Réconciliation dans un Environnement Mobile

Gérald Oster et Pascal Molli
Equipe ECOO - LORIA, INRIA Lorraine,
BP 239, 54506 Vandœuvre-lès-Nancy, France
{oster,molli}@loria.fr

1. INTRODUCTION

La mobilité se caractérise par une alternance de phases de travail en mode connecté et en mode déconnecté. Pour supporter le travail en mode déconnecté les données sont répliquées sur le support mobile. Le travail est alors rythmé par des phases de divergence (en période de déconnexion) où les différentes copies évoluent en parallèles, et des phases de convergence durant lesquelles les copies devront être réconciliées. Le problème clé de la réconciliation est la fusion de modifications concurrentes conflictuelles.

2. PROBLÉMATIQUE

Différents types de synchroniseurs permettent aujourd'hui de réconcilier des copies divergentes.

Les synchroniseurs [1, 8], que ce soient de fichiers ou de données (agendas partagés par exemple) sont capables de réconcilier deux copies d'un même objet. Cependant, la réconciliation reste grossière et peu sûre.

Par exemple, supposons q'un utilisateur possède une station de travail et un ordinateur portable. Supposons que cet utilisateur modifie un fichier sur son ordinateur portable, et qu'il fait de même sur sa station de travail. Au moment de réconcilier les deux copies de ce même fichier, le synchroniseur, en utilisant les dates de dernière modification, détecte que le fichier a été modifié sur les deux supports, et se retrouve face à une situation de conflit. Pour résoudre ce conflit, il va demander à l'utilisateur de choisir la copie qu'il veut conserver.

Dans le domaine des gestionnaires de configuration [4, 2] où les utilisateurs peuvent avoir des copies divergentes d'un même objet, le problème est résolu en utilisant des outils de fusion. Ces outils de fusion (tel que Diff [6, 7] ou encore XMLDiffMerge [18]) peuvent être considérés comme des solutions ad-hocs : en effet, il est nécessaire de fournir au gestionnaire de configuration un outil de fusion pour chaque type d'objet partagé; chaque outil possède son propre algorithme de réconciliation : chaque outil de fusion est donc une solution spécifique pour un type d'objet particulier.

Une approche originale au problème de réconciliation est celle présentée dans le domaine des environnements collaboratifs temps-réels [3, 13, 14, 12, 10, 16, 9]. Ceux-ci reposent sur des algorithmes qui utilisent une fonction de transformation pour fusionner les opérations concurrentes et ainsi assurer la convergence des copies. Cette fonction de transformation exploite les propriétés sémantiques des opérations et permet une réconciliation plus fine des copies divergentes. Les avantages majeurs de cette approche sont :

- L'algorithme de synchronisation/réconciliation est le même quel que soit le type des données manipulées. Seule la définition de la fonction de transformation utilisée diffère (celle-ci dépend des opérations définies sur les objets typés).
- Ces algorithmes sont sûrs : il existe un certain nombre de propriétés [13, 16, 5] que la fonction de transformation se doit de vérifier pour respecter l'intention des utilisateurs, la causalité des opérations et assurer la convergence des copies.

Dans certains cas, pour assurer la convergence, la fonction de transformation est amenée à prendre des décisions que l'on peut qualifier d'"arbitraires".

Par exemple, supposons que deux utilisateurs u_1 et u_2 travaillent sur un même article. L'utilisateur u_1 change la couleur du titre en *rouge*. L'utilisateur u_2 change lui aussi la couleur du titre, mais il choisit la couleur *bleu*. Afin d'assurer la convergence un choix va être fait, c'est à dire que l'environnement va choisir par exemple que la couleur du titre devient *rouge* ; l'opération de l'utilisateur u_2 va être transformée en une opération sans effet.

Dans les environnements synchrones (comme les environnements temps-réels par exemple) pour lesquelles l'approche transformée opérationnelle a été développée, ce genre de choix arbitraires du système est peu gênant, et ce, pour deux raisons :

- les cas qui induisent ces choix, ont très peu de chance de se produire ; en effet, pour que de tels cas soient engendrés, il faut que les utilisateurs travaillent au même moment sur la même partie de l'objet. Or, dans un environnement synchrone, la période où les copies peuvent diverger, et donc que des modifications concurrentes peuvent réellement avoir lieu, est très courte (de l'ordre de la seconde : elle correspond au délai de propagation des modifications sur les autres sites).
- les utilisateurs qui interviennent dans cette situation de concurrence, ont un retour instantané ("feedback") de ce qui est exécuté par le système. Ainsi, dans notre exemple, si l'utilisateur u_1 voit que le titre devient *rouge* alors qu'il vient d'effectuer un changement de couleur en *bleu*. Il peut, soit décider

qu'effectivement la couleur rouge lui convient et ne rien faire, soit préférer sa modification et re-itérer son changement de couleur.

Cependant, dans les environnements supportant la mobilité la période de divergence est souvent beaucoup plus longue puisque les périodes de déconnexion peuvent durer plusieurs heures. Le coté arbitraire de la fonction de transformation va alors s'amplifier, les copies vont toujours converger, mais cet état ne sera plus satisfaisant. Un autre obstacle, et que les utilisateurs ne sont plus tous connectés au même moment, et ne peuvent donc plus compter sur l'effet bénéfique de retour instantané ("feedback") permettant de percevoir les modifications apportées par les autres utilisateurs. La perception de ce qui est effectivement exécuté par le système n'est donc plus aussi bonne. Ainsi, l'utilisateur peut, par exemple, ne plus travailler sur le même objet au moment de la résolution de la situation conflictuelle par le système ; il ne verra donc pas si ses modifications ont été refusées ou non.

Un certain nombre de problèmes subsistent, nous allons donc dans la suite de ce papier nous efforcer à répondre aux questions suivantes : Qu'est-ce qu'un conflit ? Quand et comment générer ces conflits ? Quand et comment résoudre ces conflits ? Comment présenter ces conflits à l'utilisateur ?

3. ETAT DE L'ART

Une première approche qui tient compte du problème des conflits dans les environnements à base de transformées opérationnelles est présentée dans [11]. Elle consiste à mettre en place un système supportant des objets multi-versionnés.

Ainsi, lorsque le système est amené à intégrer deux opérations concurrentes op_1 (l'opération locale) et op_2 (l'opération reçue) qui sont en conflit sur le même objet O_0 ; au lieu de prendre une décision arbitraire, une nouvelle version O_1 de l'objet va être créée sur laquelle l'opération op_2 sera exécutée. Les prochaines opérations devront donc être exécutées sur les deux versions O_0 et O_1.

Les auteurs, conscients du problème que pose l'explosion combinatoire du nombre de versions lorsque le nombre d'opérations en conflit augmente, proposent également un algorithme (MOVIC) qui minimise le nombre de version à créer, et ce, en calculant des groupes d'opérations compatibles.

Prenons comme exemple un objet partagé O_0, et trois opérations concurrentes op_1, op_2 et op_3 telles que :

- op_1 est en conflit avec op_2
- op_1 est en conflit avec op_3
- op_2 n'est pas pas en conflit avec op_3 (op_2 est dite compatible avec op_3)

L'algorithme "naïf" de création de version génère trois versions de l'objet O :

- $O'_0 = O_0.op_1$: la version sur laquelle seule op_1 a été exécutée.
- $O_1 = O_0.op_3$: la version sur laquelle seule op_3 a été exécutée.
- $O_2 = O_0.\{op_2, op_3\}$: la version sur laquelle op_2 et op_3 ont été exécutées.

Tandis que l'algorithme MOVIC va générer seulement deux versions :

- $O'_0 = O_0.op_1$
- $O_1 = O_0.\{op_2, op_3\}$

[17] propose un autre algorithme utilisant l'approche à base de version. Cette approche diffère de la précédente de par le fait que certains conflits n'ont pas besoin de créer de versions. Une distinction est donc faite entre les opérations qui nécessitent de créer des versions lorsqu'elles sont en conflit, et les opérations qui ne nécessitent pas la création de versions. Cet article propose également une autre manière de structurer les ensembles d'opérations compatibles qui offre un gain de performance au niveau de la phase d'intégration.

Le reproche principal que l'on peut faire à ces approches est qu'ils n'abordent pas le problème de la résolution des conflits, c'est-à-dire plus précisément, de quelles façons peut-on converger vers un état où il n'existe plus qu'une et une seule version de chaque objet partagé.

L'explosion combinatoire du nombre de versions constitue également un problème non négligeable : celle-ci augmente les temps de traitement, l'espace mémoire requis, et pose aussi des problèmes de représentation de ces différentes versions au niveau de l'interface utilisateur.

La gestion des conflits utilisée dans le gestionnaire de versions CVS [2] se rapproche de notre proposition, dans le sens où les conflits sont détectés, représentés, et certains sont résolus.

Lorsque CVS doit intégrer des modifications concurrentes sur des objets de type $Texte$ par exemple, il fait appel à un outil fusion ad hoc (Diff [6]). Cet outil utilise en fait trois versions de l'objet :

- Les deux versions V_{i+1} et V'_{i+1} de l'objet partagé contenant les modifications concurrentes.
- La version V_i qui l'ancêtre commun des deux versions V_{i+1} et V'_{i+1}.

Il calcule ensuite les ensembles d'opérations E (respectivement E') qui correspondent aux modifications apportées à V_i pour obtenir V_{i+1} (respectivement V'_{i+1}).

Il applique enfin les ensembles E et E' sur une copie de V_i. Les blocs de texte résultants de l'exécution d'opérations conflictuelles sont délimités afin de marquer les zones de conflits.

Le principe de réconciliation utilisé par CVS est donc assez proche de celui utilisé par les algorithmes à bases de transformées opérationnelles, dans le sens où celui-ci repose sur la fusion des deux ensembles E et E'

d'opérations qui ont permis de générer les états V_{i+1} et V'_{i+1} à partir du même état V_i.

Cependant, dans CVS, l'état qui contient les fichiers comportant des conflits n'est pas un état de convergence. En effet, tant qu'il y a des conflits sur un état, il n'est plus possible de publier cet état ; et donc, seul l'utilisateur qui possède cet état est conscient de ces conflits et peut agir dessus conflits.

L'autre aspect négatif de l'approche utilisée dans CVS provient du fait qu'il utilise des outils ad hocs ce qui affecte la portée du mécanisme de gestion des conflits. Ainsi, par exemple, les conflits du genre "Modifications Concurrentes d'un même fichier" sont détectés, représentés et résolus, alors que ceux du type "Modifications d'un fichier/Destruction de ce fichier", ne sont que détectés.

4. PROPOSITION

4.1 Principe

Notre approche consiste à représenter les conflits comme des objets à part entière du système. Ainsi, puisque les conflits sont des objets partagés d'un nouveau type, ils seront traités par le système de la même façon que les autres objets ; et donc, sous les mêmes conditions, le système pourra assurer la convergence des données vers un même état commun ; l'unique différence étant que cet état sera susceptible de contenir des conflits.

Le principe de fonctionnement de notre approche est le suivant : Lorsque le système est amené à intégrer des opérations concurrentes qui entrent en conflit, il ne va plus seulement prendre une décision arbitraire comme explicité précédemment (cf. section 2), mais il va en plus marquer cette décision en créant un objet de type $Conflit$ contenant les opérations qui sont en conflit.

L'avantage d'une telle approche est que la gestion des objets conflits reste indépendante des opérations qui sont en conflit, elle demeure donc indépendante du type des objets manipulés.

4.2 Un nouveau type d'objet : $Conflit$

Pour ajouter un nouveau type d'objet dans un environnement à base de transformées opérationnelles, il faut définir quelles sont les opérations qui peuvent être exécutées sur cet objet, et définir également la fonction de transformation pour les différents couples d'opérations concurrentes possibles.

Pour gérer les conflits, nous ajoutons à notre système les opérations relatives à ce nouveau type d'objet :

CreateConflict : CC(*cid*, op_1, op_2) : crée un nouvel objet de type $Conflit$ ayant comme identifiant unique *cid* ; les opérations op_1 et op_2 étant à l'origine du conflit.

AddInConflict : AC(*cid*, *op*) : ajoute l'opération *op* au conflit d'identifiant *cid*.

RemoveInConflict : RC(*cid*, *op*) : supprime l'opération *op* du conflit d'identifiant *cid*.

InvalidateConflict : IC(*cid*) : détruit entièrement le conflit d'identifiant *cid*. Cette opération peut être utilisée pour confirmer a posteriori le choix que le système avait effectué de manière arbitraire.

Il est possible d'ajouter une quatrième opération, de plus haut niveau qui utilise ces opérations primitives :

ChooseOpInConflict : COIC(*cid*, op_1) : permet de choisir une nouvelle opération qui était inhibée par le conflit (son utilisation est décrite plus en détail dans la section 4.6.2).

4.3 Génération des conflits

Un nouvel objet conflit va être crée à chaque fois que le système devra effectuer un choix arbitraire. Il faut donc modifier la définition de la fonction de transformation de telle sorte qu'elle instancie un nouvel objet de type $Conflit$, à chaque fois qu'elle se trouve face à une situation où elle doit transformer deux opérations qui ont des effets conflictuels (c'est-à-dire à chaque fois qu'elle devait prendre une décision arbitraire, dans sa version sans gestion de conflit).

Par exemple, pour les opérations définies sur un système de fichiers. Si la fonction de transformation qui évite les conflits est définie de la façon suivante :

```
Transp( EditFile(path1), RemoveDir(path2) ) {
  return EditFile(path1)
}

Transp( RemoveDir(path1), EditFile(path2) ) {
  if ( path2 isSubPathOf path1 ) {
     return NoOp() /* operation sans effet */
  } else {
     return RemoveDir(path1)
  }
}
```

Ce qui signifie que lorsque le système doit intégrer deux opérations concurrentes EditFile (modification d'un fichier) et RemoveDir (suppression d'un répertoire), il choisit toujours d'éviter le conflit en ne tenant pas compte du RemoveDir (choix arbitraire non destructif).

Cette fonction de transformation (pour supporter la génération des conflits) va alors s'écrire de la façon suivante :

```
Transp( EditFile(path1), RemoveDir(path2) ) {
  if ( path1 isSubPathOf path2 ) {
       return CC(cid, EditFile(path1), RemoveDir(path2))
  } else {
       return EditFile(path1)
  }
}

Transp( RemoveDir(path1), EditFile(path2) ) {
  if ( path2 isSubPathOf path1 ) {
     return CC(cid, EditFile(path2), RemoveDir(path1))
  } else {
     return RemoveDir(path1)
  }
}
```

4.4 Génération de conflit : Exemple

Supposons qu'un utilisateur possède une station de travail ainsi qu'un ordinateur portable. Entre ceux-ci, il partage un dossier Dir qu'il synchronise régulièrement.

La structure du dossier est illustrée par la figure 1.

```
Dir
 |
 |-- SubDir
 |    |
 |    |-- File2
 |
 |-- File1
```

Figure 1: Etat initial

Supposons maintenant que l'utilisateur modifie sur sa station de travail le fichier /Dir/SubDir/File2, et que sur son ordinateur portable il supprime le dossier /Dir/SubDir.

Lorsqu'il va re-synchroniser son dossier, le système va donc se trouver face à une situation de conflit entre les opérations op_1 = EditFile('/Dir/SubDir/File2') et op_2 = RemoveDir('/Dir/SubDir').

Il va donc être amener à faire un choix ; par exemple ici il choisit l'opération op_1 et génère un nouvel objet conflit C qui contient les opérations op_1 et op_2. L'état de convergence obtenu est décrit par la figure 2.

```
Dir
 |
 |-- SubDir
 |    |
 |    |-- File2*
 |
 |-- File1

Conflits :
   C = ( op_1 = EditFile('/Dir/SubDir/File2'),
         op_2 = RemoveDir('/Dir/SubDir')
       )

Journal :
   op_1  = EditFile('/Dir/SubDir/File2')
   op_2' = CC(op_1, RemoveDir('/Dir/SubDir'))
```

Figure 2: Etat après convergence

4.5 Principe de la résolution des conflits

La résolution d'un conflit ne peut être faite de manière automatique, elle demande l'intervention de ou des utilisateurs. En effet, résoudre un conflit, revient à valider ou à rejeter le choix arbitraire qu'a pris le système.

Dans le cas où l'utilisateur voudrait valider le choix du système, il lui suffit de générer une opération de destruction de l'objet conflit.

Dans le cas contraire, l'utilisateur, assisté par le système, va devoir annuler le choix du système, et donc les opérations et leurs effets "privilégiés" par ce choix ; détruire l'objet conflit associé à ce choix ; et enfin, il devra rejouer les opérations qui intervenaient dans le conflit, mais n'étaient pas prises en compte.

4.6 Résolution de conflit : Exemple

Reprenons notre exemple dont l'état de convergence obtenu est décrit par la figure 2.

Deux alternatives sont possibles : soit les utilisateurs sont satisfaits de l'état de leurs objets partagés, et le conflit doit être supprimé ; soit ils ne sont pas satisfaits par le choix du système et souhaitent le corriger.

4.6.1 *Alternative 1 : L'état obtenu est satisfaisant*

Dans le cas où l'utilisateur serait satisfait de l'état de convergence obtenu, et donc des choix "arbitraires" qui ont été effectués par le système, il peut supprimer l'objet conflit C. Pour cela il génère une nouvelle opération op_3 = InvalidateConflict(cid). L'exécution de cette opération va alors détruire l'objet conflit C, et ce sur tous les sites. L'état final obtenu est décrit par la figure 3.

```
Dir
 |
 |-- SubDir
 |    |
 |    |-- File2*
 |
 |-- File1

Conflits :
   aucun

Journal :
   op_1  = EditFile('/Dir/SubDir/File2')
   op_2' = CC(cid, op_1, RemoveDir('/Dir/SubDir'))
   op_3  = IC(cid)
```

Figure 3: Etat après suppression du conflit

4.6.2 *Alternative 2 : L'état obtenu n'est pas satisfaisant*

Dans le cas où l'utilisateur ne serait pas satisfait de l'état obtenu, il peut décider de vouloir exécuter l'autre alternative du conflit, c'est-à-dire dans ce cas l'opération op_2. Pour cela, il exécute l'opération ChooseOpInConflict(cid, $RemoveDir('/Dir/SubDir')$). Ce qui se répercute dans son journal par l'exécution successive des opérations :

- op_3 = Annuler(op_1) : Afin d'annuler l'effet de l'opération op_1 qui a été exécutée lors de l'intégration. Le problème de l'annulation d'une opération n'étant pas simple, ceci peut être réalisé en utilisant l'approche présentée dans [15].
- op_4 = RemoveInConflict(cid, RemoveDir('/Dir/SubDir')) : Afin de supprimer du conflit C l'opération qui a été choisie par l'utilisateur (nouveau choix pour le système). L'objet conflit C ne contenant plus qu'une seule opération, est alors détruit.
- op_5 = RemoveDir('/Dir/SubDir') : L'opération "choisie" est alors re-exécutée.

L'état obtenu après exécution de cette séquence est illustré par la figure 4.

5. CONCLUSION

Dans cet article nous avons présenté une approche à base de transformées opérationnelles pour répondre au problème des conflits qui peuvent survenir lors de la réconciliation de copies divergentes

Cette approche se caractérise par le fait qu'elle représente les conflits comme des objets du système ; l'état de convergence pouvant contenir ce type d'objet. Elle possède également l'avantage d'être générique : que ce soit la

```
Dir
 |
 |-- File1

Conflits :
   C = empty

Journal :
   op_1  = EditFile('/Dir/SubDir/File2')
   op_2' = CC(cid, op_1, RemoveDir('/Dir/SubDir'))
   op_3  = Annuler(op_1)
   op_4  = RC(cid, RemoveDir('/Dir/SubDir'))
   op_5  = RemoveDir('/Dir/SubDir')
```

Figure 4: Etat après correction du conflit

manière d'engendrer ou de résoudre les conflits, cette approche ne dépend pas du type des objets manipulés. Enfin, elle repose sur des algorithmes sûrs.

Cet article donne également une réponse à la façon de résoudre ces conflits. Faisant de l'activité de résolution des conflits, une activité collaborative à part entière.

Afin de valider cette approche un travail d'implémentation dans un prototype d'environnement collaboratif [1] est actuellement en cours.

Cependant, un problème subsiste : Comment peut-on présenter les conflits à l'utilisateur ? Comment lui présenter également les interactions sur ces objets (destruction, choix d'une opération, ...) ?

REFERENCES

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer ? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998. Full version available as Indiana University CSCI technical report #507, April 1998.

[2] B. Berliner. CVS II : Parallelizing software development. In *Proceedings of USENIX*, Washigton D. C., 1990.

[3] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[4] J. Estublier, editor. *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science. Springer-Verlag, Oct. 1995.

[5] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Development of transformation functions assisted by a theorem prover. In *ACM CSCW'2002 Workshop on Collaborative Editing Systems*, New Orleans, Louisiana, USA, November 2002.

[6] W. Miller and E. W. Myers. A file comparison program. *Software – Practive and Experience*, 15(11):1025–1024, 1985.

[7] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[8] N. Ramsey and E. Csirmaz. An algebraic approach to file synchronization. Technical report, Harvard University Department. of Computer Science, Cambridge MA (USA),, May 2001.

[9] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Computer Supported Cooperative Work*, pages 288–297, 1996.

[1] une version de démonstration de ce prototype est disponible en ligne à l'adresse http://woinville.loria.fr/sams/

[10] M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 36–45. IEEE Computer Society, 1998.

[11] C. Sun and D. Chen. A multi-version approach to conflict resolution in distributed groupware systems. In *Proceedings of The Twentieth IEEE International Conference on Distributed Computing Systems*, pages 316–325, Taipei, Taiwan, April 2000.

[12] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction*, 9(1):1–41, March 2002.

[13] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, pages 59–68, 1998.

[14] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems. In *ACM Transactions on Computer-Human Interactions*, volume 5, pages 63–108, 1998.

[15] N. Vidot. *Convergence des Copies dans les Environnements Collaboratifs Répartis*. PhD thesis, Université de Montpellier II, September 2002.

[16] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW-00)*, Philadelphia, Pennsylvania, USA, December 2000. ACM Press.

[17] X. Wang, J. Bu, and C. Chen. Research on conflict resolution and operation consistency in real-time collaborative graphic designing system. In *Proceedings of The Seventh International Conference on Computer Supported Collaborative Work in Design*, pages 145–150, Rio de Janeiro, Brazil, September 2002.

[18] XMLDiff. Xml diff and merge tool. *Online http://alphaworks.ibm.com/tech/xmldiffmerge/*, 2002.