- ***Préemption de l'Exécution***

L'exécution d'une règle est susceptible de produire des événements. Soit R' une règle déclenchée par l'exécution de la règle R. L'exécution de R est suspendue et on exécute R' avant de revenir à l'exécution de la règle R; R' a un droit de préemption sur R. Les événements déclenchants de R' sont mémorisés jusqu'à la fin de l'exécution de R où ils seront alors pris en compte pour l'exécution de R' : R' n'a pas de droit de préemption sur R.

- ***Mode de Couplage***

La notion du mode de couplage a été introduite, lors du travail effectué dans le cadre de projet HIPAC [DAY 88], afin de préciser le modèle d'exécution des règles actives. Les couplages considérés sont le couplage E-C, et le couplage C-A. Le mode couplage est caractérisé par le quadruplet suivant < mode de déclenchement, mode de transition, mode de synchronisme, mode de dépendance >.

- *Mode de Déclenchement*

Le mode de déclenchement spécifie quand la partie déclenchée est exécutée par rapport à la partie déclenchante.

Si le mode de déclenchement est *Immédiat* alors la partie déclenchée est considérée immédiatement à la suite de la partie déclenchante.
Exemple: si le mode de déclenchement pour E-C est immédiat alors la condition C est évaluée dès qu'un événement E est reconnu.

Si le mode de déclenchement est *Différé,* la partie déclenchée est différée à un point ultérieur de la transaction (ce point correspond généralement à la fin de la transaction et précède sa validation).

Remarque : Les couplages E-C et C-A d'une même règle peuvent combiner ces deux modes (E-C immédiat, C-A différé).

- *Mode de transaction*

Le mode de transaction sert à spécifier la transaction dans laquelle est exécutée la partie déclenchée, relativement à la partie déclenchante (même transaction ou séparé). La valeur « même transaction » signifie que la partie déclenchée s'exécute dans la même transaction que la partie déclenchante. La valeur « séparé » signifie que la partie déclenchée s'exécute dans une nouvelle transaction différente de la transaction déclenchante. Par exemple, la valeur « même transaction » pour (C-A), spécifie que l'action A est exécutée dans la même transaction que la condition C.

- *Mode de Synchronisation*

Le mode de synchronisation décrit le mode de déclenchement d'une règle qui peut se faire de manière *Synchrone* où la transaction est arrêtée le temps de l'exécution de la règle, ou *Asynchrone* où la transaction continue à s'exécuter en parallèle (si le SGBD supporte de lancer une transaction en parallèle avec la transaction déclenchante).

- *Mode de Dépendance*

Le mode de dépendance est optionnel lorsque le mode de transaction est séparé. Il spécifie la dépendance de la validation entre transaction déclenchante et transaction déclenchée. Le mode *Dépendant* désigne que la transaction déclenchée ne peut être validée que si la transaction déclenchante a été validée. Sinon le mode est *Independant*; dans ce cas la validation de la transaction déclenchée peut avoir lieu indépendamment de celle de la transaction déclenchante.

4.3.3 Caractérisation du Comportement d'un Ensemble de Règles

Quand plusieurs règles sont candidates à l'exécution, il faut déterminer un ordonnancement des règles qui produit un comportement cohérent du système. Deux plans d'exécution ont été adoptés [COL 96]:

- ***Plan d'Exécution Local*** (exécution de règles multiples)

Le plan d'exécution permet de traiter les règles déclenchées par un ou plusieurs événements. Quatre stratégies ont été proposées :

1. *Exécution séquentielle aléatoire* : Les règles sont exécutées l'une après l'autre dans un ordre aléatoire. Si les règles interfèrent entre elles, l'exécution ne sera pas déterministe.
2. *Exécution séquentielle ordonnée* : Les règles sont exécutées l'une après l'autre avec un ordre d'exécution déterminé de différentes manières.
3. *Exécution parallèle* : Il s'agit d'exécuter les règles actives dans plusieurs sous-transactions concurrentes ou bien dans des transactions séparées.

Un mécanisme de priorités entre règles est défini pour donner un ordre partiel aux règles :

- Associer une priorité numérique (Postgres),
- Spécifier des priorités relatives entre règles; c'est la solution la plus proposée (NAOS, STARBUST…) [WID 96],
- Définir une fonction de coût associée à chaque règle (EXACT)**.**

4. *Stratégie mixte* : La stratégie la plus simple, consiste à combiner les deux stratégies à savoir l'exécution parallèle et l'exécution séquentielle.

Remarque : le choix de la stratégie a une influence sur le déterminisme de l'exécution de l'ensemble de règles considéré.

- ***Plan d'Exécution Global*** (exécution en cascade)

L'occurrence d'un événement peut générer l'exécution d'un groupe de règles, et ce dernier peut à son tour produire d'autres événements et engendrer ainsi de nouvelles exécutions de règles. Dans ce cas, différentes stratégies d'exécution sont envisageables pour établir le plan d'exécution pour déterminer la façon dont va être exécuté l'ensemble de règles formé des règles déclenchées initialement R_I et l'ensemble des règles R_D déclenchées par l'exécution des règles de R_I. Le plan d'exécution consistera à ne pas autoriser les cascades ou à en limiter le nombre ou bien à traiter uniformément les règles de R_I et de R_D (les règles de R_D sont insérées dans l'ensemble R_I).

Exécution à plat : Insérer les règles R_D dans R_I et appliquer les différentes stratégies définies dans le plan d'exécution local.

Arbre d'exécution (cycle d'exécution) : Un cycle est défini par une séquence d'instructions (susceptibles de produire des événements) appartenant à un programme, transaction ou une règle. Les règles sont exécutées dans des cycles d'exécution.

Exemple : un événement e_1 déclenche R_1 et R_2. L'exécution de R_1 et R_2 produit deux événements e_2 et e_3 qui déclenchent à leur tour, trois règles R_{1a}, R_{1b} et R_{2a}. Ces règles sont définies avec la relation de précédence (<): $R_1 < R_2$ et $R_{1a} < R_{2a} < R_{1b}$.

Dans les règles immédiates[1] (exécution en profondeur d'abord), l'exécution d'une règle implique la création d'un nouveau cycle. Et l'exécution en profondeur implique que pour chaque événement produit dans un cycle les règles déclenchées par cet événement sont exécutées tout de suite, sans tenir compte des règles qui ne sont pas encore exécutées et déclenchées dans les cycles antérieurs.

Alors que dans les règles différées[2] (exécution en largeur d'abord), il faut exécuter à la fin de la transaction qui les a déclenchées, mais avant sa validation. Et un cycle n+1 exécute les règles déclenchées dans le cycle n puis dans un même cycle les règles sont exécutées selon la relation (<).

- ***Terminaison de l'exécution***

Pour les systèmes actifs, il n'est pas évident de vérifier les propriétés de terminaison et de confluence, vu la complexité due aux déclenchements multiples et cascadées,

Terminaison : L'exécution d'une règle est-elle assurée de se terminer, quelque soit l'opération déclenchée et quelque soit l'état préalable de la base de données ou du système ?

Confluence : L'exécution de règles non ordonnées produit-elle le même état final de la base de données, quelque soit son état initial et quelque soit l'ordre d'exécution de ces règles ?

La terminaison de l'exécution d'un ensemble de règles est souvent indécidable. Pour la détection des cas de non-terminaison, certains systèmes actifs ont envisagé une détection

1. Règle immédiate utilise la combinaison [immédiat, immédiat]
2. Règle différée utilise la combinaison [differé, immédiat]

empirique au moyen de délais de garde, ou de compteurs d'exécutions ; ceci permettra de limiter le nombre d'exécutions d'une règle. Par contre, d'autres systèmes ont imposé des restrictions syntaxiques aux définitions de règles pour écarter tout risque de non terminaison.

Mais dans la plupart des systèmes actifs, le programmeur doit lui-même s'assurer de la terminaison de règles (outils d'aide à la mise au point des applications).

4.4 Architecture Globale d'un Système Actif

L'architecture d'un système actif peut être en couche ou intégrée. Dans l'architecture en couche, le détecteur des événements et le gestionnaire des règles sont construits au dessus du SGBD pour donner des fonctions actives à un système passif sans modifier le système mais l'inconvénient réside dans le fait que le mécanisme de règles n'interagit pas avec les composants du SGBD (gestionnaire des transactions, par exemple). Cependant dans l'architecture intégrée le détecteur des événements et le gestionnaire des règles sont implantés à l'intérieur du SGBD, et la communication est directe avec les composants du SGBD. En plus la mise en œuvre de cette approche nécessite des modifications à l'intérieur du système. Cette architecture est adoptée par la plupart des prototypes de recherche et des produits commerciaux existants.

5. Les Systèmes Actifs

La recherche dans le domaine des bases de données actives s'est initialement concentrée sur le comportement actif dans les SGBD relationnels. Une deuxième génération de projets s'est investie dans les SGBD Orientés Objets. Nous présentons brièvement quelques systèmes basés sur le modèle Orienté Objets.

5.1 Le Système SAMOS

Le système SAMOS (Swiss Active Mechanism Based Object Oriented Database System) est un système de base de données actif orienté-objet. Il a été développé au sein de l'université de ZÜRICH en 1991 au dessus du SGBD ObjectStore. Ce système permet la définition et la détection d'événements complexes [GAT 94a].

Le système SAMOS présente des règles ECA internes ou externes aux classes. Les événements primitifs considérés sont : avant et après appel de méthode, des événements externes, transactionnels et temporels (absolus, périodiques et relatifs). Le système SAMOS construit les événements composites à partir des événements primitifs via des opérateurs tels que la disjonction, la séquence et la conjonction.

Son moteur d'exécution des règles considère les aspects suivants:

- Spécifier quand une condition doit-être évaluée et/ou une action doit être exécutée.
- Supporter l'exécution de règles multiples.
- Intégrer les opérations déclenchées dans un modèle de transactions SGBD.
- Tenir compte du mode de couplage (immédiat, différé, séparé).
- Imposer un ordre de priorité pour exécuter les règles multiples.

La règle ECA selon SAMOS est définie comme suit:

```
Rule_def ::=DEFINE rule_name
         ON event_clause
         IF condition_clause
         Do action_clause
```

Le système SAMOS via son noyau permet le signalement d'un événement, l'exécution de règle, la recherche et le stockage des objets tels que les événements et les règles.

Le gestionnaire de règles permet le stockage d'informations de définition de règles et d'événements : les règles et les événements sont créés en utilisant les méthodes offertes par le gestionnaire de règles. A la détection d'un événement, le gestionnaire est informé par la réception d'un message «`raise event`». Il réagit en traitant les opérations suivantes :

- Rechercher les événements correspondants,
- Informer le détecteur d'événements composés et,
- Rechercher la règle et informer l'exécuteur de règles.

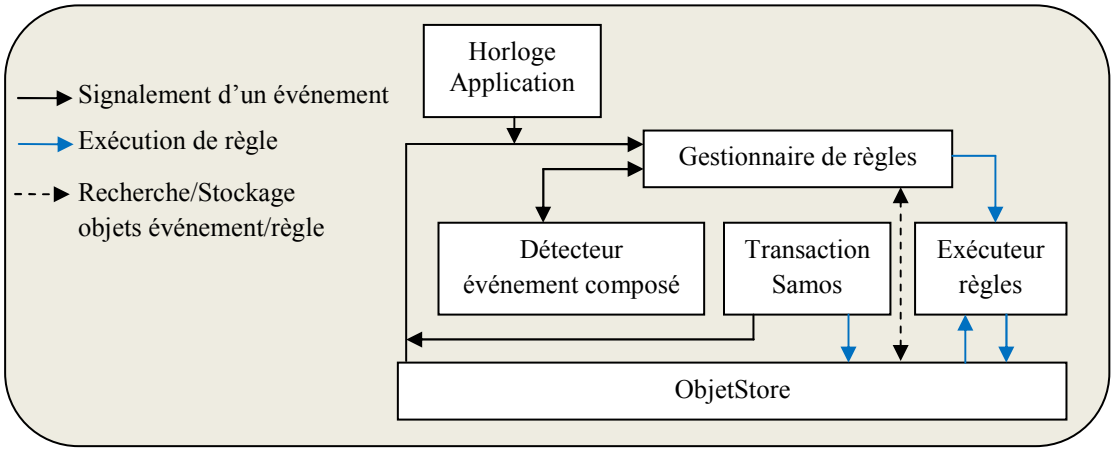


Figure 30. Structure du noyau SAMOS

Chaque type d'événement primitif requiert une façon différente pour sa détection. Par exemple, l'événement temporel est détecté par l'horloge du système, alors que la détection des événements composés est basée sur le mécanisme des RdP colorés.

Lorsqu'une seule règle est définie pour un événement, après sa détection, le mode de couplage associé à cette règle est immédiat et la règle est récupérée et exécutée. Chaque exécution simple de règle commence par l'évaluation de condition ensuite l'exécution d'action si la condition est satisfaisante [GEP 95].

Le système SAMOS supporte aussi l'exécution de règles multiples, quand un événement déclenche plusieurs règles. Il impose une priorité basée sur le mode de couplage pour exécuter les règles multiples.

- Si le mode de couplage est immédiat, l'exécuteur de règles détermine un index de la fonction condition et appelle cette fonction directement.

- Si le mode de couplage est différé, le déclenchement de la transaction est notifié pour ajouter l'évaluation de la condition dans la transaction spécifique de la règle.
- Si le mode de couplage est séparé, l'évaluation de la condition commence dans une nouvelle transaction.

5.2 Le Système NAOS [COU 96]

Le système NAOS (Native Active Object System) a été développé dans le cadre du projet ESPRIT II GOODSTAR (General Object Oriented Database Software Engineering Processes) en 1994. Il a enrichi le système à objets O2 en lui fournissant des définitions de règles ECA. Le système NAOS est un composant supplémentaire par rapport aux composants élémentaires (O_2C, interface C++, O_2SQL).

Les premières spécifications du système NAOS ont débuté en 1992. Le langage de règles et une première définition des constructeurs des événements composites ont été élaborés en 1993, et les aspects propres aux modèles d'exécution ont été finalisés avec Sevensen en 1994.

Comme s'est illustré dans la figure 31, les composants d'une application O_2 appartiennent à un schéma et manipulent les données stockées dans les bases associées à ce schéma.

Les règles de ce système sont basées sur le formalisme ECA.

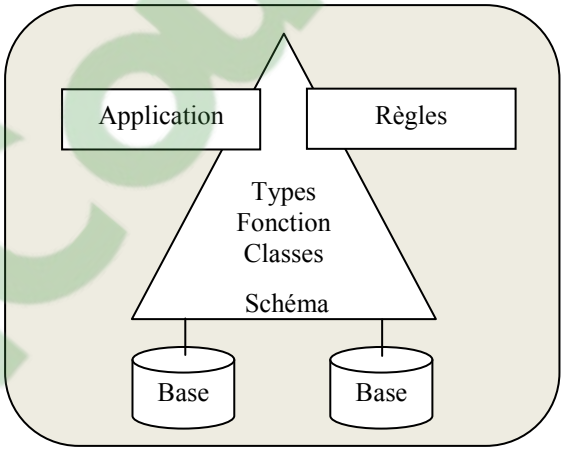


Figure 31. Les règles NAOS et le SGBD O_2

La Structure générale d'une règle NAOS est la suivante :

```
Create Rule < Identificateur de règle>
Precedes    < Liste de règles>
Coupling    < Mode de couplage>
On          < Spécification de type événement>
With        < Identificateur de delta structure>
If          <Condition>
Do [Instead] < Action>
```

Exemple de schéma O2 **:**

```
Class Personne public type tuple(
                nom: tuple(nom-famil:string,prenom:string),
                conjoint: Personne,
                age:integer,
                enfants:set(tuple(prenom:string,age:integer)))
       end;
Class Employe inherits Personne
Public type tuple(profil:string,salaire:integer) end;
Name Le-chef : Employe;
```

Pour la détection des événements, le système NAOS abonne les événements primitifs et détecte uniquement les événements abonnés. La détection est intégrée dans le noyau du système O2 pour minimiser et accélérer la communication avec O2. Il reconnait les événements composites grâce au graphe d'événements et signale les événements avec leur environnement pour passer à l'exécution.

Après la reconnaissance d'un événement, l'événement et son environnement sont envoyés à la fonction fournie au moment de l'abonnement. Cette fonction ″réveille″ le moteur d'exécution qui stocke l'événement dans un journal (Log) et exécute les règles déclenchées soit immédiatement, soit à la fin de la transaction. La construction des environnements d'exécution pour les règles est faite selon l'effet-net des opérations.

Quand un événement déclenche plusieurs règles le système impose un ordre total entre les règles en assurant un comportement déterministe. Les règles immédiates sont exécutées en profondeur d'abord et les règles différées sont exécutées en largeur d'abord.

5.3 Le Système HIPAC

Le système HIPAC (High Performance Active Database System) [DAY 88] est un projet de recherche lancé en 1987 par Computer Corporation of America (CCA) avec la collaboration du centre de recherches Xerox et de DEC.

L'objectif de ce projet était de concevoir un système orienté objet de gestion de bases de données actives en intégrant les règles actives ECA et de l'adapter aux applications ayant des contraintes temporelles et visant à couvrir des applications telles que : le contrôle d'intégrité, la dérivation automatique de données, et les règles d'inférence. Le projet HIPAC propose un modèle de connaissance qui décrit la sémantique des règles ECA et un modèle d'exécution qui spécifie comment ces règles sont exécutées dans les transactions de base de données.

Les événements reconnus par HIPAC peuvent être primitifs ou composites (la disjonction, la séquence et l'opérateur de fermeture). HIPAC a introduit en particulier, la notion de mode de couplage pour spécifier le mode d'exécution de la condition par rapport à l'événement.

Selon HIPAC, les règles sont traitées comme des objets de première classe et chaque règle est une instance de la classe règle. Les règles dans HIPAC sont définies en spécifiant un identificateur de règle, un événement, une condition, une action, des contraintes de temps, des plans d'urgence, et des attributs de la règle. Comme les règles sont des objets de première classe dans HIPAC, elles sont traitées de manière uniforme que d'autres objets dans le système.

HIPAC n'intègre pas toute forme de résolution de conflit lorsque plusieurs règles sont déclenchées. En fait, toutes les règles déclenchées sont exécutées simultanément sans critères de sérialisation. Le principal avantage de HIPAC est qu'il traite les règles de manière uniforme que d'autres objets. Les règles sont créées, supprimées et modifiées de la même façon que d'autres objets. Les opérations de règle sont implémentées comme méthodes qui assurent l'accès à l'état de la règle uniquement par son interface.

6. Les Avantages des Système Actifs

Les avantages apportés par les systèmes de gestion de base de données actifs sont déterminés par :

- La simplicité apportée aux programmes d'application, puisqu'une partie de programme peut être programmée avec les règles actives de SGBD.
- L'automatisation car les actions sont déclenchées automatiquement et sans l'intervention de l'utilisateur.
- L'augmentation de la fiabilité des données parce que plus il y a d'actions de vérification et de réparation, et mieux ça aide à la décision.
- L'augmentation de la flexibilité en réduisant le coût de développement et de maintenance.
- En plus, dans le contexte des entrepôts de données [POL 06], le système actif réduit ainsi le temps entre l'exploitation de l'information et les dernières mises à jour dans l'entrepôt de données.

7. Conclusion

Nous avons introduit dans ce chapitre les modèles qui structurent les systèmes actifs : modèles de connaissance, de détection et d'exécution des règles. La plupart des systèmes actifs existants adoptent la règle active de la forme Evénement-Condition-Action (règle ECA).

Du point de vue comportement, des recherches dans ce contexte ont mis au point trois mécanismes de détection s'appuyant sur les automates, les réseaux de Petri et les graphes, et différentes stratégies pour la conception du moteur d'exécution des règles.

Nous avons passé en revue quelques systèmes actifs qui diffèrent sur l'aspect descriptif des règles ECA, et sur le mode de détection des événements et d'exécution des règles.

Enfin, rappelons que les règles actives ont ouvert des perspectives pour prendre en considération les exigences des nouvelles applications telles que les applications multi média, à temps réel, travail coopératif,…

1. Introduction

La gestion de versions est essentielle dans les environnements où plusieurs utilisateurs manipulent une même base de ressources (outil de travail en groupe) dont les données sont partagées et sont en évolution continue. Ainsi dans ces cas, la gestion de versions se charge de fournir la dernière version à l'ensemble du groupe manipulant les mêmes données.

Comme les web services ne prennent pas en compte l'historique des modifications des documents échangés entre les différents web services et leurs clients ainsi que le suivi des modifications; nous proposons un modèle de gestion de versions basé sur les règles actives ECA afin de maintenir et d'accéder à l'historique des modifications pour les documents qui sont sous forme XML [BEN 12] [BEN 13a] [BEN 13b], et qui nous permet d'avoir pour chaque document la traçabilité de toutes les modifications effectuées au long de son cycle de vie dans un environnement distribué (caractéristique des Web Services).

L'approche proposée repose sur les deux stratégies « Verrouiller-Modifier-Déverrouiller » et « Copier-Modifier-Fusionner ». Elle s'appuie aussi sur l'utilisation des règles actives ECA pour la gestion du nombre de versions, le contrôle des données partagées et la gestion de propagation des modifications ainsi que pour les modifications du contenu des documents de type XML.

2. Le Modèle proposé

Nous proposons une nouvelle approche de gestion de versions destinée aux informations échangées dans les web services et entre les fournisseurs et leurs clients. Cette approche n'est pas destinée à gérer les différentes versions des services des web services, chose qui en cours de développement mais pas encore introduite actuellement dans le fonctionnement des web services, elle vise à gérer les différentes versions des documents qui sont utilisés dans les web services. Vue la nature des données échangées, nous nous intéressons à la gestion et au contrôle des documents qui sont sous forme XML.

2.1 L'Environnement du Travail

L'environnement du travail dans les web services est à la fois distribué pour l'ensemble des web services et l'ensemble des clients qui coexistent dans le Net, et centralisé pour un web service donné et l'ensemble de ses clients.

Afin de simplifier le suivi et la gestion des différentes versions des documents XML et pour ne pas limiter le modèle par les caractéristiques des web services, nous traitons la gestion de versions des documents indépendamment des plateformes des web services pour que par la suite nous puissions l'incorporer facilement dans les sites où se trouvent les applications des web services et les clients. Cette autonomie de gestion donne au modèle la possibilité de se développer facilement et indépendamment puis de s'intégrer dans d'autres domaines que les web services.

L'environnement de travail adopté est distribué (figure 32), contenant des groupes de travail centralisés. Un groupe de travail dispose d'un nombre d'utilisateurs, d'un dépôt où il stocke les données de son groupe de travail, et les différentes informations nécessaires pour la gestion de son groupe de travail.

Le nombre d'utilisateurs dans un groupe de travail peut être égal ou différent dans d'autres groupes de travail existants dans le système.

Le dépôt est situé dans un site central qui joue aussi le rôle de serveur central. La connexion entre les différents dépôts du système global distribué est obligatoire pour se synchroniser les uns avec les autres dans le but de mettre à jour leurs contenus. Les utilisateurs disposent chacun d'eux d'une copie de travail qui est mise à jour à travers le dépôt de son groupe.

Cependant en cas de panne, les utilisateurs se connectent automatiquement à un autre dépôt du système. Les dépôts existants ne se différencient pas ni dans le contenu ni dans le rôle que joue chacun dans le groupe de travail.

Le schéma global suivant (figure 32) illustre le dépôt central d'un groupe de travail vis-à-vis le système global distribué ainsi que les relations entre les différents utilisateurs et les bases de données.

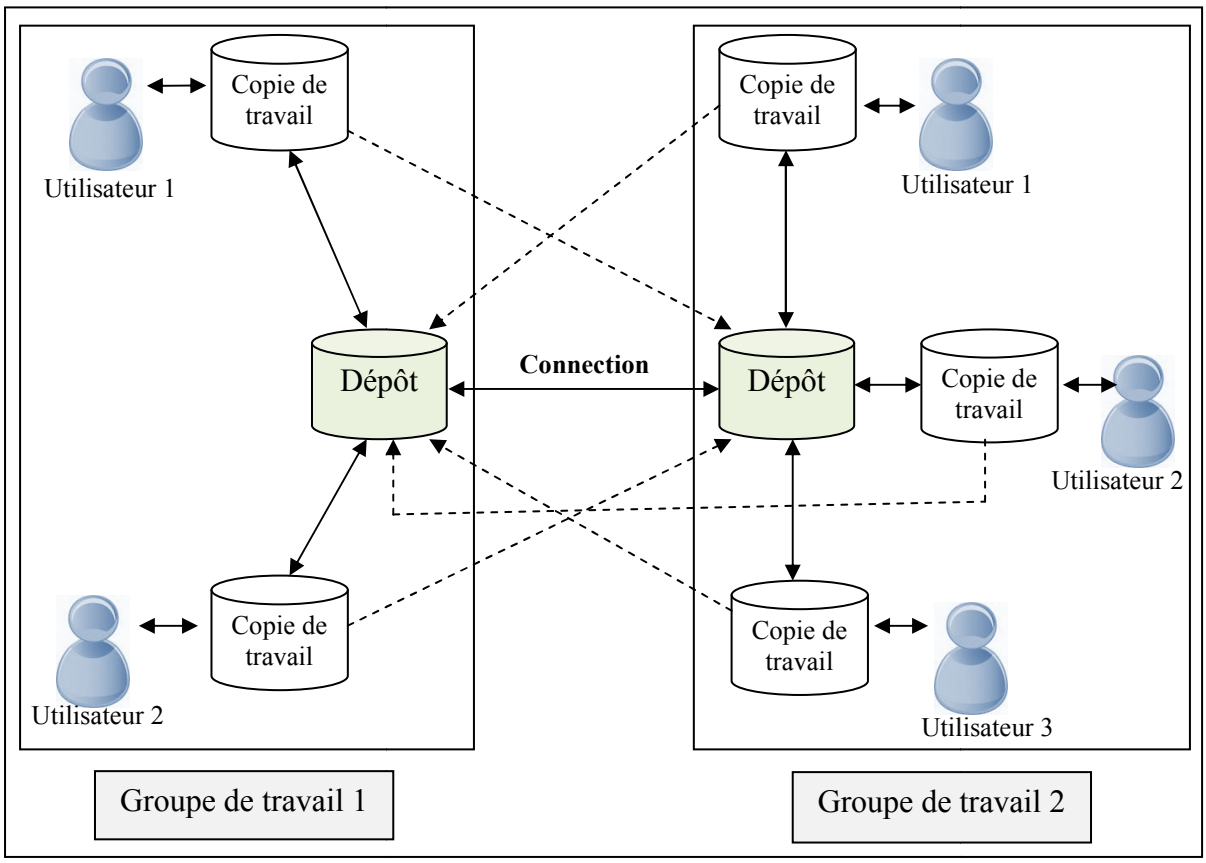


Figure 32. Schéma global du Modèle

Le modèle que nous proposons de la gestion de versions basée sur l'utilisation des règles actives ECA, est appliqué dans un groupe de travail et la gestion du système global est envisagée dans le futur.

2.2 L'Approche du Modèle Proposé

Le mécanisme de gestion de versions destinée aux documents XML de notre modèle repose sur les deux techniques « Verrouiller-Modifier-Déverrouiller » et « Copier-Modifier-Fusionner » des systèmes de gestion de versions des codes sources, dans un système centralisé où un seul dépôt de versions existe comme dépôt de référence. Ainsi, le modèle propose de sauvegarder un historique des différentes versions des documents dont dispose le système, d'autoriser le retour à une version antérieure quelconque et de garder un historique sur les opérations réalisées au niveau d'un document, leur date, leur utilisateur, etc…

Et afin de rendre cette gestion active et automatique, nous incorporons les règles actives ECA dans notre modèle pour exécuter toutes les fonctionnalités requises.

Dans l'environnement de travail il y'a plusieurs utilisateurs disposant chacun d'eux d'une base de données contenant un ensemble de documents de la dernière version sous forme XML comme copie de travail. Les contenus des copies de travail des utilisateurs peuvent être identiques comme ils peuvent être différents, en ayant quelques documents identiques. Les documents qui coexistent dans plus de deux copies de travail doivent contenir régulièrement le même contenu de la dernière version disponible dans le système.

Chaque utilisateur peut apporter des modifications aux documents en déclenchant un événement qui à son tour déclenche la règle active ECA correspondante et exécute des actions modifiant le document sélectionné. La modification d'un document donné est effectuée après que tous les utilisateurs soient informés et aient accepté la réalisation de cette modification.

Le résultat de cette opération est la création d'une nouvelle version de ce document qui remplacera l'ancienne version dans la base des copies de travail au niveau de chaque utilisateur tout en gardant l'ancienne version seulement dans la base centrale de tous les documents XML; et pour cela nous disposons aussi d'une méthode de suivi et de propagation de la nouvelle version.

Dans le cas d'accès concurrent à un document par plus d'un utilisateur, il est proposé de travailler selon le modèle « Verrouiller-Modifier-Déverrouiller » ou bien selon le modèle « Copier-Modifier-Fusionner ».

Si les utilisateurs déclenchent des modifications qui apporteront au contenu du document des transformations importantes, et qui pourront conduire par la suite à l'apparition des conflits lors de la fusion des résultats des modifications de chaque utilisateur, il faut verrouiller l'accès au document selon le modèle « Verrouiller-Modifier-Déverrouiller », pour éviter les conflits.

Cependant, si les modifications déclenchées par un utilisateur n'influenceront pas sur les modifications des autres utilisateurs et donc ne mèneront pas à l'apparition des conflits quand le système fusionne les résultats des modifications de tous les utilisateurs; le modèle « Copier-Modifier-Fusionner » s'impose.

Afin de réaliser ces opérations nous présentons une gestion de versions d'un document XML basée sur l'utilisation des systèmes actifs ECA dans un modèle client-serveur (centralisé), comme c'est le cas pour un service web et ses clients, pour sauvegarder toutes les actions effectuées dans le système et au niveau de chaque version pour un document XML donné.

Remarque : L'utilisation des règles ECA ne se focalise pas uniquement sur la modification des documents, mais elle est présente aussi dans la notification des utilisateurs sur les modifications établies dans les documents, dans l'activation du verrouillage des documents, et dans la gestion de l'arbre de version.

3. L'architecture du Modèle Proposé

Pour répondre aux objectifs du modèle proposé et afin de visualiser la gestion de versions des documents XML, il est proposé une architecture référence qui montre les différentes applications et modules qui assurent une gestion conforme des différentes versions pour un ensemble de documents.

L'architecture ci-dessous englobe le système de gestion de versions centralisé, d'un groupe de travail contenant deux utilisateurs et un dépôt central. Elle comprend deux types d'application : une application utilisateur et une application destinée au serveur central. Dans l'environnement des web services, le serveur central est le fournisseur de service et les utilisateurs sont les clients du service.

L'application du serveur central contient le dépôt où se trouvent toutes les données de la gestion et du suivi de toutes les versions des documents. Elle permet le contrôle d'accès au dépôt central, la gestion de l'historique du système, la réception des rapports sur le travail des utilisateurs, et la fusion des documents résultats des deux modifications apportées au même document.

Tandis que l'application utilisateur contient la copie de travail que nécessite l'utilisateur dans son travail. Elle a pour rôle de faciliter à l'utilisateur les opérations de mise à jour des documents dont il dispose. C'est à ce niveau, que le système capture toutes les informations nécessaires pour la gestion des opérations effectuées par l'utilisateur. L'application utilisateur dispose des modules suivants : *La Gestion du document XML*, *La Modification du document*, *Le Gestionnaire des versions du document XML*, et *La Gestion de la nouvelle version du document XML*.

Ainsi, la structure globale du système est illustrée dans l'architecture du système de gestion de versions basée sur les systèmes actifs ECA dans la figure 33.

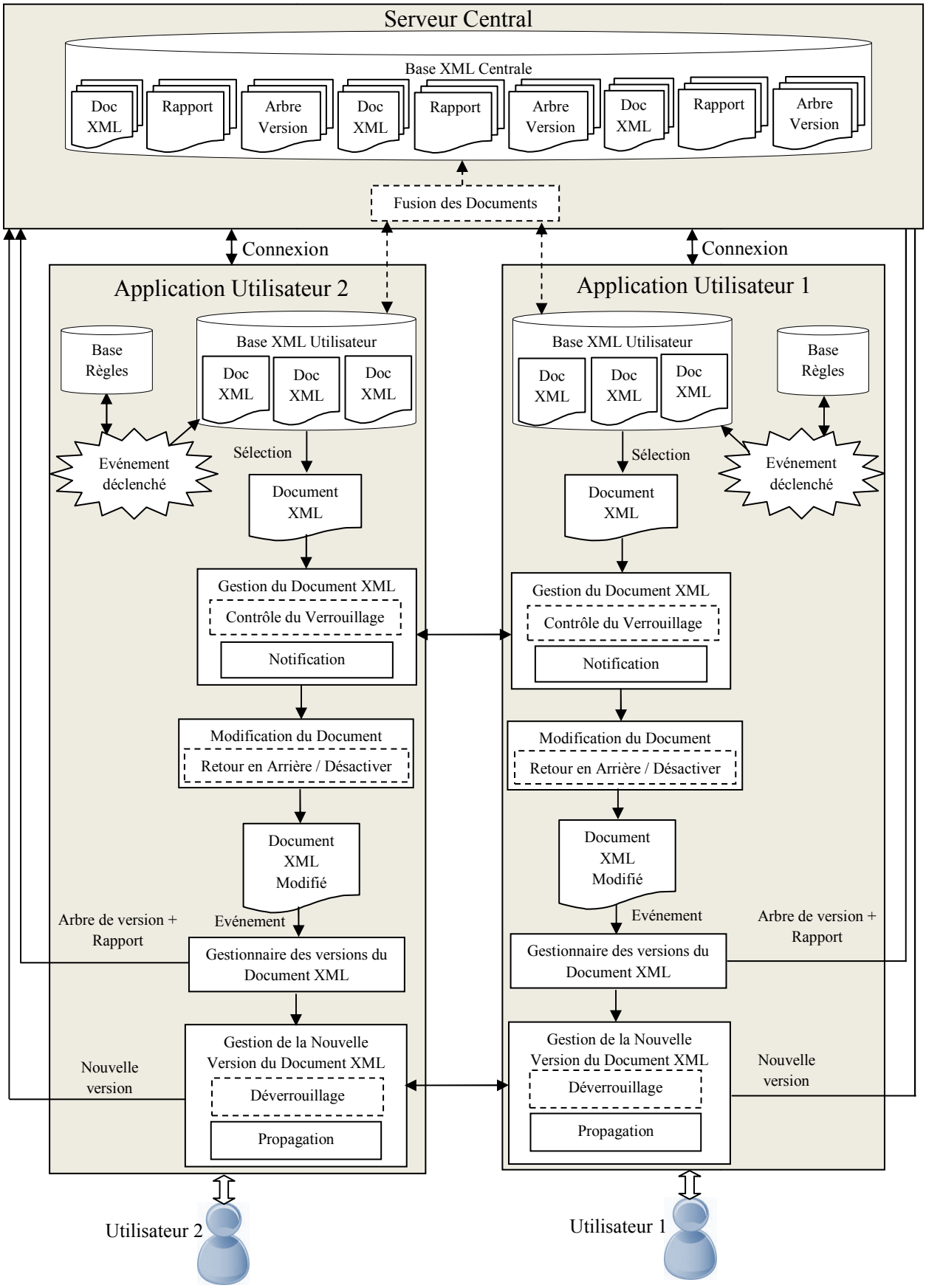


Figure 33. Architecture du système de gestion de versions basée sur les systèmes actifs ECA (Entre deux utilisateurs)

Dans ce qui suit nous allons détailler les différents composants de l'architecture de notre modèle.

Remarque : Nous supposons que les deux utilisateurs sont connectés au serveur central.

3.1 Les Bases de Données

Le système dispose de trois types de base de données : La *Base XML Centrale*, la *Base XML Utilisateur*, et la *Base Règles* représentant respectivement le dépôt central, la copie de travail et l'ensemble des règles modifiant la *Base XML Utilisateur* et la *Base XML Centrale*.

3.1.1 La Base XML Centrale

La *Base XML Centrale* se trouve dans le serveur central, elle contient toutes les informations nécessaires pour le suivi et la gestion des modifications ainsi que tout l'historique du système. La *Base XML Centrale* stocke tous les documents du groupe de travail sous forme XML et leurs différentes versions dans des répertoires. Chaque document lui est attribué un répertoire. Ce répertoire contient toutes les versions du document, suivi d'un rapport des modifications qui résume l'ensemble des opérations effectuées sur le document de la version précédente, et l'arbre de version. L'arbre de version schématise le toutes les versions du document. Les répertoires sont stockés dans un fichier.

La *Base XML Centrale* est unique dans un groupe de travail, elle se localise dans un site central accessible uniquement par les utilisateurs connectés.

Remarque : La version initiale d'un document n'a pas de rapport de modification, et l'arbre de version existe seulement si le document a subi au moins une modification.

3.1.2 La Base XML Utilisateur

La *Base XML Utilisateur* représente l'environnement de travail nécessaire pour un utilisateur, c'est la copie de travail proprement dit. Chaque utilisateur lui est attribué une *Base XML Utilisateur*. Cette dernière contient uniquement la dernière version de chaque document sous forme XML dont l'utilisateur exige dans son travail. Elle est mise à jour chaque fois qu'il y a une modification par un utilisateur au niveau d'un document qu'elle contient. Lorsqu'un événement est déclenché par un utilisateur, il conduit à la modification du document et à la création d'une nouvelle version du document puis le système, dans ce cas, remplacera l'ancienne version du document par la nouvelle version.

Les *Bases XML utilisateur* ne contiennent pas forcément les mêmes documents; cependant si elles partagent un ensemble de documents, le contenu de ces documents doit être le même. Ces bases doivent régulièrement contenir les documents des versions les plus récentes qui existent dans la *Base XML Centrale*.

Le schéma suivant (figure 34) illustre un exemple d'une *Base XML Utilisateur* pour deux utilisateurs en connexion avec une *Base XML Centrale* contenant toutes les versions des documents avec des exemples sur le contenu de chaque *Base XML Utilisateur*.

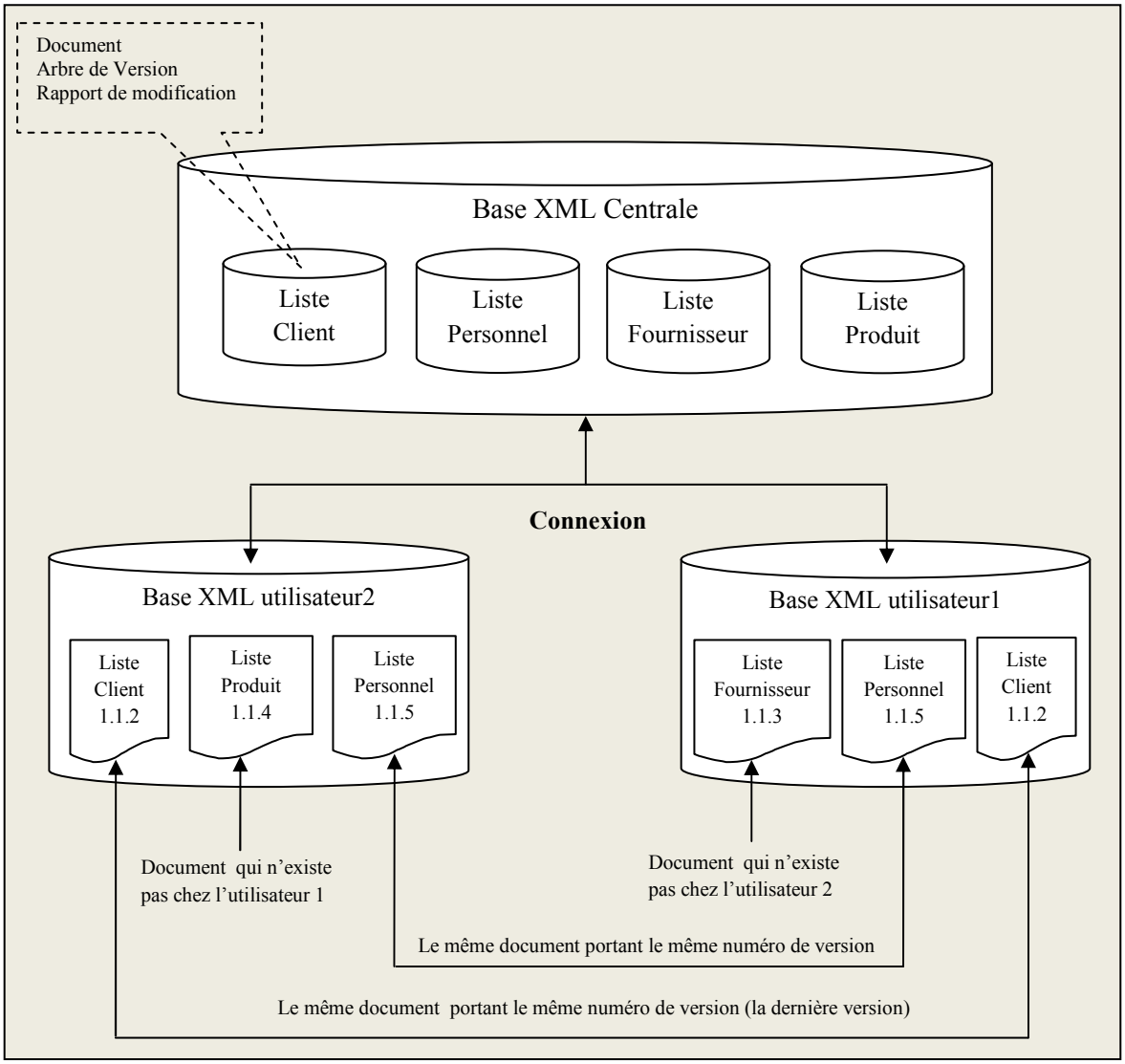


Figure 34. Exemple de Base XML utilisateurs de deux utilisateurs en connexion avec la Base XML Centrale

Notons que la connexion entre la *Base XML Centrale* et les deux *Bases XML Utilisateurs* est établie lorsqu'un utilisateur désire consulter le contenu d'une version antérieure d'un document et le rapport de modification correspondant ou s'il souhaite effectuer des modifications. Cette connexion offre aux utilisateurs une extension de l'environnement de travail quand ils le demandent et selon leurs besoins.

3.1.3 Le Document XML

L'utilisateur sélectionne un document parmi les documents XML qui existent au niveau de sa *Base XML Utilisateur* ou dans la *Base XML Centrale* s'il s'agit d'un document d'une version antérieure. Ce document représente le document qui va subir des modifications. L'ensemble des actions effectuées sur le document sont déterminées suite au déclenchement de la règle ECA, correspondant à l'événement produit. Le document sélectionné représente l'entrée du module *Gestion du Document XML*.

Quand l'utilisateur sélectionne un document de version antérieure à partir de la *Base XML Centrale*, l'application utilisateur établit une connexion entre l'utilisateur et le serveur central afin de récupérer une copie du document sélectionné.

Un document XML sélectionné peut être modifié si après sa sélection un événement est déclenché, il peut remplacer la dernière version, et il peut être aussi désactivé. Notons qu'une version antérieure d'un document ne peut pas remplacer la dernière version de ce document plus d'une fois et que la désactivation d'un document implique le blocage de toute opération sur ce document qui sera ainsi accessible seulement en lecture.

Exemple de document XML :

Le document Liste Client (voir le document complet dans l'annexe 1):

```
<?xml version="1.0" encoding="UTF-8"?>
<clients>
  <client Num="C001">
    <nom>Abed</nom>
    <prenom>Ahmed</prenom>
    <adresse>Arzew</adresse>
    <num_telephone>0611111111</num_telephone>
  </client>
  <client Num="C002">
    <nom>BenAhmed</nom>
    <prenom>Siham</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0722222222</num_telephone>
  </client>
  <client Num="C003">
    <nom>Brahimi</nom>
    <prenom>Mohamed</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0733333333</num_telephone>
  </client>
  <client Num="C004">
    <nom>Dali</nom>
    <prenom>Slimane</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0644444444</num_telephone>
  </client>
</clients>
```

Figure 35. Extrait du document Liste Client 1.0.0

Remarque : Un document XML ne peut être supprimé, puisque le système ne supporte pas cette opération.

3.1.4 L'Accès aux Bases de Données

Pour accéder à la *Base XML Utilisateur*, l'utilisateur doit se connecter au serveur central. Cette connexion lui permettra de consulter les documents de sa *Base XML Utilisateur* et ceux de la *Base XML Centrale*. Cette étape nous permet d'imposer une connexion au serveur central même si l'utilisateur ne consultera pas le contenu de la *Base XML Centrale.* Elle nous

garantit l'envoi des modifications effectuées par les utilisateurs au serveur central connecté afin de mettre à jour régulièrement la *Base XML Centrale*.

Pour l'accès, un nom utilisateur et un mot de passe, attribués par le serveur, doivent être introduits.

3.1.5 La Base Règles

L'application utilisateur dispose d'une *Base Règles* contenant l'ensemble des règles actives ECA dont ses événements peuvent survenir pour modifier les documents de la *Base XML Utilisateur*. Chaque document lui est associé un ensemble de règle ECA, qui est accessible dans l'application utilisateur par une liste de l'ensemble des événements. L'événement sélectionné déclenche la règle ECA correspondante pour modifier le contenu du document.

1. *Exemple d'événement primitif pour le document Liste Client* :

Evénement : le client «C004» déménage.
L'événement est de type primitif, à sa détection l'adresse et le numéro de téléphone fixe du client, dans le document Liste Client sont mis à jour.

```
Règle ECA : Evénement : changement de local du client «C004»
            Condition : Si le numéro client égal à «C004»
            Action : Modifier l'adresse du client & Modifier le numéro de
                     téléphone dans le document Liste Client
```

2. *Exemple d'événement composé pour le document Liste client* :

Evénement : Ajouter le champ email ensuite Ajouter le client «C011».
L'événement est composé de deux événements primitifs en séquence : Ajouter l'information email dans le document Liste Client puis insérer le client « C011 ». La règle ECA correspondante est définie comme suit :

```
Règle ECA : Evénement : Insertion du champ email; insertion d'un client
            Condition : Si le numéro client égal à «C011»
            Action : Insérer le client «C011» qui contient le champ email
```

Remarque : Le ";" dénote le constructeur de séquence.

Remarque : La liste de tous les événements utilisés dans le système est disponible dans l'annexe 2.

3.1.6 Le document XML Modifié

Le document XML modifié représente le résultat du module Gestion du Document et de l'événement qui déclenche la règle ECA qui traite les différentes versions d'un document XML. Cet élément correspond à l'événement déclenchant de la gestion de versions des documents par la suite.

3.2 Les Modules du Modèle Proposé

3.2.1 Gestion du Document XML

Le module *Gestion du Document XML* est en mesure de gérer le document sélectionné afin d'organiser l'accès concurrent à ce document par les utilisateurs connectés (dans notre cas il y a deux utilisateurs connectés au serveur central) avant d'entamer la modification de ce document et il organise la communication entre les utilisateurs. Il contient deux principaux composants qui interviennent au même temps, le module du *Contrôle du Verrouillage* et le module de la *Notification*.

3.2.1.1 La Notification

Ce module se charge d'informer les utilisateurs connectés qu'il y aura une modification au niveau d'un document en précisant le nom de ce document et l'événement déclenché. Ce module est construit d'un ensemble de règles ECA qui gère la modification du document par un utilisateur, selon les intentions du deuxième utilisateur vis-à-vis de cette modification.

Initialement, le module de *Notification* informe via un message les utilisateurs connectés qu'un utilisateur particulier a déclenché un événement sur un document. Ensuite il les invite d'accepter, de refuser ou de bloquer la modification déclenchée par cet utilisateur.

Dans notre système, quand l'utilisateur1 informe l'utilisateur2 qu'il désire apporter des modifications dans un document particulier, l'utilisateur2 doit donner son avis au système de la gestion sur ces modifications, selon ses besoins. L'avis de l'utilisateur2 est ensuite retransmis à l'utilisater1.

L'opération de l'utilisateur1 est exécutée si l'utilisateur2 accepte la modification, sinon l'utilisateur2 peut émettre :

1. Un refus et la modification souhaitée par l'utilisateur1 est arrêtée par le système,
2. Un blocage de l'opération afin d'apporter ses propres modifications. A la fin de cette opération, le système invite l'utilisateur1 à exécuter sa modification qui a été bloquée.

Les deux dernières décisions de l'utilisateur2 (l'arrêt et le blocage de la modification) deviennent effectives seulement si l'utilisateur1 accepte de les mettre en application. Dans le cas contraire, l'utisateur1 obligera l'utilisateur2 d'accepter la modification du document et il imposera la création d'une nouvelle version du document.

Les échanges entre les deux utilisateurs sont illustrés par le diagramme de séquence ci-dessous (figure 36). Ce diagramme représente la succession chronologique des décisions prises par l'utilisateur2 suite à la modification du document déclenchée par l'utilisateur1 et les actions qui seront effectuées par les applications des utilisateurs.

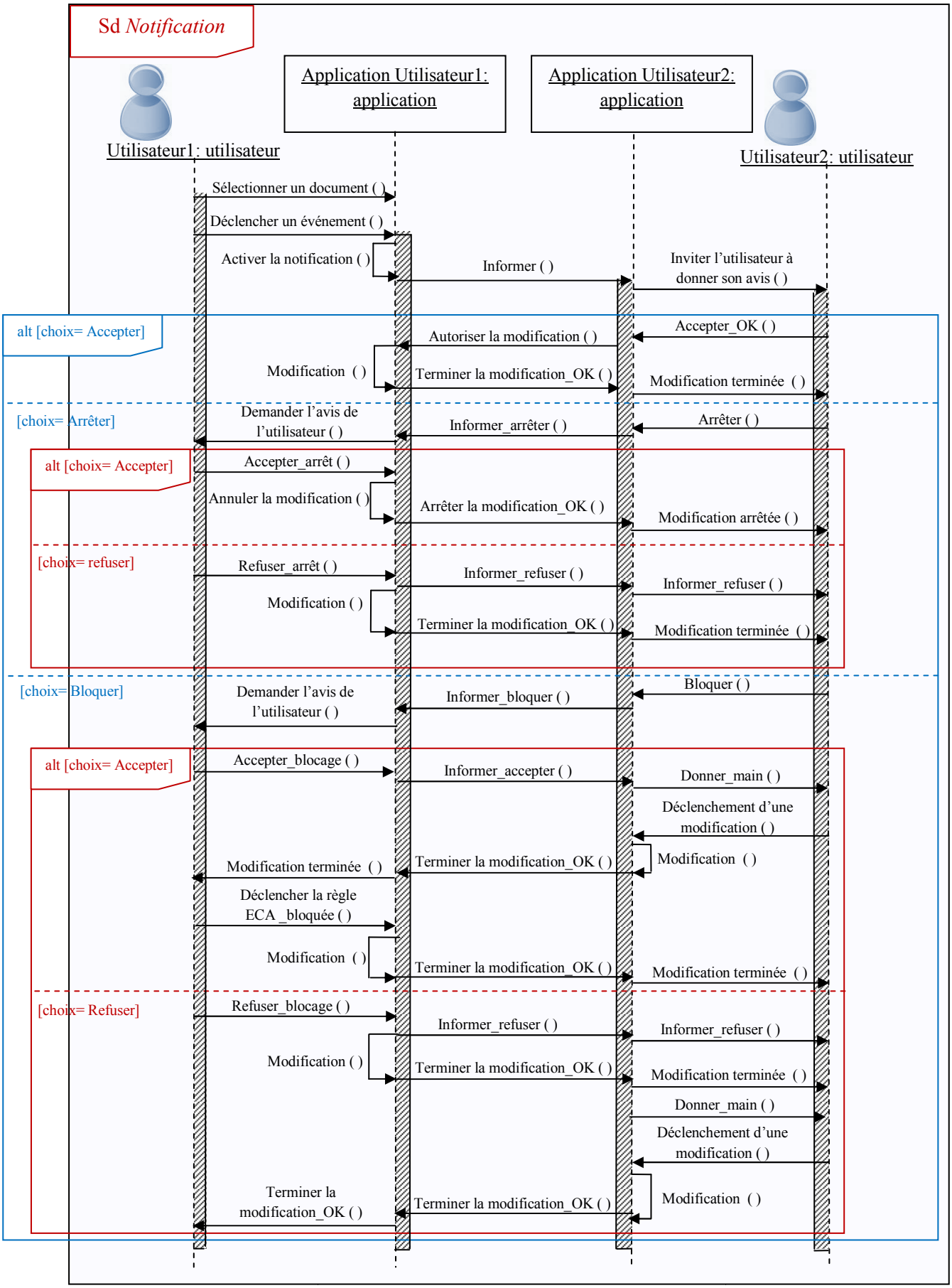


Figure 36. Diagramme de séquence du composant Notification

Remarque : On suppose que l'utilisateur1 commence le déclenchement de la modification pour un document qu'il a sélectionné.

Nous traduisons toutes ces alternatives par les règles actives ECA suivantes :

```
Règle ECA 1 : Evénement : Acceptation de la modification par l'utilisateur2
              Condition : Si avis=accepter
              Action : Lancer la modification de l'utilisateur1

Règle ECA 2 : Evénement : Arrêt par l'utilisateur2; acceptation de l'arrêt
                          par l'utilisateur1
              Condition : Vrai
              Action : Annuler la modification et arrêter l'exécution du
                       système

Règle ECA 3 : Evénement : Blocage par l'utilisateur2; acceptation du
                          blocage par l'utilisateur1
              Condition : Vrai
              Action : Déclencher la modification de l'utilisateur2, après
                       Déclencher la modification de l'utilisateur1

Règle ECA 4 : Evénement : Blocage par l'utilisateur2; Refus du blocage par
                          l'utilisateur1
              Condition : vrai
              Action : Déclencher la modification de l'utilisateur1, après
                       Déclencher la modification de l'utilisateur2

Règle ECA 5 : Evénement : Arrêt par l'utilisateur2; Refus de l'arrêt par
                          l'utilisateur1
              Condition : vrai
              Action : Déclencher la modification de l'utilisateur1
```

3.2.1.2 Le Contrôle du Verrouillage

Le principe du module *Contrôle du Verrouillage* est de bloquer l'accès en écriture au document qui subira des modifications par un utilisateur particulier, et quand le deuxième utilisateur accepte la modification et néglige le message du module du *Notification*. Lorsque l'utilisateur déclenche une modification, le verrouillage du document est activé si les actions de cette modification sont différentes par rapport à l'ensemble des actions pouvant être appliquées sur le document sélectionné auparavant par l'autre utilisateur.

La nécessité du verrouillage s'impose lorsque les deux utilisateurs sélectionnent le même document et déclenchent un événement de modification importante sur le contenu de ce document et lorsque la modification de l'un influence sur le résultat de l'autre (cas où l'utilisateur entame la modification du document pendant que le premier utilisateur n'a pas encore déposé sa dernière version). Le verrouillage du document garantit aux utilisateurs de manipuler la dernière version.

Cependant, le verrouillage d'un document n'est pas nécessaire lorsque les opérations des utilisateurs sont indépendantes l'une de l'autre, et le fait de verrouiller interdit le travail en parallèle et provoque une perte de temps inutile. Si les deux utilisateurs désirent insérer des données dans le même document (ou quand ils désirent supprimer des données dans le même document), la modification de l'un n'influencera pas sur la modification de l'autre. Ils pourront facilement éditer le document simultanément, sans que cela ne pose de problème et

sans passer par le verrouillage du document, et ensuite il suffit de fusionner les deux modifications.

Exemple 1: Cas où le Verrouillage est inutile

L'exemple suivant montre l'utilisation du verrouillage qui s'avère inutile.

Soit le document concernant le client identifié par le numéro `C001`, le nom `Abed`, le prénom `Ahmed`, et l'adresse `Arzew`.

L'utilisateur1 désire insérer le champ email `@email` au client qui a le numéro `Num="C001"` et l'utilisateur2 désire insérer le champ du numéro de téléphone `num_tel` pour le même client.

Soient les deux scénarios suivants :

1. Dans le premier scénario, la première démarche est effectuée par l'utilisateur1. L'utilisateur1 verrouille le document Liste Client et exécute l'insertion de l'email du client C001, pendant que l'utilisateur2 est bloqué et ne peut accéder au document qu'en lecture. Lorsque l'utilisateur1 termine, il libère le document, et l'utilisateur2 à son tour verrouille le document et exécute l'insertion du champ numéro de téléphone du client (à cet instant l'utilisateur1 est bloqué). A la fin de l'opération, il libère le document.

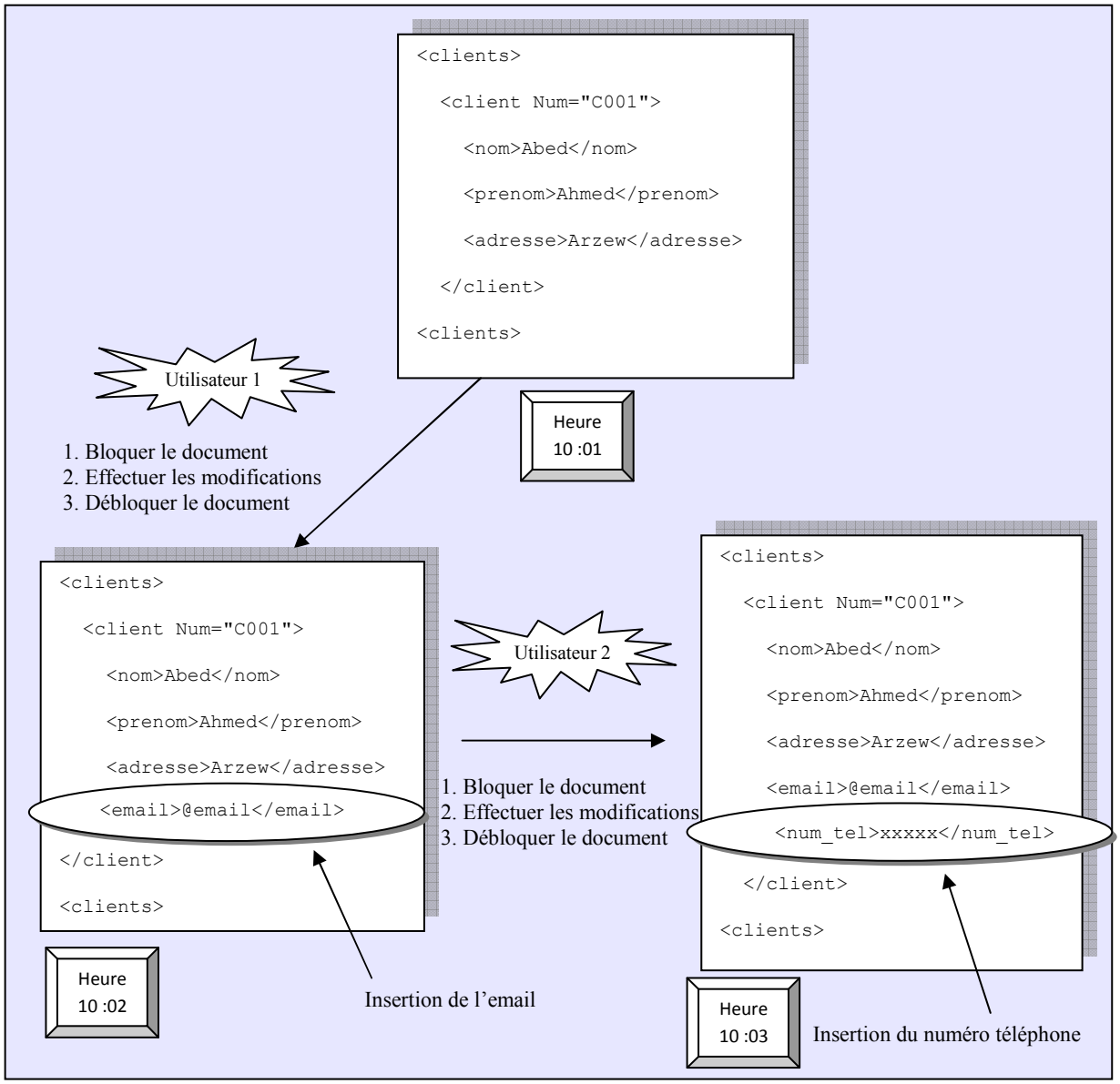


Figure 37. Scénario 1 de l'exemple 1 (avec Verrouillage)

2. Dans le deuxième scénario le module *Contrôle du Verrouillage* n'active pas le verrouillage du document pendant l'insertion des données dans le document sélectionné en même temps par les deux utilisateurs (l'utilisateur 1 déclenche la règle ECA de l'insertion de l'email, et l'utilisateur 2 de l'insertion du numéro de téléphone).

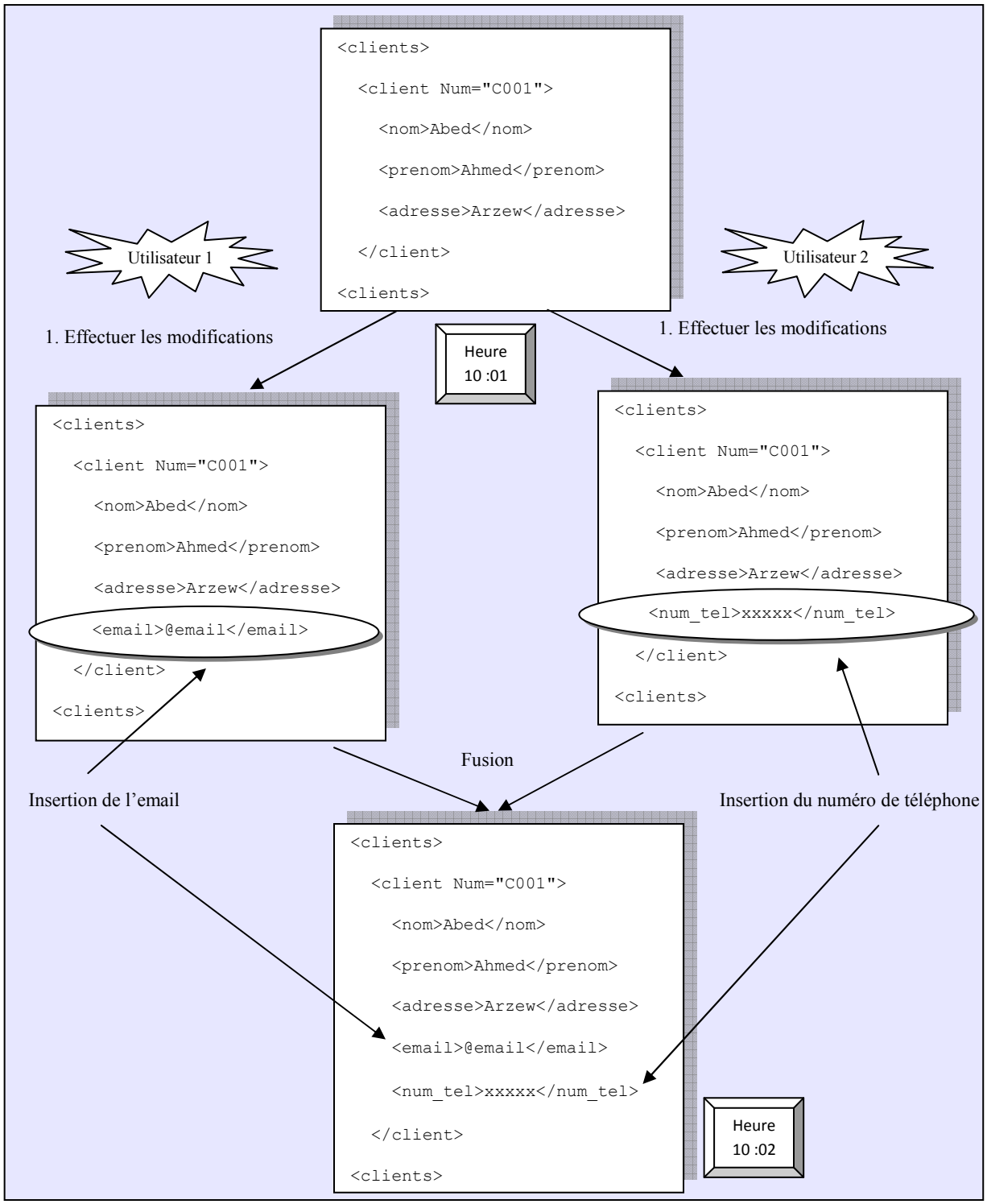


Figure 38. Scénario 2 de l'exemple 1 (sans intervention du Verrouillage du document)

Le résultat de ces deux scénarios est le même, mais il consomme un temps d'attente lorsqu'un utilisateur est bloqué alors que les deux utilisateurs peuvent effectuer en même temps l'exécution de l'insertion de l'email et du numéro de téléphone simultanément. A la fin des deux opérations, les deux résultats sont fusionnés et le résultat trouvé est le même que dans le premier scénario.

Remarque : Nous supposons que la sélection du document et la modification du document nécessitent en moyenne une minute pour chaque utilisateur.

L'exemple1 montre l'inutilité de travailler selon le modèle « Verrouiller-Modifier-Déverrouiller » dans un cas où le verrouillage du document en commun engendre une attente. Afin d'éviter à l'utilisateur l'attente inutile, il est intéressant de travailler selon le modèle « Copier-Modifier-Fusionner ».

Exemple 2: Cas où le Verrouillage est nécessaire

Supposons que l'utilisateur1 désire modifier la valeur de l'adresse du client du document client dont le numéro est "`C001`" tandis que l'utilisateur2 désire insérer le champ email pour le même client et que l'utilisateur1 entame l'opération de modification.

1. Considérons le premier scénario, l'utilisateur1 verrouille le document et exécute la modification de l'adresse du client pendant ce temps l'utilisateur2 est bloqué. Une fois la modification effectuée le document est déverrouillé, et l'utilisateur2 verrouille à son tour le document et exécute l'insertion du champ email au client C001, à cet instant l'utilisateur1 est bloqué. Lorsque l'utilisateur2 termine, il libère le document.

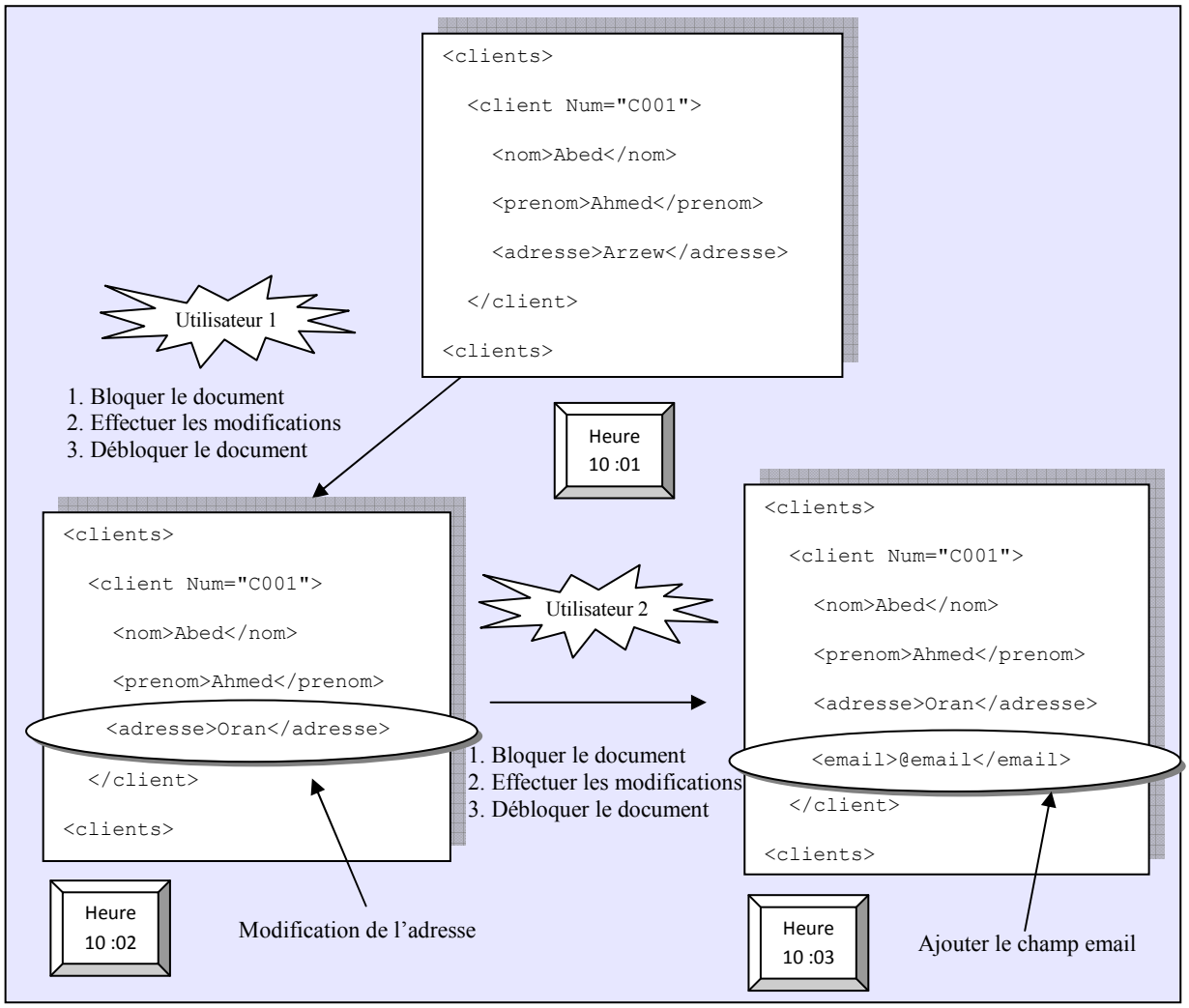


Figure 39. Scénario 1 de l'exemple 2 (avec Verrouillage du document)

2. Dans le deuxième scénario, le module Contrôle du Verrouillage n'active pas le verrouillage du document. Les deux utilisateurs déclenchent en même temps deux événements différents sur le même document client.

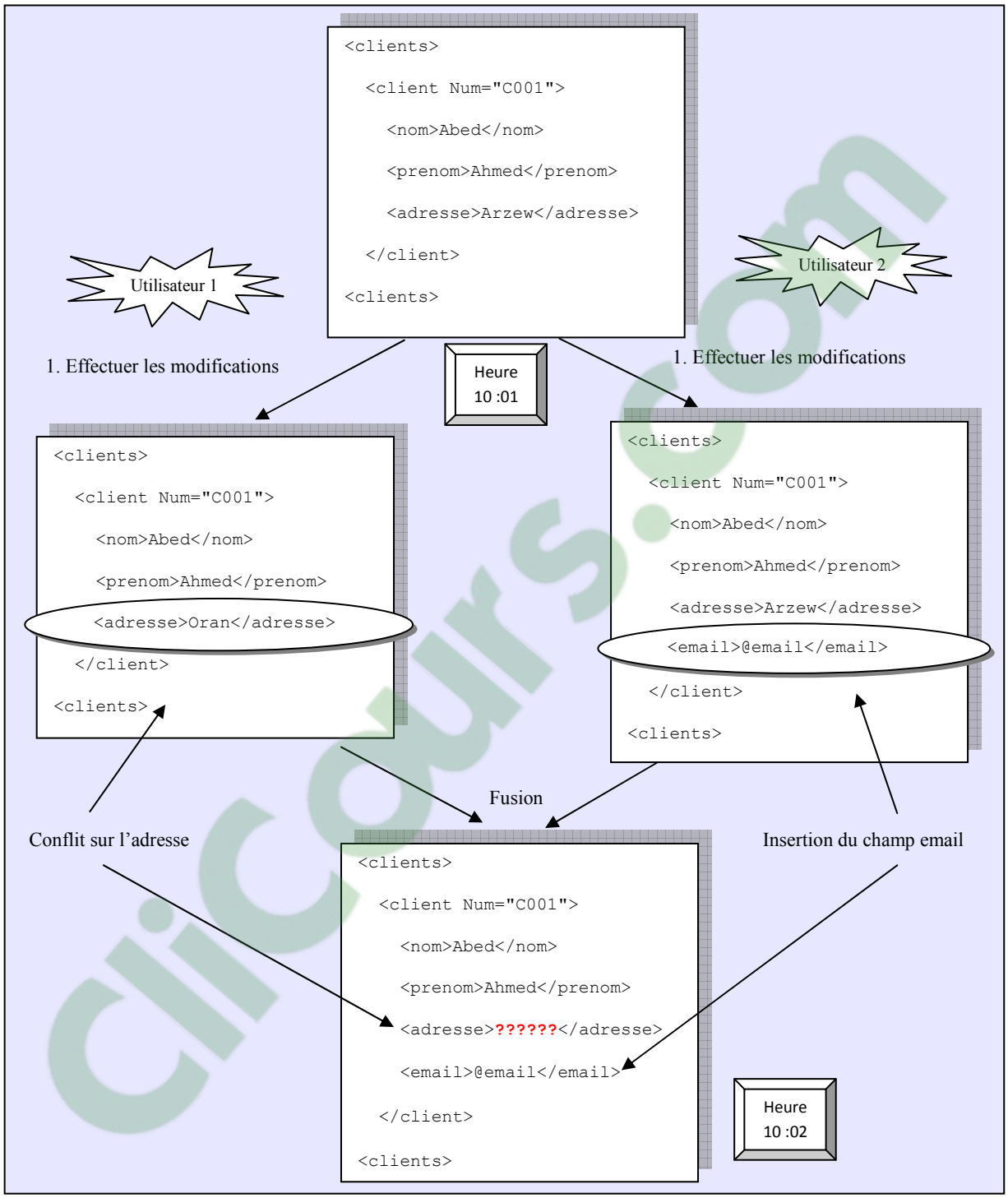


Figure 40. Scénario 2 de l'exemple 2 (sans verrouillage du document)

Le résultat de la fusion des deux documents modifiés engendre un conflit au niveau de la valeur de l'adresse.

Cet exemple montre un cas où le verrouillage du document partagé est important et travailler selon le modèle «Copier-Modifier-Fusionner» ne devient plus intéressant puisqu'il provoque des conflits lors la fusion. Le modèle «Verrouiller-Modifier-déverrouiller» dans des cas pareils, est intéressant.

Pour remédier à ces problèmes de conflits qui peuvent être engendrés lors des opérations de modification, nous ajoutons une information au niveau de la règle ECA. Cette information supplémentaire représente la densité des actions de la règle déclenchée.

La densité est de type booléen dont la valeur vrai ou faux, dépend du type de l'action et de son influence sur le contenu des documents.

La densité prend la valeur « vrai » quand les actions de la règle ECA apportent des changements totaux au document sélectionné. Il est nécessaire d'activer le verrouillage du document pour les autres utilisateurs et le composant *Contrôle du Verrouillage* par conséquent active le verrouillage du document selon le modèle « Verrouiller-Modifier-Déverrouiller ».

En revanche, si les actions de la règle ECA apportent des changements partiels au document (l'ajout ou la suppression d'une partie du document), nous affectons alors la valeur « faux » à la densité. Dans ce cas, il n'est pas nécessaire d'activer le verrouillage du document pour l'accès en écriture des autres utilisateurs (donner la main aux utilisateurs pour réaliser leurs modifications). Le type des actions exécutées par les deux règles déclenchées par les utilisateurs doivent être de même type, c'est-à-dire si le premier utilisateur effectue une insertion alors le deuxième effectue aussi une insertion quelque soit la valeur des données insérées mais dans des parties différentes du document, ou si le premier utilisateur effectue une suppression alors le deuxième effectue aussi une suppression quelque soit la valeur des données supprimées.

Cette information a pour but de rendre le système décisionnel et la valeur de la densité précisera le comportement du composant *Contrôle du Verrouillage* du module *Gestion du Document XML*.

Pour la gestion du composant *Contrôle du Verrouillage,* nous envisageons deux politiques dépendant du fait que le document sélectionné par un utilisateur est utilisé ou non par l'autre utilisateur.

Quand l'utilisateur1 sélectionne un document déjà utilisé par l'utilisateur2, pour une modification, une vérification de la densité de l'opération déclenchée par l'utilisateur2 s'impose et l'application utilisateur1 s'exécute selon sa valeur :

Si densité = vrai

- L'utilisateur1 doit attendre que l'utilisateur2 termine la modification du document.
- Après que l'application utilisateur2 termine, l'utilisateur1 pourra modifier la dernière version du document.
- L'application utilisateur1 via le composant *Contrôle du Verrouillage* verrouille le document.
- Cette application déclenche la modification ensuite elle envoie les résultats de cette modification à l'utilisateur2 et au serveur central.
- Le document est déverrouillé par le composant *Déverrouillage*.

Si densité = Faux

- L'utilisateur1 peut effectuer sa modification en parallèle de la modification de l'utilisateur2.
- Quand l'utilisateur2 termine son opération, il attend la fin de la modification de l'utilisateur1, pour que ce dernier notifie le serveur central qui pourra lancer l'opération de fusion.

Ce scénario est représenté dans le diagramme de séquence de la figure 41.

Cependant, quand l'utilisateur1 sélectionne un document non utilisé et veut effectuer une modification. L'application utilisateur1 fonctionne selon la valeur de la densité.

Si densité = vrai

- Le document est verrouillé par le composant *Contrôle du Verrouillage* de l'application utilisateur1.
- La modification est déclenchée et les résultats de cette modification seront envoyés à l'utilisateur2 et au serveur central.
- Le document est déverrouillé par le composant *Déverrouillage*.

Si densité = Faux

- L'utilisateur1 effectue sa modification et envoie le résultat à l'utilisateur2 et au serveur central.
- Dans le cas où le document est utilisé par le deuxième utilisateur, le système fonctionnera selon le scénario d'un document en utilisation.

Ce scénario est représenté dans le diagramme de séquence de la figure 42.

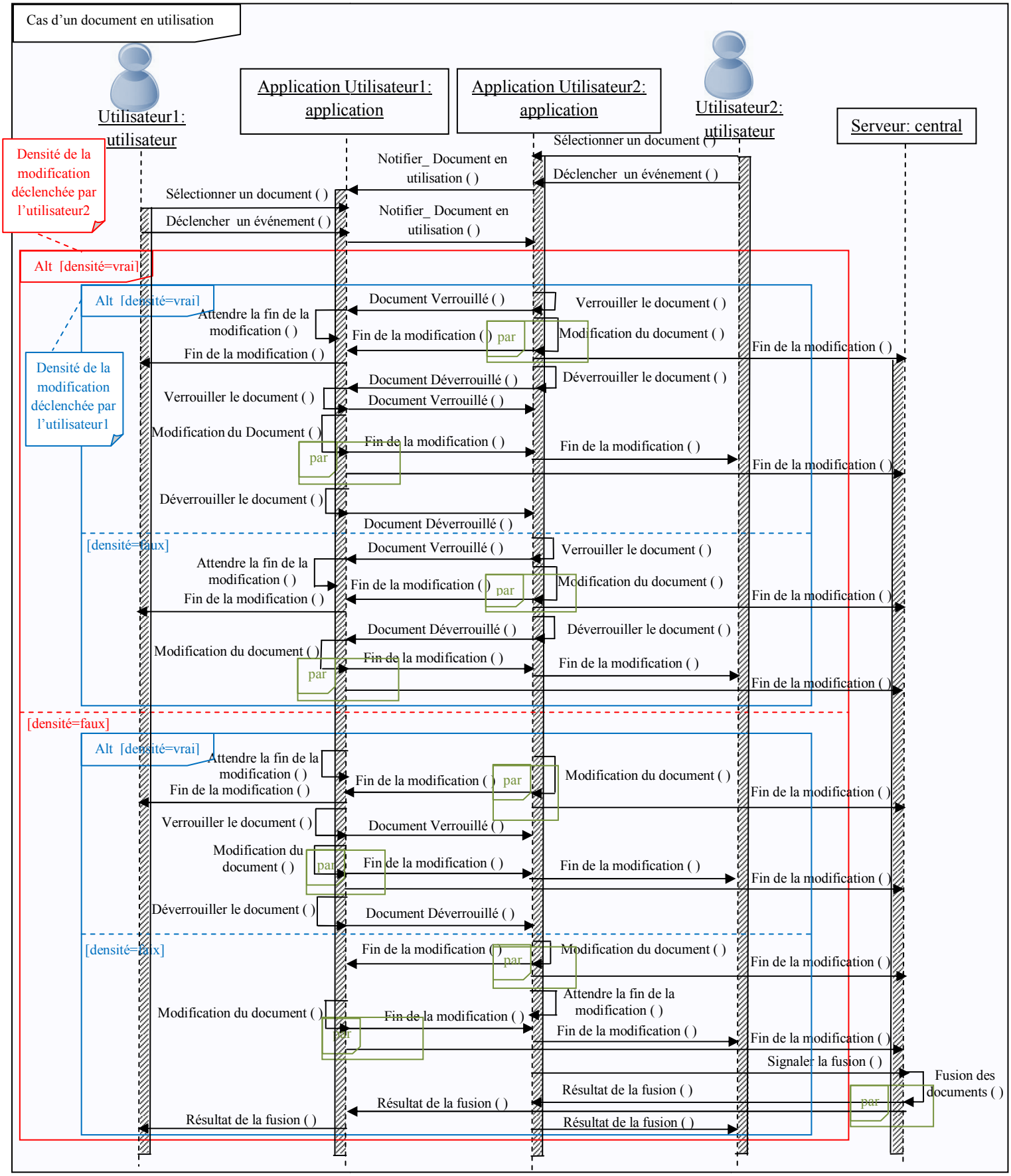


Figure 41. Diagramme de séquence de La stratégie du module Contrôle du Verrouillage (Cas d'un document en utilisation)

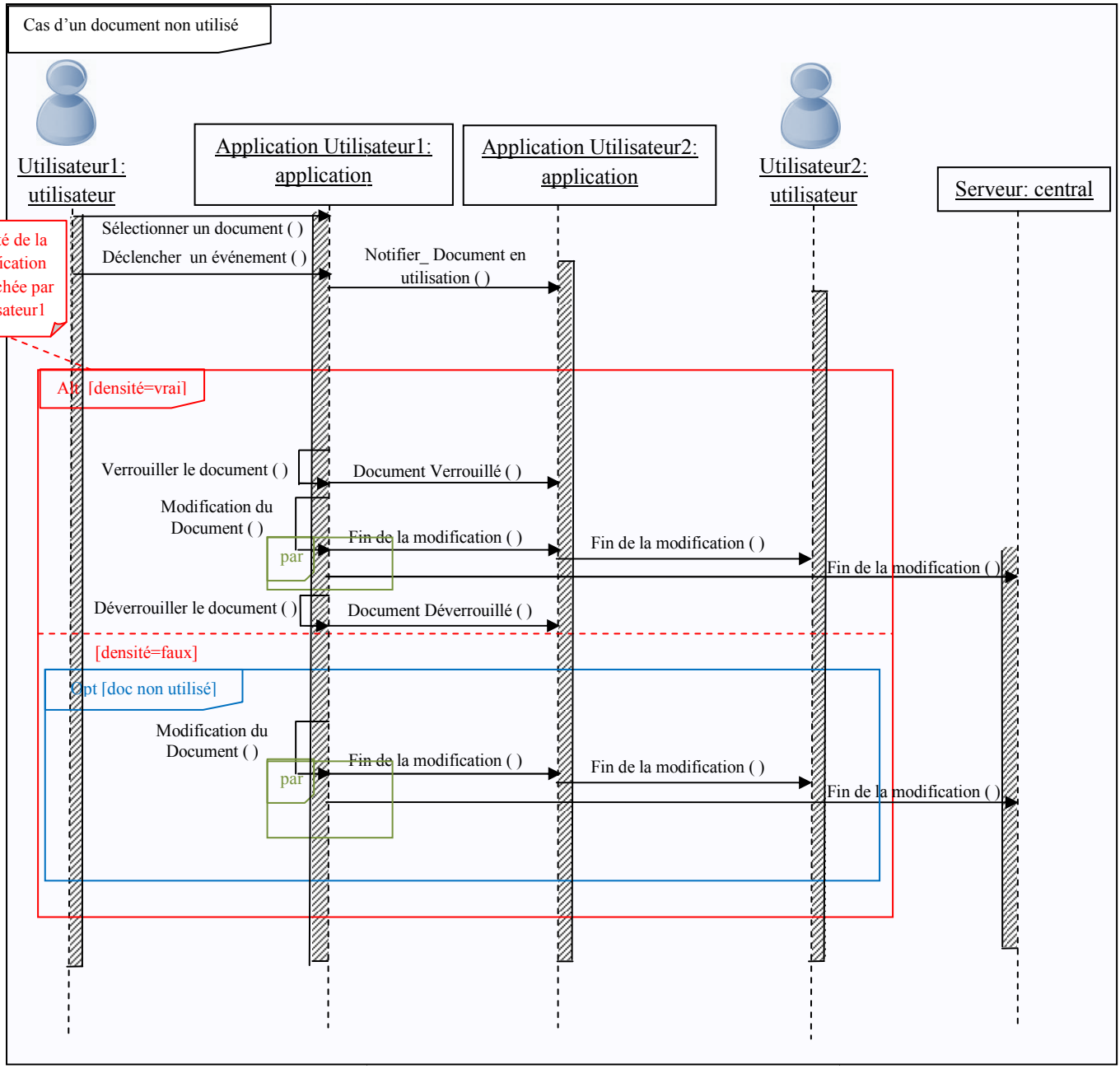


Figure 42. Diagramme de séquence de La stratégie du module Contrôle du Verrouillage (Cas d'un document qui n'est pas en utilisation)

Notons que, la fusion des documents est réalisée au niveau du serveur central.

Dans le modèle « Copier-Modifier-Fusionner » des systèmes de gestion de versions de codes sources, les utilisateurs contactent le dépôt du projet, et créent une *copie de travail* personnelle, qui est une réplication locale du dépôt. Les utilisateurs peuvent ainsi travailler en parallèle, en modifiant directement et uniquement leur copie privée. Ensuite, les copies privées sont fusionnées en une nouvelle version finale. Le système de gestion de versions permet ensuite d'assister les utilisateurs dans la fusion des documents, mais le contrôle final

est laissé à l'humain, qui est alors responsable des modifications et de leur intégrité. Il doit donc vérifier que tout se passe correctement.

Ce modèle n'est pas adapté tel qu'il est, parce qu'il fait appel au contrôle humain. Pour cela nous avons jugé utile d'utiliser les règles ECA dans la gestion de versions des documents et l'extension du modèle « Verrouiller-Modifier-Déverrouiller » afin de rendre notre système automatique au maximum.

Les règles ECA exprimant le fonctionnement du module *Contrôle du Verrouillage* sont les suivantes :

```
Règle ECA 6 : Evénement : sélection du document ; modification de document
                          déclenchée
              Condition : Si densité=vrai
              Action : Activer le verrouillage

Règle ECA 7 : Evénement : sélection du document ; modification de document
                          déclenchée
              Condition : Si densité=faux
              Action : Ne pas Activer le verrouillage
```

3.2.2 Modification du Document

Le Module de *Modification du Document* est conçu pour permettre l'évaluation et le déclenchement de la règle ECA. Il permet également de revenir à une ancienne version sans déclencher une règle ECA avec le composant de *Retour en Arrière / Désactiver*. Le produit de ce module est un *Document XML Modifié*.

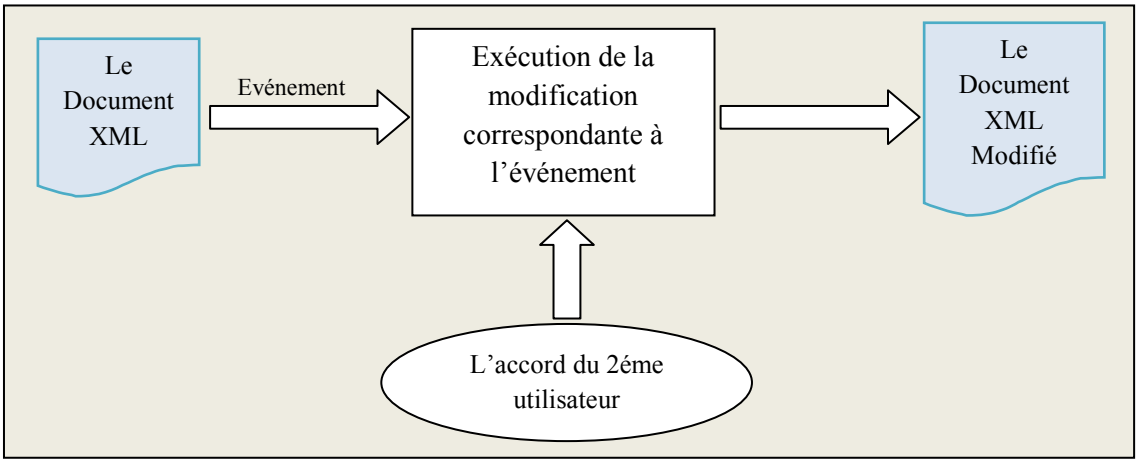


Figure 43. Les entrées et sorties du module Modification du document

Le module *Modification du Document* intervient lorsque les deux utilisateurs se mettent d'accord d'effectuer la modification au niveau d'un document donné.

La règle exécutée dépend de l'événement déclenché et du document sélectionné; car il est possible que les deux utilisateurs lancent l'exécution du module *Modification du Document* en même temps pour des documents différents, et dans ce cas, deux règles différentes vont être exécutées en même temps dans le système.

Pour un document donné le module *Modification du Document* appelle la règle ECA correspondante. Une fois l'événement détecté, le module évalue la condition de la règle, si cette dernière est satisfaite alors il exécute l'ensemble des actions correspondantes. La détection des événements est basée sur le mécanisme des réseaux de Petri synchronisés [HOC 09].

Ce module se charge de contrôler et de suivre le déclenchement d'une règle. Un événement peut survenir une seule fois. Un événement déclenché par un utilisateur est désactivé dés que la modification est acceptée, et par conséquent la règle ECA correspondante à cet événement est désactivée.

Afin d'éviter la redondance, les actions exécutées sur un document suite à un événement particulier ne peuvent pas se reproduire sur le document. Par exemple, on ne peut pas insérer une même donnée plus d'une fois.

3.2.2.1 Retour en Arrière

Le *Retour en Arrière* est un composant du module *Modification du Document* qui remplace la dernière version d'un document particulier par une autre version antérieure sélectionnée. Ce composant permet aux utilisateurs d'annuler toutes les modifications appliquées à la dernière version disponible dans le système.

Le travail réalisé par le composant *Retour en Arrière* est important et indispensable dans le système puisqu'il donne aux utilisateurs la possibilité de récupérer les anciens documents en cas d'erreurs ou en cas d'une mauvaise communication entre ces utilisateurs.

Le schéma suivant représente un exemple de retour en arrière appliqué dans la version 1.1.1 du document « Doc » qui devient par la suite la dernière version du document « Doc ». Cette opération ignore les modifications établies dans la version 1.1.2 du document.

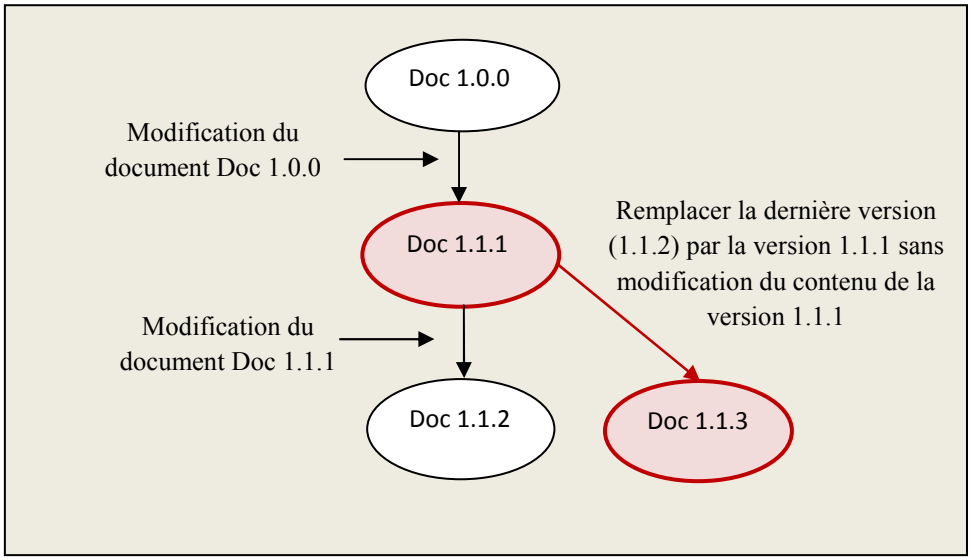


Figure 44. Exemple de remplacement de la dernière version du document Doc

3.2.2.2 Désactiver

Ce composant permet de désactiver la modification d'un document sélectionné. Ainsi si un utilisateur effectue cette opération, le document désactivé ne sera plus disponible pour les

autres utilisateurs. Sa version sera la dernière dans la branche de l'arbre de version, et le document reste accessible seulement en lecture.

3.2.3 Gestionnaire des Versions de Document XML

Le *Gestionnaire des Versions de Document XML* a pour rôle la nomination des nouvelles versions ainsi que la création et la gestion de l'arbre de version. Ce module est basé sur les règles ECA, définies comme suit :

```
Règle ECA 8 : Evénement : Document original ; présence d'un document
                          modifié
              Condition : Si l'arbre de version existe
              Action : 1. Parcourir l'arbre de version
                       2. Affecter un numéro à la nouvelle version
                       3. Ajouter le nœud à l'arbre de version

Règle ECA 9 : Evénement : Document non original ; présence d'un document
                          modifié
              Condition : Si l'arbre de version n'existe pas
              Action : 1. Créer l'arbre de version
                       2. Affecter un numéro à la nouvelle version
                       3. Ajouter le nœud à l'arbre de version
```

L'arbre de version créé à ce niveau n'est pas similaire à celui effectué dans les systèmes de gestion de versions existants car ces derniers visent à représenter la chronologie du développement du code source du programme jusqu'à arriver à la dernière version et les numéros des versions sont liés aux branches correspondantes, alors que l'arbre de version pour notre modèle vise à représenter la chronologie des modifications des utilisateurs et la numérotation des versions est liée aux nombres des modifications effectuées et à l'utilisateur responsable.

Le module *Gestionnaire des Versions du Document XML* affecte des numéros aux nouvelles versions des documents après leur création d'une manière automatique et sans l'intervention des utilisateurs. Afin de visualiser le maximum d'informations à partir du numéro de version, le module du *Gestionnaire des Versions de Document XML* adopte la numérotation selon le modèle «1.X.Y» tel que :

1 : Représente la version racine.

X : Le numéro de l'utilisateur qui a provoqué la création de cette nouvelle version (Si X=1 alors c'est l'utilisateur 1 qui a déclenché la règle ECA de la modification du document, et Si X=2 alors c'est l'utilisateur 2 qui a déclenché la règle ECA. Mais si la nouvelle version est le résultat de la fusion de deux documents résultants de l'utilisateur 1 et l'utilisateur 2 alors X=0)

Y : Le numéro de la version modifiée; il est incrémenté d'une unité après chaque création d'une nouvelle version.

Remarque : Un document original signifie que sa version (1.0.0) n'a pas subie des modifications.

L'arbre de version est stocké dans la *Base XML Centrale* et la *Base XML utilisateur*. L'arbre de version est la représentation graphique de la *Base XML Centrale*. Il illustre le

document qui a subi des modifications et de quelle version il résulte. Il donne ainsi la possibilité de garder un historique sur toutes les modifications établies dans le système.

Comme tout système de gestion de version, la création d'une nouvelle version implique la création d'un rapport de modification. Ce rapport contient toutes les informations concernant la modification : le nom du document qui a été modifié ainsi que le numéro de version, le nom de l'utilisateur qui a effectué la modification, la date de la création de la nouvelle version, le commentaire sur les opérations réalisées au niveau du document XML, et enfin le degré de modification (densité de la règle ECA).

Exemple du rapport de modification du document Liste produit 1.1.1.

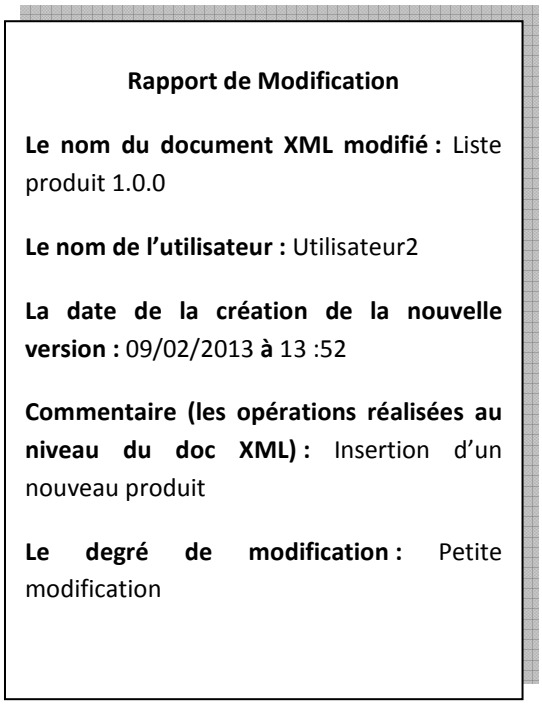


Figure 45. Exemple d'un rapport de modification

Remarque : Le numéro de l'utilisateur est attribué par l'application utilisateur.

La version 1.0.0 d'un document est le numéro de la première version d'un document, c'est la version initiale.

Pour gagner de l'espace mémoire, le système associe un arbre de version pour un document particulier seulement si la version 1.0.0 originale de ce dernier subit des modifications au moins une fois.

Les tâches du module *Gestionnaire des Versions du Document XML* sont résumées dans figure 46.

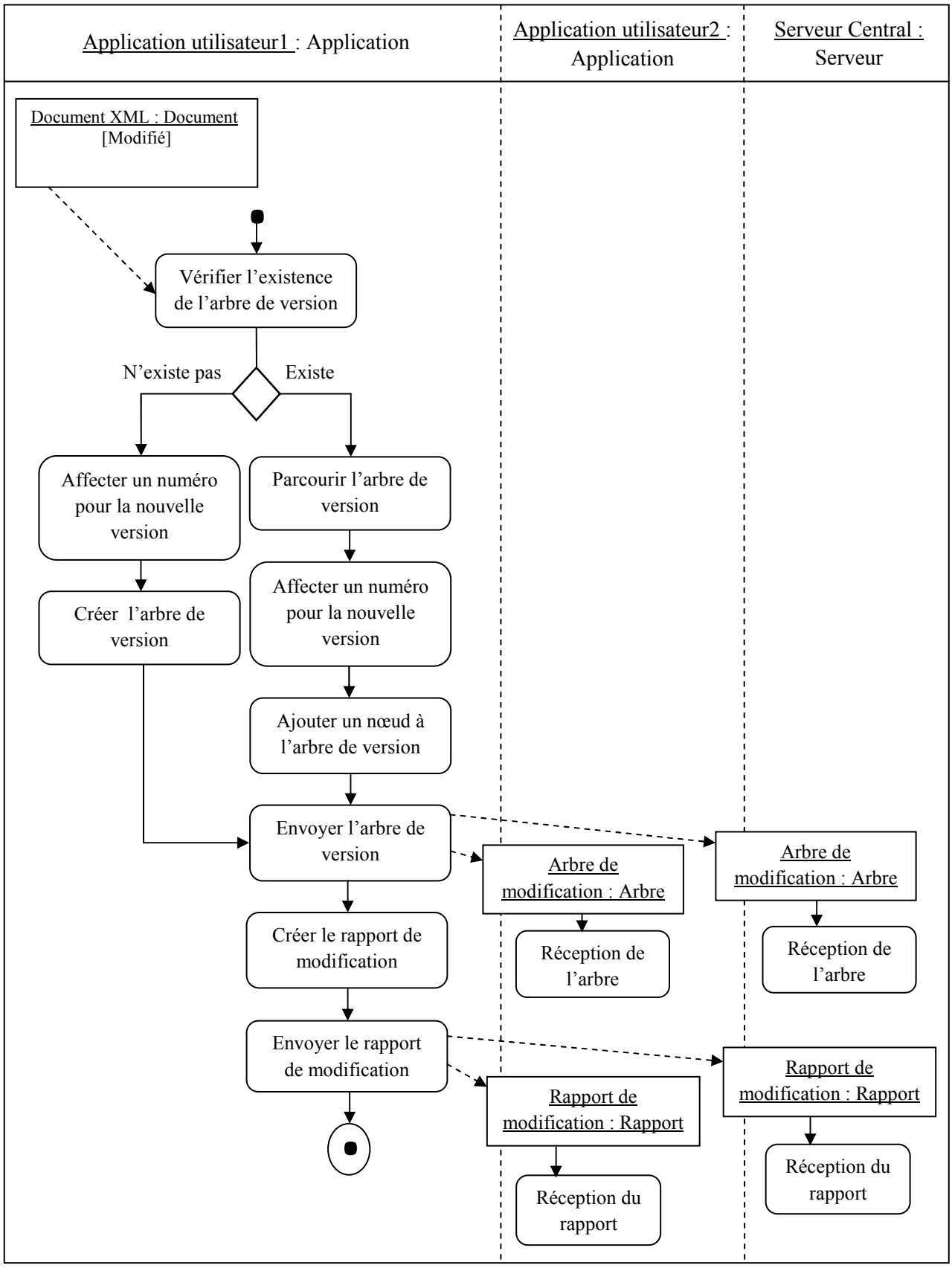


Figure 46. Diagramme d'activité du Gestionnaire des versions du document XML

Exemple 1:

Après la modification du document Doc 1.0.0 par l'utilisateur 1 (première modification) et la création de la copie, le module *Gestionnaire des Versions du Document XML* considère l'ancienne version du document Doc 1.0.0 comme nœud racine de l'arbre de version, et la nouvelle version du document Doc 1.1.1 comme fils du nœud Doc 1.0.0 (figure 47).

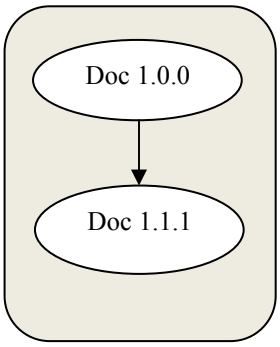


Figure 47. Exemple de l'arbre de version selon scénario 1

Exemple 2:

Cas d'insertion du nœud Doc 1.1.4 dans l'arbre de version correspondant au document Doc à partir du nœud Doc 1.0.0.

Le document Doc 1.0.0 est sélectionné pour recevoir des modifications par l'utilisateur, ensuite le système déclenche la création du document Doc 1.1.4 et insère le nom de cette nouvelle version dans l'arbre de version à partir du nœud du document modifié. Le système donne la possibilité d'apporter des modifications à des documents de version récente ou ancienne selon le choix des utilisateurs.

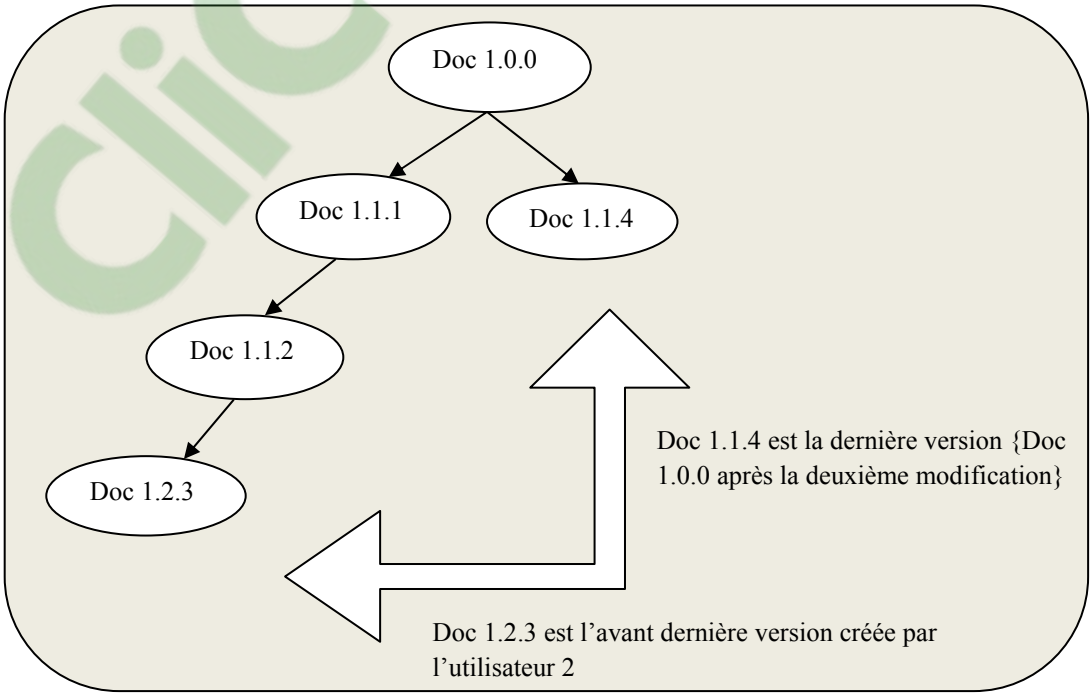


Figure 48. Exemple de l'arbre de version selon scénario 2

3.2.4 Gestion de la Nouvelle Version du Document

Une fois la nouvelle version du document créée, le module de *Gestion de la Nouvelle Version du Document* fait appel à ses deux composants *Propagation* et *Déverrouillage*. Ce module contient deux principales actions qui se produisent en même temps dans le but d'intégrer et de propager (distribuer) la nouvelle version du document dans les *Bases XML Utilisateurs* et la *Base XML Centrale*, de déverrouiller l'ancienne version du document et de supprimer l'ancienne version des *Bases XML Utilisateur*.

3.2.4.1 Déverrouillage

Ce composant intervient seulement dans le cas où le composant *Contrôle du Verrouillage* du module *Gestion du Document* a procédé au verrouillage du document sélectionné. Si la valeur de la densité de la modification déclenchée est égale à « vrai », le système fait appel au composant *Déverrouillage*.

Après avoir modifié le document sélectionné, le composant *Déverrouillage* débloque l'accès en écriture au document modifié par l'un des utilisateurs, permettant ainsi aux autres utilisateurs l'accès à la dernière version du document.

3.2.4.2 Propagation

Le rôle de ce composant est d'assurer la propagation de la nouvelle version du document. Après avoir créé la nouvelle version du document sélectionné, le composant *Propagation* distribue cette version aux utilisateurs en l'insérant dans le répertoire du document correspondant des *Bases XML Utilisateurs* ensuite il supprime l'ancienne version des *Base XML Utilisateurs*. En même temps ce composant ajoute la nouvelle version dans le répertoire du document correspondant en plus des versions qui coexistent au niveau de ce répertoire. Ce module veille à ce que l'ensemble de tous les utilisateurs, en plus du serveur central, aient constamment la bonne dernière version des documents et que leurs contenus doivent être identiques pour tous.

3.2.5 Le Serveur Central

L'application *Serveur Central* dispose de la *Base XML Centrale* qui regroupe l'ensemble de toutes les versions des documents existants dans l'ensemble des *Base XML Utilisateurs* associées avec leurs rapports de modifications ainsi que l'arbre de version correspondant à chaque document.

En plus de cette fonctionnalité, le *Serveur Central* dispose du composant *Fusion* dont le rôle principal est de fusionner les documents qui résultent d'une modification à partir d'un même document par les deux utilisateurs (figure 49).

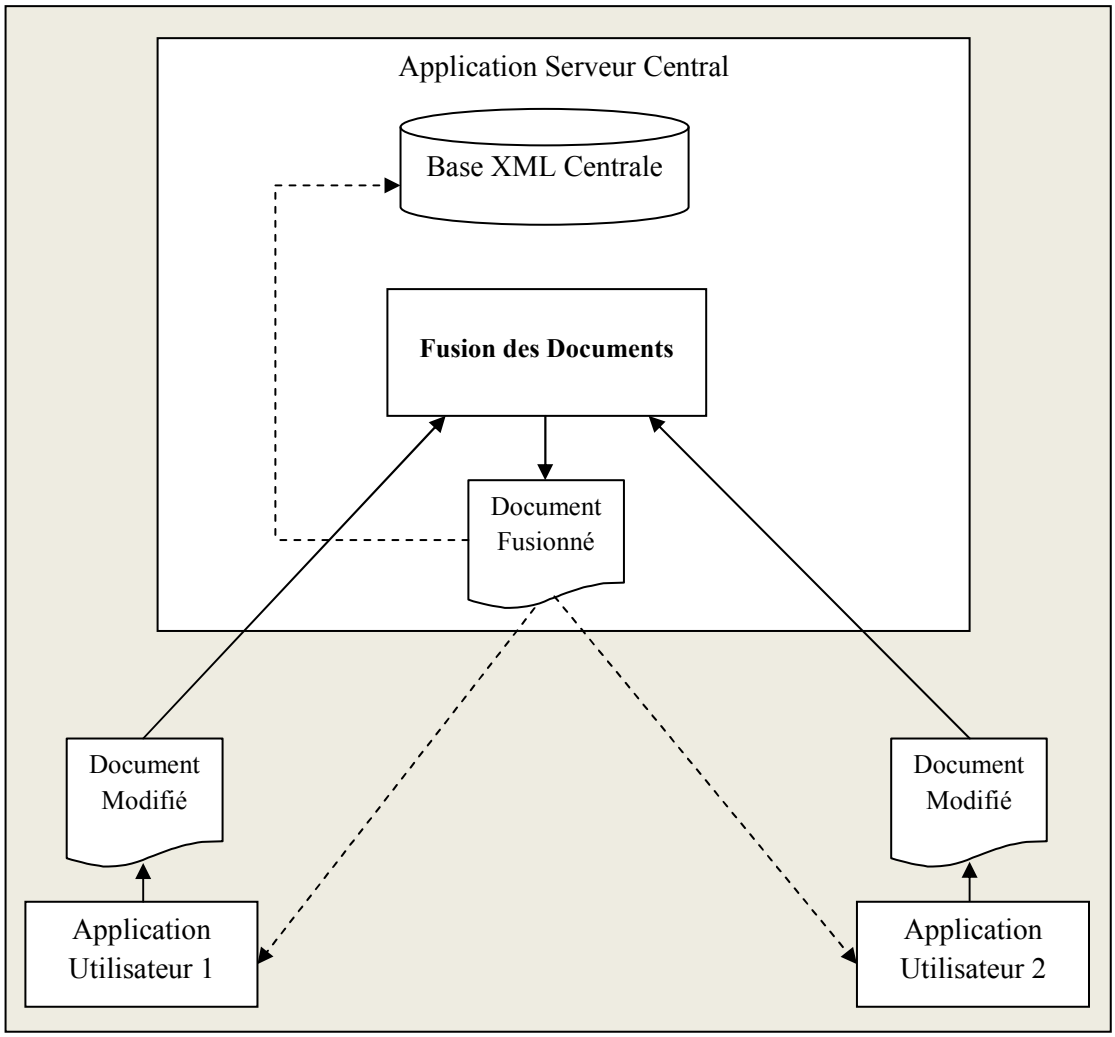


Figure 49. La fusion dans le Serveur Central

La fusion est réalisée au niveau de l'application *Serveur Central* afin de renforcer l'activité du système, puisque la fusion est contrôlée par le composant *Contrôle du Verrouillage* du module *Gestion du Document XML* qui réduit le nombre de fusion. Les résultats des modifications appliquées en parallèle par les utilisateurs à un document en commun ne sont pas toujours fusionnés. Lorsque la valeur de la densité d'une modification est égale à « vrai », le composant *Contrôle du Verrouillage* impose le verrouillage du document et réduit le nombre de fusion dans le *Serveur Central*.

Les fusions autorisées évitent les conflits grâce à l'information « densité » de la règle ECA de la modification du document, contrairement au système de gestion de versions des codes sources qui procède à la fusion au niveau d'un des utilisateurs concurrents.

4. Etude Comparative

L'objectif de cette étude est d'analyser le fonctionnement des solutions technologiques proposées dans les systèmes de gestion de versions existants tels que les modèles «Verrouiller-Modifier-Déverrouiller» et «Copier-Modifier-Fusionner» et de notre modèle proposé.

Les modèles « Verrouiller-Modifier-Déverrouiller» et «Copier-Modifier-Fusionner» sont des stratégies conçues pour gérer l'accès concurrent aux différents documents en commun et pour permettre aux utilisateurs des systèmes de gestion de versions le travail collaboratif tout en autorisant le partage de ces documents.

Cependant, Les modèles « Verrouiller-Modifier-Déverrouiller» et «Copier-Modifier-Fusionner», engendrent respectivement aux systèmes qui les utilisent des problèmes de blocage et de conflit. Le verrouillage imposé dans le modèle « Verrouiller-Modifier-Déverrouiller» cause un retard et une perte de temps inutile lorsque l'utilisateur ignore le déverrouillage du document ou quand il manipule une partie de document indépendante de celle des autres utilisateurs concurrents. Quant à la fusion, dans le modèle «Copier-Modifier-Fusionner», elle peut causer une perte de temps lorsque les utilisateurs procèdent à la résolution manuelle des conflits.

Afin d'éviter au maximum ces problèmes tout en gardant les avantage apportés par les modèles « Verrouiller-Modifier-Déverrouiller» et «Copier-Modifier-Fusionner» pour les web services, nous avons proposé un nouveau modèle ayant les points forts suivants :

1. Combiner ces deux modèles suivant le type de la modification d'un document partagé et l'avis des utilisateurs sur la modification; ce qui minimise la perte de temps.
2. Tirer d'avantage profit de l'utilisation des systèmes actifs; pour cela nous avons rajouté des fonctionnalités actives, à certains composants.
3. Etre exploité pour un environnement web service.
4. Enfin, concevoir un modèle indépendant du système global afin d'autoriser et d'accepter les améliorations.

5. Conclusion

L'utilité des systèmes de gestion de versions s'impose dans l'accès concurrentiel par plusieurs utilisateurs, le suivi de l'historique, la visualisation des différences et dans le retour en arrière à des anciennes versions. Ainsi l'intérêt de garder plusieurs versions d'un document XML dans un lieu central, est de faciliter l'accès et le retour en arrière à une version antérieure sans alourdir les utilisateurs par l'ensemble de toutes les versions disponibles des documents, autoriser la correction et la rendre toujours possible, et de garder l'historique de toutes les opérations (listes des modifications) qui est indispensable pour le travail en équipe.

Le modèle proposé combine la technique des modèles « Verrouiller-Modifier-Déverrouiller » et « Copier-Modifier-Fusionner » tout en évitant les points négatifs de ces deux modèles. Et l'aspect essentiel de notre approche est l'utilisation des règles actives ECA, qui contribuent aux modifications apportées aux documents, et à la gestion de la numérotation et de la propagation des versions de documents.

Nous avons proposé, ainsi une architecture de référence qui montre le concept et le fonctionnement du système de gestion de versions du modèle proposé entre deux utilisateurs et un serveur central, tout en représentant les différents échanges de données entre ces utilisateurs et ce serveur central.

1. Introduction

L'implémentation du modèle de gestion de versions proposé nécessite des outils de développement capables de répondre aux exigences et aux opérations mentionnées dans le chapitre 3. Pour ce faire nous avons opté pour un environnement Orienté Objet utilisant le langage Java, vu les opportunités apportées à l'implémentation d'un système sur un monoposte.

L'objectif de la programmation orientée objet est de concevoir l'organisation de grands projets informatiques autour d'entités précisément structurées, d'améliorer la sûreté des logiciels, et de simplifier la réutilisation de code [DEL 08] [PEC 06]. Parmi les langages de programmation orienté objet il y a le langage Java.

Le langage Java, développé par Sun Microsystems, est très performant et adopté par la majorité des fournisseurs. Ses caractéristiques intégrées de sécurité assurent les programmeurs et les utilisateurs des applications. De plus, Java incorpore des fonctionnalités qui facilitent grandement certaines tâches de programmation avancée comme la gestion des réseaux, la connectivité des bases de données ou le développement des applications multitâches. Il possède selon SUN les qualités suivantes [DOU 02] [HOR 04] [HOR 08] [HEB 11] :

- Interprété : Le code source est compilé en pseudo code puis exécuté par un interpréteur JVM (Java Virtual Machine). Java est indépendant de toute plateforme il n'a pas de compilation spécifique pour chaque plateforme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une JVM. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du byte code. Le processus de développement peut se révéler plus rapide et exploratoire.
- Simple : Java est fortement typé, toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données.
- Java assure la gestion de la mémoire : L'allocation de la mémoire pour un objet est automatique dès sa création ensuite il récupère automatiquement la mémoire inutilisée grâce au garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.
- Sécurité : Java a été conçu pour être exploité dans des environnements serveurs et distribués. Un programme Java ne menace pas le système d'exploitation.
- Java est économe : le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.
- Multithread : Il permet l'utilisation de threads qui sont des unités d'exécution isolées. La JVM, elle même utilise plusieurs threads. Les avantages du multithread sont une meilleure interréactivité et un meilleur comportement en temps réel.
- Distribué : Java possède une importante bibliothèque de routines permettant de gérer les protocoles TCP/IP tels que HTTP et FTP. Les applications Java peuvent charger et accéder à des programme sur Internet via des URL avec la même facilité qu'elles accèdent à un fichier local sur le système.
- Fiabilité : Java a été conçu pour que les programmes qu'ils utilisent soient fiables sous différents aspects. Sa conception encourage le programmeur à traquer préventivement les éventuels problèmes, à lancer des vérifications dynamiques en cours d'exécution et à éliminer les situations génératrices d'erreurs... La seule et unique grosse différence

entre C++ et Java réside dans le fait que ce dernier intègre un modèle de pointeur qui écarte les risques d'écrasement de la mémoire et d'endommagement des données.

- Orienté objet : La conception orientée objet est une technique de programmation qui se concentre sur les données (les objets) et sur les interfaces avec ces objets.
- Architecture neutre : Le compilateur génère un format de fichier objet dont l'architecture est neutre, le code compilé est exécutable sur de nombreux processeurs, à partir du moment où le système d'exécution de Java est présent. Pour ce faire, le compilateur Java génère des instructions en bytecode qui n'ont de lien avec aucune architecture particulière. Au contraire, ces instructions ont été conçues pour être à la fois faciles à interpréter et à traduire en code natif.
- Portable : Java ne dispose pas des aspects de dépendance de la mise en œuvre dans la spécification. Les tailles des types de données primaires sont spécifiées, ainsi que le comportement arithmétique qui leur est applicable.
- Performances élevées : En général, les performances des bytecodes interprétés sont tout à fait suffisantes, il existe toutefois des situations dans lesquelles des performances plus élevées sont nécessaires. Les bytecodes peuvent être traduits en code machine pour l'unité centrale destinée à accueillir l'application.

Pour l'implémentation du système et ses différentes fonctions ainsi que les documents de travail, nous avons exploité l'outil MyEclipse.

2. Les Outils de développement

Le système de gestion de versions est développé par l'environnement de programmation MyEclipse, tous les documents versionnés sont sous forme XML, leurs créations et modifications sont effectuées par le JDOM. L'arbre de version qui visualise l'historique d'un document est développé par le JTREE [HEB 11], son enregistrement est établi suivant le principe de la sérialisation [HOR 04] et l'accès à l'arbre est établi suivant le principe de la désérialisation. Les différentes applications du système communiquent entre elles et propagent leurs travaux par le bai des Sockets.

2.1 MyEclipse

MyEclipse [GEN 03] est un environnement de programmation orienté objet permettant de développer des applications 32 bits en vue de leur déploiement sous Windows, Linux, et Mac OS. Il propose un ensemble d'outils de conception pour le développement rapide d'application. Cet outil est choisi pour les raisons suivantes :

- Une solution complète : MyEclipse est l'IDE Java EE « J2EE » le plus complet de la plateforme open source d'éclipse.
- MyEclipse est peu coûteux.
- MyEclipse incorpore les technologies les plus innovatrices d'aujourd'hui pour fournir un environnement de développement pour J2EE WEB, XML, UML et les bases de données, ainsi qu'une grande sélection de connecteurs de serveur d'application pour améliorer le développement, le déploiement, le test, et la portabilité.

2.2 Le XML

Le XML (eXtensible Markup Language) est une norme du W3C depuis 1998 [HUN 01], [BRI 07]. Le XML est un méta-langage de structuration de données permettant de distinguer

et de décrire les informations de présentation et de structure d'un document via des balises. Il permet de créer des pages Internet sophistiquées, de structurer un document, de décrire des données. Le XML est devenu un standard incontournable d'échanges de données sur le Web. Il est supporté par toute l'industrie informatique, prenant une place majeure dans les applications Web, les systèmes d'informations, l'intégration de données et des applications, et le commerce électronique B2B. Ce standard est intégré aujourd'hui par toutes les nouvelles technologies de l'information. Il permet de fédérer les systèmes d'entreprises en facilitant la communication. Les web services apportent une réponse efficace aux besoins d'intégration interapplications tout en offrant un nouveau modèle d'architecture centré sur l'accès aux services, le modèle SOA (Service Oriented Architecture).

2.2.1 Les Parseurs XML

L'analyseur syntaxique « parseur » est un outil logiciel permettant de parcourir un document et d'en extraire des informations. Le Java dispose d'une API (Application Programming Interface) qui permet de travailler avec le XML. Cet outil permet de lire le fichier XML et le rend disponible à être programmé. Il peut le faire de deux manières différentes :

- En construisant une représentation du document en mémoire, sous forme d'un arbre (figure 50). Cet arbre est envoyé au programme à l'issue du parsing. C'est ainsi que fonctionnent les parseurs DOM « Document Object Model » ou JDOM (qui n'est pas un acronyme officiellement). Le DOM est une spécification W3C permettant ainsi de modéliser, parcourir et transformer un document XML. Le DOM permet de définir la structure logique des documents mais également de quelle manière les documents peuvent être parcourus et modifiés. Il permet de construire de nouveaux documents, de naviguer dans un document, créer, modifier et détruire des éléments. Le DOM est défini pour être indépendant du langage dans lequel il sera implémenté. Il n'est qu'une spécification qui, pour être utilisée, doit être implémentée par un éditeur tiers. Le DOM n'est donc pas spécifique à Java.

L'exemple suivant représente un document XML de la liste des clients.

```
<?xml version="1.0" encoding="UTF-8"?>
<clients>
  <client Num="C001">
    <nom>Abed</nom>
    <prenom>ahmed</prenom>
    <adresse>Arzew</adresse>
    <num_telephone>0611111111</num_telephone>
  </client>
  <client Num="C002">
    <nom>Badi</nom>
    <prenom>Bilal</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0722222222</num_telephone>
  </client>
</clients>
```

L'arbre DOM du document XML correspondant est représenté dans le schéma suivant :

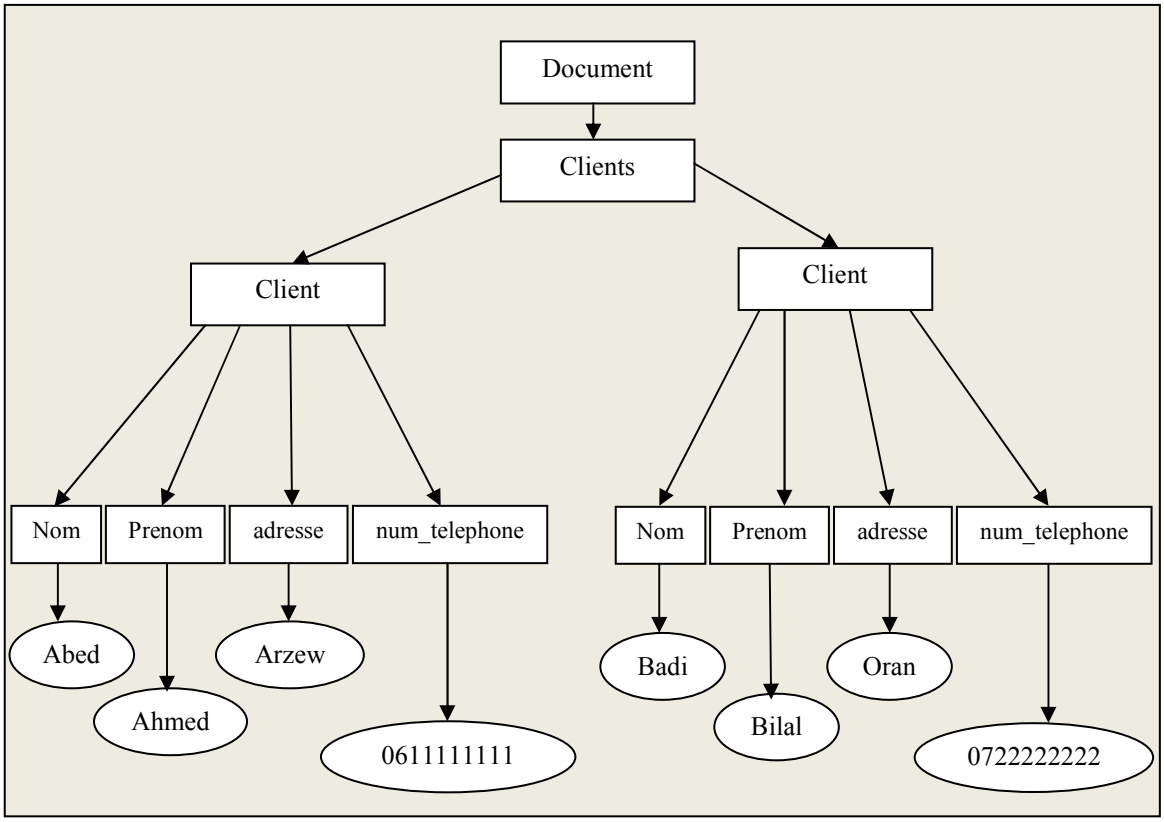


Figure 50. Représentation de l'arbre DOM

- La deuxième méthode est réalisée en appelant des méthodes du code Java lorsqu'il se produit des évènements donnés (comme l'ouverture ou la fermeture d'un élément). Ce type de parseur utilise des événements pour piloter le traitement d'un fichier XML. C'est le principe des parseurs SAX « Simple API for XML Parsing ». L'API SAX définit des triggers qui se déclenchent sur certaines balises. Le SAX est adapté aux applications qui extraient de l'information d'un document [SCH 01].

2.2.2 Le JDOM

Le JDOM est une API du langage Java développé indépendamment de Sun Microsystems [FIS 09]. Il permet de manipuler des données XML plus simplement qu'avec les API classiques afin de représenter et manipuler un document XML de manière intuitive pour un développeur Java sans requérir une connaissance de XML. Par exemple, JDOM utilise des classes plutôt que des interfaces. Ainsi pour créer un nouvel élément, il faut simplement instancier une classe. Son utilisation est pratique pour tout développeur Java et repose sur les API XML de Sun. La facilité d'utilisation de JDOM lui permet d'être une API dont l'utilisation est assez répandue. Il est en vérité très laborieux de développer des applications complexes autour de XML avec DOM, qui n'a pas été développé spécifiquement pour Java [CYN 05] [HAR 02]. Le JDOM présente de grandes similitudes avec le DOM puisqu'il représente un document XML via une structure arborescente. Cependant, il se distingue parce que JDOM est spécifiquement conçu pour JAVA et que du point de vue développement JAVA, le JDOM intègre DOM et SAX car il peut employer les arbres de DOM et les événements de SAX pour concevoir les documents JDOM [TID 04]. Bien que JDOM possède moins de fonctionnalités que DOM; mais en contre partie il offre une plus grande facilité pour répondre aux cas les plus classiques d'utilisation.

Autrement dit JDOM n'est pas un parseur, il a d'ailleurs besoin d'un parseur externe de type SAX ou DOM pour analyser un document et créer la hiérarchie des objets relatifs à un document XML. L'utilisation d'un parseur de type SAX est recommandée car elle consomme moins de ressources que DOM pour cette opération. Par défaut, JDOM utilise le parseur défini via JAXP (Java API for XML Processing) [FLE 04].

Le JDOM propose les fonctionnalités suivantes:

- La création des documents XML
- L'encapsulation d'un document XML sous la forme d'objets Java de l'API
- L'exportation d'un document dans un fichier, un flux SAX ou un arbre DOM
- Le support de XSLT [TID 01] [LAP 06].
- Le support de XPath [TID 01].

Les points caractéristiques de l'API JDOM sont les suivants:

- Est développée spécifiquement en Java et pour Java en utilisant les fonctionnalités de Java au niveau syntaxique et sémantique (utilisation des collections de Java 2, de l'opérateur `new` pour instancier des éléments, redéfinition des méthodes `equals()`, `hashCode()`, `toString()`, implémentation des interfaces `Cloneable` et `Serializable`),
- Se veut intuitive et productive notamment grâce à des classes dédiées à chaque élément instancié via leur constructeur et l'utilisation de getter/setter. Exemple pour obtenir le texte d'un élément,
  DOM : `String content = element.getFirstChild().getValue();`
  JDOM : `String text = element.getText();`
- Se veut rapide et légère,
- Veut masquer la complexité de certains aspects de XML tout en respectant ses spécifications et,
- Doit permettre les interactions entre SAX et DOM. JDOM peut encapsuler un document XML dans une hiérarchie d'objets à partir d'un flux, d'un arbre DOM, ou d'événements SAX. Il est aussi capable d'exporter un document dans ces différents formats.

La raison de créer une nouvelle API pour manipuler des documents XML en Java alors que plusieurs standards existent déjà, est le fait que JDOM propose des réponses à certaines faiblesses de SAX et DOM.

Dans la création d'un document, le JDOM instancie un objet de type Document grâce à l'opérateur new sur un des constructeurs de la classe. Une instance de la classe Document peut aussi facilement être créée à partir d'un document XML existant en utilisant les classes `SAXBuilder` ou `DOMBuilder` du package `org.jdom.input`. Ensuite le JDOM encapsule un document XML dans une arborescence d'objets dont le point d'entrée est un objet de type `org.jdom.Document`.

La création du client C001 du document XML Liste des Clients via JDom est effectuée par la classe `creation_liste_client`, et le code de la classe est le suivant :

```
public class creation_liste_client {
// création de la racine XML
static Element racine = new Element("clients");
//création d'un nouveau Document JDOM basé sur la racine clients
static org.jdom.Document document = new Document(racine);

   public static void main(String[] args) {
   //On crée de nouvel Element client1 et on l'ajoute
   Element client01 = new Element("client");
   racine.addContent(client01);
   Attribute classe1 = new Attribute("Num","C001");
   client01.setAttribute(classe1);
   Element nom1 = new Element("nom");
   nom1.setText("Abed");
   client01.addContent(nom1);
   Element prenom1 = new Element("prenom");
   prenom1.setText("ahmed");
   client01.addContent(prenom1);
   Element adr1 = new Element("adresse");
   adr1.setText("Areew");
   client01.addContent(adr1);
   Element tel1 = new Element("num_telephone");
   tel1.setText("06111111111");
    client01.addContent(tel1);

  ……

   }

}
```

Et l'enregistrement du document est effectué comme suit :

```
static void enregistre(String fichier){

   try {
      // utilisation de l'affichage classique avec getPrettyFormat()
      XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
      // Créationde l'instance de FileOutputStream avec  le nom
      // du fichier comme argument pour effectuer une sérialisation.
     sortie.output(document, new FileOutputStream(fichier));
       }
   catch (java.io.IOException e){}
}
```

Le choix de l'utilisation de JDOM dans l'implémentation du modèle et non pas de DOM ou de SAX est effectué parce que l'utilisation de DOM prend une occupation mémoire élevée et le SAX n'accorde pas la possibilité de modifier les documents, d'accéder aléatoirement, ou de générer une sortie physique.

2.3 L'arbre JTree

Java propose un mécanisme général pour parcourir les arbres par le JTree [HOR 05], ce dernier est un arbre qui sert à représenter graphiquement une hiérarchie entre des éléments. En les représentant graphiquement, le JTree simplifie aussi l'utilisation de l'application pour l'utilisateur. Il consiste à associer à chaque collection un objet qu'on appelle une « énumération » ou un « itérateur ». Cet objet sera une instance d'une classe.

Le JTree est facile à créer et à gérer. La création d'un arbre se fait en deux étapes:

- La première étape est celle de la création de la structure de l'arbre. L'arbre est composé de nœud `DefaultMutableTreeNode`. Cet objet représente une feuille ou une branche de l'arbre (dépendamment si elle a des enfants). Chaque nœud possède un objet qui spécifie le texte affiché sur l'écran (La méthode `toString()` est utilisée pour obtenir le texte à afficher). On crée donc la structure de l'arbre à partir des nœuds, le premier nœud sert de racine, les autres sont considérés comme des enfants. La méthode `add(object)` est utilisée pour spécifier qu'un nœud est l'enfant d'un autre :

```
DefaultMutableTreeNode root = new
DefaultMutableTreeNode(objet,true);
root.add(enfant);
```

- La deuxième étape est la création de l'arbre. Une fois que la structure de l'arbre a été créée avec les nœuds, il faut les rajouter à l'arbre et les afficher:

```
// On met le nœud créer comme racine root
JTree arbre = new JTree(root);
panel.add(arbre);
```

Il y a sept autres classes utilisées pour manipuler un arbre JTree :

- `TreeModel` : contient les données figurant dans l'arbre.
- `TreeNode` : implémentation des noeuds et de la structure de l'arbre.
- `TreeSelectionModel` : contient le ou les noeuds sélectionnés.
- `TreePath` : cet objet contient un chemin (de la racine vers le sommet sélectionné par exemple).
- `TreeCellRenderer` : est appelé pour dessiner un nœud.
- `TreeCellEditor` : l'éditeur pour un nœud est éditable.
- `TreeUI` : look-and-feel.

2.4 La sérialisation

La sérialisation est un procédé introduit dans le JDK version 1.1 qui permet de rendre un objet persistant. La sérialisation est le processus de conversion d'un objet en un flux d'octets afin de le faire persister dans la mémoire, dans une base de données, dans un fichier, ou pour le transmettre à travers un réseau pour le créer dans une autre JVM [HEB 11]. Son objectif principal est d'enregistrer l'état d'un objet pour pouvoir le reconstituer si nécessaire. Le processus inverse est appelé désérialisation.

La sérialisation permet au développeur d'enregistrer l'état d'un objet et de le recréer si nécessaire, en fournissant un stockage d'objets ainsi qu'un échange de données. La sérialisation permet au développeur d'effectuer des actions comme l'envoi de l'objet à une application distante via un Web Service, le passage d'un objet d'un domaine à un autre, via un pare-feu sous forme de chaîne XML ou la conservation des informations de sécurité ou bien des informations spécifiques à l'utilisateur sur les applications.

Ainsi, il est inutile de créer un format particulier pour sauvegarder et relire un objet parce qu'avec la sérialisation, le format utilisé est indépendant du système d'exploitation. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour recréer l'objet. Et l'ajout d'un attribut à l'objet est automatiquement pris en compte lors de la sérialisation. Toutefois, la désérialisation de l'objet doit se faire avec la classe qui a été utilisée pour la sérialisation. Et la sérialisation peut s'appliquer facilement à tous les objets.

Pour sérialiser un objet, il faut définir un fichier avec la classe `FileOutputStream`, puis il faut instancier un objet de classe `ObjectOutputStream` en lui fournissant en paramètre le fichier, et le résultat de la sérialisation sera envoyé dans le fichier. Ensuite il faut appeler la méthode `writeObject()` en lui passant en paramètre l'objet à sérialiser et on appelle la méthode `flush()` pour vider le tampon dans le fichier et la méthode `close()` pour terminer l'opération. Lors de ces opérations une exception de type `IOException` peut être réclamée si un problème intervient avec le fichier.

La classe `ObjectOutputStream` contient aussi plusieurs méthodes qui permettent de sérialiser des types élémentaires et non des objets : `writeInt()`, `writeDouble()`, `writeFloat()`, ...

Nous faisons appel à la sérialisation dès que l'arbre de version est créé ou modifié et avant de l'envoyer aux utilisateurs et au serveur central. La sérialisation de l'arbre de version est réalisée par le code suivant :

```
FileOutputStream fichier = new FileOutputStream(
"C:\\Documents and Settings\\Benhamed\\Workspaces\\MyEclipse for Spring
8.6[2]\\Gestion de Version\\Utilisateur 2\\Liste des Fournisseurs\\arbre
version.ser");
ObjectOutputStream oos = new ObjectOutputStream(fichier);
// tree2 est l'arbre de version fournisseur
oos.writeObject(tree2);
oos.flush();
oos.close();
System.out.println("arbre version fournisseur sauvegardé");
```

Quand les utilisateurs ou le serveur central désirent modifier l'arbre de version, nous appliquons l'opération inverse de la sérialisation « la désérialisation » pour pouvoir récupérer les données de l'arbre de version nécessaires pour leurs traitements sur l'arbre. La désérialisation est réalisée par le code suivant :

```
FileInputStream fichier1 = new FileInputStream( "C:\\Documents and
Settings\\Benhamed\\Workspaces\\MyEclipse for Spring 8.6[2]\\Gestion de
Version\\Utilisateur 2\\Liste des Fournisseurs\\arbre version.ser");
ObjectInputStream ois = new ObjectInputStream(fichier1);
JTree tempTree = (JTree)ois.readObject();
tree2.setModel(tempTree.getModel());
DefaultTreeModel model2= (DefaultTreeModel)tree2.getModel();
DefaultMutableTreeNode root2   =
(DefaultMutableTreeNode)model2.getRoot();
root2.getRoot();
```

2.5 Les Sockets

Un socket est une notion très répandue en programmation réseau [FIS 09] [GOE 09]. Il s'agit en fait d'un point de communication entre un processus et un réseau. Un processus client et un processus serveur, lorsqu'ils communiquent, ils ouvrent donc chacun un socket. Les sockets sont une implémentation réseau de bas niveau, c'est le système d'exploitation qui alloue ces sockets sur demande d'une application. A chaque socket est associé un port de connexion. Ces numéros de port sont uniques sur un système donné, et une application peut en utiliser plusieurs (un serveur par exemple exploite un socket par client connecté).

Une connexion est identifiée de façon unique par la donnée de deux couples, une adresse IP et un numéro de port, un pour le client et un autre pour le serveur. Il est important de noter qu'un dialogue client/serveur n'a pas forcément lieu via un réseau, il peut être virtuel.

Java distingue les sockets clients des sockets serveurs en deux classes, respectivement `Socket` et `ServerSocket`, contenus dans le package `java.net`. L'API "sockets" est une bibliothèque de Classes de communication entre machines sur TCP/IP contenu dans le paquetage `java.net`. Et la classe Java.net.Socket implémente un socket client TCP ensuite le constructeur nécessitera uniquement l'URL du serveur ainsi que le numéro de port du socket d'écoute.

3. L'application Utilisateur

L'application utilisateur représente par le biais de son interface (figure 51), la copie de travail de l'utilisateur (figure 52). Elle permet d'afficher les documents avec leurs rapports de modification et les arbres de version correspondants en cas de modification; elle permet à l'utilisateur de pouvoir se connecter au serveur central afin d'accéder à des versions antérieurs de tous les documents existants dans la Base XML Centrale, ensuite à modifier le document sélectionné. L'application utilisateur retransmet au deuxième utilisateur et au serveur central les opérations établies par le premier utilisateur. Cette application sert d'interface aux utilisateurs.

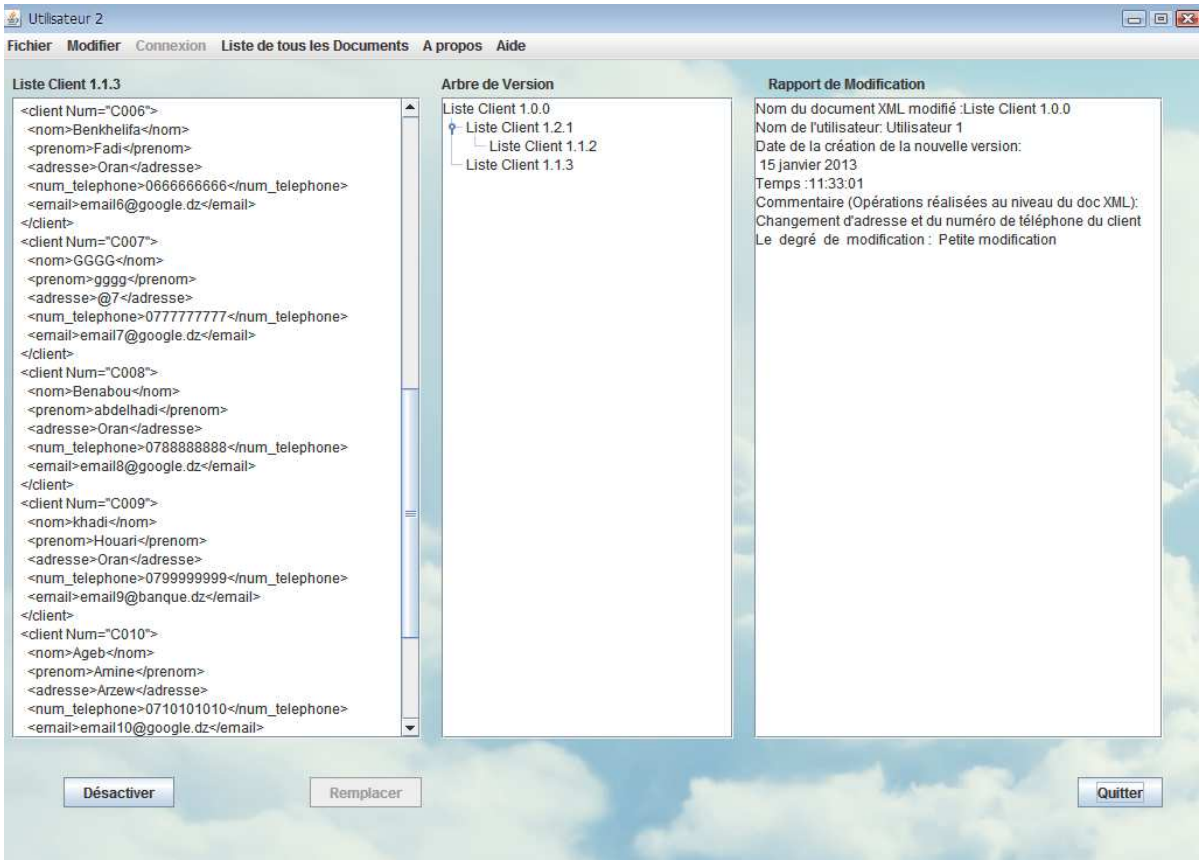


Figure 51. L'interface de l'application utilisateur

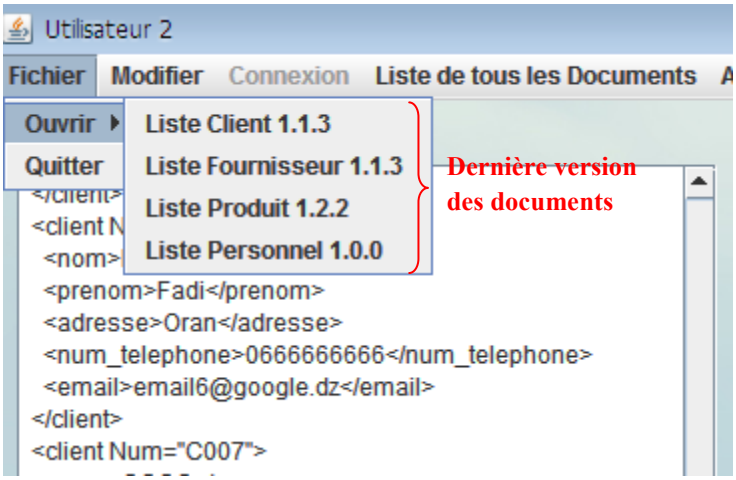


Figure 52. La liste des documents de la Base XML Utilisateur

3.1 L'accès à l'application utilisateur

Avant d'accéder à l'application utilisateur, l'utilisateur doit s'identifier en saisissant son nom et son mot de passe (figure 53). Ensuite ces derniers seront vérifiés par l'application, s'ils sont corrects l'application autorise l'utilisateur à accéder aux différentes fonctionnalités de l'application utilisateur.

La phase d'identification à l'application utilisateur est obligatoire et elle est effectuée avant chaque utilisation de l'application.

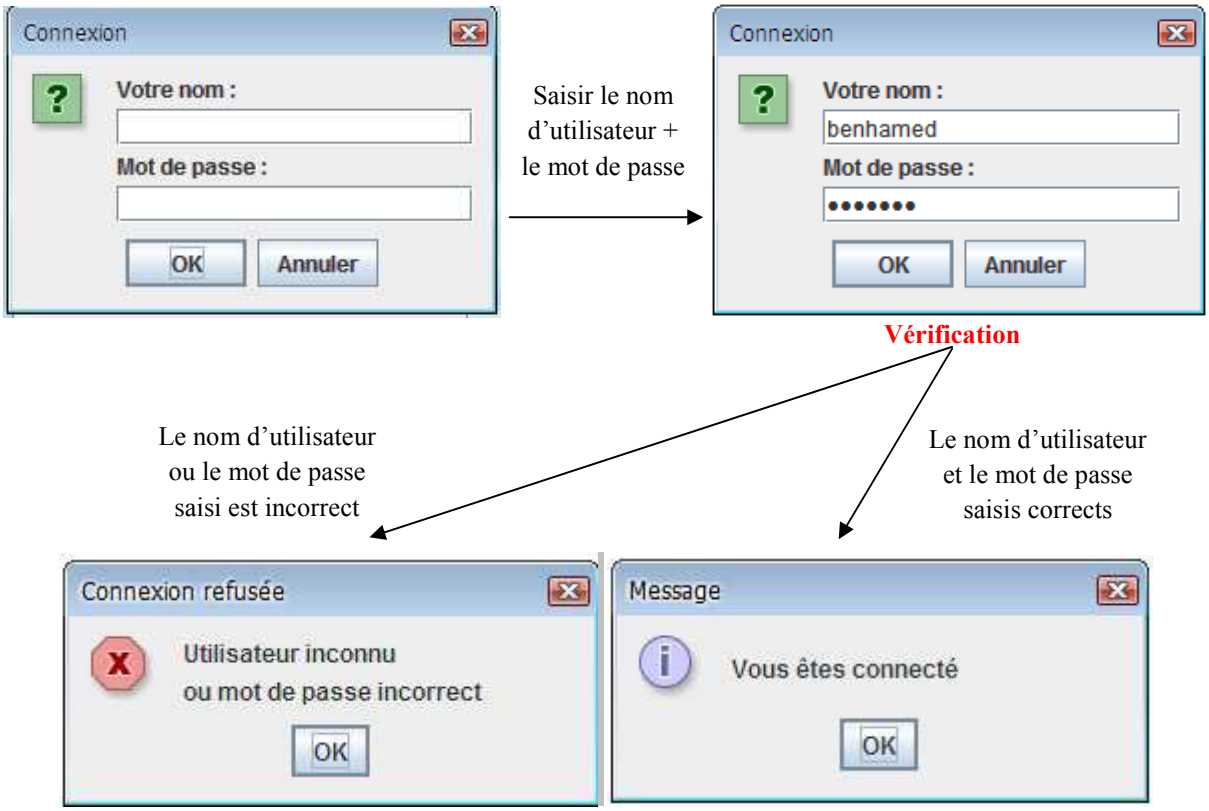


Figure 53. Les étapes de connexion à l'application utilisateur

Cette connexion permet à l'utilisateur, par la suite, d'accéder aux documents de la Base XML Centrale quand les documents de sa Base XML utilisateur sont remplacés par de nouvelles versions.

3.2 L'accès au document

Un utilisateur dispose dans son répertoire (Base XML utilisateur) uniquement des dernières versions des documents et l'accès à ces documents est établi par une simple sélection de la liste des documents dont il dispose (figure 52). Cependant il est possible d'accéder à une version antérieure d'un document par une connexion au serveur central (figure 54). L'utilisateur envoie sa demande au serveur central et ce dernier à son tour envoie le contenu du document sélectionné ainsi que le rapport de modification correspondant. Ces informations ne sont pas enregistrées au niveau de l'utilisateur mais elles sont reçues comme messages du serveur central.

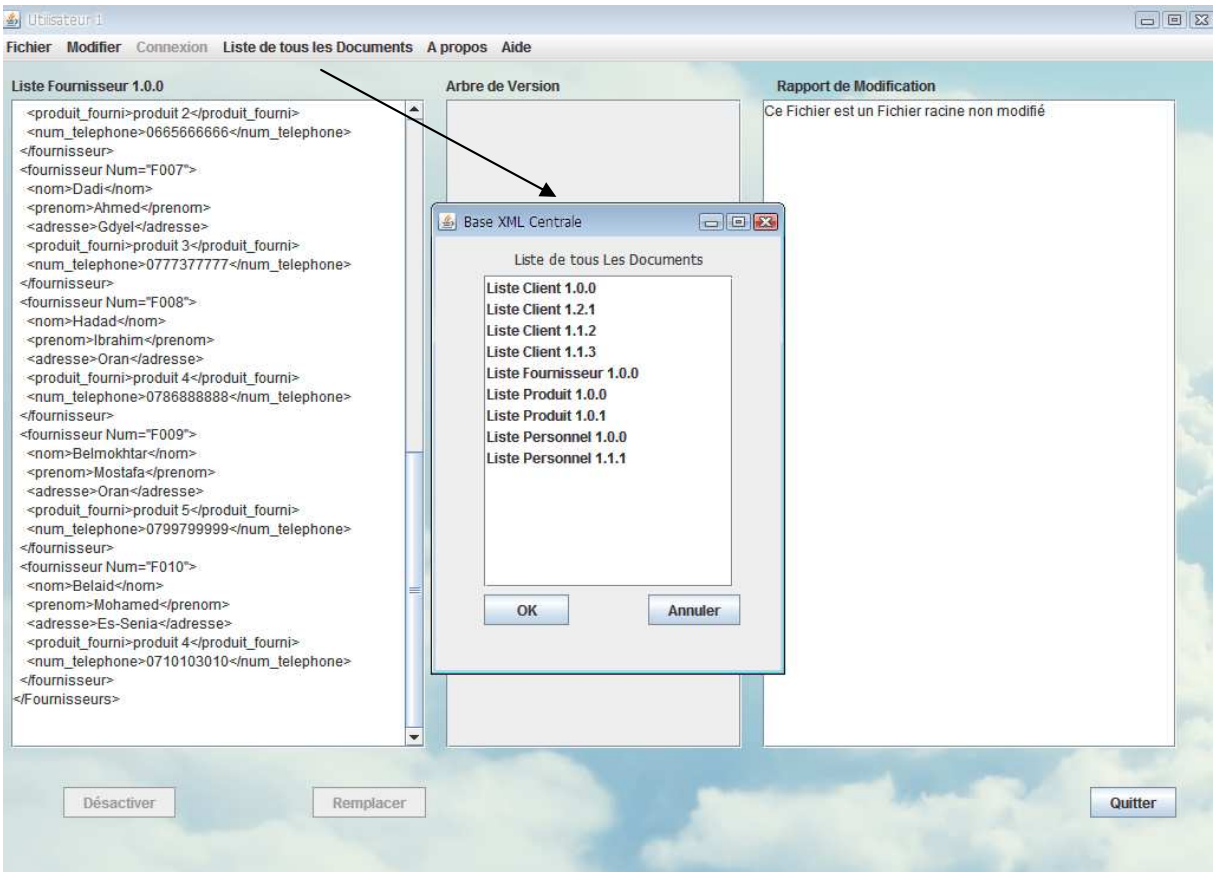


Figure 54. La Liste des documents de la Base XML Centrale

3.3 La modification du document

Afin de modifier un document particulier, l'utilisateur doit tout d'abord le sélectionner de la liste des documents de sa Base XML utilisateur ou de la liste de la Base XML Centrale (quand il s'agit d'une version antérieure), ensuite il doit sélectionner l'événement correspondant à la modification qu'il désire effectuer (figure 55).

Toutes les modifications sont appliquées après que le deuxième utilisateur ait émis son accord. Ensuite l'événement correspondant au déclenchement de ces modifications, sera désactivé de la liste des événements et dans les applications des deux utilisateurs.

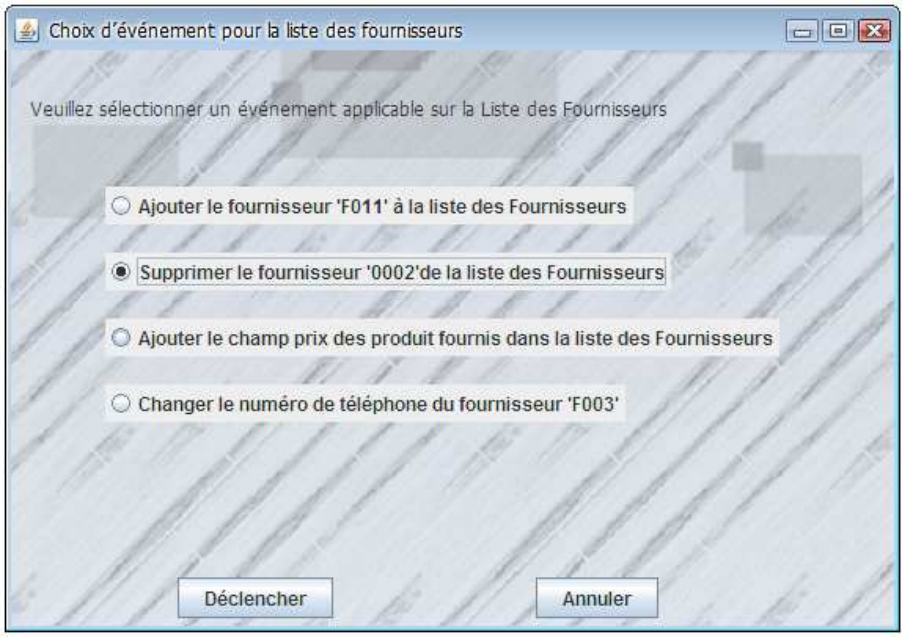


Figure 55. Liste des événements de la liste des fournisseurs

Après le déclenchement de l'événement par exemple, « suppression de l'employé F002 », le système exécute l'opération de modification comme suit :

```
private void supprFElement(String string)
{
  List listemploye = racine.getChildren();
  Iterator i = listemploye.iterator();
  //On parcourt la liste grâce à un iterator
 while(i.hasNext())
 { Element courant = (Element)i.next();
   if(courant.getAttributeValue("Num").equals("F002"))
    {
   courant.removeContent();
    }
 }
}
```

Cette modification n'est exécutée que si le deuxième utilisateur l'autorise.

3.3.1 La création de la nouvelle version

Le système récupère le résultat de la modification du document et l'enregistre à son niveau en lui attribuant un nouveau numéro de version qui succède à l'ancienne version, ensuite il supprime l'ancienne version. L'opération est réalisée comme suit :

```
// Récupérer le document
lireFichier(filePath2);
// Modifier le document
AjoutPElement("fournisseur");
// Supprimer l'ancienne version
File file = new File(filePath2);
boolean successc = filec.delete();
// incrémenter le numéro de version
l=l+1;
// enregistrer la nouvelle version
enregistre(filePath2);
```

Et la récupération du document est effectuée par la méthode suivante :

```
static void lireFichier(String fichier) throws Exception

      {
      SAXBuilder sxb = new SAXBuilder();
      document = sxb.build(new File(fichier));
      racine = document.getRootElement();
}
```

3.3.2 La Création du Rapport de Modification

Chaque document modifié lui est associé le rapport de modification correspondant. La création du rapport succède à la création de la nouvelle version du document. Le rapport de modification est stocké dans le répertoire du document correspondant (figure 56). Ce rapport possède le même numéro que la nouvelle version du document.

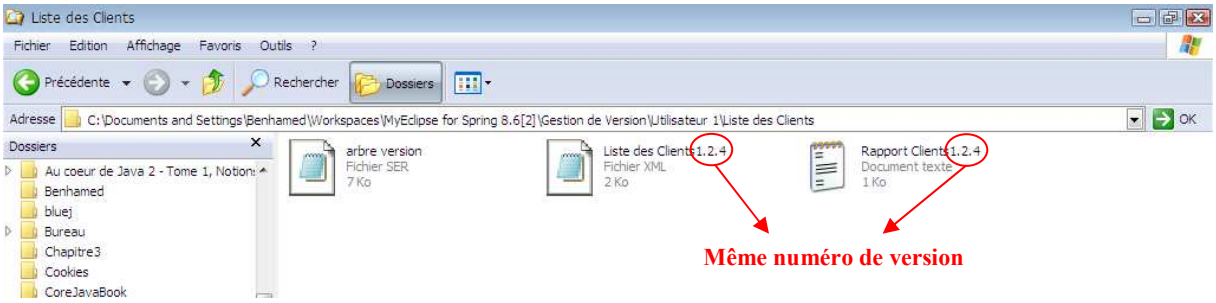


Figure 56. Le répertoire de la Liste des Clients

Le code source de la création du rapport de modification est le suivant :

```
…..
boolean success = file.delete();
File filec = new File(filePath2c);
PrintWriter pw;
pw =  new PrintWriter(new BufferedWriter(new FileWriter(filePath2c)));
pw.println("Nom du document XML modifié :"+ jLabel1.getText());
pw.println("Nom de l'utilisateur: Utilisateur 1");
pw.println("Date de la création de la nouvelle version:");
Date d=new Date();
SimpleDateFormat df=new SimpleDateFormat( " dd MMMM yyyy ");
pw.println(df.format(d));
Calendar dateDepart = Calendar.getInstance();
Locale locale = Locale.getDefault();
DateFormat formatHeure = DateFormat.getTimeInstance(DateFormat.MEDIUM,
locale);
pw.println("Temps :" + formatHeure.format(dateDepart.getTime()));
pw.println("Commentaire (Opérations réalisées au niveau du doc XML):"
);
pw.println("Insertion du nouveau numéro de téléphone du fournisseur
'F003'" );
pw.println("Degré de modification : " +  densite );
pw.close();
……
```

3.3.3 L'arbre de version

L'arbre de version est mis à jour dans les cas suivants :

- Modification d'un document.
- Remplacement d'un document par un autre d'une version précédente.
- Désactivation de la modification d'un document.

Cependant l'arbre de version n'existe pas pour un document qui n'a jamais été modifié. Pour cela, lors la première modification du document le système crée l'arbre de version via JTree.

3.3.3.1 La création de l'arbre de version

Pour que le système crée l'arbre de version, il doit en premier lieu vérifier qu'il n'existe pas. Dans l'affirmative, il entame la création de l'arbre en lui affectant le nom du document qui a subi des modifications comme nœud racine de cet arbre et le nom de la nouvelle version du document comme un nœud enfant de l'arbre. Le code de la création de l'arbre de version est le suivant :

```
if (arbrepr==false){
jButton3.setEnabled(true);
root3 = new DefaultMutableTreeNode(produitar,true);
model3 = new DefaultTreeModel(root3);
tree3 = new JTree(model3);
tree3.setEditable(true);
DefaultTreeCellRenderer renderer2 = new DefaultTreeCellRenderer();
renderer2.setOpenIcon(null);
renderer2.setClosedIcon(null);
renderer2.setLeafIcon(null);
tree3.setCellRenderer(renderer2);
produit="Liste Produit "+1+"."+1+"."+h;
produitar="Liste Produit "+1+"."+1+"."+h;
DefaultMutableTreeNode autre = new
DefaultMutableTreeNode(produitar,true);
root3.add(autre);
arbrepr=true;
FileOutputStream fichier = new FileOutputStream(
"C:\\Documents and Settings\\Benhamed\\Workspaces\\MyEclipse for Spring
8.6[2]\\Gestion de Version\\Utilisateur 1\\Liste des produits\\arbre
version.ser");
ObjectOutputStream oos = new ObjectOutputStream(fichier);
oos.writeObject(tree3);
oos.flush();
oos.close();
System.out.println("arbre version produit sauvegardé");
jScrollPane33.setViewportView(tree3);
}
```

Le résultat de ce code est le suivant :

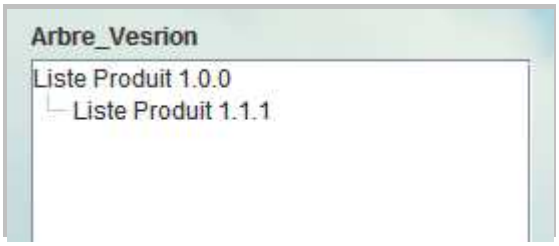


Figure 57. Création de l'arbre version

3.3.3.2 L'insertion d'un nœud dans l'arbre de version

L'insertion d'un nouveau nœud fils dans l'arbre de version se fait en quatre étapes :

1. La première étape consiste à désérialiser l'arbre pour pouvoir accéder au contenu de l'arbre en lecture et en écriture (cette étape est optionnelle quand l'utilisateur vient juste de le créer ou de le modifier).
2. L'étape qui suit consiste à parcourir l'arbre en profondeur et à tester tous les nœuds rencontrés par rapport au nom du document modifié.
3. Insérer le nouveau nœud de la nouvelle version dans le nœud qui vérifie le test de l'étape 2.
4. La dernière étape consiste à sérialiser la nouvelle version de l'arbre de version pour pouvoir l'envoyer au deuxième utilisateur et au serveur central.

D'où le code suivant :

```
else{
FileInputStream fichier1 = new FileInputStream( "C:\\Documents and
Settings\\Benhamed\\Workspaces\\MyEclipse for Spring 8.6[2]\\Gestion de
Version\\Utilisateur 1\\Liste des produits\\arbre version.ser");
ObjectInputStream ois = new ObjectInputStream(fichier1);
JTree tempTree = (JTree)ois.readObject();
tree3.setModel(tempTree.getModel());
DefaultTreeModel model3= (DefaultTreeModel)tree3.getModel();
DefaultMutableTreeNode root3   =
(DefaultMutableTreeNode)model3.getRoot();
root3.getRoot();
int row1 = 0;
 while (row1 < tree3.getRowCount()) {
 tree3.expandRow(row1);
 row1++;
  }
 int ee=1;
 Enumeration e = root3.breadthFirstEnumeration();
```

```
while (e.hasMoreElements()&& ee==1)
{
DefaultMutableTreeNode node = (DefaultMutableTreeNode)e.nextElement();
if (node.getUserObject().equals(produitar))
{    DefaultMutableTreeNode parent;
     parent = node;
     produit="Liste Produit "+1+"."+1+"."+h;
     produitar="Liste Produit "+1+"."+1+"."+h;
     DefaultMutableTreeNode newNode = new
DefaultMutableTreeNode(produitar);
     model3.insertNodeInto(newNode, parent,
     parent.getChildCount());
     activer=false; ee=2;
 }
if (node.getUserObject().equals(produitar+ " (Désactiver)")){
JOptionPane.showMessageDialog(null, "Le document est désactivé Veuillez
sélectionner un autre document");
activer=true;
jFrame12.setVisible(false);
ee=2;jButton3.setEnabled(false);
}
  }
int row = 0;
while (row < tree3.getRowCount()) {
tree3.expandRow(row);
row++;
}
FileOutputStream fichier = new FileOutputStream("C:\\Documents and
Settings\\Benhamed\\Workspaces\\MyEclipse for Spring 8.6[2]\\Gestion de
Version\\Utilisateur 1\\Liste des produits\\arbre version.ser");
ObjectOutputStream oos = new ObjectOutputStream(fichier);
oos.writeObject(tree3);
oos.flush();
oos.close();
System.out.println("arbre version produit sauvegardé");
jScrollPane33.setViewportView(tree3);
  }
```

Le résultat de l'ajout du nœud « Liste Produit 1.2.2 » est le suivant :

Figure 58. Insertion du nœud Liste Produit 1.2.2 dans l'arbre de version de la Liste Produit

3.4 Remplacer la dernière version d'un document

Cette opération de remplacement de la dernière version est exécutée suite à la demande de l'utilisateur. Le système récupère l'ancienne version à remplacer du serveur central et l'incorpore dans la copie de travail de l'utilisateur, et crée ensuite le rapport de modification et enfin met à jour l'arbre de version (figure 59).

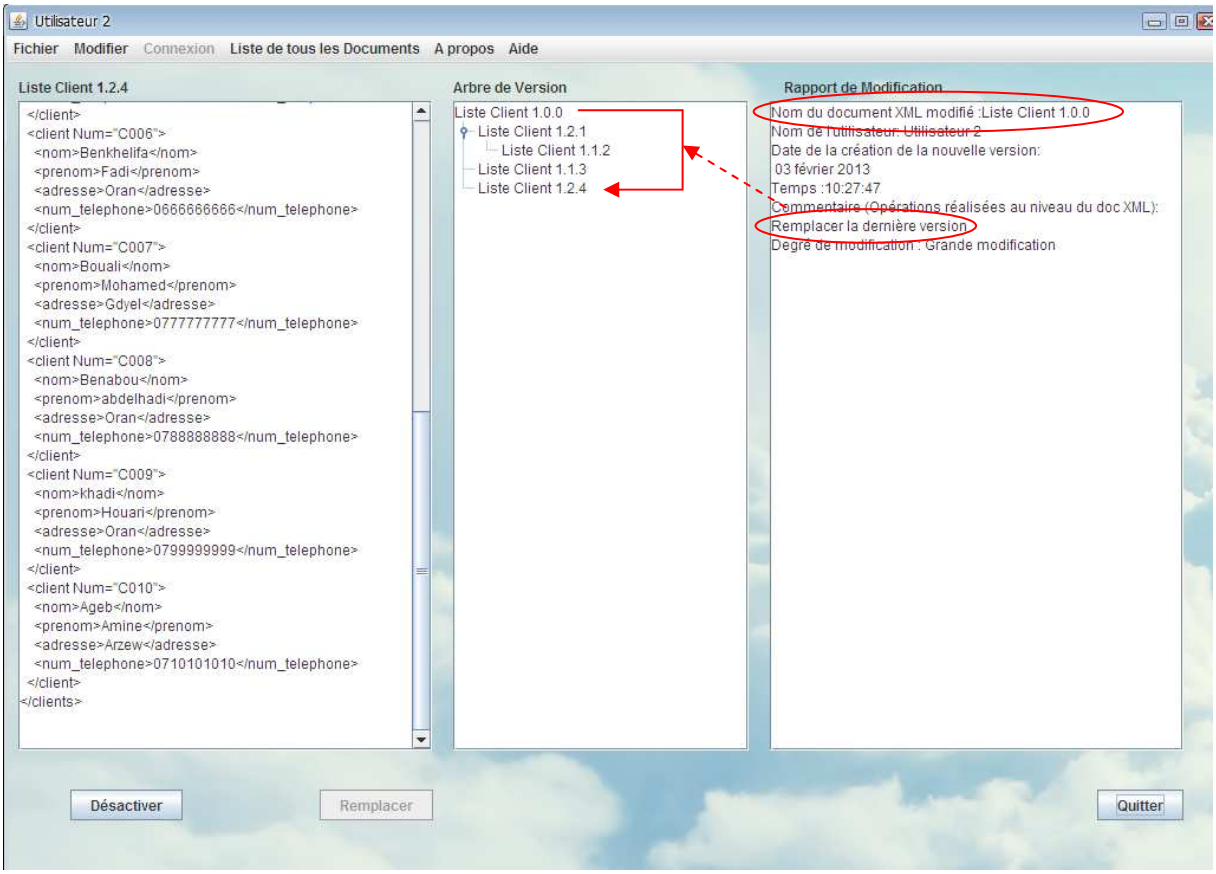


Figure 59. Remplacement d'un document

3.5 Désactiver un document

Quand une version d'un document est désactivée, toutes les opérations qui peuvent être réalisées sur cette version seront par la suite désactivées. La version sera accessible seulement en lecture par les deux utilisateurs. L'arbre de version après la désactivation du document ressemblera au schéma de la figure 60.

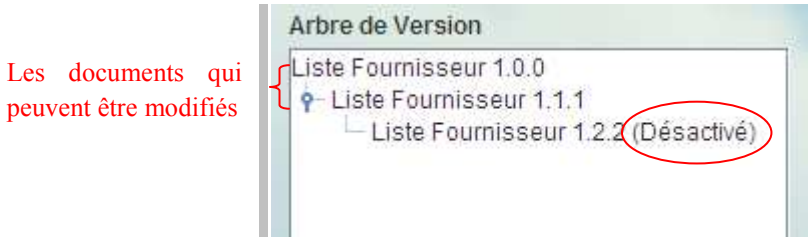


Figure 60. Désactivation du document Liste Fournisseur 1.2.2

Le deuxième utilisateur est notifié par le message suivant :

Figure 61. Message de Désactivation d'un document

4. L'Application Serveur Central

Le serveur central sert à afficher toutes les versions des documents avec les informations correspondantes stockées dans la Base XML centrale (figure 62), et à envoyer aux utilisateurs les versions antérieures des documents candidats à des modifications. Il sert aussi à fusionner les nouvelles versions concernant le même document modifié, en même temps, par les deux utilisateurs.

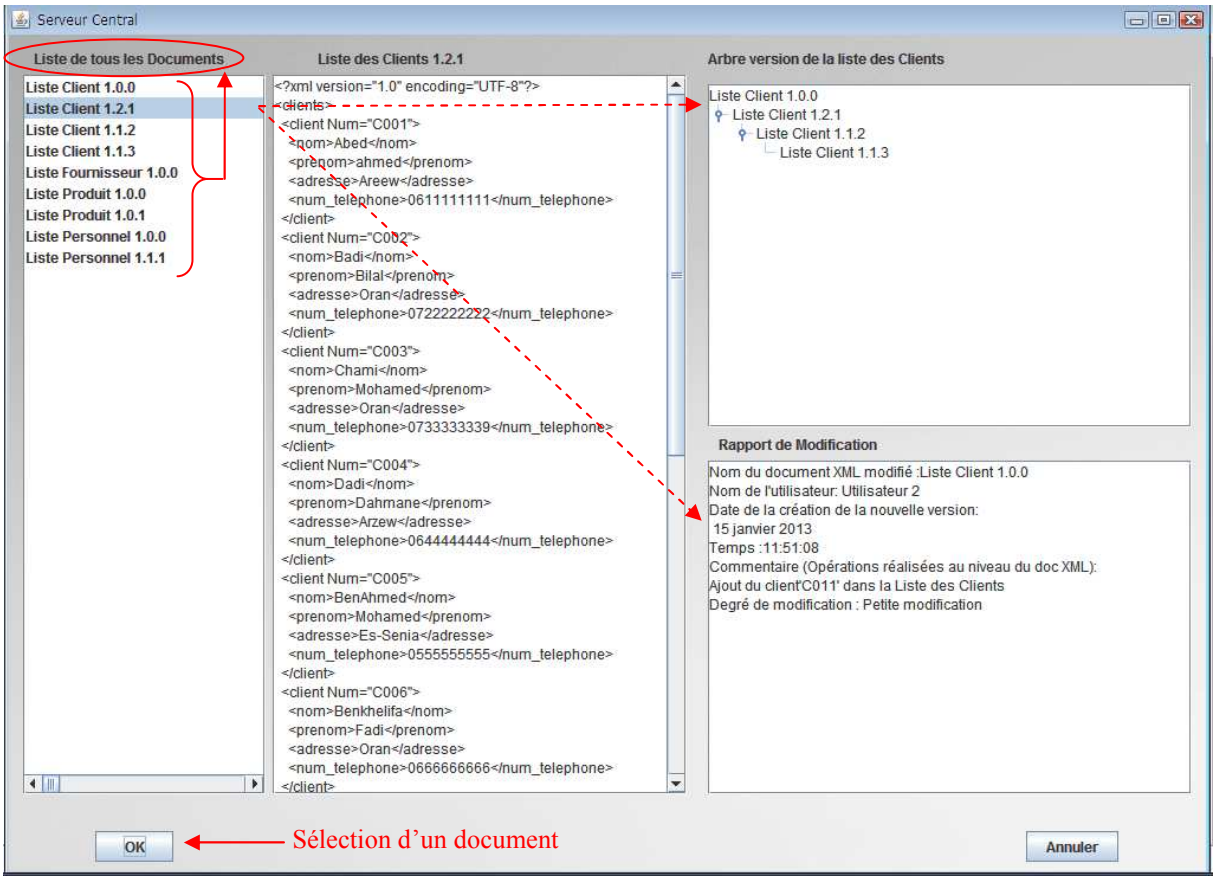


Figure 62. Application Serveur Central

Le schéma suivant illustre le contenu de la Base XML Centrale.

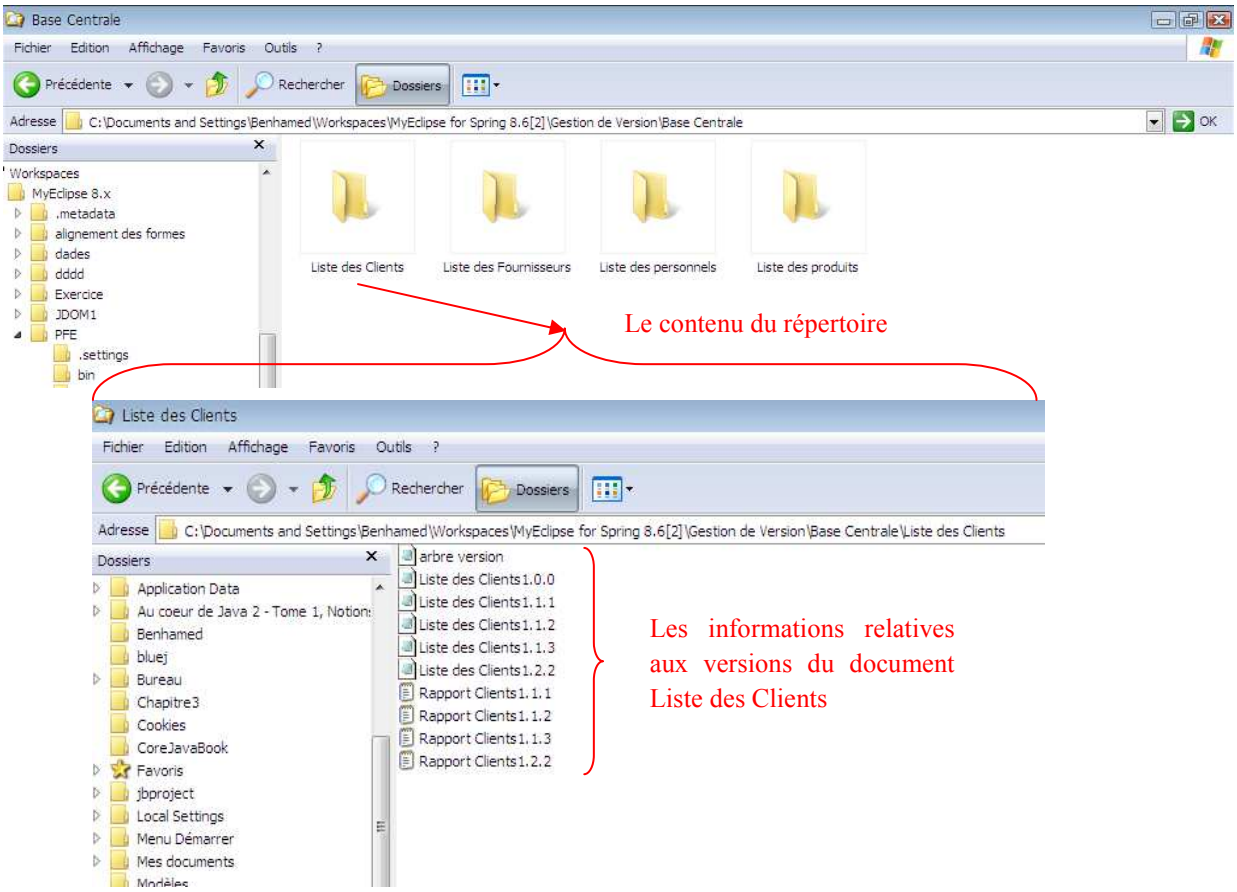


Figure 63. La Base XML Centrale

4.1 Verrouillage et déverrouillage de document

Un document est verrouillé que lorsque les opérations de modification sont des modifications globales (densité=vrai). Et négliger cette étape engendre un conflit quand les deux utilisateurs modifient le même document au même temps.

Quand un utilisateur désire modifier un document verrouillé, un message apparait lui notifiant le verrouillage du document (figure 64).

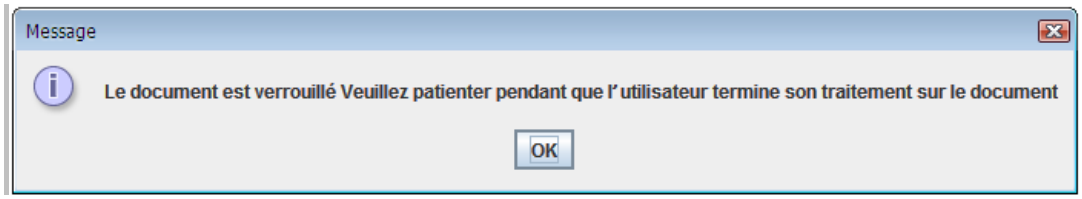


Figure 64.Message d'un document verrouillé

Une fois l'opération terminée, l'utilisateur envoie la nouvelle version au deuxième utilisateur, et déverrouille le document pour donner la main à cet utilisateur.

4.2 Fusion de document

Le serveur central reçoit les nouvelles versions modifiées des utilisateurs et fusionne ces résultats en un seul document par la procédure `MergeXML`.

Quand le serveur central est informé par les utilisateurs de la fusion, il se met en attente pour pouvoir traiter les documents, les rapports de modification, et l'arbre de version.

A la fin de la fusion, le serveur central diffuse ces résultats aux utilisateurs.

La méthode `MergeXML` consiste à réécrire les deux documents en un seul document XML, ensuite à supprimer les éléments qui sont en double. Le code de la méthode est le suivant :

```
private static void MergeXML (Document books, Document onebook) {
 try{
  SAXBuilder builder = new SAXBuilder();
  books = builder.build(new File(file1));
  onebook = builder.build(new File(file2));
  File file = new File(file1);
  boolean success = file.delete();
  File files = new File(file2);
  boolean successs = files.delete();
  Element root = onebook.getRootElement();
  List rows = root.getChildren();
     for (int i = 0; i < rows.size(); i++) {
      Element row = (Element) rows.get(i);
      books.getRootElement().addContent(row.detach());
    }
  new XMLOutputter(Format.getPrettyFormat()).output(books, System.out);
  XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
  resultat="Liste Produit "+1+"."+0+"."+h;
  file3="C:\\Documents and Settings\\Benhamed\\Workspaces\\MyEclipse for
Spring 8.6[2]\\Gestion de Version\\Base Centrale\\Liste des
produits\\Liste des produits"+1+"."+0+"."+h+".xml";
  filer3="C:\\Documents and Settings\\Benhamed\\Workspaces\\MyEclipse
for Spring 8.6[2]\\Gestion de Version\\Base Centrale\\Liste des
produits\\Rapport Produit"+1+"."+0+"."+h+".txt";
  sortie.output(books, new FileOutputStream(file3));
   }catch(Exception e){
      e.printStackTrace();
   }
}
```

Après la fusion, les utilisateurs procèdent à la mise à jour de leurs interfaces (figure 65).

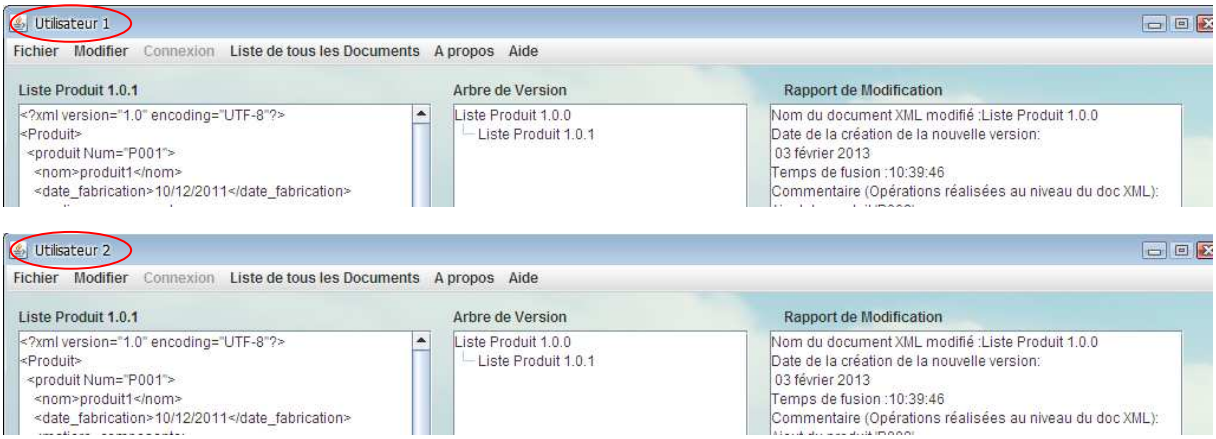


Figure 65. Les données reçues après fusion

4.3 Rapport d'un document fusionné

Le rapport de modification d'un document fusionné est le résultat de la fusion des rapports de modification reçus des utilisateurs, il contient les informations des opérations établies par les utilisateurs responsables de la modification ainsi que le temps de sa création. La fusion des rapports de modification est décrite par la méthode `mergerapport` ci-dessous :

```
public static void mergerapport(String filer1, String filer2,String
filer3) throws IOException {
FileReader in;
FileReader inn;
ConcatReader concatReader = new ConcatReader();
in = new FileReader(filer1);
concatReader.addReader(in);
inn = new FileReader(filer2);
concatReader.addReader(inn);
concatReader.lastReaderAdded();
FileWriter fw = new FileWriter(filer3);
int c;

while ((c = concatReader.read()) != -1) {
fw.write((char)c);
}
fw.close();
concatReader.close();
File file1 = new File(filer1);
boolean success1 = file1.delete();
File files1 = new File(filer2);
boolean successs1 = files1.delete();
Vector monVector = new Vector();
File f = new File(filer3);
BufferedReader B = new BufferedReader(new FileReader(f));
String ligne = B.readLine();
while (ligne != null){
monVector.addElement(ligne);
ligne = B.readLine();
}
monVector.removeElementAt(1);
monVector.removeElementAt(3);
monVector.removeElementAt(5);
monVector.removeElementAt(5);
monVector.removeElementAt(5);
```

```
monVector.removeElementAt(6);
monVector.removeElementAt(6);
monVector.removeElementAt(8);
Calendar dateDepart = Calendar.getInstance();
Locale locale = Locale.getDefault();
DateFormat formatHeure = DateFormat.getTimeInstance(DateFormat.MEDIUM,
locale);
monVector.insertElementAt("Temps de fusion :" +
formatHeure.format(dateDepart.getTime()),3);
monVector.removeElementAt(6);
PrintWriter P = new PrintWriter (new FileWriter(f));
    for (int i = 0; i < monVector.size(); i++){
        P.println(monVector.get(i));
    }

    P.close();
}
```

5. Les Echanges des Données entre les Applications

5.1 L'envoi des données

Le système adopte trois modes d'échanges de données entre les différentes applications (figure 67) :

1. Un utilisateur envoie après chaque modification d'un document, la nouvelle version du document, le rapport de modification, et l'arbre de version au deuxième utilisateur et au serveur central.
2. Lorsqu'un utilisateur désire consulter un document d'une version antérieure, il envoie au serveur central le nom du document et sa version qui à son tour lui communique le document avec toutes les informations correspondantes.
3. Un utilisateur est toujours informé des opérations réalisées par le deuxième utilisateur. Et ce dernier ne peut modifier ces documents, en commun, que si l'autre utilisateur lui envoie son accord. Un utilisateur envoie trois types d'information : accepter, arrêter, ou bloquer chaque fois que l'autre utilisateur désire modifier un document (figure 66).

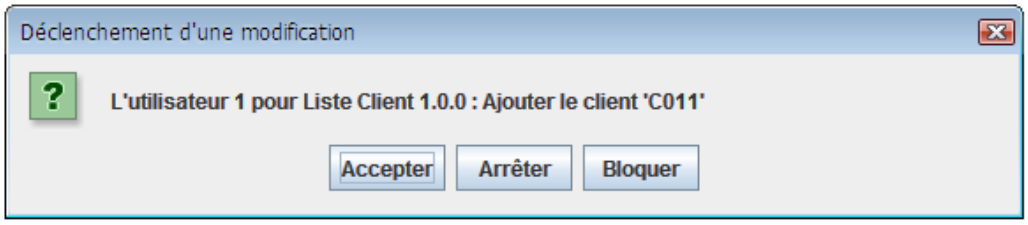


Figure 66. Information sur une modification d'un document

Remarque : L'utilisateur et le serveur central restent toujours en attente de la réception des nouvelles versions, de l'arbre de version et du rapport de modification correspondants.

Dans le cas de fusion les utilisateurs envoient les documents, les arbres de version, et les rapports de modification au serveur central et ce dernier leur transmet ces données fusionnées.

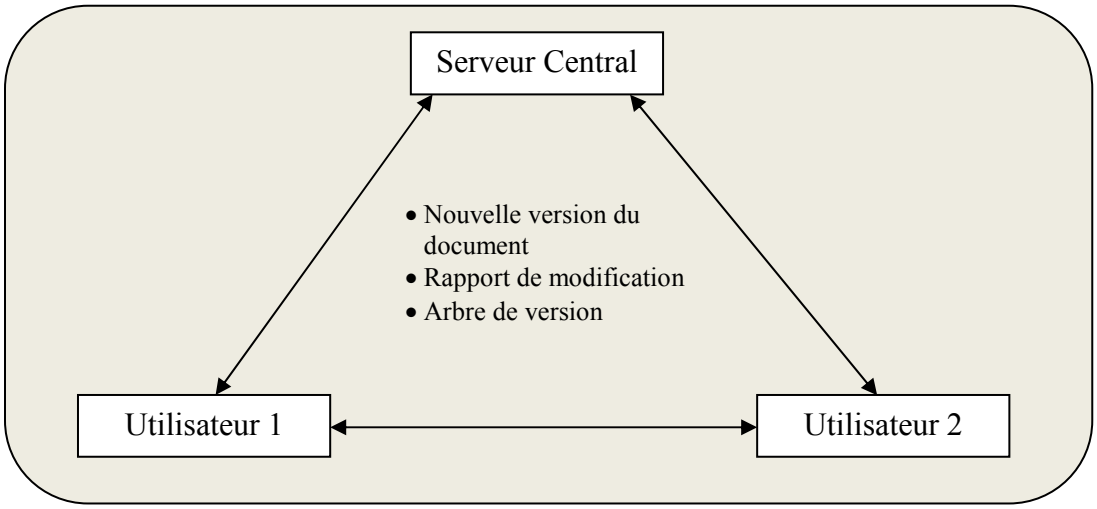


Figure 67. Les échanges des données

Le code de l'envoi des données :

```
byte[] bufferm = getspemg().getBytes();
envoi(hostname,hostportm,bufferm);

public static void envoi (String hostname, int hostport, byte[] buffer){
try{
InetAddress host_address = InetAddress.getByName(hostname);
DatagramSocket socket = new DatagramSocket();
DatagramPacket paquet = new DatagramPacket(buffer,
buffer.length,host_address,hostport);
socket.send(paquet);
socket.close();socket.disconnect();
}catch(Exception e){
  System.err.println("Erreur: "+e);
   }

}
```

Le code de l'envoi des documents :

```
String[] tab = {"Serveur"};
//Ouverture de la communication
Socket sock = new Socket(InetAddress.getLocalHost(),9000);
ObjectOutputStream out = new ObjectOutputStream(sock.getOutputStream());
//Envoi de l'en-tête
out.writeInt(tab.length);
for (int i=0; i<tab.length; i++)
out.writeUTF(tab[i]);
//Envoi du fichier
communn.transfert( new FileInputStream(new File(filePath1)),out,true);
//Fermeture de la communication
sock.close();
```

5.2 La réception des données

Après chaque modification, chaque utilisateur reçoit les informations concernant les opérations effectuées par l'autre utilisateur. Ces informations sont réceptionnées par le serveur central (figure 67).

La réception des données se traduit par le code suivant :

```
static public DatagramPacket receptionde (int hostportmm){
    try{
DatagramSocket socket = new DatagramSocket(hostportmm);
DatagramPacket paquets = new DatagramPacket(new byte[512],512);
socket.receive(paquets);
String siham=new String(paquets.getData()) ;
if (siham.startsWith("accepter")){
decision2="accepter";
JOptionPane.showMessageDialog(null, "La modification a été acceptée par
l'utilisateur 1");
}
if (siham.startsWith("arreter")){
int choix = JOptionPane.showOptionDialog(null,
new Object[] {"La modification a été arrêtée par l'utilisateur 1, Voulez
vous arrêter la modification?"}, "Arrêr d'une modification",
JOptionPane.NO_OPTION, JOptionPane.QUESTION_MESSAGE, null, new String[]
{"Accepter", "Refuser"}, null);
if (choix==JOptionPane.OK_OPTION) {
JOptionPane.showMessageDialog(null, "La modification est arrêtée");
//  /envoi de la décision
decision2="arreter";
message="arreter";
byte[] bufferm = getspemg().getBytes();
envoi(hostname,hostportm1,bufferm);
}
else{
 JOptionPane.showMessageDialog(null, "La modification est déclenchée");
//  /envoi  de la décision
 decision2="accepter";
 message="accepter";
 byte[] bufferm1 = getspemg().getBytes();
 envoi(hostname,hostportm1,bufferm1);
 }
}
```

```
if (siham.startsWith("bloquer")){
int choix = JOptionPane.showOptionDialog(null,
new Object[] {"La modification à été bloquée par l'utilisateur 1, Voulez
vous bloquer la modification?"}, "blocage d'une modification",
JOptionPane.NO_OPTION,JOptionPane.QUESTION_MESSAGE, null, new String[]
{"Accepter", "Refuser"}, null);
if (choix==JOptionPane.OK_OPTION)
{
JOptionPane.showMessageDialog(null, "La modification est déclenchée");
///envoi  de la décision
bloquer=false;activer=true;
decision2="accepterr";
message="accepterr";
byte[] bufferm = getspemg().getBytes();
envoi(hostname,hostportm1,bufferm);
}
else{
JOptionPane.showMessageDialog(null, "La modification est bloquée");
///envoi de la décision
      bloquer=false;
      if (c==true){jFrame1.setVisible(true);}
      if (f==true){jFrame11.setVisible(true);}
      if (pr==true){jFrame12.setVisible(true);}
      if (pe==true){jFrame13.setVisible(true);}
      decision2="bloquerr";
      message="bloquerr";
      byte[] bufferm = getspemg().getBytes();
      envoi(hostname,hostportm1,bufferm);
}
}
socket.close();socket.disconnect();
}catch(Exception e){
System.err.println("Erreur: "+e);
}
return null;
}
Socket sock = new ServerSocket(9000).accept();
ObjectInputStream in = new ObjectInputStream(sock.getInputStream());
// Ouverture d'une communication
// Réception de l'en-tête
int n = in.readInt();
String[] tab = new String[n];
for (int i=0; i<n; i++)
      tab[i]=in.readUTF();
```

```
// Réception du fichier
communn.transfert( in, new FileOutputStream(new File("C:\\Documents and
Settings\\Benhamed\\Workspaces\\MyEclipse for Spring 8.6[2]\\Gestion de
Version\\Utilisateur 2\\Liste des Clients\\Liste des
Clients"+1+"."+1+"."+m+".xml")), true);
// Fermeture de la communication
sock.close();
// Affichage de l'en-tête
for (String s : tab)
System.out.println(s);
}
}
.....
```

6. Conclusion

L'utilisation de l'API JDOM nous a permis de créer les documents XML, récupérer les données des documents en toute simplicité et de modifier ces derniers tout en économisant l'espace mémoire avec une grande performance en terme de temps d'exécution. Pour la création et la modification de l'arbre de version et l'affichage des documents par une représentation arborescente hiérarchique, nous avons utilisé le JTree. L'envoi de la nouvelle version a été réalisé d'une manière aisée par le traitement de la sérialisation. . Et les sockets nous ont permis d'établir la communication rapide entre les utilisateurs et entre les utilisateurs et le serveur central en temps réel.

Conclusion

Les web services se présentent aujourd'hui comme un support crédible permettant à des applications d'exposer leurs fonctionnalités aux travers des interfaces standardisées et de plus en plus éprouvées. Cette technologie permet de créer des systèmes d'informations interopérables (échangeant des données) et faiblement couplés (peu sensibles aux changements). Son principe se fonde sur la possibilité d'invoquer une application sur le web, en utilisant les protocoles d'Internet, et en faisant abstraction aux caractéristiques d'implémentation du service (la plateforme, le langage de programmation, et les API). Cependant, ils ne supportent pas la notion de gestion de versions des données échangées entre ces applications.

Le travail présente une conception d'un nouveau modèle de gestion de versions destiné aux documents échangés entre les web services et leurs clients, et entre les web services tout en respectant les caractéristiques des web services. La conception du système de gestion de versions proposé s'appuie sur l'utilisation des concepts des règles actives ECA. Les règles actives ECA sont utilisées dans la gestion du document à modifier (contrôle de verrouillage et notification), dans la modification du document, et dans la gestion de la nouvelle version du document. De même que ce modèle utilise les deux stratégies « *Verrouiller-Modifier-Déverrouiller* » et « *Copier-Modifier-Fusionner* » des systèmes de gestion de versions classiques tout en évitant les points faibles de ces modèles. Le nouveau modèle de gestion de versions est proposé dans un des systèmes centralisés qui appartiennent à un système décentralisé global.

Nous présentons aussi dans ce travail, l'implémentation du modèle proposé dans la partie conception qui s'est portée sur plusieurs outils informatiques notamment une API JDOM dédiée à la création, l'extraction de données, et la modification des documents XML, un constructeur d'arbre pour la création et la modification de l'arbre de version, une sérialisation pour maintenir les données à envoyer, et les Sockets pour pouvoir communiquer en temps réel.

L'implémentation du modèle proposé a nécessité, l'implémentation de deux types d'application : application *Serveur Centrale* où se trouve la base de tout l'historique du système, et l'application *Utilisateur* qui interprète les actions établies par les utilisateurs. Ces deux applications communiquent entre elles par une connexion TCP.

Notre motivation pour le domaine de gestion de versions des documents échangés entre les web services et leurs clients et entre les web services, se justifie par la non existence d'un produit fini qui s'occupe de la gestion de versions de ce type de cas.

Perspectives

Notre travail constitue une petite contribution pour la gestion de versions dans les web services. Il nous semble intéressant de mener une recherche visant à compléter ce travail :

- En augmentant le nombre des utilisateurs qui se partagent les même documents dans un système centralisé sur un serveur central et,
- En contrôlant la gestion de versions entre plusieurs serveurs centraux dans un système décentralisé contenant plusieurs systèmes centraux englobant chacun d'eux un serveur central et un ensemble d'utilisateurs.

Bibliographie

[AGO 09] AGOSTINI, Y., Gestion collaborative de l'administration des systèmes. Décembre 2009. JRES 2009. Centre de Ressources Informatiques. Université Paul Verlaine - Metz Ile du Saulcy

[AIX 10] "General Programming Concepts: Writing and Debugging Programs". Avril 2010. Copyright IBM AIX 5L, version 5.3

[ALT 08] "Working with a Version Control System". Cours d'instruction d'Altium TU011. 18 Mars 2008. Version 2.4

[BAI 04] BAILEY, J., DONG, G., RAMAMOHANARAO, K., On the Decidability of the Termination Problem of Active Database Systems. 2004. Theoretical Computer Science, pp. 389-433

[BAR 03] BAR, M., FOGEL, K., Open Source Development with CVS, 3RD EDITION. 2003. Karl Fogel et Paraglyph Press

[BAZ 10] Site officiel du système Bazaar. (http://bazaar.canonical.com/en/)

[BEE 11] BEEN, H., Extending WSDL with versioning information. 2011. Université de Twente.

[BEN 09] BEN HALIMA, R., Conception, implantation et expérimentation d'une architecture en bus pour l'auto-réparation des applications distribuées à base de services web. 14 Mai 2009. Thèse de Doctorat. Université Toulouse III - Paul Sabatier et Université de Sfax

[BEN 12] BENHAMED, S., HOCINE, S., BENHAMAMOUCH, D., Use of the ECA Rule for version control of Xml Documents. Mars 2012. IEEE International Conference on Information Technology and e-Services (ICITeS'2012) Sousse, Tunisie

[BEN 13a] BENHAMED, S., HOCINE, S., BENHAMAMOUCH, D., Conception of a Version Control System Approach Based by ECA Active Rules. Mars 2013. International Conference on Information and Intelligent Systems 2013 (ICIIS'2013), Sousse, Tunisie

[BEN 13b] BENHAMED, S., HOCINE, S., BENHAMAMOUCH, D, Proposition for a new Approach of Version Control System Based on ECA Active Rules. May 2013. International Conference on Computer Science and Information Engineering (ICCSIE), Tokyo Japon

[BOR 00] BORTZMEYER S., PERRET O., Versionnage : garder facilement trace des versions successives d'un document Exemple avec un outil de contrôle de versions (CVS). 2000. Document numérique. Vol. 4, no 3-4/2000, pp. 69- 81

[BOR 05] BORTZMEYER, S., Les nouveaux Systèmes de Gestion de Version. 2005. Rapport d'activité AFNIC.

[BOU 08] BOUATTOUR, S., FEKI, J., BENABDALLAH, H., BENMESSOUD, R., BOUSSAID, O., Proposition d'une taxonomie d'événement dans les entrepôts de données actifs. 2008. 8ème Journées scientifiques des jeunes chercheurs en génie électrique et informatique (GEI 08), Sousse, Tunisie

[BRE 06] BRESSON, A. F., Gestion de contenu Web. (http://www.axidea.org/form_info.htm e 17 janvier 2006)

[BRI 07] Brillant, A., XML cours et exercices. 2007. Edition Eyrolles

[BRO 02] BROWN, W. C., Version Control is Not Configuration Management. 11 Fevrier 2002. Spectrum Software. (www.spectrumscm.co)

[BUR 98] BURK, R., DOUBA, S., Unix Unleashed. Août 1998. Edition Sams, 2éme édition révisée

[CAD 04] CADIOU, A., Introduction à CVS : Un système de gestion de version, Journée de Calcul/Codiciel LMFA UMR CNRS 5509, FLCHP, 2004, Ecole Centrale de Lyon

[CAR 00] CARLSSON, M., LINDGREN, A., LJUNGSTRÖRD, A. M., MARKLUND, D., RANTA, J. O., REUTERSWÄRD, M., NetSync. Mai 2000. Rapport Contribué par OAIster. Université de Technologie Lulea, SE 971 97, Suède

[CAS 96] CASSALAS-GUTIERREZ, R., Objets historiques et annotations pour les environnements logiciels. 24 mai de 1996. Thèse de Doctorat. Université Joseph Fourier Grenoble I

[CED 93] CEDERQVIST, P., Version Management with CVS. 1993. Manuel de Signum Support AB

[CHA 94] CHAKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., KIM, S. K., Composite events for Active Databases: Semantics, Contexts and Detection. Septembre 1994. Dans la 20 éme Conférence Internationale sur de très grande base de données. Santiago, Chilie. pp. 606-617

[CHA 10] CHACON S., Pro Git (Expert's Voice in Software Development). 2010. Apress

[CHA 11] CHACON S., Pro Git (Expert's Voice in Software Development). 11 Mai 2011. Apress

[COL 96] COLLET, C., Bases de données Actives : des systèmes relationnels aux systèmes à objets. 1996. Rapport RR965-ILSR4, Laboratoire IMAG

[COL 08] COLLINS-SUSSMAN, B., FITZPATRICK, B. W., PILATO, C. M., Version Control with Subversion For Subversion 1.5. 2008. Edition TBA, version 3734

[COL 11] COLLINS-SUSSMAN, B., FITZPATRICK, B. W., Pilato. C. M., Version Control with Subversion For Subversion 1.6. 2011. Edition TBA, version 4259

[COU 96] COUPAYE, T., COLLET, C., Primitive and composite events in NAOS, Août 1996. Dans les actes des 12èmes journées de la base de données avancées, Cassis France

[COU 98] COUPAYE, T., COLLET, C., Modèles de comportement des SGBD actifs : caractérisation et comparaison. 1998. Technique et Science Informatiques (TSI), 17(3) :299-328

[CYN 05] CYNOBER, N., Manipuler des données XML avec Java et JDOM. 2005. (www.developpez.com)

[DAY 88] DAYAL, U., BLAUSTAN, B., CHAKRAVARTHY, U., MCCARTHY. D., The HIPAC Project : Combining active databases and timing constraints. Mars 1988. ACM-SIGMOD Record. Vol. 17 , no 1, pp. 51-70

[DAY 89] DAYAL, U., Active Database Management Systems. 1989. ACM Sigmod Record. Vol. 18(3), pp. 150–169

[DAY 95] DAYAL, U., HANSON, E., WIDOM, J., Active Database Systems. 1995. W.Kim editor, Modern Database Systems. pp. 434-456. ACM Press

[DEL 08] DELNNOY, C., Programmer en Java. 2008. Edition Eyrolles

[DEV 11] DEVELOPPEUR, 16 Mars 2011, par Idelways. (http://www.developpez.com)

[DON 09] O'DONOGHUE, S., RATCLIFFE, A., Configuration Management for SAS Software Projects. 2009. SAS Global Forum 2009, pp. 271-2009

[DOU 02] DOUDOUX, J. M., Développons en Java. 23 décembre 2002. Version 0.60 béta. (http://perso.wanadoo.fr/jm.doudoux/java/)

[DOU 03] DOUGLAS, K. B., Web Services and Service-Oriented Architectures: the savvy Manager's Guide, 2003. Morgan Kaufmann

[EAS 01] EASTLAKE, D., JONES, P. Algorithme 1 de hachage sécurisé (SHA-1). Septembre 2001. The Internet Society

[ERL 09] ERL, T., KARMARKAR, A., WALMSLEY, P., HAAS, H., YALCINALP, Ph.D., RICHARD, D. U., TOST, A., PASLEY, J., Web Service Contract Design and Versioning for SOA. 2009. Edition Prentice Hall

[FIS 09] FISHER, T. R., LE Guide de Survie Java L'essentiel du code et des commandes. 2009. Edition CampusPress

[FLE 04] FLEURY, R., Java / XML Les cahiers du programmeur. 02 Décembre 2004. Edition Eyrolles

[FOG 11] FOGEL, K., Produire du logiciel libre. 17 Mars 2011. Publié sous la licence libre Paternité-Partage à l'identique (3.0)

[GAT 93] GATZIU, S., DITTRICH, K.R., SAMOS: an Active Object-Oriented Database System. Janvier 1993. Groupe de Recherche Technologique, Institut d'Informatique Université de Zurich, IEEE Quartely Bulletin on Data Engineering

[GAT 94a] GATZIU, S., DITTRICH, K.R., Detecting Composite Events in an Active Database Systems Using Petri Nets. Février 1994. Proceedings of the 4th Intlernational Workshop on Research Issues in Data Engineering: Active Database Systems, Houston

[GAT 94b] GATZIU, S. GEPPERT, A. DITTRICH, K. R. SAMOS: an Active Object- Oriented Database System. 1994. Rapport technique 94.16, Université de Zurich

[GEH 91] GEHANI, N. H., JAGADISH, H.V., Ode as an Active Database: Constraints and Triggers. Septembre 1991. VLDB'91 Barcelone, Espagne. pp. 327-336

[GEN 03] GENUITEC. 2003. (http://www.genuitec.com)

[GEP 95] GEPPERT, A., GATZIU, S., DITTRICH, K.R., FRITSCHI, H., VADUVA, A., Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS. 1995. Rapport technique 95.29, Université de Zurich

[GNU 99] "Le manuel de GNU Privacy Guard". GNU Free Documentation License. 1999. Version 1.1. (http://www.gnupg.org/gph/fr/manual.html)

[GOE 09] GOETZ B., PEIERLS, T., BLOCH, J., BOWBEER, J., HOLMES D., LEA D., Programmation concurrente en Java. 2009. Edition Pearson

[GUE 11] GUESDON, M., ROUSSE, G., Gestion de configuration avec CVS et Subversion. Mars 2011. Support de cours (http://form-cvssvn.gforge.inria.fr/)

[HAR 02] HAROLD, E. R., Processing XML with Java. 2002. Edition Elliotte Rusty Harold

[HEB 11] HERBY, C., Apprenez à programmer en JAVA. 30 Mars 2011. Le Livre du Zéro. (http://www.siteduzero.com)

[HEI 04] HEINISCH, C., FEIL, V., SIMONS, M., Efficient configuration management of automotive software. Janvier 2004. European Congress ERTS Embedded Real Time Software

[HER 10] HERRB, M., Les systèmes de gestion de version. Envol 2010. LASS-CNRS

[HOA 11] HOARE, G., LEAKE, S., Monotone A distributed version control system. 2011. Manuel GNU GPL, version 2.0

[HOC 09] HOCINE, S., BELBACHIR, H., Events in Active Database : Formalization and Detection in their Interval of Validity. Juillet 2009. International Review on Computers and Software (IRECOS) ISSN 1828-6003

[HOC 10] HOCHSCHULE, D. M., Distributed Version Control Systems. Juillet 2010. Master degree course Computer Science and Media

[HOR 04] HORSTMANN, C. S., CORNELL, G., Au cœur de Java 2, volume 1 Notions fondamentales. 25 Novembre 2004. Edition CampusPress

[HOR 05] HORSTMANN, C. S., CORNELL, G., Au cœur de Java 2, volume 2 Fonctions avancées. 10 Mai 2005. Edition CampusPress

[HOR 08] HORSTMANN, C. S., CORNELL, G., Au cœur de Java volume 1 Notions fondamentales, 8ème édition. 2008. Edition Pearson Education

[HUN 92] HUNG, T., KUNZ, P. F., UNIX Code Management and Distribution. Septembre 1992. Université de Stanford. USA

[HUN 01] HUNTER, D. CAGLE, C., OZU, N., PINNOCK, J., SPENCER, P., Initiation à XML. 2001. Édition Eyrolles

[JAY 11] JAYAKRISHNA, K., AADITYA, A., MANASA, T., PHANIKUMAR, T. J., TALLAM S., Porting Jlint, Camlimages and RCS to Mac OS X. Avril 2011. International Institute of Information Technology, Bangalore. (http://www.iiitb.ac.in/). Rapport technique IIITB-OS-2011-8A

[JGE 05] JGENTI, T., Le point sur la gestion de configuration. Mai 2005. Mémoire d'examen probatoire. CHAM (Conservatoire National des Arts et Métiers), Paris. Informatique, Réseaux, Systèmes et Multimédia

[KAD 03] KADIMA, H., MONTFORD, V., Les web Services- Techniques, démarche et outils. Mars 2003. Edition Dunod

[KAM 06] KAMINSKI, P., MÜLLER, H., LITOIU, M., A Design for Adaptive Web Service Evolution. Mai 2006. SEAMS'06, Shanghai, China. ACM 1-59593-085-X/06/0005

[KUH 10] KUHN, D., Distributed Version Control Systems. Juillet 2010. Thèse de master, Université des Médias de Stuttgart (HdM)

[LAP 06] LAPEYRE, D. A., USDIN B. T., Introduction to XSLT Concepts. Janvier 2006. Publier par Mulberry Technologies

[LEI 08] LEITNER, P., MICHLMAYR, A., ROSENBERG, F., DUSTDAR, S., End-to-End Versioning Support for Web Services. 11 Mars 2008. Conférence de SCC08. Université Technique de Vienna Autriche

[LLI 97] LLIRBAT, F., Aide à la Spécification et à l'Optimisation des Applications Actives. 1997. Thèse de doctorat, Université de Paris 06

[LOE 09] LOELIGER, J., Version Control with Git. 2009. Edition O'Reilly

[LYN 07] LYNN, B., Git Magique Historique des versions. Août 2007. Revu par : BL publié sous la GNU General Public License, version 3

[MAE 03] MAESANO, L., BERNARD, C., LE GALLES, X., Services Web avec J2EE et .NET Conception et implémentations. 2003. Editions Eyrolles

[MAL 06] MALTE, T., TODD, A., PAUL, H., How to use monotone. 14 Juin 2006.Manuel d'utilisation

[MAS 05] MASON, M., Pragmatic Version Control using Subversion. 2005. The Pragmatic Programmers LLC

[MAT 09] MATJAZ B. JURIC, A., ANASASA, A., BOSTJAN, B. A., IVAN, R. A., WSDL and UDDI extensions for version support in web services. 2009. The Journal of Systems and Software 82 (1326–1343)

[MEY 00] MEYER, B., Conception et programmation orientée objet. 2000. Editions Eyrolles

[MOL 06] MOLLI, P., Une introduction à la gestion de configurations. 25 Septembre 2006. (http://www.loria.fr/~molli). Equipe ECOO

[MOT 95] MOTAKIS L., ZANIOLO, C., Composite Temporal Events in Active Database: A Formal Semantics. Université de Californie. Vol. 1013/1995, pp.19-37

[MSD 00] MSDN, Le langage WSDL 1.0 (Web Services Description Language), 2000, (http://msdn.microsoft.com/fr-fr/library/bb469923.aspx#_bindings)

[NGU 11] NGUYEN, T. T., GNU RCS for version 5.8. 8. Août 2011

[PAL 10] PALANIVEL, K., KUPPUSWAMI, S., A Service Vesrsionning Model for Personalized E- Learning System. 2010. International Journal of Engineering Science and Technology. Vol. 2(10), pp. 5583-5593

[PAR 07] PARACHURI, D., MALLI, S., Service Versioning in SOA Part 1: Issues in and approaches to Versioning. Mai 2007

[PAT 99] PATON N. W., DIAZ O., Active database systems, 1999. ACM Computing Survey

[PEC 06] PECQUET, L., Programmation Orientée Objet, Introduction illustrée en Java. 19 janvier 2006. Université de Poitiers, document version 1.1

[PER 12] "System Administrator's Guide". Perforce. Janvier 2012. (www.perforce.com)

[PET 10] PETAZZONI, T., Gestion de version Subversion. 1 octobre 2010

[POL 06] POLYZOTIS, N., SKIADOPOULOS, S., VASSILIADIS, P., SIMITSIS, A., FRANTZELL N.E., Supporting Streaming Updates in an Active Data Warehouse. 2006. Vol. 8, pp. 55-68

[RAM 06] RAMPACEK, S., Sémantique, interactions et langage de description des services web complexes. 10 novembre 2006. Thèse de doctorat de l'Université de Reims Champagne-Ardenne

[ROC 75] ROCHKIND, M. J., The source code control system. Décembre 1975. IEEE Transactions on software Engineering. Vol. SE-1, no 4

[SCH 95] SCHROD, J., The rcs Package. 2 Août 1995. De rcs.sty, version 2.10

[SCH 01] SCHNEIDER, D. K., SYNTETA, V., Code: Java – XML. 9 Avril 20001. Version 0.4. TECFA

[SCH 02] SCHMELZER, R., VANDERSYPEN T., XML and Web Services Unleashed. 2002. Edition Sams

[SEG 04] SEGHROUCHNI, A.E., HADDAD, S., MELITTI, T., SUNA, A ., Interopérabilité des services Web complexes. Application aux systèmes multi-agents. 8 Décembre 2004. JFSMA

[SUL 09] O'SULLIVAN, B., Mercurial: The Definitive Guide. 2009. Edition O'Reilly

[SWI 10] SWICEGOOD, T., Pragmatic Guide to Git The Pragmatic Bookshelf. Octobre 2010. Texas

[SYB 99] SYBASE. Août 1999. TDS 5.0 Functional Specification, document version 3.4 (http://www.sybase.com)

[TEL 84] TELEPHONE BELL LABORATIRIES, WESTERN ELECTRONIC COMPANY, Uniplus + System V Support Tools Guide. 1984. Université de California. Edition UniSoft Corporation

[TIC 85] TICHY, W. F., RCS - A System for Version Control. 1985. Software-Practice and Experience, Université de West Lafayette, Inde. Vol. 15, no 7 pp. 637-654

[TID 01] TIDWELL, D., Mastering XML Transformations XSLT. Août 2001. Edition O'Reilly

[TID 04] TIDWELL, D., XML programming in Java technology, Part 1 XML Evangelist. Janvier 2004 Copyright IBM

[VAS 03] VASUDEVAN A., CVS−RCS−HOWTO Document for Linux (Source Code Control System). 28 March 2003. Document GNU GPL, version 22.9

[VAU 07] Vaughan, D., .NET Web Services Version Control System. Mai 2007. Project Dissertation submitted to the University of Wales Swansea

[VER 06] VESPERMAN, J., Essential CVS, 2nd Edition Version Control and Source Code Management. November 2006. Edition O'Reilly

[VIA 06] VIALETTE, M., Web Services « communication inter Langage ». Mars 2006. Document version 2.0. Laboratoire Supinfo Des Technologies SUN

[WHI 04] WHITEHEAD, E. J., GUOZHENG, G. J., PAN, K., Automatic Generation of Version Control Systems. 2004. manuscrit soumis à ICSE, Université de Californie Santa Cruz

[WID 96] WIDOM, J., The Starburst active database rule system. Août 1996. Knowledge and Data Engineering, IEEE Transactions on. Université de Stanford Canada. Vol. 8, pp. 583-595

[WIL 04] WILDE, E., Semantically Extensible Schemas for Web Service Evolution. 27 Septembre 2004. European conference on web services (ECOWS 2004), Erfurt, ALLEMAGNE. (http://dret.net/netdret/publications#wil04j). Vol. 3250, pp. 30-45

[WOL 04] WOLIN, E., Why BitKeeper. 7 juin 2004. Rapport Gluex Collaboration

[W3C 04] "Web Service Architecture. W3C Working Group Note 11". Février 2004. W3C World Wide Web Consortium. (http://www.w3.org\TR\ws-arch)

[W3C 06a] "XML Schema Versioning Use Cases". 31 Janvier 2006. W3C XML Schema Working Group. (http://www.w3.org/XML/2005/xsd-versioning-use-cases/2006-01-31.html)

[W3C 06b] "Extensible Markup Language (XML) 1.1". 2éme édition, 29 Septembre. W3C Recommendation. (http://www.w3.org/TR/2006/REC-xml11-20060816)

[W3C 07] W3C, Extending and Versioning Languages: Terminology Draft TAG Finding 13 Novembre 2007. (http://www.w3.org/2001/tag/doc/versioning-20071113.html)

[ZAC 12] ZACCHIROLI, S., Environnements et Outils de Développement. 2012. License Creative Commons Attribution-ShareAlike 3.0 Unported License. Laboratoire PPS, Université Paris Diderot, Paris 7. (http://upsilon.cc/~zack/teaching/1112/ed6/)

[ZUI 03] ZULIANI, J. M., ABOUHARB, G., Les Web Services dans l'EAI-BtoB : Opportunité technique ou choix stratégie pour l'intégration étendue. Juin 2003. Edition Centre de Compétence EAI (Enterprise Integration Application)

Annexe 1 : Notre application active «*Gestion des Versions de documents*» manipulent les documents XML : «Liste des Clients», «Liste des Personnels», «Liste des Fournisseurs», et «Liste des Produits».

Liste des Clients 1.0.0

```
<?xml version="1.0" encoding="UTF-8"?>
<clients>
  <client Num="C001">
    <nom>Abed</nom>
    <prenom>ahmed</prenom>
    <adresse>Arzew</adresse>
    <num_telephone>0611111111</num_telephone>
  </client>
  <client Num="C002">
    <nom>Badi</nom>
    <prenom>Bilal</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0722222222</num_telephone>
  </client>
  <client Num="C003">
    <nom>Chami</nom>
    <prenom>Mohamed</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0733333339</num_telephone>
  </client>
  <client Num="C004">
    <nom>Dadi</nom>
    <prenom>Dahmane</prenom>
    <adresse>Arzew</adresse>
    <num_telephone>0644444444</num_telephone>
  </client>
  <client Num="C005">
    <nom>BenAhmed</nom>
    <prenom>Mohamed</prenom>
    <adresse>Es-Senia</adresse>
    <num_telephone>0555555555</num_telephone>
  </client>
  <client Num="C006">
    <nom>Benkhelifa</nom>
    <prenom>Fadi</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0666666666</num_telephone>
  </client>
  <client Num="C007">
    <nom>Bouali</nom>
    <prenom>Mohamed</prenom>
    <adresse>Gdyel</adresse>
    <num_telephone>0777777777</num_telephone>
  </client>
  <client Num="C008">
    <nom>Benabou</nom>
    <prenom>abdelhadi</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0788888888</num_telephone>
  </client>
  <client Num="C009">
    <nom>khadi</nom>
    <prenom>Houari</prenom>
    <adresse>Oran</adresse>
    <num_telephone>0799999999</num_telephone>
  </client>
  <client Num="C010">
    <nom>Ageb</nom>
    <prenom>Amine</prenom>
```

```
    <adresse>Arzew</adresse>
    <num_telephone>0710101010</num_telephone>
  </client>
</clients>
```

Liste des Personnels 1.0.0

```
<?xml version="1.0" encoding="UTF-8"?>
<personnels>
  <employe code="0001">
    <nom>Rahmani</nom>
    <prenom>Mohamed</prenom>
    <adresse>Arzew</adresse>
    <grade>Technicien en informatique</grade>
    <niv_etude>Technicien superieur en informatique</niv_etude>
    <agence>Agence Arzew</agence>
    <service>Service Caisse</service>
  </employe>
  <employe code="0002">
    <nom>Selimani</nom>
    <prenom>Samir</prenom>
    <adresse>Arzew</adresse>
    <grade>Ingenieur en informatique</grade>
    <niv_etude>Ingenieur en informatique</niv_etude>
    <agence>Agence Arzew</agence>
    <service>Service Informatique</service>
  </employe>
  <employe code="0003">
    <nom>Touati</nom>
    <prenom>Mohamed</prenom>
    <adresse>Oran</adresse>
    <grade>Chef du service Informatique</grade>
    <niv_etude>Ingenieur en Informatique</niv_etude>
    <agence>Agence Oran</agence>
    <service>Service Inforamtique</service>
  </employe>
  <employe code="0004">
    <nom>benabed</nom>
    <prenom>Hicham</prenom>
    <adresse>Oran</adresse>
    <grade>Chef du Service Commercial</grade>
    <niv_etude>Licence en Sciences Commercial</niv_etude>
    <agence>Agence Oran</agence>
    <service>Service Informatique</service>
  </employe>
  <employe code="0005">
    <nom>Dahou</nom>
    <prenom>Slimane</prenom>
    <adresse>Oran</adresse>
    <grade>Agent d'Administration</grade>
    <niv_etude>Licence en sciences de gestion</niv_etude>
    <agence>Agence Oran</agence>
    <service>Administration</service>
  </employe>
  <employe code="0006">
    <nom>Mahmoudi</nom>
    <prenom>Mahmoud</prenom>
    <adresse>Arzew</adresse>
    <grade>Cassier</grade>
    <niv_etude>DEA en Informatique de Gestion</niv_etude>
    <agence>Agence Arzew</agence>
    <service>Service Caisse</service>
  </employe>
</personnels>
```

Liste des Fournisseurs 1.0.0

```
<?xml version="1.0" encoding="UTF-8"?>
<Fournisseurs>
  <fournisseur Num="F001">
    <nom>Koussa</nom>
    <prenom>Mohamed</prenom>
    <adresse>Es-Senia</adresse>
    <produit_fourni>produit 1</produit_fourni>
    <num_telephone>0511121111</num_telephone>
  </fournisseur>
  <fournisseur Num="F002">
    <nom>Benabdalah</nom>
    <prenom>Amine</prenom>
    <adresse>Es-Senia</adresse>
    <produit_fourni>produit 2</produit_fourni>
    <num_telephone>041434444</num_telephone>
  </fournisseur>
  <fournisseur Num="F003">
    <nom>Sadik</nom>
    <prenom>Abdelkader</prenom>
    <adresse>Oran</adresse>
    <produit_fourni>produit 3</produit_fourni>
    <num_telephone>0732333333</num_telephone>
  </fournisseur>
  <fournisseur Num="F004">
    <nom>Daoud</nom>
    <prenom>Mohamed</prenom>
    <adresse>Oran</adresse>
    <produit_fourni>produit 4</produit_fourni>
    <num_telephone>0554555555</num_telephone>
  </fournisseur>
  <fournisseur Num="F005">
    <nom>Chergi</nom>
    <prenom>Said</prenom>
    <adresse>Es-Senia</adresse>
    <produit_fourni>produit 5</produit_fourni>
    <num_telephone>0554555555</num_telephone>
  </fournisseur>
  <fournisseur Num="F006">
    <nom>Fethi</nom>
    <prenom>Amine</prenom>
    <adresse>0ran</adresse>
    <produit_fourni>produit 2</produit_fourni>
    <num_telephone>0665666666</num_telephone>
  </fournisseur>
  <fournisseur Num="F007">
    <nom>Dadi</nom>
    <prenom>Ahmed</prenom>
    <adresse>Gdyel</adresse>
    <produit_fourni>produit 3</produit_fourni>
    <num_telephone>0777377777</num_telephone>
  </fournisseur>
  <fournisseur Num="F008">
    <nom>Hadad</nom>
    <prenom>Ibrahim</prenom>
    <adresse>Oran</adresse>
    <produit_fourni>produit 4</produit_fourni>
    <num_telephone>0786888888</num_telephone>
  </fournisseur>
  <fournisseur Num="F009">
    <nom>Belmokhtar</nom>
    <prenom>Mostafa</prenom>
    <adresse>Oran</adresse>
    <produit_fourni>produit 5</produit_fourni>
    <num_telephone>0799799999</num_telephone>
  </fournisseur>
  <fournisseur Num="F010">
```

```
    <nom>Belaid</nom>
    <prenom>Mohamed</prenom>
    <adresse>Es-Senia</adresse>
    <produit_fourni>produit 4</produit_fourni>
    <num_telephone>0710103010</num_telephone>
  </fournisseur>
</Fournisseurs>
```

Liste des Produits 1.0.0

```
<?xml version="1.0" encoding="UTF-8"?>
<Produit>
  <produit Num="P001">
    <nom>produit1</nom>
    <date_fabrication>10/12/2011</date_fabrication>
    <matiere_composante>
      <matiere1>matiere1</matiere1>
      <matiere2>matiere2</matiere2>
      <matiere3>matiere3</matiere3>
    </matiere_composante>
    <date_expiration>11/02/2014</date_expiration>
    <date_expiration>21/12/2014</date_expiration>
  </produit>
  <produit Num="P002">
    <nom>produit2</nom>
    <date_fabrication>23/02/2011</date_fabrication>
    <matiere_composante1>
      <matiere1>matiere1</matiere1>
      <matiere2>matiere2</matiere2>
    </matiere_composante1>
  </produit>
  <produit Num="P003">
    <nom>produit3</nom>
    <date_fabrication>10/08/2014</date_fabrication>
    <matiere_composante>
      <matiere1>matiere1</matiere1>
      <matiere2>matiere2</matiere2>
    </matiere_composante>
    <date_expiration>11/02/2014</date_expiration>
  </produit>
  <produit Num="P004">
    <nom>produit4</nom>
    <date_fabrication>10/10/2011</date_fabrication>
    <matiere_composante>
      <matiere1>matiere1</matiere1>
      <matiere2>matiere2</matiere2>
      <matiere3>matiere3</matiere3>
    </matiere_composante>
    <date_expiration>22/12/2013</date_expiration>
  </produit>
  <produit Num="P005">
    <nom>produit5</nom>
    <date_fabrication>08/11/2011</date_fabrication>
    <matiere_composante>
      <matiere1>matiere1</matiere1>
      <matiere2>matiere2</matiere2>
      <matiere3>matiere3</matiere3>
    </matiere_composante>
    <date_expiration>28/09/2013</date_expiration>
  </produit>
</Produit>
```

Annexe 2 : *Liste des règles ECA de modification des documents*

Les événements de modification

```
E1 : Ajouter-entité
E2 : Modifier-document
E3 : Changer-local
E4 : Supprimer-entité
E5 : Changer-Numtél
E6 : Présence-défaut
E7 : Non respect-délai
E8 : Mutation
```

Ci-dessous, les règles de modification des documents «Liste des Clients», «Liste des Fournisseurs», «Liste des Produits», et «Liste des Personnels».

Les règles de modification du document Liste des Clients

R1 : Ajouter-client
Cette règle consiste à ajouter un client identifié par le numéro « C011» dans la liste des clients.

```
Règle ECA R1₁ : Evénement : E2; E1
                 Condition : Si document=Liste des Clients & Num= «C011»
                 Action : autoriser ajout client «C011» avec le champ email

Règle ECA R1₂ : Evénement : E1
                Condition : Si document=Liste des Clients & Num= «C011»
                Action : Insérer le client «C011» sans le champ email
```

R2 : déménagement-client
Elle gère la situation d'un client qui déménage; elle enregistre l'adresse et le numéro de téléphone du client 'C004' qui a déménagé.

```
Règle ECA R2 : Evénement : E3
               Condition : Si Num = «C004»
               Action : 1. Modifier l'adresse du client
                        2. Modifier le numéro de téléphone dans le
                           document Liste Client
```

R3 : Modifier-client
La règle ajoute le champ email des clients dans le document Liste des Clients.

```
Règle ECA R3 : Evénement : E2
               Condition : Si document=Liste des Clients
               Action : Insérer le champ email de tous les clients
```

R4 : Supprimer-client
Elle supprime le client 'C001' du document.

```
Règle ECA R4 : Evénement : E4
               Condition : Si document=Liste des Clients & Num = «C001»
               Action : Supprimer le client «C001»
```

Les règles de modification du document Liste des Fournisseurs

R5 : Ajouter-fournisseur
Cette règle ajoute le fournisseur «F011» dans la liste des fournisseurs.

```
Règle ECA R5₁ : Evénement : E2; E1
                Condition :  Si  document=Liste  des  Fournisseurs  &
                             Num= «F011»
                Action : Insérer le fournisseur «F011» avec le champ prix
                         du produit

Règle ECA R5₂ : Evénement : E1
                Condition : Si document=Liste des Fournisseurs & Num= «F011»
                Action : Insérer le fournisseur «F011» sans le champ prix du
                         produit
```

R6 : Supprimer-fournisseur
La règle supprime un fournisseur du document.

```
Règle ECA R6 : Evénement : E4
               Condition :  Si  document=Liste  des  Fournisseurs  &  Num  =
                            «F002»
               Action : Supprimer le fournisseur «F002»
```

R7 : Modifier-fournisseur
La règle permet d'ajouter le champ prix du produit fourni par les fournisseurs dans le document «Liste des Fournisseurs».

```
Règle ECA R7 : Evénement : E2
           Condition : Si document=Liste des Fournisseurs
           Action : Insérer le champ prix du produit dans la liste des
                    fournisseurs
```

R8 : Changer le numéro de téléphone d'un fournisseur.
La règle impose que le numéro de téléphone du fournisseur 'F003'soit modifié.

```
Règle ECA R8 : Evénement : E5
               Condition : Si Num = «F003»
               Action : Modifier le numéro de téléphone
```

Les règles de modification du document Liste des Produits

R9 : Ajouter-produit
Elle ajoute le produit «P006» dans la liste produit.

```
Règle ECA R9₁: Evénement : E2; E1
               Condition : Si document=Liste des Produit & Num=  «P006»
               Action :  Insérer  le  produit «P006» avec le champ prix de
                         revient

Règle ECA R9₂: Evénement : E1
             Condition : Si document=Liste des Produit & Num=  «P006»
             Action :  Insérer  le  produit «P006» sans  le  champ  prix  de
                       revient
```


R10 : Remplacement-produit
La règle remplace le produit défectueux « P002 » par le produit «P007».

```
Règle ECA R10₁ : Evénement : E2; E6
                 Condition : Si Num = «P002»
                 Action : remplacer le produit «P002» par «P007» avec le
                          champ prix de revient

Règle ECA R10₂ : Evénement : E6
                 Condition : Num= «P002»
                 Action : remplacer le produit «P002» par «P007» sans le
                          champ prix de revient
```

R12 : Modifier-produits
La règle consiste à insérer le champ prix de revient des produits dans le document « Liste des Produits ».

```
Règle ECA R12 : Evénement : E2
                Condition : Si document=Liste des produit
                Action : Insérer le champ prix de revient dans la liste des
                         produits
```

R13 : Ajouter-produit
Elle ajoute le produit «P008 » dans la liste des produits.

```
Règle ECA R13₁ : Evénement : E2; E1
                 Condition : Si document=Liste des Produits & Num= «P008»
                 Action : Insérer le produit «P008» avec le champ prix du
                          revient

Règle ECA R13₂ : Evénement : E1
                 Condition : Si document=Liste des Produits & Num= «P008»
                 Action : Insérer le produit «P008» sans le champ prix du
                          revient
```

Les règles de modification du document Liste des Personnels

R14 : Supprimer-employé
Cette règle supprime un employé de la liste des personnels.

```
Règle ECA R14 : Evénement : E4
                Condition : Si document=Liste des Personnels & code =
                            «0005»
                Action : Supprimer l'employé «0005»
```

R15 : Ajouter-employé
La règle ajoute l'employé « 0007 ».

```
Règle ECA R15₁: Evénement : E2; E1
                Condition : Si document=Liste des Personnels & code =
                            «0007»
                Action : Insérer l'employé «0007» avec le champ email

Règle ECA R15₂ : Evénement : E1
                 Condition : Si document=Liste des Personnels & code =
                             «0007»
                 Action : Insérer l'employé «0007» sans le champ email
```

R16 : Modifier-personnels
Elle insère le champ email des employés dans le document « Liste des Personnels».

Règle ECA R16 : Evénement : E2
Condition : Si document=Liste des Personnels
Action : Insérer le champ email à tous les employés

R17 : problème-délais
La règle dégrade l'employé « 0003 » qui n'a pas respecté les délais et le remplacer par l'employé « 0008 ».

Règle ECA R17 : Evénement : E7
Condition : Si code = «0003»
Action : 1. Dégrader l'employé «0003»
2. remplacer l'employé «0003» par l'employé «0002»

R18 : Mutation-employé
Cette règle gère la mutation d'un employé; l'adresse et l'agence de l'employé «0001» muté sont modifiées et le remplacer par l'employé « 0008 ».

Règle ECA $R18_1$: Evénement : E2; E8
Condition : Si code = «0001»
Action : 1.Modifier l'adresse de l'employé & modifier l'agence de l'employé
2. Insérer l'employé «0008» avec le champ email

Règle ECA $R18_2$: Evénement : E8
Condition : Si code = «0001»
Action : 1.Modifier l'adresse de l'employé & modifier l'agence de l'employé
2. Insérer l'employé «0008» sans le champ email

Les opérations concernant les versions des documents

Le Remplacement de la dernière version

E1 : Sélection d'une version antérieure
E2 : création d'une nouvelle version

Règle ECA : Evénement : E1; E2
Condition : Si le document n'est pas désactivé
Action : 1. Remplacer par la version sélectionnée
2. Incrémenter son numéro de version
3. Mettre à jour l'arbre de version
4. Envoyer les modifications effectuées

La Désactivation d'un Document

E3 : Désactiver une version

Règle ECA : Evénement : E3
Condition : Si la version n'est pas désactivée
Action : 1. Désactiver la version
2. Mettre à jour l'arbre de version
3. Envoyer les modifications effectuées

Annexe 3 : Le code source exprimant le fonctionnement du module *Contrôle du Verrouillage* selon les règles ECA est le suivant :

```
Règle ECA 7 : Evénement : Sélection du document; modification de document
                          déclenchée
              Condition : Si densité=faux
              Action : Ne pas Activer le verrouillage
```

```
// Sélection du document & modification de document déclenchée
if (jRadioButton1.isSelected()&jRadioButton1.isEnabled()==true){
// Vérifier si la densité=faux
if (densitec==false){
int selection=JOptionPane.showConfirmDialog(jFrame1, "Est vous sûr d'ajouter le
client'C011' dans la Liste des Clients ?") ;
  if (selection == JOptionPane.OK_OPTION){
// Ne pas Activer le verrouillage et activer le travail sur le document
……
activer=true;
……
}
}
}
```

```
Règle ECA 6 : Evénement : sélection du document; modification de document
                          déclenchée
              Condition : Si densité=vrai
              Action : Activer le verrouillage
```

```
// Sélection du document & modification de document déclenchée
if (jRadioButton1.isSelected()&jRadioButton1.isEnabled()==true)
{
if (densitec==false){
……
}
// Vérifier si la densité=vrai
else{
// Verrouiller le document
JOptionPane.showMessageDialog(null, "Le document est verrouillé, Veuillez
patienter pendant que l'utilisateur termine son traitement sur le document");
jFrame1.setVisible(false);
       }

……
```

Résumé

Ce travail constitue une contribution à la gestion de versions des documents, particulièrement les documents XML dans un système décentralisé au niveau d'un de ces dépôts centralisés. Il montre comment les règles actives ECA ont été intégrées comme mécanisme pour le suivi des modifications des documents XML. La gestion de versions est essentielle dans les environnements où plusieurs utilisateurs manipulent une même base de ressources. Comme les Web Services ne prennent pas en compte l'historique des modifications des documents échangés entre les services et leurs clients, nous proposons une nouvelle approche de gestion de versions pour les documents XML dans les web services afin de permettre aux utilisateurs d'accéder à l'historique des modifications des documents et d'avoir pour chaque document la traçabilité de toutes les modifications effectuées au long de son cycle de vie.

Cette approche est basée sur le mécanisme des deux stratégies « *Verrouiller-Modifier-Déverrouiller* » et « *Copier-Modifier-Fusionner* » définies dans les systèmes de gestion de versions existants. Afin de rendre active certaines fonctionnalités de notre approche; nous avons utilisé les règles actives ECA dans la modification des documents, la notification des utilisateurs sur les modifications établies dans les documents, l'activation du verrouillage des documents, et dans la gestion de la nouvelle version. La nécessité d'intégrer les règles ECA est le fait qu'elles fournissent une sémantique claire et déclarative ainsi qu'elles offrent une réalisation immédiate et opérationnelle dans le système sans avoir besoin à l'intervention humaine.

Mots clés :

Document XML; Système de Gestion Version; Web service ; Règle ECA, Systèmes Actifs; Modèle *Verrouiller-Modifier-Déverrouiller*; Modèle *Copier-Modifier-Fusionner*; propagation; Notification; Modèle de Gestion de Version.